Decrypting Local Type Inference

THÈSE N^{O} 6741 (2016)

PRÉSENTÉE LE 15 JANVIER 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Hubert PLOCINICZAK

acceptée sur proposition du jury:

Prof. A. Wegmann, président du jury
Prof. M. Odersky, directeur de thèse
Prof. O. Lhoták, rapporteur
Prof. J. Hage, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Only those who will risk going too far
can possibly find out how far one can go.
— T.S. Eliot

To my parents. I owe them everything.

Abstract

Statically typed languages verify programs at compile-time. As a result many programming mistakes are detected at an early stage of development. A programmer does not have to specify types for every single term manually, however. Many programming languages can reconstruct a terms type using type inference algorithms. While helpful, programmers often find it hard to comprehend the choice of typing decisions that led to the derived type for a term. A particularly serious consequence is that the reporting of type errors yields cryptic messages and misleading program locations.

In this thesis we propose a novel approach to explaining type checking decisions by exploring fragments of type derivation trees. Our approach applies to programming languages that use local type inference: typing decisions are made locally and the type information is only propagated between the adjacent AST nodes. We design an algorithm that backtracks through the nodes of type derivation trees in order to discover the typing decisions that introduce the types for the first time during the type inference process. Our algorithm has two properties

- it is type-driven, meaning that we only visit the nodes and their respective typing decisions if they participated in the inference of a type.
- it is autonomous, meaning that it does not require continues user-input in its operation.

These properties allow us to identify the complete and precise set of locations defining the source of a type; previous work mostly focused on heuristics or used approximations for locating the cause of an error.

Our algorithm is not tied to a particular implementation of a type checker: our type derivation trees can be reconstructed from a pre-existing type checker without modifying its internal logic or affecting its regular compilation times. It therefore readily applies to existing programs: we can not only provide improved feedback for them, we also expose limitations of local type inference algorithms and their implementations, without artificially limiting the language features.

We implement our type debugging algorithm on top of Scalas type checker. Our analysis applies to a range of erroneous scenarios. It provides better error locations than the standard type error reporter of the Scala compiler.

This type debugging analysis is just a starting point from which many interesting and useful applications around type debugging can be built:

- we implement an interactive type debugger that guides the users through the decisions of local type inference for erroneous and error-free programs alike.
- with precise and minimal source code locations we can also offer surgical-level code modifications that fix for example the limitations of local type inference.
- we open the door for programmatically defined, application-specific error feedback or corrections.

To the best of our knowledge this thesis is the first to address the problem of type errors for programming languages that use local type inference. Current trends suggest that this scheme is gaining in popularity with mainstream languages other than Scala.

Key words: type inference, type debugging, type errors, type checking

Les langages de programmation statiquement typés empêchent les programmeurs de faire certains types derreur pendant le cycle de développement dun logiciel. Un vérificateur de types, ou typeur, permet daccomplir cette tâche. Spécifier manuellement le type de chaque terme peut devenir extrêmement verbose. Un algorithme dinférence de types permet de pallier ce problème en reconstruisant automatiquement le type pour un certain terme. Il arrive malheureusement que linférence de types influe négativement sur la compréhension du typeur. Ceci se reflète sérieusement dans les messages derreurs liés au types: les messages deviennent cryptiques, ou suggèrent des sources derreurs imprécises.

Dans cette thèse, nous étudions le problème qui consiste à correctement, et exhaustivement, expliquer les décisions que prend un typeur pour arriver à un certain type pour un terme. Notre approche consiste à explorer les arbres de dérivation de types, et sapplique aux langages qui utilisent linférence locale: dans ce contexte, les décisions de typage sont prises de manière locale, et linformation concernant un type est propagée uniquement entre des noeuds voisins. Notre algorithme remonte, à partir dun certain noeud donné, larbre de dérivation, afin de découvrir le lieu original où un type a été introduit. Notre algorithme a deux propriétés:

- il est dirigé par les types: il visite seulement les noeuds qui participent à linférence dun type donné.
- il est autonome: il ne requiert aucune aide externe pour fonctionner.

Ces deux propriétés permettent didentifier, de manière exhaustive et précise, lensemble des sources de définition dun type; précédemment il était plus commun dutiliser des heuristiques ou encore dapproximer le lieu dune erreur.

Notre algorithme nest pas lié à un typeur spécifique: les arbres de dérivation de types peuvent être reconstruits à partir dun typeur pré-existent, sans devoir modifier ce dernier. Il est ainsi possible de directement lappliquer à des programmes pré-existents: nous pouvons proposer des messages derreurs plus précises, et aussi exposer certaines limitations des implémentations de linférenceur.
Nous proposons une implémentation de notre algorithme pour le typeur de Scala. Notre analyse sapplique à une game derreurs qui arrivent fréquemment en pratique, et nous permet de proposer de meilleures sources derreurs que le rapporteur standard de Scala.
Lanalyse de déboggage de types nest quun point de départ à partir de laquelle il est possible de développer beaucoup dapplications utiles et intéressantes:

- nous proposons un débogueur de types interactif, qui permet aux programmeurs dexplorer les décisions prises par le typeur, autant pour les programmes corrects querronnés.
- grâce aux sources précises derreur, il est même possible de proposer certaines modifications de code qui permettent de surmonter certaines lacunes de linférence locale de types.

- il est possible de développer, par dessus le débogueur, des extensions qui permettent de proposer des messages derreurs et corrections spécifiques à lapplication en question.

Au meilleur de nos connaissances, cette thèse est la première à adresser le problème derreurs de typage dans les langages a inférence locale. Les récents développements suggèrent que ce type dinférence gagne en popularité et se propage à des langages autres que Scala.

Mots clefs: inférence de type, le type de débogage, les erreurs de type, la vérification de type

Acknowledgements

During my time at EPFL I had a pleasure to meet and work with incredible people. I guess as I start writing this last chapter in the thesis I realize that this is going to be a pretty long night and I am going to omit someone. So I will keep it short.

First of all, I would like to thank my PhD thesis advisor Martin Odersky for letting me be part of the LAMP team and trusting in my capabilities. It's been a pleasure to be part of and contribute to the Scala project.

I would also like to thank Viktor Kunak, Ondřej Lhoták, and Jurriaan Hage for agreeing to be on my thesis committee and for reviewing my thesis, as well as Alain Wegmann for agreeing to be the thesis committee president.

I want to thank my colleagues at LAMP with whom I had a chance to spend time over the years: Sandro Stucki, Lukas Rytz, Iulian Dragos, Aleksandar Prokopec, Miguel Garcia, Chris Vogt, Adriaan Moors, Donna Malayer, Ingo Maier, Manohar Jonnalagedda, Vlad Ureche, Nada Amin, Denys Shabalin, Eugene Burmako, Sébastien Doeraene, Vojin Jovanovic, Heather Miller, Dmitry Petrashko, Philipp Haller, Tiark Rompf, Nicolai Stucki, Tobias Schlatter, Antonio Cunei. I cannot forget Danielle Chamberlain, our lab secretary, for her incredible optimism and administrative skills, and Fabien Salvi, for fixing my laptop so many times. Thank you so much.

Special thanks go to Philipp Haller and Heather Miller for giving me some shelter at your place when I needed it the most. I owe you big time for this.

And as I passed by the office of Stefan Lienhard he said that I should thank him too. So here it is, thanks mate for all the bouldering together.

A big thank goes to Vera Salvisberg for introducing me to blues dance. My life will never be the same after that.

I would like to thank Halina Kotyk, Maciej Baranowski and Olga Biskup for being my first math teachers. Those are the people with an enormous amount of patience that spend their extra hours doing incredible work for the community in my hometown.

I cannot forget about my high-school friends Jasiu Witajewski, Angelika Modelska, and Michal Dziedziniewicz whom I could always call and ask for help.

I started my studies at Imperial with Anton Stefanek and he kept my spirits up ever since. Thanks for all the chats about computer science and life in general. Thanks for giving me a CD with Ubuntu. That was a game changer.

I would like to thank here to my previous supervisor at Imperial, Susan Eisenbach, for luring me into the area of programming languages and encouraging me to stay in research after my studies.

Also I would like to thank Gergely Huszka, Jefferson Elbert and Simon Marmet for being probably the best flatmates one could ever wish for. And for being so patient with my cats.

I would like to thank my extended family whom I have only managed to see maybe once a year over the past couple of years. I especially would like to thank my uncle Wojciech Izydorek for showing me the beauty of cycling. You will always be a role model for me.

Finally, I would like to thank my parents, Ewa and Stefan, my brother, Łukasz, his wife Gosia and their little Emiś (well not so little anymore). It's hard to express the amount of gratitude I owe them, especially after all those months of my grumpiness. To be honest my journey to this thesis did not start 5 years ago at EPFL. It started around 13 years ago when I made a decision to leave my hometown, go, to a pretty unusual for those times, high-school with the Internal Baccalaureate, and 3 years later landed in London where I did my studies. For my parents, both retired teachers, this meant really high burden but they did everything they could to provide me with the best education that was available (I should mention that teachers in Poland don't get paid well). Thank you for supporting me with all my crazy ideas and letting me free so early. Thank you Łukasz and Gosia for tolerating my inner weirdness. I will try to fulfill my duties now as the uncle to Emiś (and no longer be just known as the uncle with cats).

As a side note, I would also like to thank Robin Williams (whom I never met personally). He is no longer with us but his sense of humor kept me sane and happy when I needed it the most.

Lausanne, 29 Mai 2015

Contents

Chapter 1

Introduction

The craft of writing software is inherently associated with the choice of the technology, and the choice of a programming language in particular. Irrespective of the preference of the programmers writing software also means dealing with errors which manifest themselves during the runtime execution of the program or during the compilation that statically verifies the properties of the program.

The dynamically typed languages, such as Javascript or Ruby, are prime examples of languages associated with rapid application development; programmers do not have to deal with additional type annotations and can debug their errors by inspecting the runtime values that failed to be covered by the logic of the program. On the other end we also have statically typed languages, such as Java or Scala, which can detect many of the runtime errors prior to the actual execution of the program. Static types provide additional guarantees for the written code, allow for a number of important compile-time optimizations or type-driven synthesis, among others. Types also provide a poor man's equivalent of a documentation of a program. The benefits of statically typed languages come at a cost: the source code can be packed with type annotations and the restrictive type systems can limit the expressiveness of the users in favor of the *safety* of the logic of the program.

Nowadays we experience a general tendency of the new programming languages to stay away from the two extremes in order to provide the best of the two worlds. The dynamically typed languages are being replaced with gradually typed languages, such as TypeScript[1] or Hack[2], that can verify the properties of the explicitly annotated fragments of the code. Alternatively, the dynamically typed languages are equipped with static analyzers, such as flow (http://flowtype.org/) or Phantm (Kneuss et al. [2010]), performing flow-analysis in order to discover the non-trivial errors and thus providing additional guarantees for the correctness of the code. The mainstream statically typed languages ease the adoption by allowing for the

[1]typescriptlang.org
[2]hacklang.org

type annotations to be elided either partially or fully, without potentially sacrificing the safety guarantees of the language with respect to programs with explicit type annotations. The type inference process can significantly improve the readability of the source code but the rules that govern the process tend to be underspecified or unclear to the programmers (Vytiniotis et al. [2011] gives one example where a more powerful type inference mechanism is not necessarily good for the programmers); the inferred type values are different from the intended ones, or they are not inferred at all, oftentimes leading to obscure type error messages that omit the involvement of the decisions process of type inference.

The type errors are, after the type annotations, the second most complained feature that programmers have to deal with in every day programming. Not only are the messages too abstract to many users, referring to the types of values, but are also rarely precise, reporting locations that are far from the real source of the error or from the types that participate in the conflict. With the elided type annotations the meaning of the error messages becomes even more cryptic since the reported types only indirectly refer to the source code provided by the programmer. Furthermore, with the type systems becoming more powerful and more expressive, their types rarely become less complicated, making the eliding of type annotations more unpredictable or even undecidable.

To illustrate how a complete confidence in the type inference process can lead to confusing type errors, we consider a short code snippet from Scala involving a local variable and a conditional expression:

```
var x = None
...
x = (if ( y > z ) Some(y)
     else         Some(z))

// error: type mismatch;
// found:    Some[Int]
// required: None.type
//  (if ( y > z ) Some(y)
//                      ^
```

In the example the user assigns some option value to a variable ‘`x`’. In Scala, ‘`None`’ and ‘`Some(y)`’ for some value ‘`y`’, are both values of an `Option` type, where the former represents an empty `Option` value and the latter the non-empty one with the value ‘`y`’. It may therefore be surprising to discover that the subsequent conditional statement results in a type mismatch involving the two subtypes of the `Option[T]` type. In situations like this programmers should be directed to a real source of the error (the inferred type of the type variable is `None.type` rather than the intended `Option[Int]`), or be explained the decision process of the type inference employed by Scala, or, in the worst case, be able to investigate the problem on their own, similarly to how one can use the runtime debuggers to analyze the execution of the program in dynamically typed languages. In the aftermath of errors like this, programmers loose confidence in the capabilities of type inference and start adding explicit type annotations in more locations than necessary, making their source code unreadable.

In general we distinguish between two main approaches to the type inference process: the global and the local type inference. Both come with their own sets of advantages and problems, that we will now briefly summarize.

Global type inference collects type constraints from the complete programs and only later attempts to solve them, potentially using some type unification technique. This kind of type inference, introduced for the first time in Hindley-Milner type inference algorithm (Damas and Milner [1982]), has been implemented in some variations in languages such as Haskell (Vytiniotis et al. [2011]) or OCaml (Russo and Vytiniotis [2009]). With the advent of advanced type system features, such as type-classes (Hall et al. [1994]), GADTs (Schrijvers et al. [2009]), type-families (Kiselyov et al. [2010]), it has become increasingly hard to provide a sound and decidable process that at the same time infers types that are intuitive for users. HM(X) (Odersky et al. [1999]) and OutsideIn(X)(Vytiniotis et al. [2011]) provide type inference algorithms that abstract over the domain of constraints and with a small number of requirements can still prove the inference of principal types. Unfortunately, global type inference has not been the technique of choice for other mainstream languages due to its non-trivial implementation, notorious mislocation of the error messages (McAdam [2000]), and, most importantly, the intractability and lack of principal types for nominal subtyping (Odersky et al. [1999] and Odersky [2002]). Due to the above reasons local type inference has become a technique of choice for eliding type annotations in many of the mainstream and recent programming languages, such as Scala, Rust[3], Ceylon[4], Typed Racket[5], and, in a limited form, in Java or C#.

Local type inference (Pierce and Turner [2000]) refrains from solving the constraints that are separated by a *long distance* by only propagating type information between the adjacent nodes in the abstract syntax trees of the source code. Due to the locality of the approach the integration of the core type system with additional type system features does not compromise the soundness of the type inference in general. On the other hand local type inference is not *complete*, meaning it will reject fully unannotated programs. Local type inference improves significantly the localization of type errors with respect to global type inference approach but does not eliminate the problem.

The Scala example presented above illustrates the fact that types that are inferred locally may lead to type errors for correctly defined programs, because they only take into account the type checking decisions up to the point of the given AST node. Various constructs, such as function applications with the elided type arguments, infer optimal approximations from the locally collected type constraints. The locations of the type constraints, the semantics of the approximation, and its effect on the inferred type are often hard to comprehend to

[3]rust-lang.org/
[4]ceylon-lang.org/
[5]docs.racket-lang.org/ts-reference

the programmers. To add to the confusion the implementations of local type inference may introduce their own limitations which are not explained by either the specification of the language nor the type error message, and have a tendency to linger in the language for years to come. The limitations lead to puzzling type errors, such as the one shown in Figure 1.1 for an innocuous snippet of Scala code. In this example, we have a list of integers that we want to increment using the `foldRight` method from the Scala standard library. A call to '`foldRight`' method, applies a binary operator, a function which appends an incremented elements, starting with the argument '`NiL`', to each element of a list from right to left.

```
  val xs = List(1, 2, 3)
  xs.foldRight(Nil)( (x, ys) => (x + 1) :: ys)

// error: type mismatch;
// found   : List[Int]
// required: scala.collection.immutable.Nil
// xs.foldRight(Nil)( (x, ys) => (x + 1) :: ys)
//                                      ^
```

Figure 1.1: Incrementing a list of integers gone wrong.

The type error exposes one of the limitations of type inference for local type parameters – type inference flows from left to right and only from parameter list to parameter list. Without a prior knowledge of the type inference algorithm users are unable to make a distant connection between the location of the error in '`::`' and the value '`Nil`'.

The improvements to the error reporting infrastructure have largely ignored the presence of programming languages using local type inference. With the advent of generic programming, available through the introduction of parametric polymorphism (or *generics*), and such features as subtyping polymorphism, implicit resolution (Oliveira et al. [2010]), path-dependent types (Odersky et al. [2003]) or mixin composition (Odersky et al. [2006]) languages like Scala are being increasingly criticized for the incomprehensible decisions of the type checking process. On the other hand languages like Java or C# chose to support only a limited form of local type inference, and refrained from adding more advanced type system features, in favor of a more predictable behavior (Cimadamore [2015]) and a simpler implementation. We believe that such compromise is both unnecessary and undesirable. Instead, the programming languages should provide enough of a type checking context information in order to build separate tools that explain at least some of the type errors and guide the user in understanding the sometimes necessary details of the type checking process.

The locality property and the fact that local type inference process is defined in an incremental manner not only helps with the implementation, but also offers important debugging opportunities that have not been taken into account in any of the related work: the analysis of the source of any type essentially resolves to traverse backwards through the already committed decisions of the type checking process. The insight does not apply to global type inference approach where the inferred types are a result of constraints collected and solved from the complete programs separately.

By traversing the type checking decisions backwards we aim to identify precisely the minimal source code locations, such as the explicit type annotations, constants, identifiers or type parameter instantiations, that introduce the type or, more importantly, a part of it for the first time. Due to a range of possible type checking decisions and their number for each of the AST nodes, that may introduce the type for the first time, the process itself must not employ any of the brute-force techniques that simply scan all type checking decisions. Furthermore, with the inferred types being a result of type constraints approximations, the process has to be well-defined for the possible semantics of the *least upper bound* and the *greatest lower bounds* decisions as well, meaning that in order to identify the minimal sources of types we have to go beyond just the identification of the existence of type constraints (El Boustani and Hage [2010]) and analyze their source as well.

The problem of incomprehensible type errors is not new and has induced a number of research projects which have focused on the two areas - finding the minimal source code locations representing the type errors and the improvements in the quality of their messages. Surprisingly, almost all of the work (the Java heuristics in El Boustani and Hage [2010] being an exception) ignored the mainstream, statically typed languages that are using some form of local type inference.

None of the mainstream implementations of type checkers for statically typed languages were built with the intention of representing the decision process. That is why any attempts to improving the error feedback resolve either to a definition of a new type system or type inference calculus that carries the lost information (Stuckey and Sulzmann [2005], Heeren et al. [2003a]) or infers better localized type error messages (Lerner et al. [2007]), or creating a separate, post-type checking phase that extracts the essential details of the process (El Boustani and Hage [2010]). In practice, the separate implementation supports only a subset of the existing language and its implementation, which significantly reduces its target audience and the chances of its widespread adoption in the future. With the ongoing development of the languages it is also likely that the separate debugging-oriented implementation will drag behind or diverge from the original implementation and result in different kinds of type errors.

With a new model of the type checking process, the improved error feedback is achieved by employing more elaborate constraint solving techniques (Pavlinovic et al. [2014], Stuckey and Sulzmann [2005]), that infer their unsatisfiable subsets of the collected constraints and lead to the minimal number of program locations that characterize the type error message (Haack and Wells [2004]).

An orthogonal approach to improving error reporting is aimed at developing techniques that automatically resolve the errors. These heuristics, defined by the language architects, approximate the type errors to one of possible templates in an effort to provide source code modifications that fix the error (Gvero and Kuncak [2015], Chitil [2001], Chen and Erwig [2014b]) or simply generate more informative type error messages (Hage and Heeren [2007], Weijers et al. [2013]). In addition, one can employ statistical models from the specially inferred constraints sets (Zhang et al. [2015]) and find the *most likely* sources of errors with a high-level of confidence.

The approximated models of type checking are perfectly suitable for educational purposes where a subset of the original language is likely to help with explaining the type checking process in general (Heeren et al. [2003c]). However, with local type inference approach we want to avoid the risk of reporting false positives (when compared with the reference language), or defining heuristics on the already approximated models of the type checking or sacrificing on the features of the underlying language in general. This way we want to focus on the understanding of the type checking of the existing implementation and all the limitations that come with it. To the best of the author's knowledge we are the first to propose the type debugging of the *complete* existing implementation. The previous attempts have either only allowed a limited subset of the OCaml language (Tsushima and Asai [2013]) or considered only the toy language implementation (Duggan and Bent [1996]), both of which are unsatisfactory for our purposes. The extraction of the decisions of the type checker is a non-trivial task in itself because one has to operate within the strict limitations of the existing compiler which logic must not be changed. Furthermore, with the straightforward logging of the type checking process we are more than likely to affect the performance of the regular compilations, which is unacceptable.

The compilers of the mainstream programming languages are programs themselves. Intuitively, this means that we should be able to provide runtime debuggers or trace analyzers with an automatic or a manual instrumentation of its implementation, done by the language architects. For the purposes of backtracking and navigation capabilities, in general, over the type checking decision process, we require a much richer representation than what is likely to be generated from plain bytecode instrumentation points (Kiczales et al. [2001]).

The two insights, the precise representation of the type checking process, and its analysis in search for the minimal sources of types do not cancel the contributions of the related work. In fact with the precise locations of the origin of the conflicting types we offer solid foundations for developing our own family of heuristics and interactive type debuggers, specifically tailored to the needs of local type inference and mainstream languages.

In this thesis we address the following problems:

- Can we extract the details of the existing type checker in a non-intrusive way that does not modify its internal logic?
- What are the good high-level data structures for representing and navigating the decision process of local type inference and how can we infer them from the existing implementation?

- How can we infer the minimal fragments of the source code that are responsible for the introduction of types, or their fragments, for the first time in local type inference algorithm?
- How can we explain and correct the limitations of local type inference algorithm and its implementations?
- How can we improve the localization and the explanation of the type errors for programming languages using local type inference?
- What are the necessary abstractions to define an interactive type debugger tool for the programming languages using local type inference, an equivalent of a runtime debugger, that applies to both invalid and valid programs? How can programmers control the exploration of its decision process in an intuitive way without prior deep understanding of its theoretical foundations?

1.1 Desired properties of the algorithm

The idea of an algorithm that traces backwards through the already committed decision process of local type inference is different from the existing approaches; related work mostly manipulates some of the specially inferred constraints, collected either globally or locally. As the algorithm is a foundation of any type debugging technique proposed in this thesis, we briefly describe three of its properties that have to be satisfied in order to be useful in our desired applications.

White box

Local type inference propagates types or synthesizes types in an incremental manner, from one AST node to another. The process of type inference is AST-specific and, due to its complexity, oftentimes not obvious to the user. The primary purpose of the algorithm developed in this thesis is to find some minimal source locations that explain the inference of a given type. In other words we reduce the problem of type debugging to a small set of program fragments that should explain the type inference. In practice however, we also want to be able to inspect the intermediate decisions that led to such results in order to provide custom error feedback, define program-specific heuristics or allow for a more interactive approach to type debugging.

In the *black box* approach we only take an invalid program and return some improved error feedback. The approach is desirable but insufficient for our applications, hence the name of the property coined for our technique.

To illustrate, we consider a simple example of a generic case class in Scala:

```
1   def id(x: Int): Int = x
2   case class Either[A, B](left: A, right: B)
3
4   val v = Either(id(y), id(z))
5   ...
6   val x1: Boolean = v.left
7     // error: type mismatch;
8     // found:    Int
9     // required: Boolean
10    // val x1: Boolean = v.left
11    //                     ^
12
13  val x2: Boolean = v.right
14    // error: type mismatch;
15    // found:    Int
16    // required: Boolean
17    // val x2: Boolean = v.right
18    //                     ^
```

In the example, we define a local identity method `id` which takes an integer value and returns the same value. The `Either` class defines two values, `left` and `right` of some generic type. With the assignment in line 4, the inferred type of the local value `v` is intuitively `Either[Int, Int]`, based on the return type of the identity function. The assignment to the similarly looking local values `x1` and `x2` leads to a type mismatch error because of the conflict with the explicit type annotation expecting a boolean value.

The type error messages generated by the Scala compiler are clear in a sense that they precisely describe the conflicting types. However, lacking any type debugging method, they do not explain the origin of the individual types. While in the above example, tracing back the origin of types `Int` and `Boolean` is trivial, in real-life situations, and especially for the generic libraries, this is hardly the case.

Furthermore, a black box type debugging algorithm that only finds the minimal source locations that led to the inferred type, and hides any intermediate decisions, would (correctly return the following location for both of the cases:

```
def id(x: Int): Int = // ...
                ~~~
```

In practice, we always want to remain in control of the algorithm and its decision process, meaning that we still want to get a high-level overview of how types were inferred. For example, a white box algorithm gives us access to the intermediate decision points of the type checking process, meaning that we should be able to provide their corresponding locations and generate a more comprehensive type error report (for the synthesized type):

```
Type Int has been inferred in location(s):
val x1: Boolean = v.left
                  ~
val v = Either.cond(cond, id(y), id(z))
                          ~~~~~
def id(x: Int): Int = // ...
                ~~~
```

```
Type Int has been inferred in location(s):
val x2: Boolean = v.right
                  ~
val v = Either.cond(cond, id(y), id(z))
                                 ~~~~~
def id(x: Int): Int = // ...
                ~~~
```

The access to the intermediate decisions that led to the inference of types is particularly important for libraries or programs that are generic - programmers typically find it hard to track the instantiations of multiple type parameters (Jun et al. [2002]). With the analysis of the intermediate decisions of the type checking process we also improve its applicability to the advanced type system features. The latter may exhibit some non-standard decision process, and therefore should be considered separately without affecting the integrity of the complete algorithm.

Precision

In order to find the minimal locations that determine the source of the type, it is important that the algorithm remains precise while crossing the boundaries of the adjacent AST nodes. The property also implies that the intermediate decisions reported by the algorithm also preserve the precision of the type that is being analyzed.

To illustrate, we consider a simple hierarchy of three classes and a conditional expressions that involves their instances:

```
class A; class B extends A; class C extends A
val b: B = // ...
val c: C = // ...
def single[T](x: T): List[T] = // ...
val cond: Boolean = // ...

val x = if (cond) single(b)
        else      single(c)
...
val y: List[Int] = x
// error: type mismatch;
// found:    List[A]
// required: List[Int]
// val y: List[Int] = x
//                    ^
```

In the example, the inferred type of of the local value 'x', `List[A]`, implies that the type inference has approximated the types of its two branches, `List[B]` and `List[C]`, by calculating their least upper bound. The inferred type later conflicts with the user expectation of type

`List[Int]`, as indicated in the type error message generated by the Scala compiler.

A precise algorithm, that traces back through the decisions of local type inference, can take into account the failed subtyping check between the two conflicting list values. In particular, for the above example we would have to be able to analyze the source of the type element `A` and type element `Int` from the synthesized type `List[A]` and the expected type `List[Int]`, respectively, since the real source of the error lies in their type arguments.

Furthermore, a precise algorithm has to analyze only those types that participate in the type approximations that directly contribute to the inference of the initial type, or its fragment. The latter is particularly important for polymorphic function calls that involve the inference of type parameter instantiations using the type information from a number of local type checking decisions.

In the above example, the precision translates to the ability to find the minimal locations representing the source of the type element `A` in a type mismatch conflict. This means that the algorithm can return locations

```
Location (1):                                     Location (2):
val b: B = // ...                                 val c: C = // ...
       ~                             and                 ~
val x = if (cond) single(b)                               else        single(c)
                         ~                                                   ~
```

rather than only some non-minimal solution that cannot cross the boundaries of function applications with the elided type arguments in

```
Location (1):                                     Location (2):
val x = if (cond) single(b)          and                  else        single(c)
                  ~~~~~~~~~                                           ~~~~~~~~~
```

or worse, cannot handle the least upper bound approximations of the conditionals by reporting location

```
Location (1):
val x = if (cond) single(b)
        ~~~~~~~~~~~~~~~~~~~
        else      single(c)
        ~~~~~~~~~~~~~~~~~~~
```

Autonomous

The algorithm analyzing the decisions of local type inference has to be *autonomous*. The latter means that the information about which type we want the algorithm to analyze must be sufficient for its execution.

An algorithm that is not autonomous, meaning that it requires continues feedback from programmers in order to guide the analysis of the type checking decisions, would have two important drawbacks:

1. The analysis of the intermediate results of the analysis, and the type checking decisions in general, by the users can be a time consuming process.
2. Users would have to have extensive prior knowledge of the type system, type inference, and its implementation in the programming language.

While some of the related work, such as Chen and Erwig [2014a] and Sulzmann [2002], improve their type debugging results by requiring user input, in our algorithm such input can only be optional.

1.2 The target audience of the type debugger

The algorithm that finds the minimal locations that determine the source of a type, solves only half of the problem of decrypting local type inference. The other half involves its applicability to the advanced type system features, and its presentation to the users in an intuitive form. Due to the range of the possible type system features, and their varying complexity, it is not always satisfactory to explain the type error through a simple type error message.

Another complication for generating improved feedback lies in the target audience. Users come with a varying level of expertise and it is unfeasible to provide a single solution that fits everybody's expectations. Both aspects, which are accounted for in the thesis, are now briefly discussed.

Beginner users

Lack of experience makes beginner users particular vulnerable to the confusing type error messages. In general, with time, users tend to recognize the patterns in the type error messages and get better with scanning programs for the source of the conflicting types.

In order to improve the language experience, we generate error messages that provide more type checking context information, especially in case of known implementation limitations.

Moreover beginner users tend to rely less on the advanced features, making it easier to define heuristics that workaround the typing problems. That is why in our thesis we explore the possibility of suggesting local code modifications that correct invalid program fragments. The corrections discussed in this thesis have drawn inspiration from the questions asked on the mailing lists and language forums, that are typically used by the beginner users.

Intermediate users

With the increased confidence in the language, grows also the complexity of the type errors that we have to tackle in the type debugger. Many of the examples presented in this thesis will be synthetically constructed but they will always exhibit properties of errors that have been encountered in real life applications, only with the hundreds of lines of code involved. Such errors are typically encountered by the intermediate and the advanced users who less often need more feedback, but when they do it usually has to be quite detailed and precise.

By providing a number of examples that combine multiple type system features, we aim to show that our techniques can apply to non-trivial problems that are typically encountered by the intermediate users.

Library designers and compiler hackers

The library designers and the compiler hackers typically have a very good understanding of the limitations of the underlying language and can push the limits of the type system and the type inference. Since in such situations it is hard to predict the kind of errors that can be encountered, we want to make sure that our tool can provide the necessary freedom of exploration. In particular, we explore the possibility of supporting interactive, user-driven analysis of the type checking decisions.

Our type debugging framework exposes the data structures representing the decisions of the type checking process. Together with a set of specialized functions that analyze the intermediate results of the core algorithm, it allows us to explore the possibility of customized error feedback. The customization of the DSL and library error messages has received relatively little attention (Heeren et al. [2003b] and Sackman and Eisenbach [2008] being exceptions), which is surprising because those kinds of errors are very often highly specific and should be easily identifiable.

1.3 Terminology

Before we begin discussing the details of the algorithm that analyzes the decisions of local type inference and its applications, we need to agree on terminology. We give an overview of the terms that will be used in the rest of the thesis.

The terms *type inference* and *type checking* are used interchangeably and refer to the process of assigning types to terms and verifying their correctness with respect to the underlying type system. The term type inference is also known in the literature as *type reconstruction*.

Local type inference, and its variant Colored Local Type Inference in particular, are realized using the *type inference rules*. The type inference rules realize the *inference judgment* that essentially assigns types to terms given some type checking context and some environment. The inference judgment does not provide information about the type inference rule used in the term, unless explicitly stated. The instantiations of type inference rules form *derivations*, with terms and types replaced with the concrete values. In the thesis we also employ the term *type derivation trees* which underlines the shape of a structure created by a repeated application of the type inference rules to the expression. Unless otherwise specified, the derivation refers to a fragment (or a node) of the type derivation tree.

We distinguish between the different kinds of types, based on the fact of how they are involved in the process of local type inference and during the debugging of its decisions. The *inferred type* refers to the final type assigned to the term, as a result of a type checking process. The types in the type inference process can be either *synthesized* or *inherited*, meaning that the type either comes from the term or from the expected type checking context. The types can be also partially synthesized and partially inherited, which refers to some of the individual type elements of the type. In the case of a type mismatch, the *conflicting types* refer to types that participate in a type mismatch where the type of the term fails to conform to the expected inherited type.

The algorithm analyzing the decisions of local type inference attempts to find the of the type that is provided as input. To avoid ambiguous notation we always refer to this type as the *target type*.

The formal approach to type debugging, as well as the theoretical foundations of type inference use the term *type variable* for the parametric polymorphism, while the implementation sections and the Scala examples use the term *type parameter*. Both are equivalent. In addition, we also divide type parameters into local and non-local ones, depending if they are defined as part of the method type signature or if they are defined as part of the class or trait type signature, respectively.

The *instrumentation* refers to the process of inserting the low-level side-effecting function applications that do not modify the execution of a program and only collect its runtime values with a minimal runtime overhead. The instrumentation can be either added explicitly by the programmer (or manually) in the source code, or in a semi-autonomous way through a third-party framework and its instrumentation rules.

Throughout the thesis we will interchangeably use terms *type selection* and *type extraction*.

Both of them refer to the process of taking a type element from a type, where the identity of the element is determined by the particular semantics of type selection at a given point.

1.4 Overview

Before we delve into the details of the type debugging technique, in Chapter 2 we briefly introduce the formalization of Colored Local Type Inference (Odersky et al. [2001]), a variant of Local Type Inference which strictly supersedes it in terms of the capabilities. The formal language and its type inference rules will serve as a basis for our formal definition of the type debugging technique.

The algorithm that defines the analysis of the decisions of local type inference has been described in the Chapter 3. The chapter starts with an informal explanation of the encodings of two invalid programs using the visual interpretation of type derivation trees. Later we provide an informal overview of the algorithm that realizes such analysis. The description provides an overview of the algorithm, including its possible input and output values, as well as the reasoning behind returning values that can undergo further analysis.

With the intuition in place, in Section 3.3 we introduce formally an abstraction that controls the traversing of the nodes of the type derivation trees, later known as *TypeFocus*. Later we formally introduce the analysis of type checking decisions through a series of examples and explanations of the individual type inference rules (Section 3.5.2) only to provide a complete algorithm in Section 3.5.3. The second part of the chapter discusses in detail possible intermediate results of the algorithm, how they can be further analyzed and how do they translate to program locations.

With the analysis algorithm being defined only for the valid derivations, it is necessary to explain . In particular, we take a formal stand at explaining type mismatch errors and show how can we extract from it the information necessary to trigger the type debugging algorithm (Chapter 4).

Chapter 5 introduces the instrumentation technique used to extract the internal details of the type checking process. We provide the details of the data structure representing the high-level decisions process (Section 5.2) and its translation from the low-level instrumentation data (Section 5.3). The chapter concludes with a discussion on the practical challenges of instrumenting an existing type checker and our solutions for the Scala implementation (Section 5.5).

We describe the elements of the implementation of the type debugging tool in Chapter 6, including the necessary minimal support in the compiler infrastructure. In particular, we describe the examples of implementing an improved error feedback using the data structures and the analysis algorithm described in the previous chapters. In Section 6.7 we present our technique to providing surgical-level code modifications that can for example workaround the limitations of local type inference.

We conclude our thesis with a description of a number of applications that are possible with our type debugging tool, including the improved feedback for the non-trivial language constructs and library-specific plugins (Section 7.2) and the interactive type debugging techniques (Section 7.3). We also describe the main properties of the analysis of the type-driven implicit resolution, which turned out to be a special case of our core analysis algorithm (Section 7.1).

1.5 Contributions

This thesis makes the following contributions:

- A novel approach to exploring decisions of local type inference exposed through type derivation trees. We define a complete algorithm that traverses the decisions of type derivation trees in a controlled fashion. In order to enable the deterministic navigation we define a new abstraction, named *TypeFocus*, that encapsulates information about type propagation as we walk the nodes of derivations.
- A formal reduction of type mismatch conflicts to an application of the *TypeFocus*-driven algorithm.
- A lightweight instrumentation framework for extracting the low-level data from the existing type checker implementations for programming languages that use local type inference. We provide an automatic method to infer the high-level type derivation trees from the extracted low-level instrumentation.
- A technique, based on the results of the *TypeFocus*-based analysis for defining surgical-level code modifications that correct the type errors. In particular, the mechanism allows for generating precise error feedback that workarounds the limitations of local type inference.
- A specialization of the *TypeFocus*-based algorithm to analyzing the decisions of implicit resolution.
- A real world validation of these techniques; we integrate our tool into the full-fledged Scala compiler, and show a detailed analysis of a number of type problems that could not have been tackled before.
- An interactive type debugger that allows for a guided exploration of the typing decisions of the valid and invalid Scala programs. The interactive mode is an extension of the *TypeFocus*-based algorithm that can analyze the typing scenarios autonomously as well as using the user-provided type input.

Chapter 2

Preliminaries

The formalism presented in this dissertation builds on top of the existing formalisms for Local Type Inference defined in Pierce and Turner [2000]. In contrast to global type inference, local type inference has proved to be a suitable choice for languages supporting nominal subtyping and parametric polymorphism, such as `System` $F_{\leq}$, where a complete type inference for full programs is known to be undecidable (Wells [1999]).

In Section 2.1 we introduce the formalism of Colored Local Type Inference by Odersky et al. [2001]. Colored Local Type Inference refines, and essentially subsumes, the simple Local Type Inference by means of *coloring* individual types to indicate the propagation of partial type information. Thus it allows for omission of a larger number of type annotations in situations that are typically expected by the users. The formalism uses a more succinct representation, which in turn allowed us to define a clearer debugging formalism on top of it.

In Section 2.2 we outline the most significant differences between the Colored Local Type inference formalism and its implementation in the Scala language. Type inference in Scala is defined only in terms of the informal discussions (Odersky [2002]), or semi-formal language specification (Odersky [2015]), neither does it provide formalization of its advanced type system features, such as higher-kinded types (Moors et al. [2008]) or implicit inference (Oliveira et al. [2010]).

2.1 Colored Local Type Inference

The correctness of type inference is typically expressed in relation to the underlying type system for which it elides some type annotations. Therefore formalizations that we summarize in this section is expressed in terms of three individual parts:

- *internal language* - the fully typed language which provides all type annotations. In the case of discussed formalizations we mean some variant of `System` $F_{\leq}$.
- *external language* - a superset of the *internal language*, where some of the type arguments and the types of the parameters of anonymous functions can be omitted. The *external language* is visible to the end users. The *external language* does not specify how the missing type annotations are inferred but only defines the constraints that the *guessed* solution has to satisfy in order to be correct and classified as the optimal one.
- *type inference* - a formal relation between the *external language* and the *internal one* that describes how type annotations are inferred.

The formalism and type system described in Colored Local Type Inference by Odersky *et al.* combines type checking and type synthesis rules of simple Local Type Inference into one coherent system. In Section 2.1.1 we present the grammar for the core language used in the formalization. For clarity we use the same core language for our formal treatment of debugging techniques, presented later in the thesis. In Sections 2.1.2 and 2.1.3 we briefly introduce a selection of the core type assignment rules for the internal and the external language of Colored Local Type Inference. Subsequently, we can define a formal connection between the two languages with the detailed description of the type inference rules (Section 2.1.4). The debugging of type inference decisions requires a good understanding of the collection and solving of local type constraints which, due to its complexity, deserve a separate discussion in Section 2.1.5.

2.1.1 Grammar

Definition 1 shows the syntax of the external language of the Colored Type System, which itself is based on `System` $F_{\leq}$ extended with records, as per Odersky et al. [2001]. The grammar gives terms, types and environments of the language.

A term can be a variable x, a record constructor $\{x_1 = E_1, \ldots, x_n = E_n\}$ or a record selection $E.x$. It also has two versions of function application and abstraction: ones with explicit type parameters and type arguments ($F[T](E)$ and $\mathtt{fun}[\overline{a}](x : T)E$) and those that elide them, if possible, by conveniently inferring them from the context ($F(E)$ and $\mathtt{fun}(x)E$), respectively. The overbar in $\overline{a}$ means a finite sequence of local type parameters, equivalent to $a_1, \ldots, a_n$ for some n. The empty sequence is represented using the ϵ symbol. The examples used in the thesis often use multi-parameter functions, that can be always encoded using record constructors.

A type is a either a type variable a, the top $\top$ or the bottom type $\bot$ in the type hierarchy[1], a potentially polymorphic function type $\forall\overline{a}.T \to S$ (the universal quantifier extends over the

[1] In the Scala type hierarchy the top and the bottom types are equivalent to `Any` and `Nothing`, respectively. Java has no bottom type and the top is equivalent to `Object`.

Definition 1 Core language syntax, from [Odersky et al., 2001, pg 3].

Terms	E, F	$=$	$x \mid E.x \mid \texttt{fun}[\overline{a}](x\colon T)E \mid \texttt{fun}(x)E \mid F[\overline{T}](E)$
		$\mid$	$F(E) \mid \{x_1 = E_1,\ \ldots,\ x_n = E_n\}$
Types	T, S, R	$=$	$a \mid \top \mid \bot \mid \forall \overline{a}.T \to S \mid \{x_1 : T_1,\ \ldots,\ x_n : T_n\}$
Environments	Γ	$=$	$x : T \mid \epsilon \mid a \mid \Gamma, \Gamma'$

whole function type), or a record type. In contrast to the source formalization, which uses a $T \xrightarrow{\overline{a}} S$ notation for polymorphic function types (with type variables written over the arrow), we chose an equivalent notation that is more common in the literature.

2.1.2 Internal language

The internal language is based on the established formalization of `System` $\texttt{F}_{\leq}$, and does not allow for the elided versions of function applications and anonymous functions. For completeness, Figure 2.1 includes a complete set of type assignment rules for the internal language, as well as the subtyping rules of the internal Colored Type System.

The subtyping rules assume the standard semantics, where the $\bot$ and $\top$ types are the subtype (the `(BOT)` rule) and the supertype (the `(TOP)` rule) of every type, respectively, and every type variable is a subtype of itself (the `(VAR)` rule). Since the polymorphic function types are contravariant in the parameters and covariant in the result type, the order of types in the former is reversed for the subtype derivation to be established (the `(FUN)` rule). Similarly, the two record types are subtypes (the `(REC)` rule) if and only if their corresponding type elements are subtypes as well.

2.1.3 External Colored Type System

The external language of the Colored Type System allows for *partial erasure* of terms; types of parameters in the abstractions and type arguments of the function applications can be elided. To allow for type assignment for terms with elided type information Odersky et al. [2001] introduced colored types which carry information about the direction in which type information can be propagated. The formalization uses the ${}^{\vee}T$ superscript and the ${}_{\wedge}T$ subscript annotations to indicate if the type information has been inherited from the type checking context or synthesized directly from the term, respectively.

For example, type ${}^{\vee}(\forall \overline{b}.b \to {}_{\wedge}\{x : T,\ y : S\})$ implies that the function type has been enforced by the type checking context, along with the type of the parameter b, but the result type of the function type involving a record type has been synthesized from some record term. Lack of color next to the type implies that the type could be either synthesized or inherited. To

$$(\text{VAR})\ \Gamma \vdash x:\ \Gamma(x)$$

$$(\text{ABS})\ \frac{\Gamma,\ \overline{a},\ x:T \vdash E:\ S}{\Gamma \vdash \mathsf{fun}\left[\overline{a}\right](x:T)E:\ \forall\overline{a}.T \to S}$$

$$(\text{APP})\ \frac{\Gamma \vdash F:\ \forall\overline{a}.S \to T \qquad \Gamma \vdash E:\ S' \qquad S'\ <:\ [\overline{R}/\overline{a}]\,S}{\Gamma \vdash F\left[\overline{R}\right](E):\ [\overline{R}/\overline{a}]\,T} \qquad (\text{APP}_\perp)\ \frac{\Gamma \vdash F:\ \perp \qquad \Gamma \vdash E:\ R}{\Gamma \vdash F\left[\overline{R}\right](E):\ \perp}$$

$$(\text{SEL})\ \frac{\Gamma \vdash F:\ \{x_1:T_1,\ ....,\ x_n:T_n\}}{\Gamma \vdash F.x_i:\ T_i} \qquad (\text{SEL}_\perp)\ \frac{\Gamma \vdash F:\ \perp}{\Gamma \vdash F.x_i:\ \perp}$$

$$(\text{REC})\ \frac{\Gamma \vdash F_1:\ T_1 \quad ... \quad \Gamma \vdash F_n:\ T_n}{\Gamma \vdash \{x_1 = F_1, ..., x_n = F_n\}:\ \{x_1:T_1, ..., x_n:T_n\}}$$

$$(\text{BOT})\ \perp\ <:\ T \qquad (\text{REC})\ \frac{T_1\ <:\ T_1' \quad ... \quad T_m\ <:\ T_m'}{\{x_1:T_1,\ ...,\ x_m:T_m,\ ...,\ x_n:T_n\}\ <:\ \{x_1:T_1',\ ...,\ x_m:T_m'\}}$$

$$(\text{TOP})\ T\ <:\ \top$$

$$(\text{VAR})\ a\ <:\ a \qquad (\text{FUN})\ \frac{T_1'\ <:\ T_1 \quad T_2\ <:\ T_2'}{\forall\overline{a}.T_1 \to T_2\ <:\ \forall\overline{a}.T_1' \to T_2'}$$

Figure 2.1: A Colored Type System $\Gamma \vdash E : T$ for the internal language and the subtyping relation $S\ <:\ T$ (as presented in [Odersky et al., 2001, pg. 4]).

Definition 2 Syntax for the types of the external Colored Type System, from [Odersky et al., 2001, pg 4].

Synthesized Types	${}_\wedge T,\ {}_\wedge S,\ {}_\wedge R$	$=$	${}_\wedge a \mid {}_\wedge\top \mid {}_\wedge\perp \mid {}_\wedge(\forall\overline{a}.{}_\wedge T \to_\wedge S) \mid \{x_1 :_\wedge T_1,\ ...,\ x_n :_\wedge T_n\}$
Inherited Types	${}^\vee T,\ {}^\vee S,\ {}^\vee R$	$=$	${}^\vee a \mid {}^\vee\top \mid {}^\vee\perp \mid {}^\vee(\forall\overline{a}.{}^\vee T \to^\vee S) \mid \{x_1 :^\vee T_1,\ ...,\ x_n :^\vee T_n\}$
Environments	Γ	$=$	$x :_\wedge T \mid \epsilon \mid {}_\wedge a \mid \Gamma, \Gamma'$

avoid an excessive use of colors, a type that lacks it always assumes the color of the closest enclosing type constructor it is part of. The diamond notation next to the type annotation, $\diamond T$, implies that the type can be either synthesized or inherited from the context.

A fragment of the subtyping relation $\leq$ for the colored types is shown in Figure 2.2. The construction of the rules ensures that when going from a subtype to a supertype in a subtyping relation the change is only allowed from the synthesized subtype to the inherited supertype. This way the complete formalization ensures that synthesized types, in particular type constructors, cannot be guessed using the subsumption rule, *i.e.,* if the type is synthesized, then it has really been synthesized from the term at some point in the type derivation tree.
For example, ${}_\wedge(\forall\overline{a}.Int \to Int)\ \leq\ {}_\wedge(\forall\overline{a}.{}^\vee\perp \to {}^\vee\top)\ \leq\ {}^\vee\top$ but ${}_\wedge(\forall\overline{a}.Int \to Int)\ \not\leq\ {}_\wedge\top$.

The subtyping relation for colored types is safe with respect to color-less types, as stated

$$T \leq T \qquad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \qquad \frac{T_1' \mathrel{\overline{<}} T_1 \quad T_2 \leq T_2'}{\forall \overline{a}.T_1 \rightarrow T_2 \leq \forall \overline{a}.T_1' \rightarrow T_2'}$$

$$ {}_{\wedge}a \leq {}^{\vee}a \qquad {}_{\wedge}\bot \leq {}^{\vee}(\forall \overline{a}.{}_{\wedge}\top \rightarrow {}_{\wedge}\bot) \qquad {}_{\wedge}(\forall \overline{a}.{}^{\vee}\bot \rightarrow {}^{\vee}\top) \leq {}^{\vee}\top$$

$$ {}_{\wedge}\bot \leq {}^{\vee}\bot \qquad {}_{\wedge}\bot \leq {}^{\vee}\top \qquad {}_{\wedge}\bot \leq {}^{\vee}a \qquad {}_{\wedge}a \leq {}^{\vee}\top$$

Figure 2.2: A fragment of the subtyping relation for colored types, $S \leq T$ (as presented in [Odersky et al., 2001, pg. 6]). The $S \mathrel{\overline{<}} T$ is equivalent to $S \leq T$ but with the inverted colors.

$$(\text{VAR})\ \frac{\Gamma(x) = {}_{\wedge}T}{\Gamma \vdash^{c} x :\ {}_{\wedge}T} \qquad (\text{SUB})\ \frac{\Gamma \vdash^{c} E :\ T \quad T \leq T'}{\Gamma \vdash^{c} E :\ T'}$$

$$(\text{ABS})\ \frac{\Gamma, {}_{\wedge}\overline{a}, x :{}_{\wedge}T \vdash^{c} E :\ S \qquad \overline{a} \not\in \mathtt{tv}(E)}{\Gamma \vdash^{c} \mathtt{fun}(x)E :\ {}^{\vee}(\forall \overline{a}.T \rightarrow \diamond S)} \qquad (\text{ABS}_{tp})\ \frac{\Gamma, {}_{\wedge}\overline{a}, x :{}_{\wedge}T \vdash^{c} E :\ S}{\Gamma \vdash^{c} \mathtt{fun}\big[\overline{a}\big](x : T)E :\ {}_{\wedge}(\forall \overline{a}.T \rightarrow \diamond S)}$$

$$(\text{APP}_{tp})\ \frac{\Gamma \vdash^{c} F :\ {}^{\vee}(\forall \overline{a}.{}_{\wedge}S \rightarrow {}_{\wedge}T) \qquad \Gamma \vdash^{c} E :\ [\overline{R}/\overline{a}]^{\vee}S}{\Gamma \vdash^{c} F\big[\overline{R}\big](E) :\ [\overline{R}/\overline{a}]_{\wedge}T} \qquad (\text{SEL})\ \frac{\Gamma \vdash^{c} E :\ {}^{\vee}\{x : \diamond T\}}{\Gamma \vdash^{c} E.x :\ T}$$

$$(\text{APP})\ \frac{\begin{array}{c}\Gamma \vdash^{c} F :\ {}^{\vee}(\forall \overline{a}.{}_{\wedge}S \rightarrow {}_{\wedge}T) \qquad \Gamma \vdash^{c} E :\ S' \qquad S' \lhd_{\overline{a}} {}^{\vee}S \\ S' \leq [\overline{R}/\overline{a}]^{\vee}S \quad [\overline{R}/\overline{a}]_{\wedge}T \leq T' \\ \forall \overline{R}', T''.\ (S' \leq [\overline{R}'/\overline{a}]^{\vee}S \ \wedge\ [\overline{R}/\overline{a}]_{\wedge}T \leq T'' \sim T' \implies [\overline{R}/\overline{a}]_{\wedge}T \leq [\overline{R}'/\overline{a}]^{\vee}T)\end{array}}{\Gamma \vdash F(E) :\ T'}$$

Figure 2.3: A fragment of the Colored Type System $\Gamma \vdash^{c} E :\ T$ for the external language (as presented in [Odersky et al., 2001, pg. 6]).

through the two properties ([Odersky et al., 2001, Lemma 5.1]):

- $T \leq S$ implies $T <: S$, *i.e.,* the $\leq$ is a restriction of $<:$ in a sense that any two colored types that are subtypes in the external language are always subtypes in the internal language.
- $T <: S$ implies ${}_{\wedge}T \leq {}^{\vee}S$, guarantees that a term of type ${}_{\wedge}T$ will be a subtype of any supertype of T that is given from the outside.

We present a fragment of the Colored Type System for the external language in Figure 2.3. The type assignment is defined using the $\Gamma \vdash^{c} E :\ T$ judgment. The type system is realized through a set of declarative rules that includes the subsumption rule `(SUB)`. We give only a brief overview of the essential elements of the rules, to provide a basis for further discussion on the type inference in the next section; for the complete description we refer the reader to the formalization of Odersky et al. [2001].

The `(VAR)` rule synthesizes the type of the variable from the environment; the resulting type

${}_{\wedge}T$ and its color clearly relates the source of the assigned type with the term.

The type system gives separate rules for the two kinds of abstraction, one with elided type information and one with a complete type signature, `(ABS)` and `(ABS`$_{tp}$`)` respectively. The crucial difference is visible in the color of the type assigned to the term; the type constructor as well as the parameter type of the function type in the `(ABS)` rule has to be inherited from the context, while in the `(ABS`$_{tp}$`)` rule it is synthesized from the term; the syntax of the term provides sufficient type information for the synthesis to take place. The rules also highlight the difference with respect to the type system of Local Type Inference formalization; the latter can either synthesize the complete function type, *i.e.,* ${}_{\wedge}(\forall\overline{a}.S \to T)$, or inherit the complete function type, *i.e.,* ${}^{\vee}(\forall\overline{a}.S \to T)$, but it cannot assign a type that has been partially inherited, as it is the case for the `(ABS)` rule.

Similarly, the record selection rule, `(SEL)`, requires that the type of record term itself be inherited while the type of the record member itself can be either synthesized or inherited; the requirement is propagated to the premise of the rule given the known shape of the term. Similarly as in the case of the abstraction, such partial type information propagation is not possible in the Local Type Inference formalization.

Not surprisingly, the rule for assigning the type to a function application, `(APP)`, is the most complicated one. The first premise of the rule restricts the type that can be assigned to a function term - the propagated function type requirement is expressed through the inherited *color* in ${}^{\vee}(\forall\overline{a}.{}_{\wedge}S \to {}_{\wedge}T)$. Then, through the $S' \lhd_{\overline{a}} {}^{\vee}S$ premise, the rule ensures that the type assigned to the argument of the function, $\Gamma \vdash^{c} E : S'$, coincides with the parameter part of the function type, modulo the occurrences of the $\overline{a}$ type variables. The latter condition does not impose any restrictions on the instances of type variables; the selection of the optimal $\overline{a}$ type variables is expressed by the remaining premises.
The *guessed* type arguments $\overline{R}$ have to satisfy a number of conditions in order to assign a sound type to the function application:

- $S' \leq [\overline{R}/\overline{a}]^{\vee}S$:
  The type of the parameter of the function type with the *guessed* type arguments has to be a supertype of the type of the argument.

- $[\overline{R}/\overline{a}]_{\wedge}T \leq T'$:
  The type of the result of the function type with the *guessed* type arguments has to be a subtype of some minimal, *guessed* type T'.

- The specification of the minimal, *guessed* type T', that will be assigned to the function application term, is determined in the last implication. The condition ensures that any other choice of type arguments, *i.e.,* $\overline{R}'$, and the final type which coincides with type T' on the type elements that have been inherited, *i.e.,* T'', will always yield a type that is larger, with respect to the subtyping ordering.

The described Colored Type System has been shown to be sound with respect to the colorless type system.

2.1.4 Type inference rules

The inference judgment, $(P, \Gamma \vdash^{w} E : T)$, consists of the term to be typed, E, that eventually is assigned type T in a type environment Γ, and a prototype P representing parts of the type of E that are inherited from the context. P can be treated as a regular type, potentially having type holes, ?, representing the unknown parts. The prototype fully encapsulates the concept of colored types and their partial type information propagation.

Figure 2.4 presents a selection of the most interesting inference rules for Colored Type Inference, which follow naturally from the formalization of the external language. While simple Local Type Inference distinguished synthesis and type checking of type annotations through separate rules, the concept of a prototype and holes in it combines it. Moreover, Colored Local Type Inference allows for propagating strictly more type information between the nodes of the terms; in the simple Local Type Inference only ? or fully defined types, *e.g.,* $Int \rightarrow Int$, would be allowed as prototypes, but no partial ones, *e.g.,* $Int \rightarrow ?$.

For example, for some function g of type $\forall \overline{a}.((Int \rightarrow a) \rightarrow a)$ applied to an anonymous function $(\texttt{fun}(\,x\,)\,x)$ in '$g\ (\texttt{fun}(\,x\,)\,x)'$, the Colored Type Inference is capable of inferring the desired type of the application in the derivable judgment $(Int \rightarrow\ ?, \epsilon \vdash^{w} g\ (\texttt{fun}(\,x\,)\,x) :\ Int)$. The type of the parameter of x in the anonymous function $\texttt{fun}(\,x\,)\,x$ is inherited from the context, and the result type is synthesized from the body of the function. On the other hand, when some function h of type $\forall \overline{a}.((a \rightarrow a) \rightarrow a)$ is applied to the same anonymous function in $h\ (\texttt{fun}(\,x\,)\,x)$, the inference fails to come up with the right type of the parameter, as represented by a non-derivable $(? \rightarrow ?, \epsilon \vdash^{w} \texttt{fun}(\,x\,)\,x :\ T)$ fragment of the type derivation tree (T is unknown). Such type inference, or lack thereof, follows the expectation of the user, since the context of the function application has to provide enough type information for the type annotation not to be guessed.

To avoid soundness problems, any type assigned to the term has to conform to the type expected by the type checking context (whether the latter contributed to the inference of the type or not). Matching between the expected inherited type and the synthesized type is expressed through a $\nearrow$ operator. The $T \nearrow P$ notation means that either T is structurally equal to P, with ? filled by some arbitrary types, or we can find the smallest supertype of T which is structurally equal to P. The operation $T \searrow P$ is the dual of $T \nearrow P$, where the greatest subtype of T is structurally equal to P.

For brevity, we now only discuss a selection of the type inference rules that will be analyzed in our type debugging formalization, *i.e.,* (abs), $(\texttt{abs}_{tp})$ and (app), and refer the reader to Odersky et al. [2001] for a complete description of the rest of the rules.

$$\text{(VAR)}\ P, \Gamma \vdash^{w} x : \Gamma(x) \nearrow P \qquad \text{(sel)}\ \frac{\{x : P\}, \Gamma \vdash^{w} F : \{x : T\}}{P, \Gamma \vdash^{w} F.x : T}$$

$$\text{(abs)}\ \frac{P, \Gamma, \overline{a}, x : T \vdash^{w} E : S}{\forall\overline{a}.T \to P, \Gamma \vdash^{w} \mathbf{fun}(x)E : \forall\overline{a}.T \to S} \qquad (\text{abs}_{tp,?})\ \frac{?, \Gamma, \overline{a}, x : T \vdash^{w} E : S}{?, \Gamma \vdash^{w} \mathbf{fun}\left[\overline{a}\right](x : T)E : \forall\overline{a}.T \to S}$$

$$(\text{abs}_{tp})\ \frac{P', \Gamma, \overline{a}, x : T \vdash^{w} E : S}{\forall\overline{a}.P \to P', \Gamma \vdash^{w} \mathbf{fun}\left[\overline{a}\right](x : T)E : \forall\overline{a}.T \to S \nearrow \forall\overline{a}.P \to P'}$$

$$(\text{app}_{tp})\ \frac{?, \Gamma \vdash^{w} F : \forall\overline{a}.S \to T \quad [\overline{R}/\overline{a}]\, S, \Gamma \vdash^{w} E : [\overline{R}/\overline{a}]\, S}{P, \Gamma \vdash^{w} F\left[\overline{R}\right](E) : [\overline{R}/\overline{a}]\, T \nearrow P}$$

$$\text{(app)}\ \frac{\begin{array}{lll} ?, \Gamma \vdash^{w} F : \forall\overline{a}.S \to T & \vdash_{a} S' <: S & \Rightarrow C_1 \\ [?/\overline{a}]\, S, \Gamma \vdash^{w} E : S' & \vdash_{a} T <: \top \searrow P & \Rightarrow C_2 \end{array}}{P, \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T} T \nearrow P}$$

$$\text{(rec)}\ \frac{(P_1, \Gamma \vdash^{w} F_1 : T_1)\ \ldots\ (P_m, \Gamma \vdash^{w} F_m : T_m)\quad (\top, \Gamma \vdash^{w} F_{m+1} : T_{m+1})\ \ldots\ (\top, \Gamma \vdash^{w} F_n : T_n)}{\{x_1 : P_1, \ldots, x_m : P_m\}, \Gamma \vdash^{w} \{x_1 = F_1, \ldots, x_n = F_n\} : \{x_1 : T_1, \ldots, x_m : T_m\}}$$

Figure 2.4: A fragment of the type inference rules that realize the $(P, \Gamma \vdash^{w} E : T)$ inference judgment (from [Odersky et al., 2001, pg. 11])

The ability to infer the type of the parameter of the abstraction in the 'g (`fun(` x `)` $x)'$ example is formally defined in the rule `(abs)`; the rule requires a ?-free type T in the propagated parameter part of the prototype, $\forall\overline{a}.T \to P$. The result type of the prototype, P, does not directly influence the inferred of the type of the function. Indirectly, however, the P part is used in inferring the type of the body of the function, as expressed by an assignment of P as input to the rule's only premise. This agrees with the intuition of the Colored Type System; the result type of the inherited function type can only impose a requirement on the type of the body of the abstraction and that information is only passed around between the adjacent nodes of the type derivation tree.

The `(abs`$_{tp}$`)` and `(abs`$_{tp,?}$`)` rules apply to the abstraction term with the explicit type of the parameter, `fun`$\left[\overline{a}\right](x : T)E$. The shape of the propagated type information ($\forall\overline{a}.P \to P'$ and ?, respectively) allows to disambiguate the application of the rules to the abstraction term. Thus, together both of the rules correspond to the `(ABS`$_{tp}$`)` rule of the Colored Type System in Figure 2.3, where the assigned type could be both, synthesized and inherited (we recall from the subtyping rules of the Colored Type System that ${}_{\wedge}(\forall\overline{a}.\diamond T \to \diamond S) \leq {}^{\vee}(\forall\overline{a}.\diamond T \to \diamond S)$).

Similarly as in the Colored Type System, the most complicated type inference rule, `(app)`, infers the type of function application with elided type arguments. The first premise, $(?, \Gamma \vdash^{w} F : \forall\overline{a}.S \to T)$, requires that the synthesized type of the function is a function type. The inferred type of the function directly corresponds to the type assigned to the function in the `(APP)` rule of the Colored Type System, *i.e.,* $\Gamma \vdash^{c} F : \forall\overline{a}.{}^{\vee}({}_{\wedge}S \to_{\wedge} T)$; in both cases the function type constructor is enforced by the context of the function application.
The synthesized type elements of the function provide partial type information for inferring

$$\texttt{(CG-Top)}\ V \vdash_{\overline{X}} T <: \top \Rightarrow \emptyset \qquad \texttt{(CG-Bot)}\ V \vdash_{\overline{X}} \bot <: T \Rightarrow \emptyset$$

$$\texttt{(CG-Upper)}\ \frac{Y \in \overline{X} \quad S \Downarrow^{V} T \quad \texttt{fv}(S) \cap \overline{X} = \emptyset}{V \vdash_{\overline{X}} Y <: S \Rightarrow \{\bot <:\ Y <:\ T\}} \qquad \texttt{(CG-Lower)}\ \frac{Y \in \overline{X} \quad S \Uparrow^{V} T \quad \texttt{fv}(S) \cap \overline{X} = \emptyset}{V \vdash_{\overline{X}} S <: Y \Rightarrow \{T <:\ Y <:\ \top\}}$$

$$\texttt{(CG-Refl)}\ \frac{Y \notin \overline{X}}{V \vdash_{\overline{X}} Y <: Y \Rightarrow \emptyset}$$

$$\texttt{(CG-Fun)}\ \frac{\begin{array}{c} V \cup \overline{a} \vdash_{\overline{X}} T <: R \Rightarrow C' \quad V \cup \overline{a} \vdash_{\overline{X}} S <: U \Rightarrow C'' \\ \overline{a} \cap (V \cup \overline{X}) \end{array}}{V \vdash_{\overline{X}} \forall \overline{a}.R \rightarrow S <: \forall \overline{a}.T \rightarrow U \Rightarrow C' \wedge C''}$$

Figure 2.5: A complete constraint generation algorithm as defined by the rules of the $V \vdash_{\overline{a}} S <: T \Rightarrow C$ judgment in [Pierce and Turner, 2000, pg. 12].

the type of the other term elements in the function application. The type of the argument E can therefore be inferred with the help of the prototype involving the parameter of the function type, with all the unknown type variables substituted by wildcard constants, *i.e.,* $[{}^{?}/\overline{a}]\, S$; the substitution directly corresponds to the structural equality $S' \lhd_{\overline{a}}\ {}^{V}S$ premise, modulo the uninstantiated type variables, in the `(APP)` rule of the Colored Type System.
In order to infer concrete instantiations for type variables, the rule collects local type constraints using the $\vdash_{\overline{a}} S <: T \Rightarrow C$ judgment. The constraints originate from the two subtyping relations and correspond directly to the subtyping relations in the `(APP)` rule that specify the conditions for *guessing* the *minimal* set of type argument values. The next section describes in detail the process of collecting and solving of type constraints that leads to optimal type arguments.

2.1.5 Type constraints

Colored Local Type Inference infers locally optimal instantiations for all type variables that appear in the inferred polymorphic function types. The inference is a two-stage process - first the inference collects type constraints from subtyping relations, and later it solves them in an attempt to come up with an optimal solution with respect to the result type of the function. The constraint generation technique used in Colored Local Type Inference is a direct translation of the one used in the simple Local Type Inference, as described in Pierce and Turner [2000].

To represent the collected type constraints, Local Type Inference uses so called $\overline{a}$-constraint sets. The $\overline{a}$-constraint set C is defined by Pierce and Turner [2000] as a set of inequalities $\{S_i <:\ a_i <:\ T_i\}$ for each of the type variables $a_i \in \overline{a}$. Since the inequalities are essentially the lower and upper type bounds of the type variable, we will use the latter terminology in our discussion.

Individual type constraints collected from the subtyping relation <: are in a form of either upper $\{a <: R\}$, or lower $\{R <: a\}$ type bounds, for some type variable a and type R. The corresponding lower $\bot$ and upper $\top$ type bounds are added implicitly, respectively. For reference, Figure 2.5 provides a complete set of rules that realize the constraint generation judgment, and that need to be analyzed by any type debugging mechanism if one needs to explain the inferred type bounds.

The constraint generation algorithm for Local Type Inference assumes that $(\mathtt{fv}(S_i) \cup \mathtt{fv}(T_i)) \cap \overline{a} = \emptyset$, where `fv` returns a set of free variables from a type. The condition ensures that the subsequent inference of the type substitution is a *matching-modulo-subtyping* problem rather than a *unification-modulo-subtyping* problem which would prevent as from guaranteeing the principality of the inferred types.

For illustration purposes, we present the inference of constraints sets for four subtyping relations:

$$\begin{array}{lll}
(1) & \vdash_a Int \to Int <: a & \Rightarrow \{Int \to Int <: a <: \top\} \\
(2) & \vdash_a Int \to Int <: a \to a & \Rightarrow \{Int <: a <: Int\} \\
(3) & \vdash_{a,b}\, a \to b <: (\bot \to \bot) \to Int & \Rightarrow \{\bot \to \bot <: a <: \top, \bot <: b <: Int\} \\
(4) & \vdash_a a \to Int <: \forall \overline{b}.b \to Int & \Rightarrow \{\top <: a <: \top\}
\end{array}$$

Type constraints for the same type variable are combined using the *meet* operation. For constraint sets C and D, their *meet*, $C \wedge D$, calculates least upper bound, $\vee$, of their lower bounds, and greatest lower bound, $\wedge$, of their upper bounds:

$$\{\, S_i \vee U_i <: a_i <: T_i \wedge V_i \mid S_i <: a_i <: T_i \in C \text{ and } U_i <: a_i <: V_i \in D \,\}$$

For example, for $\{Int \to Int <: a <: \top\} \subseteq C$ and $\{(\bot \to \bot) <: a <: \top, \bot <: b <: Int\} \subseteq D$, $E = C \wedge D$ is equivalent to $\{\bot \to Int <: a <: \top,\ \bot <: b <: Int\}$. Similarly, the second constraint set in the above examples results from the approximation of the $\{Int <: a <: \top\}$ and $\{\bot <: a <: Int\}$ constraints.

The last generated a-constraint set in the above examples provides a surprising lower type bound for the type variable a, $\{\top <: a\}$, rather than $\{b <: a\}$. The former is a result of a *variable-elimination-by-promotion/demotion* operation which eliminates the occurrences of *out-of-scope* type variables by substituting them with either the supertypes (*promotion*, denoted as $\Uparrow$) or subtypes (*demotion*, denoted as $\Downarrow$) of the types that are type variable-free, a process that has been formally described in [Pierce and Turner, 2000, Section 3.2].
For example,

- *variable-elimination-by-promotion*: $b \Uparrow^{\{b\}} \top$ and $b \to \bot \Uparrow^{\{b\}} \bot \to \bot$.

- *variable-elimination-by-demotion*: $b \Downarrow^{\{b\}} \bot$ and $b \to \bot \Downarrow^{\{b\}} \top \to \bot$.

The point of using variable elimination is to avoid generating $\overline{a}$-constraint sets that have variables in the type bounds that are outside of their scopes.

In practice, such variable-elimination operations do not pose any problems in our debugging techniques; the *variable-elimination* of the individual type bounds can be delayed based on the position of the *out-of-scope* type variable until the instance of the type variable is approximated or used in some type operation.

An $\overline{a}$-type substitution $\sigma_{C,R}$ represents an instantiation of type variables inferred from the $\overline{a}$-constraint set C with respect to some type R. Formally, the $\sigma_{C,R}$ type substitution is a finite map, with a domain that ranges over the set of type variables, *i.e.,* $dom(\sigma_{C,R}) = \overline{a}$. The domain $dom(\sigma_{C,R})$ of a substitution $\sigma_{C,R}$ is the set of type variables which do not map to themselves by the type substitution. The inferred map has to satisfy the individual approximated type bounds for each of the type variables

$$\forall a_i.\ a_i \in \overline{a} \wedge \{S_i <: a_i <: T_i\} \subseteq C \implies S_i <: \sigma_{C,R}(a_i) \wedge \sigma_{C,R}(a_i) <: T_i$$

and make the subtyping relation from which the constraints are collected from (*i.e.,* $S <: T$) satisfiable

$$\sigma_{C,R} S <: \sigma_{C,R} T$$

The optimality of the inferred $\sigma_{C,R}$ substitution is determined in terms of the position of each of the individual type variables from the $\overline{a}$-constraint set with respect to some type R. Pierce and Turner [2000] provides a formal specification of each of the potential positions; we only briefly recall that any type R can be either constant, covariant, contravariant or invariant in some type variable a, where $a \in \overline{a}$. In general the type can be determined to be covariant or contravariant in a type variable by examining whether a type variable occurs to the right or left of an arrow; in the case of function types taking other functions as arguments the rule can be applied recursively.
Since for any $\overline{a}$-constraint set there can be many different possible type substitutions satisfying the above subtyping requirements, the algorithm aims to find a *minimal substitution* $\sigma_{C,R}$. Pierce and Turner [2000] defines the *minimal substitution* $\sigma_{C,R}$ as follows:

For each $S_i <: a_i <: T_i \in C$:

- If R is constant or covariant in a_i, then $\sigma_{C,R}(a_i) = S_i$
- Else if R is contravariant in a_i, then $\sigma_{C,R}(a_i) = T_i$
- Else if R is invariant in a_i and $S_i = T_i$, then $\sigma_{C,R}(a_i) = S_i$
- Else $\sigma_{C,R}$ is undefined.

The information from the constraint sets is sufficient to infer optimal solutions. For example, given the $\{a, b\}$-constraint set E, such that $\{\bot \to Int <: a <: \top,\ \bot <: b <: Int\} \subseteq E$, the *minimal* substitution from the constraint set E with respect to some type T, such that $T = \{x : a,\ y : b \to Int\}$, would be inferred as $\sigma_{E,T} = [a \Rightarrow\ \bot \to Int,\ b \Rightarrow\ Int]$.

The simple examples that inferred the instantiations for the type variables a or b highlights an important challenge in understanding local type inference: any debugging technique that analyzes the typing decisions of local type inference has to be aware not only of the final instantiations of type variables, but also the specification that determines the minimal substitution, any least upper bound or greatest lower bound approximations of the involved type constraints, and the types of type constraints themselves.

2.2 Type inference in Scala

The type system implemented by the Scala language is largely based on the combination of the Colored Type System, described in the previous section, and the type system for path-dependent types formalized in Odersky et al. [2003]. The language has adopted a less restrictive variant of Colored Local Type Inference. The modifications improve the support for some of the common type system features, such as the implicit resolution, lower and upper type bounds, or higher-kinded types, by inferring type arguments in common scenarios. As a result, the implementation is closer to the type system proposed for Generic Java (GJ) in Bracha et al. [1998]. Unfortunately, apart from an informal description of type inference provided in the specification of the language in Odersky [2015], and a semi-formal note sent to the types mailing list in Odersky [2002], there exists no complete formalization of the implemented type inference algorithm. The debugging techniques in this thesis are fully based on the sound formalization of Colored Local Type Inference, but we will also show how our techniques apply to its more realistic variant based on the implementation in the Scala language.

$$(\text{INST-APP})\ \frac{\begin{array}{cc}\Gamma \vdash F : \forall\overline{a}.T' \to T'' \hookrightarrow F' & \Gamma \vdash E : U \hookrightarrow E' \\ \multicolumn{2}{c}{\{\overline{a}\},\ T';\Gamma \vdash E' : U\ \textbf{inst}\ E'' : T_E} \\ \sigma \text{ min } \overline{a} \text{ solution in } \quad \Gamma \vdash T_E \leq T' & \overline{a}' = \overline{a} \setminus dom(\sigma)\end{array}}{\Gamma \vdash F(E) : \forall\overline{a}'.\sigma T'' \hookrightarrow \Lambda\overline{a}'.\ F'\langle\sigma\overline{a}\rangle(E'')}$$

Figure 2.6: A typing rule that assigns type to the function application term in the [Odersky, 2002, pg. 8] proposal. The rule is part of the ($\Gamma \vdash E : U \hookrightarrow E'$) typing judgment.

2.2.1 Inferred Type Instantiation for *GJ*

The semi-formal type system for Generic Java proposed in Odersky [2002] addresses the soundness problems identified in the preliminary version of the language (Jeffrey [2001]), and infers type arguments for many of the problematic scenarios of Local Type Inference that have been identified in Hosoya and Pierce [1999]. While informal and incomplete, it provides important insights into the type system and the type inference specification that is implemented in the Scala language. In this section we outline only the crucial differences between the proposed type system and the Colored Type System.

The formalization divides the type hierarchy into two categories:

Type scheme	U	$::=$	$\forall\overline{a}.T$
Non-polymorhphic type	T	$::=$	$T \to T \mid \{x_1 : T_1, \ldots, x_n : T_n\} \mid \top \mid \bot$

The *type scheme* encapsulates type variables that need to be instantiated in the context of function application, while the non-polymorphic type, as the name suggests, cannot introduce any new type variables.

The typing judgment used in the proposed formalism is of a ($\Gamma \vdash E : U \hookrightarrow E'$) form. It assigns a *type scheme* U to the term E under the environment Γ and transforms it to some other term E', if necessary.

Type assignment for function applications

Figure 2.6 recalls a typing rule `(INST-APP)`[2] that assigns a type to a function application with elided type arguments. In comparison to the Colored Type System, the type assigned to the argument of the function can be polymorphic; together with the subtyping relation between the type of the argument and the parameter of the function, it allows for propagating more type information along the adjacent nodes of the type derivation tree.

[2]The type system presented in Odersky [2002] used the name `(APP)` that conflicts with the previously described formalization.

The *type schemes* are eliminated by an instantiation judgment of a form

$$\overline{a},\ R;\Gamma \vdash \texttt{e} : U\ \ \textbf{inst}\ \texttt{e'} : T$$

The instantiation judgment states that a term `e` of *type scheme* U is instantiated to an expression `e'` of type T, given the environment Γ and the required context type R. The set of type variables $\overline{a}$, such that $\overline{a} \subseteq \texttt{fv}(R)$, represents the unknown type variables information in the expected type that can potentially be replaced by the *don't care*, wildcard types. The judgment instantiates the polymorphic type by inferring a *maximal* type substitution σ_R with respect to the type R such that $\sigma_R U <: T$.

The `(INST-APP)` rule instantiates the inferred *type scheme* of the argument to a regular, non-polymorphic type T_E in a separate type checking operation. The instantiation of the polymorphic type is crucial for inferring single best solution for the elided type arguments; a type system that allows for uninstantiated type variables on both sides of the subtyping relation does not guarantee finding principal types.
The function application rule only infers instantiations for type parameters that could be constrained with lower, non-implicit type bounds. Uninstantiated type parameters of the function application are further propagated down the type derivation tree using the $\forall\overline{a}'.T''$ *type scheme* that is assigned to the term.

The proposed type system uses a more relaxed specification of the *minimal* and *maximal* type substitution; the modification is particularly important for inferring types that are invariant in some type variable, *e.g.,* the *minimal* type substitution $\sigma_{C,R}$ would be derivable in the proposed type system for some constraint set C, such that $\{\bot <: a <: \top\} \subseteq C$ and type R, such that R is invariant in the type variable a. The changes to the specifications will be discussed in detail in Section 3.7.5.

Similarly as in the Colored Type System, the proposed type system uses type constraints from the subtyping relation to *guess* the *minimal* and the *maximal* type substitution. While the authors of Odersky [2002] do not provide the necessary changes to the constraint generation rules, based on the proposed type system and the detailed examples, we adapted the `CG` rules from the constraint generation algorithm from Section 2.1.5. A summary of the changes to the `CG` rules is provided in Figure 2.7.

The modified constraint generation judgment, (W, V $\vdash_{\overline{X}} S <: T \Rightarrow C$), includes the additional W set, that carries information about the so called *constant* type variables. The *constant* type variables differ from the *out-of-scope* type variables (represented as V in the judgment), in a sense that they refer to the type variables of the application context, and must not be *promoted* or *demoted* by the constraint generation rules. The change is highlighted in the rules `(CG-Upper)` and `(CG-Lower)`. Lack of *variable-elimination* for *constant* type variables implies that such type variables may appear in the type bounds of some constraints sets. We notice that the modest extension is still sound and complete with respect to the colorless

$$\texttt{(CG-Upper)}\ \frac{Y \in \overline{X} \quad S \Downarrow^{V \setminus W} T \quad (\mathtt{fv}(S) \setminus W) \cap \overline{X} = \emptyset}{W,\ V\ \vdash_{\overline{X}} Y <: S \Rightarrow \{\bot <:\ Y <:\ T\}}$$

$$\texttt{(CG-Lower)}\ \frac{Y \in \overline{X} \quad S \Uparrow^{V \setminus W} T \quad (\mathtt{fv}(S) \setminus W) \cap \overline{X} = \emptyset}{W,\ V\ \vdash_{\overline{X}} S <: Y \Rightarrow \{T <:\ Y <:\ \top\}}$$

$$\texttt{(CG-Var-Refl)}\ \frac{a \in \overline{X}}{W,\ V \vdash_{\overline{X}} a <: a \Rightarrow \emptyset}$$

$$\texttt{(CG-(?, <))}\ \frac{}{W,\ V \vdash_{\overline{X}} T <:\ ? \Rightarrow \emptyset} \qquad \texttt{(CG-(?, >))}\ \frac{}{W,\ V \vdash_{\overline{X}}\ ? <: T \Rightarrow \emptyset}$$

...

Figure 2.7: Extension of the $V \vdash_{\overline{a}} S <: T \Rightarrow C$ constraint generation judgment that applies to the type inference alluded to in the GJ proposal. Unmodified rules (modulo the *constant* type variables set W that is ignored in the other rules) are represented using the '...' notation. Changes to the existing rules are emphasized.

subtyping relation, thanks to the `(VAR)` subtyping rule from Figure 2.1.
For example, given a *constant* type variable b,
$(\{b\}, \emptyset \vdash_{\{a\}} a \to a <: Int \to b \Rightarrow \{Int <:\ a <:\ b\})$.

The existence of *constant* type variables implies that type variables may later appear on both sides of the subtyping relation, *e.g.*, $Int \to a\ <:\ \bot \to a$. The `(CG-Var-Refl)` rule represents a single, valid scenario, where a subtyping relation is allowed to have uninstantiated type variables on both sides of the subtyping relation. Such a subtyping relation, though allowed, carries no information on the kind of type constraints, and produces an empty constraint set, *e.g.*, $(\emptyset, \emptyset \vdash_{\{a\}} Int \to a <: \bot \to a \Rightarrow \emptyset)$. Thus, the type inference still ensures that a single best solution can be found from the constraints sets.
The two additional constraint generation rules, `(CG-(?, <))` and `(CG-(?, >))`, allow for the *don't care* ? constant type to be present in the subtyping relation. Since wildcards stand for an unknown type information they do not lead to any new type constraints for uninstantiated type variables. Furthermore the rules require that for any type S and T in $S\ <:\ T$ either $? \notin S$ or $? \notin T$.

The presented semi-formal function application rule, and the adapted constraint generation judgment, give an intuition behind the type inference technique used in the Scala language and its connection to the Colored Local Type Inference approach. This relationship allows us to base the formalization of our debugging techniques on the well-established Colored Local Type Inference, only to later extend some of its elements to the implementation of Scala's type system without compromising the soundness of our approach, or tying it to a particular local type inference implementation.

Chapter 3

Guided-analysis for type derivation trees

Type checking of any program can be thought of as a recursive application of type inference rules to individual terms of the program. Such applications essentially attempt to build complete *type derivation trees* which reveal the details of the typing decisions.

It is important to note that using type derivation trees as a basis for debugging typing decisions has been attempted in the past (*e.g.,* Tsushima and Asai [2013], Chitil [2001]) with mixed results. The main challenge remains in tracking numerous constraints that affect any kind of typing decisions, especially when applied to applications that go beyond simple toy examples. Type derivation trees constructed as part of the Local Type Inference have however interesting properties: type information is propagated only between adjacent nodes and types are approximated locally, rather than being solved globally in order to achieve a complete solution. We show that debugging such local type derivation trees, which comes with its own set of challenges, becomes a feasible solution to understanding decisions of local type inference.

In Section 3.1 we describe challenges in debugging Colored Local Type Inference that are present when having access to complete type derivation trees. Section 3.3 provides a novel abstraction that is sufficient to encapsulate type information about the types, and their evolution along the nodes of the type derivation tree. The abstraction is used for a guided navigation through the nodes of the type derivation trees (Section 3.5) without resorting to any heuristic-based techniques. The algorithm which defines such navigation is able to find nodes in the type derivation trees (or rather type checking decisions they represent), where types are introduced for the first time. In other words, for any inferred type (or part of it) we are able to locate its origin in the type derivation tree.
The results of the algorithm, the type checking decisions explaining the source of the type (Section 3.4), may not be final, in a sense that they themselves may exist as a result of some

other type checking decisions. Fortunately, we can always assign them to one of the three possible categories, and explain them separately: the type may be inferred from the expected type of the context (Section 3.6), the inferred instantiation of a type variable (Section 3.7), or an explicit type annotation (Section 3.8).

3.1 Using type derivation trees for type debugging

To illustrate the debugging challenges using type derivation trees we first compare our approach to explaining the motivating example from Chen and Erwig [2014a]. We will later consider a more interesting example from the viewpoint of local type inference, which is harder to debug using the traditional means of type constraint manipulation.

3.1.1 Debugging simple function applications

The example in Listing 3.1 defines an *identity* function, and a `boolFunc` function expecting a single argument of type `Boolean`. Similarly to the problem presented by Sheng *et al.*, the role of the polymorphic *identity* function is to add a slight twist to the problematic code and illustrate the analysis of parametric polymorphism (rather than analyzing a straightforward mismatch in a trivial `boolFunc(1)` application). The error message included in the Scala code informs the user precisely on the nature and location of the type mismatch. It also leaves the process of finding out where the types `Int` and `Boolean` originate from entirely to the user.

```
def identity[A](x: A): A = x
def boolFunc(v: Boolean): Int = if (v) 1 else 0

boolFunc(identity(0)) // error: type mismatch;
                      // found   : Int
                      // required: Boolean
                      //   boolFunc(identity(0))
                      //                     ^
```

Figure 3.1: Type mismatch with the propagated expected type of the argument.

We can encode the problematic application in the underlying language of Colored Local Type Inference in a straightforward manner, as presented in Listing 3.2 (we assume the traditional definition of `Bool` values from Simply Typed Lambda Calculus in Pierce [2002]).
By applying the inference rules of Colored Local Type Inference from Figure 2.4 to this application, we can construct an equivalent type derivation tree, as shown in Figure 3.3.

The typing error from listing in Figure 3.1 is marked in the derivation tree in Figure 3.3 as **(type mismatch)** and results from the structural inequality between a synthesized type of the

```
(fun(v: Bool) v 1 0) ((fun[a](x: a)x) 0)
```

Figure 3.2: Encoding of the Scala code from Figure 3.1 in the language of Colored Local Type Inference.

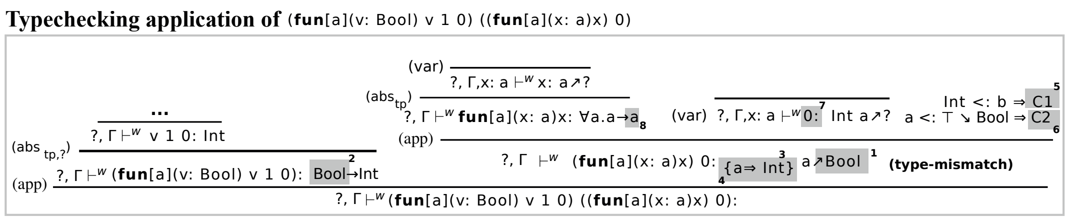

Typechecking application of (fun[a](v: Bool) v 1 0) ((fun[a](x: a)x) 0)
(var)
?, Γ,x: a ⊢w x: a↗?
(abs tp)
?, Γ ⊢w fun[a](x: a)x: ∀a.a→a 8
(var) ?, Γ,x: a ⊢w 0: 7 Int a↗?
Int <: b ⇒ C1 5
a <: ⊤ ↘ Bool ⇒ C2 6
...
?, Γ ⊢w v 1 0: Int
(abs tp,?)
(app)
?, Γ ⊢w (fun[a](x: a)x) 0: {a⇒ Int} 3 4 a↗Bool 1
(type-mismatch)
?, Γ ⊢w (fun[a](v: Bool) v 1 0): Bool→Int 2
(app)
?, Γ ⊢w (fun[a](v: Bool) v 1 0) ((fun[a](x: a)x) 0):

Figure 3.3: A fragment of the type derivation tree for the encoding from Figure 3.2. Highlighted elements identify intermediate type decisions that led to a type mismatch.

(fun$[a](x:\ a)x$) 0 application, Int, and an inherited expected type, Bool. The difference in the location of the error between the Scala and the Colored Local Type Inference versions stems from the implementation details outlined in Section 2.2 and it can be ignored for the purpose of the present example.

The encoding of the small example consist of only six inference rules, yet it gives a glimpse at the kind of complexity that users should be expected to deal with when using type derivation trees. The term (fun$[a](v:\ Bool)\ v$ 1 0) ((fun$[a](x:\ a)x$) 0), for which the judgment fails to infer the correct type, becomes the *root* of the tree. We call the branch of the type derivation tree from the *root* to the **(type mismatch)** the *failed path*.

In order to discover how the inherited type Bool (superscript 1) has been first introduced in the type derivation tree, we have to understand from which direction its information has been propagated. From the presented type derivation tree, we can see that it flows from the parameter part of the anonymous function (superscript 2) towards the argument of the function. Moreover, there is a connection between the part of the inferred type of the anonymous function and the type annotation for the parameter v in fun(v : Bool) v 1 0. The informal description of the link between the types can be traced back to the way the types are propagated in the type inference rules.

The user might also want to find out the source of the synthesized Int type (superscript 3). From the formalization in Section 2.1 we know that type variable substitution (superscript 4), σ, always has its source in the collected constraints (superscripts 5 and 6), $C1$ and $C2$. It quickly becomes a non-trivial task to understand what are inferred constraint sets and how bounds of constraint $C1$ have its source in the inferred type of the constant 0 (superscript 7), while the type bounds of constraint $C2$ come from the result type of the inferred type of the identity function (superscript 8).

With this simple type derivation tree, it becomes apparent that using type derivation trees for debugging purposes is powerful but should ignore some premises of inference rules that are irrelevant to the nature of the problem that we try to understand. Additionally, the direction of the analysis clearly depends on whether we investigate the source of the inherited or the synthesized type.

In the next example we give a more in-depth intuition on how we can apply that approach to a more realistic scenario, where non-trivial polymorphic function types typically make the analysis hard to follow.

3.1.2 Debugging `foldRight` application

The introduction has provided an example of a confusing error for an application involving the `foldRight` function (Figure 1.1). By encoding the example in the core language, and explaining the decisions of the type inference process, we show how limitations of local type inference leak to type error messages without offering any obvious type debugging techniques or workarounds to correct it.

We define the problematic snippet using the straightforward encoding of lists (summarized for reference in Appendix A). We assume that the type of the `foldRight` term has been inferred as $\forall a.\ \textsc{List}[a] \to (\forall b.\ \textsc{List}[b] \to ((a, b) \to b) \to b)$, where $List$ denotes a type constructor of list collection, and a `1` constant is of a base type Int, where $Int <: \top$. Therefore the erroneous `foldRight` application can be encoded as:

$$\texttt{foldRight(Cons(1,Nil())(Nil())}((x, y) \to \texttt{Cons}(x{+}1, y))$$

An application of the Colored Local Type Inference rules (defined in Figure 2.4) to this application leads to a type derivation tree shown in Figure 3.4.

The type mismatch is marked in the derivation tree in Figure 3.4 as **(type mismatch)** and results from the structural inequality between the assigned type of the body of the function, $\textsc{List}[Int]$, and an expected type, $\textsc{List}[\bot]$[1].

Debugging typing decisions through a simple visual assessment of the type derivation tree was still a viable option for the function application example from Section 3.1.1, but it is no

[1] In Scala, type $\bot$ is equivalent to type `Nothing` but has no Java correspondence. The top type $\top$ is equivalent to types `Any` and `Object` in Scala and Java, respectively.

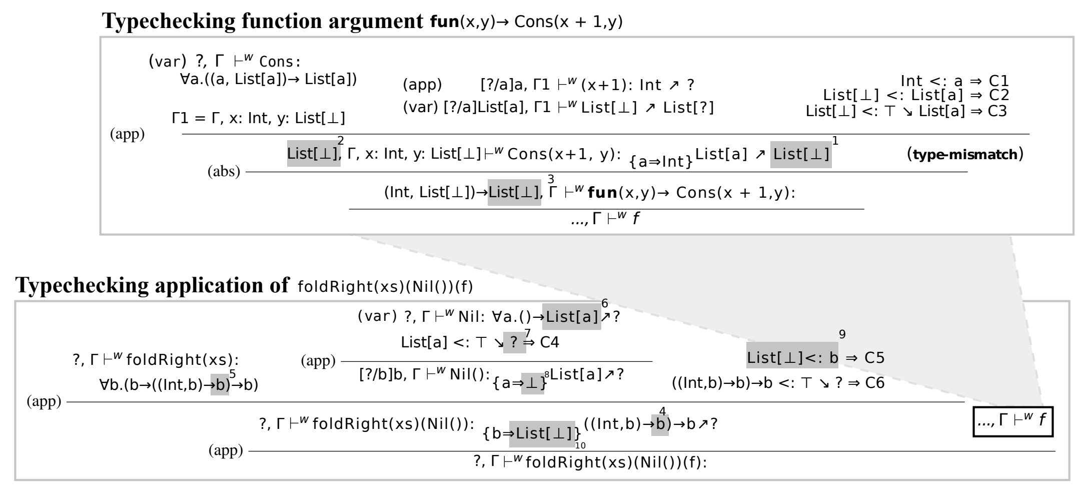

Typechecking function argument fun(x,y)→ Cons(x + 1,y)
(var) ?, Γ ⊢ʷ Cons: ∀a.((a, List[a])→ List[a])
Γ1 = Γ, x: Int, y: List[⊥]
(app) [?/a]a, Γ1 ⊢ʷ (x+1): Int ↗ ?
(var) [?/a]List[a], Γ1 ⊢ʷ List[⊥] ↗ List[?]
Int <: a ⇒ C1
List[⊥] <: List[a] ⇒ C2
List[⊥] <: ⊤ ↘ List[a] ⇒ C3
(app)
List[⊥], Γ, x: Int, y: List[⊥]⊢ʷ Cons(x+1, y): {a⇒Int}List[a] ↗ List[⊥]
(type-mismatch)
(abs)
(Int, List[⊥])→List[⊥], Γ ⊢ʷ fun(x,y)→ Cons(x + 1,y):
..., Γ ⊢ʷ f
Typechecking application of foldRight(xs)(Nil())(f)
(var) ?, Γ ⊢ʷ Nil: ∀a.()→List[a]↗?
List[a] <: ⊤ ↘ ? ⇒ C4
?, Γ ⊢ʷ foldRight(xs): ∀b.(b→((Int,b)→b)→b)
(app)
[?/b]b, Γ ⊢ʷ Nil(): {a⇒⊥}List[a]↗?
List[⊥]<: b ⇒ C5
((Int,b)→b)→b <: ⊤ ↘ ? ⇒ C6
(app)
?, Γ ⊢ʷ foldRight(xs)(Nil()): {b⇒List[⊥]}((Int,b)→b)→b↗?
..., Γ ⊢ʷ f
(app)
?, Γ ⊢ʷ foldRight(xs)(Nil())(f):

Figure 3.4: Fragment of a type derivation tree for the application `foldRight`(xs)(`Nil`())(f), where xs = `Cons(1,Nil())` and f = `fun(x,y)` → `Cons(x + 1,y)`. Superscripts identify intermediate type decisions that led to a type mismatch.

longer the case for the `foldRight` application. Nevertheless, the systematic type propagation, primarily used to elide more type annotations, also offers a means to backtrack through the type checking process as long as we are able to meticulously follow the decisions of the inference rules.

For example, in Figure 3.4 we notice that the conflicting expected type is being consistently propagated on the *failed path* in locations 1, 2 and 3. Furthermore, by looking only at the position of the conflicting element in the propagated type information, we see that it is first introduced in the type derivation tree during type variable substitution (superscript 4).
The substituted type variable b is first introduced in the inferred type of the partially applied function `foldRight`(xs). Since b is an abstract type variable, there is no need to further continue the analysis of LIST`[⊥]` in the left branch of the type derivation tree, ($? ,\Gamma \vdash^{w}$ `foldRight`$(xs) : \forall b.\ b \rightarrow ($ `Int`$, b\ ) \rightarrow b \rightarrow b$); we have found the location where the conflicting expected type LIST`[⊥]` is first introduced. The informal description relied only on the information on how elements of types flow from one type inference rule to the other, irrespective of actual instances of types.

A visually easier to interpret Figure 3.5 presents a stripped down version of the previous derivation tree, which only shows how elements of the inferred type of the partially applied function `foldRight`(xs)(`Nil`()) are systematically propagated to infer the type of the anonymous function. It becomes apparent that the *root* of the tree becomes essentially a point where we shift from backtracking on the type derivation tree to actively using the collected type information to navigate to a different part of the type derivation tree. We call this point a *Propagation Root*.

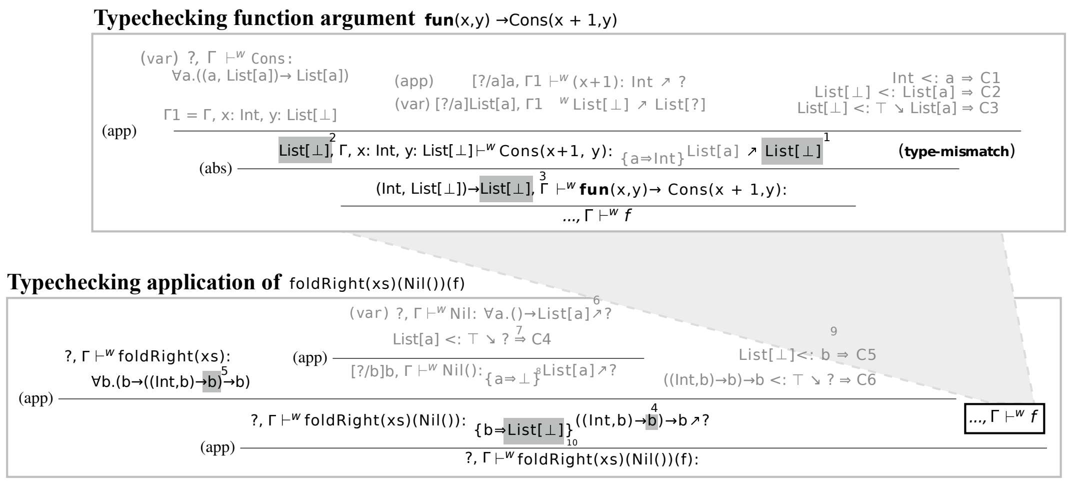

Typechecking function argument fun(x,y) →Cons(x + 1,y)
(var) ?, Γ ⊢^w Cons: ∀a.((a, List[a])→ List[a])
Γ1 = Γ, x: Int, y: List[⊥]
(app) [?/a]a, Γ1 ⊢^w (x+1): Int ↗ ?
(var) [?/a]List[a], Γ1 ⊢^w List[⊥] ↗ List[?]
Int <: a ⇒ C1
List[⊥] <: List[a] ⇒ C2
List[⊥] <: ⊤ ↘ List[a] ⇒ C3
(app)
List[⊥], Γ, x: Int, y: List[⊥]⊢^w Cons(x+1, y): {a⇒Int} List[a] ↗ List[⊥]
(type-mismatch)
(abs)
(Int, List[⊥])→List[⊥], Γ ⊢^w fun(x,y)→ Cons(x + 1,y):
..., Γ ⊢^w f
Typechecking application of foldRight(xs)(Nil())(f)
(var) ?, Γ ⊢^w Nil: ∀a.()→List[a]↗?
List[a] <: ⊤ ↘ ? ⇒ C4
?, Γ ⊢^w foldRight(xs): ∀b.(b→((Int,b)→b)→b)
(app)
[?/b]b, Γ ⊢^w Nil(): {a⇒⊥} List[a]↗?
List[⊥]<: b ⇒ C5
((Int,b)→b)→b <: ⊤ ↘ ? ⇒ C6
(app)
?, Γ ⊢^w foldRight(xs)(Nil()): {b⇒List[⊥]} ((Int,b)→b)→b↗?
..., Γ ⊢^w f
(app)
?, Γ ⊢^w foldRight(xs)(Nil())(f):

Figure 3.5: A simplified fragment of the type derivation from Figure 3.4 where the grayed-out elements do not explain the source of the expected type LIST[⊥].

It is important to note, that the *Propagation Root* is likely to be different from the actual *root* of the type derivation tree since the problematic fragment will typically be part of a bigger type derivation tree in the complete program.

Before we continue our informal type derivation tree exploration, in search for the source of the LIST[⊥] type, we first formally define the concept of the *Propagation Root* in order to better understand its significance.

Propagation Root for the expected types

Definition 3 characterizes the *parent* of the type inference judgment, used later in the chapter and in the definition of the *Propagation Root.*

Definition 3 *Parent of the type inference judgment.*
The *parent* of the inference judgment $(P, \Gamma \vdash^w E : T)$ in a type derivation tree is the judgment which has $(P, \Gamma \vdash^w E : T)$ listed as one of its premises.
The $((P, \Gamma \vdash^w E : T) \downarrow)$ notation defines a function that returns the parent of the $(P, \Gamma \vdash^w E : T)$ inference judgment. The result of the function is undefined for the *root* of the type derivation tree.

The existence of the *Propagation Root* can now be defined formally in Theorem 1.

Theorem 1 *Propagation of prototype information and Propagation Roots.*
We consider any type inference judgment of a form $(P, \Gamma \vdash^{w} E : T)$, where T might or might not have been inferred, and $P \neq\ ?$, and let the *parent* of $(P, \Gamma \vdash^{w} E : T)$ be represented as $(P^{p}, \Gamma^{p} \vdash^{w} E^{p} : T^{p})$.
The prototype P in $(P, \Gamma \vdash^{w} E : T)$ has been either propagated from its parent, *i.e.,* P is part of the prototype P^{p}, or it has been introduced for the first time in its parent and P is not part of P^{p}. If the prototype P is introduced for the first time by the *parent* of the type inference judgment, we call such *parent* the *Propagation Root* for prototype P.

Proof.
A proof by induction on the type inference rule for the *parent* of the type inference judgment. A complete proof is provided in Appendix C. □

Lemma 3.1 allows us to state that we can always find the type inference judgment that introduces the prototype for the first time, through simple backtracking through the nodes of the type derivation tree. The subject will be formally explored in more detail later for both error-free (Section 3.6) and erroneous (Section 4.1) type derivation trees.

Lemma 3.1 *Existence of the Propagation Roots for type derivation trees.*
For any inference judgment of the form $(P, \Gamma \vdash^{w} E : T)$, where T might or might not have been inferred and $P \neq\ ?$, we can always find the inference judgment $(P', \Gamma' \vdash^{w} E' : T')$, abbreviated as $\vdash^{w}_{P'}$, where T' might or might not have been inferred, such that $(P, \Gamma \vdash^{w} E : T)$ is a subtree that is part of the $\vdash^{w}_{P'}$ inference judgment, and partial type information P is first being propagated in $\vdash^{w}_{P'}$ and it is not part of $\vdash^{w}_{P'}$'s own prototype. We call the path from the $(P, \Gamma \vdash^{w} E : T)$ inference judgment down the type derivation tree to $\vdash^{w}_{P'}$, the *Prototype Propagation Path.*

Proof.
The proof of Lemma 1 follows directly from Theorem 1, the formulation of the *parent* of the type inference judgment in Definition 3, and the requirement on the *root* of the type derivation tree on having the ? prototype, *i.e.,* a program has to start with no expected type. □

Debugging type variable instantiations

Our informal description of the `foldRight` application has identified the type variable b and its instantiation to type $List[\bot]$ in the `foldRight`(xs)`(Nil())` partial application as the first node in the type derivation tree where the conflicting expected type has been fully stated. This is an important step in the analysis of our motivating example, yet not the final one. The precise debugging technique would tell which of the type constraints (superscript 9 in Figure 3.4) were **used** in the instantiation of the type variable b, and how do they relate to the inferred type of the argument `Nil()`, $\textsc{List}[\bot]$.

If we were only interested in the source of the inherited expected type, then reporting the argument `Nil()`, as the source of type constraints would be correct. However, rather than treating the (**type mismatch**) failure as a black box, we can gain much more information from taking into account the failed subtyping derivation tree. Since we assume that $List$s in our encoding are covariant, we know that the actual mismatch originates from its type arguments:

$$\frac{Int \not<: \bot}{\textsc{List}[Int] \not<: \textsc{List}[\bot]}$$

The premise of such failed subtyping derivation tree, provides further information on how to narrow down the analysis of the subtree that instantiates the type variable b - we can focus on tracking the origin of type element $\bot$ in from the $\textsc{List}[\bot]$ type rather than $\textsc{List}[\bot]$. This in turn implies that the analysis of the type derivation trees has to understand how the type parameter a (superscript 7) from type $forall a.\ () \rightarrow \textsc{List}[a]$ was instantiated to type $\bot$ (superscript 8) in the application `Nil()`. Careful look at the inference rule `(app)` in $([{}^{?}/_{b}]\, b,\ \Gamma \vdash^{w} Nil() : \textsc{List}[\bot])$ reveals that even though constraints are collected for the type variable a (superscript 7), none of them are relevant for the inference of the most optimal type, and in consequence, the $\bot$ type, the true source of the type error is inferred. That is the level of detail, in terms of finding minimal explanations of types, our analysis aims to achieve.

Similarly as before, our informal explanation has only relied on how type inference rules infer the instantiations of the type parameters and propagate the elements of types, rather than being specific to the `foldRight` application. We exploit this insight in two ways:

- We are able to locate the smallest fragment of the type derivation that encapsulates the node where the error is reported to the user and all the nodes that either propagate or introduce the conflicting type for the first time.

- By identifying the nodes of the type derivation that introduce types for the first time, we can suggest opportunistic source code fixes at program locations associated with such nodes.
  For example, a naive, non-minimal analysis could suggest to the user to annotate the complete argument in the application with an explicit type annotation, *e.g.,* `Nil() :`

$List[Int]$[2]. We, on the other hand, can *improve* the instantiation of the type variable a in function application `Nil()` and suggest an explicit type argument, *e.g.,* `Nil`$[Int]$(). We will come back to the subject of precise heuristics when discussing our type debugging implementation in Section 6.7.

A focus during the informal analysis only on particular elements of prototypes and types allowed us to ignore other non-trivial typing decisions in the type derivation tree. The type selection, presented in the form of grayed-out boxes, is completely oblivious to the specifics of the problem and relies solely on the information of how partial type information flows between the type inference rules.

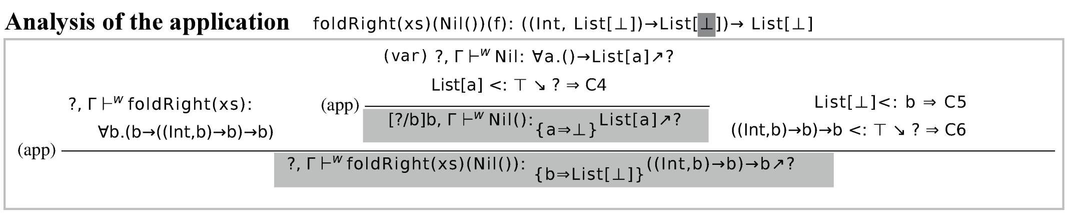
The next section provides an overview of the components of the algorithm that analyzes the decisions of a generic type derivation tree based on only on the applied type inference rules and the type selection information. Later we give a formal definition to the abstraction behind the grayed-out boxes in the type derivation trees, and how it can be built in an incremental manner when navigating type derivation trees. Importantly, from the formal point of view the type selection maintains a focus only on the *semantically* identical elements of types.

3.2 Introduction to the analysis of type derivation trees

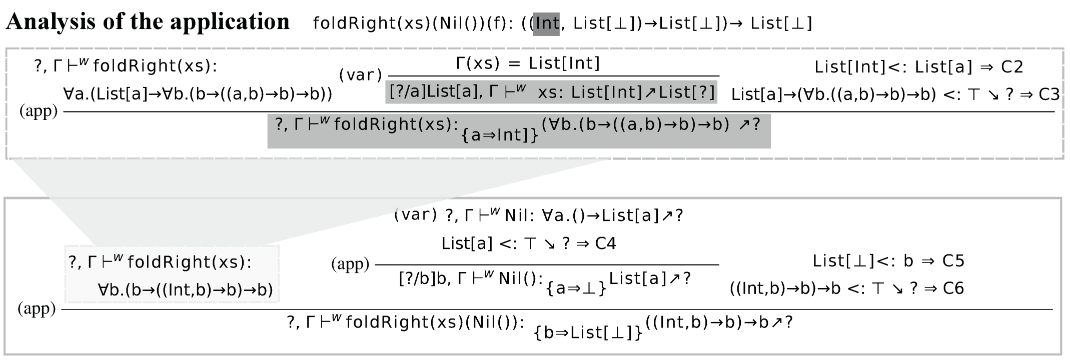
The aim of our thesis is to be able to explain the typing decisions of the erroneous and error-free programs, or their fragments, using the same mechanism. The algorithm defined in this chapter only applies to the error-free type derivation trees. As we will describe in detail in Chapter 4, the analysis of the errors encountered in failed type derivation trees reduces to locating the correct, error-free subtrees of the derivations and finding the correct input values for our algorithm based on the shape of the path that led to the type inconsistencies.

In high-level terms the algorithm analyzing the typing decisions of Colored Local Type Inference takes as input any error-free type derivation tree, or rather a root node of such derivation, and a type selection information that extracts part of the inferred type of the term that resulted from such derivation. The purpose of the algorithm is to find the typing decision that introduced the selected part of the inferred type, known in our notation as the *target type*, for the first time in the provided type derivation tree. The reason for considering the individual type elements that make the inferred type is that it allows us to potentially extract only small fragments of the tree, visually interpreted as paths, that affected them; the complete inferred type clearly results from the complete type derivation tree. Therefore the desired output of the algorithm are the nodes of the type derivation tree which represent the source of the target type. In reality, the result is somewhat more involving and requires further discussion.

[2]An explicit type annotation $t : T$ is just a syntactic sugar for $(\texttt{fun}(x : T)x)\ t$ application in the core language of Colored Local Type Inference


Analysis of the application foldRight(xs)(Nil())(f): ((Int, List[⊥])→List[⊥])→ List[⊥]
(var) ?, Γ ⊢ʷ Nil: ∀a.()→List[a]↗?
List[a] <: ⊤ ↘ ? ⇒ C4
?, Γ ⊢ʷ foldRight(xs): ∀b.(b→((Int,b)→b)→b)
(app)
[?/b]b, Γ ⊢ʷ Nil(): {a⇒⊥} List[a]↗?
List[⊥]<: b ⇒ C5
((Int,b)→b)→b <: ⊤ ↘ ? ⇒ C6
(app)
?, Γ ⊢ʷ foldRight(xs)(Nil()): {b⇒List[⊥]} ((Int,b)→b)→b↗?

(a) Source of type ⊥


Analysis of the application foldRight(xs)(Nil())(f): ((Int, List[⊥])→List[⊥])→ List[⊥]
?, Γ ⊢ʷ foldRight(xs): ∀a.(List[a]→∀b.(b→((a,b)→b)→b))
(var)
Γ(xs) = List[Int]
[?/a]List[a], Γ ⊢ʷ xs: List[Int]↗List[?]
List[Int]<: List[a] ⇒ C2
List[a]→(∀b.((a,b)→b)→b) <: ⊤ ↘ ? ⇒ C3
(app)
?, Γ ⊢ʷ foldRight(xs): {a⇒Int]} (∀b.(b→((a,b)→b)→b) ↗?
(var) ?, Γ ⊢ʷ Nil: ∀a.()→List[a]↗?
List[a] <: ⊤ ↘ ? ⇒ C4
?, Γ ⊢ʷ foldRight(xs): ∀b.(b→((Int,b)→b)→b)
(app)
[?/b]b, Γ ⊢ʷ Nil(): {a⇒⊥} List[a]↗?
List[⊥]<: b ⇒ C5
((Int,b)→b)→b <: ⊤ ↘ ? ⇒ C6
(app)
?, Γ ⊢ʷ foldRight(xs)(Nil()): {b⇒List[⊥]} ((Int,b)→b)→b↗?

(b) Source of type Int

Figure 3.6: The simplified result of the analysis of the judgment (?, $\Gamma \vdash^w$ `foldRight`(xs)(`Nil`()) : $((Int, \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]$) for two different parts of its inferred type (highlighted). The grayed-out inference judgments represent the complete *path* of the type derivation tree explaining the source of the target type.

For example, the visual interpretation of the input of the algorithm is presented in Figure 3.6. There we provide a familiar partial application of the `foldRight` function. For the input judgment (?, $\Gamma \vdash^w$ `foldRight`(xs)(`Nil`()) : $((Int, \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]$) and the type selection on the inferred type $((Int, \text{LIST}[\bot]) \rightarrow \text{LIST}[\,\bot\,]) \rightarrow \text{LIST}[\bot]$ (Figure 3.6a) the algorithm would return two grayed-out nodes of the type derivation tree that inferred the type of terms `foldRight`(xs)(`Nil`()) and `Nil`(). On the other hand for the same inference judgment but a different type selection on the inferred type, $((\,Int\,, \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]) \rightarrow \text{LIST}[\bot]$ (Figure 3.6b), the algorithm would traverse the type derivation tree different and return different set of grayed-out nodes of the type derivation tree.

The design of the algorithm that looks for the source of types in the derivation trees of Colored Local Type Inference has to deal with a number of additional challenges:

1. The algorithm has to explore only those adjacent nodes of the tree that directly contributed to the inference of the selected type element of the inferred type. This allows us to avoid the explosion of the search space.

2. Such exploration translates naturally to constructing complete paths from the root of the type derivation tree to the nodes that introduce the target type, as illustrated in

Figure 3.6. In reality it should be sufficient to return the minimal number of nodes corresponding only to the final points of the exploration paths.

3. The programmers trying to understand the type checking of some source code do not want to deal with inference judgments and type derivation trees in general. We notice however that the nodes of type derivation trees assign types to individual terms. Thus it becomes possible to explain the source of the selected type elements by means of the program locations of the located terms.

4. Each of the nodes encountered during the exploration of the tree may involve a number of different kinds of typing decisions that may or may not somehow affect the value of the target type. For example the type elements of the inferred types may be synthesized from the term, inherited from the type checking context, or both.
   This in turn means that type inference rule level of granularity is insufficient. Reporting only the individual nodes, or the complete exploration paths, of the provided type derivation tree would be insufficient to explain the source of the target type. For example the highlighted nodes in Figure 3.6 do not express the important role and origin of type variable substitutions $\{a \Rightarrow \bot\}$ and $\{b \Rightarrow \text{LIST}[\bot]\}$.

The number of different combinations of type checking decisions that may take place within a single type derivation node motivated a different design and output of the algorithm. Rather than defining the algorithm that attempts to explain the source of the target type by finding in one exploration pass the minimal number of nodes of the type derivation tree, our core algorithm finds only the first node of the provided type derivation tree, or rather its subtree, that introduced the selected type element. This means that the result of the core algorithm does not necessarily return the minimal path. For the returned node we also capture the kind of the typing decision that led to the inference of the type element, as well as the type selection information that extracts the type element from the inferred type of the node. The refined output of the analysis is described in detail in Section 3.4.

For example, for the inference judgment and the type selection from Figure 3.6a, we would return:

- the same node,
- the same type selection, and
- the information that the target type was inferred as part of the type variable instantiation that takes place in the returned inference judgment.

On the other hand for the same inference judgment but a different type selection from Figure 3.6b we would return

- the inference judgment that assigned type to the `foldRight`(xs) term,

- the type selection from the inferred type $\forall b.\ b \rightarrow (((\boxed{\texttt{Int}}, b) \rightarrow b) \rightarrow b)$, and
- the information that the selected target type was inferred as part of the type variable instantiation that takes place in the returned inference judgment.

The returned nodes of the type derivation tree are grouped into 4 categories, based on the kind of the typing decision that inferred the target type. For each of the categories we define a set of specialized analysis functions that allow us to further explore the nodes and their typing decisions, if necessary. Such exploration is also known as the *expansion* of the nodes in our terminology.

Depending on the kind of the typing decision and the kind of the term, the individual analysis function may further invoke the core algorithm, traverse towards the root of the type derivation tree, starting with the node returned by the core algorithm, or both. In that sense, we can perceive the result of the core algorithm, the triple describing the node, as a continuation. The continuation can be further explored, if necessary, in order to filter the relevant nodes of the type derivation that led to the inference of the target type.

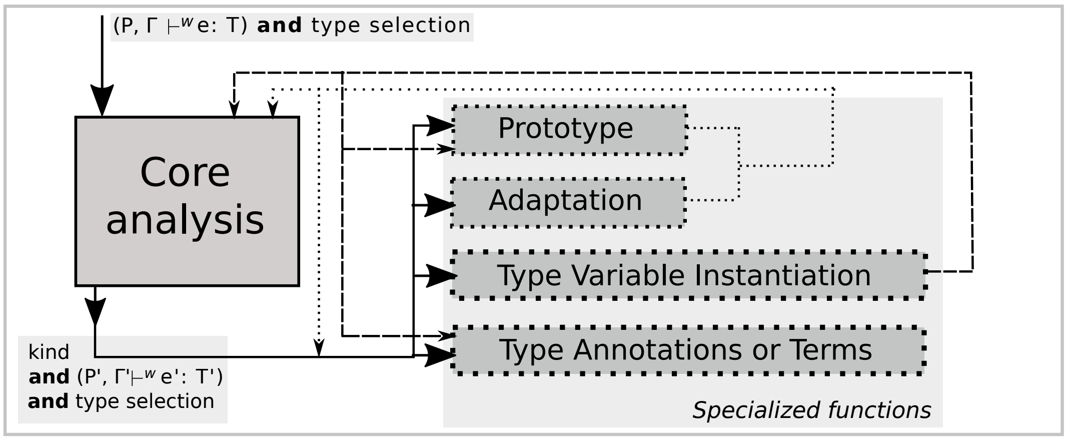

(P, Γ ⊢ʷ e: T) **and** type selection
Core analysis
Prototype
Adaptation
Type Variable Instantiation
Type Annotations or Terms
kind
and (P', Γ'⊢ʷ e': T')
and type selection
Specialized functions

Figure 3.7: An overview of the input and output values of the algorithm, and its components, that analyzes the decisions of the type derivation trees in order to explain the source of the selected part of the inferred type T in the $P, \Gamma \vdash^{w} e : T$ typing judgments.

Figure 3.7 illustrates the main components outlined in the above description. The generic core algorithm (explained in Sections 3.5.1 and 3.5.2) takes any type inference judgment and navigates the type derivation tree in order to find the node that introduced the selected part of the inferred type, T, for the first time. For example, the nodes that only propagate the type information of the target type, without modifying or inferring its type elements, will be stepped through.

As we describe in Section 3.4 the target type can be either enforced by the type inherited from the type checking context (the analysis of the *Prototype* component is described in Section

3.6), synthesized as part of the $\nearrow$ adaptation process (the *Adaptation* component is reusing the analysis of the *Prototype* component), result from the non-trivial type variable instantiation (the analysis of the *Type Variable Instantiation* component is defined in Section 3.7), or be synthesized directly from the explicit type annotation or the term (the analysis of the *Type Annotations or Terms* component is defined in Section 3.8). The outgoing arrows of the four components indicate that the specialized analysis functions either directly explain the typing decisions through other nodes of the type derivation tree by returning the same kind of the output as the core algorithm, or delegate to the core analysis algorithm. As we will show in Section 3.8, nodes that infer the target type from the type annotations or terms are considered to be minimal and will not be further expanded.

The core algorithm is generic, in a sense that it accepts any error-free type inference judgment. The algorithm uses the provided type selection to specialize the analysis and identify the source of the target type. At the same time, we notice that its output also includes the type selection, allowing for the continuation of the analysis without losing track of the target type in the specialized functions. For example, this way the specialized function in the *Type Variable Instantiation* component will not analyze the source of the complete type variable substitution but only of the selected type element of it.

The type selection, formally defined in the next section, serves as a *glue* that binds together the analysis between the individual components and the core algorithm; all specialized functions take as input a node of the type derivation tree **and** the type selection that extracts the desired part of the inferred type.

3.3 Foundations of *TypeFocus*

In this section we formally introduce a *TypeFocus*, an abstraction that drove our informal exploration of type derivation trees. In short, the *TypeFocus* is a *type selection* on type expressions. Visually, we can perceive any type expression as a tree structure, where the internal vertices always represent the type constructors. In the case of type applications, the children of the internal vertices, connected through undirected edges, represent the type arguments. The *TypeFocus* can be interpreted as a *path* from the root of such type expression to some (internal or external) node of the type expression. The length of the path is equivalent to the number of edges of the tree it selects.

For example, Figure 3.8 illustrates four possible type selections on the inferred type of the partially function `foldRight(`xs`)(Nil())` from Section 3.1.2.

The type selection is based on the possible shapes of types rather than particular type instances. Therefore such type selection is allowed on types and prototypes, both of which affect how types are inferred in local type inference.
Terms *type selection*, *type extraction*, or simply *type focus* on types are used interchangeably

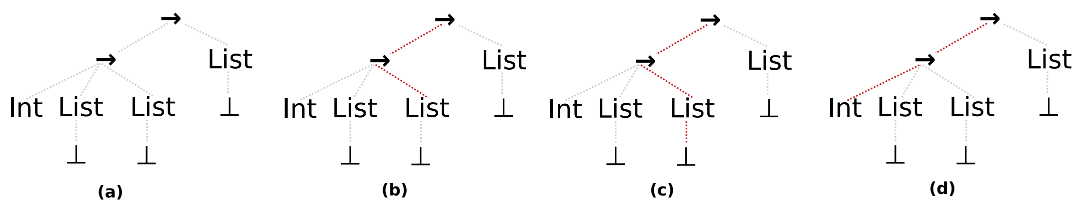


Figure 3.8: Examples of different type selections on the inferred type of the $\texttt{foldRight}(xs)(\texttt{Nil}())$ function application from Figure 3.4. The darker edges provide the visual interpretation of the type selection from the inferred type in (a) $((\texttt{Int}, \textsc{List}[\bot]) \to \textsc{List}[\bot]) \to \textsc{List}[\bot]$, (b) $((\texttt{Int}, \textsc{List}[\bot]) \to \textsc{List}[\bot]) \to \textsc{List}[\bot]$, (c) $((\texttt{Int}, \textsc{List}[\bot]) \to \textsc{List}[\bot]) \to \textsc{List}[\bot]$, and (d) $((\texttt{Int}, \textsc{List}[\bot]) \to \textsc{List}[\bot]) \to \textsc{List}[\bot]$.

and refer to the same process of returning a part of the given type expression.

Later in the section, we provide a formal definition of the *TypeFocus* abstraction and illustrate its usage on the types inferred as part of the non-trivial typing judgments.

3.3.1 A *type selection* on type expressions

With the visual interpretation of the *TypeFocus* abstraction, it becomes clear that the *TypeFocus* path is a list of individual type selectors, or edge selectors in the type expression tree. We distinguish 3 different kinds of *TypeFocus* selectors, ϕ, for our core language; one for each of the types and their type elements. The *TypeFocus*, denoted as Θ, is defined recursively as an empty path, or a concatenation of some *TypeFocus* selector and *TypeFocus*:

$$\begin{array}{lll} \phi & ::= & \phi_{\texttt{fun-param}} \mid \phi_{\texttt{fun-res}} \mid \phi_{\texttt{sel}_x} \qquad \text{(type selector)} \\ \Theta & ::= & [\,] \mid \phi :: \Theta \qquad\qquad\qquad\qquad (\textit{TypeFocus}) \end{array}$$

Similarly to list notation, the $\phi_1 :: \phi_2 :: [\,]$ *TypeFocus* is equivalent to the shorter $[\phi_1, \phi_2]$ notation, for any ϕ_1 and ϕ_2.

When applied to any type the *TypeFocus*, Θ, is treated as a function of type

$$\Theta : \; T \to (T + (T \times \Theta)) \qquad \text{(type extraction with } \textit{TypeFocus}\text{)}$$

We adopt the definition of sum and product types that is present in Pierce [2002]. The *TypeFocus* takes a type and returns a value of a sum type. The left (tagged) value of the sum type contains the selected part of the input type, while the right (tagged) value is a product type of type and *TypeFocus*. Before we explain the reasons behind the unusual return type, we will first define complete semantics for the application of *TypeFocus* to types in Definition 4.

Definition 4 *Semantics of TypeFocus* Θ *of type* $T \to (T + (T \times \Theta))$ *for the core language.*

$$([\,])(T) = \texttt{inl}\ T$$

$$(\phi_{\texttt{fun-param}} :: \Theta')(T) = \begin{cases} \Theta'(S) & \textbf{if} \quad T = \forall \overline{a}.S \to R \\ \texttt{inr}\ \langle T, \phi_{\texttt{fun-param}} :: \Theta' \rangle & \textbf{else} \end{cases}$$

$$(\phi_{\texttt{fun-res}} :: \Theta')(T) = \begin{cases} \Theta'(R) & \textbf{if} \quad T = \forall \overline{a}.S \to R \\ \texttt{inr}\ \langle T, \phi_{\texttt{fun-res}} :: \Theta' \rangle & \textbf{else} \end{cases}$$

$$(\phi_{\texttt{sel}_{x_i}} :: \Theta')(T) = \begin{cases} \Theta'(T_i) & \textbf{if} \quad T = \{x_1 : T_1, \ldots, x_n : T_n\} \textbf{ and } 1 \le i \le n \\ \texttt{inr}\ \langle T, \phi_{\texttt{fun-res}} :: \Theta' \rangle & \textbf{else} \end{cases}$$

The individual type selector $\phi_{\texttt{fun-param}}$ extracts the parameter type of a function type, $\phi_{\texttt{fun-res}}$ extracts the result type of a function type, and $\phi_{\texttt{sel}_{x_i}}$ extracts the type of a member x_i of a record type.

Intuitively, the semantics of *TypeFocus* demand that if the type selection path is empty we return the identical type in a left tagged value. Since individual *TypeFocus* selectors, ϕ, are the elements of the path, the type selection attempts to apply them in sequence from left to right; if the head of the *TypeFocus* successfully selected the type, the resulting type is applied recursively to the *tail* of the type selection path until an empty *TypeFocus* is encountered or the internal structure of the type is different from the *TypeFocus* selector expectation. If an internal structure of the type is different, the application of *TypeFocus* returns the right tagged tuple consisting of the input type, and the *TypeFocus* instance.

For example,

- $[\phi_{\texttt{fun-res}}](A \to (B \to C)) = \texttt{inl}\ (B \to C)$, the *TypeFocus* extracts a part of the input type $A \to (B \to C)$.
- $[\phi_{\texttt{sel}_z}](A \to (B \to C)) = \texttt{inr}\ \langle A \to (B \to C), [\phi_{\texttt{sel}_z}] \rangle$, the function type does not match the expected record type of the *TypeFocus*, and the application returns the *failed* right tagged tuple.

Similarly,

- $[\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}](A \to (B \to C)) = \texttt{inl}\ C$, the *TypeFocus* extracts a part of the nested function type.

- $[\phi_{\texttt{fun-param}}, \phi_{\texttt{sel}_\texttt{x}}, \phi_{\texttt{fun-res}}](A \to (B \to C)) = \texttt{inr}\ \langle A, [\phi_{\texttt{sel}_\texttt{x}}, \phi_{\texttt{fun-res}}]\rangle$, the extraction could only perform a *partial* type selection on the provided type.

Typically, type selection can be envisioned as a straightforward extractor function of type $(T \to T)$. Our definition of *TypeFocus* allows us to deal gracefully with real examples when type selection fails to extract part of a type. Such failure happens when the input type does not conform to the type expected by the instance of *TypeFocus*. In the case of a type selection failure, the value of the product type consists of the part of the type on which selection failed, and of *TypeFocus* instance that could not perform the type selection on the remaining part of the type.

Under certain circumstances a *failed* type selection is not a sign of unsoundness, but in fact may be desired and reveal important typing properties. For example, our formulation of *TypeFocus* will allow us also to apply the same type selection to the type variables that have been instantiated or not. Let's consider the *TypeFocus* Θ', where $\Theta' = [\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}, \phi_{\texttt{sel}_\texttt{x}}]$, and its application to some function type with instantiated type variable $\Theta'_{(\{b \Rightarrow \{x\colon B\}\}}(b \to (Int \to b))) = \texttt{inl}\ B$. The type extraction from the given type succeeds by returning the left tagged value. The application of the same *TypeFocus* to the same type, but with the abstract type variables, only *partially* selects part of the function type in $\Theta'(\forall \overline{b}.(b \to (Int \to b))) = \texttt{inr}\ \langle b, [\phi_{\texttt{sel}_\texttt{x}}]\rangle$. In both cases we want to be able to apply the same type selection to the same type, modulo the type variable substitution.

Two *TypeFocus* values can be composed together using the ':::' notation, *i.e.,* $\Theta^{'}\ :::\ \Theta^{''}$, for any $\Theta^{'}$ and $\Theta^{''}$. The semantics of the composition intuitively define a concatenation of two *TypeFocus* paths, which ensures that Θ'' is applied to the type that was first extracted in the Θ' application if and only if the latter selection was successful, *i.e.,* if the application of Θ' returned a left tagged value. Otherwise, it returns a *failed* right tagged tuple; the tuple consists of the part of the type that was extracted until a shape mismatch occurred in Θ' selection, and a composition of Θ'' and the *failed* Θ' instance. Similarly as in the list notation, the $[\phi_1, \phi_2]\ :::\ [\phi_3, \phi_4]$ concatenation is equivalent to the $[\phi_1, \phi_2, \phi_3, \phi_4]$ notation for any ϕ_1, ϕ_2, ϕ_3, ϕ_4 type selectors.

For example,

- $([\phi_{\texttt{fun-res}}] ::: [\phi_{\texttt{fun-res}}])(A \to (B \to C)) = \texttt{inl}\ C$.
- $([\phi_{\texttt{fun-param}}, \phi_{\texttt{sel}_\texttt{x}}] ::: [\phi_{\texttt{fun-res}}])(A \to (B \to C)) = \texttt{inr}\ \langle A, [\phi_{\texttt{sel}_\texttt{x}}, \phi_{\texttt{fun-res}}]\rangle$, and the result represents only a *partial* type selection.

For convenience, Definition 5 defines an auxiliary function $\Theta_{\texttt{tpe}}$ of type $(T + T \times \Theta) \to T$ that takes the result of the application of *TypeFocus* and returns its type component, irrespective of whether it returned a left or right tagged value.

Definition 5 *Extracting type selection from TypeFocus application.*

$$\Theta_{\texttt{tpe}}(v) \;=\; \textbf{case}\ v\ \textbf{of} \left|\begin{array}{lcl} \texttt{inl}\ T & \Rightarrow & T \\ \texttt{inr}\ \langle T, \Theta' \rangle & \Rightarrow & T \end{array}\right.$$

To avoid ambiguous terms, throughout the rest of the work we use the following terminology for describing the analysis of type derivation trees:

- *Conflicting types* - refers to types that participate in a type mismatch where the type of the term fails to conform to the expected type.
- *Source of the type* - refers to the inference judgment where the given type is first introduced in the type derivation tree. For instance, in the `foldRight` application analyzed in Figure 3.4, the *source* of the expected type $List[\bot]$ in a type mismatch refers to the `(app)` inference judgment $[^?/_b]b, \Gamma \vdash^{w} \texttt{Nil()} :_{a\Rightarrow\bot} List[a] \nearrow ?$.
- *Target type* - refers to the type for which we want to find *source(s)* by analyzing the typing decisions of the type derivation tree. When we say that a target type is represented by some *TypeFocus* instance, we mean that a type selection on some type corresponds semantically to that target type.
  To help with the interpretation of *TypeFocus type selection*, we typically represent the extracted type component through a grayed-out selection on types on which *TypeFocus* is applied to.
  For example, $[\phi_{\texttt{fun-param}}]$ applied to type $\forall a.\ a \rightarrow Int$ extracts $\forall a.\ \boxed{a} \rightarrow Int$
- *Basic and complex TypeFocus* - we classify a *TypeFocus* instance as a basic one if it is either $[\,]$, $[\phi_{\texttt{fun-param}}]$, $[\phi_{\texttt{fun-res}}]$ or $[\phi_{\texttt{sel}_x}]$. A complex *TypeFocus* instance consists of at least two *TypeFocus* selectors *i.e.,* the *length* of the type selection path is at least of size 2.
- *Type inference rule of the type inference judgment* - always refers to the last type inference rule used in the given inference judgment.

The correlation between the nodes of derivation and *TypeFocus*

The analysis of type derivation trees can quickly become infeasible due to the amount of type information. To make it practical, our analysis will associate every node of the type derivation tree with a particular *TypeFocus* instance, thus reducing the typing information to a simple concept of type selection.

We illustrate the intuition behind such *TypeFocus* and type inference rule association using Figure 3.9. The figure summarizes the essential elements of the informal analysis of the

term	prototype	inferred type	*TypeFocus*
`Cons(x + 1)`	$\boxed{\text{List}[\bot]}$	$List[Int] \not\nearrow List[\bot]$	$[\,]$
`fun`$((x,y) \to$ `Cons`$(\texttt{x}+1,y))$	$(\texttt{Int}, List[\bot]) \to \boxed{\text{List}[\bot]}$	*Undefined*	$[\phi_{\texttt{fun-res}}]$
`foldRight`(xs)`(Nil())`	?	$_{\{b \Rightarrow List[\bot]\}}(((Int, b) \to \boxed{\text{b}}) \to b)$	$[\phi_{\texttt{fun-param}}, \phi_{\texttt{fun-res}}]$
`foldRight`(xs)	?	$\forall \overline{b}.b \to (((Int, b) \to \boxed{\text{b}}) \to b)$	$[\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}, \phi_{\texttt{fun-res}}]$

Figure 3.9: Summary of the analysis of the type derivation tree for `foldRight` application from Figure 3.4 leading to the source of type LIST[⊥]. The columns represent the elements of the $\vdash^w$ inference judgment (the environment is omitted). The *TypeFocus* value encapsulates the target type information at each node.

source of type LIST[⊥] in the `foldRight` function application (Figure 3.5 in Section 3.1) by listing all involved $\vdash^w$ type inference judgments. Each row corresponds to the type derivation tree node in a path from the conflicting types to the source of the inherited type. The last column reduces the target type information to *TypeFocus* instances.
The summary highlights how *TypeFocus* unifies type selection on prototypes up to the *Propagation Root* (the first two rows) with type selection on the inferred types for terms up to the source of type LIST[⊥] (the last two rows).

Since *TypeFocus* instances correspond to partial type information that is propagated at each node of the type derivation tree, the *TypeFocus* information can be further simplified to the following *TypeFocus* composition:

term	type inference rule	*TypeFocus*
`Cons(x + 1)`	`(app)`	$\Theta_1 = [\,]$
`fun`$((x,y) \to$ `Cons`$(\texttt{x}+1,y))$	`(abs)`	$\Theta_2 = \phi_{\texttt{fun-res}} :: \Theta_1$
`foldRight`(xs)`(Nil())`	`(app)`	$\Theta_3 = \phi_{\texttt{fun-param}} :: \Theta_2$
`foldRight`(xs)	`(app)`	$\Theta_4 = \phi_{\texttt{fun-res}} :: \Theta_3$

The summary reveals the relation between the incremental *TypeFocus* construction and propagation of partial type information in the rules that define the Local Type Inference.

The *TypeFocus*-based analysis - the intuition

With the semantics of *TypeFocus* explained, we are now in a position to illustrate its practical application on a non-trivial example that analyzes the decisions of the type inference rule, `(app)` (from Figure 2.4), that infers the type of function applications.

For the purpose of the example we can consider a fragment of the type derivation tree that inferred the type of a function application, say $f(e)$, where $(?,\ \epsilon \vdash^w f(e) :\ \{x : A \to B, y : C\})$.

We assume that the type of the function f has also been inferred in the $(?,\, \epsilon \vdash^{w} f :\ \forall a.\ a \rightarrow \{x : a, y : C\})$ judgment, and let the inferred type variable substitution be $\sigma_{f(e)}$, where $\sigma_{f(e)} = [a \Rightarrow (A \rightarrow B)]$.
The example aims to explain the source of two different elements of the inferred type of the function application, represented by the $\Theta^{\mathtt{f(e)}}$ *TypeFocus*:

Case $\Theta^{f(e)} = [\phi_{sel_x}, \phi_{fun\text{-}res}]$*:*
The *TypeFocus* represents the target type B in the inferred type of function application because $\Theta^{\mathtt{f(e)}}(\{x : A \rightarrow B, y : C\}) = \mathtt{inl}\ B$.

From the `(app)` type inference rule we know that:

1. The inferred type of function application is the same as the result type of the inferred type of the function (T in $(?,\, \epsilon \vdash^{w} f' :\ \forall \overline{a}.S \rightarrow T)$ for some f') modulo the type variable substitution (we ignore the consequences of the $\nearrow$ adaptation for the moment).

2. The type variable substitution does not modify the components of types, except for providing type instantiation for the abstract type variables.

This means that the type resulting from the application of the $\Theta^{\mathtt{f(e)}}$ to the inferred type of the function application, and the type resulting from the application of the $\Theta^{\mathtt{f(e)}}$ to the result type of the function, refer to the same type even though they might return different type values.

An application of $\Theta^{\mathtt{f(e)}}$ to the result type of the inferred function type of f gives $\Theta^{\mathtt{f(e)}}(\{x : a, y : C\}) = \mathtt{inr}\ \langle a, [\phi_{\mathtt{fun\text{-}res}}]\rangle$. The extracted type variable a reveals that in order to understand the origin of the target type B we have to find out how $\sigma_{f(e)}$ type substitution, that instantiated the type variable a, was inferred in the first place, but we do not have to continue the analysis of the premise that inferred the type of the function. In other words, we have found a desired source of the target type.
The function application node in the type derivation represents the source of the target type but it does not yet reveal how the $\sigma_{f(e)}$ type substitution was inferred. That is why, the source can be categorized as the *intermediate* one, and requires further analysis as we will explain later in the section. Intuitively, the *failed TypeFocus*, $[\phi_{\mathtt{fun\text{-}res}}]$, will allow us continue the analysis of the inferred instantiation of the type variable since $[\phi_{\mathtt{fun\text{-}res}}](\sigma(a)) = \mathtt{inl}\ B$ still extracts the target type information.

Case $\Theta^{f(e)} = [\phi_{sel_y}]$*:*
The *TypeFocus* represents the target type C in the inferred type of the function application because $\Theta^{\mathtt{f(e)}}(\{x : A \rightarrow B, y : C\}) = \mathtt{inl}\ C$.

Using the same argument as in the previous case, $\Theta^{\mathtt{f(e)}}$ can be applied to the inferred result type of the function, *i.e.,* $\Theta^{\mathtt{f(e)}}(\{x : a, y : C\}) = \mathtt{inl}\ C$. The extracted type, which is not a type variable, indicates that for locating the source of the target type, one can immediately

navigate to the node that inferred the type of the function, and ignore the type variable substitution and the decisions that inferred the type of the argument e.
The analysis of the inferred type of the function, $(?,\ \epsilon \vdash^{w} f:\ \forall a.\ a \rightarrow \{x:a, y:C\})$, must not use the same $\Theta^{\mathtt{f(e)}}$ *TypeFocus*. The latter fails to represent the desired target type when applied to the inferred type of the function, *i.e.,* $\Theta^{\mathtt{f(e)}}(\{x:A \rightarrow B,\ y:C\}) = \mathtt{inl}\ C$ but $\Theta^{\mathtt{f(e)}}(\forall a.\ a \rightarrow \{x:a,\ y:C\}) = \mathtt{inr}\ \langle \forall a.\ a \rightarrow \{x:a,\ y:C\},\ \Theta^{\mathtt{f(e)}} \rangle$.
To derive rules for constructing *TypeFocus* instances that cross the boundaries of individual type inference rules, we look at the expected shape of the inferred types. In the case of the `(app)` rule, the inferred type of the function has to be a polymorphic function type. Therefore in order to provide a type selection that faithfully represents the initial target type information in the $(?,\ \epsilon \vdash^{w} f:\ \forall a.\ a \rightarrow \{x:a, y:C\})$ judgment, we append $\phi_{\mathtt{fun\text{-}res}}$ to $\Theta^{\mathtt{f(e)}}$ since $(\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{f(e)}})(\forall a.\ a \rightarrow \{x:a,\ y:C\}) = \mathtt{inl}\ C$, as desired.

With the above examples we have illustrated the construction and application of *TypeFocus*, The process is dependent only on the formal definition of the type inference rule, and yet can guide the navigation over the type derivation tree.

3.3.2 The well-formedness property

TypeFocus represents a type selection for the already inferred type of an inference rule. We have shown that in the application of *TypeFocus* to error-free examples, one can still return some *failed* partial type selections. In order to distinguish incorrectly constructed *TypeFocus* instances, we now formally define the difference between the *correct* and the *invalid* partial type selections.

Correct partial type selections exist only when dealing with types that have uninstantiated type variables. In the previous section we have applied *TypeFocus* to the result type of the inferred function type, which resulted in a right tagged tuple. The *failed* type selection is still correct since the used *TypeFocus* has been constructed for the same type but with type variables already instantiated. We summarize such well-behaved type selections in Definition 6.

Definition 6 *Well-formedness of TypeFocus for some type T ($\Theta, \overline{a} \vdash^{\mathtt{WF}} T$).*
The *TypeFocus* Θ is well-formed with respect to some type T in the context of some uninstantiated type variables $\overline{a}$, denoted as $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$, iff

- $\Theta(T) = \mathtt{inl}\ T'$, or
- $\Theta(T) = \mathtt{inr}\ \langle T', \Theta' \rangle$ and $(T' \in \overline{a}\ \vee\ T' = ?)$

The *TypeFocus* Θ is *strictly* well-formed with respect to some type T iff $\Theta(T) = \mathtt{inl}\ T'$ for some T'.

The well-formedness judgment ensures that an application of a *TypeFocus* to some type can either fully extract the desired part of the target type it represents, or it extracts up-to an uninstantiated type variable. The definition also ensures that *TypeFocus* instances are well-behaved with respect to prototypes and can handle wildcard constant types.

Definition 7 *Partial type selection.*
The *partial type selection* on type T using *TypeFocus* Θ (in the context of uninstantiated type variables $\overline{a}$), refers to a well-formed type selection, $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$, such that $\Theta(T) = \mathtt{inr}\ \langle T', \Theta' \rangle$ for some T' and Θ', where $T' \in \overline{a}$.

For example, for *TypeFocus* $\Theta^{\mathsf{S}} = [\phi_{\mathtt{sel}_{\mathtt{x}}}]$:
$(\Theta^{\mathsf{S}}, \emptyset \vdash^{\mathtt{WF}} \{x : A \to B, y : C\})$ and $(\Theta^{\mathsf{S}}, \{\, a \,\} \vdash^{\mathtt{WF}} a)$ but $(\Theta^{\mathsf{S}}, \{\, a \,\} \nvdash^{\mathtt{WF}} a \to \{x : a, y : C\})$.

The well-formedness property ensures that an application of *TypeFocus* to some type is safe and performs a correct, potentially partial, type selection. An inversion lemma in Lemma 3.2 summarizes the finding. Later in the chapter, the guarantees of the lemma will prove to be sufficient to guide the analysis of the type inference rules in a directed way.

Lemma 3.2 *Inversion lemma for well-formed type selection.*
If $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ for any Θ, T, and $\overline{a}$, then the result of $\Theta(T)$ is:

- $\mathtt{inl}\ T'$ for some T', or
- $\mathtt{inr}\ \langle T'', \Theta'' \rangle$ for some T'' and Θ'', where $T'' \in \overline{a} \vee T'' = \,?$.

Proof.
Straightforward. Directly from Definition 6 on well-formedness of *TypeFocus* with respect to types. □

Lemma 3.3 *Inversion lemma for strictly well-formed type selection.*
If $(\Theta, \emptyset \vdash^{\mathtt{WF}} T)$ for any Θ, and T, where $? \not\in T$, then the result of the $\Theta(T)$ application is $\mathtt{inl}\ T'$ for some T'.

Proof.
Straightforward. Directly from Definition 6 on type selection that is strictly well-formed with respect to types. □

3.4 Foundations of *Typing Slice*

With the *TypeFocus* abstraction we have shown a technique for navigating type derivation trees. In this section we describe the actual outcome of the *TypeFocus*-based analysis.

Previous work on analyzing type errors and type system decisions has typically represented the results of its analysis through minimal program source locations, minimal sets of conflicting type constraints or program modifications, as described in Chitil [2001], Stuckey and Sulzmann [2005], Haack and Wells [2004], and Chen and Erwig [2014b]. Our algorithm will instead find type derivation subtrees that introduce the target type for the first time. This way we can explain if the target type has been synthesized, inherited, or a mixture of both, and still choose the most suitable representation for expressing the result (*e.g.,* source code modification, visual type derivation tree exploration or source code location). The result also includes the *TypeFocus* value associated with the type derivation subtree, or rather the type that it inferred. The *TypeFocus*, as in all the previous cases encapsulates the focus on the target type that is part of the inferred type.

The result of the algorithm may not necessarily be final, in a sense that further typing decisions from the returned type derivation subtree could potentially be irrelevant when explaining the source of the target type. At the same time, as we will show, the result contains all the necessary inputs for further analysis, *i.e.,* the typing judgment and the *TypeFocus*.

The conscious choice has significant consequences for the end-users, as well as for the construction of the analysis of the typing decisions. The intermediate type derivation subtrees allow us to inform users about the intermediate typing decisions, and their corresponding program locations, that led to the inference of some target type, especially important for explaining non-trivial dependencies that span over the different program locations.

Intermediate typing decisions returned by the algorithm can be grouped together based on how they affected the target type (such as, was the type partially or completely synthesized, inherited or a mixture of both?). Each of the groups has to be analyzed in a different way but the intermediate results allow us to define analysis *components* that are well-defined and isolated from the other reasons, keeping the core of the algorithm simple and the approach in general viable to language extensions.

Finally, because local type inference propagates type information locally between the adjacent nodes, our approach to finding such adjacent intermediate typing decisions is on a par with the principles of Colored Local Type Inference (and in contrast to global type inference techniques where it would be redundant).

3.4.1 Typing Slice

The result of a *TypeFocus*-based analysis is a final or an intermediate source of the target type. Both are represented through an abstraction named Typing Slice. Due to a range of typing decisions that may effectively infer the target type, we have to allow for different kinds of *Typing Slices*. To represent them we use a triple $\langle \nu, (P, \Gamma \vdash^{w} E : T), \Theta \rangle$ consisting of:

- A slice kind, ν, identifying the kind of typing decision that led to the target type. We provide a classification of the slices later in the section.
- An error-free type inference judgment (or type derivation subtree), $(P, \Gamma \vdash^{w} E : T)$.
- A *TypeFocus*, Θ, representing the target type information in the inferred type of the included inference judgment. The *TypeFocus* satisfies the well-formedness property with respect to the inferred type T, *i.e.*, $\Theta, \mathtt{fv}(T) \vdash^{\mathtt{WF}} T$.

For presentation reasons, we use the ν_3 notation for *Typing Slice* triples later in the work ($\nu_3 ::= \langle \nu,\ (P, \Gamma \vdash^{w} E : T),\ \Theta \rangle$).

The kind of the typing slice ranges over four different categories, symbolically represented as:

$$\nu ::= \nu_{\mathtt{PT}} \mid \nu_{\mathtt{ADAPT}} \mid \nu_{\mathtt{TVAR}} \mid \nu_{\mathtt{TSIG}}$$

all of which we will now discuss in turn. The *Typing Slice*s belonging to the same category exhibit the same properties, in a sense that they can be further analyzed with the same category of well-defined techniques that will be subject of sections 3.6, 3.7 and 3.8, respectively.

3.4.2 Prototype Typing Slice

A *Prototype Typing Slice* ($\nu_{\mathtt{PT}}$) allows us to represent typing judgments where the target type is inferred in a type checking mode, meaning that it has been fully inherited from the context. In practice, this means that any premise or auxiliary typing decision that is part of the inference rule can be safely ignored. In the Prototype Typing Slice the included *TypeFocus* not only represents a well-formed type selection on the inferred type of the term, but also on the prototype that is part of the inference judgment.

To illustrate the objective of the Prototype Typing Slice we consider the analysis of the same type inference judgment that inferred the type of some term to be $(Int \to Int)$ but with a different target type:

- $(\boxed{Int} \to Int)$ - we need to explain the source of the parameter type.
- $(Int \to \boxed{Int})$ - we need to explain the source of the result type.

The difference means that the value of *TypeFocus*, say $\Theta^{\texttt{fun}}$, just reflects the fact that as part of the analysis of the type derivation tree we have reached the particular node with a different objective (or rather different target type to explain).

In the first case $\Theta^{\texttt{fun}} = [\phi_{\texttt{fun-param}}]$:

$$(\texttt{abs}_{tp})\ Int \rightarrow ?,\ \emptyset \vdash^{w} \mathbf{fun}(x\colon Int)x\colon Int \rightarrow Int \nearrow \colorbox{lightgray}{Int} \rightarrow ?\ \mid \colorbox{lightgray}{Int} \rightarrow Int$$

The type after the | symbol (not part of the official type inference judgment) shows the computed result of the $\nearrow$ operation that adapts the type of the term to the provided prototype. In the presented inference judgment, the target type has been fully inherited from the prototype - the application of $\Theta^{\texttt{fun}}$ to the prototype, $\Theta^{\texttt{fun}}(Int \rightarrow\ ?) = \texttt{inl}\ Int$, and the application of $\Theta^{\texttt{fun}}$ to the inferred type, $\Theta^{\texttt{fun}}(Int \rightarrow Int) = \texttt{inl}\ Int$, yield the same parameter type of the function type.

In the second case $\Theta^{\texttt{fun}} = [\phi_{\texttt{fun-res}}]$:

$$(\texttt{abs}_{tp})\ Int \rightarrow ?, \emptyset \vdash^{w} \mathbf{fun}(x\colon Int)x\colon Int \rightarrow Int \nearrow Int \rightarrow \colorbox{lightgray}{$?$}\ \mid Int \rightarrow \colorbox{lightgray}{Int}$$

In the presented inference judgment, the target type has **not** been fully inherited from the prototype - the application of $\Theta^{\texttt{fun}}$ to the prototype, $\Theta^{\texttt{fun}}(Int \rightarrow ?) = \texttt{inl}\ ?$, yields a constant wildcard type, meaning that no information regarding the particular fragment of the inferred type has been enforced from the outside context at this node of the derivation.

In consequence, the first target type will be explained using the Prototype Typing Slice, while the second one must not. The practical implications of such statement are that in the former case we can completely ignore the analysis of the premises of the type inference rule $(\texttt{abs}_{tp})$, while for the latter example this is not the case.

The relation between the application of the same type selection of the target type to the inferred type and to the prototype is formally defined in Lemma 3.4. The statement establishes that the application of *TypeFocus* to the prototype is safe, provided that it is safe with respect to the inferred type of the term. This in turn implies that such *TypeFocus* will extract the same part of the prototype that inferred the target type.

Lemma 3.4 *Well-formedness of TypeFocus with respect to the prototype.*
If $(P, \Gamma \vdash^{w} E : T)$ and $(\Theta, \overline{a} \vdash^{\texttt{WF}} T)$ for $\texttt{fv}(T) \subseteq \overline{a}$, then $(\Theta, \overline{a} \vdash^{\texttt{WF}} P)$.

Proof.
Proof by induction on the structure of the *TypeFocus* instances. A full proof is provided in Appendix D.1. □

3.4.3 Type Variable Typing Slice

Type inference rules for function applications, (app) and ($\mathtt{app}_{tp}$) determine optimal instantiation for all type variables that are present in the polymorphic function type. Type inference rules specify that the instantiation comes either from the collected and solved type constraints, or from the explicit type arguments, as illustrated in our informal introduction to *TypeFocus*.

We use the *Type Variable Typing Slice* (ν_{TVAR}) to identify a type inference judgment where the instantiation of the type variable is the source of the target type.

3.4.4 Adaptation Typing Slice

An *Adaptation Typing Slice* ($\nu_{\mathtt{ADAPT}}$) stands for a typing decision that infers the target type as part of the result of the $\nearrow$ adaptation. We recall that the $\nearrow$ operation adapts the type of a term to a prototype, which may lead to type synthesis when their shapes are structurally different.

For example, let *TypeFocus* $[\phi_{\mathtt{fun\text{-}res}}, \phi_{\mathtt{fun\text{-}param}}]$ represent the target type in some inference judgment (the type after the | symbol represents the inferred type):

$$(\mathtt{abs}_{tp}) \quad A \to ? \to ?,\ \epsilon \vdash^{w} e :\ A \to \bot \nearrow A \to ? \to ? \ \mid A \to \top \to \bot$$

In the example term e was assigned type $A \to \bot$. Such type is a subtype of the inferred one but does not match the shape of the required prototype. Reporting the Adaptation Typing Slice indicates that the selected part of the inferred type has been synthesized during the $\nearrow$ adaptation, as highlighted in the involved components of types: $A \to \bot \nearrow A \to ? \to ?\ \mid A \to \top \to \bot$.

Therefore, in comparison to the Prototype Typing Slice, returning the Adaptation Typing Slice indicates the two elements that explain the source of the target type: the part of the inherited prototype and the synthesized type of the term.

3.4.5 Type Signature Typing Slice

The *Type Signature Typing Slice* ($\nu_{\mathtt{TSIG}}$) represents typing decisions that have fully synthesized the type of the target type from the term. Therefore, for our core language, the source of the target type, which is represented by the Type Signature Typing Slice, stands for one of the following:

- An explicit type annotation for the parameter of the abstraction.
- A variable, whose type is synthesized from the environment.

- An abstraction term.
- A record term.

For example, we consider the typing judgment where the last type inference rule used is `(var)` and the target type is represented through a *TypeFocus* $[\phi_{\texttt{fun-param}}]$:

$$\texttt{(var)} \quad ?,\ (\epsilon, x : Int \to Int) \vdash^{w} x : Int \to Int \nearrow ? \ \mid \ Int \to Int$$

The highlighted part of the inferred type of the term represents the information about the target type.
Reporting the Type Signature Typing Slice identifies the part of the type of a variable x, coming from the environment ($(\epsilon, x : Int \to Int)$), as the source of the target type. Such Typing Slice can be inferred because none of the previous kinds of the *Typing Slice* (involving prototype or the type variables) applied for the given scenario.

An abstraction term becomes the source of the target type if the target type is a function type, and the inferred type of the abstraction has been synthesized from the term. Similarly, a record term can become the source of the target type if the target type is a record type, and the inferred type of the record has been synthesized from the term.

As we will indicate in Section 3.8, the *Type Signature Typing Slice*s are final, in a sense that they do not require further analysis of the associated type derivation trees and can directly be associated with program locations.

3.5 *TypeFocus*-based analysis of type derivation trees

This section presents an algorithm to locate the source of the target type in a type derivation tree. The algorithm is *TypeFocus*-based, meaning that we navigate only those typing decisions, or nodes of the type derivation tree, which affect the inference of the target type. As a result, at each such node, we can extract the target type information from the type inferred as part of the inference judgment of the node.

We first give examples on how previously described *Typing Slices* fit as an outcome of the algorithm for analyzing inference typing decisions, and follow with definitions of auxiliary functions used in the algorithm. Next, we describe in detail the algorithm for three representative inference rules. Finally, we show that algorithms used for analyzing typing decisions of those rules can be generalized and be applied to the rest of the Colored Local Type Inference formalization.

The *TypeFocus*-based algorithm presented in this section is **only** defined for derivable (or error-free) type derivation subtrees. As we illustrate in Section 4, the information about the erroneous parts of the type derivation trees can always be reduced to the appropriate *Type-Focus* abstraction. Such an approach allows us to apply the core algorithm not only for exploring typing decisions (with error-free types) but also for debugging type errors in general, all without modifying the core algorithm.

The algorithm - overview of the results

The algorithm is realized using the SLICES function of type $((P,\ \Gamma \vdash^{w} E :\ T),\ \Theta) \rightarrow \overline{\nu_3}$. The function takes a derivable type inference judgment, corresponding to a subtree of the type derivation tree and represented by its local root, and a *TypeFocus*, which represents a selection on the inferred type of the given type inference judgment.. The function returns a sequence of *Typing Slices* triples explaining the source of the selected target type. In order to analyze type derivation trees, the SLICES function is defined for every inference rule of the Colored Local Type Inference formalization. The function is organized in a recursive manner, since the type inference algorithm is recursive itself.

To illustrate the result of the algorithm, we apply the SLICES function to the inference judgment that inferred the type of some nested abstraction $(\texttt{fun}(x)\texttt{fun}(y : A)y)$ in some type checking context:

$$\begin{aligned}&\textsc{slices}\Big(\ \texttt{(abs)}\ \ (Int \rightarrow\ ? \rightarrow\ ?, \epsilon \vdash^{w} \texttt{fun}(x)\texttt{fun}(y : A)y : Int \rightarrow \colorbox{lightgray}{A} \rightarrow A),\ [\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}]\ \Big) = \\ &\quad \Big\{\Big\langle\ \nu_{\texttt{TSIG}}, (? \rightarrow\ ?,\ \epsilon \vdash^{w} \texttt{fun}(y : A)y : \colorbox{lightgray}{A} \rightarrow A), [\phi_{\texttt{fun-param}}]\ \Big\rangle\Big\}\end{aligned}$$

In the example, the last type inference rule used in the judgment is `(abs)` and when applied to the provided term it infers the type $Int \rightarrow A \rightarrow A$. The target type is highlighted in the inferred type of the abstraction using the information from the provided *TypeFocus* value. The algorithm returns the Type Signature Typing Slice, which explains the source of the target type - it was first introduced by the nested abstraction (or rather the type inference judgment that determined its type). We encourage the reader to write down the complete type derivation tree (consisting all together of three type inference rules) to verify the result. The returned Typing Slice also includes the *TypeFocus* value which selects the part of the inferred type, thus allowing us to associate the source of the target type with the type annotation of the parameter of the abstraction.

As the algorithm is *TypeFocus*-based, it will return different results for different target types of the same judgment. For example, let's consider a different target type, expressed through a *TypeFocus* $[\phi_{\texttt{fun-param}}]$:

$$\begin{aligned}&\textsc{slices}\Big(\ \texttt{(abs)}\ \ (Int \rightarrow\ ? \rightarrow\ ?,\ \epsilon \vdash^{w} \texttt{fun}(x)\texttt{fun}(y : A)y : \colorbox{lightgray}{Int} \rightarrow A \rightarrow A),\ [\phi_{\texttt{fun-param}}]\ \Big) = \\ &\quad \Big\{\Big\langle\ \nu_{\texttt{PT}}, (Int \rightarrow\ ? \rightarrow\ ?,\ \epsilon \vdash^{w} \texttt{fun}(x)\texttt{fun}(y : A)y : \colorbox{lightgray}{Int} \rightarrow A \rightarrow A), [\phi_{\texttt{fun-param}}]\ \Big\rangle\Big\}\end{aligned}$$

The result, a Prototype Typing Slice, correctly identified that the highlighted target type Int has been fully inherited from the context.

$$
\begin{array}{lll}
\texttt{is-hole} & : & P \rightarrow \texttt{Bool} \\
\texttt{is-tvar} & : & (T, \overline{a}) \rightarrow \texttt{Bool}
\end{array}
\quad
\begin{array}{lll}
\texttt{shape-match} & : & (T,\ P,\ \Theta) \rightarrow \texttt{Bool} \\
\texttt{head} & : & \Theta \rightarrow \Theta \\
\texttt{tail} & : & \Theta \rightarrow \Theta
\end{array}
\quad
\begin{array}{lll}
\texttt{prefix} & : & (\Theta, \Theta) \rightarrow \texttt{Bool} \\
\texttt{normalize} & : & (\Theta,\ T,\ \overline{a}) \rightarrow \Theta
\end{array}
$$

Figure 3.10: Type signatures of auxiliary functions used in the definition of the SLICES algorithm. `Bool` type stands for the type of Boolean values `true` and `false`.

Auxiliary functions

The definition of the algorithm makes use of a few auxiliary functions, which are summarized in Figure 3.10. For reference, Appendix B provides a complete implementation for each of the defined functions. Rather than providing a detailed motivation behind each of them at this point, we encourage the reader to come back to this section, if necessary, once they experience their usage in our core algorithm in Section 3.5.1.

`is-hole` The `is-hole` function takes a prototype, and returns a Boolean value `true` if the prototype is a ? constant type, and `false` otherwise.

`is-tvar` The `is-tvar` function returns a Boolean value `true` if the provided type is a type variable and it is within the provided set of uninstantiated type variables. The function returns `false` otherwise.

`shape-match` The `shape-match` function verifies if a provided type, T, and a prototype, P, are structurally equal within the type selection of the provided *TypeFocus*. The function assumes that $(P, \texttt{fv}(P) \vdash^{\texttt{WF}} \Theta)$, where Θ represents the provided *TypeFocus* value.
By taking into account the type selection of the input *TypeFocus*, the function can determine if only the desired part of the synthesized type and the prototype are structurally unequal, rather than considering the full synthesized type and the full prototype. In consequence, we can precisely identify when the type synthesis of the $\nearrow$ adaptation is the source of the part of the inferred type.

For example, the $A \rightarrow \bot$ type is clearly not structurally equal to the $A \rightarrow\ ? \rightarrow\ ?$ prototype, but their type elements can be:

- $\texttt{shape-match}(A \rightarrow \bot, A \rightarrow\ ? \rightarrow ?,\ [\phi_{\texttt{fun-res}}]) = \texttt{false}$:
  In the case of the $A \rightarrow \bot \nearrow A \rightarrow\ ? \rightarrow\ ?\ =\ A \rightarrow\ \top \rightarrow\ \bot$ adaptation, the `false` result indicates that the selected part of the inferred type ($A \rightarrow \top \rightarrow \bot$) has been synthesized as part of the $\nearrow$ adaptation.
- $\texttt{shape-match}(A \rightarrow \bot,\ A \rightarrow\ ? \rightarrow\ ?,\ [\phi_{\texttt{fun-param}}]) = \texttt{true}$:
  In the case of the $A \rightarrow \bot \nearrow A \rightarrow\ ? \rightarrow\ ?\ =\ A \rightarrow\ \top \rightarrow\ \bot$ adaptation, the `true` result indicates that the selected part of the inferred type ($A \rightarrow\ \top \rightarrow\ \bot$) has not been synthesized as part of the $\nearrow$ adaptation.

head* *and* *tail The `head` and the `tail` functions extract the elements of *TypeFocus* path in a similar way as *head* and *tail* extract elements of list collection.
The main difference from their list counterparts is that `head` and `tail` are both total functions, returning [ ] when `head` is applied to an empty type selection, and when `tail` is applied to any basic *TypeFocus*. Therefore the `head` and `tail` functions break *TypeFocus* down into a first non-empty selection *TypeFocus* that would have been applied to any type, and the remaining *TypeFocus* composition directly following it, respectively. For example,

- $\texttt{head}([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}]) = [\phi_{\texttt{fun-res}}]$, $\texttt{head}([\phi_{\texttt{fun-res}}]) = [\phi_{\texttt{fun-res}}]$, and $\texttt{head}([\,]) = [\,]$
- $\texttt{tail}([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}]) = [\phi_{\texttt{fun-param}}]$, and $\texttt{tail}([\phi_{\texttt{fun-res}}]) = [\,]$.

The `head` and `tail` deconstruction of any *TypeFocus* satisfies the decomposition condition that is specified in Definition 8. The condition ensures that an application of *TypeFocus* to any type is equivalent to an application of the `head` of the *TypeFocus* followed by an application of the `tail` of the *TypeFocus*.

Definition 8 *Decomposition of TypeFocus.*
$\forall \Theta, T.\ \Theta(T) == (\texttt{head}(\Theta) ::: \texttt{tail}(\Theta))(T)$

The ability to decompose the *TypeFocus* will prove crucial for navigating type derivation trees when combined with the well-formedness property. In Lemma 3.5 we present Canonical Forms of *TypeFocus* instances that can be inferred from the type selections that are well-formed with respect to some types. As we will explain later in the section, the lemma will prove to be crucial for defining rules that navigate type derivation trees based solely on the shapes of types and the preservation of the well-formedness property.

Lemma 3.5 *Canonical Forms.*
For any *TypeFocus* Θ and type T, such that $(\Theta, \overline{a} \vdash^{\texttt{WF}} T)$ and $\texttt{fv}(T) \subseteq \overline{a}$:

1. If T is a type $\bot$, then $\texttt{head}(\Theta)$ is $[\,]$.
2. If T is a type $\top$, then $\texttt{head}(\Theta)$ is $[\,]$.
3. If T is a type $\forall \overline{a}.T_1 \to T_2$, then $\texttt{head}(\Theta)$ is either $[\,]$, $[\phi_{\texttt{fun-param}}]$, or $[\phi_{\texttt{fun-res}}]$.
4. If T is a type $\{x_1 : T_1, \ldots, x_n : T_n\}$, then $\texttt{head}(\Theta)$ is either $[\,]$ or $[\phi_{\texttt{sel}_{x_i}}]$ where $1 \le i \le n$.
5. If T is a type variable, then $\texttt{head}(\Theta)$ is undefined.

Proof.
Straightforward. From the well-formedness property in Definition 6 and the type selection specification in Definition 4. □

Lemma 3.6 *Canonical Forms for strict well-formed type selections.*
For any *TypeFocus* Θ and type T, such that $\Theta, \emptyset \vdash^{\mathtt{WF}} T$:

1. If T is a type $\bot$, then $\mathtt{head}(\Theta)$ is $[\,]$.
2. If T is a type $\top$, then $\mathtt{head}(\Theta)$ is $[\,]$.
3. If T is a type $\forall \overline{a}.T_1 \rightarrow T_2$, then $\mathtt{head}(\Theta)$ is either $[\,]$, $[\phi_{\mathtt{fun\text{-}param}}]$, or $[\phi_{\mathtt{fun\text{-}res}}]$.
4. If T is a type $\{x_1 : T_1, \ldots, x_n : T_n\}$, then $\mathtt{head}(\Theta)$ is either $[\,]$ or $[\phi_{\mathtt{sel}_{x_i}}]$ where $1 \le i \le n$.
5. If T is a type variable, then $\mathtt{head}(\Theta)$ is $[\,]$.

Proof.
Straightforward. A trivial extension of proof for Lemma 3.5 with *TypeFocus* instances that do not allow for partial type selections. □

`prefix` The `prefix` function determines if one *TypeFocus* is a *prefix* of the latter. The `prefix` function is realized by the prefix *property* provided in Definition 9.

Definition 9 *The TypeFocus prefix property.*
For any *TypeFocus* values, say Θ and Θ', if Θ is a *prefix* of Θ', then the prefix *property*, denoted as $\mathtt{prefix}^{P}_{(\Theta,\Theta')}$, is satisfied.
The prefix *property* is defined recursively as:

$$\mathtt{prefix}^{P}_{(\Theta,\Theta')} \Leftrightarrow (\Theta = [\,]) \text{ or } (\mathtt{head}(\Theta) \; == \; \mathtt{head}(\Theta') \text{ and } \mathtt{prefix}^{P}_{(\mathtt{tail}(\Theta),\mathtt{tail}(\Theta'))})$$

For example,

- $\mathtt{prefix}([\phi_{\mathtt{fun\text{-}res}}], [\phi_{\mathtt{fun\text{-}res}}, \phi_{\mathtt{fun\text{-}param}}]) = \mathtt{true}$, and
  $\mathtt{prefix}([\,], [\phi_{\mathtt{fun\text{-}res}}, \phi_{\mathtt{fun\text{-}param}}]) = \mathtt{true}$.

- $\texttt{prefix}([\phi_{\texttt{fun-res}}], [\phi_{\texttt{fun-param}}]) = \texttt{false}$, and
  $\texttt{prefix}([\phi_{\texttt{fun-param}}, \phi_{\texttt{fun-res}}], [\phi_{\texttt{fun-param}}]) = \texttt{false}$.

The `prefix` function only verifies if one *TypeFocus* value is a prefix of the other. From the definition of the prefix property it can be easily deducted that any *TypeFocus* instance has a finite number of prefixes. In practice, we want to order the prefixes in terms of the length of the path; typically we are interested in the largest prefix of some *TypeFocus*, as characterized by Definition 10.

Definition 10 *The largest well-formed prefix of TypeFocus*
Given any type T, a set of free variables $\overline{a}$, such that $\texttt{fv}(T) \subseteq \overline{a}$, and any *TypeFocus* Θ, then Θ' represents the largest prefix of Θ well-formed with respect to type T iff

- Θ' is a prefix of Θ, *i.e.,* $(\texttt{prefix}(\Theta', \Theta) = \texttt{true})$, and
- Θ' is well-formed with respect to type T, *i.e.,* $(\Theta', \overline{a} \vdash^{\texttt{WF}} T)$, and
- Θ' is the largest possible prefix of Θ:
  $\forall \Theta''.\ (\texttt{prefix}(\Theta'', \Theta) = \texttt{true}) \wedge (\Theta'', \overline{a} \vdash^{\texttt{WF}} T) \implies (\texttt{prefix}(\Theta'', \Theta') = \texttt{true}) \vee (\Theta'' == \Theta')$

normalize The `normalize` function allows us to relax the precision of *TypeFocus*, necessary for dealing with types that are synthesized as part of the $\nearrow$ adaptation, or with approximations of type constraints[3] (Section 3.7.2). In other words the role of the `normalize` function is to transform *TypeFocus* instances that are not well-formed with respect to some type into the well-formed ones.

For example,

- $\texttt{normalize}([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}], Int \to \bot, \epsilon) = [\phi_{\texttt{fun-res}}]$,
  because $([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}](Int \to \bot) = \texttt{inr}\ \langle \bot, [\phi_{\texttt{fun-res}}] \rangle$.

- $\texttt{normalize}([\phi_{\texttt{fun-res}}], Int \to \bot, \epsilon) = [\phi_{\texttt{fun-res}}]$,
  because $[\phi_{\texttt{fun-res}}](Int \to \bot) = \texttt{inl}\ \bot$.

- $\texttt{normalize}([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}], \forall a.\ Int \to a, \{\ a\ \}) = [\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}]$,
  because $[\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-res}}](\forall a.\ Int \to a) = \texttt{inr}\ \langle a, [\phi_{\texttt{fun-res}}] \rangle$.

[3]The approximation of types refers to calculating least upper bound or greatest lower bound of multiple type constraints.

The `normalize` function takes a *TypeFocus*, Θ, a type, T, and a set of undetermined type variables $\overline{a}$, and returns a *TypeFocus*. The returned value is well-formed with respect to T, and at the same time is a prefix of the input *TypeFocus*. The function is total since we can always find the *TypeFocus* satisfying those conditions (Lemma 3.7) but the `normalize` always returns the largest well-formed prefix of the input *TypeFocus*.

Lemma 3.7 *Existence of the normalized TypeFocus.*
Given any type T, and a set of free type variables $\overline{a}$, such that $\mathtt{fv}(T) \subseteq \overline{a}$, and any *TypeFocus* Θ, we can always find Θ' such that $\mathtt{prefix}(\Theta', \Theta) = \mathtt{true}$ and $\Theta', \overline{a} \vdash^{\mathtt{WF}} T$.

Proof.
If $(\Theta, \mathtt{fv}(T) \vdash^{\mathtt{WF}} T)$ then $(\Theta' \;==\; \Theta)$ since $\mathtt{prefix}(\Theta'', \Theta'') = \mathtt{true}$ for any Θ''.
Otherwise, $\forall \Theta.\, \mathtt{prefix}([\,], \Theta) = \mathtt{true} \wedge ([\,], \mathtt{fv}(T) \vdash^{\mathtt{WF}} T)$, *i.e.,* the identity type selection is a prefix of any *TypeFocus*. $\square$

With such description in mind we can characterize the result of the application of `normalize` function to any *TypeFocus* Θ, any type T, and a set of undetermined type variables $\overline{a}$ as:

- If $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ then the result is Θ itself.
- Else the result is the largest prefix of Θ that is well-formed with respect to type T, according to Definition 10.

3.5.1 Algorithm for analyzing type inference decisions - a fragment

The complete algorithm, represented by the SLICES function, analyzes the decisions of type derivation trees by considering the last type inference rule used in the judgment given as an input. The algorithm is realized by the rule-specialized partial functions, denoted as $\text{SLICES}_{\mathtt{rule}}$, where the *rule* subscript refers to a particular type inference rule of the Colored Local Type Inference formalization. In Figure 3.11 we provide a fragment of the complete algorithm that analyzes decisions of the three representative rules of the Colored Local Type Inference algorithm.

For clarity, each case of the SLICES function provides parameters of the analyzed type inference rule, *i.e.,* the prototype, the environment, the term, and the inferred type. We use a $\vdash^{w}_{*}$ notation that is equivalent to the analyzed inference judgment, to avoid unnecessary duplication of the inference rule elements.
For example, in the $\text{SLICES}_{(\mathtt{abs})}$ function $\vdash^{w}_{*}$ stands for the inference judgment of shape

($\forall\overline{a}.T \to P,\ \Gamma \vdash^{w} \texttt{fun}(x)E : \forall\overline{a}.T \to S$), and for the $\textsc{slices}_{\texttt{(var)}}$ function $\vdash^{w}_{*}$ stands for the inference judgment of shape ($P,\ \Gamma \vdash^{w} x : \Gamma(x) \nearrow P$).

The arguments of the *slices* function, the inference judgment ($P,\ \Gamma \vdash^{w} E : T$) and the *TypeFocus* Θ, must satisfy only one requirement - the provided *TypeFocus* value has to be well-formed with respect to the inferred type, *i.e.,* $(\Theta, \texttt{fv}(T) \vdash^{\texttt{WF}} T)$. The condition is necessary to perform a guided navigation of the type derivation tree.

Analyzing the inferred type of the abstraction

We first describe the $\textsc{slices}_{\texttt{(abs)}}$ algorithm for analyzing typing judgments where the last used type inference rule is `(abs)`. The definition of the algorithm is provided in Figure 3.11.

Given the definition of the type inference rule (in Figure 2.4), the function has to determine if the target type has been inherited from the prototype or whether it has been synthesized in its only premise (no other typing decision could affect any of the possible target types at this point).

The first step of the algorithm determines if the target type has been fully inherited from the expected type of the context. Using the well-formedness pre-condition and Lemma 3.4, we can apply the provided *TypeFocus* to the prototype to identify the part corresponding to the target type (line 1), P_Θ.

If $P_\Theta \neq\ ?$, then the target type information has already been enforced by the context of the inference judgment. In consequence, further analysis of the premises of the `(abs)` rule is fruitless. To represent such type inference decision, the $\textsc{slices}_{\texttt{(abs)}}$ function returns immediately with the Prototype Typing Slice, $\langle \nu_{\texttt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle$, in line 3.

If $P_\Theta =\ ?$, then the target type has been synthesized as part of the typing decisions of the inference judgment. We use the `head` of the input *TypeFocus* to decide on the direction of the analysis. By the $(\Theta, \overline{a} \vdash^{\texttt{WF}} \forall\overline{a}.T \to S)$ precondition and the Canonical Forms lemma (Lemma 3.5), the only allowed values for $\texttt{head}(\Theta)$ are: [ ], $[\phi_{\texttt{fun-param}}]$, and $[\phi_{\texttt{fun-res}}]$.
Both, $[\ ](\forall\overline{a}.T \to P) = \texttt{inl}\ \forall\overline{a}.T \to P$ and $[\phi_{\texttt{fun-param}}](\forall\overline{a}.T \to P) = \texttt{inl}\ T$, extract a non-wildcard prototype when applied to the prototype, and by contradiction, are impossible.
If $\texttt{head}(\Theta) = [\phi_{\texttt{fun-res}}]$ and $P_\Theta =\ ?$, then the prototype carries no partial type information on the target type and the latter is synthesized in the premise of the rule. Therefore $\textsc{slices}_{\texttt{(abs)}}$ analyzes the only premise of the type inference rule, ($P,\ \Gamma, \overline{a}, x : T \vdash^{w} E : S$), in a recursive call to the $\textsc{slices}$ algorithm (line 2), if and only if $\texttt{head}(\Theta) = [\phi_{\texttt{fun-res}}]$.

The *TypeFocus* provided as the argument in the recursive invocation of the $\textsc{slices}$ function differs from the initial one. Knowing that the `head` of Θ selects the result type of the function type, we must exclude this type selection when analyzing the premise of the rule. By definition of `tail` and the decomposition property (Definition 8), $(\texttt{tail}(\Theta), \texttt{fv}(S) \vdash^{\texttt{WF}} S)$, which satisfies the SLICES function precondition in its recursive invocation and correctly represents the target type information when analyzing the premise of the inference judgment.

```
FUNCTION SLICES_(abs) $\big( (\forall\overline{a}.T \to P,\ \Gamma \vdash^{w} \mathbf{fun}(x)E : \forall\overline{a}.T \to S),\ \Theta \big) =$
  $\Theta(\forall\overline{a}.T \to P)_{\mathtt{tpe}} = P_\Theta$
  IF (is-hole($P_\Theta$))        $slices((P,\ \Gamma, \overline{a}, x : T \vdash^{w} E :\ S),\ \mathtt{tail}(\Theta))$
  ELSE                            $\{\ \langle \nu_{\mathtt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$

FUNCTION SLICES_(var) $(\ (P,\ \Gamma \vdash^{w} x : \Gamma(x) \nearrow P),\ \Theta\ ) =$
  $\Theta(P)_{\mathtt{tpe}} = P_\Theta$
  IF (is-hole($P_\Theta$))
    IF (shape-match($\Gamma(x),\ P,\ \Theta$)) $\{\ \langle \nu_{\mathtt{TSIG}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
    ELSE
      normalize($\Theta,\ \Gamma(x),\ \mathtt{fv}(\Gamma(x))$) $= \Theta''$
      $\{\ \langle \nu_{\mathtt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle, \langle \nu_{\mathtt{TSIG}}, \vdash^{w}_{*}, \Theta'' \rangle\ \}$
  ELSE $\{\ \langle \nu_{\mathtt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$

FUNCTION SLICES_(app) $\big( (P, \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T} T \nearrow P),\ \Theta \big) =$
  $\Theta(P)_{\mathtt{tpe}} = P_\Theta$
  IF (is-hole($P_\Theta$))
    IF (shape-match($\sigma_{C_1 \cup C_2, T} T,\ P,\ \Theta$))
      $\Theta(T)_{\mathtt{tpe}} = T_\Theta$
      IF (is-tvar($T_\Theta, \overline{a}$))   $\{\ \langle \nu_{\mathtt{TVAR}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
      ELSE                       $slices(\ (?, \Gamma, \vdash^{w} F : \forall\overline{a}.S \to T),\ \phi_{\mathtt{fun\text{-}res}} :: \Theta)$
    ELSE
      normalize($\Theta,\ \sigma_{C_1 \cup C_2, T} T,\ \emptyset$) $= \Theta''$
      $\Theta''(T)_{\mathtt{tpe}} = T_\Theta$
      $\{\ \langle \nu_{\mathtt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \} \cup$
      ( IF (is-tvar($T_\Theta, \overline{a}$))  $\{\ \langle \nu_{\mathtt{TVAR}},\ \vdash^{w}_{*},\ \Theta'' \rangle\ \}$
        ELSE                     $slices(\ (?,\ \Gamma \vdash^{w} F : \forall\overline{a}.S \to T),\ \phi_{\mathtt{fun\text{-}res}} :: \Theta''\ )$ )
  ELSE $\{\ \langle \nu_{\mathtt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

Figure 3.11: A representative fragment of the SLICES algorithm that analyzes type inference rules of Colored Local Type Inference. A complete algorithm is provided in Section 3.5.3.

Analyzing the inferred type of the variable

The SLICES$_{\mathtt{(var)}}$ function in Figure 3.11 analyzes typing decisions of the type derivation tree, if the last type inference rule used in it, was `(var)` (defined in Figure 2.4).

The analysis of the inference judgment proceeds by checking if the target type has been fully inherited, identically as in the previous case. The algorithm will return a Prototype Typing Slice if the extracted prototype, P_Θ, is not a wildcard constant type. Otherwise we have to search for a different source of the target type.

This time we have to take into account also the consequences of the $\nearrow$ adaptation between the synthesized type of the variable and the inherited prototype. We use the `shape-match` function for that purpose. The function returns `false` if and only if the synthesized type, $\Gamma(x)$, does not match structurally the prototype, P, within the type selection of *TypeFocus*, meaning that the $\nearrow$ operation synthesized the target type.

If `shape-match`$(\Gamma(x),\ P,\ \Theta)$ = `false`, the algorithm returns the Adaptation Typing Slice to reflect the discovery of the source of the target type. The Adaptation Typing Slice (line 6) implies that the source of the target type is both inherited from the context, P, **and** synthesized from the term, x. Since the type `(var)` type inference does not involve any further type inference in its premises, the source of the synthesized type can be immediately represented with the Type Signature Typing Slice (line 6).
We note that the result of the `shape-match` application also implies that the well-formedness of the provided *TypeFocus* value with respect to the synthesized term is not guaranteed, *i.e.,* $(\Theta, \emptyset \vdash^{\mathtt{WF}} P)$ by Lemma 3.4 but $(\Theta, \mathtt{fv}(\Gamma(x)) \vdash^{\mathtt{WF}} \Gamma(x))$ is not necessarily satisfied. In order to return a well-formed type selection in the Type Signature Typing Slice, the former is always normalized with respect to the $\Gamma(x)$ type (line 5).

If `shape-match`$(\Gamma(x),\ P,\ \Theta)$ = `true`, then the analysis of the type inference rule is much simpler - the target type is fully synthesized from the type of the variable and $(\Theta, \mathtt{fv}(\Gamma(x)) \vdash^{\mathtt{WF}} \Gamma(x))$. The algorithm returns immediately with the Type Signature Typing Slice since the target type is introduced in the type derivation tree directly from the environment Γ.

Analyzing the inferred type of the function application

The SLICES$_{\mathtt{(app)}}$ function in Figure 3.11 analyzes typing decisions of the inference judgment where the last used type inference rule is `(app)`.

From the definition of the type inference rule we identify four typing decisions that may be the source of the part of the inferred type:

1. The inherited prototype information, P.
2. The type synthesized from the $\nearrow$ adaptation in $(\sigma_{C_1 \cup C_2, T}\, T) \nearrow P$.
3. The instantiation of a single type variable in the polymorphic function type resulting from the $\sigma_{C_1 \cup C_2, T}$ type substitution.
4. The inference of the type of the function in the $(?,\ \Gamma \vdash^{w} F : \forall \overline{a}.S \to T)$ premise.

Similarly as in the previous cases, the SLICES`(app)` function first checks if the target type has been fully inherited from the context and returns immediately with the Prototype Typing Slice if $P_\Theta \neq$? in line 13.
If $P_\Theta =$?, the target type was not inherited from the context and we have to consider the other possible options.

Similarly as in the case of the `(var)` rule, the `shape-match` function is used to understand the consequences of the $\nearrow$ adaptation. We notice, however, that further analysis, which analyzes how the target type has been synthesized from the term, follows exactly the same steps, irrespective of whether the adaptation affected the target type or not, modulo the value of the used type selection. For clarity, we can summarize the relation between the $\nearrow$ adaptation and the used type selection (here denoted as Θ^{app}) as follows:

- If `shape-match`$(\sigma_{C_1 \cup C_2, T} T,\ P,\ \Theta) =$ `true` then $(\Theta, \emptyset \vdash^{\mathsf{WF}} \sigma_{C_1 \cup C_2, T} T)$ and $\Theta^{\mathsf{app}} = \Theta$,
  from the precondition of the SLICES function.
- If `shape-match`$(\sigma_{C_1 \cup C_2, T} T,\ P,\ \Theta) =$ `false` and `normalize`$(\Theta,\ \sigma_{C_1 \cup C_2, T} T,\ \emptyset) = \Theta''$ then $(\Theta', \emptyset \vdash^{\mathsf{WF}} \sigma_{C_1 \cup C_2, T} T)$ and $\Theta^{\mathsf{app}} = \Theta''$,
  from the precondition of the SLICES function and the definition of the `normalize` function.

In both cases, the resulting *TypeFocus*, Θ^{app}, is well-formed with respect to the synthesized type of the function application. This in turn allows us to decide between the two remaining typing decisions that could have synthesized the target type: the instantiation of a single type variable or the inferred type of the function.

In order to decide between the two, equally valid, possibilities, we apply the *TypeFocus* to the result type of the polymorphic function type which can potentially involve some uninstantiated type variables. The intuition, formally stated in the substitution lemma (Lemma 3.8), uses the fact that the synthesized type of the function application is exactly the same as the result type of inferred type of the function (modulo type substitution). The immediate consequence of the substitution lemma is that a *TypeFocus* that is well-formed with respect to the type with instantiated type variables is also well-formed with respect to the same type but with type variables not being instantiated.

Lemma 3.8 *Well-formedness of TypeFocus over type substitution.*
For any *TypeFocus* Θ, and a type T, such that $(\Theta, \emptyset \vdash^{\mathsf{WF}} T)$, if T results from a type substitution, σ, on some type S, such that $T = \sigma S$ and $dom(\sigma) = \overline{a}$, then $(\Theta, \overline{a} \vdash^{\mathsf{WF}} S)$.

Proof.
Proof by induction on the structure of T. A complete proof is available in Appendix D.3 □

We now turn our attention to the consequences of the application of the Θ^{app} *TypeFocus* to the inferred result type of the $\forall\overline{a}.S \to T$ function type.

If $\Theta^{\mathsf{app}}(T) = \texttt{inl}\ T'$ or $\Theta^{\mathsf{app}}(T) = \texttt{inr}\ \langle T', \Theta'' \rangle$, for all Θ'', such that $T' \in \overline{a}$, then by definition of the `(app)` inference rule

1. The type variable extracted from the inferred polymorphic function type, $\forall\overline{a}.S \to T$, is indirectly the source of the target type.
2. The type variable is only instantiated with the inferred type substitution, $\sigma_{C_1 \cup C_2, T}$, in the analyzed function application judgment.
3. The target type is first introduced as a result of the inferred type substitution and the type substitution itself is the source of the target type.

Our algorithm does not attempt to immediately analyze the source of the extracted type variable instantiation due to our policy of reporting intermediate Typing Slices. Instead the algorithm returns a Type Variable Typing Slice that identifies the inference judgment for the function application, and its inferred type substitution, as the source of the target type. The result can be further analyzed using the type variable-specific analysis methods (Section 3.7).

If $\Theta^{\mathsf{app}}(T) = \texttt{inl}\ T'$ such that $T' \notin \overline{a}$, then the source of the target type is in the type derivation tree that inferred the type of the function. The algorithm analyzes the subtree using the recursive call to the SLICES function with the updated *TypeFocus*, $\phi_{\texttt{fun-res}} :: \Theta^{\mathsf{app}}$, representing a well-formed type selection from the inferred type of the premise.

3.5.2 Algorithm for analyzing type inference decisions - a template

The detailed analysis of the algorithm for the representative rules of Colored Local Type Inference reveals an import insight; most of the decisions of the algorithm can be generalized to form the template for analyzing type inference rules. The generalization illustrates also the key elements that need to be taken into account when defining the analysis functions for new type inference rules. Figure 3.12 presents a `slices-template` function which can be applied to each of the type inference rules to create a type inference rule-specific SLICES function.

The first step of the template (lines 1 and 3) determines if the target type has been fully inherited from the context. An application of the *TypeFocus* to the prototype selects the part corresponding to the target type (by Lemma 3.4). That is why if the extracted part of the prototype is a wildcard constant type then we can be sure that the inference judgment did not inherit the target type from the context. If the extracted part of the prototype, P_Θ is not a wildcard constant type then we always return immediately with a Prototype Typing Slice (line 14) that reflects the source of the target type.

```
FUNCTION slices-template( ($P$, $\Gamma \vdash^{w} E : T$), $\Theta$ ) =
 1  $\Theta(P)_{\texttt{tpe}} = T_{\Theta}$
 2  $T_E \nearrow P = T$
 3  IF (is-hole($T_{\Theta}$))
 4    IF (shape-match($T_E$, $P$, $\Theta$))
 5      IF (head($\Theta$) $\neq$ [ ])
 6        slices-template(..., $\Theta^{\texttt{target-type}}$)
 7      ELSE $\{\ \langle v_{\texttt{TSIG}},\ (P, \Gamma \vdash^{w} E : T),\ \Theta\rangle\ \}$
 8    ELSE
 9      normalize($\Theta$, $T_E$, fv($T_E$)) = $\Theta''$
10      $\{\ \langle v_{\texttt{ADAPT}},\ (P, \Gamma \vdash^{w} E : T),\ \Theta\rangle\ \} \cup$
11      ( IF (head($\Theta''$) $\neq$ [ ])
12        slices-template(..., $\Theta^{\texttt{target-type}}$)
13       ELSE $\{\ \langle v_{\texttt{TSIG}},\ (P, \Gamma \vdash^{w} E : T),\ \Theta''\rangle\ \}$ )
14  ELSE $\{\ \langle v_{\texttt{PT}},\ (P, \Gamma \vdash^{w} E : T),\ \Theta\rangle\ \}$
```

Figure 3.12: A generalization of the SLICES algorithm in the form of a `slices-template` function of type $((P, \Gamma \vdash^{w} E :\ T), \Theta) \rightarrow \overline{v_3}$. The grayed-out analysis is provided only for illustration purposes, since it represents rule-specific analysis dependent upon the types inferred in the premises and the provided *TypeFocus* instance.

If the type inference rule involves the $T \nearrow P$ adaptation we have to check if the type that resulted from the operation is the source of the target type. The adaptation operation only synthesizes types if the type does not match structurally the prototype. To determine that we use the previously defined `shape-match` function.
If the application of `shape-match` returns `false` (line 4), it implies that the type of the term has been synthesized during the $\nearrow$ adaptation (line 2), and we return an Adaptation Typing Slice to indicate the source of the target type (line 10). As discussed in the case of the SLICES$_{\texttt{app}}$ and SLICES$_{\texttt{var}}$ functions, the steps for analyzing the synthesis of the target type from the term (lines 5 – 7 and 11 – 13) are exactly the same, irrespective of the adaptation, modulo the value of the used *TypeFocus*.

The template indicates that analyzing the role of the propagated prototype is independent from the premises of the inference rules and has to be checked for first. However, as visible in parts between lines 5 – 7 and 11 – 13, the analysis of how the target type was synthesized from the term is rule-dependent. The template only hints that such analysis is dependent upon the type selection of the *head* of the provided *TypeFocus*, as we have illustrated on the concrete examples in the previous section. If the source of the target type is determined to come from one of the premises of the type inference rule, then a recursive call to the algorithm is triggered (lines 6 and 12) with a *TypeFocus* ($\Theta^{\texttt{target-type}}$) that reflects the target type information in the inferred type of the premise.

The construction of the $\Theta^{\texttt{target-type}}$ *TypeFocus* is again rule-dependent but it typically involves either appending the type selector to the provided *TypeFocus* value or a *TypeFocus* decomposition. The *TypeFocus* reflects how the type information is propagated from the conclusions to the premises of the type inference rules, irrespective of the concrete types found in the type derivation trees.

3.5.3 A complete algorithm

In this section we present a complete algorithm which provides analysis steps for every inference rule of the Colored Local Type Inference formalization from [Odersky et al., 2001, pg. 11]. For presentation reasons the algorithm had to be separated into 4 parts:

- Figure 3.13 describes the analysis of the inference rules `(var)`, $(\texttt{abs}_{tp,?})$, $(\texttt{abs}_{tp,\top})$ and $(\texttt{abs}_{tp})$.
- Figure 3.14 describes the analysis of the inference rules `(abs)`, $(\texttt{app}_{tp})$ and $(\texttt{app}_{\bot})$.
- Figure 3.15 describes the analysis of the inference rules `(app)`, $(\texttt{app}_{\bot})$ and `(sel)`.
- Figure 3.16 describes the analysis of the inference rules $(\texttt{rec}_{?})$, $(\texttt{rec}_{\top})$ and `(rec)`.

The algorithm applies the analysis template presented in Section 3.5.3 to each of the inference rules. Crucially, each of the rules provides the rule-specific logic on how the target type could be synthesized from the underlying term and uses the result of the Canonical Forms lemma to guide the analysis using the provided *TypeFocus* value.
Some of the rules (visually) differ from the provided generalization, even when identifying Adaptation Typing Slices and Prototype Typing Slices that we classified as being rule- independent. This is because, when possible, we have simplified the formalization of the algorithm without comprising its integrity, to provide a more succinct definition.

For example, only a small number of rules involves the $\nearrow$ adaptation, therefore we ignore the check when the operation is not present in the rule. Rules $(\texttt{abs}_{tp,?})$, $(\texttt{abs}_{tp,\top})$, `(abs)`, `(sel)`, `(rec)`, $(\texttt{rec}_{\top})$ and `(rec)` are prime examples of such simplification.

Furthermore, corner case rules of the inference, such as $(\texttt{abs}_{tp,\top})$ and $(\texttt{rec}_{\top})$, return immediately with the appropriate Typing Slices since their prototype, $\top$, will never involve the wildcard constant types.

```
FUNCTION SLICES$_{(\mathtt{var})}$ ( ($P$, $\Gamma \vdash^{w} x : \Gamma(x) \nearrow P$), $\Theta$ ) =
$\boxed{(\mathtt{var})\ P,\ \Gamma \vdash^{w} x :\ \Gamma(x) \nearrow P}$
$\Theta(P)_{\mathtt{tpe}} = P_{\Theta}$
IF ($\mathtt{is\text{-}hole}(P_{\Theta})$)
  IF ($\mathtt{shape\text{-}match}(\Gamma(x),\ P,\ \Theta)$)  $\{\ \langle v_{\mathtt{TSIG}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
  ELSE
    $\mathtt{normalize}(\Theta,\ \Gamma(x),\ \mathtt{fv}(\Gamma(x))) = \Theta''$
    $\{\ \langle v_{\mathtt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle,\ \langle v_{\mathtt{TSIG}}, \vdash^{w}_{*}, \Theta'' \rangle\ \}$
ELSE $\{\ \langle v_{\mathtt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$

FUNCTION SLICES$_{(\mathtt{abs}_{tp,?})}$ $\big($ (?, $\Gamma \vdash^{w}$ $\mathbf{fun}[\overline{a}](x : T)E : \forall \overline{a}.T \to S$), $\Theta$ $\big)$ =
$\boxed{(\mathrm{abs}_{tp,?})\ \dfrac{?,\ \Gamma, \overline{a}, x : T \vdash^{w} E : S}{?,\ \Gamma \vdash^{w} \mathbf{fun}[\overline{a}](x : T)E : \forall \overline{a}.T \to S}}$
IF ($\mathtt{head}(\Theta)\ \neq\ [\phi_{\mathtt{fun\text{-}res}}]$)   $\{\ \langle v_{\mathtt{TSIG}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
ELSE                              SLICES( (?, $\Gamma, \overline{a}, x : T \vdash^{w} E :\ S$), $\mathtt{tail}(\Theta)$)

FUNCTION SLICES$_{(\mathtt{abs}_{tp,\top})}$ $\big($ ($\top$, $\Gamma \vdash^{w}$ $\mathbf{fun}[\overline{a}](x : T)E : \top$), $\Theta$ $\big)$ =
$\boxed{(\mathrm{abs}_{tp,\top})\ \dfrac{\top,\ \Gamma, \overline{a}, x : T \vdash^{w} E : S}{\top,\ \Gamma \vdash^{w} \mathbf{fun}[\overline{a}](x : T)E : \top}}$
    $\{\ \langle v_{\mathtt{PT}},\ \vdash^{w}_{*},\ [\,] \rangle\ \}$

FUNCTION SLICES$_{(\mathtt{abs}_{tp})}$ $\big($ ($\forall \overline{a}.P \to P'$, $\Gamma \vdash^{w}$ $\mathbf{fun}[\overline{a}](x : T)E : \forall \overline{a}.T \to S \nearrow \forall \overline{a}.P \to P'$), $\Theta$ $\big)$ =
$\boxed{(\mathrm{abs}_{tp})\ \dfrac{P,\ \Gamma, \overline{a}, x : T \vdash^{w} E : S}{\forall \overline{a}.P \to P',\ \Gamma \vdash^{w} \mathbf{fun}[\overline{a}](x : T)E : \forall \overline{a}.T \to S \nearrow \forall \overline{a}.P \to P'}}$
$\Theta(\forall \overline{a}.P \to P')_{\mathtt{tpe}} = P_{\Theta}$
IF ($\mathtt{is\text{-}hole}(P_{\Theta})$)
  IF ($\mathtt{shape\text{-}match}(\forall \overline{a}.T \to S, \forall \overline{a}.P \to P', \Theta)$)
    IF ($\mathtt{head}(\Theta)\ \neq\ [\phi_{\mathtt{fun\text{-}res}}]$)  $\{\ \langle v_{\mathtt{TSIG}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
    ELSE                            SLICES( ($P'$, $\Gamma, \overline{a}, x : T \vdash^{w} E :\ S$), $\mathtt{tail}(\Theta)$)
  ELSE
    $\mathtt{normalize}(\Theta,\ \forall \overline{a}.T \to S, \overline{a}) = \Theta''$
    $\{\ \langle v_{\mathtt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \} \cup$
    $\Big($ IF ($\mathtt{head}(\Theta'')\ \neq\ [\phi_{\mathtt{fun\text{-}res}}]$)  $\{\ \langle v_{\mathtt{TSIG}},\ \vdash^{w}_{*},\ \Theta'' \rangle\ \}$
      ELSE                         SLICES( ($P'$, $\Gamma, \overline{a}, x : T \vdash^{w} E :\ S$), $\mathtt{tail}(\Theta'')$) $\Big)$
ELSE $\{\ \langle v_{\mathtt{PT}}, \vdash^{w}_{*}, \Theta \rangle\ \}$
```

Figure 3.13: *(Part 1)* Algorithm for locating typing decisions that infer types. The algorithm is realized through the SLICES function of type ((P, $\Gamma \vdash^{w} E :\ T$), Θ) $\to \overline{v_3}$.

FUNCTION SLICES$_{(\mathtt{abs})}$ $\big((\forall\overline{a}.T \to P,\ \Gamma \vdash^{w} \mathbf{fun}(x)E : \forall\overline{a}.T \to S),\ \Theta \big) =$

$$\boxed{(\text{abs})\ \frac{P, \Gamma, \overline{a}, x : T \vdash^{w} E : S}{\forall\overline{a}.T \to P, \Gamma \vdash^{w} \mathbf{fun}(x)E : \forall\overline{a}.T \to S}}$$

```
$\Theta(\forall\overline{a}.T \to P)_{\mathtt{tpe}} = P_\Theta$
IF ($\mathtt{is\text{-}hole}(P_\Theta)$)        $slices((P,\ \Gamma, \overline{a}, x : T \vdash^{w} E :\ S),\ \mathtt{tail}(\Theta))$
ELSE                            $\{\ \langle \nu_{\mathsf{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

FUNCTION SLICES$_{(\mathtt{app}_{tp})}$ $\Big((P,\ \Gamma \vdash^{w} F \Big[\overline{R}\Big] (E) : [\overline{R}/\overline{a}]\ T \nearrow P), \Theta \Big) =$

$$\boxed{(\text{app}_{tp})\ \frac{?,\ \Gamma, \vdash^{w} F : \forall\overline{a}.S \to T \qquad [\overline{R}/\overline{a}]\ S,\ \Gamma \vdash^{w} E : [\overline{R}/\overline{a}]\ S}{P, \Gamma \vdash^{w} F \Big[\overline{R}\Big] (E) : [\overline{R}/\overline{a}]\ T \nearrow P}}$$

```
$\Theta(P)_{\mathtt{tpe}} = P_\Theta$
IF ($\mathtt{is\text{-}hole}(P_\Theta)$)
  IF ($\mathtt{shape\text{-}match}([\overline{R}/\overline{a}]\ T,\ P,\ \Theta)$)
    $\Theta(T)_{\mathtt{tpe}} = T_\Theta$
    IF ($\mathtt{is\text{-}tvar}(T_\Theta, \overline{a})$)   $\{\ \langle \nu_{\mathsf{TVAR}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
    ELSE                       SLICES( $(?,\ \Gamma \vdash^{w} F : \forall\overline{a}.S \to T),\ \phi_{\mathtt{fun\text{-}res}} :: \Theta$)
  ELSE
    $\mathtt{normalize}(\Theta, (\ \overline{R}/\overline{a}\ )\ T, \overline{a}) = \Theta''$
    $\Theta''(T)_{\mathtt{tpe}} = T'_\Theta$
    $\{\ \langle \nu_{\mathsf{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \} \cup$
    $\Big($ IF ($\mathtt{is\text{-}tvar}(T'_\Theta, \overline{a})$)   $\{\ \langle \nu_{\mathsf{TVAR}},\ \vdash^{w}_{*},\ \Theta'' \rangle\ \}$
         ELSE                     SLICES( $(?,\ \Gamma \vdash^{w} F : \forall\overline{a}.S \to T),\ \phi_{\mathtt{fun\text{-}res}} :: \Theta''$) $\Big)$
ELSE $\{\ \langle \nu_{\mathsf{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

FUNCTION SLICES$_{(\mathtt{app}_{tp,\bot})}$ $\Big((P,\ \Gamma \vdash^{w} F \Big[\overline{R}\Big] (E) : \bot \nearrow P), \Theta \Big) =$

$$\boxed{(\text{app}_{tp,\bot})\ \frac{?,\ \Gamma, \vdash^{w} F : \bot \quad \top, \Gamma \vdash^{w} E : S}{P,\ \Gamma \vdash^{w} F \Big[\overline{R}\Big] (E) : \bot \nearrow P}}$$

```
$\Theta(P)_{\mathtt{tpe}} = P_\Theta$
IF ($\mathtt{is\text{-}hole}(P_\Theta)$)
  IF ($P \neq\ ?$)   $\{\ \langle \nu_{\mathsf{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$ $\cup$ SLICES($(?,\ \Gamma \vdash^{w} F : \bot),\ [\,]$)
  ELSE          SLICES($(?,\ \Gamma \vdash^{w} F : \bot),\ \Theta$)
ELSE $\{\ \langle \nu_{\mathsf{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

Figure 3.14: *(Part 2)* Algorithm for locating typing decisions that infer types. The algorithm is realized through the SLICES function of type $(\ (P,\ \Gamma \vdash^{w} E :\ T),\ \Theta\ ) \to \overline{\nu_3}$.

FUNCTION SLICES$_{\texttt{(app)}}$ $\big(\,(P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T}\, T \nearrow P),\ \Theta\,\big) =$

$$\boxed{(\text{app})\ \dfrac{?,\ \Gamma, \vdash^{w} F : \forall\overline{a}.S \rightarrow T \quad [?/\overline{a}]\, S, \Gamma \vdash^{w} E : S' \quad \begin{array}{ll} \vdash_a S' <: S & \Rightarrow C_1 \\ \vdash_a T <: \top \searrow P & \Rightarrow C_2 \end{array}}{P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T}\, T \nearrow P}}$$

```
$\Theta(P)_{\texttt{tpe}} = P_\Theta$
IF (is-hole($P_\Theta$))
  IF (shape-match($\sigma_{C_1 \cup C_2, T} T,\ P,\ \Theta$))
    $\Theta(T)_{\texttt{tpe}} = T_\Theta$
    IF (is-tvar($T_\Theta, \overline{a}$))   $\{\ \langle v_{\texttt{TVAR}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
    ELSE                      $slices(\ (?, \Gamma, \vdash^{w} F : \forall\overline{a}.S \rightarrow T),\ \phi_{\texttt{fun-res}} :: \Theta)$
  ELSE
    normalize($\Theta,\ \sigma_{C_1 \cup C_2, T} T,\ \emptyset$) $= \Theta''$
    $\Theta''(T)_{\texttt{tpe}} = T_\Theta$
    $\{\ \langle v_{\texttt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \} \cup$
      $\Big($ IF (is-tvar($T_\Theta, \overline{a}$))   $\{\ \langle v_{\texttt{TVAR}},\ \vdash^{w}_{*},\ \Theta'' \rangle\ \}$
        ELSE                      $slices(\ (?,\ \Gamma \vdash^{w} F : \forall\overline{a}.S \rightarrow T),\ \phi_{\texttt{fun-res}} :: \Theta''))$ $\Big)$
ELSE $\{\ \langle v_{\texttt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

FUNCTION SLICES$_{\texttt{(app}_\bot\texttt{)}}$ $(\,(P,\ \Gamma \vdash^{w} F(E) : \bot \nearrow P), \Theta\,) =$

$$\boxed{(\text{app}_\bot)\ \dfrac{?,\ \Gamma, \vdash^{w} F : \bot \quad \top, \Gamma \vdash^{w} E : S}{P,\ \Gamma \vdash^{w} F(E) : \bot \nearrow P}}$$

```
$\Theta(P)_{\texttt{tpe}} = P_\Theta$
IF (is-hole($P_\Theta$))
  IF ($P \neq ?$))   $\{\ \langle v_{\texttt{ADAPT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \} \cup$ SLICES($(?,\ \Gamma \vdash^{w} F : \bot),\ [\,]$)
  ELSE              SLICES($(?,\ \Gamma \vdash^{w} F : \bot),\ \Theta$)
ELSE $\{\ \langle v_{\texttt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

FUNCTION SLICES$_{\texttt{(sel)}}$ $(\,(P,\ \Gamma \vdash^{w} F.x : T), \Theta\,) =$

$$\boxed{(\text{sel})\ \dfrac{\{x : P\},\ \Gamma, \vdash^{w} F : \{x : T\}}{P,\ \Gamma \vdash^{w} F.x : T}}$$

```
$\Theta(P)_{\texttt{tpe}} = P_\Theta$
IF (is-hole($P_\Theta$))   SLICES( $(\{x : P\},\ \Gamma \vdash^{w} F : \{x : T\}),\ \phi_{\texttt{sel}_x} :: \Theta$ )
ELSE                     $\{\ \langle v_{\texttt{PT}},\ \vdash^{w}_{*},\ \Theta \rangle\ \}$
```

Figure 3.15: *(Part 3)* Algorithm for locating typing decisions that infer types. The algorithm is realized through the SLICES function of type $(\,(P,\ \Gamma \vdash^{w} E :\ T),\ \Theta\,) \rightarrow \overline{v_3}$.

FUNCTION SLICES$_{(\texttt{rec}_?)}$ $(\ (?,\ \Gamma \vdash^w \{x_1 = F_1, \dots, x_n = F_n\} : \{x_1 : T_1, \dots, x_n : T_n\}),\ \Theta\ ) =$

$$\boxed{(\text{rec}_?)\ \dfrac{?,\ \Gamma \vdash^w F_1 : T_1 \ \ \dots\ \ ?,\ \Gamma \vdash^w F_n : T_n}{?,\ \Gamma \vdash^w \{x_1 = F_1,\ \dots,\ x_n = F_n\} : \{x_1 : T_1,\ \dots,\ x_n : T_n\}}}$$

```
IF $(\texttt{head}(\Theta)\ \neq\ [\phi_{\texttt{sel}_{x_k}}])$    $\{\ \langle \nu_{\texttt{TSIG}},\ \vdash^w_*,\ \Theta \rangle\ \}$
ELSE                                    SLICES$((?,\ \Gamma \vdash^w F_k :\ T_k),\ \texttt{tail}(\Theta))$        $1 \leq k \leq n$
```

FUNCTION SLICES$_{(\texttt{rec}_\top)}$ $(\ (\top,\ \Gamma \vdash^w \{x_1 = F_1, \dots, x_n = F_n\} : \top),\ \Theta\ ) =$

$$\boxed{(\text{rec}_\top)\ \dfrac{\top,\ \Gamma \vdash^w F_1 : T_1 \ \ \dots\ \ \top, \Gamma \vdash^w F_n : T_n}{\top,\ \Gamma \vdash^w \{x_1 = F_1,\ \dots,\ x_n = F_n\} : \top}}$$

```
$\{\ \langle \nu_{\texttt{PT}}, \vdash^w_*, [\,] \rangle\ \}$
```

FUNCTION SLICES$_{(\texttt{rec})}$
$(\ (\{x_1 : P_1, \dots, x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1, \dots, x_n = F_n\} : \{x_1 : T_1, \dots, x_m : T_m\}),\ \Theta\ ) =$

$$\boxed{(\text{rec})\ \dfrac{(P_1,\ \Gamma \vdash^w F_1 : T_1) \dots (P_m,\ \Gamma \vdash^w F_m : T_m)\ \ \ (\top,\ \Gamma \vdash^w F_{m+1} : T_{m+1}) \dots (\top,\ \Gamma \vdash^w F_n : T_n)}{\{x_1 : P_1, \dots, x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1, \dots, x_n = F_n\} : \{x_1 : T_1, \dots, x_m : T_m\}}}$$

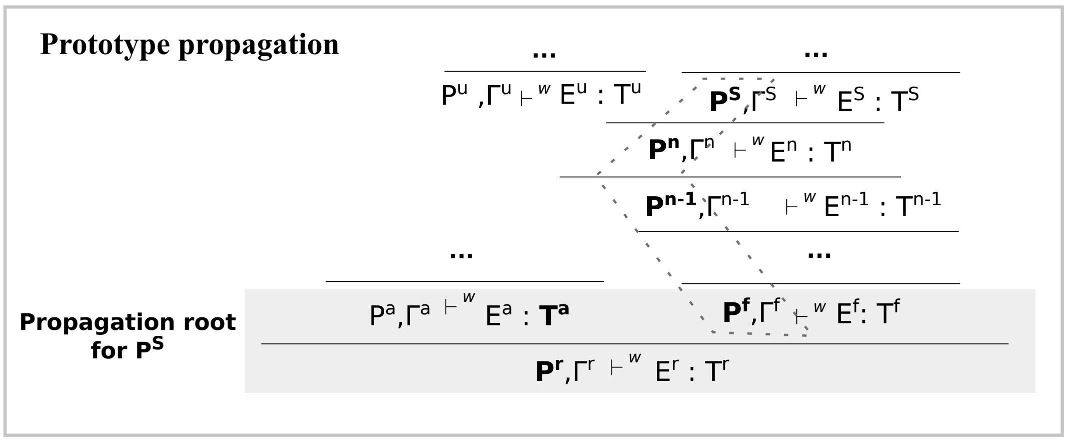
```
$\Theta(\{x_1 : P_1,\ \dots,\ x_m : P_m\})_{\texttt{tpe}} = P_\Theta$
IF $(\texttt{is-hole}(P_\Theta))$
 $\texttt{head}(\Theta)\ =\ [\phi_{\texttt{sel}_{x_k}}]$
 SLICES$((P_k,\ \Gamma \vdash^w F_k :\ T_k),\ \texttt{tail}(\Theta))$        $1 \leq k \leq m$
ELSE $\{\ \langle \nu_{\texttt{PT}},\ \vdash^w_*,\ \Theta \rangle\ \}$
```

Figure 3.16: *(Part 4)* Algorithm for locating typing decisions that infer types. The algorithm is realized through the SLICES function of type $(\ (P,\ \Gamma \vdash^w E :\ T),\ \Theta\ ) \rightarrow \overline{\nu_3}$.

3.6 On understanding the propagation of the expected type

The algorithm for analyzing type derivation trees deliberately returns intermediate *Typing Slices*. Among different kinds of Typing Slices, Prototype and Adaptation are the only ones which specifically recognize the expected type (the prototype) as the source of some target type. The expected type information does not translate to program locations, or makes sense without a lengthy description of the context of the program in general. It is therefore necessary to formulate a technique that can automatically locate the node in the type derivation tree, where the expected type from the Typing Slices is first introduced.

In this section, we show that the problem of finding the source of the inherited type is no harder than finding the *Propagation Root* for the given prototype, the problem that we informally introduced in Section 3.1.2.



Prototype propagation
...
...
$P^u, \Gamma^u \vdash^w E^u : T^u$
$P^s, \Gamma^s \vdash^w E^s : T^s$
$P^n, \Gamma^n \vdash^w E^n : T^n$
$P^{n-1}, \Gamma^{n-1} \vdash^w E^{n-1} : T^{n-1}$
...
...
Propagation root for P^S
$P^a, \Gamma^a \vdash^w E^a : T^a$
$P^f, \Gamma^f \vdash^w E^f : T^f$
$P^r, \Gamma^r \vdash^w E^r : T^r$

Figure 3.17: Elements of prototype propagation for some prototype P^s. The P^s prototype is implicitly propagated in prototypes P^n, P^{n-1}, ..., P^f. The (P^r, $\Gamma^r \vdash^w E^r : T^r$) typing judgment represents the Propagation Root for prototypes P^s, P^n, P^{n-1}, ..., P^f because it introduces in its premise a *fresh* prototype P^f that is not part of the P^r prototype. Being not propagated, the fresh prototype P^f results from the inferred type of one of the premises of the Propagation Root, such as T^a. The dotted premises indicate potentially non-empty premises of the inference rules.

A summary of the possible nodes of the type derivation tree that participate in the propagation of type elements of the inherited prototype P^s is presented in Figure 3.17. The figure serves as a reference point, when discussing typing judgments, their prototypes, and how they were propagated. In the figure prototype P^n includes the P^s prototype, prototype P^{n-1} includes the P^n prototype but prototype P^r does not include prototype P^f. Therefore the (P^r, $\Gamma^r \vdash^w E^r : R^r$) judgment symbolically represents the Propagation Root of the P^s, P^n, P^{n-1}, ..., P^f prototypes and the *fresh* prototype P^f derives from the inferred type (T^a) of one of the other premises of the Propagation Root.

The Prototype (and Adaptation) Typing Slices consist of the $(P^s,\ \Gamma^s \vdash^w E^s : T^s)$ inference judgment and the Θ^{s} *TypeFocus*, where the ^{s} superscript stands for the Typing Slice information. From the definition of those Typing Slices, we can infer that:

- $(\Theta^{\mathsf{s}}, \mathtt{fv}(T^s) \vdash^{\mathsf{WF}} T^s)$ and $\Theta^{\mathsf{s}}(T^s) = \mathtt{inl}\ T^{target}$ for some T^{target}, *i.e.,* the information about the target type, can be extracted from the part of the inferred type T^s.
- $(\varnothing, \mathtt{fv}(P^s) \vdash^{\mathsf{WF}} P^s)$, $\Theta^{\mathsf{s}}(P^s) = \mathtt{inl}\ P^{target}$ for some P^{target}, and $P^{target} \neq ?$, *i.e.,* the target type has been fully inferred from the extracted part of the prototype, P^{target}.

By referring to the elements of the type derivation tree in Figure 3.17, we give an overview of a two-part generic algorithm that relates the prototype P^{target} with its distant source, type T^a:

1. We define the algorithm that traces backwards through the nodes of the type derivation tree (Section 3.6.1). The algorithm walks from the $(P^s,\ \Gamma^s \vdash^w E^s : T^s)$ judgment to the Propagation Root $(P^r,\ \Gamma^r \vdash^w E^r : T^r)$ judgment.
2. We identify the $(P^f,\ \Gamma^f \vdash^w E^f : T^f)$ inference judgment which is a direct premise of the Propagation Root. The prototypes that participate in the propagation of its elements, *i.e.,* from P^f to P^s, define a so called *Prototype Propagation Path* represented in Figure 3.17 through a dotted selection.
3. We reduce the Prototype Propagation Path to a type selection, symbolically defined as $\Theta^{\mathsf{prototype}}$, such that $(\Theta^{\mathsf{prototype}}, \varnothing \vdash^{\mathsf{WF}} P^f)$ and $\Theta^{\mathsf{prototype}}(P^f) = \mathtt{inl}\ P^s$. The reduction is only based on the kind of type inference rules that were used in the judgments.
4. We define the algorithm that locates the origin of P^f in the premises of the Propagation Root (Section 3.6.2).
5. We compose the type selections of Θ^{s} and $\Theta^{\mathsf{prototype}}$ to define a complete algorithm that locates the source of the target prototype in the *Propagation Root* (Section 3.6.3).

We elaborate on each of the steps in the discussion that follows.

3.6.1 Inference of a Propagation Root

The search for the Propagation Root of some prototype is formulated in the propagation judgment

$$\boxed{\Theta^{\mathsf{i}} \vDash^p (P^i,\ \Gamma^i \vdash^w E^i :\ T^i) \rightsquigarrow \langle (P^o,\ \Gamma^o \vdash^w E^o :\ T^o), \Theta^{\mathsf{o}} \rangle \qquad (\textit{propagation judgment})}$$

The ^{i} and ^{o} superscripts are used to distinguish between the input and output inference judgments and *TypeFocus* instances.

The $\vDash^p$ propagation judgment takes a *TypeFocus* instance $(\Theta^{\mathtt{i}})$ and an inference judgment $((P^i,\ \Gamma^i \vdash^w E^i :\ T^i))$, such that $(\Theta^{\mathtt{i}}, \varnothing \vdash^{\mathtt{WF}} P^i)$ and $(\Theta^{\mathtt{i}}(P^i)_{\mathtt{tpe}} \neq\ ?)$. The propagation judgment infers a tuple consisting of the inference judgment $(P^o,\ \Gamma^o \vdash^w E^o :\ T^o)$ and a *TypeFocus* $\Theta^{\mathtt{o}}$. The returned inference judgment represents the Propagation Root of P^i and the returned *TypeFocus* $(\Theta^{\mathtt{o}})$ reduces the Prototype Propagation Path of P^i to a type selection, such that $\Theta^{\mathtt{o}}(P^f)_{\mathtt{tpe}}\ ==\ \Theta^{\mathtt{i}}(P^i)_{\mathtt{tpe}}$.

The $\vDash^p$ propagation judgment is defined using the set of recursive declarative rules in Figure 3.18. The algorithm recursively traces backwards through the type derivation tree until it reaches a *Propagation Root* for the corresponding prototype. That is why the rules that realize the propagation judgment divide into two groups - those that backtrack through the type derivation tree, and those that stop it and return the collected information. For space reasons, the definition uses the $\vdash^w_{\mathtt{premise}}$ and $\vdash^w_{\mathtt{parent}}$ notation for representing the input inference judgment, $(P^i,\ \Gamma^i \vdash^w E^i : T^i)$, and its parent, respectively.

The first 5 rules - $\mathtt{Prop}_{abs}$, $\mathtt{Prop}_{abs_{tp}}$, $\mathtt{Prop}_{abs_{tp,\top}}$, $\mathtt{Prop}_{rec_{\top}}$, and $\mathtt{Prop}_{rec_1}$ - identify type inference rules which only propagate prototype information from the conclusion of the rule to one of its premises. The two important elements shared by each of the mentioned rules involve:

- Recursively invoking the propagation judgment to backtrack through the type derivation tree. We navigate the type derivation tree backwards by referring to the parent of the judgment $(\vdash^w_{\mathtt{premise}}\downarrow)$.
- Reducing the prototype propagation of a single type inference rule to a type selection. For example, in rule $\mathtt{Prop}_{abs_{tp}}$ the prototype propagation that takes place in the $(\mathtt{abs}_{tp})$ type inference rule (P' in $\forall\overline{a}.P \to P'$) is reduced to an equivalent type selection, *i.e.,* if the type selection on any prototype P' is $\Theta^{\mathtt{i}}$, then the type selection on the prototype P' that is part of the $\forall\overline{a}.P \to P'$ type is $(\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{i}})$.

The remaining 6 rules - $\mathtt{Prop}_{app}$, $\mathtt{Prop}_{app_{\perp}}$, $\mathtt{Prop}_{app_{tp}}$, $\mathtt{Prop}_{app_{tp,\perp}}$, $\mathtt{Prop}_{\mathtt{sel}}$ and $\mathtt{Prop}_{rec_2}$ - identify type inference rules where the direct prototype propagation between the rule's conclusion and the premise does not take place. Judgments involving such type inference rules are what we formally define as Propagation Roots. We call such instances of type inference rules *Propagation Root*s.

Rule $(\mathtt{rec})$ of Colored Local Type Inference leads to two distinct $\mathtt{Prop}$ rules, $\mathtt{Prop}_{rec_1}$ and $\mathtt{Prop}_{rec_2}$. The difference stems from how the prototype of the conclusion of the type inference rule, $\{x_1 : P_1,\ \ldots,\ x_m : P_m\}$, is used to infer the type of the record member. Depending on which premise of the rule we backtrack from, the prototype used in the premise has been either simply propagated from the rule's conclusion (P_k where $1 \leq k \leq m$) or synthesized ($\top$

$$\texttt{Prop}_{abs}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(abs)}\\ \phi_{\texttt{fun-res}} :: \Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{parent}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle}$$

$$\texttt{Prop}_{abs_{tp}}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(abs}_{tp}\texttt{)}\\ \phi_{\texttt{fun-res}} :: \Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{parent}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle}$$

$$\texttt{Prop}_{abs_{tp,\top}}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(abs}_{tp,\top}\texttt{)}\\ \Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{parent}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle}
\qquad
\texttt{Prop}_{rec_{\top}}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(rec}_{\top}\texttt{)}\\ \Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{parent}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle}$$

$$\texttt{Prop}_{rec_1}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(rec)}\ \{x_1 : P_1, ..., x_m : P_m\},\ \Gamma \vdash^{w} \{x_1 = F_1, ..., x_n = F_n\} : \{x_1 : T_1, ..., x_m : T_m\}\\ (\vdash^{w}_{\texttt{premise}}) = (P_k,\ \Gamma \vdash^{w} F_k : T_k) \qquad \text{for } 1 \le k \le m\\ \phi_{\texttt{sel}_{x_k}} :: \Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{parent}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} : T^{o}), \Theta^{o}\rangle}$$

$$\texttt{Prop}_{app}\;\frac{(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(app)}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(app)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}
\qquad
\texttt{Prop}_{app_{\bot}}\;\frac{(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(app}_{\bot}\texttt{)}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(app}_{\bot}\texttt{)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}$$

$$\texttt{Prop}_{app_{tp}}\;\frac{(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(app}_{tp}\texttt{)}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(app}_{tp}\texttt{)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}
\qquad
\texttt{Prop}_{app_{tp,\bot}}\;\frac{(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(app}_{tp,\bot}\texttt{)}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(app}_{tp,\bot}\texttt{)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}$$

$$\texttt{Prop}_{\texttt{sel}}\;\frac{(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(sel)}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(sel)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}$$

$$\texttt{Prop}_{rec_2}\;\frac{\begin{array}{l}(\vdash^{w}_{\texttt{premise}})\downarrow \;=\; (\vdash^{w}_{\texttt{parent}}) \;=\; \texttt{(rec)}\ \{x_1 : P_1, ..., x_m : P_m\},\ \Gamma \vdash^{w} \{x_1 = F_1, ..., x_n = F_n\} : \{x_1 : T_1, ..., x_m : T_m\}\\ (\vdash^{w}_{\texttt{premise}}) = (\top,\ \Gamma \vdash^{w} F_k : T_k) \qquad \text{for } m < k \le n\end{array}}{\Theta^{\mathrm{i}} \models^{p} (\vdash^{w}_{\texttt{premise}}) \rightsquigarrow \left\langle \texttt{(rec)}\ (\vdash^{w}_{\texttt{parent}}), \Theta^{\mathrm{i}}\right\rangle}$$

Figure 3.18: Definition of a $\Theta^{\mathrm{i}} \models^{p} (P^{i},\ \Gamma^{i} \vdash^{w} E^{i} :\ T^{i})) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w} E^{o} :\ T^{o}), \Theta^{\mathrm{o}}\rangle$ judgment using the recursive `Prop` rules. The algorithm is driven by the kind of the last type inference rule used in the parent of the input inference judgment, *i.e.,* $(\vdash^{w}_{\texttt{premise}})\downarrow$.

for $m < k \le n$).

The `(sel)` type inference rule is also classified as a *Propagation Root*. We notice that, even though the prototype P from the conclusion of the type inference rule appears in the premise's prototype in $\{x :\ P\}$, the record prototype itself is introduced for the first time in the `(sel)` rule.

For completeness, Figure 3.19 realizes the $\models^{p}$ judgment in an algorithmic fashion. The `PrototypeBacktrack` function, identically to the deductive rules of the $\models^{p}$ judgment, takes an input *TypeFocus* and a type inference judgment, and returns the Propagation Root of the latter and the Prototype Propagation Path reduced into a *TypeFocus* value. The algorithm traces

```
FUNCTION PrototypeBacktrack($\Theta^{\mathtt{i}}$, $(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)$) =
  $(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)\downarrow\ =\ \vdash^w_{\mathtt{parent}}$
  MATCH $(\vdash^w_{\mathtt{parent}})$ OF
    CASE (abs) :        PrototypeBacktrack($\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{i}}$, $\vdash^w_{\mathtt{parent}}$)
    CASE (abs$_{tp}$) :      PrototypeBacktrack($\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{i}}$, $\vdash^w_{\mathtt{parent}}$)
    CASE (abs$_{tp,\top}$) :   PrototypeBacktrack($\Theta^{\mathtt{i}}$, $\vdash^w_{\mathtt{parent}}$)
    CASE (rec$_\top$) :      PrototypeBacktrack($\Theta^{\mathtt{i}}$, $\vdash^w_{\mathtt{parent}}$)
    CASE (rec) :
      $(\vdash^w_{\mathtt{parent}})$ == (rec) $\{x_1 : P_1,...,x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1,...,x_n = F_n\} : \{x_1 : T_1,...,x_m : T_m\}$
      $(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)$ == $(P_k,\ \Gamma \vdash^w F_k : T_k)$
      IF $(1 \le k \le m)$  PrototypeBacktrack($\phi_{\mathtt{sel}_{x_k}} :: \Theta^{\mathtt{i}}$, $\vdash^w_{\mathtt{parent}}$)
      ELSE             $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (app) :         $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (app$_\bot$) :        $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (app$_{tp}$) :        $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (app$_{tp,\bot}$) :     $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (sel) :         $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
```

Figure 3.19: The algorithmic definition of the $\Theta^{\mathtt{i}} \vDash^p (P^i,\ \Gamma^i \vdash^w E^i :\ T^i)) \rightsquigarrow \langle (P^o,\ \Gamma^o \vdash^w E^o :\ T^o), \Theta^{\mathtt{o}}\rangle$ judgment from Figure 3.18. The algorithm pattern matches on the kind of the last type inference rule used in the parent of the input inference judgment. Pattern matching distinguish between the prototype propagation and the introduction of the *fresh* prototype value.

backwards through the nodes of the type by pattern matching on all of the possible type inference rules that do not require a non-wildcard prototype. Identically to the deduction rules of Figure 3.18 the algorithmic definition distinguishes between the rules that only propagate prototype information and the rules that serve as Propagation Roots.

3.6.2 Analysis of a Propagation Root

The `Prop` rules that realize the propagation judgment (Figure 3.18) identify 6 type inference rules that can serve as a Propagation Root for a prototype. The typing decisions that infer the fresh prototype P^f in those rules may differ significantly. That is why in this section we define a `slicesPtRoot` function that takes any Propagation Root and finds the source of the fresh prototype based on the typing decisions of the formal type inference rules.

The `slicesPtRoot` partial function, of type $(\ (P,\ \Gamma \vdash^w E :\ T),\ \Theta\ ) \rightarrow \overline{v_3}$ takes an inference judgment representing the Propagation Root of some prototype, say P', and a *TypeFocus*, say Θ, such that $(\Theta, \epsilon \vdash^{\mathtt{WF}} P^f)$ and $\Theta(P^f) = \mathtt{inl}\ P'$. The function returns a sequence of Typing Slices explaining the source of the prototype P'.

The `slicesPtRoot` function is realized through a set of rule-specific functions defined in Figure 3.20. Each type inference rule is considered separately, as indicated through the *rule* subscript in the function name $\texttt{slicesPtRoot}_{rule}$. For clarity, Figure 3.20 highlights the position of P^f, that is inference rule-specific, with gray boxes.

Having presented the purpose of the `slicesPtRoot` function we will now delve into the details of each of the type inference rule of Colored Local Type Inference that can serve as the Propagation Root for some prototype.

The `(app)` *type inference rule*

The $\texttt{slicesPtRoot}_{\texttt{(app)}}$ function finds the source of the highlighted prototype $([^?/_{\overline{a}}]\, S)$ used in the inference of the type of the argument. The type inference rule states that the non-wildcard elements of the used prototype can only come from the inferred type of the function. Therefore in order to locate the source of the prototype we delegate to the established *TypeFocus*-based analysis from Section 3.5.3.

We recall that in order to trigger the *TypeFocus*-based analysis we have to provide a type selection that is well-formed type selection with respect to the inferred type of the function. The *TypeFocus* used in the $\texttt{slicesPtRoot}_{\texttt{(app)}}$ function, $\phi_{\texttt{fun-param}} :: \Theta$, satisfies that condition because: $(\Theta, \emptyset \vdash^{\texttt{WF}} [^?/_{\overline{a}}]\, S)$ (from the precondition of the `slicesPtRoot` function) implies $(\Theta, \texttt{fv}([^?/_{\overline{a}}]\, S) \vdash^{\texttt{WF}} [^?/_{\overline{a}}]\, S)$, and, by Lemma 3.8, $(\Theta, \overline{a} \vdash^{\texttt{WF}} S)$ and $(\phi_{\texttt{fun-param}} :: \Theta, \overline{a} \vdash^{\texttt{WF}} \forall \overline{a}.S \to T)$.

The $(\texttt{app}_{tp})$ *type inference rule*

The $\texttt{slicesPtRoot}_{(\texttt{app}_{tp})}$ function analyses the inference of a type of a function application using a similar approach as in the case of the `(app)` rule, except that it also has to take into account the presence of the explicit type arguments.

By a similar argument as in the previous case, $\phi_{\texttt{fun-param}} :: \Theta$ defines a type selection well-formed with respect to the inferred type of the function because: $(\Theta, \emptyset \vdash^{\texttt{WF}} [^{\overline{R}}/_{\overline{a}}]\, S)$ (from the precondition of the `slicesPtRoot` function) implies $(\Theta, \texttt{fv}([^{\overline{R}}/_{\overline{a}}]\, S) \vdash^{\texttt{WF}} [^{\overline{R}}/_{\overline{a}}]\, S)$, and, by Lemma 3.8, $(\phi_{\texttt{fun-param}} :: \Theta, \overline{a} \vdash^{\texttt{WF}} S)$, and $(\phi_{\texttt{fun-param}} :: \Theta, \overline{a} \vdash^{\texttt{WF}} \forall \overline{a}.S \to T)$.

In contrast to the function application with elided type arguments $([^?/_{\overline{a}}]\, S)$, we have to take into account the possibility that the fresh prototype could involve one of the explicit type arguments $([^{\overline{R}}/_{\overline{a}}]\, S)$. The reconstructed type selection is sufficient to distinguish between the two possible sources of the prototype:

- $(\phi_{\texttt{fun-param}} :: \Theta)(\forall \overline{a}.S \to T)_{\texttt{tpe}} = a$ and $a \in \overline{a}$:
  The extraction of the type variable a implies that the explicit type argument is the source of the prototype represented by the Θ *TypeFocus*. Since type arguments defines

FUNCTION `slicesPtRoot`$_{\texttt{(app)}}$ ((app), Θ) =

$$\text{(app)}\ \frac{?,\ \Gamma, \vdash^w F : \forall\overline{a}.S \to T \qquad [?/\overline{a}]\,S, \Gamma \vdash^w E : S' \qquad \begin{array}{ll} \vdash_a S' <: S & \Rightarrow C_1 \\ \vdash_a T <: \top \searrow P & \Rightarrow C_2 \end{array}}{P,\ \Gamma \vdash^w F(E) : \sigma_{C_1 \cup C_2, T}\, T \nearrow P}$$

```
SLICES(($?,\ \Gamma \vdash^w F:\ \forall\overline{a}.S \to T$), ($\phi_{\texttt{fun-param}}$ :: $\Theta$))
```

FUNCTION `slicesPtRoot`$_{(\texttt{app}_\perp)}$ ((app$_\perp$), Θ) =

$$\text{(app}_\perp\text{)}\ \frac{?,\ \Gamma, \vdash^w F : \perp \qquad \top, \Gamma \vdash^w E : S}{P,\ \Gamma \vdash^w F(E) : \perp \nearrow P}$$

```
SLICES(($?,\ \Gamma \vdash^w F:\ \perp$), [ ])
```

FUNCTION `slicesPtRoot`$_{(\texttt{app}_{tp})}$ ((app$_{tp}$), Θ) =

$$\text{(app}_{tp}\text{)}\ \frac{?,\ \Gamma, \vdash^w F : \forall\overline{a}.S \to T \qquad [\overline{R}/\overline{a}]\,S,\ \Gamma \vdash^w E : [\overline{R}/\overline{a}]\,S}{P,\ \Gamma \vdash^w F\left[\overline{R}\right](E) : [\overline{R}/\overline{a}]\,T \nearrow P}$$

```
$\Theta^{\texttt{cont}} = \phi_{\texttt{fun-param}}$ :: $\Theta$
IF (is-tvar($\Theta^{\texttt{cont}}(S)_{\texttt{tpe}}, \overline{a}$))   { ⟨$\nu_{\texttt{TVAR}}$, ($?,\ \Gamma \vdash^w F:\ \forall\overline{a}.S \to T$), $\Theta^{\texttt{cont}}$⟩ }
ELSE                                      SLICES(($?,\ \Gamma \vdash^w F:\ \forall\overline{a}.S \to T$), $\Theta^{\texttt{cont}}$)
```

FUNCTION `slicesPtRoot`$_{(\texttt{app}_{tp,\perp})}$ ((app$_{tp,\perp}$), Θ) =

$$\text{(app}_{tp,\perp}\text{)}\ \frac{?,\ \Gamma, \vdash^w F : \perp \qquad \top, \Gamma \vdash^w E : S}{P,\ \Gamma \vdash^w F\left[\overline{R}\right](E) : \perp \nearrow P}$$

```
SLICES(($?,\ \Gamma \vdash^w F:\ \perp$), [ ])
```

FUNCTION `slicesPtRoot`$_{\texttt{(sel)}}$ ((sel), Θ) =

$$\text{(sel)}\ \frac{\{x : P\},\ \Gamma, \vdash^w F : \{x : T\}}{P,\ \Gamma \vdash^w F.x : T}$$

```
IF (head($\Theta$) == [$\phi_{\texttt{sel}_x}$])   { ⟨$\nu_{\texttt{PT}}$, ($P,\ \Gamma \vdash^w F.x:\ T$), tail($\Theta$)⟩ }
ELSE                        { ⟨$\nu_{\texttt{TSIG}}$, ($\{x : P\},\ \Gamma \vdash^w F:\ \{x : T\}$), $\Theta$⟩ }
```

FUNCTION `slicesPtRoot`$_{\texttt{(rec)}}$ ((rec), Θ) =

$$\text{(rec)}\ \frac{(P_1,\ \Gamma \vdash^w F_1 :\ T_1)\ \ldots\ (P_m,\ \Gamma \vdash^w F_m :\ T_m) \quad (\top,\ \Gamma \vdash^w F_{m+1} :\ T_{m+1})\ \ldots\ (\top,\ \Gamma \vdash^w F_n :\ T_n)}{\{x_1 : P_1,\ \ldots,\ x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1,\ \ldots,\ x_n = F_n\} : \{x_1 : T_1,\ \ldots,\ x_m : T_m\}}$$

```
{ ⟨$\nu_{\texttt{PT}}$, ($\{x_1 : P_1, \ldots, x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1, \ldots, x_n = F_n\} : \{x_1 : T_1, \ldots, x_m : T_m\}$), [ ]⟩ }
```

Figure 3.20: Algorithm for finding the source of the fresh prototype P^{fresh} that is introduced in the Propagation Root judgments. The algorithm is realized through the `slicesPtRoot` function of type (($P,\ \Gamma \vdash^w E:\ T$), Θ) $\to \overline{\nu_3}$ which analyzes typing decisions of every type inference rule that can serve as a Propagation Root. The gray boxes highlight the position of the fresh P^f prototype source of which the function explains.

an instantiation of the type variable the discovery is represented by the Type Variable Typing Slice.

- $(\phi_{\texttt{fun-param}} :: \Theta)(\forall \overline{a}.S \to T)_{\texttt{tpe}} = a$ and $a \not\in \overline{a}$:
  None of the type variables, and, in consequence, none of the type arguments are the source of the selected part of the prototype. We will explain the source of the prototype by delegating to the generic *TypeFocus*-based analysis.

The `(app`$_\perp$`)` *and* `(app`$_{tp,\perp}$`)` *type inference rules*

The `(app`$_\perp$`)` and `(app`$_{tp,\perp}$`)` type inference rules propagate prototype $\top$ as a consequence of the inferred type of the function, $\perp$. Consequently, the corresponding `slicesPtRoot`$_{(\texttt{app}_\perp)}$ and `slicesPtRoot`$_{(\texttt{app}_{tp,\perp})}$ functions analyze the indirect relation between the typing decisions by delegating to the *TypeFocus*-based SLICES algorithm. Since both $\top$ and $\perp$ represent types with no inner type components, we simplify the type selection to an identity *TypeFocus*, [ ], without compromising the correctness of the algorithm.

The `(rec)` *type inference rule*

The source of the highlighted $\top$ prototype in rule `slicesPtRoot`$_{(\texttt{rec})}$ cannot be inferred from other premises of the `(rec)` rule. By returning the Prototype Typing Slice we explain the indirect source of the prototype - the record type prototype which did not define the expected type for any of the members x_k where $m < k \leq n$.

The `(sel)` *type inference rule*

The $\{x : T\}$ prototype is used to infer the type of the record term in the `(sel)` type inference rule. Indirectly, the source of the fresh prototype lies in the record selection term. Rather than always returning the record selection term as an explanation of the fresh prototype, the algorithm takes into account the type selection in order to provide a correct explanation of the part of the fresh prototype.

For example, a simple inference judgment ($Int \to ?,\ \epsilon \vdash^{w}$ { $x =$ `fun`$(y)y$ }$.x :\ Int \to Int$) uses prototype $Int \to ?$ to infer the type of the `fun`$(y)y$ abstraction. If the analysis seeks to explain the source of only a fragment of the given prototype (Int $\to$?), then the Int prototype is a result of a prototype propagation and the decisions of the $.x$ record selection are irrelevant.

We use the properties of the type selection in order to distinguish between the two scenarios. From the precondition of the `slicesPtRoot` function ($\Theta, \emptyset \vdash^{\mathsf{WF}} \{x :\ P\}$) and by the Canonical Forms lemma (Lemma 3.5) `head`(Θ) is either [ ] or $[\phi_{\texttt{sel}_\texttt{x}}]$. If `head`$(\Theta)$ `==` $[\phi_{\texttt{sel}_\texttt{x}}]$ for any x then the record selection operations is irrelevant for explaining the source of the prototype and it has been inherited from the context (as explained by the Prototype Typing Slice in line 1). If `head`(Θ) = [ ] then Θ = [ ] (by Lemma D.2) and the source of the prototype lies in the record selection term, as explained by the Type Signature Typing Slice.

Final remarks

The `slicesPtRoot` function has to handle every type inference rule that may serve as a Propagation Root for some prototype. The essence of every case lies in identifying a link between the fresh prototype and a typing decision that introduced it in the formal type inference rule. The `slicesPtRoot` functions are generic in a sense that the whole information about the part of the fresh prototype, source of which we seek to explain, is encapsulated in the *TypeFocus* abstraction and the well-formedness property is maintained based on the formal specification of the type inference rules.

3.6.3 Source of the Prototype and Adaptation Typing Slices

We have divided the process of analyzing prototype propagation into two separate problems. Section 3.6.1 has defined the propagation judgment that identifies a Propagation Root, a typing judgment that introduces a fresh prototype in one of its premises. Section 3.6.2 has defined a generic function that locates the source of the freshly introduced prototype in any Propagation Root. Using the *TypeFocus* value we can now combine the two techniques to define a complete algorithm that explains the source of any non-wildcard prototype that has been propagated in the type derivation tree.

The observation leads to a definition of the SLICESPT function in Figure 3.21, which takes a Prototype or an Adaptation Typing Slice and returns an explanation of the source of the prototype that is represented by those Typing Slices.

```
SLICESPT ( $\langle v,\ (P^s,\ \Gamma^s \vdash^w E^s : T^s),\ \Theta^s \rangle$ ) =
  $[\,] \vDash^p (P^s,\ \Gamma^s \vdash^w E^s : T^s) \rightsquigarrow \left\langle (\vdash^w_r), \Theta^{\texttt{prototype}} \right\rangle$
  slicesPtRoot($\vdash^w_r$, $\Theta^{\texttt{prototype}}$ ::: $\Theta^s$)
```

Figure 3.21: Algorithm for explaining the source of prototype represented by Prototype and Adaptation Typing Slices. Algorithm is realized through the SLICESPT partial function of type $(v_3 \to \overline{v_3})$, which takes a Prototype or an Adaptation Typing Slice and returns other Typing Slices. The SLICESPT function is partial as it assumes that $v = v_{\texttt{PT}}$ or $v = v_{\texttt{ADAPT}}$.

The first step of the function infers the Propagation Root of the P^s prototype included in the Typing Slice. Having found the corresponding Propagation Root judgment, symbolically represented using the $\vdash^w_r$ notation, we can find the source of the fresh prototype that it introduces using the `slicesPtRoot` function. The `slicesPtRoot(`$\vdash^w_r$,$\Theta^{\texttt{prototype}}$`)` application would return the source of the P^s prototype, while the `slicesPtRoot(`$\vdash^w_r$,$\Theta^{\texttt{prototype}}$ ::: Θ^s`)` application returns the source of the target prototype P^{target}, which can be illustrated as:

$(\Theta^{\mathtt{prototype}} ::: \Theta^{\mathtt{s}})(P^{f}) =$

$\Theta^{\mathtt{s}}(\Theta^{\mathtt{prototype}}(P^{f})_{\mathtt{tpe}}) =$ (by the *TypeFocus* composition)

$\Theta^{\mathtt{s}}(([\,])(P^{s})_{\mathtt{tpe}}) =$ (by definition of $\Theta^{\mathtt{prototype}}$ in the propagation judgment)

$\Theta^{\mathtt{s}}(P^{s}) =$ (by definition of $[\,]$)

$\mathtt{inl}\ P^{target}$ (by definition of *Typing Slices*)

The call to the auxiliary `slicesPtRoot` function will always return the sequence of *Typing Slices* reflecting the initial type selection.

The SLICESPT function may produce further non-final Typing Slices. Their analysis is finite thanks to two basic properties of type propagation in Local Type Inference:

1. The types are consistently propagated in any type derivation tree from left to right, and from the *root* towards the leaf nodes. Since the algorithm of the SLICESPT function traces backwards through the adjacent nodes of the type derivation tree the involved type inference judgments come closer to the actual *root* of the type derivation tree with every analysis of the source of a prototype.
2. The *root* inference judgment must be inferred with a wildcard prototype.

In summary, the SLICESPT explains the origin of the target prototype represented by Prototype and Adaptation Typing Slices, which in turn explains the target type represented by those Typing Slices as well.

3.7 On understanding the type variable instantiation

The algorithm for analyzing the decisions of type derivation trees returns Typing Slices to explain the origin of a particular target type. Among different kinds of Typing Slices, the Type Variable Typing Slice is the only one which identifies the instantiation of a type variable as the source of the target type. In our formalization the type variable instantiation takes place only while inferring the type of function application, either with elided type arguments (the `(app)` type inference rule) or when they are explicitly provided (the $(\texttt{app}_{tp})$ type inference rule). For a complete type debugging experience we will define techniques that explain the instantiations of type variables.

In this section we show that the *TypeFocus* abstraction is flexible enough to allow for a convenient representation of type constraints that are used in the instantiation process. Consequently, the *TypeFocus* abstraction is sufficient to explain the source of type variable instantiation by explaining the source of the individual type constraints used in the process. Section 3.7.1 presents an algorithm for translating type constraints into their equivalent *TypeFocus* abstractions. The translation of type constraints not only provides a succinct and faithful representation of type constraints but allows us to trigger a *TypeFocus*-based analysis that will explain their origin (Section 3.7.2).

With such definitions in place, we are able to formalize the explanation of the typing decisions that inferred the type of functions applications with elided type arguments (Section 3.7.3), as well as those with explicit type arguments (Section 3.7.4). We conclude with the discussion about the possibility of debugging the variants of type variable instantiations (Section 3.7.5), such as the one present in the Scala implementation.

3.7.1 From a type constraint to a *TypeFocus*

The type variables instantiations are inferred from the collected type constraints (as explained in Section 2.1.5). The type constraints lead to $\overline{a}$-constraint sets that define lower and upper type bounds for each of the type variables in $\overline{a}$. The $\overline{a}$-constraint set does not carry information about the origin of type bounds, nor is it suitable for locating them due to the implicit approximation of type constraints. The key idea that links type constraints to our thesis is that individual type bounds can be represented as type selections on types that participate in the subtyping derivation.

We make use of the fact that all type constraints that are added to the $\overline{a}$-constraint set are of the form $\{A <: B\}$, where A and B are just part of the same subtyping check $S\ <:\ T$ and either $A \in \overline{a}$ or $B \in \overline{a}$. Therefore a *TypeFocus*, say Θ, that represents some type constraint from the subtype derivation extracts either $\Theta(S) = \texttt{inl}\ A\ \wedge\ \Theta(T) = \texttt{inl}\ B$ or $\Theta(S) = \texttt{inl}\ B\ \wedge\ \Theta(T) = \texttt{inl}\ A$.
The $\{A <: B\}$ form of the collected type constraint implies that only a lower or upper type

bound of a single type variable can be represented by a single *TypeFocus* instance at the same time. To distinguish between them, we will use the variance information $\psi_{\pm}$, defined as

$$\boxed{\psi_{\pm} \quad ::= \quad + \mid - \qquad\qquad (\textit{variance information})}$$

We define a *TypeFocus*-generation judgment of the form

$$\boxed{a,\, \psi_{\pm} \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta} \qquad\qquad (\textit{TypeFocus-generation})}$$

to infer a sequence of *TypeFocus* instances from the subtyping derivation between two types S and T, where either $\texttt{fv}(S) = \emptyset$ or $\texttt{fv}(T) = \emptyset$. We use a $\overline{\Theta}$ notation that is equivalent to $\overline{\Theta} = \left\{ \Theta^1, ..., \Theta^n \right\}$ for $n \geq 0$. The $\overline{\Theta}$ sequence represents individual lower (if $\psi_{\pm} = +$), or upper (if $\psi_{\pm} = -$) type bounds for some type variable a. Such definition means that Θ, where $\Theta \in \overline{\Theta}$, is equivalent to a single type constraint for some type variable a such that the type of the constraints is either $\Theta(S)_{\texttt{tpe}}$ or $\Theta(T)_{\texttt{tpe}}$.

***Example**: Representation of simple type constraints*

We consider an example of an $\{ a \}$-constraint set, C_1, generated from the type of the argument $((A \to B) \to A)$ and the type of the parameter $((A \to a) \to A)$ of some function application:

$$\vdash_{\{ a \}} (A \to B) \to A <: (A \to a) \to A \Rightarrow C_1, \text{ where } \{\bot <: a <: B\} \in C_1$$

The sequence of *TypeFocus* instances inferred for the identical subtyping derivation would therefore result in:

$$\begin{aligned} &a,\, + \vdash_{gen} (A \to B) \to A <: (A \to a) \to A \rightsquigarrow \epsilon \\ &a,\, - \vdash_{gen} (A \to B) \to A <: (A \to a) \to A \rightsquigarrow \left\{ [\phi_{\texttt{fun-param}}, \phi_{\texttt{fun-res}}] \right\} \end{aligned}$$

The type extracted using the generated sequence agrees with the corresponding upper type bound of the type variable a from the C_1 constraint set, since $([\phi_{\texttt{fun-param}}, \phi_{\texttt{fun-res}}])((A \to B) \to A) = \texttt{inl}\ B$. At the same time the empty sequence of *TypeFocus* instances inferred for the lower type bound of the type variable a corresponds to the implicitly added type $\bot$. The judgment is correct in a sense that it does not generate a *TypeFocus* instance for implicitly added $\bot$ and $\top$ type bounds.

The sequence of *TypeFocus* instances resulting from the judgment is oblivious to any approximations that take place in the regular constraint generation process. As we show in the next example, the approximated type bounds from the inferred constraint sets can always be recovered thanks to the variance information.

Example: *Representation of approximated type constraints*

We let C_2 represent the $\{\, a, b\,\}$-constraint set generated from the type of the argument and the type of the parameter of some function application:

$$\vdash_{\{\, a,b\,\}} \{x\colon\ a \to b,\ y\colon\ a \to Int\} <: \{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\} \Rightarrow C_2\ ,$$
$$\text{where } \{\bot \to Int <:\ a <:\ \top,\ \bot <:\ b <:\ Int\} \subseteq C_2$$

The sequence of *TypeFocus* instances inferred for the identical subtyping derivation for each of the type variables would therefore result in:

$$\begin{array}{l}
a,\ + \vdash_{gen}\ \{x\colon\ a \to b,\ y\colon\ a \to Int\} <:\ \{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\} \leadsto \\
\qquad \{\, [\phi_{\mathtt{sel}_x}, \phi_{\mathtt{fun\text{-}param}}],\ [\phi_{\mathtt{sel}_y}, \phi_{\mathtt{fun\text{-}param}}]\,\} \\
a,\ - \vdash_{gen}\ \{x\colon\ a \to b,\ y\colon\ a \to Int\} <:\ \{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\} \leadsto \epsilon \\
b,\ + \vdash_{gen}\ \{x\colon\ a \to b,\ y\colon\ a \to Int\} <:\ \{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\} \leadsto \epsilon \\
b,\ - \vdash_{gen}\ \{x\colon\ a \to b,\ y\colon\ a \to Int\} <:\ \{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\} \leadsto \\
\qquad \{\, [\phi_{\mathtt{sel}_x}, \phi_{\mathtt{fun\text{-}res}}]\,\}
\end{array}$$

The translation is not one-to-one equivalent, as in the previous example, because for the lower type bound of the type variable a it infers two *TypeFocus* instances, corresponding to the two *individual* lower type bounds; we can always manually calculate the *least upper bound* of the extracted types to reflect the approximated type bounds of the inferred constraint sets:

$$\begin{array}{l}
[\phi_{\mathtt{sel}_x}, \phi_{\mathtt{fun\text{-}param}}](\{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\})_{\mathtt{tpe}} \vee \\
\quad [\phi_{\mathtt{sel}_y}, \phi_{\mathtt{fun\text{-}param}}](\{x\colon\ (Int \to Int) \to Int,\ y\colon\ (\bot \to \bot) \to Int\})_{\mathtt{tpe}} = \\
\qquad Int \to Int \vee \bot \to \bot = \bot \to Int
\end{array}$$

The semantics of the TypeFocus translation - formally

The $\vdash_{gen}$ judgment is realized through a set of algorithmic $\Theta\mathtt{G}$ rules, defined in Figure 3.22. The $\Theta\mathtt{G}$ rules mimic the constraint generation $\mathtt{CG}$ rules defined in Pierce and Turner [2000] which in turn realize the ($\vdash_{\overline{a}}\ S <:\ T \Rightarrow C$) constraint generation judgment. The $\Theta\mathtt{G}$ rules recursively construct *TypeFocus* instances based on the shape of the subtyping derivation and the kind of type bounds considered. For clarity, the definition uses a $\left\{\, \Theta^{\mathtt{X}} :: \overline{\Theta}\,\right\}$ notation to abbreviate $\{\, \Theta^{\mathtt{X}} :: \Theta^1,\ \ldots,\ \Theta^{\mathtt{X}} :: \Theta^{\mathtt{n}}\,\}$, where $\Theta^{\mathtt{i}} \in \overline{\Theta}$ and $1 \le i \le n$. If $\overline{\Theta} = \epsilon$, then $\left\{\, \Theta^{\mathtt{X}} :: \overline{\Theta}\,\right\}$ is equivalent to an empty sequence.

The $\Theta\mathtt{G}$ algorithm defines four base rules: $\Theta\mathtt{G}_{(-,\,<)}$, $\Theta\mathtt{G}_{(+,\,<)}$, $\Theta\mathtt{G}_{(-,\,>)}$ and $\Theta\mathtt{G}_{(+,\,>)}$. We recall that the $\mathtt{CG}$ constraint generation algorithm defines only two rules that generate the base type

$$\Theta\mathtt{G}_{(+,\,<)}\ \frac{}{a,\ +, W \vdash_{gen}\ a <:\ T \leadsto \epsilon} \qquad \Theta\mathtt{G}_{(+,\,>)}\ \frac{}{a,\ +, W \vdash_{gen}\ T <:\ a \leadsto \{\,[\,]\,\}}$$

$$\Theta\mathtt{G}_{(-,\,<)}\ \frac{}{a,\ -, W \vdash_{gen}\ a <:\ T \leadsto \{\,[\,]\,\}} \qquad \Theta\mathtt{G}_{(-,\,>)}\ \frac{}{a,\ -, W \vdash_{gen}\ T <:\ a \leadsto \epsilon}$$

$$\Theta\mathtt{G}_{(\text{TOP})}\ \frac{S \not\in \{\,a, \bot\,\}}{a,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ \top \leadsto \epsilon} \qquad \Theta\mathtt{G}_{(\text{BOT})}\ \frac{S \not\in \{\,a, \top\,\}}{a,\ \psi_{\pm}, W \vdash_{gen}\ \bot <:\ S \leadsto \epsilon}$$

$$\Theta\mathtt{G}_{(\emptyset)}\ \frac{a \not\in (\mathtt{fv}(S) \cup \mathtt{fv}(T))}{a,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ T \leadsto \epsilon}$$

$$\Theta\mathtt{G}_{(\text{FUN})}\ \frac{\begin{array}{c} a \not\in \overline{b} \quad a \in (\mathtt{fv}(\forall\overline{a}.R \to S) \cup \mathtt{fv}(\forall\overline{a}.T \to U)) \\ a,\ \psi_{\pm},\ W \cup \overline{b} \vdash_{gen}\ T <:\ R \leadsto \overline{\Theta}' \qquad a,\ \psi_{\pm},\ W \cup \overline{b} \vdash_{gen}\ S <:\ U \leadsto \overline{\Theta}'' \end{array}}{a,\ \psi_{\pm},\ W \vdash_{gen}\ \forall\overline{b}.R \to S <:\ \forall\overline{b}.T \to U \leadsto \left\{\phi_{\mathtt{fun\text{-}param}} :: \overline{\Theta}'\right\} \cup \left\{\phi_{\mathtt{fun\text{-}res}} :: \overline{\Theta}''\right\}}$$

$$\Theta\mathtt{G}_{(\text{REC})}\ \frac{\begin{array}{c} a \in (\mathtt{fv}(S_1) \cup \ldots \cup \mathtt{fv}(S_n) \cup \mathtt{fv}(T_1) \cup \ldots \cup \mathtt{fv}(T_n)) \\ a,\ \psi_{\pm},\ W \vdash_{gen}\ S_1 <:\ T_1 \leadsto \overline{\Theta}^1 \quad \ldots \quad a,\ \psi_{\pm},\ W \vdash_{gen}\ S_m <:\ T_m \leadsto \overline{\Theta}^m \end{array}}{\begin{array}{r} a,\ \psi_{\pm},\ W \vdash_{gen}\ \{x_1 : S_1,\ \ldots,\ x_m : S_m,\ \ldots,\ x_n : S_n\} <:\ \{x_1 : T_1,\ \ldots,\ x_m : T_m\} \leadsto \\ \left\{\phi_{\mathtt{sel}_{x_1}} :: \overline{\Theta}^1\right\} \cup \ldots \cup \left\{\phi_{\mathtt{sel}_{x_m}} :: \overline{\Theta}^m\right\} \end{array}}$$

Figure 3.22: Algorithmic rules ΘG that define the $(a,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ T \leadsto \overline{\Theta})$ judgment. The rules mimic the corresponding constraint generation algorithm CG defined in Pierce and Turner [2000]. The implicit W variable set keeps track of bounded *out-of-scope* type variables.

constraints: (CG-Lower) and (CG-Upper).

The difference stems from our choice to ignore or accept type constraint information based on whether we seek to represent a lower or upper type bound of the type variable. There is no need no perform *variable-elimination promotion* ($\Uparrow$) and *demotion* ($\Downarrow$) directly within the ΘG rules because we do not perform any approximation. For example, rather than always inferring a *TypeFocus* instance for a subtyping derivation such as $a\ <:\ Int \to Int$, we will only do so, if we seek to represent the upper bounds of the type variable a.

Rules $\Theta\mathtt{G}_{(\text{TOP})}$ and $\Theta\mathtt{G}_{(\text{BOT})}$ directly correspond to their constraint generation counterparts (CG-(top)) and (CG-(bot)), respectively, where the top type and the bottom type is a supertype and a subtype of any type, respectively, and do not lead to type constraints. The additional premises in the rules, along with the $\Theta\mathtt{G}_{(\emptyset)}$ rule ensure that the definition is algorithmic. We notice that the implicitly and explicitly added $\bot$ and $\top$ type bounds are not distinguishable in the $\overline{a}$-constraint set; initially every $\overline{a}$-constraint set is $\{\bot <:\ a_i <:\ \top\}$ for all $a_i \in \overline{a}$. Our *TypeFocus* translation faithfully represents every type constraint, including those involving explicit top or bottom types.

The $\Theta\mathtt{G}_{(\mathrm{FUN})}$ and $\Theta\mathtt{G}_{(\mathrm{REC})}$ rules define the inference of *TypeFocus* for arrow and record type constructors, respectively. Whenever the subtyping derivation between their type elements returns a non-empty sequence, we simply compose it with an appropriate *TypeFocus* instance; the composition ensures that the type selection is well-formed with respect to the type application involving the given type constructor.

The algorithmic rules $\Theta\mathtt{G}$ are well-behaved, meaning that every type selection extracts a left tagged value from the types of the subtyping derivation. The statement is formally specified in Lemma 3.9.

Lemma 3.9 *Well-formedness of the* $\overline{\Theta}$ *sequence generated from the subtyping derivation.*
Let any S and T, a set of type variables $\overline{a}$, a type variable a_i such that $a_i \in \overline{a}$, and variance information $\psi_\pm$, such that either $\mathtt{fv}(T) \cap \overline{a} = \emptyset$, or $\mathtt{fv}(S) \cap \overline{a} = \emptyset$.
If $(a_i,\ \psi_\pm \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$ then
$\forall \Theta . \exists T' . \exists S' .\ \Theta \in \overline{\Theta}\ \wedge\ \Theta(T) = \mathtt{inl}\ T'\ \wedge\ \Theta(S) = \mathtt{inl}\ S' \wedge (T' = a_i \vee S' = a_i)$

Proof.
By induction on the last $\Theta\mathtt{G}$ rule used.
A complete proof is available in Appendix E.1. □

The generated sequences of *TypeFocus* instances are *sound* with respect to the type constraints that are generated by the original constraint generation judgment of Local Type Inference. The *soundness* property, formally stated in Lemma 3.10, states that for any type variable a_i the types that are extracted by the $\overline{\Theta}$ instances approximate (with least upper bound approximation) to the same type as the lower bound inferred from the constraint generation judgment. Similarly for the upper bound types of the type variable, except that the approximation means greatest lower bound approximation of types.

Lemma 3.10 *Soundness of the TypeFocus translation with respect to the inferred* $\overline{a}$*-constraint set.*
For any types S and T, a set of type variables $\overline{a}$ and a set of *out-of-scope* bounded variables W:

If $(a_i,\ + \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$ and $a_i \in \overline{a}$ and $(W \vdash_{\overline{a}} S <: T \Rightarrow C)$ then
$\{A <: a_i <: B\} \in C$ and
$(\mathtt{fv}(S) \cap \overline{a} = \emptyset \implies \bigvee_W \overline{\Theta}(T) = A)$ and $(\mathtt{fv}(T) \cap \overline{a} = \emptyset \implies \bigvee_W \overline{\Theta}(S) = A)$.

If $(a_i,\ - \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$ and $a_i \in \overline{a}$ and $(W \vdash_{\overline{a}} S <: T \Rightarrow C)$ then

$\{A <: a_i <: B\} \in C$ and
$(\mathtt{fv}(S) = \emptyset \implies \bigwedge_W \overline{\Theta}(T) = B)$ and $(\mathtt{fv}(T) = \emptyset \implies \bigwedge_W \overline{\Theta}(S) = B)$.

Proof.
By induction on the last rule used in the $(a_i, \psi_\pm \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$ judgment.
A complete proof is available in Appendix E.2. □

We use a $\bigvee_W \overline{\Theta}(T)$ notation to abbreviate a calculation of the least upper bound approximation from the type selections, *i.e.,* $\bigvee_W \overline{\Theta}(T)$ is equivalent to
$\Theta^1(T)_{\mathtt{tpe}} \Uparrow^W \ \vee \ldots \vee \ \Theta^n(T)_{\mathtt{tpe}} \Uparrow^W$ (for $\Theta^i \in \overline{\Theta}$ where $1 \le i \le n$). Similarly $\bigwedge_W \overline{\Theta}(T)$ abbreviates a calculation of the greatest lower bound approximation from the type selections, *i.e.,* $\bigwedge_V \overline{\Theta}(T)$ is equivalent to $\Theta^1(T)_{\mathtt{tpe}} \Downarrow^W \ \wedge \ldots \wedge \ \Theta^n(T)_{\mathtt{tpe}} \Downarrow^W$ (for $\Theta^i \in \overline{\Theta}$ where $1 \le i \le n$). If $\overline{\Theta} = \emptyset$ then $\bigvee_W \overline{\Theta}(T) = \bot$ and $\bigwedge_W \overline{\Theta}(T) = \top$. The approximations take into account the potential *variable-elimination promotion* ($\Uparrow$) and *demotion* ($\Downarrow$) with respect to the type variable set W, as carried out in the `(CG-Lower)` and `(CG-Upper)` rules, respectively. For presentation reasons, when the W set is omitted, the *promotion* and *demotion* is performed with respect to an empty set of bounded type variables.

The *TypeFocus* instances representing the type constraints of some type variable are also *complete* with respect to the $\overline{a}$-constraint set inferred by the corresponding constraint generation judgment. The *completeness* property, formally stated in Lemma 3.11, ensures that the *TypeFocus* instances inferred from the generation judgment reflect all the possible constraints corresponding to the lower and upper type bound, respectively. The *completeness* property is divided into two parts, one for each of the possible type bounds. Apart from the check for the inclusion of a type constraint (*i.e.,* $\Theta'(T) = \mathtt{inl}\ a_i$ implies $\Theta' \in \overline{\Theta}^+$) we also have to verify that the type constraint belongs to the appropriate type bound. The *TypeFocus*-sequences themselves do not carry information about the kind of the type bound they represent therefore both definitions rely on the fact that knowing which of the types of the subtyping derivation is type variable-free links the position of the type variable in the type with the kind of lower or upper type bound it can produce. For example, the $a <: \top$ subtyping check defines a type constraint that is valid as an upper type bound type constraint but not as the lower type bound type constraint.

The definition uses the implicit variance position function $\mathtt{pos}_a$ of type $T \to \psi$. The function returns information about the type T being constant, covariant, contravariant, or invariant in the given type variable a. The complete variance information is defined symbolically as

$$\psi \quad ::= \quad \psi_\pm \mid \mathbf{0} \mid \pm \qquad\qquad (\textit{complete variance information})$$

For simplicity, we assume now that if $\texttt{pos}_a(T)$ returns ± then both statements ± = + and ± = - are correct, and if $\texttt{pos}_a(T)$ returns **0** then statement **0** = + is valid, and defer the explanation of such semantics until Section 3.7.2.

Lemma 3.11 *Completeness of the TypeFocus translation with respect to lower and upper type bounds of the $\overline{a}$-constraint sets .*

We let $\overline{a}$ represent a set of free type variables,
let types S and T such that either $\texttt{fv}(S) \cap \overline{a} = \emptyset$ or $\texttt{fv}(T) \cap \overline{a} = \emptyset$.

If $(a_i, + \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta}^{+})$ then
$(\texttt{fv}(S) \cap \overline{a} = \emptyset) \implies$
$(\forall \Theta'. (\Theta'(T) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} S)$ and $(\texttt{pos}_{a_i}(T) = +) \implies \Theta' \in \overline{\Theta}^{+})$
and
$(\texttt{fv}(T) \cap \overline{a} = \emptyset) \implies$
$(\forall \Theta'. (\Theta'(S) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} T)$ and $(\texttt{pos}_{a_i}(S) = -) \implies \Theta' \in \overline{\Theta}^{+})$

If $(a_i, - \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta}^{-})$ then
$(\texttt{fv}(S) \cap \overline{a} = \emptyset) \implies$
$(\forall \Theta'. (\Theta'(T) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} S)$ and $(\texttt{pos}_{a_i}(T) = -) \implies \Theta' \in \overline{\Theta}^{-})$
and
$(\texttt{fv}(T) \cap \overline{a} = \emptyset) \implies$
$(\forall \Theta'. (\Theta'(S) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} T)$ and $(\texttt{pos}_{a_i}(S) = +) \implies \Theta' \in \overline{\Theta}^{-})$

Proof.
Prove by induction on the structure of S and T, the two types that participate in the subtyping derivation.
The complete proof is available in Appendix E.3. □

Together, the *soundness* and *completeness* properties ensure that the generated sequence of *TypeFocus* instances faithfully represent all type constraints that are used to infer lower or upper type bounds for the involved type variables, irrespective of any least upper bound or greatest lower bound approximations.

3.7.2 Relating type constraints to their source

With the *TypeFocus* translation from the previous section, we have shown that type constraints can be represented as type selections on types that participate in the subtyping deriva-

```
FUNCTION constraint-to-slice((app), $a$, $\psi_\pm$, $\Theta^{\mathsf{a}}$) =
    $$\boxed{\text{(app)}\ \dfrac{?, \Gamma, \vdash^{w} F : \forall\overline{a}.S \rightarrow T \quad [?/\overline{a}]\, S, \Gamma \vdash^{w} E : S' \quad \begin{array}{ll} \vdash_{a} S' <: S & \Rightarrow C_1 \\ \vdash_{a} T <: \top \searrow P & \Rightarrow C_2 \end{array}}{P, \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T}\, T \nearrow P}}$$
    constraint-to-slice$_{arg}$((app), $a$, $\psi_\pm$, $\Theta^{\mathsf{a}}$) $\cup$
     constraint-to-slice$_{pt}$((app), $a$, $\psi_\pm$, $\Theta^{\mathsf{a}}$)

FUNCTION constraint-to-slice$_{arg}$((app)($P, \Gamma \vdash^{w} F(E) : \sigma_{C_1, C_2, T}\, T \nearrow P$), $a$, $\psi_\pm$, $\Theta^{\mathsf{a}}$) =
    $a, \psi_\pm \vdash_{gen} S' <: S \rightsquigarrow \overline{\Theta}$
    $\bigcup^{\overline{\Theta}}$ SLICES(($[?/\overline{a}]\, S, \Gamma \vdash^{w} E : S'$), normalize($\Theta^{\mathsf{i}}$ ::: $\Theta^{\mathsf{a}}$, $S'$, $\emptyset$))        for $\Theta^{\mathsf{i}} \in \overline{\Theta}$

FUNCTION constraint-to-slice$_{pt}$((app)($P, \Gamma \vdash^{w} F(E) : \sigma_{C_1, C_2, T}\, T \nearrow P$), $a$, $\psi_\pm$, $\Theta^{\mathsf{a}}$) =
    $a, \psi_\pm \vdash_{gen} T <: \top \searrow P \rightsquigarrow \overline{\Theta}$
    $\bigcup^{\overline{\Theta}}$ $\langle v_{\mathsf{PT}}$, ($P, \Gamma \vdash^{w} F(E) : \sigma_{C_1 \cup C_2, T}\, T \nearrow P$), normalize($\Theta^{\mathsf{i}}$ ::: $\Theta^{\mathsf{a}}$, $P$, fv($P$))$\rangle$  for $\Theta^{\mathsf{i}} \in \overline{\Theta}$
```

Figure 3.23: Definition of the partial function `constraint-to-slice` of type $\big((P, \Gamma \vdash^{w} E : T),\ T,\ \psi_\pm, \Theta \big) \rightarrow \overline{v_3}$, which analyzes the source of type constraints used to infer the instantiation of a type variable. The origin of type constraints is explained using Typing Slices.

tion. In this section we show that such representation is sufficient to explain the origin of every individual type constraint.

We recall from Section 2.1.5, that the `(app)` inference rule infers the *minimal* type substitution $\sigma_{C_1 \cup C_2, T}$, from the two distinct $\overline{a}$-constraint sets:

1. The $\overline{a}$-constraint set C_1, in $\vdash_{\overline{a}} S' <: S \Rightarrow C_1$, infers type constraints from the subtyping check between the type of the argument and the type of the corresponding formal parameter. Importantly, the S' type has been inferred through a regular type inference judgment and can be analyzed using the *TypeFocus*-based algorithm (Section 3.5.3).

2. The $\overline{a}$-constraint set C_2, in $\vdash_{\overline{a}} T <: \top \searrow P \Rightarrow C_2$, infers type constraints from the subtyping check between the inferred result type of the function and the inherited prototype. Importantly, the $\top \searrow P$ type has been inferred from the expected type context and can be analyzed using the SLICESPT function that analyzes the source of the prototype (Section 3.6).

Figure 3.23 defines the algorithm that locates the source of used type constraints, based on the above observations. A `constraint-to-slice` partial function of type
$\big((P, \Gamma \vdash^{w} E : T),\ T,\ \psi_\pm, \Theta \big) \rightarrow \overline{v_3}$ is implemented in terms of two auxiliary partial functions, `constraint-to-slice`$_{arg}$ and `constraint-to-slice`$_{pt}$, of the same type; both are

parametrized by the function application inference judgment, a type variable which instantiation is to be analyzed, variance information indicating which kind of type bounds where used to infer the optimal solution, and a *TypeFocus*. The purpose of the *TypeFocus* value will be discussed later in the section. Both functions return the sequence of *Typing Slices*, which explain the origin of type constraints that instantiate the requested type variable.

We will now delve into the details of the two functions in order to explain how they analyze the two subtyping derivations.

The `constraint-to-slice`$_{arg}$ function

The `constraint-to-slice`$_{arg}$ function relies on the fact that the type of the argument, S', is inferred through a regular type inference judgment and can be directly explained with a *TypeFocus*-based SLICES algorithm.
Given the type selection $\Theta^{\mathtt{i}}$ representing the individual type constraint, the *TypeFocus*-based analysis can explain the source of the type constraint. This implies that the result will also explain the origin of the *complete* type variable instantiation. The SLICES$(([{}^{?}/_{\overline{a}}]\, S, \Gamma \vdash^{w} E : S'),\ \Theta^{\mathtt{i}})$ application would reflect such a correct semantics but it would also ignore the need for a more precise analysis, where we want to find the origin of only the *part* of the inferred type variable instantiation.

Explaining the source of only the fragment of the type variable instantiation

The `constraint-to-slice`$_{arg}$ function triggers the analysis with a `normalize`$(\Theta^{\mathtt{i}} ::: \Theta^{\mathtt{a}}, S', \emptyset)$ *TypeFocus*. To illustrate the need for the *normalization* we consider two possible values of $\Theta^{\mathtt{a}}$, representing the partial type selection on the type variable instantiation:

- **Case** $\Theta^{\mathtt{a}}$ == [ ] :
  `normalize`$(\Theta^{\mathtt{i}} ::: [\,], S', \emptyset) =$
  `normalize`$(\Theta^{\mathtt{i}}, S', \emptyset) =$ (by definition of the application of the [ ] *TypeFocus*)
  $\Theta^{\mathtt{i}}$ (by Lemma 3.9 and the definition of `normalize`)
- **Case** $\Theta^{\mathtt{a}}$!= [ ] :
  `normalize`$(\Theta^{\mathtt{i}} ::: \Theta^{\mathtt{a}}, S', \emptyset) =$
  `normalize`$(\Theta^{\mathtt{a}}, S'', \emptyset)$ (by Lemma 3.9 and $\Theta^{\mathtt{i}}(S') = \mathtt{inl}\ S''$)
  $\Theta^{\mathtt{a}\prime}$ (for some $\Theta^{\mathtt{a}\prime}$ such that $\Theta^{\mathtt{a}\prime}, \epsilon \vdash^{\mathtt{WF}} S''$)

The first case illustrates the scenario when the *TypeFocus* provided to the `constraint-to-slice` function is structurally equivalent to the identity type selection. The composition is equivalent to the $\Theta^{\mathtt{i}}$ *TypeFocus*, and is well-formed with respect to the type of the argument. This in turn means that it satisfies the pre-condition of the SLICES function and can be used to analyze the inferred type of the argument.
The second case illustrates the scenario when the *TypeFocus* provided to the

`constraint-to-slice` function will extract some part of the type variable instantiation. Since the type selection Θ^{a} does not guarantee the well-formedness with respect to the type of the argument, we have to apply the `normalize` function to it.

***Example**: Representing partial type variable instantiations*

To illustrate the challenges of explaining type variable instantiations through their type constraints, we consider an example of the $\{\,a\,\}$-constraint set inferred from the type of the argument, S', and type of the formal parameter of the function, S, in the constraint generation judgment:

$$\begin{aligned}&\vdash_a (Int \to \top) \to ((Int \to (Int \to Int)) \to Int) <: a \to (a \to Int) \Rightarrow \\ &\qquad \{\bot <: a <: Int \to (Int \to Int)\}\end{aligned}$$

The inferred $\{\,a\,\}$-constraint set had to calculate the greatest lower bound between the two upper bounds: $\{a \;<:\; Int \to \top\}$ and $\{a \;<:\; Int \to (Int \to Int)\}$. Using the inferred type constraints, we let the inferred type substitution be $\sigma = [a \Rightarrow Int \to (Int \to Int)]$, and for simplicity assume that the target type is part of the instantiated type variable a.

The type selections inferred for the upper bound of the type variable a are then:

$$\begin{aligned}&a,\ \text{-} \vdash_{gen}\ (Int \to \top) \to ((Int \to (Int \to Int)) \to Int) <: a \to (a \to Int) \rightsquigarrow \\ &\qquad\qquad \big\{\, [\phi_{\texttt{fun-param}}],\ [\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}] \,\big\}\end{aligned}$$

We consider two potential cases of the target type T_{target}:

- For $\Theta^{\mathsf{a}} = [\,]$, $T_{target} = \Theta^{\mathsf{a}}(\sigma a) = \Theta^{\mathsf{a}}(\,Int \to (Int \to Int)\,) = \texttt{inl}\ Int \to (Int \to Int)$: The target type refers to the complete type variable instantiation.

  Locating the source of the target type T_{target} resolves to understanding the source of its two type constraints represented by the $[\phi_{\texttt{fun-param}}]$ and $[\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}]$ type selection. The two values can guide the *TypeFocus*-base analysis of the argument of type S', which was part of the subtyping derivation, because $([\phi_{\texttt{fun-param}}], \epsilon \vdash^{\texttt{WF}} S')$ and $([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}], \epsilon \vdash^{\texttt{WF}} S')$.

- For $\Theta^{\mathsf{a}} = [\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}]$, $T_{target} = \Theta^{\mathsf{a}}(\sigma a) = \Theta^{\mathsf{a}}(Int \to (Int \to \,Int\,)) = \texttt{inl}\ Int$: The target type refers to a part of the type variable instantiation.

  Locating the source of the target type T_{target} resolves to understanding the source of its two type constraints represented by the $([\phi_{\texttt{fun-param}}] ::: \Theta^{\mathsf{a}})$ and $([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}] ::: \Theta^{\mathsf{a}})$ type selection. The straightforward composition comes at a cost of not necessarily being safe with respect to the inferred type of the argument. For our particular subcase we notice that $([\phi_{\texttt{fun-param}}] ::: \Theta^{\mathsf{a}}, \epsilon \nvdash^{\texttt{WF}} S')$ and $([\phi_{\texttt{fun-res}}, \phi_{\texttt{fun-param}}] ::: \Theta^{\mathsf{a}}, \epsilon \vdash^{\texttt{WF}} S')$, meaning that the former *TypeFocus* composition cannot guide the *TypeFocus*-based analysis.

The comparison shows the need for precision, meaning that we want to debug parts of the type variable instantiations, and completeness, meaning that we want to be able to debug the origin of the involved type constraints. The definition of the `constraint-to-slice` function (Figure 3.23), expresses the former through the *TypeFocus* composition, while the *normalization* ensures that the *TypeFocus*-based analysis still applies, without any loss in its precision.

The `constraint-to-slice`$_{pt}$ function

The `constraint-to-slice`$_{pt}$ function explains the source of the part of the prototype used in the subtyping derivations. It uses a similar methodology as in the `constraint-to-slice`$_{arg}$ function, except that it cannot trigger the *TypeFocus*-based analysis. Rather, we explain the source of the prototype directly, with the appropriately constructed type selection, by returning the Prototype Typing Slice.

Final remarks

The algorithm realized by the `constraint-to-slice` function highlights the advantages of separating the *TypeFocus*-based analysis and its related Typing Slices; by considering the individual type constraints and their source we can delegate their analysis to the previously defined *TypeFocus*-based algorithms. The `constraint-to-slice` function is generic, in a sense that it is parametrized by the type variable and variance information, and can be applied to any inference judgment where the last type inference rule used was `(app)`. The separation of the analysis of the lower and upper type bounds of the type variable is crucial for explaining different specifications of the *minimal substitution* that can be used to infer the type of the variable.

3.7.3 Explaining function applications with elided type arguments

Type Variable Typing Slices inform that the source of the target type lies in the inferred instantiation of some type variable. In this section we define an algorithm that explains the source of the inferred instantiation for function applications with elided type arguments. The analysis of the inference judgment that is included in the Typing Slice has to take into account different locations where type constraints are collected from as well as different semantics of the *minimal* type substitution that determine the inferred instantiation of the type variable.

The algorithms is realized by the SLICESTVAR partial function defined in Figure 3.24 which returns the sequence of other, potentially non-intermediate, Typing Slices explaining the source of the instantiation. The function is partial because it is defined only for the inference judgment where the last type inference rule used is `(app)`.

```
FUNCTION SLICESTVAR( $\langle v_{\mathtt{TVAR}},\ \mathtt{(app)}\ (P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P),\ \Theta^{\mathsf{s}} \rangle$ ) =
  case $\Theta^{\mathsf{s}}(T)$ of $\left| \begin{array}{lll} \mathtt{inl}\ T' & \Rightarrow & \langle T',\ [\,] \rangle \\ \mathtt{inr}\ \langle T',\ \Theta' \rangle & \Rightarrow & \langle T',\ \Theta' \rangle \end{array} \right| = \langle a,\ \Theta^{\mathsf{a}} \rangle$
  IF ($\mathtt{pos}_a(T) = + \lor \mathtt{pos}_a(T) = \mathbf{0}$)
    slicesTVARAux₊((app), $a$, $\Theta^{\mathsf{a}}$)
  ELSE IF ($\mathtt{pos}_a(T) = -$)
    slicesTVARAux₋((app), $a$, $\Theta^{\mathsf{a}}$)
  ELSE
    slicesTVARAux₊((app), $a$, $\Theta^{\mathsf{a}}$) ∪ slicesTVARAux₋((app), $a$, $\Theta^{\mathsf{a}}$)

FUNCTION slicesTVARAux₊( (app) $(P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P),\ T_a,\ \Theta^{\mathsf{a}}$ ) =
  constraint-to-slice((app), $T_a$, +, $\Theta^{\mathsf{a}}$) $= \overline{v_3}$
  IF ( $\overline{v_3} = \epsilon$ )   $\{ \langle v^{\perp}_{\mathtt{TSIG}},\ (P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P),\ \Theta^{\mathsf{a}} \rangle \}$
  ELSE   $\overline{v_3}$

FUNCTION slicesTVARAux₋( (app) $(P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P),\ T_a,\ \Theta^{\mathsf{a}}$ ) =
  constraint-to-slice((app), $T_a$, -, $\Theta^{\mathsf{a}}$) $= \overline{v_3}$
  IF ( $\overline{v_3} = \epsilon$ )   $\{ \langle v^{\top}_{\mathtt{TSIG}},\ (P,\ \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P),\ \Theta^{\mathsf{a}} \rangle \}$
  ELSE   $\overline{v_3}$
```

Figure 3.24: Algorithm for analyzing the source of the Type Variable Typing Slice for the inference judgment that infers the type of function application with elided type arguments. The algorithm is defined using a partial function SLICESTVAR of type $v_3 \to \overline{v_3}$, where *dom*(SLICESTVAR) is the Type Variable Typing Slice with the inference judgment having `(app)` as the last used type inference rule. The function is implemented in terms of two auxiliary functions which explain the origin of the lower or upper type bound of the given type variable separately.

The function is implemented in terms of two auxiliary partial functions, `slicesTVARAux₊` and `slicesTVARAux₋`, which explain the origin of type constraints that defined the lower and upper type bound of the type variable, respectively. By delegating to the appropriate auxiliary function, the algorithm can reflect the specification of the *minimal* type substitution that has been defined in the Local Type Inference formalization. The position of the underlying type variable with respect to result type of the function (T in $\forall \overline{a}.S \to T$) dictates its minimal substitution and the debugging technique to analyze it:

- If the T type is constant or covariant in the type variable, then we delegate to the `slicesTVARAux₊` function.
- If the T type is contravariant in the type variable, then we delegate to the `slicesTVARAux₋` function.
- Otherwise, the T type has to be invariant in the type variable; the approximated lower and upper type bounds of the type variable in the $\{\, a \,\}$-constraint set have to be equal,

meaning that both type bounds are the source of the inferred type variable instantiation and need to be analyzed.

Both of the auxiliary functions delegate to the previously described `constraint-to-slice` function which is parametrized by the variance information. The functions also verify the result of the `constraint-to-slice` function; lack of constraints manifests itself through an empty sequence, which corresponds to the decision of Local Type Inference to infer the bottom or the top type in the type hierarchy. We identify such decisions by returning the Type Signature Typing Slice with the underlying function application inference judgment. To avoid ambiguity between the two cases, we append an additional label as a superscript of the $\nu_{\texttt{TSIG}}$ tag, $\nu^{\perp}_{\texttt{TSIG}}$ and $\nu^{\top}_{\texttt{TSIG}}$, respectively.

3.7.4 Explaining function applications with explicit type arguments

Type Variable Typing Slices inform that the source of the target type lies in the inferred instantiation of some type variable. In this section we define an algorithm that explains the source of the inferred instantiation for function applications with explicit type arguments.

To relate the Type Variable Typing Slice with an explicit type argument we recall two circumstances where the Typing Slice can be returned by our algorithms:

- The core *TypeFocus*-based algorithm that analyzes typing decisions of the *(app$_{tp}$)* inference rule (the $\text{SLICES}_{(\texttt{app}_{tp})}$ function in Figure 3.14). The type variable has been extracted from the result type of the function type, *i.e.,* T in $\forall \overline{a}.S \rightarrow T$.
- Analysis of typing decisions of the Propagation Root (the $\texttt{slicesPtRoot}_{(\texttt{app}_{tp})}$ function in Figure 3.20). The judgment of the Typing Slice refers to the inference of the type of the function in function application. The type variable has been extracted from the parameter of the function type, *i.e.,* S in $\forall \overline{a}.S \rightarrow T$.

The distinction is visible in the $\texttt{sliceTVAR}_{(targ)}$ partial functions in Figure 3.25 that together realize the algorithm.

Both functions take the Type Variable Typing Slice which included the inference judgment and the *TypeFocus* information, necessary to identify the type variable. To explain the origin of the type variable instantiation the two functions return directly the corresponding type argument and a *TypeFocus* that is well-formed with respect to the type argument. This means that our *TypeFocus*-based analysis can not only identify the precise type argument as the source of some target type, but also identify the exact *part* of the type argument, based on the *TypeFocus* value. For example, in the `foldRight`[LIST`[`⊥`]]`(...) function application from Section 3.1.2 we would be capable of highlighting `foldRight`[LIST`[` ⊥ `]]`(...) rather than just `foldRight`[LIST`[`⊥`]`]`(...)`. In that sense, our *TypeFocus*-based analysis is not only limited to analyzing the elided type arguments but also the explicit ones.

```
FUNCTION sliceTVAR$_{(targ)}$ ( $\left\langle \nu_{\mathsf{TVAR}}, (\mathsf{app}_{tp})\ (P, \Gamma \vdash^{w} F\left[\overline{R}\right](E) : [\overline{R}/\overline{a}]\, T \nearrow P),\ \Theta^{s} \right\rangle$ ) =
    case $\Theta^{s}(T)$ of $\left|\begin{array}{lll} \texttt{inl}\ T' & \Rightarrow & \langle T', [\,]\rangle \\ \texttt{inr}\ \langle T', \Theta'\rangle & \Rightarrow & \langle T', \Theta'\rangle \end{array}\right| = \langle a,\ \Theta^{a}\rangle$
    $\langle R_a,\ \Theta^{a}\rangle$

FUNCTION sliceTVAR$_{(targ)}$ ( $\left\langle \nu_{\mathsf{TVAR}}, (?,\ \Gamma \vdash^{w} F : \forall\overline{a}.T \to S),\ \Theta^{s} \right\rangle$ ) =
    case $\Theta^{s}(\forall\overline{a}.T \to S)$ of $\left|\begin{array}{lll} \texttt{inl}\ T' & \Rightarrow & \langle T', [\,]\rangle \\ \texttt{inr}\ \langle T', \Theta^{a}\rangle & \Rightarrow & \langle T', \Theta^{a}\rangle \end{array}\right| = \langle a,\ \Theta^{a}\rangle$
    $(?,\ \Gamma \vdash^{w} F :\ \forall\overline{a}.T \to S) \downarrow=\ (\mathsf{app}_{tp})\ (P,\ \Gamma \vdash^{w} F\left[\overline{R}\right](E) :\ [\overline{R}/\overline{a}]\, T \nearrow P)$
    $\langle R_a,\ \Theta^{a}\rangle$
```

Figure 3.25: Analysis of Type Variable Typing Slice for function applications with explicit type arguments. The first partial function analyzes the Typing Slice reported as part of the SLICES$_{(\mathsf{app}_{tp})}$ function in Figure 3.14, the second partial function analyzes the Typing Slice reported as part of the `slicesPtRoot`$_{(\mathsf{app}_{tp})}$ function in Figure 3.20. For brevity, we assume that the type argument is linked with the type variable based on their indices, *i.e.,* R_a instantiates type variable a.

3.7.5 Discussion

The *TypeFocus*-based techniques for analyzing type variable instantiation have been formally based on the Local Type Inference. We discuss challenges of applying the same approach to the more realistic variants that are less strict by not always attempting to infer optimal type arguments.

Debugging Local Type Inference variants

The simple version of Local Type Inference presents a number of expressiveness problems by inferring unintended type variable instantiations or not inferring them at all, which may hinder its adoption. The work by Hosoya and Pierce [1999] illustrates two of the main problems of Local Type Inference as:

- *Hard-to-synthesize-arguments* - a category of problems largely related to the inability to infer types of anonymous functions that are arguments for polymorphic parameter types. Most of the non-polymorphic parameter types that exhibit such issues have been dealt with by the introduction of Colored Local Type Inference which propagates partial type information between adjacent nodes of the type derivation tree.
- *No best type argument* - synthesis of local type arguments defines strict rules on how a type substitution is chosen from the collected type constraints. This not only leads to

situations where the inferred type is under-approximated but may also imply that the type substitution is undefined when a suboptimal solution is available.

The second argument has been the source of the biggest criticism of the local type inference approach, especially for type parameters appearing covariantly and contravariantly in the result type of the function. We illustrate the problem using the function application involving reference values (we assume the notation of Simply Typed Lamda Calculus with References defined in Pierce [2002]).

Example: *Inference of instantiations for type variables that appear invariantly*

For illustration purposes we consider a function application '`ref` 1' where the `ref` function of type $\forall b.\ b \rightarrow \textsc{Ref}[b]$ is applied to a constant value 1 of type Int. The application returns a reference value.

The application of the `(app)` type inference rule yields two subtyping derivations ($\vdash_{\{b\}} Int <: b \Rightarrow \{Int <: b <: \top\}$) and ($\vdash_{\{b\}} \textsc{Ref}[x] <: \top \searrow\ ? \Rightarrow \{\bot <: b <: \top\}$). The constraint generation infers some $\{b\}$-constraint set C where $\{Int <: b <: \top\} \in C$. The specification for inferring the *minimal* type substitution dictates that the $\sigma_{C,\textsc{Ref}[b]}$ substitution is undefined because type $\textsc{Ref}[b]$ is invariant in the type variable b. The real-world implementations of Local Type Inference, such as the one present in Scala, do not attempt to be complete and instead instantiate the type variable to the lower type bound Int. Any complete type debugging technique that analyzes the synthesis of type arguments has to take into account the possibility of such non-optimal choices.

Modifying the semantics of the minimal substitution

The specification of the *minimal* type substitution for the type that is invariant in some type variable a requires both of its type bounds to be equal. This dictates the definition of the SLICESTVAR function in line 7 of Figure 3.24. The analysis of a non-optimal implementation, that selects either a lower or upper type bound, can be directly reflected by modifying the fragment of the function with

- either `slicesTVARAux`$_+$`((app),` a, Θ^a),
- or `slicesTVARAux`$_-$`((app),` a, Θ^a), respectively,

leaving the core of the *TypeFocus*-based analysis unchanged.

Minimal and maximal type substitution in a variant of GJ

The work of *Hosoya et al.* does not propose a definite solution when an optimal *minimal* type substitution does not exist. The first formalization attempt of Generic Java provided an early attempt to the challenge in Bracha et al. [1998]; the proposed formalization of type inference for Java with Generics used a special *undefined* type $\star$ which allowed for expressing compatibility between different type constructors and, as a result, removing the need for explicit type arguments all together. Unfortunately, the Java formalization has been shown to lead to soundness issues in Jeffrey [2001]. Due to a lack of any formal specification of the variant of GJ that Scala implements, this section discusses how the *TypeFocus*-based technique proposed in this thesis applies to the elements of the informal type system from Section 2.2.

The proposed type system of GJ uses a modified *minimal* type substitution specification in order to infer type arguments for function applications with elided type arguments, as defined in the `(INST-APP)` rule in Listing 2.6. Unlike Local Type Inference it also defines the specification of the *maximal* type substitution. The *maximal* type substitution is used for propagating partial type information when instantiating polymorphic types of the arguments.

The specifications for inferring the *minimal* and the *maximal* type substitution $\sigma_{C,T}$ from the $\overline{a}$-constraint set C with respect to some type T, can be summarized as follows:

- The *minimal* type substitution:

For each $\{S_i <: a_i <: T_i\} \in C$:

- If T is constant or covariant in a_i, then $\sigma_{C,T}(a_i) = S_i$
- else if T is contravariant in a_i, then $\sigma_{C,T}(a_i) = T_i$
- else if T is invariant in a_i, then $\sigma_{C,T}(a_i) = S_i$
- else $\sigma_{C,T}$ is undefined.

- The *maximal* type substitution:

For each $\{S_i <: a_i <: T_i\} \in C$:

- If T is constant or covariant in a_i, then $\sigma_{C,T}(a_i) = T_i$
- else if T is contravariant in a_i, then $\sigma_{C,T}(a_i) = S_i$
- else if T is invariant in a_i, then $\sigma_{C,T}(a_i) = T_i$
- else $\sigma_{C,T}$ is undefined.

The specifications refine the inference conditions for types that are invariant in a given type variable. The change does not pose any problems for our type debugging techniques since we already separate the analysis of lower and upper type bounds. The main challenge for understanding the decisions of Scala's type inference, in comparison to Colored Local Type Inference, is the mechanism that instantiates the polymorphic types of the arguments. In the remainder of the section we illustrate how a modest modification to the *TypeFocus* generation rules can infer *TypeFocus* sequences that faithfully represent type constraints that keep track of such inferred instantiations. Unlike the other parts of the chapter, the explanation will be example-driven, due to a lack of formal specification for the type system.

Extending the core language with $List$ type constructors

For illustration purposes the types of the core language are extended with a $List$ type constructor that is *invariant* in its only type parameter:

Terms	E	$=$	$\ldots$
Type schemes	U	$=$	$\forall\overline{a}.T$
Types	T, S, R	$=$	$a \mid \top \mid \bot \mid T \to T \mid \{x_1 : T,\ \ldots,\ x_n : T\} \mid \textsc{List}[T]$
Environments	Γ	$=$	$\ldots$

This, in turn, dictates an extension to the *TypeFocus* specification from Definition 4, such that $\Theta\ =\ \ldots \mid \phi_{\texttt{List}}$. Semantically, $\phi_{\texttt{List}}$ extracts the type argument from the type application involving $List$ type constructor such that:

$$(\phi_{\texttt{List}} :: \Theta')(T) \quad = \quad \begin{cases} \Theta'(A) & \textbf{if} \quad T = \textsc{List}[A] \\ \texttt{inr}\ \langle T,\ \phi_{\texttt{List}} :: \Theta' \rangle & \textbf{else} \end{cases}$$

To illustrate the analysis of *type scheme* instantiation we will use a few auxiliary functions, defined in Odersky [2002]. The type signatures of the functions are:

`nil:`	$\forall a.\ () \to \textsc{List}[a]$	
`cons:`	$\forall b.\ (b, \textsc{List}[b]) \to \textsc{List}[b]$	
`singleList:`	$\forall b.\ b \to \textsc{List}[b]$	
`toTop:`	$\top \to \bot$	*(equivalent to:* `val` x: $\top = \ldots$*)*
`toInt:`	$Int \to \bot$	*(equivalent to:* `val` x: $Int = \ldots$*)*
`toIntList:`	$\textsc{List}[Int] \to \bot$	*(equivalent to:* `val` x: $\textsc{List}[Int] = \ldots$*)*

Functions `nil`, `cons` and `singleList` provide basic operations on lists, such as creation of an empty list, appending an element to the list and creating a single element list, respec-

$$\Theta\mathtt{G}_{(+,\,<)}\ \frac{\boxed{T \neq ?}\ \boxed{T \neq a}}{a,\ +, W \vdash_{gen}\ a <:\ T \rightsquigarrow \epsilon} \qquad \Theta\mathtt{G}_{(+,\,>)}\ \frac{\boxed{T \neq ?}\ \boxed{T \neq a}}{a,\ +, W \vdash_{gen}\ T <:\ a \rightsquigarrow \{[\,]\}}$$

$$\Theta\mathtt{G}_{(-,\,<)}\ \frac{\boxed{T \neq ?}\ \boxed{T \neq a}}{a,\ -, W \vdash_{gen}\ a <:\ T \rightsquigarrow \{[\,]\}} \qquad \Theta\mathtt{G}_{(-,\,>)}\ \frac{\boxed{T \neq ?}\ \boxed{T \neq a}}{a,\ -, W \vdash_{gen}\ T <:\ a \rightsquigarrow \epsilon}$$

$$\Theta\mathtt{G}_{(\textsc{top})}\ \frac{S \not\in \{a, \bot, \boxed{?}\}}{a,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ \top \rightsquigarrow \epsilon} \qquad \Theta\mathtt{G}_{(\textsc{bot})}\ \frac{S \not\in \{a, \top, \boxed{?}\}}{a,\ \psi_{\pm}, W \vdash_{gen}\ \bot <:\ S \rightsquigarrow \epsilon}$$

$$\Theta\mathtt{G}_{(<,\,?)}\ \frac{}{a,\ \psi_{\pm}, W \vdash_{gen} ? <:\ T \rightsquigarrow \epsilon} \qquad \Theta\mathtt{G}_{(>,\,?)}\ \frac{}{a,\ \psi_{\pm}, W \vdash_{gen}\ T <:\ ? \rightsquigarrow \epsilon}$$

$$\ldots$$

$$\Theta\mathtt{G}_{\textsc{Var}}\ \frac{}{a,\ \psi_{\pm}, W \vdash_{gen}\ a <:\ a \rightsquigarrow \epsilon}$$

$$\Theta\mathtt{G}_{(\textsc{List})}\ \frac{\begin{array}{c} a \in (\mathtt{fv}(\forall\overline{b}.\textsc{List}[R] \cup \forall\overline{b}.\textsc{List}[S])) \\ a,\ \psi_{\pm},\ W \cup \overline{b} \vdash_{gen}\ R <:\ S \rightsquigarrow \overline{\Theta}' \qquad a,\ \psi_{\pm},\ W \cup \overline{b} \vdash_{gen}\ S <:\ R \rightsquigarrow \overline{\Theta}'' \end{array}}{a,\ \psi_{\pm},\ W \vdash_{gen}\ \forall\overline{b}.\textsc{List}[R] <:\ \forall\overline{b}.\textsc{List}[S] \rightsquigarrow \left\{\phi_{\mathtt{List}} :: \overline{\Theta}'\right\} \cup \left\{\phi_{\mathtt{List}} :: \overline{\Theta}''\right\}}$$

Figure 3.26: Extension of the $a,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ T \rightsquigarrow \overline{\Theta}$ *TypeFocus* generation judgment for the algorithmic rules presented in Figure 2.7 in Section 2.2. Unmodified rules are represented using the "..." notation, and any changes to the existing rules are emphasized using the grayed-out boxes.

tively. Functions `toTop`, `toInt` and `toIntList` encode variable assignment that is present in the original description; the functions take a single argument having a type variable-free type, allowing us to illustrate how a non-polymorphic function can propagate type information to its arguments.

In Figure 3.26 we define a modest extension of the *TypeFocus* generation judgment, and its $\Theta\mathtt{G}$ rules, to reflect the constraint generation rules proposed in Section 2.2. The additional rule $\Theta\mathtt{G}_{(\textsc{List})}$ generates type selectors from the subtyping derivation between two type applications involving list type constructors. The new $\Theta\mathtt{G}_{\textsc{Var}}$ rule (along with the highlighted $T \neq a$ and $T \neq ?$ premises that ensure unambiguity) corresponds directly to the constraint generation rule `(CG-Var)` that compares the type variable that appears on both sides of the subtyping check. The definition also takes into account the presence of the ? constant types that do not introduce any new type constraints to the constraint sets.

Debugging the instantiation of polymorphic types in GJ

With the auxiliary functions we will now illustrate that the decisions that instantiate the polymorphic types of arguments (*type schemes*) from the formal types of the parameters of the function can be translated to *TypeFocus* instances and thus analyzed by our *TypeFocus*-based approach. For reference, Figure 3.27 provides a summary of the instantiation rules that define the instantiation judgment.

$$\texttt{(Inst-REFL)}\ \frac{}{\emptyset,\ R;\Gamma \vdash \texttt{e} : T\ \textbf{inst}\ \texttt{e} : T} \qquad \texttt{(Inst-}\forall\texttt{)}\ \frac{\texttt{fv}(R) = W \quad W,\ \emptyset \vdash_{\overline{a}} T <: R \Rightarrow C \quad \max \sigma_{C,R}}{\emptyset,\ R;\Gamma \vdash \texttt{e} : \forall \overline{a}.T\ \textbf{inst}\ \texttt{e}[\sigma_{C,R}\ \overline{a}] : \sigma_{C,R} T}$$

$$\texttt{(Inst-1)}\ \frac{\emptyset,\ R;\Gamma \vdash \texttt{e} : U\ \textbf{inst}\ \texttt{e'} : T}{\overline{a},\ R;\Gamma \vdash \texttt{e} : U\ \textbf{inst}\ \texttt{e'} : T} \qquad \texttt{(Inst-2)}\ \frac{\begin{array}{c}\nexists e', T'.\ \emptyset,\ R;\Gamma \vdash \texttt{e} : U\ \textbf{inst}\ \texttt{e'} : T \\ R \approx_{\overline{a},\overline{b}} R' \quad \emptyset,\ [{}^{?}/_{\overline{b}}]\, R';\Gamma \vdash \texttt{e} : U\ \textbf{inst}\ \texttt{e'} : T\end{array}}{\overline{a},\ R;\Gamma \vdash \texttt{e} : U\ \textbf{inst}\ \texttt{e'} : T}$$

Figure 3.27: The formalization of the instantiation judgment as taken from Odersky [2002]. The `(Inst-∀)` rule uses the constraint generation judgment rather than the subsumption relation of $\Gamma \vdash S \leq T$, to highlight the source of type substitution.

- **Case** `(Inst-REFL) :`

  The rule applies when the inferred type of the argument is non-polymorphic, and the expected type R has been fully defined, meaning it has no unresolved type variables. By definition of the rule, the instantiation of the type of the argument is the type itself.

  For example, an argument 1 in the function application `toInt(1)` will not involve any type variable instantiation and the subtyping derivation for the type of the argument and the expected type leads to an empty constraint set, *i.e.*, $\vdash_{\emptyset} Int <: Int \Rightarrow \emptyset$. The subtyping derivation immediately leads to an equivalent empty set of *TypeFocus* instances.

- **Case** `(Inst-∀) :`

  The rule applies when the expected type R of the argument has been fully defined, but the inferred type of the argument is a *type scheme*.

  For example, the rule will type check the function application `toIntList(nil())` and assign type Int. To instantiate the type of the argument $nil()$, $\forall a.\textsc{List}[a]$, the rule uses the expected type $\textsc{List}[Int]$ coming from the parameter of the function. Using the regular constraint generation judgment ($\vdash_{\{\,a\,\}} \textsc{List}[a] <: \textsc{List}[Int] \Rightarrow C$) it infers the constraint set C, where $\{Int <:\ a <:\ Int\} \in C$, and the *maximal* type substitution Int for the type variable a. The inferred type substitution is sufficient to instantiate the $\forall a.\textsc{List}[a]$ type scheme to type $\textsc{List}[Int]$.

  The type selectors inferred from the subtyping derivation between the type of the argument and the type of the parameter of the function are:

  – $a,\ +,\ \emptyset \vdash_{gen} \textsc{List}[a] <: \textsc{List}[Int] \rightsquigarrow \overline{\Theta}'$, where $\overline{\Theta}' = \{\ [\phi_{\texttt{List}}]\ \}$.
  – $a,\ -,\ \emptyset \vdash_{gen} \textsc{List}[a] <: \textsc{List}[Int] \rightsquigarrow \overline{\Theta}''$, where $\overline{\Theta}'' = \{\ [\phi_{\texttt{List}}]\ \}$.

The sequences represent the type constraints of the type variable a and allow us reflect the equivalent maximal instantiation of type schema inferred from the type of the parameter of the function because $\bigwedge\overline{\Theta}''(\textsc{List}[Int]) = Int$. This in turn means that we can employ the *TypeFocus*-based analysis in order to analyze the instantiation of the *type scheme* for this particular case.

- **Case** `(Inst-1)` :

  The rule applies when the expected type of the argument R has been partially defined and may contain some uninstantiated type variables.

  For example, the rule allows to assign type $\textsc{List}[Int]$ to an application `cons(1, nil())`. In accordance with the rule `(INST-APP)`, types Int and $\forall a.\textsc{List}[a]$ are first assigned to the arguments of the application, respectively. For the first argument, the instantiation of type Int is type Int itself, according to the derivation involving rules `(Inst-1)` and `(Inst-REFL)`. For the type of the second argument, the `(Inst-1)` rule instantiates the type of $nil()$, $\forall a.\textsc{List}[a]$, with the expected type $\textsc{List}[b]$, where b is promoted to the set of *constant* type variables. The instantiation of the type of the $nil()$ argument infers an $\{ a \}$-constraint set C in the $(\{ b \}, \emptyset \vdash_{\{ a \}} \textsc{List}[a] <: \textsc{List}[b] \Rightarrow C)$ judgment, where $\{b <: a <: b\} \in C$. Consequently, the maximal instantiation of the $\forall a.\textsc{List}[a]$ type scheme is inferred as type $\textsc{List}[b]$.

  Using the *TypeFocus* generation rules, the subtyping derivation leads to the following *TypeFocus* instances:

  - $a, +, \{ b \} \vdash_{gen} \textsc{List}[a] <: \textsc{List}[b] \rightsquigarrow \overline{\Theta}'$, where $\overline{\Theta}' = \{ [\phi_{\texttt{List}}] \}$.
  - $a, -, \{ b \} \vdash_{gen} \textsc{List}[a] <: \textsc{List}[b] \rightsquigarrow \overline{\Theta}''$, where $\overline{\Theta}'' = \{ [\phi_{\texttt{List}}] \}$.

  The inferred *TypeFocus* instances agree with the inferred lower and upper type bounds of the type variable a, and again can reflect the maximal instantiation of type scheme from the parameter of the function because $\bigwedge\overline{\Theta}''(\textsc{List}[b]) = b$.

- **Case** `(Inst-2)` :

  Similarly as in the case of the rule `(Inst-1)`, the `(Inst-2)` applies when the expected type R has been partially defined and may contain some uninstantiated type variables. In order to disambiguate the application of the two rules, `(Inst-2)` applies only if the former has failed, as expressed through the $\nexists e', T'.\ \emptyset,\ R;\Gamma \vdash \texttt{e} : U$ **inst** $\texttt{e'} : T$ meta premise.

  For example, the rule will assign type $\textsc{List}[\top]$ to a `singleList(nil())` function application. The type scheme of the $nil()$ argument, $\forall a.\ \textsc{List}[a]$, cannot be instantiated with the expected type b because of the shape mismatch with the type of the formal parameter.
  Rather than rejecting such an expression, the `(Inst-2)` rule substitutes *constant* type variables with the wildcard constant types, as expressed by the notation of the approximated type R' $(R \approx_{\overline{a},\overline{b}} R')$ and the wildcard type substitution $([?/\overline{b}]\, R')$. This leads to a

constraint generation judgment ($\vdash_{\overline{a}}$ $\textsc{List}[a]$ <: ? $\Rightarrow C$), where $\{\bot <: a <: \top\} \in C$, the inference of the *maximal* type substitution of the type variable a to type $\top$, and the instantiation of the $\forall a.\textsc{List}[a]$ type scheme to type $\textsc{List}[\top]$.

The type selectors inferred from the same subtyping derivation faithfully represent the type constraints of the type variable a:

- $a, +, \emptyset \vdash_{gen}$ $\textsc{List}[a]$ <: ? $\rightsquigarrow \overline{\Theta}'$, where $\overline{\Theta}' = \epsilon$.
- $a, -, \emptyset \vdash_{gen}$ $\textsc{List}[a]$ <: ? $\rightsquigarrow \overline{\Theta}''$, where $\overline{\Theta}'' = \epsilon$.

The inferred *TypeFocus* instances agree with the inferred lower and upper type bounds of the type variable a, and again can reflect the maximal instantiation of the argument's type scheme from the parameter of the function because $\bigwedge\overline{\Theta}''(b) = \top$.

The informal explanation of the rules that instantiate *type schemes* and their type variables illustrates that the elements of their non-trivial typing decisions can be fully represented through *TypeFocus* instances, as well. It only took a modest extension of the ΘG generation rules to reflect the modified type system formalization. This in turn

We notice that the proposed GJ formalization, and its potential implementations, illustrate an interesting property of modern type systems. The locality and separation of the individual typing decisions, such as the instantiation of *type schemes*, inference of type substitutions or simply type assignment of types to terms, may lead to suboptimal solutions but at the same time are far more suitable for type debugging purposes. The analysis techniques can be easily modularized and customized, depending on the needs of the underlying type system.

Type graphs

Type systems using global type inference, *i.e.,* when type constraints are collected globally and solved in a separate stage, lack in general our freedom to customize the analysis algorithm without compromising or modifying the underlying type debugging technique. For example the type inference algorithms used for the traditional Hindley-Milner type systems rely on the unification of the collected type constraints. The unification leads to the inference of type variable substitutions but typically does not attempt to keep track of the steps that led to it.

Type graphs is a data structure introduced by *Heeren et al.* in Heeren et al. [2003a], Heeren [2005] to represent substitutions for type variables and which allows to mitigate the fundamental drawbacks of unification. Type graphs not only describe the direct equality type constraints collected from the Haskell programs but also the complete decision process explaining the reasons for the unification. In comparison to similar unification-based approaches, the data structure does not introduce implicitly any bias in solving the collected type constraints and is equally suitable for detecting, and representing inconsistent sets of type constraints for the erroneous programs.

Type graphs represent equality type constraints between the type constants, *e.g.,* A, and type variables, *e.g.,* v, in $A \equiv v$, as well as composite types involving type constructors, such as $F\ v_0\ v_1 \equiv F\ A\ B$, for some binary type constructor F. Type variables, type constants and type applications correspond to vertices in the graph. The *initial* equality type constraints are represented by the undirected edges between them. The vertices of the applied types are further decomposed into so called *term graphs* where the individual type elements, again represented as the *derived* vertices, are connected with the type constructors using the directed edges. That is why for the equality type constraints between the applied types, the data structure propagates the equality information down to the corresponding type elements of the types and represents them using the so called *implied* or *derived* edges. The type errors in such graphs are detected by essentially finding different type constants within the group of vertices connected using only the *initial* and the *implied* edges. The advantage of the type graph structure is that even though the inconsistencies can involve the vertices representing the type elements of the applied types, the directed edges of the *term graphs* allow us to trace back to the initial equality type constraints. Thus one can reduce the inconsistencies to erroneous paths on such type graphs without committing to any unification solving strategy.

The main disadvantage of the type graphs approach is that the data structure can grow arbitrary large. In order to limit the scope of the unification the authors of Heeren [2005] propose to limit the number of the derived edges under certain conditions. Furthermore, similarly to how we deal with space explosion in Section 5.5, they can limit the size of type graphs to individual binding groups, which ads the local aspect to the global type inference approach.

The debugging of type derivation trees representing the decision of local type inference indirectly constructs type graphs when analyzing the subtyping checks between types (either correct and the failed ones). For example, the analysis of the subtyping check `A <: B` in the function application type inference rule, for some types `A` and `B` where either type may contain some type variables, generates the local type graph, where we construct *term graph* for each of the individual type `A` and `B`, and the added lower or upper bounds of type variables represent the *derived* edges. The *TypeFocus* values that are inferred from the subtyping checks represent the same concept as the paths that are inferred from the type graph data structure. In our case however, the *TypeFocus* selections, or paths, are later used in order to navigate further decisions of the adjacent nodes of the type derivation tree, since the reconstructed type graphs are only local with respect to the type inference rules. Moreover the local type graphs are reconstructed on-the-fly from the already committed decisions of local type inference, rather than influencing it.

Conclusion

The role of *TypeFocus* is far more fundamental that just an encapsulation of target type information at each node of the type derivation tree, or an encapsulation of type constraints information from the subtyping derivation; the *TypeFocus* serves as an abstraction that can unify the consequences of different typing decisions. The latter can later be used to guide the analysis using the established *TypeFocus*-based algorithm.

3.8 On understanding the Type Signature Typing Slice

The algorithm for analyzing the decisions of type derivation trees returns Typing Slices to explain the origin of a particular target type. Among different kinds of Typing Slices, the Type Signature Typing Slice is always final and directly corresponds to some synthesized type information and a program location. In this section we describe how the *TypeFocus* that is part of every Typing Slice allows for the reconstruction of the source location from the underlying type inference judgment.

The core *TypeFocus*-based algorithm has identified a number of scenarios where a Type Signature Typing Slice is returned as an explanation of some target type. The algorithm for analyzing the source of the propagation of the expected type (Section 3.6) and the algorithm for analyzing the source of the type variable instantiation (Section 3.7.3) also explicitly return a Type Signature Typing Slice under certain circumstances.

We list all scenarios in the analysis of the decisions of the type inference rules that return the Type Signature Typing Slice, and explain how their elements translate directly to program locations:

- **Case** $\langle \nu_{\mathtt{TSIG}},\ (P,\ \Gamma \vdash^{w} x : \Gamma(x) \nearrow P),\ \Theta \rangle$ (the $\textsc{slices}_{(\mathtt{var})}$ function in Figure 3.13):

  The type of the variable, coming from the environment Γ, is identified as the source of the target type. Therefore the location of the variable itself implies the target type.

  Admittedly, such a source may be considered as not final because it depends on the node in the type derivation tree where the environment is extended with a variable type information. For type derivation trees of the core language, finding such node would simply resolve to a trivial backtracking and is omitted. For languages with mutable state or non-local scopes of the environment an implementation-dependent search on the type derivation tree has to be developed.

  By definition of Typing Slices the included *TypeFocus* is well-formed with respect to the type of the variable $(\Theta, \mathtt{fv}(\Gamma(x)) \vdash^{\mathtt{WF}} \Gamma(x))$. Consequently, the source of the target type can be identified from the source of the type assigned to the variable using the provided *TypeFocus* value.

- **Case** $\left\langle \nu_{\mathtt{TSIG}},\ (?,\ \Gamma \vdash^{w} \mathtt{fun}[\overline{a}](x : T)E : \forall \overline{a}.T \to S),\ \Theta \right\rangle$ (the $\textsc{slices}_{(\mathtt{abs}_{tp,?})}$ function in Figure 3.13) :

  By definition of Typing Slices the included *TypeFocus* is well-formed with respect to the inferred type of the function $(\Theta, \overline{a} \vdash^{\mathtt{WF}} \forall \overline{a}.T \to S)$. Using the Canonical Forms Lemma (Lemma 3.5), $\mathtt{head}(\Theta)$ is either $[\,]$, $[\phi_{\mathtt{fun\text{-}param}}]$ or $[\phi_{\mathtt{fun\text{-}res}}]$:

  - $\mathtt{head}(\Theta) = [\,]$:
    By Lemma D.2, $\Theta = [\,]$. Therefore the target type is represented by the function

type constructor and the abstraction term of the inference judgment explains the origin of the target type.

- $\texttt{head}(\Theta) = [\phi_{\texttt{fun-param}}]$:
  From the definition of the type inference rule the source of the target type lies in the part of the explicit type of the parameter of the abstraction, T. To explain the origin of the target type we can apply the tail of the *TypeFocus* to the type of the parameter, such that $\texttt{tail}(\Theta)(T) = \texttt{inl}\ T'$ for some type T'. The T' fragment of the type of the parameter represents the smallest program fragment explaining the source of the target type.
- $\texttt{head}(\Theta) = [\phi_{\texttt{fun-res}}]$:
  By definition of the $\text{SLICES}_{(\texttt{abs}_{tp,?})}$ function, the case is not possible.

- **Case** $\langle \nu_{\texttt{TSIG}}, (\forall\overline{a}.P \to P', \Gamma \vdash^{w} \textbf{fun}[\overline{a}](x : T)E : \forall\overline{a}.T \to S \nearrow \forall\overline{a}.P \to P'), \Theta\rangle$ (in the $\text{SLICES}_{(\texttt{abs}_{tp})}$ function):

  By definition of Typing Slices the included *TypeFocus* is well-formed with respect to the assigned type of the function $(\Theta, \overline{a} \vdash^{\texttt{WF}} \forall\overline{a}.T \to S)$. Using the same argument as for the $\text{SLICES}_{(\texttt{abs}_{tp,?})}$ function, the source of the target type is either the abstraction term of the judgment or the explicit type of the parameter of the function.

- **Case** $\langle \nu_{\texttt{TSIG}}, (?, \Gamma \vdash^{w} \{x_1 = F_1, ..., x_n = F_n\} : \{x_1 : T_1, ..., x_n : T_n\}), \Theta\rangle$ (the $\text{SLICES}_{(\texttt{rec?})}$ function in Figure 3.16):

  By definition of Typing Slices the included *TypeFocus* is well-formed with respect to the inferred type of the record $(\Theta, \texttt{fv}(\{x_1 : T_1, ..., x_n : T_n\}) \vdash^{\texttt{WF}} \{x_1 : T_1, ..., x_n : T_n\})$. Using the Canonical Forms Lemma (Lemma 3.5), $\texttt{head}(\Theta)$ is either $[\,]$ or $[\phi_{\texttt{sel}_{x_k}}]$ for some k such that $1 \le k \le n$:

  - $\texttt{head}(\Theta) = [\,]$:
    By Lemma D.2, $\Theta = [\,]$. Therefore the target type is a record type and the record term of the judgment, $\{x_1 = F_1, ..., x_n = F_n\}$, explains the origin of the target type.
  - $\texttt{head}(\Theta) = [\phi_{\texttt{sel}_{x_k}}]$:
    By definition of the $\text{SLICES}_{(\texttt{rec?})}$ function, the case is not possible.

- **Case** $\langle \nu_{\texttt{TSIG}}, (\{x : P\}, \Gamma \vdash^{w} F : \{x : T\}), \Theta\rangle$ (the $\texttt{slicesPtRoot}_{(\texttt{sel})}$ function in Figure 3.20):

  By definition of Typing Slices the included *TypeFocus* is well-formed with respect to the inferred type of the term $(\Theta, \texttt{fv}(T) \vdash^{\texttt{WF}} T)$. From the definition of the $\texttt{slicesPtRoot}_{(\texttt{sel})}$ function, $\texttt{head}(\Theta) = [\,]$, and $\Theta = [\,]$ (Lemma D.2). Since the *TypeFocus* is always an identity type selection, the target type refers to the record type, and the underlying term F explains the source of the target type.

- **Case** $\langle \nu^{\perp}_{\texttt{TSIG}}, (P, \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P), \Theta\rangle$ (the $\texttt{slicesTVARAux}_{+}$ function in Figure 3.24):

The Type Signature Typing Slice explains that the target type $\bot$ has been inferred as a result of lack of type constraints in the function application with elided type arguments. The identity of the type variable that is instantiated to the $\bot$ type can trivially be recovered using the included *TypeFocus* value: given $(?, \Gamma \vdash^{w} F : \forall \overline{a}.S \to T)$ for some S and T, $\Theta(T) = \texttt{inl}\ x$ and $x \in \overline{a}$. Lack of type constraints only indirectly corresponds to the function application program location, therefore a more elaborate explanation of the target type would have to be put in place.

- **Case** $\langle \nu^{\top}_{\texttt{TSIG}}, (P, \Gamma \vdash^{w} F(E) : \sigma_{C_1,C_2,T}\, T \nearrow P), \Theta \rangle$ (in the SLICESTVAR$_{-}$ function):

  The case is analogous to the $\texttt{slicesTVARAux}_{+}$ function, except for the inferred type variable instantiation, $\top$.

The Type Signature Typing Slice represents a final step in the analysis of the source of the target type. This means that any type debugging mechanism that implements the *TypeFocus*-based analysis has to provide a detailed description for every kind of Type Signature Typing Slice, *e.g.,* explaining the link of Typing Slices, their typing judgments and *TypeFocus*value with the source code. As we have shown in the above cases, the included *TypeFocus* value is sufficient to extract fragments of the explicit type annotations that determine the source of the target type. We also notice that while the number of possible Type Signature Typing Slices is not small, it is tractable and can be trivially defined for any implementation of our type debugging technique.

3.9 Conclusions

We have presented a new approach to understanding the decisions of Local Type Inference, and its variants. We assume the existence of a type derivation tree representing the type checking of the program and provide means to navigate its decisions in a controlled way. Depending on which type element of the inferred type of the term we want to explain, there are potentially many different combinations of the nodes of the type derivation tree that introduce the type for the first time. We provide a systematic and a deterministic way of stepping through the adjacent nodes of the type derivation tree based on a concept of type selection. While simple, the type selection was shown to be sufficient to direct the analysis of for example non-trivial type variable instantiations that involve the analysis of subtype checking as well.

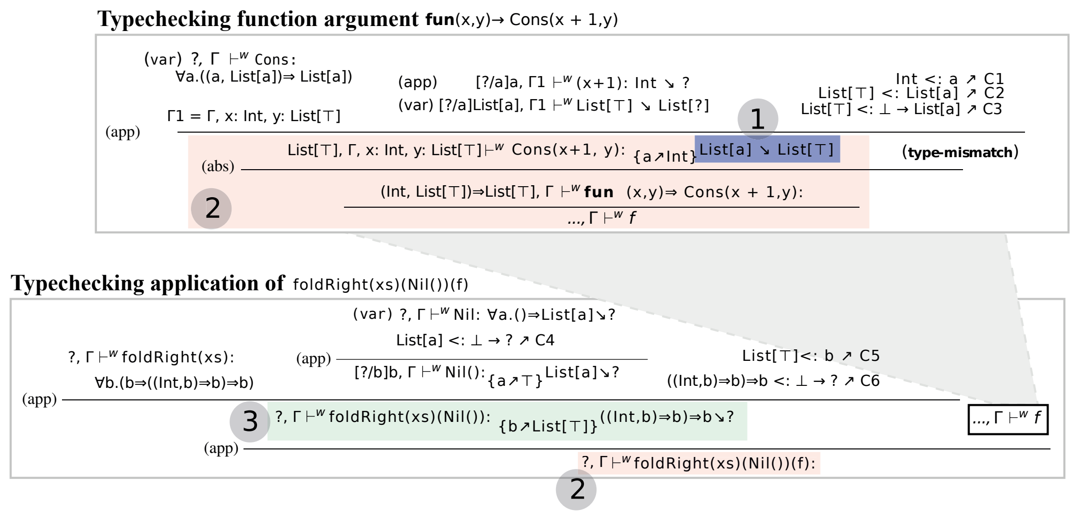
The formal approach illustrates how the analysis of some selected fragment of the inferred type of the term can be continued among the different type checking decisions that can influence it. In other words the inputs and outputs of the algorithm and the specialized functions always explain the same initial type selection that triggered the analysis in the first place. As part of the analysis of the initial inference judgment, each of the explored nodes of the type derivation is associated with a unique *TypeFocus* type selection. This in turn means that the final nodes of the *TypeFocus*-based analysis remain loosely connected with the initial inference judgment, and its type selection, without maintaining an expensive and complex data structure on top of the type derivation tree.

Apart from being able to separate the analysis of different kinds of the type checking decisions, the approach mimics the mechanism of local type inference; the local type inference approximates types and propagates the type information in multiple steps, between the adjacent nodes. Similarly, the exploration of the nodes of the type derivation tree is based on the type selection that is constructed only from the previously visited, adjacent nodes.

Chapter 4

Foundations of type mismatch errors

The core *TypeFocus*-based algorithm (Section 3.5.3) analyzes typing decisions of type derivation trees. The algorithm is defined for type derivation trees, or their fragments, that are derivable, *i.e.,* no errors were encountered during the application of type inference rules. In this section we show that the non-derivable parts of type derivation trees, *i.e.,* the branch of the type derivation that failed to infer the type of the term, can be reduced to a *TypeFocus* value and we can use it to trigger the *TypeFocus*-based analysis techniques from the previous chapter. Indirectly, the translation also shows how to find the initial inputs for the *TypeFocus*-based analysis functions for any type mismatch error.



Typechecking function argument fun(x,y)→ Cons(x + 1,y)
(var) ?, Γ ⊢w Cons: ∀a.((a, List[a])⇒ List[a])
Γ1 = Γ, x: Int, y: List[⊤]
(app) [?/a]a, Γ1 ⊢w (x+1): Int ↘ ?
(var) [?/a]List[a], Γ1 ⊢w List[⊤] ↘ List[?]
Int <: a ↗ C1
List[⊤] <: List[a] ↗ C2
List[⊤] <: ⊥ → List[a] ↗ C3
1
(app)
List[⊤], Γ, x: Int, y: List[⊤]⊢w Cons(x+1, y): {a↗Int} List[a] ↘ List[⊤]
(type-mismatch)
(abs)
2
(Int, List[⊤])⇒List[⊤], Γ ⊢w fun (x,y)⇒ Cons(x + 1,y):
..., Γ ⊢w f
Typechecking application of foldRight(xs)(Nil())(f)
(var) ?, Γ ⊢w Nil: ∀a.()⇒List[a]↘?
List[a] <: ⊥ → ? ↗ C4
?, Γ ⊢w foldRight(xs): ∀b.(b⇒((Int,b)⇒b)⇒b)
(app)
[?/b]b, Γ ⊢w Nil(): {a↗⊤} List[a]↘?
List[⊤]<: b ↗ C5
((Int,b)⇒b)⇒b <: ⊥ → ? ↗ C6
(app)
3
?, Γ ⊢w foldRight(xs)(Nil()): {b↗List[⊤]} ((Int,b)⇒b)⇒b↘?
..., Γ ⊢w f
(app)
?, Γ ⊢w foldRight(xs)(Nil())(f):
2

Figure 4.1: Fragment of a type derivation tree for the application `foldRight`(xs)(`Nil()`)(f), where xs = `Cons(1,Nil())` and f = `fun(x,y)` → `Cons(x + 1,y)`. This section provides the analysis and the translation of the highlighted elements of the tree.

We show in this section that the problem of explaining a type mismatch error in a type derivation tree constructed using the rules of Colored Local Type Inference consists of three parts. For clarity, we illustrate each of them on the familiar `foldRight` function application in Figure 4.1:

- We translate the conflicting types that failed the subtyping relation into *TypeFocus* instances (Section 4.2, the highlighted box 1 in Figure 4.1).
- We translate the non-derivable parts of type derivation trees into *TypeFocus* instances (Section 4.1, the highlighted box 2 in Figure 4.1). The non-derivable tree refers to a fragment of the complete type derivation tree that failed to infer the type of the term but nevertheless retained the structure of the derivation, including the kind of type inference rules applied and their prototype elements.
- We locate the error-free type inference judgment which introduced the conflicting expected type, or part of it, for the first time (Section 4.1, the highlighted box 3 in Figure 4.1). The located inference judgment and the reconstructed *TypeFocus* value allows us to trigger a regular *TypeFocus*-based analysis algorithm in search of the source of the expected type.

The type inference rules of Colored Local Type Inference precisely define typing decisions where a type mismatch error can occur. The errors materialize due to a failed $\nearrow$ adaptation attempt between the synthesized type of the term and the inherited expected type. In Section 4.3 we define a complete set of steps necessary to explain the source of the two types that participate in a type mismatch.

We conclude with a complete example of a non-trivial type mismatch conflict that is explained formally using only our *TypeFocus*-based approach (Section 4.4).

4.1 Inference of a Propagation Root

We recall that the conflicting expected type, in a failed adaptation process, is propagated in the non-derivable fragments of type derivation trees in an identical way as for the error-free type derivation trees, *i.e.,* it is driven by the prototype component in the type inference rules. The observation implies that the propagation of the conflicting prototype is just a special case of the analysis of the propagation of the expected type that we formalized in Section 3.6.

The `Prop` rules (Figure 3.18 in Section 3.6.1), that realize the search for the *Propagation Root*, use the prototype information in order to infer the equivalent *TypeFocus* values. Because none of the rules utilize the information about the inferred types of terms, and are only-driven by the kind of the type inference rule used, the search for the Propagation Root in a non-derivable inference judgment is just a special case of the $\vDash^p$ prototype propagation judgment. That is why we define a propagation judgment for the erroneous type derivation trees in an almost identical way to its derivable counterpart from Section 3.6.1:

$$(\Theta^{\mathtt{i}} \vDash_e^p (P^i,\ \Gamma^i \vdash^w E^i :\ T^i) \rightsquigarrow \langle (P^o,\ \Gamma^o \vdash^w E^o :\ T^o), \Theta^{\mathtt{o}} \rangle)$$

The $\vDash_e^p$ propagation judgment takes a *TypeFocus* instance ($\Theta^{\mathtt{i}}$) and a failed inference judgment ($(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)$), such that ($\Theta^{\mathtt{i}}, \varnothing \vdash^{\mathtt{WF}} P^i$) and ($\Theta^{\mathtt{i}}(P^i)_{\mathtt{tpe}} \neq\ ?$). The propagation judgment infers a tuple consisting of the inference judgment ($P^o,\ \Gamma^o \vdash^w E^o :\ T^o$) and a *TypeFocus* $\Theta^{\mathtt{o}}$. The returned inference judgment represents the Propagation Root of P^i and the returned *TypeFocus* value reduces the Prototype Propagation Path of P^i to a type selection, such that $\Theta^{\mathtt{o}}(P^f)_{\mathtt{tpe}}\ ==\ \Theta^{\mathtt{i}}(P^i)_{\mathtt{tpe}}$. The lack of the inferred type information is represented through a grayed-out part in the propagation judgment and can be simply ignored.

For easier understanding, the $\vDash_e^p$ propagation judgment is realized in the algorithmic fashion using the `PrototypeBacktrackError` function in Figure 4.2, similarly to its error-free counterpart (Figure 3.19). The `PrototypeBacktrackError` function differs in two aspects:

- The inferred type component is missing.
- The cases for the (**abs**$_{tp,\top}$), (**rec**$_\top$), (**app**$_{tp,\bot}$), (**app**$_\bot$) rules and a subcase of the (**rec**) rule are omitted because they propagate the top type. From the definition of the $\nearrow$ operation, $\forall T.\ (T \nearrow \top) = \top$, therefore type $\top$ will never lead to a type mismatch error.

The algorithm pattern matches on the kind of the last type inference rule used in the parent of the input inference judgment. Pattern matching either backtracks through the nodes of the type derivation tree on the prototype propagation in a recursive manner (the type inference rules (**abs**), (**abs**$_{tp}$) and (**rec**)), or returns the Propagation Root when the prototype value is *freshly* introduced from one of its typing decisions (the type inference rules (**app**), (**app**$_{tp}$) and (**sel**)). The individual recursive invocations reduce the prototype propagation to the appropriate type selections, based on the kind of the type inference rule. The grayed-out type elements indicate the failure to infer the type of the term.

Using the above argument, we notice that all properties of the $\vDash^p$ propagation judgment for the error-free type derivation trees apply directly to the $\vDash_e^p$ propagation judgment for the erroneous type derivation trees. We refer the reader to Section 3.6.1 for details.

Analysis of the Propagation Root (for non-derivable inference judgments)

The `PrototypeBacktrackError` function from Figure 4.2 identifies three type inference rules as a potential Propagation Root for a non-? in a partially derivable type derivation tree. Similarly as in their error-free counterparts, the typing decisions that infer the fresh prototype, P^f, in the Propagation root may differ significantly from rule to rule.

```
FUNCTION PrototypeBacktrackError($\Theta^{\mathtt{i}}$, $(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)$) =
  $((P^i,\ \Gamma^i \vdash^w E^i :\ T^i))\downarrow\ =\ \vdash^w_{\mathtt{parent}}$
  MATCH $(\vdash^w_{\mathtt{parent}})$ OF
    CASE (abs) :      PrototypeBacktrackError($\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{i}},\ \vdash^w_{\mathtt{parent}}$)
    CASE ($\mathbf{abs}_{tp}$) :  PrototypeBacktrackError($\phi_{\mathtt{fun\text{-}res}} :: \Theta^{\mathtt{i}},\ \vdash^w_{\mathtt{parent}}$)
    CASE (rec) :
     $(\vdash^w_{\mathtt{parent}})$ == (rec) $\{x_1 : P_1, ..., x_m : P_m\},\ \Gamma \vdash^w \{x_1 = F_1, ..., x_n = F_n\} : \{x_1 : T_1, ..., x_m : T_m\}$
     $(P^i,\ \Gamma^i \vdash^w E^i :\ T^i)$ == $(P_k,\ \Gamma \vdash^w F_k :\ T_k)$                    for $1 \le k \le m$
     PrototypeBacktrackError($\phi_{\mathtt{sel}_{x_k}} :: \Theta^{\mathtt{i}},\ \vdash^w_{\mathtt{parent}}$)
    CASE (app) :    $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE ($\mathbf{app}_{tp}$) :  $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
    CASE (sel) :    $\left\langle \vdash^w_{\mathtt{parent}},\ \Theta^{\mathtt{i}} \right\rangle$
```

Figure 4.2: The algorithmic definition of the $\Theta^{\mathtt{i}} \vDash^p_e (P^i,\ \Gamma^i \vdash^w E^i :\ T^i) \rightsquigarrow \langle (P^o,\ \Gamma^o \vdash^w E^o :\ T^o), \Theta^{\mathtt{o}} \rangle$ prototype propagation judgment for erroneous type derivation trees.

The search for the source of the fresh prototype introduced in a non-derivable inference judgment is realized by the SLICESPTERROR partial function. Similarly as in the case of its error-free counterpart (the `slicesPtRoot` function in Section 3.6.1), the SLICESPTERROR function is of type ($(P, \Gamma \vdash^w E \colon T)$, Θ) $\rightarrow \overline{\nu_3}$; the function takes the inference judgment representing the inferred Propagation Root, and the *TypeFocus* that expresses a well-formed type selection on a freshly introduced prototype P^f, where P^f belongs to the premise of the provided Propagation Root judgment, and returns the source of the $\Theta(P^f)_{\mathtt{tpe}}$ prototype in the form of Typing Slices. The grayed-out part of the inferred types indicates that the function accepts judgments that failed to infer the type of the term.

The SLICESPTERROR function is realized through a set of type inference rule-specialized partial functions in Figure 4.3. Each of the possible type inference rules is considered separately, as indicated through the *rule* subscript in the function name $\text{SLICESPTERROR}_{\mathit{rule}}$. For clarity, we highlight the position of the fresh prototype P^f in Figure 4.3 with gray boxes. The definition of the algorithm assumes that $\Theta, \mathtt{fv}(P^f) \vdash^{\mathtt{WF}} P^f$, where the P^f prototype is type inference rule-specific.

The SLICESPTERROR and `slicesPtRoot` functions differ only in the interpretation of the decisions of the `(sel)` type inference rule that infers the type of record member selection. Unlike the error-free counterpart, we cannot report the intermediate Prototype Typing Slice to represent the source of the conflicting prototype because Typing Slices require a derivable inference judgment. Instead, the function inlines the analysis of the implicit Prototype Typing Slice that is reported in the `slicesPtRoot` function.

FUNCTION $\textsc{slicesPtError}_{(\texttt{app})}\big(\,(\texttt{app})\ (P,\ \Gamma \vdash^{w} F(E){:}\ \sigma_{C_1\cup C_2,T}\, T \nearrow P),\ \Theta\,\big) =$

$$\boxed{(\text{app})\ \dfrac{?,\ \Gamma, \vdash^{w} F : \forall\overline{a}.S \to T \qquad [?/\overline{a}]\,S, \Gamma \vdash^{w} E{:}\ S' \qquad \begin{array}{ll}\vdash_a S' <: S & \Rightarrow C_1\\ \vdash_a T <: \top \searrow P & \Rightarrow C_2\end{array}}{P,\ \Gamma \vdash^{w} F(E){:}\ \sigma_{C_1\cup C_2,T}\, T \nearrow P}}$$

```
SLICES((?, Γ,⊢^w F : ∀ā.S → T), ϕ_fun-param :: Θ )
```

FUNCTION $\textsc{slicesPtError}_{(\texttt{app}_{tp})}\Big(\,(\texttt{app}_{tp})\ (P,\ \Gamma \vdash^{w} F\Big[\overline{R}\Big](E){:}\ [\overline{R}/\overline{a}]\ T \nearrow P),\ \Theta\,\Big) =$

$$\boxed{(\text{app}_{tp})\ \dfrac{?,\ \Gamma, \vdash^{w} F : \forall\overline{a}.S \to T \qquad [\overline{R}/\overline{a}]\,S,\ \Gamma \vdash^{w} E{:}\ [\overline{R}/\overline{a}]\,S}{P,\ \Gamma \vdash^{w} F\Big[\overline{R}\Big](E){:}\ [\overline{R}/\overline{a}]\ T \nearrow P}}$$

```
Θ^cont = ϕ_fun-param :: Θ
IF (is-tvar(Θ^cont(S)_tpe, ā))  { ⟨ν_TVAR, (?, Γ,⊢^w F : ∀ā.S → T), Θ^cont⟩ }
ELSE                            SLICES((?, Γ,⊢^w F : ∀ā.S → T), Θ^cont)
```

FUNCTION $\textsc{slicesPtError}_{(\texttt{sel})}\,(\,(\texttt{sel})\ (P,\ \Gamma \vdash^{w} F.x{:}\ T),\ \Theta\,) =$

$$\boxed{(\text{sel})\ \dfrac{\{x : P\},\ \Gamma, \vdash^{w} F{:}\ \{x : T\}}{P,\ \Gamma \vdash^{w} F.x{:}\ T}}$$

```
IF (head(Θ) == [ϕ_sel_x])
  tail(Θ) ⊨_e^p (P, Γ ⊢^w F.x: T) ⇝ ⟨(P^o, Γ^o ⊢^w E^o: T^o), Θ^o⟩
  SLICESPTERROR((P^o, Γ^o ⊢^w E^o: T^o), Θ^o)
ELSE
  { ⟨ν_TSIG, ({x : P}, Γ ⊢^w F : {x : P} ↘ ⊥), tail(Θ)⟩ }
```

Figure 4.3: The definition of the $\textsc{slicesPtError}$ partial function of type $(\,(P,\ \Gamma \vdash^{w} E :\ T),\ \Theta\,) \to \overline{\nu_3}$ that locates the source of the fresh prototype P^{fresh} that is introduced in the non-derivable judgment of the Propagation Root. The highlighted parts represent the possible fresh P^{f} prototypes introduced for the first time in the Propagation Roots.

4.2 From a failed subtyping derivation to a *TypeFocus*

A failed $\nearrow$ adaptation between the two types S and T represents an inability to find the smallest supertype of S that structurally matches the type T. In other words, a failed $S \nearrow T$ adaptation is equivalent to a failed $S <: T$ subtyping derivation. In order to explain the source of the conflicting elements of the synthesized term type and the inherited expected type, we have to take into account the details of the failed subtyping derivation in our *TypeFocus*-based exploration.

We define a translation of the failed subtyping derivation to equivalent *TypeFocus* type selections in the judgment of the form ($\vdash^{sub}_{decl} S \not<: T \to \overline{\Theta}$). The judgment defines the inference of *TypeFocus* instances ($\overline{\Theta}$) that extract the conflicting type elements from the types S and T. Figure 4.4 provides a definition of the subtyping judgment in terms of the declarative rules

$$\Theta SubD_{(\texttt{sub})}\ \frac{S <: T \quad \textit{(derivable)}}{\vdash^{sub}_{decl} S \not<: T \rightarrow \epsilon} \qquad \Theta SubD_{(\texttt{fail})}\ \frac{S <: T \quad \textit{(not derivable)}}{\vdash^{sub}_{decl} S \not<: T \rightarrow \{\,[\,]\,\}}$$

$$\Theta SubD_{(\texttt{fun})}\ \frac{\vdash^{sub}_{decl} T \not<: R \rightarrow \overline{\Theta}' \qquad \vdash^{sub}_{decl} S \not<: U \rightarrow \overline{\Theta}''}{\vdash^{sub}_{decl} \forall\overline{a}.R \rightarrow S \not<: \forall\overline{a}.T \rightarrow U \rightarrow \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}' \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}'' \right\}}$$

$$\Theta SubD_{(\texttt{rec})}\ \frac{\vdash^{sub}_{decl} P_1 \not<: R_1 \rightarrow \overline{\Theta}^1 \quad \ldots. \quad \vdash^{sub}_{decl} P_m \not<: R_m \rightarrow \overline{\Theta}^m}{\begin{array}{l}\vdash^{sub}_{decl} \{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\} \not<: \{x_1 : R_1, \ldots, x_m : R_m\} \rightarrow \\ \qquad\qquad \left\{ \phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1 \right\} \cup \ldots \cup \left\{ \phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m \right\}\end{array}}$$

Figure 4.4: Definition of the $\vdash^{sub}_{decl} S \not<: T \rightarrow \overline{\Theta}$ judgment using declarative $\Theta SubD$ rules.

$\Theta SubD$ that target the subtyping relation of the core language.

For example, the type $(Int \rightarrow Int)$ is not a subtype of the type $(\top \rightarrow Int)$, and the failed subtyping derivation translates the relation between the two types as

$$\vdash^{sub}_{decl} (Int \rightarrow Int) \not<: (\top \rightarrow Int) \rightarrow \left\{ [\phi_{\texttt{fun-param}}] \right\}$$

The inferred *TypeFocus* faithfully represents the conflicting elements of both types since

$$[\phi_{\texttt{fun-param}}](Int \rightarrow Int) = \texttt{inl}\ Int \text{ and } [\phi_{\texttt{fun-param}}](\top \rightarrow Int) = \texttt{inl}\ \top \text{ and } \top \not<: Int.$$

The $\Theta SubD$ rules return a non-empty sequence of type selectors for two types S and T that do not satisfy the subtyping relation. The base case $\Theta SubD_{(\texttt{fail})}$ rule returns an identity type selection which highlights the conflicting elements. The other base rule, $\Theta SubD_{(\texttt{sub})}$, corresponds to types that are subtypes, as required by the premise of the rule, and return an empty sequence of *TypeFocus* instances. The $\Theta SubD_{(\texttt{fun})}$ and $\Theta SubD_{(\texttt{rec})}$ rules correspond to the subtyping check between the function and record type constructors, respectively; the type constructor rules only collect the *TypeFocus* instances corresponding to their conflicting elements.

The failed subtyping judgment is sound with respect to the underlying types, meaning that every *TypeFocus* instance inferred from the failed subtyping derivation extracts type elements that are not subtypes. Lemma 4.1 formally states the soundness property. The disjunction on the right hand side of the implication is necessary to specify that the conflicting elements may arise due to type parameters being covariant, contravariant, or invariant in the type constructor.

Lemma 4.1 *Soundness of the TypeFocus translation with respect to the failed subtyping derivation.*
For any two types S and T, if $\vdash^{sub}_{decl} S \not<: T \rightarrow \overline{\Theta}$, then

$$\forall \Theta.\ \Theta \in \overline{\Theta} \text{ implies } (\Theta(S)_{\texttt{tpe}} \not<: \Theta(T)_{\texttt{tpe}}) \text{ or } (\Theta(T)_{\texttt{tpe}} \not<: \Theta(S)_{\texttt{tpe}})$$

Proof.
Straightforward. By induction on the last $\Theta SubD$ rule used.

□

The declarative $\Theta SubD$ rules are not syntax driven, nor do the they enforce inferring unique *TypeFocus* instances that extract minimal type elements. For example, the declarative $\vdash^{sub}_{decl}$ rules accept the following derivation of the previously discussed failed subtyping derivation $\vdash^{sub}_{decl} (Int \rightarrow Int) \not<: (\top \rightarrow Int) \rightarrow \{\,[\,]\,\}$ since

$[\,](Int \rightarrow Int) = \texttt{inl}\ (Int \rightarrow Int)$ and $[\,](\top \rightarrow Int) = \texttt{inl}\ (\top \rightarrow Int)$ and $Int \rightarrow Int \not<: \top \rightarrow Int$.

In Figure 4.5 we present a complete definition of the algorithmic ΘSub rules that unambiguously realize the algorithmic $\vdash^{sub}$ subtyping judgment. The ΘSub rules are syntax driven and capture the information about the conflicting type elements of the types. In contrast to the previous declarative definition, the construction of the rules ensures that type selection extracts the smallest possible conflicting type elements of the two participating types.

The ΘSub base case rules return the identity *TypeFocus* when two types cannot directly satisfy the subtyping relation: $\Theta Sub_{(var1)}$, $\Theta Sub_{(var2)}$, $\Theta Sub_{(fun1)}$, $\Theta Sub_{(fun2)}$, $\Theta Sub_{(rec1)}$, $\Theta Sub_{(rec2)}$ and $\Theta Sub_{(top-bot)}$. The type selection is further refined through the *TypeFocus* composition in the $\Theta Sub_{(fun)}$ and $\Theta Sub_{(rec)}$ rules corresponding to subtype checking for type constructors. Similarly as in the case of the declarative rules, types that always satisfy the subtyping relation yield an empty *TypeFocus* sequence, as defined by the $\Theta Sub_{(var)}$, $\Theta Sub_{(top)}$ and $\Theta Sub_{(bot)}$ rules in our core language.

The ΘSub rules also take into account the implicit presence of the wildcard constant type; the extension allows for applying the failed subtyping judgment to all types that participate in the $\nearrow$ adaptation. The grayed-out $\Theta Sub_{(>?)}$ and $\Theta Sub_{(<?)}$ rules simply generate an empty sequence when ? participates in the subtyping relation since, by its definition, it matches any type. In order to avoid ambiguous or incorrect type selectors some of the rules also have an additional grayed-out condition in their-premises with respect to the wildcard type.

$$\Theta Sub_{(var)}\ \frac{}{\vdash^{sub} a \not<: a \rightarrow \epsilon} \qquad \Theta Sub_{(var1)}\ \frac{X \not\in \{\, a,\ \top, ?\,\}}{\vdash^{sub} a \not<: X \rightarrow \{\,[\,]\,\}}$$

$$\Theta Sub_{(var2)}\ \frac{X \not\in \{\, a,\ \bot, ?\,\}}{\vdash^{sub} X \not<: a \rightarrow \{\,[\,]\,\}} \qquad \Theta Sub_{(top-bot)}\ \frac{}{\vdash^{sub} \top \not<: \bot \rightarrow \{\,[\,]\,\}}$$

$$\Theta Sub_{(top)}\ \frac{}{\vdash^{sub} T \not<: \top \rightarrow \epsilon} \qquad \Theta Sub_{(bot)}\ \frac{}{\vdash^{sub} \bot \not<: T \rightarrow \epsilon}$$

$$\Theta Sub_{(<?)}\ \frac{}{\vdash^{sub} T \not<: ? \rightarrow \epsilon} \qquad \Theta Sub_{(>?)}\ \frac{}{\vdash^{sub} ? \not<: T \rightarrow \epsilon}$$

$$\Theta Sub_{(fun)}\ \frac{\vdash^{sub} T \not<: R \rightarrow \overline{\Theta}' \qquad \vdash^{sub} S \not<: U \rightarrow \overline{\Theta}''}{\vdash^{sub} \forall\overline{a}.R \rightarrow S \not<: \forall\overline{a}.T \rightarrow U \rightarrow \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}' \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}'' \right\}}$$

$$\Theta Sub_{(fun1)}\ \frac{X \not\in \left\{ (\forall\overline{a}.R \rightarrow S),\ \bot,\ ? \right\}}{\vdash^{sub} X \not<: \forall\overline{a}.T \rightarrow U \rightarrow \{\,[\,]\,\}} \qquad \Theta Sub_{(fun2)}\ \frac{X \not\in \left\{ (\forall\overline{a}.T \rightarrow U),\ \top,\ ? \right\}}{\vdash^{sub} \forall\overline{a}.R \rightarrow S \not<: X \rightarrow \{\,[\,]\,\}}$$

$$\Theta Sub_{(rec)}\ \frac{\vdash^{sub} P_1 \not<: R_1 \rightarrow \overline{\Theta}^1 \quad \ldots\ldots \quad \vdash^{sub} P_m \not<: R_m \rightarrow \overline{\Theta}^m}{\begin{array}{r}\vdash^{sub} \{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\} \not<: \{x_1 : R_1, \ldots, x_m : R_m\} \rightarrow \\ \left\{ \phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1 \right\} \cup \ldots \cup \left\{ \phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m \right\}\end{array}}$$

$$\Theta Sub_{(rec1)}\ \frac{X \not\in \{\, \{x_1 : R_1, \ldots, x_m : R_m\},\ \top,\ ?\,\}}{\vdash^{sub} \{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\} \not<: X \rightarrow \{\,[\,]\,\}}$$

$$\Theta Sub_{(rec2)}\ \frac{X \not\in \{\, \{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\},\ \bot,\ ?\,\}}{\vdash^{sub} X \not<: \{x_1 : R_1, \ldots, x_m : R_m\} \rightarrow \{\,[\,]\,\}}$$

Figure 4.5: Definition of the $\vdash^{sub} S \not<: T \rightarrow \overline{\Theta}$ judgment using algorithmic ΘSub rules. Grayed-out parts represent an extension of the failed subtyping judgment with ?, *don't care*, constant types.

The algorithmic definition of the failed subtyping judgment is sound with respect to its declarative counterpart, as stated in Lemma 4.3.

Lemma 4.2 *Soundness of the algorithmic* $\vdash^{sub}$ *translation with respect to the declarative* $\vdash^{sub}_{decl}$ *translation.*
For any types S and T,
$(\vdash^{sub} S \not<: T \rightarrow \overline{\Theta})$ implies $\exists\overline{\Theta}'.\ \vdash^{sub}_{decl} S \not<: T \rightarrow \overline{\Theta}'$ and $(\forall\Theta'.\ \Theta' \in \overline{\Theta}' \implies \Theta' \in \overline{\Theta})$.

Proof.
Straight-forward. By induction on the last ΘSub rule used. □

The $\vdash^{sub}$ judgment is however not complete with respect to the $\vdash^{sub}_{decl}$ judgment in a traditional statement of the problem:

> For any two types S and T, if $\vdash^{sub}_{decl} S \not<: T \to \overline{\Theta}$ then $\vdash^{sub} S \not<: T \to \overline{\Theta}'$ for any $\overline{\Theta}'$ and $\forall\Theta.\, \Theta \in \overline{\Theta} \implies \Theta \in \overline{\Theta}'$.

The declarative style of the $\Theta SubD$ rules means that the $\vdash^{sub}_{decl}$ judgment may infer *TypeFocus* instances that do not extract the smallest possible conflicting type and will not be inferred in the algorithmic $\vdash^{sub}$ judgment. The algorithmic $\vdash^{sub}$ judgment does correctly reflect the failed subtyping derivations, as expressed through the refined completeness property in Lemma 4.3. The latter ensures that a failed subtyping derivation always translates to a non-empty sequence of type selections that extract the conflicting type elements.

Lemma 4.3 *Completeness property of the TypeFocus translation in the* $\vdash^{sub}$ *judgment with respect to the failed subtyping derivation.*
For any type S and T,
if $S \not<: T$, then $\vdash^{sub} S \not<: T \to \overline{\Theta}$ and $\overline{\Theta} \neq \epsilon$ and

$\forall\Theta.\, \Theta \in \overline{\Theta}$ implies $(\exists S', T'.\, \Theta(S) = \texttt{inl}\ S'$ and $\Theta(T) = \texttt{inl}\ T'$ and $(S' \not<: T'$ or $T' \not<: S'))$

Proof.
By induction (twice) on the structure of types S and T.
A complete proof is available in Appendix E.4. □

The immediate consequence of the translation of the failed subtyping derivation is the ability to represent the type elements of the conflicting types, which integrates well with the regular *TypeFocus*-based analysis approach. This in turn implies that our type debugging approach can be applied to explaining non-trivial terms involving large number of type constructors and type arguments.

4.3 On explaining type mismatch errors

A traditional approach to explaining type mismatch errors involves finding either the minimal term (or the minimal number of program locations) that synthesized the conflicting type, or the minimal term (or the minimal number of program locations) that introduced the conflicting expected type (examples involve the work of Chitil [2001] or Chen and Erwig [2014b]). In general, without involvement of the users, we can only apply heuristics to decide which of the two options conflicts with the programmer's intention. As we will show in this section the *TypeFocus* translation devised in the previous sections is generic enough to allow us to trigger the analysis of both of the possibilities.

For any two types S and T that are not subtypes and result in the failed inference judgment, $(T,\ \Gamma \vdash^{w} E:\ S \not\nearrow T)$, we can infer a sequence of *TypeFocus* instances representing the failed type elements in the $\vdash^{sub} S\ \not<:\ T \rightarrow \overline{\Theta}$ judgment. The inferred type selection Θ^{sub}, where $\Theta^{\mathsf{sub}} \in \overline{\Theta}$, is sufficient to explain

- the source of the conflicting **type element** of S using the regular *TypeFocus*-based analysis algorithm in
  $\boxed{\textsc{slices}((?,\ \Gamma \vdash^{w} E:\ S),\ \Theta^{\mathsf{sub}})}$
- the source of the conflicting **type element** of T in the analysis of the propagated expected type in
  $$\boxed{\begin{array}{l} [\,] \vDash^{p}_{e} (T,\ \Gamma \vdash^{w} E:\ S \not\nearrow T) \rightsquigarrow \langle (P^{o},\ \Gamma^{o} \vdash^{w}\ E^{o} : T^{o}), \Theta^{\mathsf{o}} \rangle \\ \textsc{slicesPtError}((P^{o},\ \Gamma^{o} \vdash^{w} E^{o}:\ T^{o}),\ \Theta^{\mathsf{o}}\ :::\ \Theta^{\mathsf{sub}}) \end{array}}$$

In the second case we apply the $\textsc{slicesPtError}$ function to the inferred Propagation Root and the *TypeFocus* $(\Theta^{\mathsf{o}}\ :::\ \Theta^{\mathsf{sub}})$. The *TypeFocus* extracts the conflicting part of the fresh prototype P^{f} that was introduced in the Propagation Root judgment, which in turn represents a type selection on the conflicting type elements of the prototype, as required. The latter statement can be visible once we write in full the type selection from the *fresh* prototype:

$$\boxed{\begin{array}{ll} (\Theta^{\mathsf{o}}\ :::\ \Theta^{\mathsf{sub}})(P^{f}) = & \\ \Theta^{\mathsf{sub}}(\Theta^{\mathsf{o}}(P^{f})_{\mathtt{tpe}}) = & \text{(by definition of the } \textit{TypeFocus} \text{ composition)} \\ \Theta^{\mathsf{sub}}([\,](T)_{\mathtt{tpe}}) = & \text{(by definition of the inferred } \Theta^{\mathsf{o}} \text{ from the propagation judgment)} \\ \Theta^{\mathsf{sub}}(T) = & \text{(by definition of } [\,] \text{ type selection)} \\ \mathtt{inl}\ T^{target} & \text{(by soundness Lemma 4.2)} \end{array}}$$

where T^{target} represents the type element of the inherited type T that conflicted with the corresponding type element of the synthesized type S, as required.

4.4 Example: Explaining the type mismatch of the `foldRight` application

Section 3.1 has used visual cues in the partially derivable type derivation trees to provide an informal explanation of the two type mismatch errors. In particular, we have considered a non-trivial example of the failed type inference in the application of the `foldRight` function to the empty list and the anonymous function. Using the identical `foldRight` function application example, we show how in practice we can apply the previously defined translation of the erroneous type derivation trees fragments to trigger the *TypeFocus*-based analysis.

For the purpose of the example, the specification of *TypeFocus* in Definition 4 has to now take into account the possibility of $List$ type constructors. The $\phi_{\texttt{List}}$ *TypeFocus* extracts the type argument from the type application involving $List$ type constructor such that

$$(\phi_{\texttt{List}} :: \Theta')(T) \quad = \quad \begin{cases} \Theta'(A) & \textbf{if} \quad T = \textsc{List}[A] \\ \texttt{inr}\ \langle T,\ \phi_{\texttt{List}} :: \Theta' \rangle & \textbf{else} \end{cases}$$

We extend the definition of the $\vdash^{sub} S \not<: T \to \overline{\Theta}$ judgment (Figure 4.5) to take into account the new type constructor (Figure 4.6).

According to the *TypeFocus* techniques defined in this chapter, any analysis of the type mismatch error begins with translating the failed subtype derivation in the erroneous type derivation tree

$$\textsc{List}[\bot],\ \Gamma, x : Int, y : List[\bot] \vdash^{w} \texttt{Cons}(x+1, y) :_{a \Rightarrow Int} \textsc{List}[a] \not\nearrow \textsc{List}[\bot]$$

to type extractors on the conflicting types in

$$\vdash^{sub} \textsc{List}[Int] \not<: \textsc{List}[\bot] \to \{\, [\phi_{\texttt{List}}] \,\}$$

The inferred type selection can then be used for the analysis of the conflicting types since $([\phi_{\texttt{List}}], \varnothing \vdash^{\texttt{WF}} \textsc{List}[Int])$ and $([\phi_{\texttt{List}}], \varnothing \vdash^{\texttt{WF}} \textsc{List}[\bot])$.

Source of the conflicting type element of the synthesized type $\textsc{List}[Int]$
Using the core *TypeFocus*-based algorithm SLICES (Section 3.5.3) and the SLICESTVAR function analyzing Type Variable Typing Slices (Section 3.7.3), we find the source of the conflicting type argument Int from the synthesized type $\textsc{List}[Int]$. For the purpose of the example we assume the existence of an implicit infix '+' function of type $\forall b.\ (b, b) \to b$ and let $\Gamma' = \Gamma, x : Int, y : \textsc{List}[\bot]$.

A summary of steps that resolve the intermediate *Typing Slices* involves:

1. $\textsc{slices}((\,?,\ \Gamma' \vdash^{w} \texttt{Cons}(x+1, y) :\ _{[a \Rightarrow Int]}\textsc{List}[\,a\,]),\ [\phi_{\texttt{List}}]) =$
   $\{\ \langle \nu_{\texttt{TVAR}},\ (\,?,\ \Gamma' \vdash^{w} \texttt{Cons}(x+1, y) :\ _{[a \Rightarrow Int]}\textsc{List}[\,a\,]),\ [\phi_{\texttt{List}}] \rangle\ \}$

2. $\textsc{slices}\text{TVAR}(\langle \nu_{\texttt{TVAR}},\ (\ \textsc{List}[\bot],\ \Gamma' \vdash^{w} \texttt{Cons}(x+1, y) :\ _{[a \Rightarrow Int]}\textsc{List}[\,a\,]),\ [\phi_{\texttt{List}}] \rangle) =$
   $\left\{ \begin{array}{l} \langle \nu_{\texttt{TVAR}},\ (\,?,\ \Gamma' \vdash^{w} x+1 :\ _{[b \Rightarrow Int]}\,b\,),\ [\,] \rangle, \\ \langle \nu_{\texttt{TSIG}},\ (\,?,\ \Gamma' \vdash^{w} y : \textsc{List}[\,\bot\,]),\ [\phi_{\texttt{List}}] \rangle \end{array} \right\}$

$$\Theta Sub_{(list)}\ \frac{\vdash^{sub} A \not<: B \rightarrow \overline{\Theta}'}{\vdash^{sub} \textsc{List}[A] \not<: \textsc{List}[B] \rightarrow \left\{ \phi_{\texttt{List}} :: \overline{\Theta}' \right\}}$$

$$\Theta Sub_{(list1)}\ \frac{X \not\in \{\textsc{List}[B], \bot, ?\}}{\vdash^{sub} X \not<: \textsc{List}[A] \rightarrow \{[\,]\}} \qquad \Theta Sub_{(list2)}\ \frac{X \not\in \{\textsc{List}[B], \top, ?\}}{\vdash^{sub} \textsc{List}[A] \not<: X \rightarrow \{[\,]\}}$$

...

Figure 4.6: Extension of the $\vdash^{sub} S \not<: T \rightarrow \overline{\Theta}$ judgment definition from Figure 4.5 with covariant Lists. The '...' notation refers to the unchanged rules.

$$\textsc{slices}\text{TVAR}(\langle \nu_{\mathsf{TVAR}}, (?, \Gamma' \vdash^{w} x+1 : {}_{[b \Rightarrow Int]} b), [\,] \rangle) =$$

3. $$\left\{ \begin{array}{l} \left\langle \nu_{\mathsf{TSIG}}, (?, \Gamma' \vdash^{w} x : \boxed{\Gamma'(x)}), [\,] \right\rangle, \\ \left\langle \nu_{\mathsf{TSIG}}, (?, \Gamma' \vdash^{w} 1 : \boxed{Int}), [\,] \right\rangle \end{array} \right\}$$

For clarity the summary highlights the fragments of types in the reported judgments corresponding to the *TypeFocus* values of the Typing Slices. We also notice that the kind of the Typing Slice immediately dictates the *TypeFocus*-based function it can be analyzed with, if necessary.

The analysis of the $\texttt{Const}(x+1, y)$ function application in the second step reveals two type constraints that affect the instantiation of the type variable a. The Typing Slices correspond to the two type constraints $\{Int <: a\}$ and $\{\bot <: a\}$ inferred from the arguments $(x+1)$ and y, respectively. The type of the former type constraint subsumes the latter in the subtyping ordering and can be safely omitted when considering only the relevant type constraints that affect the instantiation of the extracted type variable.

The result, the two final *Type Signature Typing Slices*, determine that the type argument Int of the synthesized type $\textsc{List}[Int]$ was inferred from two terms:

- the variable x (and indirectly its type in the environment).
- the literal constant 1.

Source of the conflicting type element of the inherited type $\textsc{List}[\bot]$

The search for the source of the expected type is more involving because it first has to translate the propagation of the expected type into an equivalent *TypeFocus* instance. For presentation reasons, we let

- $\Gamma' = \Gamma, x : Int, y : \textsc{List}[\bot]$.

- $\texttt{foldRightApp} = \texttt{foldRight}(xs)(\texttt{Nil}())(\texttt{fun}(x, y) \to \texttt{Cons}(x+1, y))$.

In the first steps, the algorithm first resolves the Propagation Root of the inherited type (using the algorithm from Section 4.1), finds the source of the prototype $(Int,\ \textsc{List}[\bot]) \to \textsc{List}[\bot]$ introduced in the Propagation Root in the second step, only to later expand the intermediate *Typing Slices*, if necessary. The individual steps of the analysis can be summarized as:

1. $[\,] \vDash^{p}_{e} (\textsc{List}[\bot],\ \Gamma' \vdash^{w} \texttt{Cons}(x+1, y) : {}_{[a \Rightarrow Int]}\textsc{List}[a] \not\nearrow \textsc{List}[\bot])$
   $\rightsquigarrow \langle (?,\ \Gamma \vdash^{w} \texttt{foldRightApp} : \text{undefined}), [\phi_{\texttt{fun-res}}] \rangle$

2. $\textsc{slicesPtError}((?,\ \Gamma \vdash^{w} \texttt{foldRightApp} : \text{undefined}),\ [\phi_{\texttt{fun-res}}] ::: [\phi_{\texttt{List}}]) =$
   $\{\, tvarSlice \,\}$

   where $tvarSlice =$
   $$\left\langle \begin{array}{l} \nu_{\texttt{TVAR}},\ (?,\ \Gamma \vdash^{w} \texttt{foldRight}(xs)(\texttt{Nil}()) : {}_{[b \Rightarrow \textsc{List}[\bot]]}((Int,\ b) \to b) \to b)), \\ \qquad \phi_{\texttt{fun-param}} :: [\phi_{\texttt{fun-res}}, \phi_{\texttt{List}}] \end{array} \right\rangle$$

3. $\textsc{slicesTVAR}(\ tvarSlice\ ) =$
   $\{\ \langle \nu_{\texttt{TVAR}},\ (?,\ \Gamma \vdash^{w} \texttt{Nil}() : {}_{[a \Rightarrow \bot]}\textsc{List}[\ a\ ] \nearrow ?),\ [\phi_{\texttt{List}}] \rangle\ \}$

4. $\textsc{slicesTVAR}\big(\ \langle \nu_{\texttt{TVAR}},\ (?,\ \Gamma \vdash^{w} \texttt{Nil}() : {}_{[a \Rightarrow \bot]}\textsc{List}[\ a\ ] \nearrow ?),\ [\phi_{\texttt{List}}] \rangle\ \big) =$
   $\{\ \langle \nu^{\bot}_{\texttt{TSIG}},\ (?,\ \Gamma \vdash^{w} \texttt{Nil}() : {}_{[a \Rightarrow \bot]}\textsc{List}[\ a\ ] \nearrow ?),\ [\phi_{\texttt{List}}] \rangle\ \}$

For clarity reasons the summary highlights the fragments of types in the reported judgments corresponding to the *TypeFocus* values of the Typing Slices.

The final *Type Signature Typing Slice* (step 4) identifies the application `Nil()` as the source of the conflicting expected type. More precisely, the kind of the *Typing Slice*, $\nu^{\bot}_{\texttt{TSIG}}$, determines that due to lack of type constraints the type variable a was instantiated to type $\bot$ and the `Nil()` term is the source of the target type $\bot$.

In both of the cases the formal analysis of the erroneous program agrees with our informal explanation from Section 3.1. The analysis is autonomous in a sense that the types of the type mismatch conflict drive entirely the complete analysis of the problem and the Typing Slices are deterministically associated with one of the previously defined *TypeFocus*-based techniques based on the Typing Slice kind alone.

4.5 Final remarks

In this section we have defined a formal approach to analyzing the type mismatch errors in a language that implements the formalization of a variation of Local Type Inference. Our approach integrates seamlessly with the existing *TypeFocus*-based analysis techniques by reducing the non-derivable elements to the *TypeFocus* values. Real-world programs can exhibit different category of errors, such as an inference of type arguments that do not conform to the declared type bounds of type parameters, or failed overloading resolution. In practice, most of them will involve some form of subtyping polymorphism, and we have shown examples where we can translate the latter to *TypeFocus* values with relatively little effort.

The debugging of type errors relies on the fact that the non-derivable type derivation trees preserve the structure and are not immediately discarded. This is an acceptable restriction for languages with local type inference; the errors tend to be heavily localized which allows for a separation of the erroneous *branches* of the type derivation tree and continuation of type checking for the other parts of the program.

In our approach we do not attempt to address programs that exhibit multiple type errors or errors that depend on each other, as it is the case for example in Chen and Erwig [2014b]. This in turn implies that debugging type errors may involve separate type debugging *sessions* for each of the type errors reported for the program. The limitation is acceptable since each of the type errors can be explained in an autonomous way without any user interaction.

Chapter 5

Lightweight extraction of type checker decisions

The *TypeFocus*-based analysis presented in the previous sections navigates type derivation trees in order to explain the typing decisions of the local type inference. The idea of using type derivation trees for type debugging purposes is not novel; previous attempts include prototypes for the OCaml type checker (in Tsushima and Asai [2013]) or some variants of Simply Typed Lambda Calculus (in McAdam [2002]). The past, straightforward approaches hardly apply to non-trivial examples due to incompatibility with the official type checker and a non-autonomous mode of operation that requires constant user feedback. In addition none of the approaches have considered languages using Local Type Inference.

A novelty of our approach lies in obtaining low-level data from the existing type checking runs, all without affecting the compiler's logic or reducing its features. The collected data is then sufficient to create a data structure that closely resembles the desired type derivation trees. The separation of the construction of the derivation has two main implications:

- The low-level representation can be collected using a minimal, non-intrusive instrumentation infrastructure, that minimizes the performance impact of regular, non - debugging compiler runs and is easy to maintain during the usual development of the type checker.

- Expressions having a similar structure may lead to subtle type checking differences, which in turn can lead to different runs of the type checker and different low-level data. By mapping to a common high-level representation we establish a well-defined set of expected type checking rules supported by the process. Algorithms that navigate type checker runs based on the reconstructed high-level representation are statically checked for correctness.

```
def typecheckAst(ast: Tree, pt: Type): Tree = {
  EV ⋘ EV.TypecheckAst(ast, pt)
  ... // instrumented typing of ast
  EV ≪ EV.AstTyped(...)
  ...
  EV ⋙ EV.TypecheckDone(...)
  ...
}
```

(a) A an explicit instrumentation.

```
def typecheckAst(ast: Tree, pt: Type): Tree =
  EV. instrument (EV.TypecheckAst(ast, pt), EV.TypecheckDone(_)) {
    ... // instrumented typing of ast
    EV ≪ AstTyped(...)
    ...
  }
```

(b) A compact version.

Figure 5.1: A brief look at the Instrumentation API.

In this section we will step through the construction of this high-level representation. Section 5.1 discuses the API used for instrumenting the compiler and Section 5.2 discusses how the individual type inference rules, their premises and typing judgments are represented through a high-level representation. Section 5.3 describes a one-to-many translation from low-level data traces to their high-level counterparts; the unambiguity of the mapping is determined by imposing restrictions on the possible definitions of the high-level representation.

5.1 Compiler instrumentation

The type debugger tool collects low-level type checker information by manually instrumenting the existing Scala compiler using a minimal API, a set of low-level instrumentation classes, and an infrastructure for debugging. The instrumentation primarily extracts *raw* type checking information that includes abstract syntax trees, type or symbol references; depending on the fragment of the type checking being instrumented, more specialized type information is collected, *e.g.,* type variable variance information or an inferred type substitution. Listing 5.1 provides a small example of the manually instrumented method that type checks an AST (parameter `ast` of type `Tree`) using the expected type (`pt` of type `Type`).

In the example, value `EV` represents a reference to the instrumentation *universe* that extends the main compiler class, called `DebuggerGlobal`, which controls the execution of the compiler (the implementation-dependent `DebuggerGlobal` class will be discussed in the next chapter). The instrumentation universe defines an abstract base class `Event`, that all the low-level instrumentation classes will extend from, and the instrumentation methods used for reporting them, *i.e.,* $\lll$, $\ggg$ and $\ll$. In the example, values `TypecheckAst`, `AstTyped` and `TypecheckDone` of type `Event`, have been defined in the instrumentation universe (for consistency, we chose to explicitly mention path-dependent types `EV.x`, where `x` represents a member of the universe).

The instrumentation API introduces the notion of an *instrumentation block*, which makes it possible for *structural* information to be collected during instrumentation. The additional property is sufficient for recreating traditional premises-conclusion relations in the typing rules, as opposed to typically "flat" instrumentation data. These instrumentation blocks are delimited by the $\lll$ and $\ggg$ operators, and typically also contain other (potentially nested) instrumentation blocks, delimited using the same operators, as well as single instrumentation points (defined using the $\ll$ operator). As a result, the framework understands that direct instrumentation points between the $\lll$ and $\ggg$ method calls can be considered as type checking dependencies, without having direct references to them in the source code. The equivalent *compact* version of the instrumented code in the second part of the listing is using an `instrument` method; the method *wraps* the type checker code as a by-name argument, and ensures proper *opening* and *closing* of the instrumentation block.

Listing 5.2 presents a (simplified) fragment of the instrumentation universe definition that is available in the compiler. The listing provides an overview of the instrumentation classes and methods, including the convenient overloaded `instrument` method that ensures an appropriate block handling (the difference between the two alternatives stems from the presence of the default *closing* event, or lack thereof). Due to the Scala's optimizer not performing whole-program analysis (Dragos [2008]) most of the methods are marked as final and have an `@inline` annotation. The inlining helps to avoid performance penalties associated with the additional method calls during the regular, non-debugging compiler runs.

The mode of operation of the type checker is determined using the `isOn` method. When the result is a Boolean value `false`, any instances of the instrumentation classes will be discarded. When the result is `true`, the instances of the instrumentation classes are used to construct a *raw* tree representation; the individual instances represent the values in the nodes of the tree, and the parent/child relationship between the nodes is determined by the block opening/-closing information.

The `withNoEvents` method indicates that fragments of the type checker, provided as an argument, will always execute with the instrumentation turned off. The method allows to discard the type checker executions that are implementation-dependent, unsupported, or irrelevant from the analysis point of view, and otherwise would have to be unnecessarily exposed in the high-level representation.

We chose to manually instrument the Scala compiler since the alternative is to modify bytecode (using *e.g.,* http://eclipse.org/aspectj/), which is too coarse-grained; the instrumenta-

```scala
trait EventsUniverse {
  self: DebuggerGlobal =>

  val EV: EventModel

  abstract class EventModel {
    @inline
    final def >>>(x: Event): Unit = if (isOn) { // ... }

    @inline
    final def <<<(x: Event): Unit = if (isOn) { // ... }

    @inline
    final def <<(x: Event): Unit = if (isOn) { // ... }

    @inline
    final def instrument[T](x: Event, y: T => Event)(body: => T): T = {
      <<< x
      val result = body
      >>> y(result)
      result
    }

    @inline
    final def instrument[T](x: Event)(body: => T): T = {
      <<< x
      val result = body
      >>> EV.Done
      result
    }

    @inline
    final def isOn: Boolean = // ...

    @inline
    final def withNoEvents(body: => T): T = // ...

    abstract class Event { ... }
    case class TypecheckAst(tree: Tree, tpe: Type) extends Event
    case class AstTyped(tree: Tree) extends Event
    case class TypecheckDone(tree: Tree) extends Event
    case object Done extends Event
    ...
  }
}
```

Figure 5.2: A brief look at the instrumentation universe.

tion blocks not always align at the entry and exit of some type checker method. An automatic, or semi-automatic, approach to instrumenting the compiler would admittedly be less error-prone but for practical reasons we chose the former. At the same time any bytecode manipulation library would have to be aware of the semantics of the language in which the compiler is written in order not to limit the level of detail of the decision process.

5.2 High-level representation

In this section, we provide a brief overview of our high-level type checking representation. In section 5.3, we delve into the details of how it maps to our low-level instrumentation data.

Listing 5.3 defines a `Goal` class (or represent a type checking decision) that has a reference to all its premises (or dependencies that need to be satisfied), and a conclusion, `parent`, which is also of type `Goal`. The class defines an abstract type member `U` and a member `underlying` of the type `U`; both members statically define a link between the two representations. The bare `Goal` base class is the counterpart of the low-level base instrumentation class (`EV.Event`), in the high-level representation, and the relation is reflected in the upper bound of the abstract type member.

```
abstract class Goal {
  type U <: EV.Event
  def underlying: U

  def parent:     Goal
  def premises:   List[Goal]
}
```

```
abstract class Typecheck extends Goal {
  type U <: EV.TypecheckAst

  def typeg:   TypeGoal
  def adaptg:  AdaptGoal
}

abstract class TypeGoal extends Goal {
  type U <: EV.TypeEvent
}

abstract class AdaptGoal extends Goal {
  type U <: EV.AdaptEvent
}
```

Figure 5.3: Base class of high-level representation, `Base`, and `Typecheck` goal that specifies typing operations that assign type to a generic Abstract Syntax Tree.

The role of the subclasses of `Goal` and their members is to express more concrete requirements of the typing decisions they represent. Subclasses also refine the upper bound of the abstract type member in order to reflect the link to the low-level instrumentation classes they stand for. Through such class hierarchy, nodes of a type derivation tree can be constructed from subclasses of `Goals` and represent numerous typing decisions, *e.g.,* typing of function application or member selection.

Class `Typecheck` is an example of such a subclass; semantically, `Typecheck` represents the decision process of every type inference rule in the Colored Local Type Inference formalization, where we first synthesize the term's type and later adapt the type to the prototype, if necessary.

In order to *satisfy* the `Typecheck` goal, for example, its members require that it first performs a typing operation, as indicated through a `TypeGoal` type, and then an adaptation, as indicated through the `AdaptGoal` type. The refinement of the upper bound in the mentioned classes ties them to their respective, more specialized low-level instrumentation classes.

For convenience, the implementation of each of the high-level abstract classes comes with a companion object that has an appropriately generated `unapply` method, such as

```
object Typecheck {
  def unapply(clazz: Typecheck): Option[(TypeGoal, AdaptGoal)] =
    Some((clazz.typeg, clazz.adaptg))
}
```

for the `Typecheck` class. The provided method of the companion object allows for a convenient pattern matching (Emir et al. [2007]) on type derivation trees.

5.3 Mapping between representations

To understand how we map from low-level instrumentation data to high-level derivation trees, we first compare them side-by-side in the instrumented code that type checks abstractions (in Listing 5.4 and Listing 5.5) and function applications (in Listing 5.6 and Listing 5.7). We explain why a naive approach of a one-to-one mapping from the low-level instrumentation to the high-level representation is inefficient in terms of polluting the code space of the compiler. The one-to-many mapping approach taken in this thesis allows us to reduce the number of instrumentation instructions and define high-level goals that hide the unnecessary details of the implementation. Using a number of examples we illustrate that a one-to-many mapping comes at a cost - ambiguous definitions can lead to different valid mappings where in general we cannot select the most specific one.

Later in Section 5.4 we formally define the one-to-many translation. We show that with a small number of restrictions on the high-level representation we can define rules that non-ambiguously map to them from the low-level instructions representing the type checking.

Towards a high-level representation for type checking functions: An Example

Listing 5.4 presents a simplification of Scala's actual implementation that assigns types to functions. In the example, the `typedFunction` method takes an argument of type `Function` (an AST node for functions), and an expected type of the function. The main instrumentation block (lines 1-12), that creates an instance of a low-level instrumentation class `TypeFun`, de-

```
1  def typedFunction(ast:Function, pt:Type): Function = EV.instrument(EV.TypeFun(ast, pt)) {
2    val Function(params, body) = ast
3    val (paramsPt, resultPt) = decompose(pt) / parameters and the result type parameters
4    val params1 = (params zip paramsPt).map {
5      case (param, paramPt) => typedParam(param, paramPt)
6    }
7    val body1 = typecheckAst (body, resultPt)
8    ...
9    val ast1 = Function(params1, body1)
10   ...
11   ast1
12 }
13 def typedParam(param:ValDef, pt:Type): ValDef = EV.instrument(EV.TypeFunParam(param)) {
14   ...
15   val param1 = typecheckAst (param, pt)
16   ...
17   param1
18 }
```

Figure 5.4: An example of the instrumented method that type checks ASTs of functions. The ... part represents the irrelevant implementation details.

limits the logical block of type checker's executions, and essentially specifies that any other invocation of the instrumentation in between is directly part of type checking the function AST. Similarly, any instrumentation invocation within the second instrumentation block (lines 13-18), is part of the decision process that type checks the type of the parameter in the abstraction. We recall that the previously defined type checking method, `typecheckAst`, is already instrumented (visually represented through a gray box around it) and does not have to be placed within the instrumentation block separately.

The method extracts the individual elements of the `Function` AST (parameters and the body of the function in line 2) and type elements of the prototype (line 3); both involve simple pattern matching on the result of the right hand side of the expression. Later, the `typedParam` function is applied to the individual parameters of the function, along with their respective prototypes (lines 4-6), in order to determine the types of the parameters (the `zip` method combines the corresponding elements of the two collections into tuples). Importantly, the individual instrumentation blocks that track the typing of the parameters of the function, as well as the instrumentation block for typing the body of the function (line 7), are all direct dependencies of the first, main, instrumentation block.

Now that we've seen an example of an instrumentation when type checking functions, we look into the high-level class hierarchy, that can accurately represent it.

Listing 5.5 provides an example of a high-level `TypeFun` class that is a subclass of the `TypeGoal`. The initial verification of the parameters of the function is represented through the `params`

```
abstract class TypeFun extends TypeGoal {
  type U <: EV.TypeFun

  def params: List[TypecheckParam]
  def body:   Typecheck
}
```

```
abstract class TypecheckParam extends Goal {
  type U <: EV.TypeFunParam

  def tParam: Typecheck
}
```

Figure 5.5: The high-level class hierarchy for representing the typing decisions that infer the type of the function, similarly to the various `(abs)` rules in the Colored Local Type Inference formalization.

member, and the verification of the body of the abstraction is represented through the `body` member. Importantly, the variable number of the possible parameters of the function is expressed through the collection class `List` type constructor. The `TypecheckParam` high-level class requires a single type checking operation - the verification of the type of the parameter - in order to be satisfied. Finally, we notice that both high-level classes refine the upper bounds of the high-level classes in order to reflect the types of the low-level events they can represent.

In order to relate the two representations of the type checking, we return to the low-level instrumentation data. These low-level instrumentation events are essentially sequences of type checking events, with an additional hierarchy information gathered from the block delimitations. By pattern matching on such instrumentation sequences, in a postfix fashion, the mapping groups the sequences into individual categories that correspond to the members of high-level classes.

Towards a high-level representation for function applications: An Example

Listing 5.6 presents a simplified view on the Scala's implementation, and instrumentation, for a more involved example that types function applications. In the example, the `typedApplication` method takes an argument of type `Apply` (an AST node for function applications), and an expected type of type `Type`, and returns an AST with the inferred type of the function application, *i.e.,* the AST has a value assigned to its type attribute.

The main instrumentation block (lines 2-8) is delimited by the instrumentation classes `TypeApp` and `TypeAppDone`. As a result, any instrumented typing decision, executed as part of the type checking function application, is essentially part of such instrumentation block. The already instrumented `typecheckAst` method, infers the type of the function, and its type checking decisions are part of the main instrumentation block as well.

The rest of listing 5.6, beginning in line 6, represents the logic that determines the type of the

```
def typedApplication(ast: Apply, pt: Type): Tree =
 EV.instrument(EV.TypeApp(ast, pt), EV.TypeAppDone(_)) {
  val Apply(funAst, argsAsts) = ast
  val fun1 = typecheckAst(funAst, WildcardType)

  if (fun1.tpe == ErrorType) typedApplicationFallback(ast, pt)
  else                       assignAppType(fun1, argsAsts, pt)
 }

def assignAppType(fun1: Tree, args: List[Tree], pt: Type): Tree = {
  EV <<< TypeApp1(...)
  val app1 = fun1.tpe match {
    case MethodType(params, resultTpe) =>
      val paramsTpes = // ...
      val args1 = (args zip paramsTpes).map { case (arg, argPt) =>
                                        typecheckAst(arg, argPt) }
      if (hasError(args1)) ... else ...
    case PolymorphicType(tParams,MethodType(params,resultType)) =>
      val paramsTpes = // ...
      val argsPt = argsPtFromResultPt(resultTpe, tparams, paramsTpes)
      val args1 = (args zip argsPt).map { case (arg, argPt) =>
                                        typecheckAst(arg, argPt) }
      if (hasError(args1)) {
        ...  // fallback mechanism
      } else {
        ...
        inferMethodInstance(args1, fun1, pt, paramsTpes)
      }
    case OverloadedType(_, alternatives) =>
      ...
  }
  EV >>> EV.Done
  app1
}

def typedApplicationFallback(ast: Apply, pt: Type): Apply =
 EV.instrument(EV.InvalidFunApp(...)) {
  ...  // A fallback type checking for an erroneous function type
 }
def argsPtFromResultPt(resultTpe: Type,tparams: List[Symbol],
                       params: List[Type]): List[Type] =
 EV.instrument(EV.InferTypeArguments(tparams, resultTpe, params), EV.InferredPt(_)) {
  ...  // An opportunistic inference of type arguments
       // from the result type of the prototype
 }
def inferMethodInstance(args: List[Tree], fun: Tree, pt: Type,
                        paramsTpes: List[Type]): Tree =
 EV.instrument(EV.InferMeth(args, paramsTpes, pt)) {
  ...  // Inference of type variable instantiations
 }
```

Figure 5.6: Example of the instrumented function that verifies function applications.

function application. The non-trivial implementation fragment defines type checking steps for different scenarios, as expressed by pattern matching on the inferred type of the function term; a function type having some unresolved local type parameters (`PolymorphicType`, line 18), a monomorphic function type (`MethodType`, line 13), an overloaded type representing multiple method alternatives (`OverloadedType`, line 29), and an erroneous function type that can possibly be adapted (`ErrorType`, line 6). Importantly, the different paths for type checking the function application will result in the instantiation of different instrumentation classes, and different sequences of low-level events within the `EV.TypeApp` and `EV.TypeApp1` instrumentation blocks.

Now that we've seen an example of an instrumentation for a method that type checks function applications, we look into the high-level class hierarchy, that can accurately represent such different execution paths.

Listing 5.7 defines a base class `TypeApp` for representing the decisions that type check function applications; the class refines an upper bound of the abstract type member in order to reflect the low-level instrumentation class it links to, and defines a required member, `typecheckFun`. The `typecheckFun` member and its type, determine a single operation that will always have to be executed for the application term - the inference of the type of the function.

The subclasses of the `TypeApp` class, *i.e.,* the `TypeAppFallback` and `TypeAppCorrect` classes, correspond to different type checker executions involving erroneous and error-free results of type checking a function. The exposed fallback mechanism of the type checker does reveal some internal details of Scala's type checking, but at the same time allows us to navigate through the type checking executions of the existing programs.

Class `TypeApplicationMain`, which is listed as a type of the member `typeApp` in the `TypeAppCorrect` class, represents the base type checking decisions that assign the type to the function application, given an error-free function type. The required type checking operations of the mutually exclusive executions paths are specified in the direct subclasses of the `TypeApplicationMain` class, *i.e.,* in the `TypeAppMonomorphic`, `TypeAppPolymorphic` and `TypeAppOverloaded` classes. The hierarchy of subclasses and their members directly reflects the type checking decisions when no type parameters are present, when some local type parameters have to be inferred or when we deal with multiple method alternatives, respectively. All the subclasses have a member named `typecheckArgs`, corresponding to type checking the arguments of the application, but the member itself is not shared through the inheritance. In our approach such member duplication is unavoidable, as the order of the declared members of the high-level classes has to accurately represent the order of the corresponding type checker's decisions.

In the `TypeAppPolymorphic` class, member `targsFromExpectedType` reveals the instantiation of type variables from the expected type of the function application, prior to type checking the arguments; such opportunistic instantiation of type variables is not formalized in any of the discussed formalizations. By instrumenting the existing compiler we can and have to

```
abstract class TypeApp         extends TypeGoal {
  type U <: EV.TypeApp

  def typecheckFun: Typecheck
}
abstract class TypeAppFallback extends TypeApp {
  def typeAdapted:  Typecheck
  ...
}
abstract class TypeAppCorrect  extends TypeApp {
  def typeApp:      TypeApplicationMain
}
                                    _ _ _ _ _ _ _ _ _


abstract class TypeApplicationMain extends Goal {
  type U <: EV.TypeApp1
}

abstract class TypeAppMonomorphic  extends TypeApplicationMain {
  def typecheckArgs:         List[Typecheck]
}

abstract class TypeAppPolymorphic  extends TypeApplicationMain {
  def targsFromExpectedType: InferTArgsFromPt
  def typecheckArgs:         List[Typecheck]
  def inferInstance:         InferMethodInstance
  def typeAppCont:           TypeAppMonomorphic
}

abstract class TypeAppOverloaded   extends TypeApplicationMain {
  def typecheckArgs:         List[Typecheck]
  def inferAlternative:      InferMethodAlternative
  def typeApp:               TypeApplicationMain
}
```

Figure 5.7: A fragment of the high-level representation corresponding to type checker decisions necessary to type a function application, under different scenarios.

expose such decisions because they can affect the inferred type of function application (the member corresponds to the instrumentation of the `argsPtFromResultPt` method in Listing 5.6) and further type checking decisions. Member `inferInstance` serves as an entry point to the act of inferring minimal type parameter substitution, while the `typeAppCont` member defines the type checking of the function application with instantiated type parameters.

The type checking of overloaded methods is represented through the `TypeAppOverloaded` class; in the `inferAlternative` member the type checker infers a single method alternative based on the types of the previously type checked arguments (the `typecheckArgs` member), the inherited expected type and the type of the alternatives, and then repeats the typing of function application, as indicated through the type of the member `typeApp`.

In order to relate the low-level instrumentation events to this high-level representation, we consider the type checking executions for the two simple function applications:

```
val xs: List[Int] = //...
xs.filter(x => x > 0)
xs.map(x => x + 1)
```

Both applications manipulate the '`xs`' collection of type `List[Int]` by either removing the elements that do not satisfy the '`> 0`' predicate, or incrementing all the elements of the collection. The '`filter`' method, as a member of the `List[Int]` collection has no local type parameters, *i.e.,* using our formal notation the type of '`xs.filter`' is $(Int \to Boolean) \to \textsc{List}[Int]$, while the type checking of the application involving '`map`' method has to instantiate a local type parameter, *i.e.,* the type of the '`xs.map`' member selection is $\forall b.(Int \to b) \to \textsc{List}[b]$ in our formal notation. Different type signatures of the methods imply different type checker executions. For example, the first function applications will be defined by a context of a single high-level instance of type `Typecheck` representing the type checking of the anonymous function '`x => x > 0`', while the second application will be defined by a sequence of high-level instances of types `InferTArgsFromPt`, `Typecheck`, `InferMethodInstance` and `TypeAppMonomorphic`, due to the local type parameter instantiation.

Given the declaration of the `TypeApplicationMain` class and its abstract type member `U`, the low-level instance of the `EV.TypeApp1` event can be potentially mapped to three different subclasses. The mapping of the low-level `EV.TypeApp1` event is determined by pattern matching on the types of the instances that constitute the dependencies of the high-level event. The mapping, being a postfix operation, pattern matches on the already mapped dependencies against the types of the members of the high-level classes.

Ambiguous mappings

Our approach uses the types of the declared members of the high-level representation to drive the pattern matching process. While flexible, it can lead to the issue of unpredictable or ambiguous mappings. For example, a member having type `List[T]` implies that zero or more type checking decisions of type `T` (or a subtype of it) have occurred. The type opens the door for different interpretations of valid pattern matching strategies. Before we define restrictions that avoid the undesired or ambiguously looking high-level representations (Section 5.4), we discuss their examples first.

Ambiguous members within the same class definition

To illustrate one of the problems, Figure 5.8 provides a simpler and more intuitive representation for typing functions than the one given in Figure 5.5. To match the high-level representation the low-level instrumentation would have to be modified. The instrumentation for the `typedParam` method in Figure 5.4 involving the low-level `EV.TypeFunParam` event would have to be removed. Consequently, the instrumentation blocks that enclose the type checking of the parameters of the abstraction and its body are all part of the main instrumentation block of type `EV.TypeFun`.

```
abstract class TypeFun extends TypeGoal {
  type U <: EV.TypeFun

  def params: List[Typecheck]
  def body:   Typecheck
}
```

Figure 5.8: An ambiguous declaration of the high-level classes representing the type checking of functions.

For the purpose of the example, we assume the existence of three already mapped high-level instances, denoted as $\{\, x_1,\ x_2,\ x_3 \,\}$. The runtime type of each of the instances is some subtype of `Typecheck`, and the sequence will serve as a context for our postfix mapping strategy. The types and number of high-level goals offers different possible mappings for the members of the class `TypeFun`[1]:

- $[\textbf{params} \rightarrow \{\, x_1, x_2 \,\}, \textbf{body} \rightarrow \{\, x_3 \,\}]$ -
  The mapping reflects that the compiler type checked two parameters of the function and then type checked the body of the function.
- $[\textbf{params} \rightarrow \{\, x_1,\ x_2,\ x_3 \,\}, \textbf{body} \rightarrow \varepsilon]$ -
  The mapping reflects that the compiler type checked three parameters of the function and the body of the function was not verified, *i.e.,* the instrumentation block that encloses the type checking of the body of the function was never executed. For example, a type mismatch for one of the type parameters could prevent the type checking of the body.
  The lack of mapping for the member `body` can be either blamed on the insufficient low-level instrumentation that omitted some execution, or lack of coverage of the high-level representation.

In the given example, it is not possible to determine the source of the ambiguity nor the actual type checker execution, when based solely on the types of the members.

Ambiguous one-to-many mappings

A straight-forward approach to instrumenting the type checker's codebase adds instrumentation blocks at:

[1]The $[a \rightarrow b]$ notation describes the choice to assign high-level values, b, to member a.

1. The beginning and the end of the typing method.
2. For every type checker's logic where its execution may diverge, such as for the conditional blocks or when pattern matching.

Listing 5.9 illustrates the result of following such a proposal to the letter; needless to say, the instrumentation blocks start to become part of the codebase, rather than only complementing it. Such *over-instrumentation* will significantly affect the maintenance of the type checker. On the positive side, the mapping between the low-level instrumentation events and the high-level representation can now be classified as a one-to-one mapping.

In our approach, we elide some of the instrumentation blocks instead, and infer the different type checker execution paths based on the possible sequences of the low-level events. The already discussed type checking of the function application (Figure 5.7) is one example of such an approach.

To consider the other extreme of the instrumentation spectrum, the *under-instrumentation*, we remove some instrumentation blocks from our reference instrumentation example in Figure 5.6. The removal of the low-level `EV.TypeApp1` and `EV.InvalidFunApp` events would lead to a simpler high-level representation, as presented in Listing 5.10. The new high-level hierarchy has four different subclasses of the `TypeApp` base class that model various type checker executions. As a result, any instrumentation block with a low-level `EV.TypeApp` event will have to be mapped to its high-level counterpart in a one-to-many relation.

The proposed simplification is ambiguous. To illustrate, we consider a mapping context with a sequence of the already mapped high-level instances, $\{ x_1, x_2 \}$, where the runtime type of each of the instances is some type S, such that S is a subtype of `Typecheck`. The mapping that is based on the given sequence can lead to different possibilities:

- [**typecheckFun** $\rightarrow \{ x_1 \}$, **typeAdapted** $\rightarrow \{ x_2 \}$] (*for the* `TypeAppFallback` *class*)
  The low-level events represent a valid mapping for the `TypeAppFallback` class, where some function application had to be typed using the fallback mechanism.
- [**typecheckFun** $\rightarrow \{ x_1 \}$, **typecheckArgs** $\rightarrow \{ x_2 \}$] (*for the* `TypeAppMonomorphic` *class*)
  The low-level events represent a valid mapping for the `TypeAppMonomorphic` class, where some function application did not have to instantiate any local type parameters of the function.
- [**typecheckFun** $\rightarrow \{ x_1 \}$, **targsFromExpectedType** $\rightarrow \{ \}$, **typecheckArgs** $\rightarrow \{ x_2 \}$, **inferInstance** $\rightarrow \{ \}$, **typeAppCont** $\rightarrow \{ \}$] (*for the* `TypeAppPolymorphic` *class*)
  The low-level events represent a partial mapping for the `TypeAppPolymorphic` class, where mappings for some members could not be satisfied.
- [**typecheckFun** $\rightarrow \{ x_1 \}$, **typecheckArgs** $\rightarrow \{ x_2 \}$, **inferAlternative** $\rightarrow \{ \}$]
  (*for the* `TypeAppOverloaded` *class*)

```
def typedApplication(ast: Apply, pt: Type): Tree =
 EV.instrument(EV.TypeApp(ast, pt), EV.TypeAppDone(_)) {
  val Apply(funAst, argsAsts) = ast
  val fun1 = typecheckAst(funAst, WildcardType)
  if (fun1.tpe == ErrorType) typedApplicationFallback(ast, pt)
  else                       assignAppType(fun1, argsAsts, pt)
 }

def assignAppType(fun1: Tree, args: List[Tree], pt: Type): Tree = {
  EV <<< TypeApp1(...)
  val app1 = fun1.tpe match {
    case MethodType(params, resultTpe) =>
      EV.instrument(...) {
        ... // same as before
        if (hasError(args1)) EV.instrument(...) {
          ...
        } else EV.instrument(...) {
          ...
        }
      }
    case PolymorphicType(tParams,MethodType(params,resultType)) =>
      EV.instrument(...) {
        ... // same as before
        if (hasError(args1)) EV.instrument(...) {
          ...  // fallback mechanism
        } else EV.instrument(...) {
          ...
          inferMethodInstance(args1, fun1, pt, paramsTpes)
        }
      }
    case OverloadedType(_, alternatives) =>
      EV.instrument(...) {
        ... // same as before
      }
  }
  EV >>> EV.Done
  app1
}

def typedApplicationFallback(ast: Apply, pt: Type): Apply =
 EV.instrument(EV.InvalidFunApp(...)) {
  ...
 }
```

Figure 5.9: Example of the overzealous instrumentation of the function that verifies function applications. In comparison to the initially proposed instrumentation from Figure 5.6, the example instruments every possible type checking path separately.

```
abstract class TypeApp         extends TypeGoal {
  type U <: EV.TypeApp

  def typecheckFun: Typecheck
}

abstract class TypeAppFallback extends TypeApp {
  def typeAdapted:  Typecheck
}

abstract class TypeAppMonomorphic  extends TypeApp {
  def typecheckArgs:        List[Typecheck]
}

abstract class TypeAppPolymorphic  extends TypeApp {
  def targsFromExpectedType: InferTArgsFromPt
  def typecheckArgs:        List[Typecheck]
  def inferInstance:        InferMethodInstance
  def typeAppCont:          TypeAppMonomorphic
}

abstract class TypeAppOverloaded   extends TypeApp {
  def typecheckArgs:        List[Typecheck]
  def inferAlternative:     InferMethodAlternative
  def typeApp:              TypeApp
}
```

Figure 5.10: A fragment of the ambiguous high-level representation corresponding to type checker decisions necessary to type a function application.

> The low-level events represent a partial mapping for the `TypeAppOverloaded` class, where the mappings for some members could not be satisfied and where left empty.

In the above example, none of the two valid mappings is more specific than the other with respect to their members and their types, leading to an ambiguity. The included two partial results illustrate the potential ambiguities that may arise if we allow for approximated mappings. The approximations would require heuristic that order partial mappings with respect to different properties, such as the types of the members, assigned values and their number, or lack thereof. Inevitably, such approximations would add up to the complexity of the high-level representation, making it harder to reason about and satisfy the requirement of precise type derivation tree navigations.

The next section defines the properties of the high-level representation that allow us to avoid such undesirable ambiguous definitions. The restrictions still elide many of the unnecessary instrumentation blocks, making our approach practical, as illustrated at the beginning of the section.

5.4 A translation from a low-level instrumentation to a high-level representation

In this section, we define the semantics of the mapping function which maps instances of low-level events into their high-level counterparts. The algorithm translates the low-level data recursively, in a depth-first postfix manner. In other words, we first map all low-level instrumentation data enclosed within the instrumentation block, and then use the mapped sequence as a context for pattern matching. The pattern matching uses the types of the members of the high-level classes, and their order, to infer the most specific mapping for the low-level event. For a low-level event that does not initiate the instrumentation block, we simply map it in a one-to-one manner.

Definition 11 *High-level Representation.*

T	::=	$\langle E, T, \{ M_1, ..., M_n \}\rangle$	(high-level type with members)
E	::=	EV.e	(low-level events)
M	::=	$\langle x, S\rangle$	(member)
S	::=	List[T]|T	(possible types of members)

To simplify the presentation, we use the notation from Definition 11 that reduces the high-level representation to only the essential elements. The high-level goal is represented as a triple consisting of the type of the underlying low-level event it relates to, E, its immediate super type, and the sequence of its declared members, $\{ M_1, ..., M_n \}$, where $n \geq 0$. The individual members of the classes are represented as tuples of names and their types. The type of the member, S, is either a type of a high-level class, or a, potentially empty, sequence of them, denoted as `List[` T `]`. For the purpose of further discussion, a term *optional member* refers to members which type is `List[` T `]`, for some T.

With such definition in mind, the base high-level class, $Goal$, is equivalent to $\langle$`EV.Event`, $\top,\epsilon\rangle$, where $\top$ is some predefined, language-dependent top type. The definition restricts the declaration of the high-level goals to single inheritance.

Auxiliary functions

Definition 12 gives type signatures of auxiliary functions and properties used for defining the mapping operation. For completeness, in Figure 5.11 we provide the definitions of the latter. The semantics of the functions are as follows:

- The `premises` function returns types of the declared members for the requested type of the high-level class. The returned sequence respects the order of the members in the class.
- The `linearization` function returns the chain of super types up to the type, given as a

Definition 12 *Mapping: Type signatures of auxiliary functions and properties*

`premises`:	$T \to \overline{S}$
`linearlization`:	$(T, T) \to \overline{T}$
`spec`:	$(T, T) \to \overline{M}$
`non-opt`:	$\overline{M} \to S$
`prefix`:	$S \to \overline{T}$
`lub`:	$(T, T) \to T$
`underlying`:	$S \to T$

Generic operations on sequences:

`idx`:	$\forall a.(a, \overline{a}) \to$ `Nat`
`head`:	$\forall a.\overline{a} \to a$
`tail`:	$\forall a.\overline{a} \to a$
`last`:	$\forall a.\overline{a} \to a$

Properties:

`opt`:	$S \to$ `Bool`
`sub`:	$(T, T) \to$ `Bool`
`sub`$_2$:	$(T, T) \to$ `Bool`

second argument of the function. For example, for the class hierarchy from Figure 5.7 `linearization(TypeAppPolymorphic,Goal)` =
{ `Goal`, `TypeApplicationMain`, `TypeAppPolymorphic` }.

- The `spec` function will return a complete sequence of the inherited and declared members of the given type. The resulting sequence essentially represents, what we call, the *specification* of the high-level class, against which pattern matching will be done.

- The partial function `non-opt` returns a type of a first non-optional member, and the `prefix` function returns types of members up to and including the first non-optional one.

- The `lub` function returns the least common super type of the two types of the high-level classes, which is known to always exist in our classes hierarchy.

- The `underlying` function returns the high-level type accepted by the member, irrespective of whether it is optional or not.

The generic partial functions `idx`, `head`, `tail`, and `last`, operate on the sequences of any elements, and, if defined, return the index of the element in a sequence, the first element of the sequence, and the rest, and the last element of the sequence, respectively.

The definition also specifies two properties, `sub` and `sub`$_2$, which use the *linearization* information to determine if the first type is a subtype of a second one, and if any of the types is a subtype of the other, respectively. Finally, the definition provides the `opt` property, which is `true` if the type of the member is *optional*, *i.e.,* it is of shape `List[` T `]` for some T.

$$\texttt{premises}(t) = \{ S_1, ..., S_n \} \text{ where } t = \langle E_t,\ T_t,\ \{ \langle x_1,\ S_1\rangle, ..., \langle x_n,\ S_n\rangle \}\rangle$$

$$\texttt{linearization}(t,t') = \begin{cases} \epsilon & \textbf{if} \quad t = \top \\ \{ t \} & \textbf{else if} \quad t = t' \\ \{ \texttt{linearization}(T',t'); t \} & \textbf{else} \quad \text{where } t = \left\langle E', T', \overline{M} \right\rangle \end{cases}$$

$$\texttt{spec}(t,t') = \{ \langle x_{(1,1)}, S_{(1,1)}\rangle, ..., \langle x_{(1,n_1)}, S_{(1,n_1)}\rangle, ..., \langle x_{(m,1)}, S_{(m,1)}\rangle, ..., \langle x_{(m,n_m)}, S_{(m,n_m)}\rangle \}$$

$$\text{where} \quad \begin{array}{l} \texttt{linearization}(t,t') = \{ T^1, ..., T^m, T^{m+1} \} \\ T^1 = \langle E^1_*,\ T^1_*,\ \{ \langle x_{(1,1)}, S_{(1,1)}\rangle, ..., \langle x_{(1,n_1)}, S_{(1,n_1)}\rangle \}\rangle \\ ... \\ T^m = \langle E^m_*,\ T^m_*,\ \{ \langle x_{(m,1)}, S_{(m,1)}\rangle, ..., \langle x_{(m,n_m)}, S_{(m,n_m)}\rangle \}\rangle \end{array}$$

$$\texttt{non-opt}(m) = \begin{cases} S_1 & \textbf{if} \quad m = \{ \langle x_1,\ S_1\rangle, ..., \langle x_n,\ S_n\rangle \} \textbf{ and } \neg\texttt{opt}(S_1) \\ \texttt{non-opt}(\texttt{tail}(m)) & \textbf{else if} \quad m \neq \epsilon \end{cases}$$

$$\texttt{prefix}(m) = \begin{cases} \{ \texttt{underlying}(S_1) \} \cup \texttt{prefix}(\texttt{tail}(m)) & \textbf{if} \quad m = \{ \langle x_1,\ S_1\rangle, ..., \langle x_n,\ S_n\rangle \} \textbf{ and } \texttt{opt}(S_1) \\ \{ S_1 \} & \textbf{else if} \quad m = \{ \langle x_1,\ S_1\rangle, ..., \langle x_n,\ S_n\rangle \} \textbf{ and } \neg\texttt{opt}(S_1) \\ \epsilon & \textbf{else} \end{cases}$$

$$\texttt{lub}(t,t') = \texttt{lub0}(\texttt{linearization}(t,\texttt{Goal}),\ \texttt{linearization}(t',\texttt{Goal}))$$

$$\text{where}$$

$$\texttt{lub0}(\overline{t},\overline{t}') = \begin{cases} T_1 & \textbf{if} \quad \overline{t} = \{ T_1, ..., T_n \}, \overline{t}' = \{ T'_1, ... T'_m \}, T_1 = T'_1 \\ \texttt{lub0}(\texttt{tail}(\overline{t}),\ \overline{t}') & \textbf{else if} \quad \overline{t} = \{ T_1, ..., T_n \}, \overline{t}' = \{ T'_1, ... T'_m \}, n > m \\ \texttt{lub0}(\overline{t},\ \texttt{tail}(\overline{t}')) & \textbf{else} \end{cases}$$

$$\texttt{underlying}(t) = \begin{cases} T & \textbf{if} \quad t = \textsc{List}[\,T\,] \\ t & \textbf{else} \end{cases}$$

$$\texttt{sub}(t,t') = t \in \texttt{linearization}(t,t')$$

$$\texttt{sub}_2(t,t') = \texttt{sub}(t,t') \vee \texttt{sub}(t',t)$$

$$\texttt{opt}(s) = \begin{cases} \texttt{true} & \textbf{if} \quad s = \texttt{List}[\,T_s\,] \text{ for some } T_s \\ \texttt{false} & \textbf{else} \end{cases}$$

$$\texttt{idx}(x_i,x') = \begin{cases} i & \textbf{if} \quad x' = \{ x_1, ..., x_i, ..., x_n \} \\ \textbf{undefined} & \textbf{else} \end{cases}$$

$$\texttt{head}(x') = \begin{cases} x_1 & \textbf{if} \quad x' = \{ x_1, ..., x_n \} \\ \textbf{undefined} & \textbf{else} \end{cases}$$

$$\texttt{tail}(x') = \begin{cases} \{ x_2, ..., x_n \} & \textbf{if} \quad x' = \{ x_1, x_2, ..., x_n \} \\ \textbf{undefined} & \textbf{else} \end{cases}$$

$$\texttt{last}(x') = \begin{cases} x_n & \textbf{if} \quad x' = \{ x_1, ..., x_n \} \\ \textbf{undefined} & \textbf{else} \end{cases}$$

Figure 5.11: Mapping: Definitions of auxiliary functions and properties

Properties of Mapping

The result of pattern-matching is a finite ordered map, denoted as $\sigma^{\overline{T}}_A$, that is inferred from a sequence of high-level instances. Each key of the map corresponds to the declared, or inherited, member of the high-level class A, and the values are sequences of high-level instances, that are themselves subsequences of $\overline{T}$. Formally, the domain, dom, of $\sigma^{\overline{T}}_A$ is specified as $dom(\sigma^{\overline{T}}_A) = \{ x_{(1,1)}, ..., x_{(1,n_1)}, ..., x_{(m,1)}, ..., x_{(m,n_m)} \}$ where $\langle x_{(i,j)},\ S_{(i,j)}\rangle \in \texttt{spec}(A,\texttt{Goal})$ and $\forall x.\ x \in dom(\sigma^{\overline{T}}_A) \implies \sigma^{\overline{T}}_A(x) \subseteq \overline{T}$. If the member is not optional, then $\sigma^{\overline{T}}_A(x)$ returns either an empty sequence, ϵ, or a single element sequence, $\{ x \}$, for any x, as expected. We determine the correctness of the inferred mapping based on a few properties that we will now define.

Any mapping inferred from a sequence of high-level instances has to respect the original order of the elements from which it was constructed. The property is formally specified in Definition 13. The ordering property ensures that

- The order of members preserves the order of the sequences assigned to the members.
- The order of the high-level instances in every assigned sequence is also preserved.

Definition 13 *Mapping: Order preservation*
Let $\sigma_A^{\overline{T}}$ be the inferred mapping for some high-level type A, and let the mapping be inferred from a sequence of high-level instances $\overline{T}$, then

$$\forall x, y.\ x \in dom(\sigma_A^{\overline{T}}) \land y \in dom(\sigma_A^{\overline{T}}) \land \mathtt{idx}(x, dom(\sigma_A^{\overline{T}})) < \mathtt{idx}(y, dom(\sigma_A^{\overline{T}})) \implies$$
$$\sigma_A^{\overline{T}}(x) = \epsilon \lor \sigma_A^{\overline{T}}(y) = \epsilon \lor (\mathtt{idx}(\mathtt{last}(\sigma_A^{\overline{T}}(x)), \overline{T}) < \mathtt{idx}(\mathtt{first}(\sigma_A^{\overline{T}}(y)), \overline{T}))$$

and

$$\forall x.\ x \in dom(\sigma_A^{\overline{T}}) \land (\sigma_A^{\overline{T}}(x) \neq \epsilon) \implies$$
$$(\forall T_x, T_y.\ T_x \in \sigma_A^{\overline{T}}(x) \land T_y \in \sigma_A^{\overline{T}}(x) \land (\mathtt{idx}(T_x, \sigma_A^{\overline{T}}(x)) < \mathtt{idx}(T_y, \sigma_A^{\overline{T}}(x))) \implies$$
$$\mathtt{idx}(T_x, \overline{T}) < \mathtt{idx}(T_y, \overline{T}))$$

The pattern matching is type-based, meaning that the high-level instances assigned to each of the members conform to the underlying type of the member itself, as described in Definition 14.

Definition 14 *Mapping: Type preservation*
Let $\sigma_A^{\overline{T}}$ be the inferred mapping for some high-level type A, where $A = \left\langle E_A,\ T'_A, \overline{M_A} \right\rangle$, and the mapping be inferred from a sequence of high-level instances $\overline{T}$, then

$$\forall x, S_x.\ x \in dom(\sigma_A^{\overline{T}}) \land \langle x, S_x \rangle \in \mathtt{spec}(A,\ Goal) \implies$$
$$(\forall A_x.\ A_x \in \sigma_A^{\overline{T}}(x) \implies \exists T_x.\ (S_x = \mathtt{List[}\ T_x\ \mathtt{]} \land \mathtt{sub}(A_x,\ T_x)) \lor (S_x = T_x \land \mathtt{sub}(A_x,\ T_x)))$$

A correctly inferred mapping also has to be *complete*, meaning that non-optional dependencies are always satisfied in it, as described in Definition 15.

Definition 15 *Mapping: Completeness*
Let $\sigma_A^{\overline{T}}$ be an inferred mapping for some high-level type A, and be inferred from a sequence of high-level instances $\overline{T}$, then

$$\forall x, S_x.\ x \in dom(\sigma_A^{\overline{T}}) \land \langle x, S_x \rangle \in \mathtt{spec}(A, Goal) \land \neg\mathtt{opt}(S_x) \implies \sigma_A^{\overline{T}}(x) \neq \epsilon$$

Avoiding ambiguity

In order to avoid common programming errors in the definition of the high-level mapping, that would make the process of mapping non-deterministic, we introduce restrictions on the possible definitions of high-level classes. The overall impact of the restrictions is minimal, meaning that a conflict between the definitions of high-level classes can always be resolved through an introduction of a distinct low-level instrumentation block around the ambiguous decisions of the type checker.

Definition 16 specifies a two-part restriction on the definition of the high-level classes which reject among others our ambiguous examples in Section 5.3:

1. The high-level class cannot have two, inherited or declared, optional members that share the same type that are separated by zero or more of other optional members.

2. The high-level class cannot have an optional member next to a non-optional one that shares the same type.

We note that the two types *share the same type* if their underlying types conform to each other. In combination with the ordering property, the restrictions define the notion of a unique assignment of the high-level instances to the members of the same class. Both restrictions rely on the order of the members and their indices in the sequence of inherited and declared members in order to define the *zero or more* and *next* separation between the two members.

Definition 16 *High-level representation: Uniqueness*
Let A be a type of a high-level class, and $\overline{S_A}$ represent types of all, inherited and declared, members of A, such that $\overline{S_A} = \{\, S_{(1,1)},\ \ldots,\ S_{(1,n_1)},\ \ldots,\ S_{(m,1)},\ \ldots,\ S_{(m,n_m)} \,\}$ where $\langle x_{(i,j)},\ S_{(i,j)} \rangle \in \mathtt{spec}(A,\ \mathtt{Goal})$. Then A can be uniquely mapped, denoted as a $\mathtt{uniq}(A)$ property, when

$$
\begin{array}{ll}
 & \forall S_i, S_j.\ \forall T_i, T_j.\ S_i \in \overline{S_A}\ \wedge\ S_j \in \overline{S_A}\ \wedge\ (\mathtt{idx}(S_i, \overline{S_A}) < \mathtt{idx}(S_j, \overline{S_A}))\ \wedge \\
(1) & \quad (\ \mathtt{opt}(S_i)\ \wedge\ \mathtt{opt}(S_j)\ \wedge\ \mathtt{sub}_2(\mathtt{underlying}(S_i), \mathtt{underlying}(S_j))\ )\ \Longrightarrow \\
 & \qquad \exists S_k.\ (\mathtt{idx}(S_i, \overline{S_A}) < \mathtt{idx}(S_k, \overline{S_A}))\ \wedge\ (\mathtt{idx}(S_k, \overline{S_A}) < \mathtt{idx}(S_j, \overline{S_A}))\ \wedge\ \neg\mathtt{opt}(S_k) \\
 & \text{and} \\
 & \forall S_i, S_j.\ \forall T_i.\ S_i \in \overline{S_A}\ \wedge\ S_j \in \overline{S_A}\ \wedge\ (\mathtt{idx}(S_i, \overline{S_A}) < \mathtt{idx}(S_j, \overline{S_A}))\ \wedge \\
(2) & \quad (\mathtt{opt}(S_i)\ \wedge\ \neg\mathtt{opt}(S_j)\ \wedge\ \mathtt{sub}_2(\mathtt{underlying}(S_i), S_j)\ \Longrightarrow \\
 & \qquad \exists S_k.\ (\mathtt{idx}(S_i, \overline{S_A}) < \mathtt{idx}(S_k, \overline{S_A}))\ \wedge\ (\mathtt{idx}(S_k, \overline{S_A}) < \mathtt{idx}(S_j, \overline{S_A}))\ \wedge\ \neg\mathtt{opt}(S_k)
\end{array}
$$

The specification so far has established restrictions with respect to the members and their

```
abstract class Base extends Goal {
  type U <: EV.LowLevel
}
```

```
abstract class HighLevelFoo extends Base {
  def fooA: A
  def fooB: B
}
```

```
abstract class HighLevelBar extends Base {
  def barA: List[A]
  def barB: B
}
```

```
abstract class HighLevelBaz extends Base {
  def fooA: List[A]
  def fooB: List[B]
}
```

Figure 5.12: An example of a high-level class hierarchy that would map some low-level `EV.LowLevel` event in a one-to-many mapping to one of the `HighLevelFoo`, `HighLevelBar` or `HighLevelBaz` classes. The pairs of the disallowed high-level mappings include the high-level classes of `HighLevelFoo` and `HighLevelBar`, and `HighLevelFoo` and `HighLevelBaz`, and `HighLevelBar` and `HighLevelBaz`, given some types `A` and `B`, such that $A \not<: B$ and $B \not<: A$.

types of individual classes. We will now formally define ambiguity conditions for distinct high-level classes that participate in the one-to-many mapping.

Definition 17 defines two conditions that need to be satisfied in order for a type of a high-level class to be non-*ambiguous* with respect to some other type of a high-level class, both of which can map from the low-level instrumentation event of the same type:

1. We ensure that a mapping between the two types that are subtypes can be distinguished thanks to the existence of a non-optional member in the subtype.

2. When the two types are not subtypes of each other, we ensure that a type of the first, non-optional member, will serve as a factor that distinguishes the two possible mappings. In other words, at least one of them has to have a declared, non-optional member, which cannot share the same type with the first declared non-optional member of the other high-level type, nor the types of the declared optional members immediately preceding it.

Figure 5.12 presents examples of pairs of the high-level mappings that will be disallowed through such definition.

Definition 17 *High-level representation: Non-ambiguity*
Let A and B be types of some two distinct high-level classes, where $A = \left\langle E_A, T'_A, \overline{M_A} \right\rangle$ and $B = \left\langle E_B, T'_B, \overline{M_B} \right\rangle$, and $E_A = E_B$. Then, the definition of class A is **not** ambiguous with respect to the definition of class B, denoted as a $\texttt{nonambig}(A, B)$ property, for

(1) If $\texttt{sub}(A, B)$ then
$\exists S_i, x_i.\ \langle x_i,\ S_i \rangle \in \texttt{spec}(A, B)\ \wedge\ \neg\texttt{opt}(S_i)$

(2) If $\neg\texttt{sub}(A, B)$ and $\texttt{lub}(A, B) = C$ then
$\exists S_x.\ \texttt{non-opt}(\texttt{spec}(A, C)) = S_x\ \wedge\ (\forall y.\ y \in \texttt{prefix}(\texttt{spec}(B, C)) \implies \neg\texttt{sub}_2(x, y))$
or
$\exists S_y.\ \texttt{non-opt}(\texttt{spec}(B, C)) = S_y\ \wedge\ (\forall x.\ x \in \texttt{prefix}(\texttt{spec}(A, C)) \implies \neg\texttt{sub}_2(x, y))$

Finally, we notice that a mapping that is inferred from a sequence of high-level instances, $\overline{T}$, has to represent a *complete* matching with respect to $\overline{T}$, meaning that it has *assigned* every element of $\overline{T}$ to one of the members of type A.

Definition 18 *Completeness of matching*
Let $\sigma_A^{\overline{T}}$ be an inferred mapping for some high-level type A, and be inferred from a sequence of high-level instances $\overline{T}$, then
$\forall T_x.\ T_x \in \overline{T} \implies \exists y.\ y \in dom(\sigma_A^{\overline{T}})\ \wedge\ T_x \in \sigma_A^{\overline{T}}(y)$

The above restrictions allow us to find out if translations from low-level events to their high-level counterparts are *safe*. The safety of the translation determines that given a sequence of the high-level instances, and a type of the low-level event, there are no two different ways of performing a one-to-many mapping for a single high-level goal, if possible at all, and that the two possible high-level goal definitions are never ambiguous with respect to each other. The property is formally stated in Definition 19.

Definition 19 *Mapping: Safeness of one-to-many mapping*
Let E be a type of the low-level instrumentation event, $\overline{T}^E$ be a sequence of unique high-level types it can map to, such that $\forall T_E.\ T_E \in \overline{T}^E \implies T_E = \left\langle E,\ T_{super},\ \overline{M_{T_E}} \right\rangle$ for some T_{super} and $\overline{M_{T_E}}$. Then $\overline{T}^E$ represents a safe one-to-many mapping from E iff
$\forall T_E.\ T_E \in \overline{T}^E \implies \texttt{uniq}(T_E)\ \wedge\ (\forall T'_E.\ T'_E \in \overline{T}^E \implies T_E = T'_E\ \vee\ \texttt{nonambig}(T_E, T'_E))$.

Lemma 5.1 uses the previous definitions to state the uniqueness of any one-to-many map-

ping: if there exists a valid mapping for the low-level event, given the sequence of high-level instances as a context for the mapping, then there is always only a single high-level type that it can be mapped to.
In addition, the lemma tells us that if no mapping for the given sequence could be found, the blame lies in either insufficient instrumentation or incompatibility between the low-level and high-level representations, but never because of the ambiguous definition of the high-level representation.

Lemma 5.1 *Uniqueness of one-to-many mapping*
Let $\overline{T}$ represent a sequence of types of the already mapped low-level events, that are direct dependencies of a low-level event E. Let $\overline{T}^E$ represent a *safe* sequence of high-level types the low-level event E can map to, and $\sigma_Z^{\overline{T}}$ denote the inferred mapping for some type Z with respect to $\overline{T}$ that matched *completely* the given $\overline{T}$ sequence.
Then,
$\forall A, B.\ A \in \overline{T}^E\ \wedge\ B \in \overline{T}^E\ \wedge\ (\sigma_A^{\overline{T}}\ \textbf{is defined})\ \wedge\ (\sigma_B^{\overline{T}}\ \textbf{is defined}) \implies A = B$

Proof.
Proof by contradiction. The details of the proof are available in Appendix F.
□

Matching algorithm

For completeness, in this section we discuss a straight-forward inference of the $\sigma_T^{\overline{T}}$ mapping for some high-level type T and a context sequence $\overline{T}$. In the following discussion, we refer to the inherited and declared members of type T as a *specification*, and the high-level instances representing the context for pattern matching as *actuals* (for actual instances that contrast with the expected ones), or simply *context*.

The pseudo-algorithm presented in Figure 5.13 defines a recursive partial function `matching` that compares one-by-one the types of members of the *specification* with the types of the *actuals*, until both of the collections finish. When both of the arguments are empty sequences, the function returns an empty mapping, ε, and the algorithm is finished.
If the type of a member involves `List` type constructor, then the *actuals* are split into a prefix/-suffix pair by the auxiliary function `matching0`; the sequence is split based on the underlying type of the optional member. Later, the algorithm continues the *specification*/*actuals* matching with the suffix part, if possible.

The definition retrieves the runtime type of high-level instances through an implicitly defined

$$\texttt{matching} : (\overline{M}, \overline{\texttt{Goal}}) \rightarrow \sigma$$

$$\texttt{matching}(\epsilon,\ \epsilon) \quad = \quad \varepsilon$$

$$\texttt{matching}(\{\ M_1,\ \dots,M_n\ \},\ \{\ a_1,\ \dots,\ a_m\ \})\ \textbf{if}\ (n > 0) \quad =$$

$$\begin{cases} [x \rightarrow \overline{a}_1] \ \cup\ \texttt{matching}(\overline{M}', \overline{a}_2) & \textbf{if} \quad M_1 = \langle x,\ \texttt{List}[\ T_x\ ]\rangle \ \wedge \\ & \quad \texttt{matching0}(T_x, \{\ a_1,\ \dots,\ a_m\ \}) = \langle \overline{a}_1, \overline{a}_2 \rangle \\ [x \rightarrow \{\ a_1\ \}] \ \cup\ \texttt{matching}(\overline{M}', \overline{A}') & \textbf{else if} \quad (m > 0)\ \wedge\ M_1 = \langle x, T_x\rangle \ \wedge \\ & \quad \texttt{sub}(\texttt{runtimeTpe}(a_1), T_x) \end{cases}$$

$$\text{where}\ \overline{M}' = \texttt{tail}(\{\ M_1,\ \dots,M_n\ \})\ \text{and}\ \overline{A}' = \texttt{tail}(\{\ a_1,\ \dots,\ a_m\ \})$$

$$\texttt{matching0} : (T, \overline{\texttt{Goal}}) \rightarrow \langle \overline{\texttt{Goal}}, \overline{\texttt{Goal}} \rangle$$

$$\texttt{matching0}(T_x,\ \epsilon) \quad = \quad \langle \epsilon, \epsilon \rangle$$

$$\texttt{matching0}(T_x,\ \overline{a}) \quad =$$

$$\begin{cases} \langle \{\ \texttt{head}(\overline{a})\ \} \cup \overline{a}_1,\ \overline{a}_2 \rangle & \textbf{if} \quad \texttt{sub}(\texttt{runtimeTpe}(\texttt{head}(\overline{a})), T_x)\ \wedge \\ & \quad \texttt{matching0}(T_x, \texttt{tail}(\overline{a})) = \langle \overline{a}_1, \overline{a}_2 \rangle \\ \langle \epsilon,\ \overline{a} \rangle & \textbf{else} \end{cases}$$

Figure 5.13: Overview of the matching algorithm implemented in terms of the `matching` partial function. The function takes a sequence of members, and their types, and a sequence of already mapped high-level instances, and returns the inferred mapping, if possible.

function `runtimeTpe`[2]. The runtime type information of the high-level instances, can then be used in the `sub` subtyping tests, to determine if the type of the high-level goal shares the same type with the expected type of the member.

The algorithm realized by the *matching* function is greedy, in a sense that when mapping against an optional member it does not attempt to look-ahead the *specification* to find if the high-level instances might be mapped to sometime later. The approach is in agreement with our formally defined restrictions and delivers predictable and easy to process results.

5.5 Discussion

A reconstruction of the high-level representation, performed during the debugging runs of the type checker, will have a non-negligible impact on the running time of any type debugging analysis. In Section 5.5.1 we present important insights on how to take the advantage of local type inference in order to reduce the footprint of the reconstructed type derivation

[2]In the Scala implementation the runtime type information of the high-level goals has to be explicitly passed around using `TypeTags`, as explained in http://docs.scala-lang.org/overviews/reflection/typetags-manifests.html, since static type information is erased.

trees. Later we discuss the advantages of statically-checked algorithms that navigate through the decisions of the type derivation trees (Section 5.5.2). We conclude with a set of guidelines for manually instrumenting a generic type checker using Local Type Inference (Section 5.5.4) and the applicability of our technique when instrumenting the compiler of Scala or Java (Section 5.5.3).

5.5.1 Decreasing the instrumentation footprint

Factors affecting the size of the constructed type derivation trees, among others, include the size of debugged programs, the level of detail of the exposed type checker decisions, or the presence of advanced type system features. As a result, the runtime execution and the memory consumption of the technique proposed in the previous section can vary significantly, from negligible to unacceptable even for Scala programs having ~ 300 LOC.

In the following discussion we address the three important inefficiencies of the proposed low-level/high-level representations:

- The creation of complete type derivation trees.
- The construction of intermediate, *raw*, instrumentation trees.
- The instrumentation of compiler hot-spots.

Direct construction of high-level type derivation trees

The presence of an intermediate, unstructured tree is useful from the implementation point of view but at the same time inefficient since it means that any type derivation tree is constructed twice. As a solution we use the fact that the translation of low-level events is a postfix operation and can be performed *on-the-fly* during an execution of the type checker in a debugging mode.

To construct the high-level representation directly the instrumentation framework will keep track of the still *unmapped* low-level events and of the already constructed parts of the high-level type derivation tree. In other words, whenever the type checker execution enters a typing decision that starts a new instrumentation block the issued low-level data is pushed onto a stack of yet unmapped events. The individual low-level elements will be translated to their high-level counterparts in a recursive fashion. On exit from the instrumentation block we pop the low-level event from the top of the stack and infer the correct high-level representation based on the already mapped context. The approach is possible because the context of the mapping is fully encapsulated in the nested high-level instances and not propagated from the outside.

Lazy type initialization

The instrumentation framework presented so far considers complete type derivation trees, thus ignoring the locality property of local type inference. We refine such basic definition by first identifying a few of the properties of the local type inference, and its implementation, that allow us to construct only partial high-level derivation trees and still accurately represent the type checking of fragments of programs. We assume that programs are scanned and parsed in a negligible amount of time when compared to type checking phase, leading to type-less trees that preserve the necessary source code positions. The *raw* trees can then be used to locate a minimal subtree that encloses the fragment of the program to be debugged.

The delayed initialization

The Scala compiler initializes symbols and their types in a *lazy* manner; the *lazy* type assignment is realized by the compiler phase preceding the actual type checking, the name analysis phase. The lazy initialization is realized by assigning the internal type value of type `TypeCompleter` (where `TypeCompleter` is a subtype of the Scala's type `Type`) as types of the publicly accessible members, such as methods, abstract types, values or classes. The values of type `TypeCompleter` can be conceptually treated as closures that delay the computation of the type of the declaration, by having a reference to the AST of its type annotation or, on lack of it, an AST of the expression that allows to infer its type. The on-demand *completion* of the delayed computation will force the verification of the type of the definition on a first attempt to retrieve its value. The latter process differs in no way from the previously defined and exposed type checking operations.

The lazy initialization presents important optimization opportunities when constructing high-level type derivation trees for sequences of statements, such as definitions of bodies of classes, or bodies of methods. The traditional model for type checking a block of statements proceeds in a sequential manner. Given a request to expose the decisions of a compiler for a fragment of a program, we only type check statements (and thus issue the low-level instrumentation events) that are enclosed or enclose such a demanded fragment. The filtering of relevant statements is based purely on the (range) position information of the source code and the individual AST statements.

The partial type derivation trees constructed from such *selective* instrumentation, still represent the detailed representation of the relevant type checking execution. The partial type derivation trees differ in the order of the initialization of the declarations, or lack thereof, since the compilation will always trigger the verification of the used symbols when needed. In other words, the construction of partial type derivation trees relies only on the correctness of the existing delayed initialization implementation in the Scala compiler rather than on a non-trivial static reachability analysis of programs or domain-specific heuristics (Pavlinovic et al. [2014]).

As a side-effect of the selective instrumentation and the lazy initialization, the low-level events representing the instrumentation blocks of the delayed initialization can be triggered at any point during type checker's execution. The non-determinism makes it also impractical to model the dependency on the delayed initialization decisions through the statically defined members of the high-level classes. We first briefly discuss how the lazy initialization is modeled in our low-level representation, and later report on our solution.

Representing the delayed initialization in type derivation trees

The low-level events representing the initialization of definitions and their types extend the `EV.NamerEvent` class (named for the compiler phase where they are instantiated). Figure 5.14a provides the definition of the base class, along with the complete set of low-level subclasses, used for representing the initialization process in the Scala's type checker. The classes represent the initialization of type signatures of classes, objects, values, methods and type members, respectively, and take as arguments only their corresponding Scala's ASTs.

```
sealed abstract class NamerEvent               extends Event
case class ClassSigNamer(classDef: ClassDef)    extends NamerEvent
case class ModuleSigNamer(moduleDef: ModuleDef) extends NamerEvent
case class ValSigNamer(valDef: ValDef)          extends NamerEvent
case class MethodSigNamer(methDef: DefDef)      extends NamerEvent
case class TypeDefSigNamer(tpeDef: TypeDef)     extends NamerEvent
```

(a) The low-level classes for representing the lazy initialization of definitions in Scala. The classes are defined within the *instrumentation* universe (Figure 5.2).

```
sealed abstract class NamerGoal extends Goal {
  type U <: EV.NamerEvent
}
abstract class MethodSignature {
   type U <: EV.MethodSigNamer

   def returnTpe: Typecheck
   def params:    List[Typecheck]
}
```

(b) The high-level base class for representing the lazy initialization of definitions in Scala, `NamerGoal`, and an example of a high-level class for representing the initialization of the method definition in Scala.

Figure 5.14: A comparison of the low-level and high-level instrumentation classes for representing the delayed type initialization.

The low-level events and their dependencies are translated to their high-level counterparts using the regular mapping technique. Figure 5.14b provides a definition of an abstract class `NamerGoal`, that is the high-level counterpart of the low-level `EV.NamerEvent` class, and the high-level class representing the initialization of the method's definition, `MethodSignature`. The `MethodSignature` class requires the type checking of the method's returns type and type checking of its parameters' types.

The high-level classes representing the delayed initialization are defined in a space separate from the rest of the type checking representation. This means that neither `NamerGoal` type, nor its direct or indirect subtypes have to be present among the types of members of the previously discussed high-level classes. The low-level events of `EV.NamerEvent` are mapped in a one-to-one mapping to their high-level counterparts, and their high-level goals can be excluded from the mapping context. Consequently, the subtrees modeling the high-level decision process of the delayed initialization are detached from the main type derivation tree and their roots are available and searchable in a flat space of local type derivation trees.

Representing frequently executed type operations

The *TypeFocus*-based approach to understanding type derivation trees (Chapter 3) analyzes the internal details of subtyping derivations, among others, to understand how type variables are instantiated. In fact, a type-driven implicit resolution (Oliveira et al. [2010]) along with subtype checking, are the two most commonly executed operations during the compilation. Consequently, the implementations of the two type checking operations also represent some of the main hotspots of the Scala's compiler and their computations are internally aggressively cached. From the type debugging point of view, such a caching is problematic because

- The equivalent typing decisions may be represented through different type derivations, when cached results are returned.
- The caching operation is not transparent to the users of the instrumentation framework.
- The caching cannot be globally disabled, even for the type debugging mode, for legitimate performance concerns.

In our approach, the frequently executed operations are also instrumented using the lightweight approach of low-level events except that issuing of those events is conditional.

To allow for a conditional instrumentation, the instrumentation universe from Figure 5.2 is extended with an `instrumentCond` method as defined in Figure 5.15. In contrast to the `instrument` method (Figure 5.2), the conditional instrumentation is controlled through a separate instrumentation flag, the `isCond` method, returning a Boolean value.

The `instrumentCond` method controls only the instrumentation of the enclosed low-level events and it needs to enclose only the entry and exit point of the frequently executed typing operations. By default, even during the debugging type checker runs, the `isCond` flag returns value `false` and the instrumentation block defined using the `instrumentCond` method issues a individual low-level (typing operation-specific) *stub* event that is mapped to the high-level *stub* goal in a one-to-one relation (line 11). The high-level *stub* goal has no dependencies, *i.e.,* the class declaration has no inherited or declared members, and its underlying

```
1  @inline
2  final def instrumentCond[T](x: Event)(
3   startEvent: => Event, stubEvent: T => Event, res: T => Event)(body: => T): T = {
4    if (isCond) {
5      EV <<< startEvent
6      val result = body
7      EV >>> resEvent(result)
8      result
9    } else {
10     val result = withNoEvents { body }
11     EV << stubEvent(result)
12     result
13   }
14 }
```

Figure 5.15: The definition of the `instrumentCond` method that allows for disabling locally the execution of the instrumentation for hotspot operations.

low-level *stub* event only keeps a reference to the input and output of the computation. By an application of the helper function `withNoEvents` (line 10), the conditional instrumentation disables the instrumentation of the type checker operation provided as the argument of the instrumentation block.

In order to recover the *lost* information, the instrumentation framework uses the fact that the frequently executed operations, such as subtype checking or implicit resolution, are idempotent when caching is turned off. Assuming that the operations themselves are exposed through the reflection API or the compiler infrastructure the low-level data included in the stub event provides sufficient information to re-execute the operations on demand. For correctness, the caching of the type checker's results has to be controlled through the `isCond` flag and is disabled if and only if `isCond` returns `true`. The stream of low-level events resulting from the instrumented executions of local operations is sufficient to construct local type derivation trees on demand.

Instrumenting the frequently executed operations

The users of the instrumentation framework should not be aware of the existence of the high-level *stub* classes, nor of the high-level classes that model the internals of the frequently executed operations, for concerns of code duplication and logical errors. Instead, any type debugging framework built on top of the instrumentation framework should provide a well-defined API that abstracts over the implementation details of the representation.

On the example of the implementation of the algorithmic subtyping in Scala we now explain the challenges of instrumenting the frequently executed operations. Due to a number of possible combinations, and frequent fallbacks on failures, the heavily optimized implementation is mostly written in an imperative style. The sheer number of choices, not isolated through functions or methods, and the similarity of subtyping operations between the elements of the involved types, makes it difficult to define an instrumentation that is consistent, *i.e.,* it is applied in the same way to a majority of the subtyping rules, and non-intrusive, *i.e.,* it limits the number of the instrumentation blocks, and that maps directly to a high-level representation in a way that is easy to define.

In our approach, the subtyping algorithm is enclosed within a logical instrumentation block and still mapped in a one-to-many fashion. To provide a non-ambiguous translation we instrument the beginning of every subtyping case (or *mark* it) with a single « instrumentation method call. To visually illustrate the solution, we present a small fragment of the instrumentation for a single case of the existing subtyping algorithm implementation:

```
1  def <:<(tp1: Type, tp2; Type): Boolean =
2   EV.instrument(EV.Conformance(tp1, tp2), EV.ConformanceResult(_)) {
3    tp2 match {
4      ...
5      case MethodType(params2, res2) =>
6        tp1 match {
7          case MethodType(params1, res1) =>
8            EV « EV.CompareMethod(tp1, tp2)
9            ...
10           (params1 zip params2).forall(_ =:= _) && res1 <:< res2
11         case _                        =>
12           EV « EV.FailedSubtyping(tp1, tp2)
13           false
14       }
15     ...
16   }
17  }
```

In the example, the `<:<` method is the entry point in the compiler to any subtype checking between the two types. The subtyping algorithm pattern matches on the values of types `tp1` and `tp2` in order to identify the individual subtyping rules. To extract the information, the entry point of the method is enclosed within the instrumentation block (line 2) and its mapping will be inferred from the instructions issued within it.

The fragment describes the instrumentation of the comparison of two method types (lines 5-7), which starts by issuing a low-level instrumentation event `CompareMethod`. Importantly, the instrumentation leaves the type equality between the parameters of the type (using the `=:=` method), and the subtyping check of the return types (using the `<:<` method) intact, meaning it is not necessary to encapsulate them in a separate instrumentation block at the level of the subtyping case. The technique applies seamlessly to instrumenting and exposing failed subtyping checks when the two types cannot be compared (line 17). The individual low-level marker events are mapped in a one-to-one fashion, and their sole purpose is to guide the one-to-many mapping. The presence and position of the unique *marker* type as a type checking dependency allows for pruning a substantial number of the high-level classes, as dictated by our matching algorithm.

Having seen the low-level details of the instrumentation, we now can compare it with its high-level representation.

Figure 5.16 presents an example of a high-level representation for the three subtype checking rules, all of which extend the base high-level representation, the `Conformance` class. The high-level classes of the low-level *marker* events are also present as subclasses of a `CompareFlag` base class.

The `ConformanceMethodType` class in Figure 5.16b gives a high-level representation for a possible subtype checking of method types - the discussed `=:=` type equality check is represented through the `params` member, and the subtyping check between the return types is represented through the `returnTpe` member dependency. Both dependencies translate naturally to a high-level representation. The lazy evaluation of the `&&` operator is also reflected in the type of the `returnTpe` member in order to allow for all possible subtyping executions.

The `ConformanceTypeRef` and `ConformanceTypeRefFallback` classes in Figure 5.16c represent faithfully the different comparison strategies between the two reference types [3], such as `List[Int] <:  List[Nothing]`. The members of the `ConformanceTypeRefFallback` class define the fallback steps when two type references do not immediately represent the same type, such as `List[Int] <:  Nil`, *i.e.,* the subtype checking will be attempted again as indicated by the `retry` member.

The high-level representation of the subtype checking has shown two, equally valid, ways to representing failures and fallbacks in the high-level representation - either through the optional types or the class inheritance. The choice depends on the type checking circumstances but it is important to point out that the latter can be always represented through the former approach but the conversion in the other direction is not always possible, due to the matching strategy.

With such definition in mind, all three subclasses of the `Conformance` class (and others, omitted from the discussion) will lead to a relatively simple mapping of the low-level instrumentation block of `EV.Conformance`. The non-inheritance restriction of the *marker* classes is enforced through a refinement of the upper bound of the type member `U` in each of the high-level marker classes. The usage of the high-level `CompareFlag` *markers* comes at a cost - their dependencies are being leaked into the public signatures of the subtyping high-level classes. This is an acceptable limitation for the frequently executed operations which are more likely to be immediately translated to their equivalent *TypeFocus* values.

[3]A full description of the reference types is available at http://www.scala-lang.org/api/2.10.4/index.html#scala.reflect.api.Types\protect\T1\textdollarTypeRef.

```
abstract class Conformance extends Goal {
  type U <: EV.Conformance
}

abstract class CompareFlag extends Goal {
  type U <: EV.CompareFlag
}
```

(a) A high-level base class representing the results of subtype checking.

```
abstract class ConformanceMethodType extends Conformance {
  def flag:            CompareMethod
  def params:          List[TypeEq]        // Compare types of parameters
  def returnTpe:       Option[Conformance] // Compare return types
  def result:          ConformanceResult   // Result of conformance
}

abstract class CompareMethod extends CompareFlag {
  type U <: EV.CompareMethod
}
```

(b) A high-level subtype checking between the two method types.

```
abstract class ConformanceTypeRefBase {
  def flag:         CompareTypeRef
  def prefix:       Conformance            // Compare "prefixes" of types
  def targs:        List[ConformanceTArgs] // Compare type arguments
}

abstract class ConformanceTypeRef extends ConformanceTypeRefBase {
  def result:       ConformanceResult      // Result of conformance
}

abstract class ConformanceTypeRefFallback extends ConformanceTypeRefBase {
  def flagFallback: CompareTypeRefFallback
  def retry:        Conformance            // Re-try conformance check
  def result:       ConformanceResult      // Result of conformance
}

abstract class CompareTypeRef extends CompareFlag {
  type U <: EV.CompareTypeRef
}

abstract class CompareTypeRefFallback extends CompareFlag {
  type U <: EV.CompareTypeReFFallback
}
```

(c) A high-level representation for the conformance of two reference types, and, in case of failure, the fallback subtyping mechanism.

Figure 5.16: A selection of a high-level representation describing the subtype checking between the two types.

```
1 val root: EV.TypeFun = // ...
2 root.premises.collect{
3   case param: EV.TypeFunParam => param
4 }.flatmap(_.premises).collect {
5   case tcheckParam: EV.TypecheckAst =>
6     tcheckParam.tree
7 }
```

(a) Navigating the *raw* type derivation trees

```
1 val root2: TypeFun = // ...
2 root2.params.flatMap(_.tParam.underlying.tree)
```

(b) Navigating the high-level type derivation trees

Figure 5.17: A comparison of navigating type derivation trees that are constructed only from the low-level instrumentation data, and when they are based on the high-level representation layer. The example uses the class hierarchy from Listing 5.5.

5.5.2 Navigating the type checker decisions

The duality of the low-level and high-level representation provides sufficient information to define algorithm that navigate the decisions of the type checkers and can ignore the low-level details when they are irrelevant. In Figure 5.17 we compare the two approaches for traversing the decisions that type check function ASTs. The first code snippet uses only the low-level instrumentation data and the unstructured instrumentation blocks, and the other relies on the recreated high-level counterpart. The two code snippets attempt to extract the low-level instrumentation data representing the ASTs of all type checked parameters of the function.

For navigating the *raw* type derivation tree, we assume that the base class `EV.Event` provides an implicit `premises` member that retrieves all direct dependencies of the event in a list collection (the List collection provides the usual higher-order functions such as `flatmap`, `map` or `collect`).

In the first case the starting point, the *root* of the considered subtree, is a low-level event of type `EV.TypeFun` (line 1). In order to retrieve the AST information from the dependencies one has to manually filter the correct low-level instances of the events: pattern matching on the `EV.TypeFunParam` type retrieves only the low-level events representing the type checking of the parameters (line 3), and one needs to process further the dependencies of the resulting low-level events (line 4); by pattern matching on the `EV.TypecheckAst` type (line 5) we can finally filter the low-level events representing the type checking of the individual type parameters. A simple example indicates that the manual navigation through the *raw* type derivation trees is an error-prone process and building a type debugging tool on top of such representation would be unrealistic.

In the second case, the *root* of the considered subtree is a high-level goal of type `TypeFun` (line 1), such that `root2.underlying` refers to the low-level `EV.TypeFun` event in accordance with our mapping specification. Defining navigation combinators through such reconstructed high-level type derivation tree comes down to performing regular member selection, where the types of the qualifiers involved the already declared `TypeFun` and `TypecheckParam` classes. The high-level goals are complemented by the low-level data and the access to it is verified by the `underlying.tree` member selection in line 2. The second approach is not only shorter but also statically verified.

5.5.3 Guidelines on instrumenting the existing type checkers

The initial instrumentation of any type checker for an industry-used language is a non-trivial task requiring a good understanding of the implementation. We discuss a set of incremental steps that can guide the integration of the described instrumentation infrastructure in a systematic way.

Instrumenting the implementation of the type checker starts with identifying the main entry points to the type checking process and enclosing them with the instrumentation blocks. The low-level events reported at such entry points will serve as the root(s) of any of the derived representations. If the idempotent operation represents a particular hotspot of the compiler, then one should use the conditional instrumentation blocks.
The next step requires enclosing the bodies of *every* type checking logic that infers the type of the individual ASTs, *e.g.,* the inference of the type of a function, a literal, an identifier, a member selection, or a function application. The low-level events of such instrumentation blocks share an immediate super type, and keep a reference to the inspected ASTs and the propagated expected type at a given point of type checking. Such an optimistic instrumentation does not yet attempt to distinguish between different type checker runs that can take place within the instrumentation blocks.

With such an instrumentation in place, the compiler programmers have to design an appropriate high-level class hierarchy that accurately reflects the main typing dependencies of each of the individual instrumentation blocks. The straight-forward approach of instrumenting only the type checking entry points will have to be refined on a case-by-case basis for each of the considered AST nodes. The different type checking runs for the AST nodes result in different sequences of low-level events which in turn can lead to ambiguous one-to-many mappings. To disambiguate the mappings one has to add the auxiliary instrumentation blocks that logically group together similar typing decisions.

The initially steep instrumentation curve, provides an instrumentation framework that can be later expanded in an incremental manner to support more language features. With the `withNoEvents` method of the instrumentation universe we define a fine-grained control for disabling the instrumentation of the unsupported language features and type checker executions. The support for the individual type checker runs can be added incrementally, on a case-by-case basis.

An Example: Instrumenting the Scala type checker

The instrumentation infrastructure proposed in this chapter has been implemented in the Scala's type checker (for versions 2.10.4 and 2.11.1). We now briefly summarize the key instrumentation points of the implementation.

The main local and non-local type checking entry points in the Scala implementation include:

- The type checking of the compilation unit, *i.e.,* the non-local *root* of any Scala source file.
- The implicit search for a value that conforms to the expected type, given the source code location that triggered the search in the first place.
- The subtype checking between the two types.
- The overload resolution method, that checks if one type is strictly more specific than some other type, based on the specification of the language. Similarly to the previous two operations, the overloading resolution is conditionally instrumented.
- The delayed initialization.

Each of the above entry points can serve as a *root* of a local or a non-local type derivation tree, and is handled accordingly in the infrastructure.

The `typecheckAst` method from Listing 5.1 serves as an entry point to type checking of any AST. The method is responsible for delegating the type checking to the AST-specific typing method, and later performs the adaptation to the expected type. The resulting high-level representation has already been discussed in the case of Listing 5.3.

With ~ 55 kinds of ASTs that cover the core of the different term and type syntax trees in the Scala compiler, the high-level representation provides a comparable number of *base* low-level and high-level classes. Due to the possible fallback mechanisms and the semantical differences of the ASTs of the same kind, the type checker runs for the same AST nodes can result in different sequences of low-level events. Therefore, a high-level representation that models the AST-typing decisions only, more than doubled the number of the required high-level classes.

For feature completeness, our instrumentation framework had to expose in detail, among others, the decision process of inferring the type variable instantiation for polymorphic methods, expressions, arguments and constructors, as well as the algorithms implementing the implicit resolution or subtyping.

The instrumentation instructions are not performance negligible, since even with the presence of inlining they still involve simple flag checks during the regular compilation. Due to a bug in the Scala's inliner, an overall impact of the instrumentation on the regular, non-debugging compilation times let to around 8%-10% performance degradation. The compilation times of the type checking with the instrumentation turned on largely depend upon the size of the selection of type debugging, and range from barely noticeable to very slow. In the case of techniques providing improved error feedback we believe it to be an acceptable behavior.

An Example: Instrumenting the Java type checker

The instrumentation infrastructure implemented in our prototype targets the Scala compiler but the lightweight instrumentation approach itself can apply to other mature type checker implementations. As an example, we will give an overview of our instrumentation technique can model the decision process of the Java's type checker.

The main context-dependent analysis phase of the Java compiler is implemented using the visitor design pattern[4]. The design choice resembles the implementation of the Scala type checker, which pattern matches on the instances of the individual AST nodes in the `typecheckAST` method, in order to delegate to an AST-specific typing logic. Importantly, the visitor pattern implies that the individual AST-specific methods of the visitor class can be conveniently enclosed using the instrumentation blocks of our infrastructure and delimit the type checking decisions of the Java AST nodes.

To illustrate the process of modeling type checking decisions of the individual Java AST nodes, we briefly describe the decision process of the `visitApply` method, which determines the type of the method invocation AST (for details we refer the reader to the source code). The `visitApply` method takes a parameter representing the AST of method invocation in the Java compiler, and a local context information (includes the expected type of the AST), and returns the result type of the method with instantiated local type parameters, if necessary. The order of typing decisions for type checking function applications differs from the Scala's implementation: in short, the Java type checker verifies the type of the arguments and type arguments, and type checks the method using the inherited expected type and the verified types. The Java compiler has a considerably smaller number of fallback mechanisms (the implicit resolution mechanism is not present in Java) leading to a simple instrumentation in general.

In Figure 5.18 we propose a (simplified) class hierarchy that models the type checking of function applications in the Java compiler. The two main classes, `TypeConstrApp` and `TypeMethodInvoc`, reflect the two diverging executions of the compiler that assign types to constructor invocations and regular polymorphic method invocations, both of which are handled in the `visitApply` method. The common starting point is reflected in the super type of both of the

[4]The details of the *attribute*, or type, assignment to trees are available at http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/tip/src/share/classes/com/sun/tools/javac/comp/Attr.java

```
abstract class TypeApp extends TypeGoal {
  type U <: EV.TypeApp
}

abstract class TypeConstrApp
 extends TypeApp {
  def constr: VerifyConstr
  ...
}

abstract class TypeMethodInvoc
 extends TypeApp {
  def args:  TypecheckArgs
  def targs: TypecheckTArgs
  def meth:  Typecheck
}
```

```
abstract class TypecheckArgs extends Goal {
  type U <: EV.TCheckArgs

  def args:  List[Typecheck]
}

abstract class TypecheckTArgs extends Goal {
  type U <: EV.TCheckTArgs

  def targs: List[Typecheck]
}
```

Figure 5.18: A fragment of the high-level representation modeling the Java's type checker decision process that assigns types to function applications.

classes, `TypeApp`, and the inherited abstract type member `U`, representing the low-level instrumentation block from which they can be mapped from.
Internally, the implementation of the Java type checker would enclose the type checking of the arguments and type arguments with the distinct low-level instrumentation blocks, using the `EV.TCheckArgs` (for type checking of the value arguments) and `EV.TCheckTArgs` (for type checking of the type arguments) events. The logical grouping is necessary to define a non-ambiguous mapping of the typing decisions to their high-level counterparts, the `TypecheckArgs` and `TypecheckTArgs` classes, respectively.

The high-level representation provides an unambiguous one-to-many mapping from the low-level entry point of the `visitApply` method (represented by the `EV.TypeApp` event) to either the `TypeConstrApp` or the `TypeMethodInvoc` class. The mapping models the type checking of different method invocations that is again only inferred from the types of high-level goals.

The example of the Java compiler illustrates that the approach of lightweight instrumentation that is mapped to a separate, implementation-specific high-level representation is not tied to a particular language, and can be used to model the decisions of different type checkers.

5.5.4 Maintenance of the instrumentation

Manual instrumentation of the implementation of the type checker couples it tightly to the particular version of the compiler. In consequence, with every minor or major compiler release the instrumentation instructions have to be merged with compiler changes, a non-trivial task requiring a good understanding of the implementation. Importantly, any significant changes in the compiler may undermine the high-level representation and its ability to accurately reflect the type checker's decision process.

In practice, the modifications to the existing instrumentation are likely to be performed in an incremental manner, rather than incorporating the complete instrumentation anew with every release. In the case of the Scala compiler development, none of its minor releases in the 2.10.x and 2.11.x branches had modified the main type checking process even when undergoing code refactoring; for changes within the same major version, the existing set of low-level, non-intrusive, events and high-level classes was sufficient.

In the case of the major releases of the compiler, there is a higher probability that the type checker will alter, thus affecting the instrumentation blocks and/or high-level representation exposed by our instrumentation framework. We first identify three possible scenarios of a change in the type checker that breaks the high-level representation between the releases, and then propose potential solutions to the problem:

- A compiler change introduces a new type checker execution path, which does not affect any of the previous executions. For example, a new fallback mechanism handles previously rejected AST nodes.

  The additional low-level instrumentation, needed to expose the new behavior, will have to map to a new high-level class representation in a one-to-many mapping. Assuming the non-ambiguity of the mapping with respect to the previously defined instrumentation block, the previously defined high-level class hierarchy will continue to represent the old type checker decisions process.

- A compiler change modifies an existing type checker decision process.

  Depending on the scale of the change, either a new high-level class has to be added to the high-level representation, or both a combination of low-level events and the modified high-level representation need to be added. In both cases, we can take advantage of the one-to-many mapping to define a set of distinct, non-ambiguous classes sharing a common super type to model the old and the new type checker execution with a single class hierarchy. By pattern matching on the sealed super type, we can identify a particular type checker execution and provide an appropriate analysis, within the same codebase.

- A compiler change modifies the existing type checker decision process, but the change still maps to a semantically different but already defined high-level representation.

  The mapping between the low-level data and high-level representation is driven only by the types of members of the high-level classes. If the change in the type checker execution results in a mapping context that still allows for the mapping to be successful the change will not be detected. It is therefore the role of the infrastructure developers to make such change explicit, by means of for example additional instrumentation.

Modeling the discrepancy of implicit search resolution: An Example

As an example of a type checker feature added in the 2.11.x release that modified the 2.10.x Scala's type checker, we consider an improvement of the encoding of functional dependencies (Hallgren [2000]) with the implicit views[5]. To illustrate the change we discuss a simple program where the functional dependency for some term is inferred implicitly and explicitly:

```
 1  class StringOps { def foo: Int = // ... }
 2  abstract class FunDep[A, B] { def u(t: A): B }
 3
 4  {
 5   implicit def FundepString: FunDep[String, StringOps] = // ...
 6   implicit def funDep[T, U](x: T)(implicit z: FunDep[T, U]): U = z.u(x)
 7
 8   val a1: Int = "x".foo                        // Type checks only in Scala 2.11.x
 9   val a2: Int = funDep("x")(FundepString).foo // Type checks in Scala 2.10.x and 2.11.x
10  }
```

The listing defines a `StringOps` class with a single member `foo` of type `Int`, and an abstract class `FunDep` representing the functional dependency with its two type parameters, such that its only method `u` takes a value having a type of the first type parameter and transforms it into a value having the type of the second type parameter. Later we define the two local implicit functions, `FundepString` and `funDep`. The `FundepString` function returns a concrete instance of the `FunDep` class. The `funDep` function encodes a generic implicit functional dependency between its two local type parameters, `T` and `U`, using the implicit parameter `z` of type `FunDep[T,U]`. In other words, the functional dependency between the instantiations of type parameters `T` and `U` will be satisfied if and only if in an application involving the `funDep` function the type checker can materialize the implicit argument of a required type.

The definition of the `a2` value represents an explicit application of the functional dependencies encoding, that is equivalent to the definition of the `a1` value, modulo the reliance on the implicit resolution mechanism.
The body of the definition of the `a1` value is a member selection on a value `"x"` of type `String` where `foo` is not among the defined or inherited members of type `String`. Before reporting an error in line 8, the Scala type checker will always try to adapt the qualifier to a type that has the member `foo`. The pattern is commonly used in the domain-specific libraries and averts the boilerplate code such as the one listed in line 9. Before we delve into the details of representing the type checking of the above function applications, we first give an informal explanation of the process.

[5]The details of the improvement are present in the bug report that is available at https://issues.scala-lang.org/browse/SI-3346.

```
sealed abstract class VerifyImplicit  extends Goal {
  type U <: EV.VerifyImplicit
  def typeImplicit:    TypeGoal
}
abstract class VerifyNonViewImplicit  extends VerifyImplicit {
  def adaptImplicit:   AdaptGoal
  def tpeConformsToPt: SubtypingGoal
  //...
}
abstract class VerifyViewImplicit     extends VerifyImplicit {
  def tpeConformsToPt: SubtypingGoal
  // ...
}

abstract class ImplicitArgForParam    extends Goal { ... }
```

Figure 5.19: A fragment of the high-level representation corresponding to verifying individual implicit arguments during the implicit resolution. The "..." represents the decisions irrelevant for the purpose of the example.

```
abstract class InferImplicitArgs           extends Goal {
  type U <: EV.ImplicitArgs
  def implicitArgs:    List[ImplicitArgForParam]
}

abstract class VerifyViewInferArgsImplicit extends VerifyImplicit {
  def inferArgs:       InferImplicitArgs
  def adaptImplicit:   AdaptGoal
  def tpeConformsToPt: SubtypingGoal
  // ...
}
```

Figure 5.20: A high-level class representing the verification of the individual implicit arguments during the implicit resolution, for the modified 2.11.x type checker.

```
def analyzeImplicitArg(impl: VerifyImplicit) = {
  impl match {
    case VerifyNonViewImplicit(typeGoal, _, tpeConformance, ...)             =>
      // ...
    case VerifyViewImplicit(typeGoal, tpeCOnformance)                        =>
      // ...
    // warning: the match may be non-exhaustive
    //case VerifyViewInferArgsImplicit(typeGOal, inferImplicitArgs, tpeConformance) =>
  }
}
```

Figure 5.21: A fragment of a logic that analyzes how the Scala type checker accepts/rejects implicit arguments using the exposed high-level representation. The pattern matcher will statically verify the sealed `VerifyImplicit` class and warn about the omitted *new* subclass.

The implicit resolution of Scala 2.10.x will try to adapt the `"x"` qualifier by checking the implicit values available in the scope of the member selection. The verification of the implicit value `funDep("x")` will assign the internal method type `(implicit z: FunDep[String, ?U]): ?U`, where the local type parameter `T` has been instantiated to the type of the qualifier (`"x"`), the instantiation of the type parameter `U` is still unresolved (the `?U` notation), and the type checker still has to find an implicit argument for the parameter `z`. The search for the implicit argument is preceded by an instantiation of all the unresolved type parameters. In particular, the unresolved type parameter `U` will be instantiated to the *maximal* possible type `Nothing` due to lack of type constraints. Consequently, the implicit resolution fails to materialize an argument of type `FunDep[String,Nothing]` and the `"x".foo` member selection is rejected.

In Scala 2.11.x, the implicit resolution modifies the verification of the implicit views so that the type checker tries harder to materialize the *witness* for an implicit parameter, if it exist. In the modified version, the implicit resolution is performed directly after typing the implicit value. Consequently, the implicit resolution for the parameter `z` is triggered with a partially determined expected type (`FunDep[String,?]`) (where the wildcard type stands for the *don't care* type) which allows it to find a unique implicit argument `FundepString` in the local scope. The change is subtle but significantly improved the expressive capabilities of the implicit resolution (Burmako [2013b]). In order to provide a comprehensive type debugging experience for different compiler releases both behaviors need to be modeled in the high-level representation.

Having explained the differences between the two implicit resolution implementations, we now turn our attention to a high-level representation that can model the differences.

Figure 5.19 defines a base class `VerifyImplicit` representing the mandatory decisions that verify an implicit view or an implicit value, *i.e.,* the implicit resolution mechanism will always assign a type to the selected implicit value (the `typeImplicit` member). The two immediate subclasses, `VerifyNonViewImplicit` and `VerifyViewImplicit`, correspond directly to the type checker's decisions necessary to verify the implicit value and the implicit view, respectively. In both cases, the type of the implicit is compared to the type expected by the implicit resolution context (the `tpeConformsToPt` member) but only the former will perform the additional type adaptation operation (the `adaptImplicit` member, its purpose is irrelevant for our example). Importantly, both classes lack any dependency on the type checking operation that opportunistically infers the implicit arguments (represented by the high-level goal of type `ImplicitArgForParam`), and would fail to model the implicit resolution mechanism that is present in Scala 2.11.

The modification introduced in the Scala 2.11 release is exposed by enclosing the change using the instrumentation block of the low-level `EV.ImplicitArgs` event. In Figure 5.20 we provide a high-level representation of the instrumentation block, the `InferImplicitArgs` class, where the `implicitArgs` member and its type define a dependency on the inference of the implicit arguments, a change that we have informally alluded to previously. Figure 5.20 extends the class hierarchy of the subclasses of the `VerifyImplicit` base class with a `VerifyViewInferArgsImplicit` class. The additional class models the modified type checker's behavior using the `inferArgs` member, without affecting the previous definitions.

Modeling the type checking decision process through a sealed class hierarchy means that it can also help with identifying the incomplete algorithms (Figure 5.21) that have to be adapted to the changes in the high-level representation.

5.6 Conclusions

We have presented the instrumentation technique for exposing the low-level type checking decisions of existing compilers. To model the high-level decision process that is more suitable for defining statically checked algorithms we have also proposed a surprisingly simple high-level class hierarchy. The gap between the two representations is eliminated through an automatic mapping function.

The presented mapping strategy introduces a small number of restrictions regarding the high-level representation; the restrictions could potentially be relaxed at a cost of runtime errors with weaker guarantees. Inadvertently, weaker guarantees lead to ambiguities that are unlikely to be possible to be fixed with basic instrumentation blocks and refactorings, and thus are harder to reason about.

Chapter 6

Type Debugger - the implementation details

Type derivation trees, their dual high- and low-level representation, and the algorithm for analyzing the decisions of local type inference form the foundations of the type debugger tool. In this chapter we outline the key components of the implementation of the infrastructure that define the tool and its capabilities. To illustrate the individual components we first give a very brief overview of the individual components that will be discussed in the chapter.

6.1 Overview

The infrastructure of the type debugging tool is divided into three main parts (illustrated in Figure 6.1 for reference):

- A compiler infrastructure that allows for the control over the execution of the type checking process and its instrumentation.
- The high-level representation of the elements of the type checking process, the algorithm that defines the core *TypeFocus*-based analysis, and the definitions of the *specialized functions* that analyze its results (Typing Slices).
- An error feedback layer. The part includes algorithms that define the generic improved error feedback, library-specific feedback defined in the plugins (Section 7.2.2), and the interactive debugger infrastructure (Section 7.3). All of the possible feedback methods define programmatically their operation using the high-level components (the high-level representation, the Typing Slices, the *TypeFocus*) and the previously exposed algorithms.

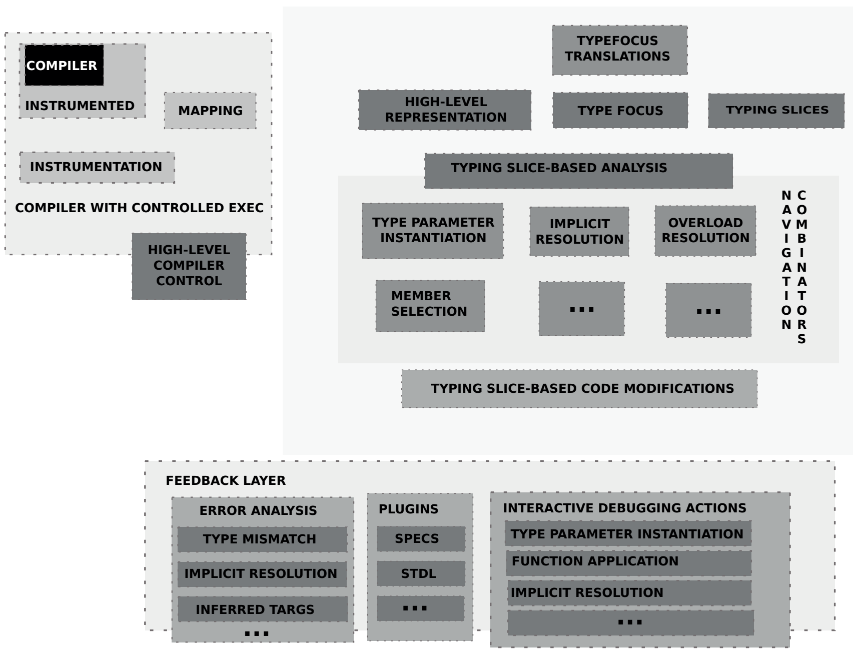


Figure 6.1: An overview of the type debugger infrastructure.

In Section 6.2 we describe the details of the integration of the instrumentation infrastructure with the existing compiler. The extended compiler has to define a clear interface to trigger the type checking process as well as to control the low-level frequently executed operations, with and without the instrumentation enabled. Any operations requiring the compiler execution do not directly deal with the low-level compiler but instead with the *high-level compiler control* facade component (visible in Figure 6.1). Consequently the component returns only the high-level goals representing the requested compiler operation.

The implementation of the high-level representation (Section 5.2), the *TypeFocus* abstraction (Section 6.3.1) and the Typing Slice abstraction (Section 6.4) define the three main components that will be used in navigating the decisions of the reconstructed type checking decision process. The infrastructure defines an API for inferring the *TypeFocus* instances directly from high-level goals, rather than dealing with the low-level type checking operations (the *TypeFocus Translations* component is described in Section 6.3.2). The core of the analysis is based on the repeated exploration of the Typing Slices inferred using the *TypeFocus*-based algorithms (Sections 6.4.2 and 6.4.3).

The Typing Slices returned by the *TypeFocus*-based algorithms provide information sufficient to continue their analysis, if necessary. Under certain circumstances, such as for generating the library-specific error feedback or interactive debugging, we might want to change the direction of the analysis in order to explore different type checking decisions. Such custom analysis is allowed in the type debugging framework by exposing the API of the specialized analysis functions for different type checking decisions. The specialized analysis functions, examples of which we present in Section 6.5 and Section 6.6, allow for the analysis of the type checking decisions at a more coarser level than the analysis of the dependencies of the high-level goals, and finer level than simple Typing Slice-to-Typing Slice exploration. This way the programmers who define custom error handlers, or any other heuristics, can define the Typing Slice-specific analysis without having to understand many of the implementation details of the compiler.

The type debugger also defines a code modification component (Section 6.7), which is independent from the improved error generation infrastructure. The proposed mechanism can not only infer corrections for the limitations of the underlying type system and the type inference, but also do it with surgical precision. The modifications rely on the custom Typing Slice-to-Typing Slice exploration, which reuses most of the existing infrastructure.

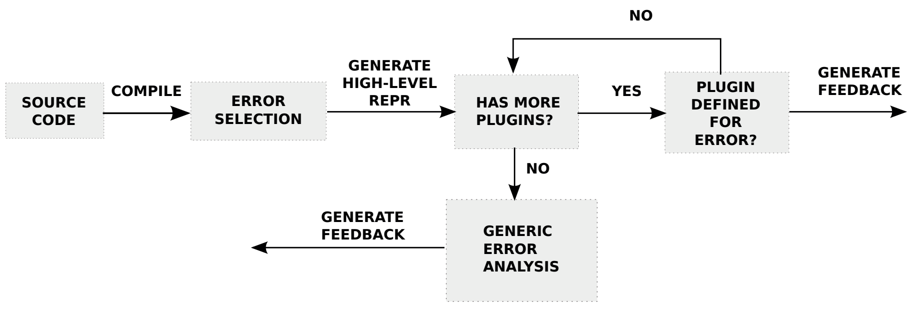


Figure 6.2: A simplified execution of the type debugger tool that generates improved error feedback.

We use Figure to illustrate how the type debugger tool generates improved error feedback in terms of a more elaborate and precise error message.

For any given program, we first trigger a regular, non-instrumented compilation which will bring a list of errors exactly the same as for the normal compiler execution. Later we trigger a targeted compilation, based on the error selected by the user (or the first of the reported errors), which in the process infers a high-level representation of the type checking of the selected region. The type debugger will later check if any of the loaded type debugger plugins (Section 7.2.2) is defined for the specific error message, and consequently generate an improved error message. If no plugins were applicable, we delegate to the generic techniques defined for the different kinds of errors. The generic and the plugin handlers of the errors use the Typing Slice exploration to quickly step through the *irrelevant* elements of the type derivation tree, and identify those that can serve as a starting point for a more in-depth analysis. For example, depending upon the type checking scenario, function applications or variable assignments would serve as a good starting point for generating the improved error feedback (one can notice a correlation with the definition of the *Propagation Root* from Section 3.1.2).

We defer the discussion of the interactive mode of the type debugger until Section 7.3.

6.2 Compiler infrastructure

In this section we present a minimal set of capabilities that a compiler implementation has to provide in order to allow for a controlled execution of the instrumentation during the type checking of Scala programs. Figure 6.3 defines an extension of Scala's main compiler class (named `Global`) that we use to illustrate the necessary functionality. The `DebuggerGlobal` class defines three methods crucial for the generation and collection of the low-level data:

- The '`withInstrumentation`' and '`withDetailedInstrumentation`' methods take a generic type checker operation, provided as a by-name argument, and execute it with a regular or detailed instrumentation enabled, respectively. The methods themselves only control the emission of the low-level events, but refrain from performing any type derivation tree reconstruction.
- The '`withEventListener`' method takes an instance of the `EventListener` listener class, and registers it for the period of the execution of the type checking operation provided as a by-name argument '`exec`'. The registered instance of the `EventListener` class collects the low-level instrumentation instances through callbacks and constructs the type derivation trees in the '`event`' method. A reactive implementation (using for example the approach of Maier and Odersky [2012]) presents a possible alternative to collecting the low-level events but for our use-case is too excessive.

In addition to capturing the low-level instrumentation, the compiler has to expose methods that can trigger general type checking operations as well as the local ones; the definition of the `DebuggerCompilerControl` trait records a number of Scala-related operations required by the Type Debugger:

- The '`tcheck`' method triggers a type checking of the source files specified in the argument list in a non-debugging mode. Unlike the traditional pipeline of Scala's compiler, which goes through a number of independent phases until code generation, we stop at the front-end of the compiler, *i.e.,* the compiler executes scanning, parsing, the name resolution phase and the type checking of the provided programs. Further compilation phases, their decisions or errors are not exposed by our debugger.
- The '`tcheckTargeted`' method triggers a type checking of a fragment of the previously parsed source file. The method assumes that the program structure information from the complete parsing of the source files is still available. The requested range position, represented through the '`pos`' parameter, holds information about the source tree and

```
abstract class DebuggerGlobal extends Global with DebuggerCompilerControl {

  val EV: EventModel  // a reference to the instrumentation universe

  def withInstrumentation[T](exec: => T): T = // ...
  def withDetailedInstrumentation[T](exec: => T): T = // ...
  def withEventListener[T](listener: EventListener)(exec: => T): T = // ...


  abstract class EventListener {
    def event(ev: EV.Event): Unit
  }
}

trait DebuggerCompilerControl {
  self: Global =>

  def tcheck(srcs: List[SourceFile]): Unit

  def tcheckTargeted(pos: Position): Unit

  def runSubtyping(tpe1: Type, tpe2: Type): Unit

  def runTpeEq(tpe1: Type, tpe2: Type): Unit

  def runImplicitSearch(pos: Position, tree: Tree, pt: Type, isView: Boolean): Unit

  def runOverloadingResolution(tpe1: Type, tpe2: Type): Unit

  def typeTAnnotation(tree: Tree, where: Position): Option[Tree]

  def typeDef(ddef: DefDef, where: Position): Option[Tree]
}
```

Figure 6.3: A brief look at the `DebuggerGlobal` class that enriches the main compiler class with the required instrumentation capabilities.

the program fragment to debug. The range position is sufficient to identify a minimal enclosing AST fragment, and a path to it, thus realizing the selective type checking.

- The '`runSubtyping`' method triggers a local subtype checking between the two low-level type values.
- The '`runTpeEq`' method triggers a local type equality check between the two provided types.
- The '`runOverloadingRes`' method triggers a local overloading resolution check between the two provided types, *i.e.,* it determines if the first type is more specific than the second one.
- The '`runImplicits`' method triggers the implicit resolution for the argument repre-

```
trait DebuggerCompilerOps {
  self: HighLevelRepr =>

  val global: DebuggerGlobal

  def compilerOps: CompilerOps

  abstract class CompilerOps {
    import global.{Type, Position, Tree, Symbol}

    def tcheck(pos: Position):

    def subtyping(goal: Conformance):      Option[Conformance]
    def subtyping(tpe1: Type, tpe2: Type): Option[Conformance]

    def tpeEquality(goal: TypeEq):           Option[TypeEq]
    def tpeEquality(tpe1: Type, tpe2: Type): Option[TypeEq]

    def implicits(goal: ImplicitSearch):       Option[ImplicitSearch]
    def implicits(pos: Position, tree: Tree,
                  pt: Type, isView: Boolean):  Option[ImplicitSearch]

    def overloadRes(goal: OverloadResolution): Option[OverloadResolution]
    def overloadRes(tpe1: Type, tpe2: Type):   Option[OverloadResolution]

    def locateIdent(sym: Symbol):          Option[NamerGoal]
    def locateDef(sym: Symbol):            Option[NamerGoal]
    def locateTMember(sym: Symbol):        Option[NamerGoal]
  }

}
```

Figure 6.4: An overview of the compiler interface available to the users of the Type Debugger. The interface defines methods that translate the low-level operations to their high-level interpretations.

sented by the parameter '`tree`'. Here the implicit search term refers to the type-directed mechanism for inferring the implicit arguments or converting the expressions to an expected type. If the argument for the '`isView`' parameter is `true`, the method realizes the search for an implicit conversion with the expected type '`pt`'. Otherwise the search attempts to find an implicit argument which type conforms to the expected type '`pt`'. The additional position information is sufficient to locate the smallest enclosing type checking context that lists the available implicit values without triggering a complete type checking process.

The methods that control the execution of the compiler are of type `Unit`, meaning that they are not expected to return a value and the instrumentation itself is collected through side-effects in the registered `EventListener` classes.

In Figure 6.4 we provide another layer of abstraction, defined on top of the previously described compiler interface with methods returning the high-level goals instead. The `DebuggerCompilerOps` interface has a private dependency on the definition of the high-level hierarchy (as expressed through the `HighLevelRepr` self-type) and is tied to an instance of the main debugger compilation class (as expressed through the '`global`' member in line 4). The '`compilerOps`' member returns an instance of the nested `CompilerOps` class. The class itself serves as bridge between the operations on the low-level data and their high-level interpretations.

The methods of the `CompilerOps` class return the high-level representations of the local low-level operations. The overloaded methods retrieve instances of the `Goal` class representing the high-level decision process of subtyping (`Conformance`), type equality (`TypeEq`), implicit resolution (`ImplicitSearch`) and overload resolution (`OverloadResolution`). The overloaded methods mean that local type derivation trees can be retrieved through either a transparent expansion of the existing stub `Goals` that lack any detailed type checking info, or by a provision of the low-level values directly, such as types or ASTs. The ability to trigger the low-level type checking operations and retrieve their high-level representation is also crucial if we want to allow the users of the debugger to diverge from a fixed, error-specific analysis of the programs. We elaborate on the potential use-cases of the user-directed and the interactive approach in Chapter 7.

The remaining '`locateIdent`', '`locateDef`' and '`locateTMember`' members return the high-level type derivation trees of identifiers, value or type members, respectively. Their type has been determined during the delayed initialization and thus is not part of the main type derivation tree. For situations where the origin of the type of the definition cannot be determined, *e.g.,* values, which type has been reconstructed from the type signature of the bytecode or for the internal constant values, the methods will return an empty option value.

6.3 The *TypeFocus* generation

The *TypeFocus* abstraction stands at a core of any type debugging technique presented in our work. In Section 6.3.1 we present its representation in Scala and provide a representative list of *TypeFocus* subclasses for the existing Scala types. Later we present a selection of the operations that infer *TypeFocus* instances from high-level goals (Section 6.3.2).

6.3.1 The *TypeFocus* for Scala

In Figure 6.5 we provide Scala's interpretation of the *TypeFocus* abstraction and its required operations. As in the case of the other exposed interfaces, the *TypeFocus* instances operate on the existing Scala's types and its definition has to have a reference to the compiler's class

```
1  trait TypesFocus {
2
3    val global: DebuggerGlobal
4    import global.{Type, Symbol}
5
6    abstract class TypeFocus {
7      def apply(tp: Type): Either[Type, (Type, TypeFocus)]
8      def focus(tp: Type): Type = apply(tp).fold(id, _._1)
9
10     def compose(tfocus: TypeFocus):   TypeFocus
11     def head:                         TypeFocus
12     def tail:                         TypeFocus
13
14     def update(tpe: Type, mod: Type): Option[Type]
15   }
16
17   // ... concrete instances, defined later
18 }
```

Figure 6.5: The *TypeFocus* interface.

(line 3 of the definition).

The main operation of the *TypeFocus* abstraction, the extraction of type elements, is realized by the ‘`apply`’ method. The method returns a disjoint union as indicated by the `Either` type constructor and reflects the two possible results of the type extraction: either a complete type selection (of type `Type`), or a partial one (of a product type `(Type, TypeFocus)`). The auxiliary ‘`focus`’ method returns a type selection on the provided type, essentially ignoring the value of the *partial TypeFocus*, included in the right projection of the union type.

The ‘`compose`’,‘ `head`’ and ‘`tail`’ methods allow for the construction and deconstruction of the *TypeFocus* instances, with the semantics almost identical to their theoretical counterparts (Section 3.3.1). The only difference is in the order of the application of the composition (for details of the differences we refer the reader to Plociniczak et al. [2014]); for historical reasons, and similarity to function composition, the ‘`tfocus1` compose `tfocus2`’ composition, for any *TypeFocus* instances `tfocus1` and `tfocus2`, means that we first apply the `tfocus2` extraction to the provided type and only later `tfocus1` to the result of the former. In other words, using the notation of Section 3.3 ‘`tfocus1` compose `tfocus2`’ is equivalent to ‘`tfocus2` ::: `tfcosu1`’.

For example, to define a deep type extraction of some type `X` in the function type $(S \to X) \to T$, we would use a `TypeFocus` that extracts the result type of the function, say `tfocusRes`, a `TypeFocus` that extracts the type of the first parameter, say `tfocusParam`, and apply their composition to some low-level type, `tp`, such as

```
(tfocusRes compose tfocusParam)(tp) match {
  case Left(tp0)               => // ...
  case Right((tp0, tfocusCont)) => // ...
}
```

The '`update`' method returns a type identical to the provided low-level type value, modulo the extracted type element which is replaced with the value of the '`mod`' parameter. For example, given Scala's interpretation of the $List[Nil]$ and Int types, and some `TypeFocus` value, '`tfocusVal`', which extracts the first type argument in the type application involving the List type constructor, we can express the modification to types as (the grayed-out types refer to their Scala interpretations):

```
tfocusVal.update(List[Nil], Int).get == List[Int]
```

The localized type modifications were not part of the core of the type debugging analysis but they offer the ability to perform surgical-level type modifications. The type modifications are particularly useful when we combine them with the type mismatch information to define heuristics that modify code (Section 6.7.2). With the '`update`' operation the `TypeFocus` class represents a variant of the Lens family (Foster et al. [2008]).

Having defined the `TypeFocus` abstraction, we are now in position to explain some of its instances, and operations that infer them.

Examples of `TypeFocus` for Scala's types

The implementation of the `TypeFocus` concept has to provide type extractors for each of the internal types, and their components. In Figure 6.6 we provide a representative selection of the `TypeFocus` class hierarchy. Apart from the identity *TypeFocus*, represented by the singleton class of `IdTFocus` and equivalent to an empty type selection in Section 3.3, the other classes correspond directly to the Scala's internal types, as defined in the specification of the language in Odersky [2015]. To illustrate their role and semantics, we will now discuss them in turn.

The `TypeArgTFocus` class:

The `TypeArgTFocus` class applies to first-order types that are constructed from an application of type constructors, such as `List` or `Set`, to other first-order types. The `TypeArgTFocus` extracts a single type argument based on its position (zero-indexed) in the type arguments

```
...
object IdTFocus extends TypeFocus { ... }

abstract class TypeArgTFocus extends TypeFocus {
  def baseSym: Symbol
  def argIdx:  Int
}
object TypeArgTFocus {
  def apply(baseSym: Symbol, argIdx: Int): TypeFocus = // ...
  def unapply(x: TypeFocus): Option[(Symbol, Int)] =   // ...
}

abstract class MethodParamTypeFocus extends TypeFocus {
  def param: Int
}
object MethodParamTFocus {
  def apply(idx: Int): TypeFocus =              // ...
  def unapply(tfocus: TypeFocus): Option[Int] = // ...
}

object MethodResTFocus extends TypeFocus { ... }

abstract class OverloadTFocus extends TypeFocus {
  def alt: Symbol
}
object OverloadTFocus {
  def apply(alt: Symbol): TypeFocus =              // ...
  def unapply(tfocus: TypeFocus): Option[Symbol] = // ...
}

abstract class MethodResTypeFocus extends TypeFocus
object MethodResTypeFocus {
  def apply(): TypeFocus = // ...
}

abstract class TypeMemberTFocus extends TypeFocus {
  def owner:  Symbol
  def memSym: Symbol
}
object TypeMemberTFocus {
  def apply(owner: Symbol, memSym: Symbol): TypeFocus = // ...
  def unapply(x: TypeFocus): Option[(Symbol, Symbol)] = // ...
}
...
```

Figure 6.6: A fragment of the *TypeFocus* hierarchy for extracting elements of Scala's types

list. The `TypeArgTFocus` class abstracts over different kinds of type constructors by holding a reference to the type designator of a class or a trait it was constructed from in the '`baseSym`' member. A type descriptor reference does not limit its application only to the type constructor which it was created from. Rather the application of the `TypeArgTFocus` instance to a type first retrieves the least type instance of a '`baseSym`' class that is a super type of the provided type. Later `TypeArgTFocus` will attempt to extract the appropriate type argument of the resulting type, based on its position. Consequently, the type selection takes into account the semantics of nominal subtyping.

For illustration purposes we consider a simple class hierarchy of

```
class A[S, T]
class B[Z] extends A[Int, Z]
```

where we define an `A` class with two type parameters, and a generic class `B` that extends `A` while instantiating type parameter `S` to a value type `Int`. Given a `TypeFocus` value '`TypeArgTFocus('A', 0)`' that refers to the type descriptor of class `A` and a type argument at position `0`, it can be applied to different Scala types and yield the expected type arguments (the grayed-out types refer to their low-level Scala interpretations):

- `TypeArgTFocus('A', 0)(`$B[String]$`) = Left(`Int`)`
- `TypeArgTFocus('A', 0)(`$A[String, Int]$`) = Left(`$String$`)`

`MethodParamTFocus` and `MethodResTFocus`:

The instances of the `MethodParamTFocus` class and the `MethodResTFocus` class correspond to the extraction of the parameter's type and the return type from the method type, respectively. A method type is a non-value type internally used to represent types of methods, and denoted as $(Ps)U$, where (Ps) represents a sequence of named parameters and their types and U is a regular value type or another method type representing the return type of the method. Method types do not exist as types of values and are implicitly converted to function types through the eta-expansion. The described `TypeFocus` classes also have to mimic the eta-expansion by extracting the corresponding elements of the function types; the implicit translation guarantees that `TypeFocus` instances applied to types after and before the eta-expansion operation transparently extract the identical type elements.

For example,

- For a method type:
  `MethodParamTFocus(0)(`$(x : Int)(y : Float)String$`) = Left(`$Int$`)`
- For an equivalent, eta-expanded method type:
  `MethodParamTFocus(0)(`$Int => (Float => String)$`) = Left(`Int`)`

TypeMemberTFocus:

The `TypeMemberTFocus` class represents a type selection extracting the value of the type member of the type. Similarly to the `TypeArgTFocus`, a type selection first retrieves the least type instance of the '`owner`' class that is a super type of the provided type, and then attempts to find the type of the publicly accessible member having the same name as its '`memSym`' value.

OverloadTFocus:

The Scala implementation assigns a so called *overloaded type*, denoted as `T-(`S_0 `<and> ... <and>` S_n`)` to member selections involving multiple alternatives, where T refers to some prefix type, and S_i refers to an i-th definition of the alternative. The overloaded type is an example of an implementation-specific type, non-existent even in the Scala specification. The type still needs to be represented in the `TypeFocus` hierarchy through the `OverloadTFocus` subclass because its type elements affect the type checking process and can guide our *TypeFocus*-based analysis.

To illustrate the need for type extractors that operate on the internal types, we briefly consider an example involving an overloaded method '`apply`' inspired by code snippets from a Scala library used for generic programming:

```scala
abstract class TConst {
  type Elem
  def bar(): Elem
}
class A[S] {
  def apply[T](x: T):       TConst { type Elem = S } = // ...
  def apply[T](x: T, y: T): TConst { type Elem = S } = // ...
}
val x:A[Int] = // ...
x.apply(1, 1).bar()
```

In the example, any analysis of the inferred type `Int` of the '`x.apply(1,1).bar()`' function application can be initially oblivious of the overloaded method `apply` and will be equivalent to the `TypeFocus` value of `TypeMemberTFocus('TConst', 'Elem')`. As we progress with the analysis of the source of the inferred type, the analysis steps through the function application ('`x.apply(1,1)`') and member selection ('`x.apply`') terms and the `TypeFocus` value has to fully encapsulate the typing decision process, including the overload resolution selection, in order to guide the analysis of the type checking process for the qualifier term.

```
1  abstract class TConst {
2    type Elem
3    def bar(): Elem
4  }
5
6  abstract class A[S](x: S) {
7    def one[T](x: T): TConst { type Elem = S } = // ...
8  }
9  abstract class B extends A[Int](0) {
10   type Rep
11   def two[T](x: T):         TConst { type Elem = Rep } = // ...
12   def three[T](x: T):       TConst { type Elem = Int } = // ...
13   def three[T](x: T, y: T): TConst { type Elem = Int } = // ...
14 }
15 class C extends B {
16   type Rep = Int
17 }
18
19 val x: C = // ...
20 val y1: String = x.one(1).bar()    // error
21                 //        ^
22 val y2: String = x.two(1).bar()    // error
23                 //        ^
24 val y3: String = x.three(1).bar()  // error
25                 //          ^
26                 // error: type mismatch;
27                 // found:    Int
28                 // required: String
```

Figure 6.7: An example of similarly looking, invalid value assignments. All value assignments lead to the identical typer error message generated by Scala compiler, modulo the reported error location, as indicated at the bottom of the listing.

The `TypeFocus`-driven analysis

The elements of the `TypeFocus` hierarchy are accompanied with companion objects providing '`apply`' and '`unapply`' methods, for creation of type selectors and pattern matching on the `TypeFocus` instances (Emir et al. [2007]). With such definitions in mind, implementation of the navigation rules that drive the analysis of the high-level goals comes down to pattern matching on the head of `TypeFocus` values. To illustrate, we consider the analysis of a set of invalid assignments in Figure 6.7; their conflicting `Int` types originate from different program locations and involve non-trivial combinations of type system features.

Figure 6.7 defines an abstract class `TConst`, with a single abstract type member `Elem` and a '`bar`' method. Later we define a single inheritance class hierarchy consisting of the `A`, `B`, and `C` classes, each of which defines methods that return a subtype of the `TConst` type. The methods of the classes differ in how the type member `Elem` is instantiated in the refined types: in the constructor (method '`one`' in lines 6 and 9), through a separate abstract type member

```
1 ...
2 def classContext(tfocus: TypeFocus, goal: TypeClass) =
3   tfocus.head match {
4     case TypeArgTFocus(owner, _) =>        // ...
5     case TypeMemberTFocus(owner, member) => // ...
6     case OverloadTFocus(owner, alt)      => // ...
7     case ...                             => // ...
8   }
9 ...
```

Figure 6.8: A fragment of the implementation of the *TypeFocus*-driven analysis. The `classContext` method analyzes the typing decisions that verify the definitions of the classes (represented by the high-level goal `TypeClass`).

`Rep` (method '`two`' in lines 10 and 16), or an explicit type annotation in the type refinement (method '`three`' in lines 12, and 13). The function applications, involving the above methods, all lead to the same type mismatch error (lines 20, 22, 24).

The regular `TypeFocus`-based analysis will identify the non-final source of the type mismatch in the function applications, and later navigate to their respective member selections '`x.one`', '`x.two`', '`x.three`', and the qualifier '`x`'. The type assigned to the '`x.one(1)`', '`x.two(1)`' and '`x.three(1)`' applications is the same, *i.e.,* `TConst { type Elem = Int }`, and is accompanied by the same type selection, `TypeMemberTFocus('TConst', 'Elem')`. However the *TypeFocus*-based analysis, being aware of the high-level goals, ends up with different `TypeFocus` instances when analyzing the type of the qualifier in the discussed assignments:

- For '`x`' in '`val y1: String`' with `TypeArgTFocus('A', 0)`
- For '`x`' in '`val y2: String`' with `TypeMemberTFocus('B', 'Rep')`
- For '`x`' in '`val y3: String`' with `OverloadTFocus('B', 'three'[0])`

Despite the *TypeFocus*-based analysis reaching the same qualifier 'x' as the source of the conflicting type, the reconstructed `TypeFocus` values encapsulate the differences in the instantiation and are sufficient to continue the analysis of the inferred type of the qualifier.

Knowing that the qualifier 'x' is of type `C` (line 19), the analysis will have to end up at the high-level goal representing the type checking of class `C`. Figure 6.8 provides an overview of the `TypeFocus`-driven analysis function which considers the high-level `TypeClass` goal representing the verification of the definitions of the classes. The `classContext` function uses the *head* of the included `TypeFocus` instance (line 3) to guide the analysis of the class. In particular, we can pattern match on the potential `TypeFocus` instances in order to define the generic steps of the analysis that also apply to the three invalid assignments from Figure 6.7:

- In the context of the `TypeClass` goal, the `TypeArgTFocus` value indicates that we seek to understand the instantiation of a non-local type parameter of the class or one of its parents. The latter information is sufficient to direct the navigation to one of the constructors of the underlying class and, eventually, display the improved error with the location at

```
abstract class B extends A[Int](0) {
                              ~~~
```

- In the context of the `TypeClass` goal, the `TypeMemberTFocus` value indicates that the instantiation of one of the type members explains the source of the target type. With such value of `TypeFocus` we can navigate to the type derivation node that verifies the declared type alias, and, eventually, display the improved error with the location at

```
abstract class C extends B {
  type Rep = Int
             ~~~
```

- In the context of the `TypeClass` goal, the `OverloadTFocus` value indicates the identity of the method when dealing with multiple alternatives. With such information we can navigate to the typing decisions that verified the overloaded method, and, eventually, display the improved error with the location at

```
def three[T](x: T): TConst { type Elem = Int } = // ...
                                         ~~~
```

The examples illustrate that the `TypeFocus` values, while simple in their definition, not only represent trivial extraction of the components of types but can also easily guide the analysis of non-trivial Scala programs, as formally alluded to in Section 3.5.3.

6.3.2 The inference of `TypeFocus` instances

Due to their low-level nature and the existence of stub goals, the translations from the low-level operations to the `TypeFocus` instances should not be performed manually by the programmers. Rather the Type Debugger has to provide inference operations that transparently deliver their interpretations and hide the internal details of the type checking.

```
trait TypeFocusOps {
  self: HighLevelRepr with TypesFocus =>

  def tfocusOps: TFocusOps

  abstract class TFocusOps {
    def toError(conf: Conformance):                              List[TypeFocus]
    def fromConstraint(conf: Conformance):                       Option[TypeFocus]
    def nonLocalTParam(memSel: TypeMemberSel, tfocus: TypeFocus): Option[TypeFocus]
  }
}
```

Figure 6.9: The interface for generating `TypeFocus` instances from the low-level type checking decisions.

To illustrate the `TypeFocus` translations required by the Type Debugger we consider a selection of representative methods from Figure 6.9. There, the `TypeFocusOps` trait defines a '`tfocusops`' abstract member of type `TFocusOps`, where the returned value offers the opportunities to translate high-level goals to their `TypeFocus` counterparts (the self-type of the trait determines the private dependence on the high-level representation and the `TypeFocus` representation).

The translation methods have to offer at least means to:

- Infer the `TypeFocus` values that represent the failed subtyping derivations (the '`toError`' method, Section 4.2).
- Translate the type constraint node, located in some subtyping derivation tree, into an equivalent `TypeFocus` instance (the '`fromConstraint`' method, Section 3.7.1).
- Infer the `TypeFocus` value that can bridge the gap between the non-local type parameters or type members, and their presence in the type signatures of the members (the '`nonLocalTParam`' method). The `nonLocalTParam` method is the only one that has not been mentioned in any way in the formal description of the *TypeFocus* concept. We will illustrate its importance by means of an example.

Example: The nonLocalTParam method

To illustrate the problem of non-local type parameters we consider a type mismatch error between the two option values:

```
1  val xs: List[Int] =  // ...
2  val y = xs.find(_ > 0)
3  val z: Option[String] = y
4  //                      ^
5  // error: type mismatch;
6  // found   : Option[Int]
7  // required: Option[String]
```

In the example, we assign first element of the list of integers ‘`xs`’ that is greater than zero to the ‘`y`’ value. The inferred type of the ‘`y`’ value involves the optional `Option` type, and eventually leads to a type mismatch error in line 3. The analysis of the error is problematic because the ‘`x.find(_ > 0)`’ application (of type `Option[Int]`), or more precisely its type argument `Int`, does not in general immediately relate to the qualifier ‘`xs`’, (of type `List[Int]`), which is needed to analyze the term using our *TypeFocus*-based algorithm.

Given a (simplified) definition of the ‘`find`’ method declared in the `List` class as:

```
class List[A] {
  ...
  def find(p: A => Boolean): Option[A]  = // ....
  ...
}
```

the problem becomes apparent from the return type of the method. The ‘`nonLocalTParam`’ method will take high-level goal of the member selection (the ‘`memSel`’ parameter) and implicitly compare the inferred type of the member selection ‘`xs.find`’ (of type `(x:  Int => Boolean)Option[Int]`) and the type of the declared method ‘`find`’ (of type `(x:  A => Boolean)Option[A]`). The comparison takes into account the type selection provided by the `tfocus` argument.

The result bridges the gap caused by non-local type parameters or type members, allowing us to continue the analysis of the qualifier with a correct type selection. In particular, given the type selection `TypeArgTFocus(‘Option’, 0)`, visually interpreted on the inferred type of the member selection as `Option[` Int `]`, the `nonLocalTParam` will translate it into a `TypeArgTFocus(‘List’, 0)` value, visually interpreted on the inferred type of the qualifier as the `List[` Int `]` type selection. The inferred `TypeFocus` allows us to analyze the high-level goal representing the type checking of the qualifier and preserves the well-formedness and precision of the initial type selection.

6.4 The *TypeFocus*-based analysis

The *TypeFocus*-based analysis of type derivation trees alleviates the need for understanding the non-trivial dependencies between the high-level goals. Such navigation is *strict*, or deterministic, meaning that the expansion of Typing Slices is entirely driven by a single initial `TypeFocus` value representing some target type. In this section we study the elements of the implementation that realize the *strict* navigation, only to later complement it with more fine-grained, Typing Slice-specific, techniques that allow the users of the debugger to deviate from it at any point.

Section 6.4.1 provides the definition of Typing Slice in our Type Debugger. Later we present functions that allow the users to infer Typing Slices from the type derivation tree (Section 6.4.2) and associate them with low-level information such as program locations (Section 6.4.3).

6.4.1 Typing Slices

The Typing Slice combines two crucial elements of the type debugging analysis - the high-level representation and the *TypeFocus*. The base class of the Typing Slice, and its four different kinds translate in a straightforward way into a Scala class hierarchy, nested within the `TypingSlices` trait:

```
trait TypingSlices {
 self: HighLevelRepr with TypesFocus =>

  sealed abstract class TypingSlice {
    type Repr <: Goal

    def repr:    Repr
    def tfocus:  TypeFocus
  }
  sealed abstract class TSigSlice extends TypingSlice { ... }   // type signature
  sealed abstract class PtSlice extends TypingSlice { ... }     // prototype
  sealed abstract class AdaptSlice extends TypingSlice { ... }  // adaptation
  sealed abstract class TVarSlice extends TypingSlice { ... }   // type parameter
}
```

The base `TypingSlice` class defines an abstract type member `Repr` with an upper bound pointing at the high-level goal. When refined in the subclasses, the type determines the type of the underlying high-level goal. The `TypingSlice` is realized by the '`repr`' method returning a reference to its underlying high-level goal, and by the '`tfocus`' method returning a well-formed `TypeFocus` value extracting a part of its underlying type.

With the definition of the `TypingSlice` in place, we are now in a position to show how one can infer its values from the decisions of the type derivation trees.

```
trait TypeFocusAnalysis {
  self: HighLevelRepr with TypesFocus with TypingSlices =>

  def analysis: AnalysisOps

  abstract class AnalysisOps {
    def typeSource(goal: TypeGoal, tfocus: TypeFocus):          Option[TypeSlice]
    def expectedTypeSource(adapt: AdaptGoal, tfocus: TypeFocus): Option[TypeSlice]
  }
}
```

Figure 6.10: An interface for a *TypeFocus*-guided analysis of type derivation trees.

6.4.2 Inference of Typing Slices

The interface of the *TypeFocus*-based core algorithm is defined by the `TypeFocusAnalysis` trait (Figure 6.10). As in the previous cases, the trait represents a dependency on the other elements of the debugger through the self-type, *i.e.,* it depends on the definition of the high-level representation (`HighLevelRepr`), the definition of *TypeFocus* values (`TypesFocus`) and Typing Slices (`TypingSlices`). More importantly, the trait defines an abstract method '`analysis`' which returns the implementation of the core *TypeFocus*-based algorithm in the form of an instance of the nested `AnalysisOps` class.

The `AnalysisOps` class provides two variants of the *TypeFocus*-based algorithm:

- The `typeSource` method takes a fragment of the type derivation tree representing the synthesis of the type for some AST (the '`goal`' parameter), and, if possible, explains the source of its underlying type. To control the search it takes an instance of the `TypeFocus` class in the parameter '`tfocus`'.

  Without going into details, the implementation of the method simply pattern matches on the subtypes of the sealed `TypeGoal` type and delegates to the corresponding term-specific analysis rule, such as the analysis of function applications or typing of the classes.

- The `expectedTypeSource` method returns the source of the expected type given a high-level goal representing the adaptation of the synthesized type (the '`adapt`' parameter). The method returns the source of the target prototype, that is extracted using the value of the '`tfocus`' parameter.

The methods define a *shallow* search for the source of the type or the prototype in any type derivation tree. The shallow search does not attempt to expand automatically the intermediate Typing Slices.

```
1  trait SlicesOps {
2    self: HighLevelRepr with TypesFocus with TypingSlices =>
3
4    val global: DebuggerGlobal
5    import global.{Tree, Position, Type}
6
7    def tslicesOps: TypingSlicesOps
8
9    abstract class TypingSlicesOps {
10
11     def extractPos(tslice: TypingSlice):  List[Position]
12     def extractType(tslice: TypingSlice): Option[Type]
13     def extractAST(tslice: TypingSlice):  Option[Tree]
14
15     def expand(tslice: TypingSlice):      List[TypingSlice]
16   }
17 }
```

Figure 6.11: The definition of the operations on the Typing Slices.

6.4.3 Exploration of type checking with Typing Slices

The `TypingSlice` offers convenient means for representing the results of the exploration of type derivation trees. At the same time, the abstraction does not immediately relate to the low-level types and ASTs that are necessary to provide improved error feedback. It is also not immediately obvious how to expand the non-final `TypingSlice` instances due to the sheer number of the possible high-level goals and Typing Slices for a type system of a mature language. That is why in Figure 6.11 we present a `TypingSliceOps` class that encapsulates a list of the commonly required Typing Slices operations. The implementation of the class, returned by the `tslicesOps` member of the `SlicesOps` trait, is tied to a particular instance of the compiler (lines 4 and 5) since the methods return the low-level compiler values.

The Typing Slice exploration method, ‘`expand`’, realizes the core idea of the guided type derivation tree analysis. When given a non-final instance of the `TypingSlice`, it will transparently use the included `TypeFocus` value and the high-level goal, in order to determine the next nodes in the type derivation tree which explain the target type. The `extractPos`, `extractType` and `extractAST` methods return the low-level position, type and AST information associated with the individual `TypingSlice` instance, respectively. The presented interface provides an abstraction layer that hides the details of the type checking algorithm (and its high-level representation) and at the same time can identify the source of target types through a repeated expansion of the inferred Typing Slices.

The expansion of the Typing Slices allows us to quickly step through the nodes of the type derivation tree: we can omit the goals that are irrelevant for the explanation of the process, and only stop and address those that we deem to be important for the purpose of the particular type debugging scenario. This means that any type debugging also has to provide access to the specialized functions that analyze the typing decisions on a more fine-grained level than just Typing Slice expansion. From the formal point of view the algorithms used in the analysis of Type Variable Typing Slices in Section 3.7 represent one example of such specialized functions.

For illustration purposes, in the next two sections we delve into the details of how the two non-trivial intermediate analysis techniques are exposed in the Type Debugger and why their availability is important for providing improved type error feedback.

6.5 Debugging function applications with elided type arguments

The *TypeFocus*-based analysis of type derivation trees reduces the analysis of function applications with elided type arguments to two distinct decisions. It first locates the type parameter which instantiation inferred the target type. Later the Typing Slice is expanded and the result, the Typing Slices, explain the origin of individual type constraints. The *TypeFocus*-based navigation hides the complex implicit relations between the high-level goals representing the inference process, such as the collection process of type constraints, their relation to the type parameter, and the handling of other local type parameters.

To complement the `TypingSlice`-approach in Figure 6.12 we present a fragment of the Type Debugger's specialized functions specific to the analysis of the type parameters' instantiation. Similarly as in the previous cases the operations on the high-level goals are defined in the nested abstract class, `InferOps`, which can be retrieved from the abstract member '`inferOps`' (line 7):

- The '`tparamInstantiation`' method (line 10) returns a typing decision representing the location in the type derivation tree where the instantiation of a single type parameter takes place. The search is triggered with a generic high-level goal and a *TypeFocus* value, rather than with a particular value of Typing Slice. This means that the method will return the answer for a range of possible type checking scenarios that elided type arguments, and the identity of the particular type parameter is defined by the `TypeFocus` value.

  The result of the analysis is then a Typing Slice, of type `SolveTParamSlice`, and belongs to the category of Type Variable Typing Slice formally defined in Section 3.4.

- The `constraints` method (lines 12-13) takes a high-level `SolveTParam` goal (the underlying goal of the `SolveTParamsSlice` Typing Slice), representing the site in the type derivation tree where the type of some type parameter is inferred. The method returns information about all low-level type constraints that were used in the process. The overloaded variant of the method allows for modifying the default optimality conditions, and return either information about the lower or upper type bounds of the type

```
1  trait InstantiationAnalysisOps {
2    self: HighLevelRepr with TypesFocus with TypingSlices =>
3
4    val global: DebuggerGlobal
5    import global.{Tree, Position, Type}
6
7    def inferOps: TypingSlicesOps
8
9    abstract class InferOps {
10     def tparamInstantiation(goal: Goal, tfocus: TypeFocus): Option[SolveTParamSlice]
11
12     def constraints(goal: SolveTParam):                        List[Constraints]
13     def constraints(goal: SolveTParam, lowerBounds: Boolean): List[Constraints]
14   }
15
16   abstract class Constraint {
17     def tparam:     Type
18     def underlying: Conformance
19   }
20
21   implicit class ConstraintOps(val self: Constraint) extends AnyVal {
22     def toType: Type =             // ...
23     def toAST: Option[Tree] =      // ...
24     def toPos: Position =          // ...
25     def isLowerBound:  Boolean =   // ...
26     def isFormalBound: Boolean =   // ...
27     def toSlice(tfocusCont: TypeFocus): Option[TypingSlice] = // ...
28   }
29 }
```

Figure 6.12: An interface for analyzing the inference of type arguments.

parameter. The details of the `Constraint` abstraction are discussed later in Section 6.5.1.

The exploration methods of the presented `InferOps` class are not complete. Rather, we provide only a glimpse of the API that should be provided by the architects of the type debugging framework since they have a good understanding of the details of the type checking process and its high-level representation.

6.5.1 Representing the type constraints

The type constraints used in the Type Debugger are defined by the `Constraint` class (lines 16-19 in Figure 6.12). The class defines only two members:

- The '`tparam`' member returns a low-level type parameter to which a type constraint corresponds.

- The '`underlying`' member returns a high-level node of the subtyping derivation tree where the type constraint was added to the type parameter's constraint set.

The abstraction on its own does not offer any capabilities; the latter are enabled implicitly using the `ConstraintOps` Value Class[1] and its members (lines 21-28). The '`toType`', '`toAST`', and '`toPos`' methods retrieve the low-level details of the individual type constraint, such as the type, the AST and the position, respectively. The '`isLowerBound`' describes if the identified type constraint belongs to the set of lower bounds, and the '`isFormalBound`' determines if the constraint comes from the type parameter or a type value that has been formally defined in the type signature. The '`toSlice`' method defines a translation from the identified type constraint into a `TypingSlice`. The translation is sufficient to trigger a *TypeFocus*-based analysis of the source of the type constraint in the type derivation tree.

Consequently, the interface provided by the `ConstraintOps` takes the burden of discovering the meaning of high-level nodes and their low-level data off the shoulders of the users, without sacrificing on the level of detail. For example, the type constraints in Scala may originate not only from the type of the argument or the type or the expected type of the application but also from the formally defined type bounds in the signatures of the methods. Without the API that exposes such information in a convenient form it would be impractical for the users of the Type Debugger to analyze in detail the process of the instantiation of type parameters.

The level of detail we attempt to deliver with the API of Type Debugger is necessary to tackle real examples. For illustration purposes we consider in Figure 6.13 a scenario of two unrelated classes `A` and `B`, and a generic `Test` class with two type parameters.

In the listing, the `Test` class defines two non-trivial polymorphic methods with a local type parameter `U` bounded either by the non-local type parameter `S` (line 3), or the local type parameter `Z` (line 5). Lines 10-11 and 17-18 of the listing give examples of function applications that later later lead to type errors. The error messages describe the nature of the conflict but refrain from explaining the source of the types involved in the conflict. The relations between the inferred types and the involved type constraints, such as in the example above, are rarely trivial to understand or analyze.

By combining the `TypingSlice`-based exploration with the specialized functions we provide tools for generating a more complete type debugging experience that can fully explain the instantiation of type parameters. To illustrate, Figure 6.14 compares the improved error feedback resulting only from the Typing Slices operations (the API defined in Figure 6.11), and the improved error feedback resulting from the Typing Slices and the analysis of the low-level details of type constraints. The error messages themselves have been adapted for the illustration purposes and their role is only to show the ability to infer detailed source code locations.

[1]Value Classes present no runtime overhead mechanism. A detailed description is provided under http://docs.scala-lang.org/overviews/core/value-classes.html.

```
class A; class B
class Test[T, S] {
  def foo[U >: S](x: U): U = x
  def bar[U >: Z, Z](x: U, y: Z): U = x
}

val x: Test[A, A] = // ...
val y: B          = // ...

val y1 = x.foo(y)
val y1Expected: Int = y1
  // error: type mismatch;
  // found    : Object
  // required : Int
  // val y1Expected: Int = y1
  //                       ^
val y2 = x.bar(y, new A())
val y2Expected: Int = y2
  // error: type mismatch;
  // found    : Object
  // required : Int
  // val y2Expected: Int = y2
  //                       ^
```

Figure 6.13: An example of the error messages caused by the inference of the type parameter instantiation from the multiple type constraints.

In fact, the operations on the type constraints reveal sufficient information to reconstruct on-the-fly the type parameter dependency graphs, along the lines of those presented in el Boustani and Hage [2011] and Hage and Heeren [2007], but without requiring a separate infrastructure.

```
Expected type comes from the inferred
instantiation for the selected
type parameter:
[U >: S](x: U): U
            ~
The type parameter U has been
instantiated using the following
locations:
  val x: Test[A, A] = // ...
                 ~
  val y: B          = // ...
         ~
  val y1 = x.foo(y)
           ~~~~~~~~
```

(a) High-level type constraints information.

```
Expected type comes from the inferred
instantiation for the selected
type parameter:
[U >: S](x: U): U
            ~
The type parameter U has been
instantiated using the following
locations:
Location (1):
  def foo[U >: S](x: U): U = x
               ~
  val x: Test[A, A] = // ...
                 ~
Location (2):
  x.foo(y)
        ~
  val y: B          = // ...
         ~
```

(b) A complete error feedback.

```
Expected type comes from the inferred
instantiation for the selected
type parameter:
[U >: Z](x: U, y: Z): U
            ~
The type parameter U has been
instantiated using the following
locations:
  val y: B          = // ...
         ~
  val y2 = x.bar(y, new A())
                    ~~~~~~~
  val y2 = x.bar(y, new A())
           ~~~~~~~~~~~~~~~~~
```

(c) High-level type constraints information.

```
Expected type comes from the inferred
instantiation for the selected
type parameter:
[U >: Z](x: U, y: Z): U
            ~
The type parameter U has been
instantiated using the following
locations:
Location (1):
  def bar[U >: Z, Z](x: U, y: Z): U = x
               ~
  val y2 = x.bar(y, new A())
                    ~~~~~~~
Location (2):
  val y2 = x.bar(y, new A())
                 ~
  val y: B          = // ...
         ~
```

(d) A complete error feedback.

Figure 6.14: A comparison of the improved error feedback for Listing 6.13 that explains the instantiation of the local type parameter. The example error messages in the first part rely only on the information from the Typing Slices, while the second part also analyzes the individual type constraints in detail.

6.6 Debugging implicit resolution

Implicit resolution is the type-driven mechanism that applies arguments implicitly. In Scala the mechanism has been used to encode the *type classes* pattern (Oliveira et al. [2010]), design the coherent architecture for the Scala Standard Library (Odersky and Moors [2009]), provide generic, highly-customizable libraries (Miller et al. [2013]), and define intuitive API for the testing libraries, such as ScalaTest and ScalaCheck. The mechanism is deeply rooted in type checking but, apart from a recent formalization attempt by Oliveira et al. [2012], efforts to explain the feature are scarce[2].

The problem of the lack of understanding of the implicit resolution is most irritating when the compiler fails to infer an implicit argument, fails to apply an implicit conversion, or selects an implicit value different from the intended one. Cryptic messages that accompany such errors leak the internal details of the libraries, leaving users with no other option than trying to fix the problem in a time-consuming *trial and error* fashion or with explicit arguments.

To reveal the lost information we instrument the implicit resolution mechanism in a similar fashion to regular type checking, and reconstruct the high-level representation in the process. In this section we present an overview of the high-level goals that are capable of representing the main selection process. The high-level representation is accompanied by another layer of abstraction that hides the undesirable internal details and provides a convenient interface for navigating its decisions. We provide two examples of self-contained algorithms that analyze the non-trivial errors involving the inference of the implicit arguments.

6.6.1 The high-level representation

The instrumentation of the implicit search, being a frequently executed type checking operation, is not enabled by default during the regular type debugging. Instead the instrumentation issues the low-level stub instrumentation event. The difference between the high-level stub goal and the complete high-level goal representing the selection process is clear in the high-level hierarchy involving the base `ImplicitSearch` class in Figure 6.15.

The two entry points of the instrumented implicit resolution process differ in the members of the `ImplicitSearchExpanded` class, since the latter reveals the internal details of the process. The '`localContext`' and the '`ptContext`' members and their types explain the two main strategies for locating and verifying the implicit values (represented by the type of the class `ContextSource`): the implicit values are first sought in the local contexts, such as the enclosing classes, or the imported packages, and only on failure the search continues in the scopes of the companion objects of the type elements of the expected type.

[2]The work Oliveira et al. [2012] differs quite significantly from the Scala implementation by, among others, missing the support for subtyping.

```
abstract class ImplicitSearch extends Goal {

  type U <: EV.ImplicitSearch
}

abstract class ImplicitSearchStub
 extends ImplicitSearch {

  type U <: EV.StubInstrumentedSearch

  def result:       ImplicitSearchResult
}

abstract class ContextSource extends Goal {

  type U <: EV.ContextSource

  def scopes:          List[ContextScope]
  def verifyImplicits: List[VerifyImplicit]
  def rankImplicits:  List[ImplicitRanking]
  def err:             Option[ErrorGoal]
}

abstract class ContextScope extends Goal {
  type U <: EV.EligibleImplicitScope

  def implicits:    List[EligibleImplicit]
}
```

```
abstract class ImplicitSearchExpanded
 extends ImplicitSearch {

  type U <: EV.ImplicitSearchExpanded

  def localContext: ContextSource
  def ptContext:   Option[ContextSource]
  def result:      ImplicitSearchResult
}

abstract class EligibleImplicit
 extends Goal {

  type U <: EV.EligibleImplicit

  def check:         Conformance
}

abstract class VerifyImplicit
 extends Goal {

  type U <: EV.VerifyImplicit

  typeImplicit:     TypeGoal
}
```

Figure 6.15: A fragment of the high-level representation of the implicit search mechanism implemented in the Scala type checker.

The individual steps of the implicit search are represented by the members of the `ContextSource` class:

- The ‘`scopes`’ member represents the pre-selection from all the available implicit values, also known as the *applicable* ones. The implicit values of the same scope are grouped together within the same `ContextScope` class, as indicated by the collection type of its only method, ‘`implicits`’.
- The ‘`verifyImplicits`’ member refers to the complete type checking operation on the previously pre-selected *eligible* implicit values; the process may involve such type operations as an inference of the elided type arguments or an inference of the arguments for the implicit parameters.
- In order to determine a single, most specific implicit value the type-correct eligible implicits are ordered based on their subtyping relation and the location of their definition. The ranking mechanism (the ‘`rankImplicits`’ method) is exposed in the `ImplicitRanking` high-level class (omitted for irrelevance).

```
trait ImplicitSearchOps {
  self: HighLevelRepr =>

  def iSearchOps: ISearchOps

  abstract class ISearchOps {

    def eligibleImplicits(isearch: ImplicitSearch):         List[VerifyImplicit]
    def usedImplicit(isearch: ImplicitSearch):              Option[VerifyImplicit]
    def allContexts(isearch: ImplicitSearch):               List[ContextSource]

    def implicitParams(impl: VerifyImplicit):               List[ImplicitArgForParam]
    def applicableImplicits(ctx: ContextSource):            List[EligibleImplicit]
    ...
  }
}
```

Figure 6.16: A fragment of the implicit search interface, allowing to discover the decision process of the implicit search without directly navigating the high-level type derivation trees. The self-type of the trait illustrates the dependency on the high-level representation when being mixed-in.

- The optional '`err`' member represents the event that reports an error that prevents the inference of the implicit value, such as the infinite expansion of the implicit values.

The high-level representation reveals not only the operations that verify the eligible implicit values, but also the hidden, typically cached pre-selection process of all the applicable implicit values identified in different scopes of the search. The `EligibleImplicit` class and its only member, '`check`', reveal the details of the pre-selection, allowing the users of the Type Debugger to programatically investigate the low-level reasons for rejecting or accepting the individual implicit values.

Having presented a high-level overview of the implicit search decision process, we are now in a position to describe the specialized analysis functions built on top of it.

6.6.2 Navigating the implicit resolution

The high-level hierarchy of the implicit resolution provides a detailed specification of the decisions that drive the process. For many applications such level of detail is likely to be unnecessary or confusing. In this section we discuss the need for the intermediate analysis interface, in a similar spirit as the one described in Section 6.5. For that purpose we discuss a selection of the methods of the interface in Figure 6.16 that are built on top of the discussed high-level representation.

The `ImplicitSearchOps` trait defines an abstract method '`iSearchOps`' member of type `ISearchOps`. The nested `ISearchOps` abstract class defines a selection of convenient navigation operations built on top of the implicit search type hierarchy:

- The '`eligibleImplicits`' method returns a list of all the eligible implicit values that matched the expected type.
- The '`usedImplicit`' finds an eligible implicit value, and its verification process, that has been selected as part of the implicit resolution process, if possible.
- The '`allContexts`' method returns the high-level goals representing the different search strategies for the inference of the implicit values.
- If the eligible implicit itself defines implicit parameters, the inference of the corresponding implicit arguments is returned by the '`implicitParams`' method.
- The '`applicableImplicits`' method locates all the applicable implicit values that are verified against the expected type.

Apart from providing the high-level analysis functions, most of the methods accept the base `ImplicitSearch` class, rather than its subclasses. The latter hides the internal details of the high-level stub goal representation, and implicitly translates the stub goals to proper high-level goals, if necessary.

6.6.3 Example: Explaining the implicit resolution selection

With the small selection of API that exposes a fragment of the implicit resolution logic we can already define algorithms that improve the non-trivial error messages. In this section we define a self-contained algorithm that improves the long standing problem of a confusing error message in the presence of ambiguous implicit values. The problem is illustrated with a code snippet in Figure 6.17 and the error messages mentioned in the comments. The problem has persisted for a number of years, and appears regularly on user-mailing lists as one of the classic pitfalls of the implementation.

The program defines two generic classes, `Foo` and `Baz` (line 1), and two polymorphic functions, '`bar`' and '`bat`' (lines 9-10). For the purpose of the example, it is only important to notice that the functions take two lists of parameters, where the second parameter list is implicit. The example also provides three implicit values of the same type, *i.e.,* '`f1`', '`f2`' and '`f3`', and an implicit polymorphic function '`f4`' that requires another implicit parameter of type `Foo[S]` (in a more realistic scenario the ambiguity is typically hidden among different scopes of the implicit resolution, making it less obvious).

The problematic scenarios, and their error messages, are illustrated in the '`test`' function where functions '`bar`' and '`bat`' are partially applied to an integer constant '`1`'. Both implicit

```scala
class Foo[A]; class Baz[B]

implicit val f1: Foo[Int] = // ...
implicit val f2: Foo[Int] = // ...
implicit val f3: Foo[Int] = // ...

implicit def f4[S](implicit ev: Foo[S]): Baz[S] = // ...

def bar[T](x: T)(implicit ev: Foo[T]): Unit =  ()
def bat[T](x: T)(implicit ev: Baz[T]): Unit = ()

def test() = {
 bar(1)         // error: ambiguous implicit values:
                // both value f1 of type => Foo[Int]
                // and value f2 of type => Foo[Int]
                // match expected type Foo[Int]
                // bar(1)
                //    ^

 bar(1)(f1)     // works

 bat(1)         // error: could not find implicit value for parameter baz: Baz[Int]
                // bat(1)
                //    ^

 bat(1)(f4(f1))  // works
}
```

Figure 6.17: An example of the error message caused by the ambiguous implicit values.

applications are rejected because the compiler will fail to statically determine a single, most specific implicit value that matches the expected type:

- The first error message indicates that the ‘`bar(1)(f1)`’ and ‘`bar(1)(f2)`’ implicit applications are equally valid. At the same time, the error message refrains from reporting the ambiguity involving the ‘`bar(1)(f3)`’ application, to keep the size of the error messages within the reasonable limits.
- The second error message simply states that no implicit of the expected type was present at all. The error is confusing because from the code snippet it is clear that at least the implicit value ‘`f4` ’should be taken into account.

Both of the problems reported by the Scala compiler are caused by the same issue, but they differ significantly in the reported error message, adding to the confusion.

We reduce the implicit resolution process to a lightweight recursive data structure that highlights the key elements of the selection process. Its sole purpose is to be able to represent the non-trivial cases of the inference of the implicit values that themselves require implicit values, or *chains of implicit values* in our terminology. The data structure consists of two case classes (defined in Figure 6.18 in lines 4-5):

- The `ImplicitPath` class represents a single eligible implicit value, as indicated through the '`underlying`' parameter. If the implicit value itself defines a dependency on some implicit parameters, the search for the implicit arguments is reflected in the non-empty value of the '`params`' parameter.
  For example, the search involving the implicit value '`f1`' would be simply represented as '`ImplicitPath('f1', Nil)`'.

- The `ImplicitParam` class holds a reference to the `ImplicitSearch` goal representing the entry point to any implicit resolution process for some implicit parameter and its expected type. The list of all eligible implicit values (potentially involving chains of implicits as well) is represented by the '`eligible`' parameter.

Figure 6.18 defines a self-contained algorithm which analyzes the implicit resolution selection and constructs the lightweight data structures representing the ambiguous chains of implicits. The analysis is divided into parts. The failed implicit resolution goal (`ImplicitSearch`) has to be first identified from a type checking node that rejected the '`bar(1)`' and '`bat(1)`' function applications (lines 7-19). Only later can we analyze the selection process itself (lines 21-29).

Without going into the details, we note that the function has to first identify the root of the failed inference of the implicit arguments in a generic way. By pattern matching with the `AdaptImplicitMethod` goal (line 6) we extract the nodes representing the decisions that infer the implicit arguments (using the '`implicitParams`' value of type `List[ImplicitArgForParam]` in line 9). The collection type indicates that the adaptation may infer a number of implicit arguments and we have to select one that failed (implicitly checked with a reference to the low-level data in '`!p.infer.underlying.result`' in line 11). The located '`failedImplParam`' value of type `ImplicitArgForParam` represents the type checking operation that failed to infer an argument for some generic implicit parameter. The `ImplicitArgForParam` class defines only a single member '`infer`' of type `ImplicitSearch` that can be interpreted as a link between the regular type checking process and the implicit resolution search process.

With the goal representing the failed implicit resolution process (`failedImplParam.infer` in line 13), we are now in a position to explain the construction of the data structure representing the chains of implicits in the '`traces`' function. Given the `ImplicitSearch` goal the algorithm first retrieves all the eligible implicit values that matched the expected type [3] (line 23). For each of the eligible implicit values, we analyze the inference of the implicit arguments

[3] To keep the example simple we assume that all such values succeeded type checking and are equally specific according to the ranking specification. The details of such encoding are irrelevant for our discussion.

```
1  trait AmbiguousImplicits {
2    self: HighLevelRepr with ImplicitSearchOps =>
3
4    case class ImplicitPath(underlying: VerifyImplicit, params: List[ImplicitParam])
5    case class ImplicitParam(underlying: ImplicitSearch, eligible: List[ImplicitPath])
6
7    def handleNoImplicitFoundError(tcheck: Typecheck): Option[String] = {
8      tcheck.adaptg match {
9        case AdaptImplicitMethod(_, implicitParams: List[ImplicitArgForParam], _) =>
10         for {
11           failedImplParam     <- implicitParams.find(p => !p.infer.underlying.result)
12         } yield {
13           val implicitWithParams = traces(failedImplParam.infer)
14           errMessageFromTraces(implicitsWithParams)
15         }
16       case _                                        =>
17         None
18     }
19   }
20
21   def traces(isearch: ImplicitSearch): List[ImplicitPath] =
22     for {
23       eligible <- iSearchOps.eligibleImplicits(isearch)
24     } yield {
25       val params = iSearchOps.implicitParams(eligible).map {
26           (implArg: ImplicitArgForParam) =>
27              ImplicitParam(implArg.infer, traces(implArg.infer))}
28       ImplicitPath(eligible, params)
29     }
30 }
```

Figure 6.18: An example of the error handler that generates improved error feedback for the rejected function applications with implicit parameters from Figure 6.17. The grayed-out parts refer to the implicit-specific code that defines the error message, and the underlined fragments highlight the usage of the specialized analysis functions available from the `ImplicitSearchOps` interface.

for the potentially non-empty list of implicit parameters (line 25). Since the goal representing the search for the implicit arguments is of type `ImplicitSearch` we trigger the ‘`traces`’ function recursively (line 27).

The `ImplicitPath` data structure reconstructed from the recursive invocation summarizes the key information of the implicit resolution selection. For example for the partial function applications ‘`bar(1)`’ and ‘`baz(1)`’ the ‘`traces`’ function will return

```
List( ImplicitPath(‘f1’, Nil), ImplicitPath(‘f2’, Nil), ImplicitPath(‘f3’, Nil) )

List(
 ImplicitPath(‘f4’,
   List(ImplicitParam(‘ev’,
     List(ImplicitPath(‘f1’, Nil), ImplicitPath(‘f2’, Nil), ImplicitPath(‘f3’, Nil))
))))
```

The result from the ‘traces’ function calls provides sufficient information to generate the improved error messages in the ‘errMessageFromTraces’ function (omitted for irrelevance). For example, the type error message resulting from the above analysis for the problematic ‘baz(1)’ function application will now be:

```
error: ambiguous implicit values:
  value f4(f1) of type => Baz[Int],
  value f4(f2) of type => Baz[Int]
  and value f4(f3) of type => Baz[Int]
  match expected type Baz[Int]
  bat(1)
     ^
```

The encoding of the algorithm is complete, in a sense that it allows us to programatically define the search for all the eligible implicit values that have failed or succeeded to be selected, and safe, because the access to the high-level dependencies and the retrieval of low-level data is statically checked. The algorithm is also concise and (with a single recursion) easy to reason about thanks to the operations provided by the API of the ISearchOps analysis (underlined in Figure 6.18).

6.6.4 Example: Implicit resolution and the limitations of local type inference

The specialized analysis functions of the implicit resolution process from Figure 6.16 are not *TypeFocus*-driven; we will discuss the *TypeFocus*-driven analysis in Section 7.1. This does not however stop it from being used in the algorithms that analyze the type inference process. In fact by combining the specialized analysis functions of different type system features we are able to explain problems that involve different type system features and were unlikely to be explained before. To illustrate our argument we will encode a simple heuristic that provides better feedback for the rejected program in Figure 6.19 where local type inference affects the implicit resolution process.

The example defines a generic comparison function ‘universalCompare’ using the type-classes pattern (line 6). The function takes two arguments of some generic type T and returns an integer value which sign communicates how the two values compare. The comparison between

```
1  abstract class A { def f: Any }
2  class B extends A { def f: Int = 5 }
3  class C extends A { def f: Long = 5L }
4
5  implicit val AOrdering: Ordering[A] = // ...
6  def universalCompare[T: Ordering](t1: T, t2: T): Int = // ...
7
8  universalCompare(2, 1)          // works
9  universalCompare(new B, new C)  // error:
10                                 // No implicit Ordering defined for A{def f: AnyVal}.
11                                 //          universalCompare(new B, new C)
12                                 //                                  ^
```

Figure 6.19: An example of a local type inference over-approximation leading to a failed inference of the implicit value.

the two generic values is possible because the type parameter `T` defines a context bound `[T: Ordering]` which is a syntactic sugar for an implicit parameter of type `Ordering[T]`[4]. The listing also defines a base class `A` with a single abstract member '`f`' of type `Any`, and subclasses `B` and `C`, each declaring the definition of the abstract member but with a refined return type. Finally, to compare values of type `A` the listing defines an '`AOrdering`' implicit value which materializes the `Ordering` for the elements of type `A`.

For example, a comparison of two integer values in line 8, allows the compiler to infer an implicit argument of type `Ordering[Int]`, resulting in a type-correct function application '`universalCompare[Int](2,1)(math.Ordering.Int)`'. It may therefore seem surprising to the user that the compiler rejects the function application in line 9 even though the intended '`AOrdering`' implicit value is available in the scope.

The interactions between local type inference and the implicit resolution, such as the one above, are typically hard to explain and may involve reporting false positives. With our precise analysis they can now be programatically identified by taking the advantage of the known typing scenarios when they can occur. In Figure 6.20 we present an intuition behind the algorithm that locates the source of the type of the implicit parameter that failed to materialize the implicit argument. In the example we focus on the role of the local type inference over-approximation in the presence of invariant type parameters which manifested itself by searching for an implicit argument of type `Ordering[Adef f:  AnyVal]` rather than `Ordering[A]`.

Similarly as in the previous algorithm, the error analysis accepts the `Typecheck` goal representing the type derivation for an individual function application and pattern matches on the adaptation goal (lines 2-3) in search of the high-level goal representing the failed inference of the implicit argument (line 5). With the index of the failed implicit parameter we can

[4]The `Ordering[S]` type for some type argument `S` defines methods that can compare elements of type `S`.

```
def analyze(tcheck: Typecheck): List[Position] = {
  tcheck.adaptg match {
    case AdaptImplicitMethod(_, params, _) =>
      for {
        failedImplParam     <- params.find(p => !p.infer.underlying.result)
      } yield {
        analyzeImplicit(failedImplParam, params.indexOf(failedImplParam))
      } getOrElse(Nil)
    case ... => // ...
}

def analyzeImplicit(implicitParam: ImplicitArgForParam, idx: Int): List[Position] = {
  for {
    context          <- iSearchOps.allContexts(failedImplParam.infer)
    applicable       <- iSearchOps.applicableImplicits(context)
    tFocus           <- tfocusOps.toError(applicable.check) if tfocus.head != IdTFocus
    tslice           <- analysis.expectedTypeSource(tcheck.adaptg,
                                                tFocus compose MethodParamTFocus(idx))
  } yield {
    val paramTpe        = failedImplParam.underlying.paramPt
    val paramTpeFocused = tfocus.focus(paramTpe)

    if (isTParam(paramTpeFocus) &&
        isInvariant(paramTpeFocus, paramTpe)) {

      val implTpeFocused  = tfocus.focus(applicable.underlying.tpe)
      inferOps.tparamInstantiation(tslice.repr, tslice.tfocus).flatMap { tparamSlice =>
        val constraints = inferOps.constraints(tparamSlice.repr)
        val found = constraints.forall { constr =>
          tparamSlice.tfocus(constr.toType).fold(
             _ <:< implTpeFocused, _ => false
          )
        }
        if (found)
          Some(
            for {
              constraint <- constraints
              slice      <- constraint.toSlice(tparamSlice.tfocus)
              // ... // expands the slice
            } yield pos)
        else       None
      }
    } else None
  } flatten
}
```

Figure 6.20: An overview of the algorithm that identifies the source of the rejected implicit value in Figure 6.19.The grayed-out parts refer to the implicit-specific code that defines the error message, and the underlined fragments highlight the usage of the previously defined specialized analysis functions.

trigger the analysis of the implicit resolution in the '`analyzeImplicit`' function (line 7).

This time, rather than looking at the eligible implicit values, we consider the applicable implicit values (line 15) that have been checked in all the available implicit scopes (line 14). Later we use the fact that the applicable implicit values are being rejected based on subtype checking and translate the failures to the equivalent *TypeFocus* values (line 16). For example, based on the failed subtype checking the rejected '`AOrdering`' implicit value infers the *TypeFocus* value:
`IdTFocus compose TypeMemberFocus('A', 'f') compose TypeArgFocus('Ordering', 0)`
When applied to the type of the implicit value

```
Ordering[A { def f: Any }]
```

and the expected type

```
Ordering[A { def f: AnyVal }]
```

the highlighted parts can already illustrate the subtyping conflict, and the reason for the rejection of the implicit value.

The reconstructed type selection is sufficient to trigger the *TypeFocus*-based analysis which finds the source of the expected type used in the implicit resolution (lines 17-18). The '`tFocus compose MethodParamTFocus(idx)`' type selection enriches the selection of '`tFocus`' in order to be compliant with the type of the partially applied function application, where the analysis is triggered from[5]. For example, for the inferred type of the partially applied function application '`universalCompare(new B, new C)`', the inferred type selection can be visually interpreted as: `(implicit ev: Ordering[A { def f: AnyVal }])Int`.

The remaining part of the algorithm (lines 20-37) limits the scope of the accepted programs to those affected by the inference of the instantiation of a single type parameter that appears in an invariant position, *e.g.,* `T` in `Ordering[T]`, as indicated through the platform-dependent '`isTParam`' (line 23) and '`isInvariant`' (line 24) functions, respectively (their definitions are omitted for irrelevance).
The result of the *TypeFocus*-based analysis, the `tslice` value, is later used to determine the part of the type derivation tree that instantiated the local type parameter (line 27), and its type constraints (line 28). In particular, the `tparamSlice` Typing Slice will represent the typing decision that instantiates the local type parameter `T` and the `constraints` value will refer to the `new B` and `new C` arguments.
The application of the `tparamSlice`'s *TypeFocus* to the types of constraints (lines 30-32) ensures that we do not report some false positives. In particular, the condition checks that the extracted type elements of the constraints, `Int` and `Long`, are both subtypes of the intended

[5]The inferred type of the function application prior to the inference of the implicit arguments is known to be of a non-value `MethodType` of shape `(implicit x0: X0, ..., xN: XN)Y`, where $X0,...,XN$ represent types of implicit parameters and Y is the result type of the method.

```
The inference of the implicit argument failed due to the inferred
instantiation of the type parameter T in
[T](t1: T, t2: T)(implicit evidence$1: Ordering[T])Int
                                                ~
Inferred instantiation of type parameter T,
Ordering[A { def f: AnyVal }]
                    ~~~~~~
used locations:
Location (1):
class B extends A { def f: Int = 5 }
                           ~~~
  universalCompare(new B, new C)
                   ~~~~~
Location (2):
class C extends A { def f: Long = 5L }
                           ~~~~
  universalCompare(new B, new C)
                          ~~~~~
You may provide explicit type arguments to fix the problem:
‘universalCompare[A](new B, new C)’
```

Figure 6.21: An example of the improved feedback generated for the example from Figure 6.19 using the algorithm from Figure 6.20.

type `Any` (as checked through the `<:<` method in line 31).
Finally, each of the identified type constraints can be transformed into a Typing Slice and further analyzed with the correct type selection on its own (lines 37-39). This not only allows us to identify the minimal conflicting elements of the type checking that led to the conflict but also suggest a correct widening of the inferred type, described in the next section. The consequence of the above algorithm is the improved feedback provided in Figure 6.21.

The reconstructed `TypeFocus` value, and its application to the types of type constraints and the type of the implicit value (lines 30-32) ensures that similarly looking, yet type-wise different typing scenarios that do not satisfy our criteria, are being rejected. For example, in a scenario

```
class D
...
universalCompare(new B, new D) // No implicit Ordering defined for Any.
```

the function application would be rejected by our algorithm and would not produce any false negatives because the type of the `D` class is not a subtype of a type `A{ def f: Any }`.

The presented algorithm is Scala-specific and ignores other type checking scenarios or applicable implicit values. This does not however diminish the results of our solution; with a small number of *TypeFocus*-based operations and statically verified type derivation tree exploration we are capable of encoding a generic algorithm that recognizes a non-trivial language limitation.

6.7 Code modifications

The *TypeFocus*-based analysis of type derivation trees provides not only minimal program locations that explain the origin of types but also foundations for developing precise heuristics on top of it. One prominent example involves code modifications that *patch* known local type inference limitations. To exploit such an opportunity we notice that `TypingSlices` represent the self-contained fragments of the type checking process that are easily identifiable.

In Section 6.7.1 we present key components of the type debugger's infrastructure, sufficient for identifying and manipulating the type checking decisions. To give an intuition behind the range of possibilities offered by our proposal, we consider the analysis of two real-world examples (Section 6.7.2 and Section 6.7.3).

6.7.1 Typing Slices-related suggestions

The localized analysis of Typing Slices dictates the correcting mechanism for each of the subclasses of the `TypingSlice`. We notice that many of the confusing type errors, for which we would like to offer code modifications, appear as terminal or close-to-terminal `TypingSlices`. This way they also identify the smallest code snippets to modify. The interface for the analysis of the individual `TypingSlices` is summarized in Figure 6.22.

The `SliceFixes` trait presents an overview of the code modification infrastructure, which consists of the three main parts:

- The sealed `FixKind` abstract class (lines 26-30) groups together possible kinds of code modifications. For brevity, our selection lists only three classes, `TypeAnnotationFix`, `TypeArgFix`, and `MultiFix`, corresponding to the low-level information that suggests an explicit type annotation (at a location specified by the low-level '`where`' parameter), an explicit list of modified type arguments, or a collection of multiple modifications, respectively. Other changes, not listed in the provided code snippet include: modifications to type signatures of the definitions, or modifications of the structure of the expressions.
- The '`FixForSlice`' abstract class (lines 7-16) defines the error correction handlers for different subclasses of the `TypingSlice` (lines 8-13). The default implementation of the `FixForSlice` class, and its handlers, will delegate to a regular `TypingSlice` expansion. Depending on the code modification heuristic, the individual methods of the default

```scala
trait SliceFixes {
  self: HighLevelRepr with TypesFocus with TypingSlices =>

  val global: DebuggerGlobal
  import global.{Tree, Type}

  abstract class FixForSlice {
    def forTParam(slice: SolveTParamSlice):           Option[FixKind]
    def forIfSlice(slice: InferLubForIfSlice):        Option[FixKind]
    def forLiteralSlice(slice: LiteralConstantSlice): Option[FixKind]
    def forIdentSlice(slice: IdentSourceSlice):       Option[FixKind]
    ... // other slice handlers

    def forSlice(slice: TypingSlice):           Option[FixKind]
    def forGoal(slice: Goal, tfocus: TypeFocus): Option[FixKind]
  }

  def sliceFixesOps: SliceFixesOps

  abstract class SliceFixesOps {
    def tpeMismatchFix(expected: Type):            Option[FixForSlice]
    def noImplEvidenceFix(iSearch: ImplicitSearch): Option[FixForSlice]
    // .. other default correction techniques
  }

  sealed abstract class FixKind

  case class TypeAnnotationFix(where: Tree, annotation: Type) extends FixKind
  case class TypeArgFix(where: Tree, targs: List[Type]) extends FixKind
  case class MultiFix(fixes: List[FixKind]) extends FixKind
}
```

Figure 6.22: An overview of the infrastructure for opportunistic code modifications.

implementation would need to be overridden.

The code modification analysis is triggered by one of the generic ‘`forSlice`’ or the ‘`forGoal`’ methods which dispatch to one of the `TypingSlice` handlers.

- The ‘`sliceFixesOps`’ member returns a default implementation of the correction mechanisms as implemented by the type debugger.

Different type checking circumstances and errors may dictate different strategies for resolving conflicts, as represented by the methods of the `SliceFixesOps` class. For example, the ‘`tpeMismatchFix`’ method will return a `TypingSlice` analysis, specialized for type mismatch problems, such that the ‘`expected`’ parameter describes the intended type of the modification, while the ‘`noImplEvidenceFix`’ method will return a `TypingSlice` analysis specialized for the failed implicit argument inference (the algorithm described in Section 6.6.4 being one example).

The structure is sufficient to define surgically-precise code modifications associated with `TypingSlices`, since it considers every instance individually. It can also be triggered at any point during the navigation of the type derivation tree. In that sense, the code modification mechanism combines seamlessly with any *TypeFocus*-based analysis, including the *strict* `TypingSlice`-expansion.

To illustrate, we now consider two examples for the existing Scala programs.

6.7.2 Example: Overcoming the limitations of local type inference

The discussion in Chapter 4 has formally explained the source of type mismatch that occurred within the motivating example of the `foldRight` application. In Scala we consider two similarly looking function applications that are equivalent to the formal encoding:

```
val xs: List[Int] = // ...
xs.foldRight(Nil)((x: Int, ys: List[Int]) => (x + 1) :: ys)
xs.foldRight(List())((x: Int, ys: List[Int]) => (x + 1) :: ys)
```

Since the object value `Nil` in Scala extends the `List[Nothing]` class, the two function applications result in the same type mismatch between the `List[Int]` and `List[Nothing]` types. At the same time since `List()` is a function application involving a polymorphic `apply` member of `List`, the two examples lead to subtle, yet significant, differences in type checking.

With the presented type debugging infrastructure we can now take advantage of the type error and the typing context of where it occurs, to provide a unified solution for both of the cases. In Figure 6.23a we define a heuristic that suggests code modifications based on the `TypingSlice` values that identify the source of the error; the `TypeMismatchFix` class extends the default code modification logic, the `BaseFix`, where the latter only defines the Typing Slice expansion but no code modification. For the purpose of the above examples we provide specialized handlers for: the `SolveTParamSlice` Typing Slice (line 2) representing the typing decisions that infer the instantiation of some type parameter, and the `IdentSourceSlice` Typing Slice (line 24) representing the typing decisions that inferred the type of some identifier. Later we describe the seamless integration of the opportunistic code modifications with the regular *TypeFocus*-based analysis that is concerned only about finding the minimal program locations for generating improved error feedback (Figure 6.23b).

The heuristic defining the analysis of the instantiation of the type parameter (line 2) comes down to the retrieval of type constraints that affected it (line 3) and the analysis of the number and types of constraints (lines 4, 15, and 20). In particular, we consider scenarios where either no constraint was involved, or one constraint, or more, respectively. The use of the specialized analysis functions allows us to concentrate on defining properties necessary for

```
class TypeMismatchFix(val expected: Type) extends BaseFix {
  override def forTApp(slice: SolveTParamSlice): Option[FixKind] = {
    inferOps.constraints(slice.repr) match {
      case Nil =>                                  //  No constraints
        val inferredTpe = slice.repr.instantiate.underlying.tpeInst
        val expectedTpe = slice.tfocus.update(inferredTpe, expected).getOrElse(ErrorType)
        if (isMinimal(slice.repr.underlying.tparam) && (inferredTpe <:< expectedTpe)) {
          val (funTree, targs) =
            codeModification.targs(slice.repr, expectedTpe, inferredTpe)
          Some(TypeArgFix(funTree, targs))
        } else {
          // ... other heuristic
        }
      case cs :: Nil =>                            // One constraint
        val inferredTpe = slice.repr.instantiate.underlying.tpeInst
        val expectedTpe = slices.tfocus.update(inferredTpe, expected).getOrElse(ErrorType)
        if ((inferredTpe <:< expectedTpe)
          if (slice.tfocus.head != IdTFocus)) default(slice)
          else                              Some(TypeAnnotation(..., expectedTpe))
        else // ...
      case cs  => // ...                           // Multiple constraints
    }
  }
  override def forIdentSlice(slice: IdentSourceSlice): Option[FixKind] = {
    val inferredTpe = slice.repr.underlying.identTpe
    val expectedTpe = slice.tfocus.update(identTpe, expected).getOrElse(ErrorType)
    if ((inferredTpe <:< expectedTpe))
      Some(TypeAnnotation(slice.repr.underlying.tree, expectedTpe))
    else // ...
  }
}
```

(a) A fragment of the `TypingSlice`-based heuristics that suggests type mismatch corrections.

```
def tpeMismatchError(failedAdaptation: AdaptGoal): Option[ErrorFeedback] = {
  failedAdaptation match {
    case TypesNotConform(conf: ConformanceCheck, fallback: FallbackAdaptation) =>
      for {
        tfocus      <- tfocusOps.toError(conf)
        inferredTpe <- tfocus.focus(conf.underlying.tpe1)
        expectedSrc <- analysis.expectedTypeSource(failedAdaptation, tfocus)
      } yield {
        val optionalFix = sliceFixesOps.tpeMismatchFix(inferredTpe).
                            flatMap(_.forSlice(expectedSrc))
        val slices = expandUntilCompletion(expectedSrc)
        val poss = slices.map(tslicesOps.extractPos)
        ErrorFeedback(poss, optionalFix,
          tslicesOps.extractType(expectedSrc), expectedSrc.tfocus)
      } headOption
    case ... =>
  }
}
def expandUntilCompletion(slice: TypingSlice): List[TypingSlice] = slice.expand match {
  case Nil             => slice :: Nil
  case expanded :: Nil => expandUntilCompletion(expanded)
  case slices          => slices.flatMap(slices)
}
```

(b) Example usage of the `TypingSlice`-based code modifications.

Figure 6.23: Explaining limitations of local type inference through code modifications.

our heuristic to apply: the ‘`expected`’ parameter (line 1) carries complete type information about the conflicting type and we want to make sure that at each `TypingSlice` the inferred type (`inferredTpe`) and the conflicting type (`expectedTpe`) are subtypes to avoid false positives (lines 7,17, and 27). While the ‘`expected`’ value, such as `Int` in our example, is not always comparable with the inferred type parameter instantiation (lines 5 and 15), such as `Int` and `List[Nothing]`, it is by design comparable with the part of the latter, represented by the `TypeFocus` of the `TypingSlice`. This means that with the ‘`update`’ method of `TypeFocus` we can reconstruct type `List[Int]` from the inferred type `List[Nothing]` and the expected conflicting type `Int` (line 16), and, using the same logic, type `Int` from the inferred type `Nothing` and the expected conflicting type `Int` (line 6).

The code modification infrastructure does not enforce of how and when a modification should be proposed, therefore a proposition to provide an explicit type annotation for a single type constraint (line 19), *e.g.,* `List(): List[Int]`, is an acceptable solution for some circumstances. However, if the `TypeFocus` of the `TypingSlice` is not an identity, we can encode even less invasive code changes by delaying the code modification to the *expanded* `TypingSlice` (line 18, represented by the omitted `default` method). It is at this stage, where the analysis of the two similar `foldRight` applications diverges because the *TypeFocus*-based expansion will

- Locate and explain the source of the selected part of the type of the `Nil` term through the `IdentSourceSlice` slice (line 24), meaning that the target type `Nothing` originates in the inferred type of the `Nil` identifier.
- Locate and explain the source of the selected part of the type of the `List()` application through the `SolveTParamSlice` slice (line 2), meaning that the target type `Nothing` is a result of type parameter instantiation in function application term.

The divergence is not problematic in our `TypingSlice` approach because we can consider each of the cases individually, as presented in lines 25-29 and 4-13 of Figure 6.23a, and suggest a `TypingSlice`-specific code modification, if possible. We summarize the key elements of the (simplified) code modification analysis as:

- Reconstructing the complete expected types with the help of `TypeFocus` type selection (lines 6 and 26) and comparing them with the type inferred originally by the compiler.
- The over-approximation of local type inference due to a lack of type constraints can be easily identified through the retrieval of type constraints, or rather lack thereof, used in instantiating the type parameters (line 4) and identifying the optimality of the solution (the definition of the implicit function `isMinimal` returns a Boolean value `true` if the inferred solution was minimal, is omitted for irrelevance).
- For code modifications proposing explicit type arguments, for completeness, we have to take into account other local type parameters and their instantiations (lines 8-9).

Only with such information can we propose a complete set of explicit type arguments that can potentially correct the type error.

- Since the identifier term does not accept type parameters, the code modification for the Scala's `Nil` term will propose an equivalent solution in terms of a type annotation (line 29).

For completeness, Figure 6.23b illustrates the convenience of using such opportunistic modifications and their seamless integration with the regular erroneous type checking scenarios. The combined analysis provides sufficient information to generate error messages such as the one in Figure 6.24.

```
Expected type comes from the inferred         Expected type comes from the inferred
instantiation for the selected type parameter: instantiation for the selected type parameter:
(z: B)(op: (Int, B) => B)B                    (z: B)(op: (Int, B) => B)B
                      ~~                                             ~~

Locations that affected the inference of      The part of the selected type has been
the selected expected type:                   instantiated due to lack of type constraints in:
  xs.foldRight(Nil)((x: Int, ys: List[Int]) =>  xs.foldRight(List())((x: Int, ys: List[Int]) =>
               ~~~                                           ~~~~
You may try to annotate your code like:       You may provide explicit type arguments like:
Nil:List[Int]                                 List[Int]()
```

Figure 6.24: An example of an improved type error message, explaining limitations of local type inference.

In Figure 6.23b, the error handler method is invoked with an adaptation goal represented by the `failedAdaptation` parameter. For simplicity, the example ignores the presence of other, similarly looking, type mismatch errors and we use the `TypesNotConform` pattern of one of the `AdaptationGoal` subclasses in order to gain access to the failed subtyping derivation tree (the type annotations in line 3 are only for illustrative purposes). With such information in place we are able to:

1. Construct a `TypeFocus` representation from a failed subtyping relation (line 5), which is part of the `TypesNotConform` goal's dependency.

2. Determine the part of the type assigned to the term that conflicts with the expected type, based on the inferred `TypeFocus` value and its application (line 6). For example type `Int` in `List[Int]`.

3. Trigger a *TypeFocus*-driven analysis of the source of the expected type, given the information from the failed subtyping relation (line 7). By definition, the '`expectedTypeSource`' analysis performs only a shallow exploration of the type derivation tree.

   For illustration purposes, we perform a simple expansion of the '`expectedSrc`' slice with the '`expandUntilCompletion`' function (line 19-23) that expands Typing Slices until completion, and returns only the terminal slices.

4. Perform a default, `TypingSlice`-based analysis of the conflict that opportunistically attempts to find a possible code modification based on the synthesized type of type mismatch, the '`inferredTpe`' value, and the first `TypingSlice` explaining the source of the expected type, the '`expectedSrc`' value (lines 9-10). Importantly, the regular analysis of the type mismatch, which focuses on finding the origin of the expected type, is independent from the logic that determines the opportunistic code modification.

5. Provide a concise summary of the error, by including the detailed position information on the source of the conflicting type as well as the optional code modification in the form of the `ErrorFeedback` class (omitted for irrelevance). The latter will be used to generate a succinct error message such as the one in Figure 6.24.

The algorithm that generates improved error feedback is generic, in the sense that it is entirely driven by the `TypeFocus` inferred from a failed type mismatch. At the same time, the innocuous subtype checking between the reconstructed expected type and the type of constraints (lines 7, 17, and 27 in Figure 6.23a), which takes into account the type selection, is sufficient to prevent us from generating false-negatives, such as the one in Figure 6.25.

```
val xs: List[Int] = // ...
xs.foldRight(0)((x: Int, ys: List[Int]) => (x + 1) :: ys)
 // error: type mismatch;
 // found    : List[Int]
 // required : Int
 // (x + 1) :: ys
 //         ^

...
You may try to annotate your code like:
0:List[Int]
```

Figure 6.25: An example of an invalid code modification suggestion that is avoided by our algorithm.

For the example from Figure 6.25, the suggested code modification is precise, in a sense that it located the source of the error, but is not type-safe because the type of the integer constant is not a subtype of the expected type `List[Int]`.

Debugging conditional constructs

Another classical example, where local type inference falls short of global type inference approach, includes the implications of calculating an over-approximated least upper bound from the different branches of the conditional construct or pattern matching.

In Figure 6.26 we consider a case where two different case classes can be returned as a result of the conditional computation in some local '`flag`' function (line 4). The result is assigned

```
abstract abstract class Base; case class Foo(x: Int) extends Base
                              case class Bar(y: Int) extends Base

def flag(cond: Boolean) = if (cond) Foo(0) else Bar(0)
var store = flag(false)
...
val modify1 = Foo(1)
val modify2: Base = Foo(2)
...
store = modify1  // works
...
store = modify2  // error: type mismatch;
                 // found   : modify.type (with underlying type Base)
                 // required: Product with Serializable with Base
                 // store = modify2
                 //         ^
```

Figure 6.26: An example of a type inference over-approximation for the conditional construct.

to a mutable variable ‘`store`’, but a consecutive assignment yields a confusing type error. The conflict is even more surprising for the ‘`modify2`’ assignment since it was initialized with an explicit type annotation, and `Base` is a subtype the `Foo` and `Bar` classes. We first delve into the details of how the type checking of the conditional construct is represented (Figure 6.27), and later we show how with our approach it becomes trivial to overcome such type inference limitations and propose a code modification algorithm that extends the one from the previous example.

```
abstract class TypeIf extends TypeGoal {
  type U <: EV.TypeIf

  def condition: Typecheck
  def thenP:     Typecheck
}
```

```
abstract class TypeIfElse extends TypeIf {
  def elseP:     Typecheck
  def lubCalc:   CalculateLub
}
```

Figure 6.27: A high-level representation for type checking the conditionals.

The high-level representation of the conditional construct (Figure 6.27) is represented through the `TypeIf` and `TypeIfElse` classes, corresponding to the conditional construct with and without the *else* statement. The dependencies of the class describe the type checking of the condition (method ‘`condition`’), the type checking of the *then* statement (method ‘`thenP`’), the type checking of the *else* statement (method ‘`elseP`’) and the calculation of the least upper bound (method ‘`lubCalc`’). That information is sufficient for our use-case, because it implicitly reveals the details of the approximation between the statements of the conditional construct.

```
1  override def forIfSlice(slice: IfSlice): Option[FixKind] = slice.repr match {
2    case TypeIfElse(_, thenPart, elsePart, _) =>
3      val inferredTpe     = thenPart.underlying.tpe
4      val expectedThenTpe = slice.tfocus.update(thenPartTpe, expected).gerOrElse(ErrorType)
5      if (thenPartTpe <:< expectedThenTpe) default(slice)
6      else                                 None
7    case _: TypeIf => // ...
8  }
```

Figure 6.28: A fragment of the `TypingSlice`-based heuristics that suggest type mismatch corrections when the origin of the error comes from type checking conditional constructs.

With the high-level representation we are able to present the extension of the code modification algorithm from Figure 6.23a without sacrificing its integrity. Figure 6.28 illustrates an extension to the `TypeMismatchFix` class that overrides a handler method for the `TypingSlice` `IfSlice`. The `IfSlice` identifies the source of some target type in the type checking of the conditional construct. In other words, the example makes use of the fact that the underlying `TypingSlice`-based analysis is correct and will eventually expand to the source of the conflicting type, *i.e.,* the conditional construct. The `TypeFocus`-substitution in line 4 allows us to state, if the narrowing of the inferred is allowed. In our example, the `<:<` operation between the two reconstructed types would verify that the inferred type `Product with Serializable with Base` is a subtype of the expected `Base` type. By delegating the construction of the code modification to the further expanded consecutive `TypingSlice` instance (line 5), we allow for a more specialized error feedback, such as

```
You may try to annotate your code like:
if (cond) Foo(0):Base else Bar(0)
```

rather than

```
You may try to annotate your code like:
(if (cond) Foo(0) else Bar(0)): Base
```

The two examples illustrated the ease with which one can encode heuristics for improving the local type inference limitations. Consequently, the `TypingSlice`-based code modification approach is sufficient to overturn many of the long standing arguments describing the limitations of local type inference with respect to its global counterpart, such as the ones mentioned prominently in the work of Hosoya and Pierce [1999].

```
<T> void foo(a: T, b: T,
       c: Map<? super T, ? super T>) {}

Number x = // ... ;
Map<Number, Double> y = // ... ;
foo(1, x, y);

// internal error; cannot instantiate
// <T>foo(T,T,Map<? super T,? super T>)
// to (int,java.lang.Number,
//   Map<java.lang.Number,java.lang.Double>)
//   foo(1,x,y);
//      ^
```

(a) A Java version

```
def foo[T](a: T, b: T, c: Map[_ >: T, _ >: T]) {}

val x: Number = // ...
val y: Map[Number, Double] = // ...
foo(1, x, y)

// error: type mismatch;
// found   : Map[Number,Double]
// required: Map[_ >: Any, _ >: Any]
//    foo(1, x, y)
//              ^
```

(b) A Scala version

Figure 6.29: A subtyping conflict for a polymorphic function application example taken from [El Boustani and Hage, 2010, pg. 13].

6.7.3 Example: Java Generics

In this section we briefly compare the capabilities of our type debugger with the work of El Boustani and Hage [2010], the only attempt at providing more informative error messages for the set of problems close to ours, on the example of Java Generics. There, the authors develop a separate constraint generation mechanism for the function application terms and a number of heuristics that inform the user about the nature of the problem as well as offer potential code modifications that might fix the type mismatch.

In Figure 6.29 we present two equivalent function applications written in Java and Scala, and their corresponding type error messages produced by their reference compilers. The error messages make it obvious that the type constraints and the inferred types differ significantly, but the variant of Java's corrective hints can still be implemented within the frames of our *TypeFocus*-based analysis.

The original Java type error message in Figure 6.29a reveals little of the nature of the problem, increasing the importance of any additional feedback that would help users to go back on track. In [El Boustani and Hage, 2010, Section 5.4] the authors provide one of the suitable heuristics for resolving the constraint set, `{Integer <: T, Number <: T, T <: Double, T <: Number}`, generated from the function application. Being a heuristic, the approach relaxes the strict optimality conditions. For example, it infers the instantiation from the constraint set by considering approximations of both type bounds of the type parameter, choosing one that causes a least number of conflicts, and suggesting the modification of those that do not conform to the inferred new instantiation (Figure 6.30a).

In our approach the expansion of the Typing Slices will by default *only* lead to an identification of the involved type constraints, `Int` and `Number`, as shown in Figure 6.30b. In order to

```
Method <T>foo(T, T, Map<? super T, ? super T>)
is not applicable to the arguments of
type(int, Number, Map<Number, Double>), because:
 [*] The type Double in Map<Number, Double> on
 11:9(11:21) is not a supertype of the inferred
 type for T: Number. However, replacing Double on
 11:21 with Number may solve the type conflict.
```

(a) Improved error feedback in Java.

```
Expected type comes from the inferred
instantiation for the selected type parameter:
(a: T, b: T, c: Map[_ >: T, _ >: T])Unit
                             ~
Locations that affected the inference of the
selected type parameter (directly or indirectly).
Location (1):
    val x: Number = // ...
           ~~~~~~
Location (2):
    val y: Map[Number, Double] = // ...
               ~~~~~~
Location (3):
    foo(1, x, y)
        ~
You may try to annotate your code like:
1: Number
```

(b) Improved error feedback in Scala.

Figure 6.30: Error feedback for the erroneous examples from Figure 6.29.

provide a correction mechanism, on a par with the Java heuristics, we would have to define the implementation of the algorithm for the case when the type parameter is instantiated with multiple type constraints (line 21 in Figure 6.23a). We refrain from modifying the involved optimality conditions that select the type constraints to avoid user confusion. Nevertheless, with a combination of the value of the expected type, that has led to the conflict, and the detection of the appropriate `TypingSlice` slice in the *TypeFocus*-based analysis, we can offer a higher confidence in the correctness of our code modification.

For example, we recall that the Scala example from Figure 6.29b will trigger a *TypeFocus*-based analysis with an expected type `Number`. This allows us to immediately identify the single conflicting type constraint (using the algorithm from Figure 6.23a) coming from the constant value and suggest a local type annotation that in fact delivers a type-correct change.

6.8 Conclusions

We have presented an overview of the key elements of the type debugging infrastructure for producing improved feedback. The compiler required only a small number operations to be exposed in order to control the instrumentation. In practice, with programming languages supporting some form of meta-programming (such as Burmako [2013a]) the ability to trigger the low-level type checking operations is more likely to be already supported by the compiler infrastructure.

The implementation reveals that the *TypeFocus* and the Typing Slice abstractions translate in a straightforward way to Scala constructs. With the amount of high-level goals representing a range of different type system features, the ability to navigate them in a controlled and still generic way becomes even more important in the type debugging techniques. At the same time the implementation reveals that for generating improved feedback for the real Scala programs we need to have the ability to analyze the particular Typing Slices and relate them to the actual source code. In our implementation of the type debugger we solve this duality by exposing the Typing Slice-specialized analysis functions that are controlled by the `TypeFocus` values. In practice, the `TypingSlice` expansion itself is largely implemented in terms of those specialized analysis techniques as well.

The *TypeFocus*-based analysis is not only suitable for explaining errors but also defining code modifications. The code modifications are local, meaning that it becomes possible to reject other program locations that did not affect the inference of the desired type. This also means that our changes may still be unsound in a global context and any inconsistencies in the proposed modifications could only be revealed through complete type checking runs. With the type errors being highly localized in languages using local type inference we believe that the latter is an acceptable limitation of our approach.

Chapter 7

Applications

Our experience with developing the type debugging tool has shown that only the integration of reconstructed high-level type derivation trees (Chapter 5) and the navigation techniques that guide its exploration (Chapter 3) yields results that are useful for beginner and expert users:

1. Interactive type derivation trees (Section 7.3), the nodes of which can be collapsed and expanded at will, are visually appealing, but without any automatic guidance through the type checking process, the non-experienced users can quickly get lost in the details of the type checking process.
2. The complete Scala language integrates a number of type system features, and language constructs. Rather than artificially limiting the already available language features we chose to use them as a challenge that tests the capabilities of our guided navigation analysis. The type checker implementation incorporates a number of exceptions and fallback mechanisms, that are often exploited by the language users and become the integral part of the language ecosystem.

In this section, we present our experience of integrating the two approaches to explain advanced type system features, that have not been yet covered in the thesis. In Section 7.1 we present a guided approach to explaining the combined decisions of local type inference and implicit resolution. The specialized *TypeFocus*-based analysis allows us to reveal the links between the inferred returned types of function application terms and their function, type or value arguments, when they are separated by non-trivial implicitly inferred and instantiated arguments.

In Section 7.2 we discuss the application of the type debugging framework as a platform for providing more informative type error messages, and potential error corrections, in the presence of real-world examples. In particular we illustrate how with the high-level representation we provide the possibility of explaining. Finally, we show that debugging the decisions

```
import shapeless._

trait EmptySeqs[L <: HList, Out <: HList] {
  def apply() : Out
}

{
 implicit def hnilEmptySeqs: EmptySeqs[HNil, HNil] = // ...
 implicit def hlistEmptySeqs[H, T <: HList, POut <: HList](
        implicit est : EmptySeqs[T, Out]): EmptySeqs[H :: T, Seq[H] :: POut] = // ...

 def emptySeqs[T <: HList, OutT <: HList](x: T)(implicit es: EmptySeqs[T, OutT]): OutT=
   es.apply()

 def foo[T](a: T): Unit = ()

 def test {
   val x = emptySeqs(1 :: "abc" :: true :: HNil)

   foo[Seq[Int] :: Seq[String] :: Seq[Boolean] :: HNil](x)   // ok
   foo[Seq[Int] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)  // error
 }
}
```

Figure 7.1: Encoding type-safe collection construction using implicit parameters.

of local type inference in an interactive and controlled way fits perfectly with the analysis approach that analyzes the source of types in multiple stages, rather than giving a final answer immediately.

7.1 Automatic explanation of the implicit resolution

The high-level representation of the implicit resolution mechanism (Section 6.6) has been primarily used to explain the decision process behind the rejected expressions. Unfortunately, a successful implicit resolution, *i.e.,* one where the implicit argument is found, is often equally, if not harder, to explain and can still lead at a later point to type errors. For example, in Scala it is common for the inferred result type of function to be path-dependent on the value of the inferred implicit argument (the principles of the design-pattern have been described for example in Doenitz [2012], and lies at the foundation of for example the type-driven serialization library, Scala Pickling in Miller et al. [2013]).

We illustrate the problem in Listing 7.1 with the program inspired by the popular Shapeless library[1] providing generic programming capabilities, and its implementation of the heterogeneous lists, `HList`. `HList` supports similar operations as the regular `List` collection, except

[1]shapeless.org

for being able also to statically define the type of the individual elements. For example,

```
val y: HList[Int :: String :: HNil] = 1 :: "a" :: HNil
```

assigns a two element list, constructed by prepending the integer value to another heterogeneous list, which in turn is constructed by prepending a string value to an empty list, `HNil`. The `::` notation defines an infix type constructor with two type parameters describing the type of the head and the tail of the list, respectively.

Listing 7.1 constructs a heterogeneous list of empty sequences from another heterogeneous list, such that the underlying type of each of the sequences corresponds to the type of the elements of the initial heterogeneous list; in short, the initial heterogeneous list serves as a schema for the encoding[2]. The type signature of the `emptySeqs` method asserts the encoding from any heterogeneous list, of a generic type `T`, to an appropriate list of sequences, of the inferred type `OutT`, through the type of the implicitly resolved parameter `es`. For example, in line 17 of Listing 7.1 the `emptySeqs` method is applied to a list of 3 elements of types `Int`, `String` and `Boolean` and returns a list of equal length, where its elements are of types `Seq[Int]`, `Seq[String]` and `Seq[Boolean]`, respectively.

The individual elements of the initial list are disassembled and assembled one-by-one in a generic head/tail fashion through the encoding involving the `EmptySeqs` data structure (lines 3-5); the type parameter `L` of the `EmptySeqs` trait represents the type of the unprocessed list, and the `Out` type parameter represents the inferred type of the translation. The encoding is inferred in a type-driven fashion, using the `hlistEmptySeqs` and `hnilEmptySeqs` implicit values; the length and type of the translation is expressed through the function applications of a form `hlistEmptySeqs(hlistEmptySeqs(... hnilEmptySeqs))`, where the size of the function application is dependent upon the length of the initial heterogeneous list that undergoes the translation. The purpose of the function applications in lines 19 and 20, involving a simple generic function `foo`, is to illustrate explicitly the inference of a desired type or lack thereof. As expected, the compiler agrees with the explicit argument of the first application but rejects the latter, resulting in a non-trivial type error message in Figure 7.2.

The type mismatch message from Figure 7.2 is still readable for a Scala programmer and, with the help of the `TypeFocus` highlighting inferred from the failed subtyping relation, it can now be further improved to show the exact conflicting type elements. In practice, the type mismatch errors often become arbitrarily long, and quickly lose the merit of any information about the involved type constructors and their type arguments. The reason is that in function applications involving implicit arguments, the errors explain the internal details of the libraries or domain-specific languages and, even when accompanied by the detailed description of the inferred implicit values, such as the one in Listing 7.3, they make little sense and hardly relate to the explicit arguments or functions defined by the users.

[2]The problem is a simplification of a real example posted on the mailing list that led an obscure type error message. While simpler, it still allows us to distill the challenging problems of the implicit resolution.

```
type mismatch;
 found   : shapeless.::[Seq[Int],shapeless.::[Seq[String],shapeless.::[Seq[Boolean],
            shapeless.HNil]]]
 required: shapeless.::[Seq[Int],shapeless.::[Seq[Boolean],shapeless.::[Seq[Boolean],
            shapeless.HNil]]]
    foo[Seq[Int] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
                                                         ^
```

(a) Original type mismatch error.

```
type mismatch;
 found   : shapeless.::[Seq[Int],shapeless.::[Seq[String],shapeless.::[Seq[Boolean],
            shapeless.HNil]]]
                                                 ~~~~~~
 required: shapeless.::[Seq[Int],shapeless.::[Seq[Boolean],shapeless.::[Seq[Boolean],
            shapeless.HNil]]]
                                                 ~~~~~~~
    foo[Seq[Int] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
                                                         ^
```

(b) Enhanced type mismatch error with `TypeFocus` highlighting.

Figure 7.2: Basic type mismatch errors for the problematic application in Listing 7.1.

```
emptySeqs(1 :: "abc" :: true :: HNil)(
  hlistEmptySeqs[Int, String :: Boolean :: HNil, Seq[String] :: Seq[Boolean] :: HNil](
    hlistEmptySeqs[String, Boolean :: HNil, Seq[Boolean] :: HNil](
      hlistEmptySeqs[Boolean, HNil, HNil](hnilEmptySeqs))))
```

Figure 7.3: An example from Listing 7.1 with the implicit arguments that are normally elided.

We present a state-of-the-art `TypeFocus`-based approach to debugging the inferred values of the implicit resolution; the analysis of the propagation of the expected type that instantiates local type parameters allows us to identify links between the inferred types of type checked function applications and the non-implicit elements of functions and arguments. Section 7.1.1 discusses a *deterministic* `TypeFocus`-based analysis, where the examination relies on maintaining a type selection only on the local type parameters of the generic implicit values, and keeping track of type propagation that instantiates them. Section 7.1.2 presents an approach with weaker guarantees where the parts of the inferred type do not only depend upon the instantiations of local type parameters of the implicit values.

Expected type	Inferred implicit
`EmptySeqs[Int :: String :: Boolean :: HNil, ?]`	$l_1 =$ `hlistEmptySeqs(`l_2`)`
`EmptySeqs[String :: Boolean :: HNil, ?]`	$l_2 =$ `hlistEmptySeqs(`l_3`)`
`EmptySeqs[Boolean :: HNil, ?]`	$l_3 =$ `hlistEmptySeqs(`l_4`)`
`EmptySeqs[HNil, ?]`	$l_4 =$ `hNilEmptySeqs`

Inferred implicit	Inferred type
l_1	`EmptySeqs[Int :: String :: Boolean :: HNil, Seqs[Int] :: Seqs[String] :: Seqs[Boolean] :: HNil]`
l_2	`EmptySeqs[String :: Boolean :: HNil, Seqs[String] :: Seqs[Boolean] :: HNil]`
l_3	`EmptySeqs[Boolean :: HNil, Seqs[Boolean] :: HNil]`
l_4	`EmptySeqs[HNil, HNil]`

Figure 7.4: A summary of the inputs and output parameters of the implicit resolution for the example from Listing 7.1. The elements of the implicit search resolution are inferred from the `emptySeqs(1 :: "abc" :: true :: HNil)(l1)` function application.

7.1.1 The *deterministic* analysis of the implicit resolution

Analysis of the chains of implicit values is reminiscent of analyzing type checking of non-implicit function applications, modulo the presence of the search mechanism and the fact that local type parameters can get partially instantiated. In both cases the propagation of the partial expected type plays a crucial role in instantiating type parameters. In consequence, the simplest, yet incomplete, form of the analysis of the implicit resolution is just a special case of the shallow `TypeFocus`-based analysis. The analysis is incomplete because it will identify only the first nested implicit that introduced the part of the inferred type.
To illustrate the problem on our motivating example in Figure 7.4 we take apart the inferred implicit values from Listing 7.3 and identify the expected type and the inferred type at every step of the implicit resolution.

Based on the failed subtyping from the type mismatch error, the regular *TypeFocus*-based analysis will locate the instantiation of the local type parameter that takes part of the implicit value l_2, as the source of the target type

```
hlistEmptySeqs[String, Boolean :: HNil, Seq[Boolean] :: HNil](
  hlistEmptySeqs[Boolean, HNil, HNil](hnilEmptySeqs))))
    :EmptySeqs[String :: Boolean :: HNil, Seq[String] :: Seq[Boolean] :: HNil]
                                              ~~~~
```

and identify the corresponding part of the implicit value definition

```
Type parameter T has been instantiated using location(s):
    val x = emptySeqs(1 :: "abc" :: true :: HNil)
                            ~~~~~
Explicit type argument for type parameter T that was in conflict:
    foo[Seq[Int] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
                         ~~~~~~~
```

Figure 7.5: Improved error feedback for the erroneous application in Listing 7.1

```
implicit def hlistEmptySeqs[H, T <: HList, POut <: HList](
  implicit est : EmptySeqs[T, POut]):
    EmptySeqs[H :: T, Seq[H] :: POut] = // ...
                          ~
```

While correct, the locations are a far cry from the information desired by users, such as the one in Figure 7.5, which describes exactly the conflicting type arguments and values. To realize the improved error feedback, we first notice that the result of the shallow analysis represents a variant of the Prototype Typing Slice, and, from the formalization in Section 3.6, it translates to an equivalent `TypeFocus` value. To make such a statement clearer, in our example we compare side-by-side the type elements extracted by the `TypeFocus` selection in

- The declared result type of the same implicit value,
  i.e., `EmptySeqs[H :: T, Seq[ H ] :: POut]`,
- The part of the inferred type of the implicit value, identified through the shallow analysis, *i.e.,*
  `EmptySeqs[String :: Boolean :: HNil, Seq[ String ] :: Seq[Boolean] :: HNil]`, and
- The expected type of the implicit search,
  i.e., `EmptySeqs[String :: Boolean :: HNil, ? ]`, where `?` represents the unconstrained type.

The `TypeFocus` value allows us to represent the type propagation information, specific to the unique local type parameter, *i.e.,* `H` in `EmptySeqs[ H :: T, Seq[ H ] :: POut]`. For our example, the type propagation is equivalent to a `TypeFocus` value of ‘`TypeArgTFocus(0, ‘::’) compose TypeArgTFocus(0, ‘EmptySeqs’)`’, highlighted through the dark gray region and denoted as Θ_{I2}.

In essence, the main difference with the original prototype propagation technique from Section 3.6 and its translation to `TypeFocus` values lies in the source of the type propagation - there we translated the *Prototype Propagation Path*, while here we operate at a more fine-grained level of the inferred types of the implicit values, the type signatures of their definitions and types of formal implicit parameters.

The reconstructed `TypeFocus` value encapsulates sufficient information to perform a controlled backtracking within the implicit context it was decided in. In the case of our example, the context is illustrated as the grayed-out fragment of the full implicit argument in

```
hlistEmptySeqs[Int, String :: Boolean :: HNil, Seq[String] :: Seq[Boolean] :: HNil](
  hlistEmptySeqs[String, Boolean ::  HNil, Seq[Boolean] ::  HNil](
    hlistEmptySeqs[Boolean, HNil, HNil](hnilEmptySeqs)))
```

and the definition of the parent implicit

```
implicit def hlistEmptySeqs[H, T <: HList, POut <: HList](
  implicit est :  EmptySeqs[T, POut]): EmptySeqs[H :: T, Seq[H] :: POut] = // ...
```

where the grayed-out part represents the parameter of the previously considered implicit value. The inferred Θ_{l2} `TypeFocus` encapsulates the type propagation for the expected type of the implicit value l_2. At the same time the Θ_{l2} `TypeFocus` is well-formed with respect to the inferred type of the implicit value l_2 and the type of formal implicit parameter, `EmptySeqs[T, POut]`. The application of `TypeFOcus` to type `EmptySeqs[T, POut]` results in a partial type selection with

- The extracted local type parameter `T` of the `hlistEmptySeqs` method.
- The `TypeArgTFocus(0, '::') TypeFocus`, denoted as the $\Theta_{l3'}$ type selection, that could not be applied to the local `T` type parameter.

The significance of such type extraction are three-fold:

- We have identified that the instantiation of the local type parameter `T`, defined in the function `hlistEmptySeqs`, is the source of the type propagation.
- The partial type selection $\Theta_{l3'}$ is well-formed with respect to the instantiation of the local type parameter `T`.
- Similarly, as in the case of the implicit value l_2 we can represent the type propagation that instantiated type parameter `T` through a `TypeFocus` type selection on the return type of the inferred l_1 implicit value: $\Theta_{l3'}$ `compose TypeArgTFocus(1, '::') compose TypeArgTFocus(0, 'EmptySeqs')`.

Through the interleaving backtracking and application of the `TypeFocus` extraction, the analysis will eventually reach the non-implicit function application and the inferred `TypeFocus` value will represent a transformation from the initial type selection on the inferred type of the complete function application

```
emptySeqs(1 :: "abc" :: true :: HNil) :Seq[Int] :: Seq[ String ] :: Seq[Boolean] :: HNil}
```

to a (partial) type selection on the type of the method with the implicit parameters

```
def emptySeqs[T <: HList, OutT <: HList](x: T)(implicit es : EmptySeqs[ T , OutT]): OutT
```

The *TypeFocus*-based analysis of the extracted type parameter `T` (of the `emptySeqs` function) leads to a regular analysis of the type constraints (as formalized in Section 3.7) and will infer the appropriate type selection on the inferred type of the argument, *i.e.*,

```
(1 :: "abc" :: true :: HNil) : Int :: String :: Boolean :: HNil
```

This, in turn, provides sufficient information to trigger the regular *TypeFocus*-based analysis on the argument and blame the `"abc"` literal as the source of the type mismatch conflict, as we have shown in the improved error feedback in Figure 7.5.

Debugging alternative type parameter instantiations in implicit resolution

The local type parameters in the presented example were all instantiated through the type propagation from the expected type of the implicit search. In general, the local type parameters can also be instantiated through the usual means of type constraints or type propagation from the inferred types of the implicit arguments of the same implicit parameter list. For example, in the definition of the implicit `combine` value in

```
class TConst[A, B]; class Concat[A, B, C]; class Contained[A, B, C]

implicit def combine[A <: HList, B <: HList, T <: HList, S <: HLIst, Out <: HList](
 implicit xs: TConst[A, T], ys: TConst[B, S], zs: Concat[T, S, Out]): Container[A, B, Out]
```

the analysis of the source of the instantiation of the type parameter `Out` depends on the type arguments `T` and `S` in the type of the parameter `zs`, for some type constructor `Concat`. Since the type parameters are not present in the return type `Container[A, B, Out]`, their analysis would not apply to our previous algorithm. Instead they are instantiated from the implicit arguments of parameters `xs` and `ys` (the inference of the implicit arguments is sequential). Knowing the types of the formal parameters, one can trivially construct `TypeFocus` values that

```
val x = emptySeqs(1 :: "abc" :: true :: HNil)

foo[Seq[Int] :: Boolean :: Seq[Boolean] :: HNil](x)  // error
```

(a) A variant on the type mismatch example from Listing 7.1.

```
Type parameter T has been instantiated using locations:
  implicit def hlistEmptySeqs[H, T <: HList, POut <: HList](
    implicit est : EmptySeqs[T, POut]):
     EmptySeqs[H :: T, Seq[H] :: POut] = // ...
                       ~~~~~~
Explicit type argument for type parameter T that was in conflict:
    foo[Seq[Int] :: Boolean :: Seq[Boolean] :: HNil](x)
                    ~~~~~~~
```

(b) A result of an implicit-specific `TypeFocus`-based analysis of the error in Listing 7.6a.

Figure 7.6: An example of the failed attempt at explaining the implicit resolution using the *deterministic* `TypeFocus`-based approach.

extract the instantiation of the desired type parameter, *e.g.,* '`TypeArgTFocus(1, 'TConst')`' for the type selection of type parameter `S` from type `TConst[B, S]`, and guide the analysis of the corresponding implicit argument. In short, the analysis has to take into account different mechanisms that may have been used to instantiate the selected type parameters, but in our type debugging framework all can be expressed through a unifying `TypeFocus` abstraction.

Limitations

The *deterministic* analysis of the inferred implicit arguments is guided by the continuous type selection on the local type parameters of the implicit values. In general, one cannot guarantee that the inferred `TypeFocus` value, when applied to the formal type of the definition of the implicit value, extracts only local type parameters.

We illustrate the problem in Listing 7.6a, where we consider a function application involving the `emptySeqs` method, identical to our motivating example from Listing 7.1, except for the different type argument (grayed-out part of the source code); rather than expecting a type argument of type `Seq[Booolean]` the example uses type `Boolean`, which will be mirrored in the failed subtyping derivation tree, and in the value of `TypeFocus` inferred from it. The modified type selection information translates directly to the reduction in the precision of the analysis, as shown in Figure 7.6b. There, the type selection extracts a complete type application, `Seq[H]`, rather than an individual type parameter.

In the next section we consider a variant of the *deterministic* technique which approximates the results of the analysis when the above conditions are encountered.

```
val x = emptySeqs(1 :: "abc" :: true :: HNil)
foo[Seq[Int] :: Seq[String] :: HNil](x)
  // type mismatch;
  // found   : shapeless.::[Seq[Int],shapeless.::[Seq[String],
  //             shapeless.::[Seq[Boolean], shapeless.HNil]]]
  // required: shapeless.::[Seq[Int],shapeless.::[Seq[String],shapeless.HNil]]
```

(a) The invalid construction of the heterogeneous list.

```
Type parameter T has been instantiated using location(s):
  implicit def hlistEmptySeqs[H, T <: HList, POut <: HList](
    implicit est : EmptySeqs[T, POut]): EmptySeqs[H :: T, Seq[H] :: POut] = // ...
                                                          ~~~~~~~~~~~~~~
Explicit type argument for type parameter T that was in conflict:
  foo[Seq[Int] :: Seq[String] :: HNil](x)
                                  ~~~~
```

(b) The result of the deterministic analysis

Figure 7.7: A variation of the heterogeneous list construction example from Listing 7.1.

7.1.2 The *heuristic-based* analysis of the implicit resolution

The *deterministic* `TypeFocus`-based analysis delivers accurate explanations for the implicit resolution, when its requirements are satisfied. We propose a variant of the *deterministic* analysis that reverts to the heuristic-based approach in situations when the source of the target type does not involve directly the instantiation of a single local type parameter. The heuristics provide less guarantees with respect to the program locations explaining the source of the target type but still they ensure that a link to the non-implicit-related type element will be identified, if possible

To elaborate on the possible heuristics, in Listing 7.7a we consider a variation of our motivating example with heterogeneous lists, where the inferred type, and the encoded size, of the heterogeneous list is larger than the expected one. Neither the accompanying type error message, nor the *deterministic* analysis (presented in Figure 7.7b), delivers satisfying information by only revealing the internal details of the encoding. In the following discussion we consider two straightforward heuristics that approximate the dependencies between the implicit arguments and the implicit parameters they are part of.

Dependencies from nested type parameters

The deterministic analysis of the function applications in Listings 7.6a and 7.7a extracted parts of the implicit value type signature, `Seq[H]` and `Seq[H] :: POut`, respectively. Rather than stopping the analysis, we continue it individually for each of the type parameters nested within the type application, starting with an identity `TypeFocus` value, *i.e.,* with no prior type selection on their instantiations or the context in which they appear. The approach allows

```
Type parameter T has been instantiated
using location(s):
val x=emptySeqs(1 :: "abc" :: true :: HNil)
                              ~~~~~   ~~~~
```

(a) Result of an analysis with heuristics

```
Type parameter T has been instantiated
using location(s):
val x=emptySeqs(1 :: "abc" :: true :: HNil)
                              ~~~~~~~~~~~~
```

(b) Result of an analysis with a perfect knowledge

Figure 7.8: A comparison of the analysis of the function application from Listing 7.7a when a heuristic-based approach is used and when we have a *perfect knowledge* assumption (the feedback tailored to the specific example).

```
val x = emptySeqs(1 :: "abc" :: true :: HNil)

foo[Seq[Int] :: Seq[String] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
 // type mismatch;
 // found   : shapeless.::[Seq[Int],shapeless.::[Seq[String],shapeless.::[Seq[Boolean],
 //            shapeless.HNil]]]
 // required: shapeless.::[Seq[Int],shapeless.::[Seq[String],shapeless.::[Seq[Boolean],
 //            shapeless.::[Seq[Boolean],shapeless.HNil]]]]
```

Figure 7.9: Variation on the example from Listing 7.1 that cannot be explained with a precise `TypeFocus`-based analysis.

us to resort to a plain, deterministic analysis that may potentially find a link with the non-implicit arguments or the non-implicit part of the function definition.
For example, a fusion of the results of the deterministic analysis for the individual type parameters `H` and `POut` yields a type error message in Figure 7.8.

Non-parametric types as dependencies

The inferred `TypeFocus` value does not guarantee a type selection on a type with some nested type parameters. To illustrate, we consider a different example of the application of the `emptySeqs` method in Listing 7.9, where the inferred type of the heterogeneous list of empty sequences is smaller than the expected one. The example is accompanied by a confusing type mismatch error produced by the Scala compiler.

Figure 7.10a illustrates the result of the deterministic analysis, where the failed subtyping derivation provides sufficient information only to identify the implicit value `hnilEmptySeqs` and part of its inferred type, `EmptySeqs[HNil,` $HNil$ `]`, as the source of the conflict (we recall that the `hnilEmptySeqs` corresponds to the l_4 implicit value in our type propagation summary in Figure 7.4). The part of the inferred type does not select any local type parameters from the formal type of the implicit value and we are unable to continue the *TypeFocus*-based analysis even with the previously proposed heuristic.

```
Type parameter T has been instantiated using location(s):
  implicit def hnilEmptySeqs: EmptySeqs[HNil, HNil] = // ...
                                             ~~~~
Explicit type argument for type parameter T that was in conflict:
  foo[Seq[Int] :: Seq[String] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
                                                ~~~~~~~~~~~~~~~~~~~~
```

(a) A result of a deterministic implicit-specific `TypeFocus`-based analysis of the error.

```
Type parameter T has been instantiated using location(s):
val x = emptySeqs(1 :: "abc" :: true :: HNil)
                                       ~~~~
Explicit type argument for type parameter T that was in conflict:
    foo[Seq[Int] :: Seq[String] :: Seq[Boolean] :: Seq[Boolean] :: HNil](x)
                                                  ~~~~~~~~~~~~~~~~~~~~
```

(b) A desired outcome of the `TypeFocus`-based analysis of the implicit values.

Figure 7.10: A comparison of a deterministic and a heuristic-based analysis of the implicit resolution for Listing 7.7a.

To solve the problem, we treat the type of the whole implicit argument as a dependency on the type of the formal parameter it is part of. In the case of our example, this implies that the type of the argument `EmptySeqs[HNil, HNil]` is dependent upon the implicit parameter `est` of the `hlistEmptySeqs` implicit value and its formal type, `EmptySeqs[T, POut]`. Knowing that the implicit value l_4 has been inferred with the expected type `EmptySeqs[HNil, ?]`, and the corresponding type of the formal parameter contains a local type parameter `T`, we can trigger the *deterministic* `TypeFocus`-based analysis at the parent of the initial implicit value `hnilEmptySeqs`, *i.e.,* at the l_3 implicit value, with the type selection on the local type parameter `T`.

The results of the heuristic-based approach are dependent upon the particular encoding of the implicit values. In the worst case scenario, when the formal parameter of the implicit value has no local type parameters, the heuristics can again identify a complete type of the implicit value with the identity `TypeFocus` as the source of the target type. In practice we found that the results of the heuristic-based approach are typically as encouraging and complete as the one presented in Listing 7.10b.

Domain-specific heuristics

The *deterministic* and the heuristic-based analysis of the implicit resolution, and its results, are exposed in the type debugging framework. Combined with the `TypingSlice` values that identify the inference of function applications involving implicit parameters, we give the power to users of the framework to further improve on our results and provide solutions that are close to, or equivalent to, perfect-knowledge scenarios.

Depending on the complexity of the encoding of the implicit values, even the domain-specific solutions may not be an ideal solution to explaining function applications involving the implicit resolution process. On such occasions only the more time-consuming interactive approach to type debugging is likely to be the only option (Section 7.3).

7.2 Improved error feedback

In this section we turn our attention to the main application of the *TypeFocus*-based analysis - generating succinct and precise type error messages. We do not define a guide to generating them, a comprehensive study has been the subject of for example Heeren [2005]; rather we aim to show a variety of scenarios, not only involving the type mismatch errors, where our approach would significantly improve the rather disappointing status quo. We dismiss the detailed technical discussion, in favor of practical applications that illustrate the advantages and limitations of both, the short messages and our *TypeFocus*-based analysis.

7.2.1 Examples

Explaining member selection

When it comes type debugging, the role of member selection in the type checking process has been silently omitted in the related work (El Boustani and Hage [2010], Chen and Erwig [2014a], Pavlinovic et al. [2014]). The disclosure is surprising, because with an increasing number of the newly introduced languages such as Dart (https://www.dartlang.org/), Kotlin (http://kotlinlang.org/), or Rust (http://www.rust-lang.org/), we see a visible trend towards introducing parametric polymorphism in object-oriented languages, and subsequently the necessity to deal with non-local type parameters, *i.e.,* the type parameters of class and trait definitions. The universality of `TypeFocus` allows us to represent the instantiation of non-local type parameters, while navigating the type derivation tree (Section 6.3.2). To illustrate the importance and the precision of our approach, we will now examine a non-trivial example involving not only the type checking of member selection but also their integration with the implicit resolution mechanism or path-dependent types.

Listing 7.11 defines a synthetically constructed class hierarchy that will allow us to exhibit the essence of some of the complex type errors linked to member selection terms, that are typically encountered in programming forums and mailing lists. The definitions of classes `Foo`, `Bar` and `TConst` illustrate the increasing trend in the generic programming libraries that mix the concepts of parametric polymorphism, abstract type members and the path-dependent types in order to define reusable abstractions (Odersky and Zenger [2005]).

The key elements of the class hierarchy are:

```
1  class Foo[A, B](x: A, y: B) {
2    def test[C](a: A, y: B, c: C): Int = // ...
3  }
4
5  class Bar[A1, B1](x: A1, y: B1) {
6    type T1 = TConst[Int, B1]
7    type T2 = TConst[A1, B1]
8    type T3
9  }
10 object Bar {
11   def create[A2, B2](x: A2, y: B2): Bar[A2, B2] { type T3 = TConst[Int, B2] } = // ...
12   def create[A2, B2](x: A2, y: B2, z: Int): Bar[A2, B2] = // ...
13 }
14
15 class TConst[+A, +B]
16 object TConst{
17   def apply[A, B](x: A, y: B): TConst[A, B] = // ...
18 }
```

Figure 7.11: An example of a class hierarchy that mixes the concepts of abstract type members and parametric polymorphism.

1. The `Foo` class (lines 1-3) defines non-local type parameters, `A` and `B`, as well as local type parameters `C` for its `test` member.
2. The `Bar` class (lines 5-9) defines non-local type parameters, as well as partially instantiated type aliases, `T1` and `T2`, and abstract type member, `T3`. The type declarations combine different scenarios that a user might have to be able to understand.
   The `Bar` values are constructed using the overloaded `create` methods, that are defined in its companion object; the first of the overloaded methods returns a value with a refined `Bar` type, where the abstract type member `T3` is explicitly instantiated (line 11), while in the second case (line 12) the type of the type member is not constrained.
3. The `TConst` class (line 15) defines two non-local type parameters, and is constructed with a polymorphic `apply` method in its companion object.

The example highlights the mixture of Scala's type system features. On its own, the definitions do not pose particular problems in visually tracing the type parameters. The problem of meaningless error messages becomes only apparent at the usage site.

The code snippet from Listing 7.12 illustrates one of the possible applications that has gone unexpectedly wrong. In the example the inferred type of the `x` value (line 4) does not have a member `test`, meaning that the member selection in line 5 has to be implicitly translated to `toFoo(x).test` member selection, in a type-driven implicit resolution. While common, the adaptation of the qualifier introduces another implicit layer of abstraction that has to be understood by the programmers in order to discover the nature of the problem.

```
1   implicit def toFoo[X1, X2](f: Bar[X1, X2]): Foo[f.T3, X2] = // ...
2
3   def test {
4     val x = Bar.create(1,"abc")
5     x.test(TConst("abc", 2), "def", 2)
6   }
7    // type mismatch;
8    //  found   : String("abc")
9    //  required: Int
10   //     x.test(TConst("abc", 2), "def", 2)
11   //                   ^
```

(a) The invalid application of the function to the arguments and the succinct error message from the Scala compiler

```
1   // Specialized error message:
2   // Expected type comes from the inferred type of the definition:
3   //     val x = Bar.create(1,"abc")
4   //     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
5   // The conflicting part of the inferred type of the definition is:
6   // Bar[Int,String]{type T3 = TConst[Int,String]}
7   //                                  ~~~
8   //
9   // Part of the conflicting expression that leads to an error:
10  //  x.test(TConst("abc", 2), "def", 2)
11  //               ~~~~~
12  // Locations affecting the inference of the part of the definition.
13  // Location (1):
14  //  def create[A2, B2](x: A2, y: B2): Bar[A2, B2] { type T3 = TConst[Int, B2] } =// ...
15  //                                                                   ~~~
16  //   val x = Bar.create(1,"abc")
17  //               ~~~~~~
18  //   x.test(TConst("abc", 2), "def", 2)
19  //   ~
```

(b) The improved error feedback

Figure 7.12: A member selection example involving the class hierarchy from Listing 7.11.

Without going into details of the application, we notice that:

1. The type mismatch error (lines 7-12) identified the location where the two types conflict. The error message lacks information about the source of the expected and the source of the inferred type of the argument; in cases like this, it is not uncommon that many of the values have similar types, such as `Int` and `String`, making them hard to distinguish visually.

2. The improved error feedback (lines 1-19) locates the source of both of the conflicting types, down to the level of the primitive values (lines 10-11) and type annotations (lines 14-15).

3. The analysis had to take into account the decisions of the implicit resolution mechanism (Section 7.1), which allowed it to extract the part of the type signature of the `create` method (lines 14-15). Lack of the analysis of the decisions of the implicit resolution would only blame the application of the `toFoo` implicit value in `toFoo(x)`.

4. The `TypeFocus` value inferred from the adapted `toFoo(x)` function application had to also take into account the path-dependent type `f.T3` (line 1) that is defined in the return type of the implicit function `toFoo`.

5. We display the intermediate non-implicit results of the *TypeFocus*-based analysis in order to help with the orientation of the locates types and values (lines 12-19).

Due to the verbosity of the error, we postulate that the improved error feedback must not be enabled by default. Rather, the comprehensive report of the problem should only be provided on demand.

With the code snippet from Listing 7.13 we illustrate how subtle type signature changes in the implicit values and function applications can lead to equivalent type mismatch errors. For example, the return type of the implicit value (line 1) now involves a different path-dependent type, `f.T1`, and we invoke a different overloaded method `create` to create an instance of `Bar` (line 4). Our *TypeFocus*-based analysis is able to reflect such level of detail, because we infer type selectors for type and value members (the `TypeMemberTFocus` class in Listing 6.6) and overloaded method types (the `OverloadTFocus` class in Listing 6.6). Subsequently, without any user input, we are able to deliver comprehensive error reports, such as the one in Listing 7.13.

In comparison to the previous example, we:

1. Identify the necessary details of the inferred type of the local value `x`, by listing not only its type (lines 6-7), but also the conflicting parts of its type member which is a type alias (lines 8-10).

2. Illustrate the complete path that inferred the conflicting expected type `Int` (lines 27-35).

3. Infer an opportunistic source code modification (lines 25-26) that will fix the type mismatch, based on the precise localization of the source of the error.

```
implicit def toFoo[X1, X2](f: Bar[X1, X2]): Foo[f.T1, X2] = // ...

def test {
  val x = Bar.create(1,"abc", 2)
  x.test(TConst("abc", 2), "def", 2)
}
// type mismatch;
//  found   : String("abc")
//  required: Int
//     x.test(TConst("abc", 2), "def", 2)
//                   ^
```

(a) The invalid application of the function to the arguments and the succinct error message from the Scala compiler

```
// Specialized error message:
// Expected type comes from the inferred type of the definition:
//     val x = Bar.create(1,"abc", 2)
//     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The conflicting part of the inferred type of the definition is:
// Bar[Int,String]
// ~~~~~~~~~~~~~~~
// (in type member T1):
// TConst[Int,String]
//        ~~~
//
// Part of the conflicting expression that leads to an error:
//     x.test(TConst("abc", 2), "def", 2)
//                   ~~~~~
// Locations affecting the inference of the part of the definition:
// Location (1):
//   type T1 = TConst[Int, B1]
//                    ~~~
//   def create[A2, B2](x: A2, y: B2, z: Int): Bar[A2, B2] = // ..
//                                             ~~~
//     val x = Bar.create(1,"abc", 2)
//                 ~~~~~~
//     x.test(TConst("abc", 2), "def", 2)
//     ~
// You may try to modify the existing type
// from Int to Any
```

(b) The improved error feedback

Figure 7.13: A member selection example involving the class hierarchy from Listing 7.11.

```
1  abstract class ImplParam[+A, +B] { type C }
2
3  implicit def genImpl[A, B]: ImplParam[Int, B] { type C = Int } = // ...
4
5  implicit def toFoo[X1, X2](f: Bar[X1, X2])(
6                 implicit p: ImplParam[X1, X2]): Foo[p.C, X2] = // ...
7
8  def test {
9    val x = Bar.create(1,"abc")
10   x.test("abc", "def", 2)
11 }
12
13 // type mismatch;
14 //  found   : String("abc")
15 //  required: Int
16 //     x.test("abc", "def", 2)
17 //            ^
```

(a) The invalid application of the function to the arguments involving a number of implicit values and the succinct error message from the Scala compiler

```
// Specialized error message:
// Part of the conflicting expression that leads to an error:
//     x.test("abc", "def", 2)
//            ~~~~~
// Locations affecting the inference of the part of the definition.
// Location (1):
//  implicit def genImpl[A, B]: ImplParam[Int, B] { type C = Int } = // ...
//                                                  ~~~~~~~~~~~~
//  implicit def toFoo[X1, X2](f: Bar[X1, X2])(
//               ~~~~~
//  x.test("abc", "def", 2)
//  ~
```

(b) The improved error feedback

Figure 7.14: A member selection example involving the class hierarchy from Listing 7.11.

Finally, it is common for the type errors linked to member selection to originate from the implicit values that adapted the type of the qualifier. Listing 7.14 defines another variation involving the class hierarchy from Listing 7.11 and a function application (line 10) that results in a type mismatch with the same type error message as before (lines 13-17). Similarly to the previous examples, the inferred type of the qualifier `x` has no member `test` and will be implicitly adapted using the implicit values to `toFoo(x)(genImpl)`. This in turn means that the *TypeFocus*-based analysis not only has to explain the role of the used implicit view, `toFoo`, but also analyze the implicit resolution involving chains of implicits because of its implicit parameter `p`.

The analysis of the implicit resolution takes into account the implicit argument (`genImpl`) of the implicit view (`toFoo`) because the type extraction of the `TypeFocus` value (reconstructed

```
def test1[T <: Number](a: List[T], b: List[_ >: T]) = // ...
val x: List[_ >: Integer] = // ...
test1(x, x)

// inferred type arguments [_$2] do not conform to
//  method test1's type parameter bounds [T <: Number]
//    test1(x, x)
//    ^
// type mismatch;
//  found   : List[_$2] where type _$2 >: Integer
//  required: List[T]
//     test1(x, x)
//           ^
```

```
def test2[T <: Number, S](a: Map[T,S], b: List[_ <: S]): List[T] = // ...
val x: Map[String, String] = // ...
val y: List[_ >: Number] = // ...
val z: List[Integer] = test2(x, y)

// inferred type arguments [String,Any] do not conform to
//  method test2's type parameter bounds [T <: Number,S]
//     val z: List[Integer] = test2(x, y)
//                            ^
// type mismatch;
//  found   : Map[String,String]
//  required: Map[T,S]
//     val z: List[Integer] = test2(x, y)
//                                  ^
// type mismatch;
//  found   : List[_$2] where type _$2 >: Number
//  required: List[_ <: S]
//     val z: List[Integer] = test2(x, y)
//                                     ^
// type mismatch;
//  found   : List[T]
//  required: List[Integer]
//     val z: List[Integer] = test2(x, y)
//                                 ^
```

Figure 7.15: Examples of the inferred type arguments that do not conform to the declared formal bounds of the type parameters. The examples were translated to Scala from El Boustani and Hage [2010].

as part of navigation process) discloses a link between the part of the inferred type of the adapted qualifier `toFoo(x)(genImpl)`, `Foo[ Int , String]`, and the formal return type of the implicit value `toFoo`, `Foo[ p.C , X2]`.

The improved error feedback aims to provide a comprehensive analysis of the conflict by listing elements of the path that inferred the conflicting types. With the `TypeFocus` abstraction encapsulating the local type debugging information while stepping through the analyzed terms, we ensure that the analysis considers only the typing decisions that directly or indi-

```
Specialized error message:
Inferred type argument (_$2) for type parameter T
does not conform to the type parameter’s bound (<: Number).
Note the origin of type constraints that affected the inference:
    val x: List[_ >: Integer] = // ...
                            ~
    test1(x, x)
          ~

Specialized error message:
Inferred type argument (String) for type parameter T
does not conform to the type parameter’s bound (<: Number).
Note the origin of type constraints that affected the inference:
    val x: Map[String, String] = // ...
               ~~~~~~
    val z: List[Integer] = test2(x, y)
                                 ~
```

Figure 7.16: Examples of the improved error feedback revealing the type constraints that led to the inferred type arguments in Listing 7.15.

rectly affected the inference of the target type.

Explaining inference of the elided type arguments

Our examples of type debugging scenarios have covered a number of functions applications where we were able to precisely investigate the type constraints behind the inferred type arguments. For practical reasons languages like Java or Scala, do not always attempt to infer optimal solutions. As a result, the inferred instantiations are always verified against the formal bounds of type parameters which may lead to incomprehensible type errors.

We illustrate the problem of suboptimal type parameter instantiations that do not conform to the formal type bounds with two examples in Listing 7.15. The examples not only provide vague information about the inferred type arguments (lines 5-8 and lines 6-9, respectively) but also add to the confusion by reporting incorrect type errors with the uninstantiated type parameters. The type errors do not provide information about which type arguments were in conflict, how were they inferred, or, more importantly, whether there exists an explicit type argument that would resolve the error.

The instrumentation of the type checking process covers not only the collection of type constraints, but also the verification of the correctness of the inferred type arguments, or lack thereof. With such high-level representation in place, we are capable of identifying the culprit type arguments and type parameters that failed to conform to their bounds, and locate the type constraints that led to the type error, as illustrated in Listing 7.16.

```
abstract class Activity[T] {
  def apply (): T
}
object Count extends Activity[Int] {
  def apply (): Int = 1
}
def run [T, A <: Activity[T]](activity: A) = ()

def test {
  run(Count)
}
// inferred type arguments [Nothing,Count.type] do not conform to
//  method run’s type parameter bounds [T,A <: Activity[T]]
//  run(Count)
//     ^
// type mismatch;
//  found   : Count.type
//  required: A
//  run(Count)
//      ^
```

(a) A single type parameter `T` that appears in a higher-kinded position

```
def flattenBySum[U <: Iterable[T], T : Numeric](listOfLists: Iterable[U]) =
  for (list <- listOfLists) yield list.sum

def test {
  flattenBySum(Vector(List(1,2,3), List(10,11,12)))
}

// inferred type arguments [List[Int],Nothing] do not conform to
//  method flattenBySum’s type parameter bounds [U <: Iterable[T],T]
//     flattenBySum(Vector(List(1,2,3), List(10,11,12)))
//     ^
// type mismatch;
//  found   : scala.collection.immutable.Vector[List[Int]]
//  required: Iterable[U]
//     flattenBySum(Vector(List(1,2,3), List(10,11,12)))
//                        ^
// could not find implicit value for evidence parameter of type Numeric[T]
//     flattenBySum(Vector(List(1,2,3), List(10,11,12)))
//                 ^
```

(b) Flattening of collections example

Figure 7.17: Examples of limitations of type inference for type parameters in higher-kinded position.

Our feedback refrains from providing source code modifications that alleviate the conflicts, simply to avoid reporting false positives. Given that our high-level representation reveals not only the type constraints but also the complete verification process, nothing stops us from developing heuristics that would attempt to correct even such type errors.

```
Specialized error message:
The current type signature of method run
(of type [T, A <: Activity[T]](activity: A): Unit)
limits the ability to infer an appropriate type argument for the type parameter T.
The inferred type argument, Count.type, is not within the upper bound Activity[Nothing].
In order to track appropriately the constraints for the type parameter T
you can modify the type signature to:
def run[T, A[_] <: Activity[_]](activity: A[T]): Unit

Specialized error message:
The current type signature of method flattenBySum
(of type [U <: Iterable[T],T: Numeric](listOfLists: Iterable[U]):Iterable[T])
limits the ability to infer an appropriate type argument for the type parameter T.
The inferred type argument, List[Int], is not within the upper bound Iterable[Nothing].
In order to track appropriately the constraints for the type parameter T
you can modify the type signature to:
def flattenBySum[U[T] <: Iterable[T],T: Numeric](listOfLists: Iterable[U[T]]):Iterable[T]
```

Figure 7.18: The generated improved error feedback for the examples from Listing 7.17 that proposes source code modifications while preserving the intended semantics.

In general, developing high-confidence source code modifications is not an easy task, even with a complete knowledge that type derivation trees provide (we chose not to take the approach of El Boustani and Hage [2010] since many of the examples would still lead to type errors). This is however different when we deal with known language implementation limitations that affect the type inference process. As part of the evaluation of our type debugging tool we have developed heuristics that specifically target such problems. We illustrate the results using the code snippets from Listing 7.17, where the inferred type arguments limit the programmers' expressiveness.

In the first example, we define a generic `Activity` class and a subclass of it (`Count`) that provides an explicit type argument `Int`. The example also defines a `run` function with two local type parameters; the intention of the declared bound on the type parameter `A` is to collect type constraints sufficient to instantiate both of the type parameters in the function application. It is therefore surprising to the programmer that the function application in line 10 is being rejected by the type checker; after all the base type of the `Count` type is `Activity`, meaning that `T` should be instantiated to `Int` as well.
In the second example, the type signature of the `flattenBySum` function encodes the flattening of a collection of collections to another collection. The function reduces the individual collections to integer values by summing their elements; the latter requires the *sum* operation to be available for the individual generic elements of the collection, as expressed through the `T: Numeric` context bound. It may therefore be surprising to the programmer that the type checker fails to infer valid type arguments from the simple argument which is a collection of two integer integer lists.

```
val o1: Ordered[Int] = 1
val o2: Ordered[Int] = 2
println(o1 < o2)
// diverging implicit expansion for type scala.math.Ordering[Ordered[Int]]
// starting with method comparatorToOrdering in trait LowPriorityOrderingImplicits
// println(o1 < o2)
//         ^
```

Figure 7.19: An example of a comparison between two values.

The two examples illustrate a problem that appears regularly on the programming forums and mailing lists. What is more surprising, the examples encode the functionality in a correct way, except that, due to the limitation of the implementation, the type inference fails to infer type constraints for type parameters that appear in a higher-kinded position in the bounds of other type parameters. In other words, the inferencer collects no type constraints for the type parameter `T` in both of the examples, and selects a maximal solution instead, *i.e.,* the Scala's bottom type `Nothing`. The limitations of the design pattern have been known to the Scala experts for a long time, but it is unclear if and how any progress will be made for the problem in the future.

Rather than improving the implementation of the algorithm, which inadvertently may lead to new bugs, we make use of the fact that limitations like above can be precisely located by pattern matching on the nodes of the high-level representation. The link between the high- and low-level representations is sufficient to navigate not only to the appropriate high-level goals, but also inspect their low-level data, such as the types of formal parameters of the functions. With such information in place, we can provide more elaborate error messages and propose type-safe modifications of type signatures that convey the same semantics of the functions, as presented in Figure 7.18.

Explaining the diverging implicits

In Section 6.6 we have shown how the exposed decision process of the implicit resolution allows us to provide better feedback for the ambiguous implicit values. The representation allows us also to display chains of implicit values that fail abruptly with no implicit argument matching the expected type, or when an infinite expansion of the same implicit values is encountered (diverging implicit values).

In both cases the improved error feedback can at least inform about the implicit values that were attempted as part of the implicit resolution process. To illustrate, in Listing 7.19 we consider an innocuous example of two values of type `Ordered` that we want to compare and print the result. When compared with a *less than* operation, the infix operation produces an alarming error message that is not very revealing - the internal details, which in this particular

```
Full implicit argument(s) expansion that led to the divergence is:
math.this.Ordered.orderingToOrdered[Ordered[Int]](o1)(
  [comparatorToOrdering(*no-implicit-arg-of-type: java.util.Comparator[Ordered[Int]]*)|
   ordered((x: Ordered[Int]) =>
     orderingToOrdered(x)(*diverging-implicits-(comparatorToOrdering/ordered)*))])
```

Figure 7.20: An example of a simplified diverging implicit resolution decision process for Listing 7.19. We use the `[...|...|...]` notation to list all the eligible implicit values that have been tried and failed.

case would be useful, are scarce.
The analysis of the implicit resolution process delivers a comprehensive description of the tried attempts of the implicit resolution in Listing 7.20.

In general, the sheer number of available implicit values makes it hard to provide error reports that list all available implicits; an interactive approach that can incrementally list different scopes and the types of the implicit values is much better suited for that purpose.

7.2.2 Library-specific plugins

The type selection inferred from the type mismatch conflicts, and the high-level representation of the type checking process grant the ability to generate error feedback that potentially improves the error for a generic type error message. While correct, it does not provide any means to incorporate domain-specific knowledge that is aware of the context of the error. For example, in a situation where a complete implicit value derivation is still unsatisfactory, such as in the case of the diverging implicit value in Listing 7.20, we could recognize the particular context of the divergence involving the low-level types of `Ordered` and generate instead an error message such as

```
    // type mismatch;
    //  found   : Ordered[Int]
    //  required: Int
    // println(o1 < o2)
    //              ^
```

In this section we propose a convenient specialization of the type debugging process, in the form of the plugins for the type debugging framework. To illustrate the application of domain-specific knowledge to the exposed high-level type derivation trees, we discuss a few examples inspired by problems reported by Scala users.

```
List(1, 2, 3)(1)                 // ok
(List(1, 2, 3) map identity)(1)   // error

// type mismatch;
//  found   : Int(1)
//  required: CanBuildFrom[List[Int],Int,?]
//   (List(1, 2, 3) map identity)(1)
//                               ^
```

(a) A type mismatch revealing the internal details of the Scala collections

```
Specialized error message:
Full type signature of the defined ‘map’ method is
[B, That](f: A => B)(implicit bf: CanBuildFrom[This,B,That]): That
Argument ‘1’ is used for the parameter ‘bf’.
Did you mean
‘List(1, 2, 3).map(identity).apply(1)’
which compiles?
```

(b) A Scala collections-specific error message

Figure 7.21: An example of an erroneous function application involving the Scala collection library, and a customized error feedback, specific to the type error.

Examples

Explaing Standard Library errors

The Scala collection library uses an implicit resolution mechanism in order to avoid code duplication and provide a more intuitive API for the users (Odersky and Moors [2009]). The implicit parts of type signatures of the methods, which are hard to understand for a regular programmer, have long been perceived as a blessing and a curse of the collections architecture; the latter even triggered a separate line of research which sole purpose was to improve the API documentation (Dubochet and Malayeri [2010]) and hide the implicit parameters from the users. To illustrate the problem we consider a function application that extracts an element of a list of integer values in Listing 7.21a.

The example presents a simple mapping of the elements of the function involving the identity function, and a failed attempt at extracting a single element from the resulting collection. A regular *TypeFocus*-based analysis would lead to improved error message such as:

```
// Expected type comes from the method ‘map’ declared in the ‘List’ class:
// [B, That](f: A => B)(implicit bf: CanBuildFrom[This,B,That]): That
//                                   ~~~~~~~~~~~~~~~~~~~~~~~~~
// (List(1, 2, 3) map identity)(1)   // error
//  ~~~~~~~~~~~~~
```

```
List(1,2).toSet.toList.sortBy(x => -x)

// missing parameter type
//            List(1,2).toSet.toList.sortBy(x => -x)
//                                          ^
// diverging implicit expansion for type scala.math.Ordering[B]
// starting with method Tuple9 in object Ordering
//            List(1,2).toSet.toList.sortBy(x => -x)
//                                         ^
```

(a) Lack of type propagation for the Set collection invariant in its type parameter

```
Specialized error message:
‘TraversableOnce’ defines a member ‘toSet’ of type
‘[B >: A]=> scala.collection.immutable.Set[B]’.
Type of the type parameter ‘A’, ‘Int’, which is a lower bound of type parameter ‘B’
is lost and type checker cannot infer the type of the parameter x.
            List(1,2).toSet.toList.sortBy(x => -x)
            ~~~~~~~~~~~~~~~
You may provide explicit type arguments to fix the problem:
List(1,2).toSet[Int].toList.sortBy(x => -x)
```

(b) A Scala collections-specific error message

Figure 7.22: An example of an unexpected lack of type propagation when working with collections from Standard Scala library.

The generic error message is correct but also unsatisfying.

Instead, with a library-specific error analysis, we have the ability to exploit the internal knowledge of the collections infrastructure and even propose a type-correct solution to the problem, as illustrated in Figure 7.21b.

The second example illustrating customizable error messages for Standard Scala Library concerns the Set collections. Unlike most of the collections, sets are invariant in the type of its elements. This fact limits Scala's ability to propagate type information, manifesting itself through incomprehensible error messages. The example in Listing 7.22a illustrates how an innocuous transformation of the list of integer values (from the list to a set, and later from the set to a list) loses type information about its elements, an operation that the Colored Local Type Inference was designed to prevent. The resulting type errors are not only misleading but also do not provide any indication on how to correct the program.

Because the type information is missing in the parameter of the function we cannot apply the regular *TypeFocus*-based analysis to identify the source of the problem. At the same time, the example is a perfect use-case for defining a problem-specific analysis that:

- Navigates the type derivation tree.
- Uses `TypeFocus` values, constructed in a problem-specific way, to guide the navigation.

To illustrate a custom application of the `TypeFocus` values, we will now delve into the details of the analysis of the problematic function application from Listing 7.22.

The type of the `sortBy` method defined in a `SeqLike[ A , Repr]` class is `[B](f: (A) => B)(implicit ord: math.Ordering[B]): List[A]`, and the 'missing parameter type' error relates to the type of the formal parameter `f`. Therefore the reconstructed `TypeFocus` value, represented visually through a grayed-out selection, will guide the analysis of the inferred type of the qualifier '`List(1,2).toSet.toList`'.
Similarly, the `toList` method defined in a `TraversableOnce[ A ]` class[3] is of type `List[ A ]`, and the non-local type parameter can again be represented through a grayed-out part in order to analyze the inferred type of the qualifier '`List(1,2).toSet`'.
Finally, the `toSet` method defined in the `List[A]` class is of type `[B >: A]Set[B]`, and the `toSet` method, as a member of the `List(1,2)` qualifier is of type `[B >: Int]Set[B]`. The application of the reconstructed `TypeFocus` value to the `[B >: Int]Set[ B ]` type will yield the type parameter with a delayed instantiation (thus no type propagation involved), *i.e.,* `B`. However, the local type parameter of the `toSet` method defines is defined with a lower bound that in our example is a known value type, namely the `Int` type that was not propagated.

The careful reconstruction and application of the `TypeFocus` values is at the core of the analysis of the delayed instantiation of non-local type parameters. It permits us to deterministically decide when to stop the analysis of the nodes of the type derivation tree and decides which type elements of the formal and the inferred types are relevant for the purpose of the problem.

In consequence the domain-specific analysis of the problem not only identifies the smallest expression where the complete type information of the collection is available but can also deliver a type-safe source code modification, as indicated in Figure 7.22b.

Explaining errors involving overloaded methods

Operator overloading is a common technique for a linguistic reuse of library or DSL constructs (Rompf et al. [2012]). The definition of a convenient API comes at a cost of error messages involving all the available alternatives of the overloaded methods. As an example, we consider a declaration of a simple test specification in the ScalaTest library (Listing 7.23), where we want to define an expectation regarding the value of one of the members of the nested class `Foo`. The details of the `ExampleSpec` class (line 3), defining the specification, and the human-readable declaration of the test case (line 7) can be ignored by reader. The innocuously looking statement in line 9 is rejected with an incomprehensible and an extremely long error message.

The simple property check leads to a rather elaborate error message involving a number of

[3]For simplicity, the reader can assume that the List collection is defined as `class List[+A] extends Seq[A]` and `class Seq[+A] extends SeqLike[A, Seq]`, and our *TypeFocus* mechanism, being owner-aware, will always extract the correct type argument.

```
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {
  class Foo {
    def status: String = "ABC"
  }
  it should "report sth" in {
    val id = new Foo
    id should ('status("ABC1"))
  }
}

// overloaded method value should with alternatives:
//   (notExist: org.scalatest.words.ResultOfNotExist)(implicit existence:
//     org.scalatest.enablers.Existence[ExampleSpec.this.Foo])Unit <and>
//   (existWord: org.scalatest.words.ExistWord)(implicit existence:
//     org.scalatest.enablers.Existence[ExampleSpec.this.Foo])Unit <and>
//   (containWord: org.scalatest.words.ContainWord)
//     org.scalatest.words.ResultOfContainWord[ExampleSpec.this.Foo] <and>
  <elided for lack of space...>
//   <and>
//   [TYPECLASS1(in method should)[_]](rightMatcherFactory1:
//     org.scalatest.matchers.MatcherFactory1[
//       ExampleSpec.this.Foo,TYPECLASS1(in method should)])(implicit typeClass1:
//         TYPECLASS1(in method should)[ExampleSpec.this.Foo])Unit
//  cannot be applied to (org.scalatest.matchers.HavePropertyMatcher[AnyRef,Any])
//     id should ('status("ABC1"))
//         ^
```

Figure 7.23: An example of a type error message that leaks the details of the overloaded method definition.

```
Specialized error message:
‘should’ is missing a concrete operator to handle a property
'status("ABC1")
Providing one of the explicit operators like:
‘should have’, ‘should be’, ‘should ===’, ‘should not’
might fix the problem.
```

Figure 7.24: An example of a human readable error message generated by the library-specific plugin for the program in Listing 7.23.

```
import org.specs2.mutable._
import org.specs2.matcher._

class Issue { def status: String = "ABC" }

class ExampleSpec extends Specification {
   "Retrieving open issues" should {
      "return expected properties with expected data" in     {
         val issue = new Issue()

         issue must not beNull
         issue.status must beEqualTo("ABC")
      }
   }
}
// method apply in trait MatchResult cannot be accessed in
//  org.specs2.matcher.MatchResult[Issue]
//     issue must not beNull
//                    ^
```

(a) Simple specification defined in the Spec library

```
Specialized error message:
Note that 'issue.must(not).beNull'
is put in the context of application.
'issue.number' is applied to it and therefore creates a confusing error message
The easiest is to wrap '(issue.must(not).beNull)' so that it
correctly parsers whitespace, or leave an empty line between the statements.
```

(b) A library-specific error message

Figure 7.25: An example of an overzealous parsing of a postfix operator leading to an unrelated type error message.

alternatives for the overloaded `should` method. The detailed message reveals a lot of internal details of the testing framework, none of which are particularly useful for the programmers who wrote the test. The scenario is not uncommon and is known to the authors of the testing framework. Due to the encoding of the methods that ensures human-readable names, an application of a generic *TypeFocus*-based analysis is unlikely to provide a comparable, human-readable, error message.

With a library-specific plugin it is, however, possible to define properties that characterize the problem, necessary for its identification. The type debugging tool instruments not only regular function applications but also the selection process of the individual alternatives. Based on the low-level data of the high-level goals, one can exploit the knowledge of the internal details of the library, and provide a human readable type error message, such as the one presented in Figure 7.24.

Explaining errors involving a postfix operator

The library-specific plugins not only can provide improved feedback to classical type mismatch errors, but also those which are indirectly caused by the overzealous parsing. Postfix and infix operators are, next to overloaded operators, a common technique for defining intuitive an API. At the same time statements involving such operators are not always easy to parse and may result in ill-defined ASTs. To illustrate the problem we consider a program written using the Specs testing framework (https://etorreborre.github.io/specs2/) in Listing 7.25a.

The specification `ExampleSpec` defines a human-readable test case where the member of the `Issue` class has to satisfy certain basic properties. Due to the overzealous parsing the type checking of the program returns a confusing type error that reveals the insignificant internal details of how the DSL is constructed.

Because the error is reported during type checking and thus is exposed in our high-level representation, a library-specific analysis of the type derivation tree is possible. Without going into the details of the high-level representation, we notice that a library-specific plugin can define properties that uniquely identify the problem, based on both the high-level derivation tree and the low-level information, and generate a library-specific error message such as the one in Listing 7.25b.

7.2.3 Infrastructure for error plugins

The library-specific error analyzers are defined through separate plugins. The plugins are defined in a separate namespace, preferably associated with the library or the DSL itself, and can be loaded on demand in a similar style to regular Scala compiler plugins[4].

Listing 7.26 defines a base class of the plugins, `DebuggerPlugin`, which contains a reference to the main type debugging tool (the `framework` member in line 3) that through the path-dependent types gives it access to its infrastructure, such as the high-level representation or the `TypeFocus` representation and operations on them. The handling is fully represented by the `analyze` method (line 8), which takes a high-level goal representing the reported error in the type derivation tree and its source code position. The method delegates to the plugin-specific `definedFor` partial function (line 5 in 7.26b, which defines a two-part selection process:

1. The first step (`definedFor.lift`) allows for the preliminary acceptance or rejection of the high-level goal representing the error, based on for example the low-level error message generated by the Scala compiler.
2. The high-level goal, for which the partial function is defined, returns the error specific handling function (the `f` parameter in line 10 in Listing 7.26a) that returns an optional

[4]http://www.scala-lang.org/old/node/140, the basic usage involves adding compiler options that specify the name of the plugin class and the classpath to its compiled sources.

```
abstract class DebuggerPlugin {

  val framework: TypeDebugger

  protected def definedFor: PartialFunction[framework.Goal,
                (framework.Goal, framework.global.Position) => Option[ErrorFeedback]]

  def analyze(err: framework.Goal, pos: framework.global.Position):
   Option[ErrorFeedback] =
    definedFor.lift(err).flatMap(f => f(err, pos))
}
```

(a) Plugin base class

```
class SpecsPlugin(val framework: TypeDebugger) extends DebuggerPlugin {

  import framework._

  val definedFor: PartialFunction[Goal,
        (Goal, global.Position) => Option[ErrorFeedback]] = {
    case ErrorGoal(errMsg, errPos) =>
      // ...
  }
}
```

(b) A fragment of the plugin for the Specs library

Figure 7.26: A fragment of the infrastructure of plugins for the type debugging tool.

error feedback (the definition of the `ErrorFeedback` class is omitted for irrelevance). Since the return value is optional, it gives the authors of the plugin a chance to perform a low-level verification of the problem and more involved patter matching, to make sure that the plugin truly applies to the desired error.

Listing 7.26b gives an example of the plugin that is specific to a Specs testing framework. The plugin class has to define a single parameter constructor with a parameter of type `TypeDebugger`. This way our tool can reflectively instantiate any plugin classes that has been registered through a separate, runtime configuration, and pass the correct instance of the type debugging framework as a dependency.

The infrastructure of the type debugging tool loads any registered error plugins in sequence, and passes the high-level goal representing the error to each of them, respecting the order of loading, until a non-empty error feedback value is returned. If no feedback is generated as a result, the goal is handled by any of the generic error handlers some of which have been presented in this thesis, if possible.

The complementary plugin infrastructure does not provide any guarantees regarding the loaded plugin classes, meaning that there can be potentially many plugins being able to handle a single error and their order matters, and that their identification of the problem is not statically verified. It is then the responsibility of the plugin author to ensure that the error they handle is specific to the library or the DSL itself, but they can rely on the high-level nodes of type derivation trees, their link to the low-level definitions, and any of the specialized high-level analysis functions. While limited in a sense of control, the types of type errors typically belong to different namespace and can be distinguished based on that low-level information.

7.2.4 Limitations

The range of issues for which a type debugging analysis is able to generate improved error feedback highlights the power of the *TypeFocus* abstraction and the high-level representation. At the same time we have to remember that the error messages themselves come with their own limitations:

- Users are unable to control the type debugging process, and are typically given ready solutions. Such approach is acceptable for languages with limited type system features, or simple examples, but for more complex examples a more in-depth analysis may be necessary.
- The programmers come with a different background and varying level of experience and understanding of the language.
- Programmers cannot better understand the type checking/type inference mechanism because they are unable to debug error-free programs.
- Programmers are unable to steer away from the primary error problem in an attempt to better understand the nature of the type checking context in which it occurred.
- Some type checking decisions, such as an implicit resolution, are impossible to fully explain with simple error messages.

To bring back the control of the type debugging analysis to the programmers and at the same time preserve the guided nature of the process, we notice that with our approach it is sufficient to satisfy two conditions:

1. We need to provide the ability to infer type selections on-the-fly, from the low-level data associated with the `TypingSlices`.
2. We need to provide a detailed list of possible expansion directions based on the located high-level goals and/or `TypingSlices`.

With such restrictions in mind we elaborate on the possible interactive type debuggers in the next section.

7.3 Interactive type debugging

We propose two approaches to interactive type debugging, both based on the inferred high-level representation representing the type checking of some source code:

- A visual interpretation of the type derivation tree, available in a form of a GUI (Section 7.3.1).
- A controlled way of navigating `TypingSlice` values, available through a console (Section 7.3.2). In a sense, the approach is similar to the solution of Sulzmann [2002] with an exception that we do not limit ourselves to just locating the source of the error.

Both of the strategies attempt to explain the type inference process for erroneous as well as error-free programs. As it turned out, the two approaches appeal to different audiences; with the visual interpretation, focused more on the ability to freely navigate through the nodes of the trees, the compiler programmers are able to discover patterns that later translate to programmable type debugging techniques, while the controlled navigation using typing slices brings a much needed order and guidance, for a regular user.

7.3.1 Visual exploration

In our formalization of the *TypeFocus*-based approach we postulate that the natural way to explaining the type checking process is through the type derivation trees themselves. Unfortunately, the related work, such as Duggan and Bent [1996], has either focused on displaying complete derivations upfront or just specific fragments, which is either unrealistic or incomplete when dealing with real-world programs.

We propose a visualization of the type checking process, described in more detail in Plociniczak [2013] and Plociniczak and Odersky [2012], where:

1. The type derivation tree is represented through an inverted tree structure, where the nodes correspond to the high-level goals, and the undirected edges between them represent the values of the declared members (or, in our notation, dependencies).
2. For orientation purposes, each of the nodes has to be accompanied with a short description of its role.
3. The visualization is accompanied by a source code editor on the side. The editor allows for selecting fragments of programs which trigger an instrumented compilation and, later, their visualizations.

4. For a complete understanding of the nodes of the type derivation tree, each has to be accompanied by a full description of its purpose, and its low-level data. The information is not essential for navigating every node of the tree, therefore will only be displayed on demand.

We tame the exploration of the reconstructed type derivation trees by reducing the amount of information that has to be considered at once and providing visual clues:

1. A selection of a fragment of the source code reconstructs a complete type derivation tree representing it, but we never display it in full. Instead, we initially only display a fragment of the tree that corresponds to type checking the smallest enclosing AST node.
2. Nodes of the type derivation tree are allowed to be expanded and collapsed at will.
3. Hovering over the nodes with a mouse pointer highlights the corresponding source code fragment it refers to. We found out that maintaining a visual link between the abstract interpretation of the type checking process and the tangible source code, whether through means of highlighting the source code fragments or printing them in a user's console, is crucial for keeping control of the exploration process and not losing the orientation in the type debugging process.

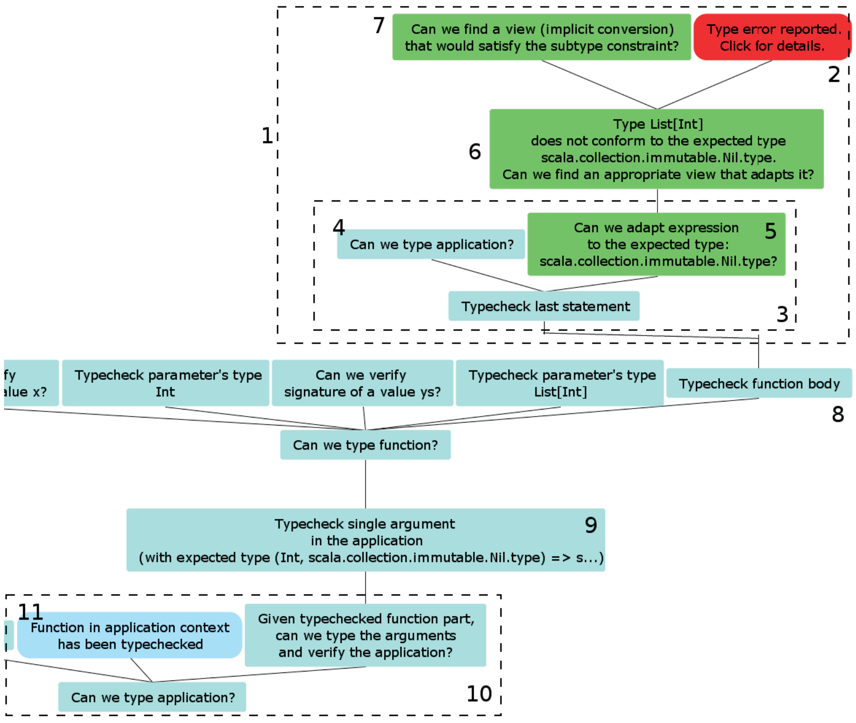
To illustrate the application of the above clues we delve into the details of the visualization that represents the type checking of our motivating `foldRight` function application example from Section 3.1.

Explaining `foldRight` visually

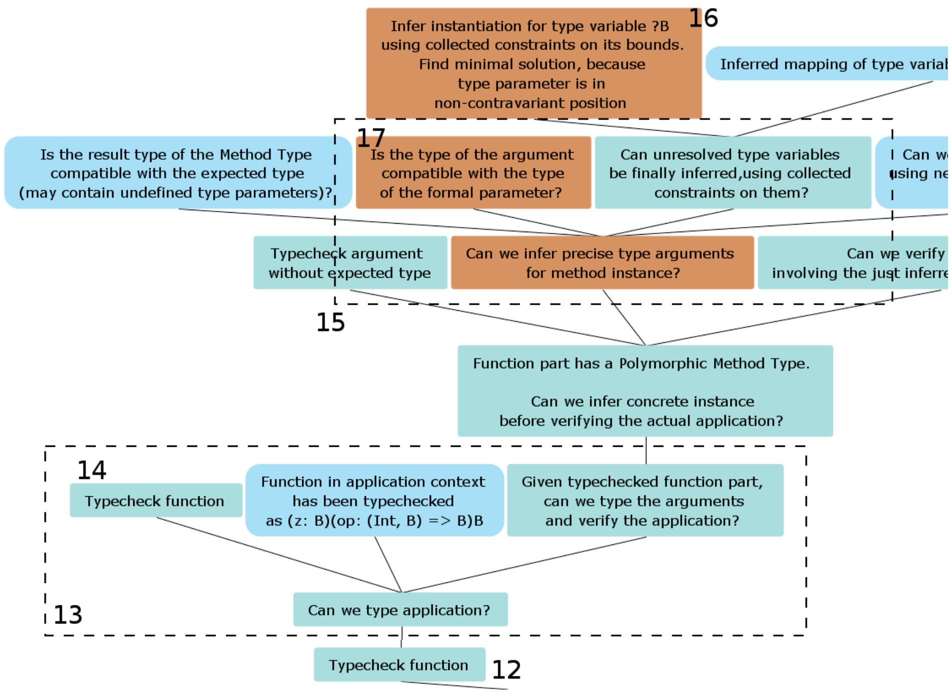
The result of the initial targeted compilation of the fragment of the erroneous `foldRight` application is visible in Figure 7.27a. To explain the dependencies of the individual goals we added auxiliary selection boxes (with dashed lines), not present in a real visualization.

The reported type error (*goal 2*) occurred while type checking the last statement of an anonymous function (*selection 3*). Its immediate subgoals consist of typing and adaptation. Typing involves type checking all the components of the abstract syntax tree (AST), and then assigning the type based on its kind and context. The typing goal (*4*) verifies the application `(x + 1)::ys`. The adaptation stage makes sure that the inferred type conforms to the expected one. This may involve inferring still undetermined type parameters, applying implicit arguments or performing necessary conversions. Therefore (*goal 5*) involves adapting `List[Int]` to `Nil`.

The error occurred as `List[Int]` does not conform to `Nil` (*goal 6*) and the implicit conversion fallback was unsuccessful (*goal 7*). One understands the context of the error by tracking back the type checking process, starting with *Typechecking the last statement*, *Typechecking function body* (*goal 8*) and arriving at *Typechecking argument with expected type (Int, Nil) => Nil*


7
Can we find a view (implicit conversion) that would satisfy the subtype constraint?
Type error reported. Click for details.
2
1
6
Type List[Int] does not conform to the expected type scala.collection.immutable.Nil.type. Can we find an appropriate view that adapts it?
4
Can we type application?
Can we adapt expression to the expected type: scala.collection.immutable.Nil.type?
5
Typecheck last statement
3
Typecheck parameter's type Int
Can we verify signature of a value ys?
Typecheck parameter's type List[Int]
Typecheck function body
8
Can we type function?
Typecheck single argument in the application (with expected type (Int, scala.collection.immutable.Nil.type) => s...)
9
11
Function in application context has been typechecked
Given typechecked function part, can we type the arguments and verify the application?
Can we type application?
10

(a) A fragment of the visual interpretation of the type checking of the `foldRight` function application (part 1)


Infer instantiation for type variable ?B using collected constraints on its bounds. Find minimal solution, because type parameter is in non-contravariant position
16
Inferred mapping of type varial
17
Is the result type of the Method Type compatible with the expected type (may contain undefined type parameters)?
Is the type of the argument compatible with the type of the formal parameter?
Can unresolved type variables be finally inferred,using collected constraints on them?
Can we using ne
Typecheck argument without expected type
Can we infer precise type arguments for method instance?
Can we verify involving the just inferre
15
Function part has a Polymorphic Method Type.
Can we infer concrete instance before verifying the actual application?
14
Typecheck function
Function in application context has been typechecked as (z: B)(op: (Int, B) => B)B
Given typechecked function part, can we type the arguments and verify the application?
Can we type application?
13
Typecheck function
12

(b) A fragment of the visual interpretation of the type checking of the `foldRight` function application (part 2) (*goal 9*). At that point, we reached type checking the full application `x.foldRight(Nil)(...)` (*selection 10*). Its function part has already been typechecked and has a concrete instance of a method type `(op: (Int, Nil) => Nil)Nil` (*goal 11*). Since the searched type `Nil` is clearly related to the type of the function part, the user would expand the goal responsible for type checking `x.foldRight(Nil)` (*goal 12*, Figure 7.27b) for which the compiler poses the question *Can we type application?*. Since we are again in the application context (*selection 13*), the derivation tree involves verifying the function part `x.foldRight` which ends up with a generic method type `(z: B)(op: (Int, B) => B)B` (*goal 14*). Hence the user can reasonably expect that the type inference will take place in the next subgoal *[...]can we type arguments and verify the application?*. Further expansion proves the existence of type inference: *Can we infer precise type argument for method instance?* (*selection 15*). The latter contains two interesting subgoals: *Is the type of the argument compatible with the type of the formal parameter?* and *Can unresolved type variables be finally inferred...?*. Both reveal the internal details of the process that instantiates type parameters, including of the type parameter `B`.

The succinct messages of the individual nodes are accompanied by the more elaborate descriptions, such as the ones above, by clicking on the goals of the type derivation tree in the GUI.

Limitations

Since the interactive type derivation tree exploration directly relates to the high-level representation, it becomes possible to apply the *TypeFocus*-based analysis to the underlying goals. This way we can avoid tedious manual inspection of nodes, and can provide interactive actions that explain the source of types by automatically expanding the nodes of the type derivation tree.

The visual type debugger provides an attractive way of presenting a complex process of type checking. For example, the tool has been used extensively in finding bugs within the implementation of the compiler or as a support tool for the experienced compiler hackers in explaining type errors to other programmers. At the same time, sheer number of the possible nodes, and implicit connections between their low-level data, make it hard for regular users to apply it in every day programming, even if the tool is accompanied by the necessary tutorials or documentation. The learning curve is particularly steep for programmers with little, or no, prior exposure to formal type systems.

The user studies with the visual type debugger have clearly indicated that the main reason for lack of adoption of the tool by the regular Scala programmers is its overwhelming freedom of navigation. The finding has led to the development of an interactive type debugger, that lacks the visual appeal of type derivation trees, but promotes type debugging in a form Q/A sessions that behind the scenes step through the nodes of trees without users realizing it.

7.3.2 Guided type debugging with Typing Slices

In this section we propose a debugging approach to navigating the decisions of type derivation trees, that does not require from the end-users to be aware of the derivations or knowledge of the dependencies of the applied typing rules. Instead, the guided navigation is implemented in terms of a limited number of *questions*, that are associated with the Typing Slices. In this interactive approach:

- Each Typing Slice kind is as associated with a template, but the questions themselves are dynamically generated. This means that the information about the types or program locations is specific to the underlying high-level goal and their low-level data, and, indirectly, the source code.
- The questions can take input from the users that will allow for modifying the direction of the analysis on-the-fly.

The difference with the visual type debugger lies in the navigation technique; rather than trying to explain the type checking process with an unfamiliar and abstract concept of high-level goals, and their dependencies, the interactive sessions are controlled only with the use of types, *e.g.,* types of type constraints, expressions, type signatures.

The key concept for such a platform-agnostic approach lies again in Typing Slices, or rather values that they represent; we recall that every `TypingSlice` value is associated with a high-level goal and a `TypeFocus` value. This implies that every `TypingSlice`-related decision can provide an interpretation that is both human-readable, and relates to the source code fragment. Furthermore, we notice that the high-level representation, and inference judgments it represents, is fixed with respect to source code, by design. Therefore the only means of controlling the direction of the analysis of the type checking process is through the `TypeFocus` values. By providing means to the end-users to construct the `TypeFocus` values on-the-fly from the existing low-level types, we allow them to change the course of the navigation at every `TypingSlice` step.

The *answers* of the debugging sessions are grouped into three kinds of actions:

1. Informative - explaining the low-level information associated with the `TypingSlice` value, such at the type of the term or the type of the method as a member of the qualifier. In that sense the action is static because it does not involve any kind of analysis of the high-level goal, or its role in the type derivation tree.

   For example, given a `TypingSlice` value representing the instantiation of a single type parameter, the possible questions would be: "Show the type of the instantiated type parameter, and its type selection", "List all local type parameters of the function application and their inferred type arguments", or "Show the complete type of the function application". In other words, we can provide a broad spectrum of questions and answers that would typically be associated with a given code fragment.

2. An analysis of the `TypingSlice` and its decision without modifying the identity of the `TypingSlice` (or taking a step).

   For example, given a `TypingSlice` value representing the instantiation of a single type parameter, say `A`, the possible action might be: "List all lower bound type constraints collected for the type parameter A", or "List all type constraints that conflict with the instatiation to the expected type [...]", where the "[...]" parts represents the input given by the users.
   The result of such query will perform a shallow expansion in place and retrieve the low-level data necessary to answer the question (in terms of types or source code locations), but will not step into a `TypingSlice` representing any of the type constraints.

3. An analysis of the `TypingSlice` that will take a step to a single `TypingSlice` based on the included or the inferred `TypeFocus` value.

   For example, the question "Explore the inferred instantiation of type parameter A" would be ambiguously formulated because it implicitly assumes the possibility of stepping into multiple `TypingSlice` values. Instead, the questions would be formed in a way that avoids the ambiguous expansion: "Explore the source of type constraint number [...]", "Explore the type constraint number [...] with the expected type [...]", or "Explore the instantiation of the type parameter [...]", where "[...]" stands for the input required from the user, such as the index of the type constraint, its expected type value, or the name of the other local type parameter instantiated for the same function application, respectively.

   The query side-effects by internally triggering a shallow *TypeFocus*-based analysis that results in a new `TypingSlice` value, that will serve as a basis for future queries.

The intermediate steps of the type debugger are not forgotten, and the user can always step back and change the direction of the process, without restarting the type debugging session.

To guide the exploration we notice that the two non-static types of questions can take as input user-provided types. We make use of the fact that the type debugger infrastructure already provides ways to construct subtyping derivations from the low-level data and translate the potentially failed subtyping derivations to `TypeFocus` values. For convenience, we define two ways to infer the user-provided `TypeFocus` values:

- The type (known as the *expected type*) provided by the user is compared to thg type associated with the `TypingSlice` (or any other low-level type, source of which we want to analyze) in a subtype check. If the two types are not comparable we can simply translate the failed subtyping derivation to a `TypeFocus` value. Since the resulting *TypeFocus* is well-formed with respect to the `TypingSlice`, we can trigger a shallow `TypingSlice` expansion with the inferred `TypingSlice` value.

- To reflect the type elements that the end-user might be interested in (or not) we allow the user-provided types to contain special `?` and `!` type constants, where `?` represents a 'don't care' type element and `!` stands for the part that she is interested in, while still preserving the regular subtyping rules.

  For example, when the expected type `Map[?, !]` is compared with the type of some expression, say `Map[Int, Int]`, the resulting `TypeFocus` value will indicate a type selection extracting the second type argument of the map collection, *i.e.,* the end-user is interested only in how the type of the values of the map collection was inferred. On the other hand, if the user-provided expected type `List[!]` is compared with `Map[Int, Int]` type, the `TypeFocus` value will be equivalent to an identity type selection, according to the algorithm that implements the subtyping check.

The user-provided types are parsed and reconstructed in the same typing context as the original type, in order to define types that can be compared. After the *TypeFocus* translation the provided types can be discarded.

In that sense, our interactive approach provides more type debugging control than the one presented in the related work (Chen and Erwig [2014a] and Sulzmann [2002]), where the navigation through the inferred types and their sources is only performed by means of yes/no answers, such as "Is the expected type of expression x, Int?", and complete types. The interactive type debugger does not require any special infrastructure changes since the expansion of `TypingSlice` values representing the intermediate typing decisions is already *TypeFocus*-driven. At the same time the programmers using the interactive type debugger do not have to understand the abstract concepts of `TypeFocus`, high-level representation or `TypingSlice`, and can control the navigation only using types, *i.e.,* the abstractions that they have to comprehend in every day programming.

We illustrate the process of controlled type debugging session with the simplified transcripts of the two small programs involving the `foldRight` function application and the implicit resolution. The interactive sessions will be presented in a form of Q/A sessions where we provide a selection of the possible questions (due to space limitations). The output from the type debugger is preceded by the '>' symbol and the input from the user is preceded with the '<' symbol. For the presentation reasons all source code fragments and type values are highlighted.

Example: Debugging `foldRight` application

In the first interactive type debugging session we consider an erroneous function application of `foldRight` from Section 3.1.2. For reference, we recall the corresponding code snippet below.

```
val xs = List(1, 2, 3)
xs.foldRight(Nil)((x: Int, ys: List[Int]) =>
 (x + 1) :: ys)
// error: type mismatch;
// found   : List[Int]
// required: Nil.type
//  (x + 1) :: ys)
//          ^
```

Figure 7.28 illustrate a two-part transcript of an interactive type debugging session exploring the type checking decisions of the above code snippet. The debugging starts with a statement of the problem and listing all possible actions possible for the given type checking decisions, with Typing Slice serving as a source of the possible actions. User control the execution by issuing commands of a format '`< [action index] [show/explore] [...]`' with the index of the requested action, whether it should expand the Typing Slice or not, and any additional input parameters, such as types.

With the additional type parameter, the users can change the direction of the debugging session and, for example explore why the type assigned to the '`(x+1) :: ys`' expression was not '`List[String]`'. The *TypeFocus* value reconstructed from the subtyping check between the two types will trigger a *TypeFocus*-based analysis that either takes a step to a resulting Typing Slice or returns a human-readable representation of it, depending on the user-provided command.

The set of actions associated with each of the Typing Slices is limited, in a sense that they only concern the decisions of the individual Typing Slice and not of the entire type derivation tree. It is also verbose, because the amount of information necessary to explain the typing decisions and its underlying low-level data is non-trivial. At the same time, the users are able to modify the *TypeFocus* value and step through the multiple type checking decisions in a quick succession, without having to deal with the details of the actions at each point. The format is sufficient to compare the previously elided information about the types with instantiated and not-yet instantiated type variables while also providing a necessary guidance with the included type selection.

```
> The type of the expression
  (x + 1) ::  ys
  ~~~~~~~~~~~~~~
  is of type List[Int] and
  does not match the type of the formal parameter
  Nil.type
  1) Explore/show the source of the expected type Nil.type.
  2) Explore/show the source of the type of the expression.
  3) Explore/show the source of the expected type with the user-provided type [type]
  4) Explore/show the source of the type of the expression with
     the user-provided type [type].
  ....
< 4 show List[String]
> The selected user-provided type
    List[String]
         ~~~~~~
  conflicts with the location(s):
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
                                          ~~~~

> // (The message repeated from the previous question)
< 1 explore

> The requested type has been instantiated in the function application
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
    ~~~~~~~~~~~~~~~~~~
  1) Show the type of the function and the inferred type of the function application.
  2) Show type constraints that instantiated the type of the type parameter.
  3) Explore/show the source of the type constraint with
     the expected type ([index] [type]).
  4) Explore/show the source of the function with the user-provided type.
  ...
< 1 show
> The function
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
    ~~~~~~~~~~~~~
  is of type
    [B](z:  B)(op:  (Int, B) => B)B
                                  ~
  The inferred type of the function application is
    (op:  (Int, Nil.type) => Nil.type)Nil.type
                             ~~~~~~~~~

> // (The message repeated from the previous question)
< 4 show [B](z:  B)(op:  (!, B) => B)B
> The user-provided type
    [B](z:  B)(op:  (Int, B) => B)B
                     ~~~
  has been inferred with the location(s):
    val xs = List(1, 2, 3)
                  ~  ~  ~
```

Figure 7.28: (Part 1) An illustration of a transcript from an interactive debugging session for the `foldRight` function application.

```
> The requested type has been instantiated in the function application
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
    ~~~~~~~~~~~~~~~~~~
  1) Show the type of the function and the inferred type of the function application.
  2) Show type constraints that instantiated the type of the type parameter.
  3) Explore/show the source of the type constraint with
     the expected type ([index] [type]).
  4) Explore/show the source of the function with the user-provided type.
  ...
< 4 show !
> The user-provided type
    [B](z:  B)(op:  (Int, B) => B)B
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  has been inferred with the location(s):
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
    ~~~~~~~~~~~~~

> // (The message repeated from the previous question)
< 2 explore
> Internal Error: Cannot explore that question.

> // (The message repeated from the previous question)
< 2 show
> Type parameter
    [B](z:  B)(op:  (Int, B) => B)B
                                   ~
  has been instantiated using the following location(s):
    (0) xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
                      ~~~
  of type Nil.type

> // (The message repeated from the previous question)
< 3 explore 0

> The requested type has been instantiated in the identifier
    xs.foldRight(Nil)((x:  Int, ys:  List[Int]) =>
                  ~~~
  1) Show the type of the identifier.
  2) Explore/show the source of the identifier with the user-defined expected type.
< 1 show
> The inferred type of the identifier is
    List[Nothing]
         ~~~~~~~~

> // (The message repeated from the previous question)
< 2 explore Int
> The selected user-provided type
    List[String]
    ~~~~~~~~~~~~
  Cannot further explore the source of the inferred type of the identifier.
  (source of the identifier is unknown).

> // (The message repeated from the previous question)
... // (the type debugging session continued by the user)
```

Figure 7.28: (Part 2) An illustration of a transcript from an interactive debugging session for the `foldRight` function application.

Example: Debugging implicit resolution

In the second interactive type debugging session we consider the encoding of the generic comparison function and its application, as already discussed in Section 6.6.4. For reference, we recall the corresponding code snippet below.

```
abstract class A { def f: Any }
class B extends A { def f: Int = 5 }
class C extends A { def f: Long = 5L }

def universalCompare[T: Ordering](t1: T, t2: T): Int = // ...
object Test {
  implicit val AOrdering: Ordering[A] = // ...
  universalCompare(new B, new C)  // No implicit Ordering defined for A{def f: AnyVal}.
                                  //          universalCompare(new B, new C)
                                  //                                ^
}
```

The opportunistic algorithm from Section 6.6.4 defined a generic way to identify the source of the rejected implicit resolution. In reality, it is not always possible to define a set of properties that define the problem or provide improved feedback that satisfies all users.

Figure 7.29 illustrates a three-part transcript of an interactive type debugging session exploring the type checking decisions of the above code snippet. The debugging starts with a statement of the problem and listing all possible actions possible for the failed implicit resolution as well as the partially inferred type of the `universalCompare(new B, new C)` function application.

By default the interactive debugging session does not attempt to trigger any domain-specific analysis and the reconstructed *TypeFocus* value will only extract the complete formal type of the implicit parameter. With the user-defined types, the type debugging session is not only limited to the analysis of the implicit resolution. The actions and their type parameters can steer the exploration of underlying type derivation tree to any of the previous typing decisions, as we illustrate in Figure 7.29.

The Typing Slices representing the decisions of the implicit resolution provide now a controlled way of exploring, for the first time, the internal details of the implementation (part (2) and part (3)). The analysis is constrained by the type checking context where the implicit resolution was triggered. Because the interactive Type Debugger is built on top of the existing high-level representation, the actions and their results return information that is specific to the particular node in the type derivation tree. This implies for example that different implicit parameters will only list their own implicit resolution scopes, as presented in the scenario.

```
> The inferred type of the function application involved an implicit parameter.
  The type checker failed to infer the implicit argument.
  universalCompare(new B, new C)
                   ~
  1) Show the inferred type of the function and the function application.
  2) Explore/show the source of type of the function application
     with the user-provided type.
  3) Show all the eligible implicit arguments for the [parameter name] [expected type].
  4) Explore the implicit resolution for the parameter of name [parameter name].
< 1 show
> The inferred type of the function
    universalCompare(new B, new C)
    ~~~~~~~~~~~~~~~~~
  is
    [T](t1:  T, t2:  T)(implicit evidence$1:  Ordering[T])T
                        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  The inferred type of the function application is
    (implicit evidence$1:  Ordering[A { def f:  AnyVal }])A { def f:  AnyVal }
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

> // (The message repeated from the previous question)
< 2 show (implicit evidence$1:  Ordering[A { def f:  !  }])?
> The user-provided type
    (implicit evidence$1:  Ordering[A { def f:  AnyVal }])A { def f:  AnyVal }
                                                 ~~~~~~~
  has been inferred with the location(s):
    (0)
      universalCompare(new B, new C)
                       ~~~~~
    (1)
      universalCompare(new B, new C)
                              ~~~~~~

> // (The message repeated from the previous question)
< 3 show evidence$1 A
> The eligible implicit values that conform to the user-provided type A:
  None.

> // (The message repeated from the previous question)
< 3 show evidence$1 Ordering[A]
> The eligible implicit values that conform to the user-provided type Ordering[A]:
    (0) AOrdering of type => Ordering[A]

> // (The message repeated from the previous question)
< 4 explore evidence$1
```

Figure 7.29: (Part 1) An illustration of a transcript from an interactive debugging session for the generic `universalCompare` function.

```
> The debugging has identified the implicit resolution with the expected type:
    Ordering[A { def f:  AnyVal }]
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  at location
    universalCompare(new B, new C)
                     ~
  The type checking has failed to infer a unique implicit value of the given type.
  1) Show all locations where the implicit values are taken from.
  2) Show all the implicit values in the user-selected locations [scope index].
  3) Find and show all the implicit values that are eligible for the expected type [type].
  4) Show all accepted implicit values matching the expected type.
  5) Explore the implicit value matching the expected type [value index].
< 1 show
> Locations of the implicit values:
  (1) Implicit members of the object Test
  (2) Explicitly imported implicits from object Predef
  (3) (Companion implicits) Implicit members of the object Ordering
  (4) (Companion implicits) Implicit members of the trait LowPriorityEquiv

> // (The message repeated from the previous question)
< 2 show 1
> Implicit value of the object Test:
  (1) AOrdering: Ordering[A { def f:  Any } ]
      [rejected]                        ~~~

> // (The message repeated from the previous question)
< 2 show 3
> Implicit value of the trait LowPriorityEquiv:
  (1) universalEquiv: [T] => scala.math.Equiv[T]

> // (The message repeated from the previous question)
< 4 show
> Eligible implicit values matching the expected type:
  (1) comparatorToOrdering:[A](implicit cmp:  java.util.Comparator[A])Ordering[A]
  (2) ordered:[A](implicit evidence$1:  A => Comparable[A])Ordering[A]

> // (The message repeated from the previous question)
< 5 explore 1

> The implicit resolution mechanism selected value
    comparatorToOrdering
  of type
    (implicit cmp:  Comparator[A { def f:  AnyVal }])Ordering[A { def f:  AnyVal }]
                                                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  at location
    universalCompare(new B, new C)
                     ~
  1) Explore the implicit resolution for the implicit parameter [parameter index].
  2) Show all inferred implicit arguments for the implicit parameters.
< 1 explore 1
```

Figure 7.29: (Part 2) An illustration of a transcript from an interactive debugging session for the generic `universalCompare` function.

```
> The debugging has identified the implicit resolution with the expected type:
    java.util.Comparator[A { def f:  AnyVal }]
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  at location
  universalCompare(new B, new C)
                   ~
  The type checking has failed to infer a unique implicit value of the given type.
  1) Show all locations where the implicit values are taken from.
  2) Show all the implicit values in the user-selected locations [scope index].
  3) Find and show all the implicit values that are eligible for
     the expected type [type].
  4) Show all accepted implicit values matching the expected type.
  5) Explore the implicit value matching the expected type [value index].
< 4 show
> Eligible implicit values matching the expected type:

> // (The message repeated from the previous question)
< 1 show
> Locations of the implicit values:
  (1) Implicit members of the object Test
  (2) Explicitly imported implicits from object Predef

> // (The message repeated from the previous question)
... // (the type debugging session continued by the user)
```

Figure 7.29: (Part 3) An illustration of a transcript from an interactive debugging session for the generic `universalCompare` function.

The current prototype of the interactive type debugger is entirely console-based. Therefore all the debugging sessions are available only for programs that report some type errors. The location in the type derivation tree, where the selected type error is reported, serves then as a starting point for the user-directed type debugging sessions. With the IDE-integration the users will be able to select a fragment of the program that they want to debug, and the type checking goal that encloses the given fragment and the identity type selection will then serve as a starting point to any user-directed type debugging session.

The user-defined types do not modify the type checking of the programs. Instead, we only use them to modify the *direction* of the exploration. This limitation is strictly dictated by the backtracking mode of operation of the Type Debugger. To persist the changes proposed by the users the Type Debugger would have to trigger a separate type checking run, that would construct a separate type derivation tree. The two derivation could be potentially merged in order to continue the analysis with the persisted type changes, but the topic is left as a subject for future work.

7.4 Conclusions

We explored a number of possible applications of the *TypeFocus*-based analysis of the type checking decisions. All of the examples have been implemented for the existing Scala implementation (versions 2.10.x and 2.11.x) which we believe to be a good indication the validity of the approach; the type system of Scala presents non-trivial challenges in terms of the number and complexity of type features, as well as their implementation in the compiler. Importantly, the analysis of all of the presented features was possible only with the *TypeFocus* instances, appropriately defined Typing Slices and the specialized functions associated with the latter.

The novelty of the interactive approach lies not in the approach (it has been explored before in for example Sulzmann [2002] and Chen and Erwig [2014a]) but our ability to implement them with minimal effort and no changes to the core analysis algorithm. The examples and the improved feedback have been either verified with the original posters of the problems or with the regular Scala programs; a complete user-study that would empirically test the usability of type debugger on the larger audience in the controlled environment requires further work.

Chapter 8

Conclusions

This thesis addresses an important subject of local type inference and the problem of lack of understanding of its limitations and the decision process by the programmers. The thesis does not seek to improve the existing type checking algorithm so that it would for example accept more unannotated programs by employing more sophisticated constraints and solving techniques. Instead we show that it is possible to take an existing type inference algorithm and its implementation, with all its imperfections, and workaround the problems by generating error messages that educate programmers. In a sense we provide the type checking history that is otherwise lost during compilation in a format that is possible to understand to regular programmers. Improved feedback involves listing minimal program fragments that determine the inference of types, providing more elaborate errors that identify and inform about the limitations of type inference or the implementation, and code modifications that can immediately correct the encountered type errors. This way we provide a key to decrypting local type inference. Our work has identified two key components necessary to debug the decisions of local type inference.

We presented a lightweight approach to extracting the details of the existing type checking process by instrumenting its implementation. The process is manual and complete, in a sense that the proposed instrumentation infrastructure does not artificially limit the kind of language features it can handle. On the example of Scala, we have shown how to extract information from the regular language constructs such as function applications or anonymous functions, as well as more more involving and vulnerable to any code changes, the implementations of the subtyping algorithm and implicit resolution.

The instrumentation on its own provides access to the low-level type checking data but, as we discovered, it is too basic to define rules that can navigate it. It is essentially impractical to model different type checker executions using only such low-level representation, not to mention defining any rules that analyze it. That is why we developed a high-level representation that models the decisions process of type checking and using the deterministic mapping strategy we can map between the two entities. The mapping ensures that the exposed high-level representation models an existing type checker execution, rather than some simplified or inaccurate version of it. The mapping is not an approximation in a sense that if there exists no high-level interpretation of the low-level execution, the debugging process is rejected. Lack of sufficient instrumentation or an appropriate high-level representation that models the type checkers execution can be fixed with relatively little effort. With such guarantees our proposed technique differs from the related work that is either based on separate, approximated models of the type checking process that do not maintain the link with the compiler execution, or a subset of the language.

We are the first to show a feasible solution to discovering the *long distance* relations between the type checking decisions that can frequently occur in the process of local type inference. We have shown that the approximations that take place in the case of local type inference do not necessarily limit our debugging capabilities. By providing a complete formalization of the analysis that is not tied to any particular implementation of the type checking process, we can navigate the decisions in a controlled way and only focus on and explain to the users the relevant parts.

The main challenge of debugging the type derivation trees inferred using local type inference, the amount and complexity of typing decisions that take place for each of the involved terms, was solved by maintaining the type propagation information while we traverse the nodes of the derivations. We proposed a lightweight abstraction that encapsulates not only the details of the type that we investigate but also allows us to deterministically guide the navigation through the complicated type derivation trees without the external help from the users. With clear rules on how to infer the lightweight abstraction and construct we are able to decrypt the type errors, *i.e.,* we are able to find the minimal type checking decisions and the minimal program locations that define it, with complete confidence.

We have shown a formal approach on how to apply the ability to discover the minimal program locations that define the source of the types that participate in type conflicts. But our approach is not just limited to explaining the type errors through more elaborate error messages. We have shown examples where with the intermediate results of the analysis we are able to swiftly navigate to the desired elements of the type checking process and define more precise heuristics for the domain-specific problems as well as to serve as a convenient abstraction that controls the interactive mode of our tool.

Our approach has been implemented in Scala, an industry-accepted programming language that exercises a number of advanced type systems. We evaluate our technique on a number of those complex features, including the analysis of implicit resolution, subtyping, polymorphic polymorphism, overload resolutions and path-dependent types, giving us confidence in its capabilities, especially since some of them have not been tackled before at all.

The type errors are here to stay, but with our work we have provided a foundation for generating better feedback that minimizes the effort to understand them.

Appendix A

Encoding of lists in Colored Local Type Inference

Using the external language of Odersky et al. [2001] we can provide intuitive definitions for lists using records with a single, `match`, member that uses the *visitor* pattern for inspection.

```
                                             b
type List[a]      = { match: (ListVis[a,b])  →  b }
type ListVis[a,b] = { caseNil: () → a, caseCons: (a, List[a]) → b }
```

Type constructor `List` takes a single type parameter, *a*, that represents the elements of the list, while the local type parameter of match, *b*, represents the type of the result of matching on such a list with a list visitor. The `ListVis` type constructor is a record type requiring two functions that are called either when an underlying list is empty (`caseNil`) or when it is not (`caseCons`). Instances of list constructors - for an empty, `Nil`, and a non-empty `Cons`, lists - can then be expressed in a following way:

```
Cons = fun[a](x:a, xs:List[a]):List[a] = { match = fun(v) v.caseCons(x,xs) }
```

Consider the following definition of the `foldRight` function mentioned in the introduction and semantically identical to Scala's `foldRight` operator in `List` collections:

```
foldRight = fun[a](elems: List[a]) fun[b](acc: b) fun(f: (a, b) -> b)
elems.match { caseNil() = acc,  caseCons(x, ys) = f(x, foldRight(ys)(acc)(f)) }
```

This definition uses a straightforward encoding of lists defined above.

`foldRight` simply traverses the initial list of elements, `elems`, until an end is reached, `caseNil`, where it returns the initial accumulator. Then it applies function `f` to the head of the list and the result of the recursive call on the suffix of the list. Using the typing rules from [Odersky et al., 2001, pg. 6], function `foldRight` has type, $\forall a.\ List[a] \rightarrow (\forall b.\ List[b] \rightarrow ((a,b) \rightarrow b) \rightarrow b)$. Type variables above are treated separately only in an effort for the encoding to be syntactically more closely to Scala's type inference limitation, however they could equally be placed in the first function type.

Appendix B

Auxiliary functions used in the *TypeFocus*-driven algorithm

This chapter provides implementation details for functions used in the definition of the *Type-Focus*-driven algorithm in Section 3.5.

$\texttt{is-hole}: P \to \texttt{Bool}$

$$\begin{array}{lcl}
\texttt{is-hole}(?) & = & \texttt{true} \\
\texttt{is-hole}(\bot) & = & \texttt{false} \\
\texttt{is-hole}(\top) & = & \texttt{false} \\
\texttt{is-hole}(a) & = & \texttt{false} \\
\texttt{is-hole}(\forall \overline{a}.A \to B) & = & \texttt{false} \\
\texttt{is-hole}(\{x_1 : T_1, \ldots, x_n : T_n\}) & = & \texttt{false}
\end{array}$$

$\texttt{is-tvar}: (T, \overline{a}) \to \texttt{Bool}$

$$\begin{array}{lcll}
\texttt{is-tvar}(?, \overline{a}) & = & \texttt{false} & \\
\texttt{is-tvar}(\bot, \overline{a}) & = & \texttt{false} & \\
\texttt{is-tvar}(\top, \overline{a}) & = & \texttt{false} & \\
\texttt{is-tvar}(a, \overline{a}) & = & \texttt{true} & \text{if } a \in \overline{a} \\
\texttt{is-tvar}(a, \overline{a}) & = & \texttt{false} & \text{if } a \notin \overline{a} \\
\texttt{is-tvar}(\forall \overline{a}.A \to B, \overline{a}) & = & \texttt{false} & \\
\texttt{is-tvar}(\{x_1 : T_1, \ldots, x_n : T_n\}, \overline{a}) & = & \texttt{false} &
\end{array}$$

Figure B.1: (Part 1) Complete implementations of auxiliary functions introduced in Section 3.5.

$\texttt{shape-match} : (T,\ P,\ \Theta) \rightarrow \texttt{Bool}$

$$\texttt{shape-match}(T, ?, \Theta) \quad = \quad \texttt{true}$$

$$\texttt{shape-match}(T, \bot, \Theta) \quad = \quad \begin{cases} \texttt{true} & \textbf{if} \quad T = \bot \\ \texttt{false} & \textbf{else} \end{cases}$$

$$\texttt{shape-match}(T, \top, \Theta) \quad = \quad \begin{cases} \texttt{true} & \textbf{if} \quad T = \top \\ \texttt{false} & \textbf{else} \end{cases}$$

$$\texttt{shape-match}(T, a, \Theta) \quad = \quad \begin{cases} \texttt{true} & \textbf{if} \quad T = a \\ \texttt{false} & \textbf{else} \end{cases}$$

$$\texttt{shape-match}(T, \forall\overline{a}.S \rightarrow R, \Theta) \ = \ \texttt{false} \ \ \textbf{if}\ T \neq \forall\overline{a}.S' \rightarrow R'$$

$$\texttt{shape-match}(\forall\overline{a}.A \rightarrow B, \forall\overline{a}.C \rightarrow D, \Theta) =$$

$$\begin{cases} \texttt{shape-match}(A,\ C,\ \texttt{tail}(\Theta)) & \textbf{if} \quad \texttt{head}(\Theta) \ = \ [\phi_{\texttt{fun-param}}] \\ \texttt{shape-match}(B,\ D,\ \texttt{tail}(\Theta)) & \textbf{if} \quad \texttt{head}(\Theta) \ = \ [\phi_{\texttt{fun-res}}] \\ \texttt{shape-match}(A,\ C,\ [\,]) \wedge \texttt{shape-match}(B,\ D,\ [\,]) & \textbf{else} \end{cases}$$

$$\texttt{shape-match}(T, \{x_1 : S_1,\ \ldots,\ x_n : S_n\}, \Theta) \ = \ \texttt{false} \ \ \textbf{if}\ T \neq \{x_1 : R_1,\ \ldots,\ x_n : R_n\}$$

$$\texttt{shape-match}(\{x_1 : T_1,\ \ldots,\ x_n : T_n\}, \{x_1 : S_1,\ \ldots,\ x_n : S_n\}, \Theta) =$$

$$\begin{cases} \texttt{shape-match}(T_i, S_i, \texttt{tail}(\Theta)) & \textbf{if} \quad \begin{array}{l} \texttt{head}(\Theta) \ = \ [\phi_{\texttt{sel}_\texttt{i}}] \\ \textbf{and}\ 1 \leq i \leq n \end{array} \\ \texttt{shape-match}(T_1, S_1, [\,]) \wedge \ldots \wedge \texttt{shape-match}(T_n, S_n, [\,]) & \textbf{else} \end{cases}$$

$\texttt{head} : \Theta \rightarrow \Theta$

$$\begin{array}{lcl} \texttt{head}([\,]) & = & [\,] \\ \texttt{head}(\phi_{\texttt{fun-param}} :: \Theta') & = & [\phi_{\texttt{fun-param}}] \\ \texttt{head}(\phi_{\texttt{fun-res}} :: \Theta') & = & [\phi_{\texttt{fun-res}}] \\ \texttt{head}(\phi_{\texttt{sel}_\texttt{i}} :: \Theta') & = & [\phi_{\texttt{sel}_\texttt{i}}] \ \text{for any } i \end{array}$$

$\texttt{tail} : \Theta \rightarrow \Theta$

$$\begin{array}{lcl} \texttt{tail}([\,]) & = & [\,] \\ \texttt{tail}(\phi_{\texttt{fun-param}} :: \Theta') & = & \Theta' \\ \texttt{tail}(\phi_{\texttt{fun-res}} :: \Theta') & = & \Theta' \\ \texttt{tail}(\phi_{\texttt{sel}_\texttt{i}} :: \Theta') & = & \Theta' \ \text{for any } i \end{array}$$

$\texttt{normalize} : (\Theta,\ P,\ \overline{a}) \rightarrow \Theta$

$$\texttt{normalize}([\,],\ T,\ \overline{a}) \quad = \quad [\,]$$

$$\texttt{normalize}(\Theta,\ T,\ \overline{a}) \ \ \textbf{if}\ \Theta \neq [\,] \quad =$$

$$\textbf{case}\ \ (\texttt{head}(\Theta))(T)\ \ \textbf{of}$$

$$\left| \begin{array}{lcl} \texttt{inl}\ T' & \Rightarrow & \texttt{head}(\Theta)\ ::: \ \texttt{normalize}(\texttt{tail}(\Theta), T', \overline{a}) \\ \texttt{inr}\ \langle T',\ \Theta' \rangle\ \textbf{if}\ \texttt{is-tvar}(T',\ \overline{a}) & \Rightarrow & \Theta \\ \texttt{inr}\ \langle T',\ \Theta' \rangle\ \textbf{if}\ \texttt{is-hole}(T') & \Rightarrow & \Theta \\ \texttt{inr}\ \langle T',\ \Theta' \rangle\ \textbf{else} & \Rightarrow & [\,] \end{array} \right.$$

$\texttt{prefix} : (\Theta, \Theta) \rightarrow \texttt{Bool}$

$$\texttt{prefix}(\Theta^1, \Theta^2) \quad = \begin{cases} \textbf{true} & \textbf{if} \quad \texttt{head}(\Theta^1)\ ==\ [\,] \\ \texttt{prefix}(\texttt{tail}(\Theta^1), \texttt{tail}(\Theta^2)) & \textbf{if} \quad \texttt{head}(\Theta^1)\ ==\ \texttt{head}(\Theta^2)\ \neq\ [\,] \\ \textbf{false} & \textbf{else} \end{cases}$$

Figure B.2: (Part 2) Complete implementations of the auxiliary functions introduced in Section 3.5.

B.1 Definition of the *free variables* function

The fv function extracts the set of *free variables* from the types of the core language.

$$\texttt{fv} : T \to \overline{a}$$

$$\texttt{fv}(T) = \texttt{fv}_{aux}(T, \emptyset)$$

$$\texttt{fv}_{aux} : (T, \overline{a}) \to \overline{a}$$

$$\begin{array}{lcll}
\texttt{fv}_{aux}(b,\ \overline{a}) & = & \emptyset & \text{if } b \in \overline{a} \\
\texttt{fv}_{aux}(b,\ \overline{a}) & = & \{\ b\ \} & \text{if } b \notin \overline{a} \\
\texttt{fv}_{aux}(\top,\ \overline{a}) & = & \emptyset & \\
\texttt{fv}_{aux}(\bot,\ \overline{a}) & = & \emptyset & \\
\texttt{fv}_{aux}(\forall \overline{b}.S \to U, \overline{a}) & = & \texttt{fv}_{aux}(S,\ \overline{a} \cup \{\ b\ \}) \cup \texttt{fv}_{aux}(U,\ \overline{a} \cup \{\ b\ \}) & \\
\texttt{fv}_{aux}(\{x_1 : P_1,\ \dots,\ x_n : P_n\},\ \overline{a}) & = & \texttt{fv}_{aux}(P_1,\ \overline{a}) \cup \dots \cup \texttt{fv}_{aux}(P_n,\ \overline{a}) &
\end{array}$$

Figure B.3: Definition of the fv function.

B.2 Definition of the *bound variables* function

The bv function extracts the set of *bound variables* from the types of the core language.

$$\texttt{bv} : T \to \overline{a}$$

$$\begin{array}{lcl}
\texttt{bv}(b) & = & \emptyset \\
\texttt{bv}(\top) & = & \emptyset \\
\texttt{bv}(\bot) & = & \emptyset \\
\texttt{bv}(\forall \overline{b}.S \to U)\overline{a} & = & \{\ b\ \} \cup \texttt{bv}(S) \cup \texttt{bv}(U) \\
\texttt{bv}(\{x_1 : P_1,\ \dots,\ x_n : P_n\}) & = & \texttt{bv}(P_1) \cup \dots \cup \texttt{bv}(P_n)
\end{array}$$

Figure B.4: Definition of the bv() function.

Appendix C

Proof of Theorem 1 on the prototype propagation

In Section 3.1.2 we have defined the implications of propagating partial type information along the adjacent nodes of the type derivation tree. In the process we have also identified Theorem 1 which states that for every type inference judgment of a form $(P, \ \Gamma \vdash^{w} E : \ T)$ for any P, Γ, E and T where $P \neq ?$, we can identify the source of the prototype by traversing towards the root of the type derivation tree.

Proof.
The proof of Theorem 1 follows directly from the definition of type inference rules for Colored Local Type Inference. which propagate parts of the prototype in the conclusion of the rule to its premises.
The proof relies on the fact that the *root* of any type derivation tree, $(P, \epsilon \vdash^{w} E : T)$, has a prototype $P = \ ?$. If the *root* of the type derivation tree was given a prototype P such that $P \neq \ ?$, then it can always be translated to the former one through an equivalent function application encoding, $(?, \ \epsilon \vdash^{w} (\texttt{fun}[\overline{a}](x : P)x)E : T)$ such that each ? in P is substituted with a fresh type variable from $\overline{a}$.

We prove the Theorem by induction on the type inference rule of the parent of the $\vdash^{w}$ judgment.

- **Case** $(\texttt{abs}_{tp,?})$: The only premise of the rule has $P = \ ?$, by contradiction trivially satisfied.
- **Case** $(\texttt{abs}_{tp,\top})$: The only premise of the rule, $(\top, \Gamma, \overline{a}, x : T \vdash^{w} E : S)$, has its prototype $\top$ propagated from the parent. Trivially satisfied.
- **Case** $(\texttt{abs}_{tp})$: The only premise of the rule, $(P', \ \Gamma, \overline{a}, x : T \vdash^{w} E : S)$, has its prototype propagated from the rule because P' is part of the prototype in the conclusion of the rule: $\forall \overline{a}.P \rightarrow P'$.

- **Case** `(abs)` : - The only premise of the rule, $(P, \Gamma, \overline{a}, x : T \vdash^{w} E : S)$, has its prototype propagated from the rule because P is part of the prototype in the conclusion of the rule: $\forall \overline{a}.T \to P$.
- **Case** `(app`$_{tp}$`)` : - The first premise of the rule, $(?, \Gamma \vdash^{w} F : \forall \overline{a}.S \to T)$, has a wildcard constant type as a prototype, therefore, by contradiction, it is trivially satisfied. The second premise of the rule $[\overline{R}/\overline{a}]\, S, \Gamma \vdash^{w} E : [\overline{R}/\overline{a}]\, S$ has a prototype $[\overline{R}/\overline{a}]\, S$, where $[\overline{R}/\overline{a}]\, S$ is not part of the rule's prototype P. Since all elements of the premise's prototype, *i.e.,* S, $\overline{R}$ and $\overline{a}$, are introduced as part of the rule's typing decisions, then `(app`$_{tp}$`)` is a Propagation Root for prototype $[\overline{R}/\overline{a}]\, S$.
- **Case** `(app`$_{tp,\bot}$`)` : - The first premise of the rule, $(?, \Gamma, \vdash^{w} F : \bot)$, has a wildcard constant type as a prototype, therefore, by contradiction, it is trivially satisfied. The second premise of the rule, $(\top, \Gamma \vdash^{w} E : S)$, has a non-? prototype which is not simply propagated from the prototype of the parent of the premise. While the $\top$ prototype is not directly part of any other type decision of the rule, by definition of rule `(app`$_{tp,\bot}$`)` it is indirectly implied by the inferred type of the function, $\bot$. Hence, `(app`$_{tp,\bot}$`)` is a Propagation Root.
- **Case** `(app)` : - Satisfied by the same argument as for rule `(app`$_{tp}$`)`.
- **Case** `(app`$_{\bot}$`)` : - Satisfied by the same argument as for rule `(app`$_{tp,\bot}$`)`.
- **Case** `(sel)` : - The premise of the rule, $(\{x : P\}, \Gamma, \vdash^{w} F : \{x : T\})$, uses the $\{x : P\}$ prototype. The record type is first introduced by the type inference judgment, therefore `(sel)` is trivially the Propagation Root.
- **Case** `(rec`$_{?}$`)` : - The only premise of the rule has $P =\ ?$, by contradiction, trivially satisfied.
- **Case** `(rec`$_{\top}$`)` : - Satisfied by the same argument as for rule `(abs`$_{tp,\top}$`)`.
- **Case** `(rec)` : - If the considered premise of the rule is $(P_k, \Gamma \vdash^{w} F_k : T_k)$, where $1 \le k \le m$, then the prototype of the premise, P_k, is simply propagated from the parent because P_k is part of the rule's prototype $\{x_1 : P_1, ..., x_m : P_m\}$. If the premise of the rule is $(\top, \Gamma \vdash^{w} F_k : \top)$, then `(rec)` becomes the Propagation Root because $\top$ is not part of the rule's prototype $\{x_1 : P_1, ..., x_m : P_m\}$.

$\square$

Appendix D

Proofs for the *TypeFocus* properties

The *TypeFocus* instance that is well-formed with respect to some type can be safely applied to the type to extract its type elements. The well-formed *TypeFocus* instance also offers a number of important properties that makes it particularly suitable for driving the navigation through the type derivation tree.

D.1 Proof of Lemma 3.4 on the well-formedness of *TypeFocus* with respect to the prototype

Lemma 3.4 states that a *TypeFocus* value that is well-formed with respect to the inferred type, then also has to be well-formed with respect to the prototype that helped to infer it:

If $(P, \Gamma \vdash^{w} E : T)$ and $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ for $\mathtt{fv}(T) \subseteq \overline{a}$, then $(\Theta, \overline{a} \vdash^{\mathtt{WF}} P)$.

Proof.
The proof for Lemma 3.4 follows directly from the invariant of Colored Local Type Inference in [Odersky et al., 2001, 8.1], which states that the inferred type in the type inference judgment has to match the shape of the prototype with respect to structural adaptation of the $\nearrow$ operation.

The proof follows by induction on the structure of the *TypeFocus* instance. We also make use of the well-formedness inversion lemma in Lemma 3.2. We use a $T_{\nearrow}$ notation for the result of type adaptation $T \nearrow P$, and $\overline{fv}$ for $\mathtt{fv}(T \nearrow P)$.

- **Case** $[\,]$: Since $[\,]$ returns a left tagged value for any type or prototype, the lemma is trivially satisfied.
- **Case** $\phi :: \Theta'$:
  We consider the individual type selectors first and later the tail of Θ.

- **Case** $\phi_{\texttt{fun-param}}$: If $(\phi_{\texttt{fun-param}}, \overline{fv} \vdash^{\texttt{WF}} T_{\nearrow})$, then:
  * **Case** $\phi_{\texttt{fun-param}}(T_{\nearrow}) = \texttt{inl}\ T'$ for some T'.
    By the inversion on the $\phi_{\texttt{fun-param}}$ definition, $T_{\nearrow}$ is of shape $\forall\overline{a}.T'_{\nearrow} \to T''_{\nearrow}$. Hence, from the invariant of Colored Local Type Inference either $P = \forall\overline{a}.P' \to P''$, for some P' and P'', and $(\phi_{\texttt{fun-param}}, \overline{fv} \vdash^{\texttt{WF}} P')$, or $P =\ ?$, meaning that $(\phi_{\texttt{fun-param}}, \overline{fv} \vdash^{\texttt{WF}} P)$.
  * **Case** $\phi_{\texttt{fun-param}}(T_{\nearrow}) = \texttt{inr}\ \langle T', \Theta' \rangle$.
    By the inversion Lemma, T' is a type variable. Since T has to be of the same structural shape as P, the application of the same *Type-Focus* will extract the corresponding part of P. Therefore from the invariant of Colored Local Type Inference $P =\ ?$ or $P =\ T'$.
- **Case** $\phi_{\texttt{fun-res}}$ :: Analogous argument as for $\phi_{\texttt{fun-param}}$.
- **Case** $\phi_{\texttt{sel}_x}$ :: Analogous argument as for $\phi_{\texttt{fun-param}}$.

From the assumption we know that $(\phi :: \Theta', \overline{fv} \vdash^{\texttt{WF}} T_{\nearrow})$. Also from the definition of *TypeFocus*, $\phi :: \Theta'$ is equivalent to $[\phi]\ :::\ \Theta'$ for any Θ' and ϕ. Therefore for the application of $\phi :: \Theta$ to type $T_{\nearrow}$:

- **Case** $([\phi])(T_{\nearrow}) = \texttt{inl}\ T'_{\nearrow}$.
  By the application of $[\phi]$ to P:
  * $([\phi])(P)\ =\ \texttt{inl}\ P'$ for some P' -
    By the invariant of Colored Local Type Inference an application of the same ϕ to the type and the prototype yields the corresponding type selection. Therefore P' becomes a prototype corresponding to the selected type $T'_{\nearrow}$. From the assumption, we know that $(\Theta', \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} T'_{\nearrow})$.
    By I.H. we have that $(\Theta', \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} P')$. Since P' is constructed from a well-formed selection of $[\phi]$, we have that $((\phi :: \Theta'), \overline{fv} \vdash^{\texttt{WF}} P)$.
  * $([\phi])(P)\ =\ \texttt{inr}\ \langle P'', [\phi] \rangle$ for some P'' -
    From the invariant of Colored Local Type Inference, $P'' =\ ?$ or $P'' \in \overline{fv}$, since a non-wildcard, non-type variable prototype has to be of the same shape as the inferred type. This in turn implies that $(\Theta', \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} P'')$ by the definition of the application of *TypeFocus* to a wildcard or a type variable, and $\phi :: \Theta', \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} P$.
- **Case** $([\phi])(T_{\nearrow}) = \texttt{inr}\ \left\langle T'_{\nearrow}, \Theta'' \right\rangle$.
  From the definition of the partial type selection we know that $T_{\nearrow}\ =\ T'_{\nearrow}$ and $T'_{\nearrow} \in \texttt{fv}(T_{\nearrow})$. By the application of $[\phi]$ to P:
  * $\texttt{inl}\ P'$ -
    By the invariant of Colored Local Type Inference, where the shape of the prototype is the same as the shape of the inferred type, the application should have selected a right tagged value as well. Case is impossible.

- $\texttt{inr}\ \langle P'', [\phi] \rangle$ -
  By the invariant of Colored Local Type Inference and the inversion lemma, we know that $P'' = ?$ or $P'' \in \overline{fv}$. An application of *TypeFocus* to a wildcard type or a type variable always yields the same type and *TypeFocus* in a right tagged value. Hence, $(\Theta', \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} P'')$. Since P'' is constructed from a well-formed type selection of $[\phi]$ we have that $((\phi :: \Theta'), \texttt{fv}(T_{\nearrow}) \vdash^{\texttt{WF}} P)$.

$\square$

D.2 Proof of the distribution of well-formedness over `head` and `tail` of *TypeFocus*

Lemma D.1 *The distribution of well-formedness of TypeFocus selection over* `head` *and* `tail` For any *TypeFocus* Θ and any type T, if $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$, then $(\Theta^{\mathsf{head}}, \overline{a} \vdash^{\mathtt{WF}} T)$ and $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T')$, where $\mathtt{head}(\Theta) = \Theta^{\mathsf{head}}$, $\Theta^{\mathsf{head}}(T)_{\mathtt{tpe}} = T'$ and $\mathtt{fv}(T) \subseteq \overline{a}$.

In order to prove the distribution of well-formedness we will first have to state one straightforward technical lemma. Lemma D.2 establishes that if the `head` of any *TypeFocus* returns an empty *TypeFocus*, it implies that the input *TypeFocus* is an identity [ ] *TypeFocus* as well.

Lemma D.2 `head` *of the identity TypeFocus.*
For any *TypeFocus* Θ, if $\mathtt{head}(\Theta) = [\,]$ then Θ is $[\,]$.

Proof.
Straightforward. From the specification of the `head` function. □

Proof of Lemma D.1.

Proof.
Proof by induction on the structure of T:

Case $T = \bot$:
By canonical forms, $\mathtt{head}(\Theta) = [\,]$. From the definition of well-formedness $\mathtt{head}(\Theta), \overline{a} \vdash^{\mathtt{WF}} \bot$. By Lemma D.2, $\Theta = [\,]$. Therefore $[\,](\bot) = \mathtt{inl}\ \bot$, and $\mathtt{tail}(\Theta) = \mathtt{tail}([\,]) = [\,]$. Therefore $\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} \bot$.

Case $T = \top$:
Analogous argument as in the case of type $\bot$.

Case $T = \forall \overline{b}.T_1 \to T_2$:
By canonical forms lemma $\mathtt{head}(\Theta)$ can be any of the three cases:

- **Case** $[\,]$ -
  From the definition of $[\,]$, $\forall T''.\ [\,](T'') = \mathtt{inl}\ T''$, therefore $[\,](\forall \overline{b}.T_1 \to T_2) = \mathtt{inl}\ \forall \overline{b}.T_1 \to T_2$.
  From the definition of `tail`, $\mathtt{tail}(\Theta) = \mathtt{tail}([\,]) = [\,]$.

Therefore $(\Theta^{\text{head}}, \overline{a} \vdash^{\mathtt{WF}} T)$ and $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T)$, from the definition of well-formedness property.

- **Case** $[\phi_{\mathtt{fun\text{-}param}}]$ -
  By definition of $[\phi_{\mathtt{fun\text{-}param}}]$, $\mathtt{head}(\Theta)(T) = \mathtt{inl}\ T_1$.
  By definition of composition of *TypeFocus* instances, $\mathtt{tail}(\Theta)$ is only applied to a left tagged value. Therefore the $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ assumption and Definition 8 imply $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T_1)$.
- **Case** $[\phi_{\mathtt{fun\text{-}res}}]$ -
  By definition of $[\phi_{\mathtt{fun\text{-}res}}]$, $\mathtt{head}(\Theta)(T) = \mathtt{inl}\ T_2$.
  By definition of composition of *TypeFocus* instances, $\mathtt{tail}(\Theta)$ is only applied to a left tagged value. Therefore the $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ assumption and Definition 8 imply $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T_2)$.

Case $T = \{x_1 : T_1, \dots, x_n : T_n\}$:
(Using the analogous argument as in the case of the polymorphic function type)
By canonical forms, $\mathtt{head}(\Theta)$ can be any of the two cases:

- **Case** $[\,]$ - From the definition of $[\,]$, $\forall T''.\ [\,](T'') = \mathtt{inl}\ T''$, therefore $[\,](\{x_1 : T_1, \dots, x_n : T_n\}) = \mathtt{inl}\ \{x_1 : T_1, \dots, x_n : T_n\}$.
  By Lemma D.2, $\mathtt{head}(\Theta) = [\,]$ implies $\Theta = [\,]$, and $\mathtt{tail}(\Theta) = [\,]$. Therefore $(\Theta^{\text{head}}, \overline{a} \vdash^{\mathtt{WF}} T)$ and $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T)$, from the definition of well-formedness property.
- **Case** $\phi_{\mathtt{sel}_{x_i}}$ for $1 \le i \le n$-
  By definition of $[\phi_{\mathtt{sel}_{x_i}}]$, $\mathtt{head}(\Theta)(T) = \mathtt{inl}\ T_i$.
  By definition of composition of *TypeFocus* instances, $\mathtt{tail}(\Theta)$ is only applied to a left tagged value. Therefore the $(\Theta, \overline{a} \vdash^{\mathtt{WF}} T)$ assumption and Definition 8 imply $\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T_i$.

Case $T = a$:
From the definition of the well-formedness property $\forall \Theta'.\ (\Theta', \overline{a} \vdash^{\mathtt{WF}} a)$ if $a \in \overline{a}$.
From the assumption $a \in \overline{a}$. Therefore $(\mathtt{head}(\Theta), \overline{a} \vdash^{\mathtt{WF}} a)$ and $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} a)$.

□

D.3 Proof of Lemma 3.8 on the well-formedness of *TypeFocus* over type substitution.

Well-formedness of TypeFocus over type substitution

For any *TypeFocus* Θ, and any type T, such that $(\Theta, \emptyset \vdash^{\mathtt{WF}} T)$, if T results from a type substitution, σ, on some type S, such that $T = \sigma S$ and $dom(\sigma) = \overline{a}$, then $(\Theta, \overline{a} \vdash^{\mathtt{WF}} S)$.

Proof.
Proof by induction on the structure of T:

Case $T = \sigma\bot$:
Since $\sigma\bot = \bot$, the result is immediate.

Case $T = \sigma\top$:
Since $\sigma\top = \top$, the result is immediate.

Case $T = \sigma(\forall\overline{b}.T_1 \to T_2)$:
From the definition of σ, $\sigma(\forall\overline{b}.T_1 \to T_2) = \forall\overline{b}.\sigma T_1 \to \sigma T_2$. By canonical forms, $\mathtt{head}(\Theta)$ can be any of the three cases:

- **Case** $[\,]$ -
  By definition of $[\,]$, we known that $[\,](\forall\overline{b}.\sigma T_1 \to \sigma T_2) = \mathtt{inl}\ \forall\overline{b}.\sigma T_1 \to \sigma T_2$.
  By Lemma D.2, $\mathtt{head}(\Theta) = [\,]$ implies $\Theta = [\,]$ and $\mathtt{tail}(\Theta) = [\,]$.
  From the definition of the well-formedness property $(\Theta, \overline{a} \vdash^{\mathtt{WF}} \forall\overline{b}.T_1 \to T_2)$ for any $\overline{a}$, because $[\,](\forall\overline{b}.T_1 \to T_2) = \mathtt{inl}\ \forall\overline{b}.T_1 \to T_2$.
- **Case** $[\phi_{\mathtt{fun\text{-}param}}]$ -
  By definition of $[\phi_{\mathtt{fun\text{-}param}}]$, $\mathtt{head}(\Theta)(T) = \mathtt{inl}\ \sigma T_1$. By definition of composition of *TypeFocus* instances, $\mathtt{tail}(\Theta)$ is only applied to a left tagged value, as shown.
  Assumption $(\Theta, \emptyset \vdash^{\mathtt{WF}} \sigma(\forall\overline{b}.T_1 \to T_2))$ implies $(\Theta, \emptyset \vdash^{\mathtt{WF}} \forall\overline{b}.\sigma T_1 \to \sigma T_2)$ which implies $(\mathtt{tail}(\Theta), \emptyset \vdash^{\mathtt{WF}} \sigma T_1)$.
  By I.H., $(\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T_1)$.
  By definition of $[\phi_{\mathtt{fun\text{-}param}}]$, $\forall S.\ [\phi_{\mathtt{fun\text{-}param}}](\forall\overline{b}.T_1 \to S) = \mathtt{inl}\ T_1$, including when $S = T_2$, and $((\mathtt{head}(\Theta)\ ::: \ \mathtt{tail}(\Theta)), \overline{a} \vdash^{\mathtt{WF}} \forall\overline{b}.T_1 \to T_2)$.
  By Definition 8 and Lemma D.1, we have that $(\Theta, \overline{a} \vdash^{\mathtt{WF}} \forall\overline{b}.T_1 \to T_2)$.
- **Case** $[\phi_{\mathtt{fun\text{-}res}}]$ -
  By definition of $[\phi_{\mathtt{fun\text{-}res}}]$, $\mathtt{head}(\Theta)(T) = \mathtt{inl}\ \sigma T_2$. By definition of composition of *TypeFocus* instances, $\mathtt{tail}(\Theta)$ is only applied to a left tagged value, as shown.
  Assumption $(\Theta, \emptyset \vdash^{\mathtt{WF}} \sigma\forall\overline{b}.T_1 \to T_2)$ implies $(\Theta, \emptyset \vdash^{\mathtt{WF}} \forall\overline{b}.\sigma T_1 \to \sigma T_2)$ which implies $(\mathtt{tail}(\Theta), \emptyset \vdash^{\mathtt{WF}} \sigma T_2)$.
  By I.H., $\mathtt{tail}(\Theta), \overline{a} \vdash^{\mathtt{WF}} T_2$.

By definition of $[\phi_{\texttt{fun-param}}]$, $\forall S.\ [\phi_{\texttt{fun-param}}](\forall \overline{b}.S \to T_2) = \texttt{inl}\ T_1$, including when $S = T_1$, and $((\texttt{head}(\Theta)\ :::\ \texttt{tail}(\Theta)), \overline{a} \vdash^{\texttt{WF}} \forall \overline{b}.T_1 \to T_2)$.
By Definition 8 and Lemma D.1, we have that $(\Theta, \overline{a} \vdash^{\texttt{WF}} \forall \overline{b}.T_1 \to T_2)$.

Case $T = \sigma(\{x_1 : T_1,\ \dots,\ x_n : T_n\})$:
(Using the analogous argument as in the case of the polymorphic function type)
From the definition of σ, $\sigma(\{x_1 : T_1,\ \dots,\ x_n : T_n\}) = \{x_1 : \sigma T_1,\ \dots,\ x_n : \sigma T_n\}$. By canonical forms, $\texttt{head}(\Theta)$ can be any of the two cases:

- **Case** $[\,]$ -
  By definition of $[\,]$, we known that $[\,](\{x_1 : \sigma T_1,\ \dots,\ x_n : \sigma T_n\}) = \texttt{inl}\ \{x_1 : \sigma T_1,\ \dots,\ x_n : \sigma T_n\}$.
  By Lemma D.2, $\texttt{head}(\Theta) = [\,]$ implies $\Theta = [\,]$ and $\texttt{tail}(\Theta) = [\,]$.
  Therefore, directly from the definition of the well-formedness property $(\Theta, \overline{a} \vdash^{\texttt{WF}} \{x_1 : T_1,\ \dots,\ x_n : T_n\})$ for any $\overline{a}$ because $[\,](\{x_1 : T_1,\ \dots,\ x_n : T_n\}) = \texttt{inl}\ \{x_1 : T_1,\ \dots,\ x_n : T_n\}$.
- **Case** $[\phi_{\texttt{sel}_{x_i}}]$ for $1 \leq i \leq n$-
  By definition of $[\phi_{\texttt{sel}_{x_i}}]$, $\texttt{head}(\Theta)(T) = \texttt{inl}\ \sigma T_i$. By definition of composition of *TypeFocus* instances, $\texttt{tail}(\Theta)$ is only applied to a left tagged value, as shown. Assumption $(\Theta, \emptyset \vdash^{\texttt{WF}} \sigma(\{x_1 : T_1,\ \dots,\ x_n : T_n\}))$ implies $(\Theta, \emptyset \vdash^{\texttt{WF}} \{x_1 : \sigma T_1,\ \dots,\ x_n : \sigma T_n\})$, which in turn implies $(\texttt{tail}(\Theta), \emptyset \vdash^{\texttt{WF}} \sigma T_i)$.
  By induction hypothesis, $(\texttt{tail}(\Theta), \overline{a} \vdash^{\texttt{WF}} T_i$.
  By definition of $[\phi_{\texttt{sel}_{x_i}}]$, $[\phi_{\texttt{sel}_{x_i}}](\{x_1 : S_1,\ \dots, x_i : S_i,\ \dots,\ x_n : S_n\}) = \texttt{inl}\ S_i$ for any $S_1,\ \dots,\ S_{i-1},\ S_{i+1},\ \dots,\ S_n$, including $T_1,\ \dots,\ T_{i-1},\ T_{i+1},\ \dots,\ T_n$, respectively. Therefore, $(\texttt{head}(\Theta)\ :::\ \texttt{tail}(\Theta), \overline{a} \vdash^{\texttt{WF}} \{x_1 : T_1,\ \dots,\ x_n : T_n\})$.
  By Definition 8 and Lemma D.1, we have that $(\Theta, \overline{a} \vdash^{\texttt{WF}} \{x_1 : T_1,\ \dots,\ x_n : T_n\})$.

Case $T = \sigma a$:
By definition of the well-formedness property in $(\Theta, \emptyset \vdash^{\texttt{WF}} \sigma a)$, $(\Theta, dom(\sigma) \vdash^{\texttt{WF}} a)$.
Since $dom(\sigma) = \overline{a}$ and $a \in dom(\sigma)$, we have immediately that $(\Theta, \overline{a} \vdash^{\texttt{WF}} a)$. □

Appendix E

Proofs on the translation of type constraints to type selectors

In Section 3.7.1 we define the translation of type constrains that are inferred from a subtype relation between some types S and T, $(W \vdash_{\overline{a}} S <: T \Rightarrow C)$, to sequences of *TypeFocus*, defined as $(a_i,\ \psi_{\pm}, W \vdash_{gen}\ S <:\ T \ \rightsquigarrow\ \overline{\Theta})$. The inferred *TypeFocus* instances extract individual type bounds for the type variable a_i such that $a_i \in \overline{a}$.

A number of properties of the translation ensures that the inferred *TypeFocus* instances can faithfully represent type constraints that are used in the inference of type variable substitution.

E.1 Proof of the well-formedness of the *TypeFocus* sequences inferred from the subtyping relation

The rules that govern the translation, ensure that every individual *TypeFocus* can be applied to the types that participate in the subtyping relation and extract either a type bound or a type variable, as formally defined in Lemma 3.9.

We provide an additional, technical lemma that will be used in the main proof.

Lemma E.1 *Lack of TypeFocus instances from type variable-free types.*
Let a be some type variable, V a set of bounded type variables, such that $a \not\subseteq V$, and some types S and T. If $a \not\subseteq (\mathtt{fv}_{aux}(S, V) \cup \mathtt{fv}_{aux}(T, V))$ then $(a,\ \psi_{\pm}, V \vdash_{gen}\ S <:\ T \ \rightsquigarrow\ \overline{\Theta})$, where $\overline{\Theta} = \epsilon$.

Proof.
Straightforward, by induction on the last $\Theta\mathtt{G}$ rule used. □

Proof of Lemma 3.9.

Proof.
By induction on the last $\Theta\mathtt{G}$ rule used:

- **Case** $\Theta\mathtt{G}_{(-,<)}$: We have to consider only a single *TypeFocus*, $[\,]$, since $a,\ - \vdash_{gen} a <:\ T \rightsquigarrow \{[\,]\}$. From the definition of $[\,]$, $\forall S.\ [\,](S) = \mathtt{inl}\ S$. Therefore $[\,](a) = \mathtt{inl}\ a$ and $[\,](T) = \mathtt{inl}\ T$ and $a \in \overline{a}$.
- **Case** $\Theta\mathtt{G}_{(+,<)}$: Since $\overline{\Theta} = \epsilon$, the result is immediate.
- **Case** $\Theta\mathtt{G}_{(-,>)}$: Since $\overline{\Theta} = \epsilon$, the result is immediate.
- **Case** $\Theta\mathtt{G}_{(+,>)}$: We have to consider only a single *TypeFocus*, $[\,]$, since $a,\ - \vdash_{gen} T <:\ a \rightsquigarrow \{[\,]\}$. From the definition of $[\,]$, $\forall S.\ [\,](S) = \mathtt{inl}\ S$. Therefore $[\,](T) = \mathtt{inl}\ T$ and $[\,](a) = \mathtt{inl}\ a$ and $a \in \overline{a}$.
- **Case** $\Theta\mathtt{G}_{\emptyset}$: Since $\overline{\Theta} = \epsilon$, the result is immediate.
- **Case** $\Theta\mathtt{G}_{(\mathrm{TOP})}$: Since $\overline{\Theta} = \epsilon$, the result is immediate.
- **Case** $\Theta\mathtt{G}_{(\mathrm{BOT})}$: Since $\overline{\Theta} = \epsilon$, the result is immediate.
- **Case** $\Theta\mathtt{G}_{\mathrm{FUN}}$: By the definition of the rule, we will consider its two premises separately:
  - $a,\ \psi_{\pm}, W \vdash_{gen}\ T <:\ R \rightsquigarrow \overline{\Theta}'$ -
    If $a \not\subseteq (\mathtt{fv}_{aux}(T, W \cup \overline{b}) \cap \mathtt{fv}_{aux}(T, W \cup \overline{b}))$, then by Lemma E.1, $\overline{\Theta}' = \epsilon$ and the result is immediate.
    Otherwise, by I.H.,
    $\forall \Theta^{\mathsf{j}}.\exists T'.\exists R'.\ \Theta^{\mathsf{j}} \in \overline{\Theta}' \implies$
    $\quad\Theta^{\mathsf{j}}(T) = \mathtt{inl}\ T' \ \wedge\ \Theta^{\mathsf{j}}(R) = \mathtt{inl}\ R' \ \wedge\ (T' = a \vee S' = a)$.
    From the definition of the application of $[\phi_{\mathtt{fun\text{-}param}}]$,
    $\forall T.\ [\phi_{\mathtt{fun\text{-}param}}](\forall \overline{b}.S \to T) = \mathtt{inl}\ S$.
    By the application of the *TypeFocus* composition
    $\forall \Theta^{\mathsf{j}}.\exists T'.\exists R'.\ \Theta^{\mathsf{j}} \in \overline{\Theta}' \implies$
    $\quad((\phi_{\mathtt{fun\text{-}param}} :: \Theta^{\mathsf{j}})(\forall \overline{a}.R \to S) = \Theta^{\mathsf{j}}(R) = \mathtt{inl}\ R'$
    $\quad\quad\wedge\ (\phi_{\mathtt{fun\text{-}param}} :: \Theta^{\mathsf{j}})(\forall \overline{a}.T \to U) = \Theta^{\mathsf{j}}(T) = \mathtt{inl}\ T')$
    as required for the particular subcase.
  - $a,\ \psi_{\pm},\ W \vdash_{gen}\ S <:\ U \rightsquigarrow \overline{\Theta}''$ -
    If $a \not\subseteq (\mathtt{fv}_{aux}(S, W \cup \overline{b}) \cap \mathtt{fv}_{aux}(U, W \cup \overline{b}))$, then by Lemma E.1, $\overline{\Theta}'' = \epsilon$ and the result is immediate.

Otherwise, by I.H.,
$\forall \Theta^j.\exists S'.\exists U'.\ \Theta^j \in \overline{\Theta}'' \implies$
$\Theta^j(S) = \mathtt{inl}\ S' \wedge \Theta^j(U) = \mathtt{inl}\ U' \wedge (S' = a \vee U' = a)$.
From the definition of the application of $[\phi_{\mathtt{fun\text{-}res}}]$,
$\forall S.\ [\phi_{\mathtt{fun\text{-}res}}](\forall \overline{b}.S \to T) = \mathtt{inl}\ T$.
By the application of the *TypeFocus* composition, similarly to the previous subcase we have that
$\forall \Theta^j.\exists S'.\exists U'.\ \Theta^j \in \overline{\Theta}'' \implies$
$((\phi_{\mathtt{fun\text{-}res}} :: \Theta^j)(\forall \overline{a}.R \to S) = \Theta^j(S) = \mathtt{inl}\ S'$
$\wedge\ (\phi_{\mathtt{fun\text{-}res}} :: \Theta^j)(\forall \overline{a}.T \to U) = \Theta^j(U) = \mathtt{inl}\ U')$
as required for the particular subcase.

Since the *head* of each of the instances of the $\left\{ \phi_{\mathtt{fun\text{-}param}} :: \overline{\Theta}' \right\}$ and $\left\{ \phi_{\mathtt{fun\text{-}res}} :: \overline{\Theta}'' \right\}$ sequences ensures that respective parameter and result type elements of the function type are extracted, the case condition is satisfied for the considered rule.

- **Case** $\Theta\mathsf{G}_{\text{REC}}$:
  - Base case (for $m = 0$):
    $(a,\ \psi_{\pm} \vdash_{gen} \{x_1 : S_1,\ ...,\ x_n : S_n\}$ `<: {}` $\leadsto \overline{\Theta})$. Trivially satisfied, since $\overline{\Theta} = \epsilon$.
  - For $1 \le k \le m$, where $m \ge 1$:
    If $a \not\in (\mathtt{fv}_{aux}(S_k, W) \cap \mathtt{fv}_{aux}(T_k, W))$, then by Lemma E.1, $\overline{\Theta}^k = \epsilon$ and the result is immediate.
    Otherwise, by I.H.,
    $\forall \Theta^j.\exists S'.\exists T'.\ \Theta^j \in \overline{\Theta}^k \implies$
    $\Theta^j(S_k) = \mathtt{inl}\ S' \wedge \Theta^j(T_k) = \mathtt{inl}\ T' \wedge (S' = a \vee T' = a)$.
    By definition of $[\phi_{\mathtt{sel}_{x_i}}]$, $[\phi_{\mathtt{sel}_{x_i}}](\{x_1 : C_1, \dots, x_i : C_i, \dots, x_n : C_n\}) = \mathtt{inl}\ C_i$ for all $C_1, \dots, C_{i-1}, C_i, C_{i+1}, \dots, C_n$.
    By the application of the *TypeFocus* composition
    $\forall \Theta^j.\exists S'.\exists U'.\ \Theta^j \in \overline{\Theta}^k \implies$
    $((\phi_{\mathtt{sel}_k} :: \Theta^j)(\{x_1 : T_1,\ ...,\ x_m : T_m,\ ...,\ x_n : T_n\}) = \Theta^j(T'_k) = \mathtt{inl}\ T_k$
    $\wedge\ ((\phi_{\mathtt{sel}_k} :: \Theta^j)(\{x_1 : S_1,\ ...,\ x_m : S_m\}) = \Theta^j(S_k) = \mathtt{inl}\ S'_k$
    as required for $1 \le k \le m$.

Since the *head* of each of the instances of the $\left\{ \phi_{\mathtt{sel}_{x_k}} :: \overline{\Theta}^k \right\}$ sequences ensures that respective type parts of the record type are extracted, the case condition is satisfied for the considered rule.

□

E.2 Proof of the soundess of the *TypeFocus* sequences inferred from the subtyping relation

Lemma 3.10 states a soundness property of the translation - the application of the inferred *TypeFocus* instances to the two types of the subtyping relation extracts lower and upper type bounds, respectively. Additionally, when approximated using the *least upper bound* and *greatest lower bound* operations, they infer type bounds that are identical to the ones that were inferred in the corresponding $\overline{a}$-constraint set.

The constraint generation rules in Pierce and Turner [2000] lack a formal rule that defines collecting type constraints between two record types in a subtyping relation. Neither does the formalization of the Colored Local Type Inference in Odersky et al. [2001] provide one. For reference, the `(CG-Rec)` rule below, implements the necessary constraint generation for such types without compromising the whole system.

$$\text{(CG-R}{\small\text{EC}}\text{)}\ \frac{\begin{array}{c} V \vdash_{\overline{x}} P_1 <: R_1 \Rightarrow C_1 \quad \dots \quad V \vdash_{\overline{x}} P_m <: R_m \Rightarrow C_m \\ \overline{x} \cap V = \emptyset \end{array}}{V \vdash_{\overline{x}} \{x_1 : P_1,\ \dots,\ x_m : P_m,\ \dots,\ x_n : P_n\} <: \{x_1 : R_1,\ \dots,\ x_m : R_m\} \Rightarrow C_1 \bigwedge \dots \bigwedge C_m}$$

We provide additional, technical lemmas that will be used in the main proof.

Lemma E.2 *Preservation of least upper bound and greatest lower bound under well-formed type selection.*
$\forall \Theta, \Theta', T, S.\ \Theta \in \overline{\Theta}\ \wedge\ (\Theta, \emptyset \vdash^{\mathtt{WF}} S)\ \wedge\ \Theta'(T) = \mathtt{inl}\ S \implies \bigvee\Theta(S) = \bigvee(\left\{ \Theta'\ :::\ \overline{\Theta} \right\})(T)$ and
$\forall \Theta, \Theta', T, S.\ \Theta \in \overline{\Theta}\ \wedge\ (\Theta, \emptyset \vdash^{\mathtt{WF}} S)\ \wedge\ \Theta'(T) = \mathtt{inl}\ S \implies \bigwedge\Theta(S) = \bigwedge(\left\{ \Theta'\ :::\ \overline{\Theta} \right\})(T)$.

Proof.
Straightforward. By induction on the *TypeFocus* structure. The $\left\{ \Theta'\ :::\ \overline{\Theta} \right\}$ notation is equivalent to $\left\{ \Theta'\ :::\ \Theta \right\}$ for all Θ' where $\Theta \in \overline{\Theta}$. □

Lemma E.3 *Lack of constraints for type variable-free types.*
Let S and T be some types, a_i be a type variable, $\overline{a}$ be a set of type variables, and W a set of bounded type variables, such that $\overline{a} \cap W = \emptyset$.
If $a_i \not\in (\mathtt{fv}_{aux}(S, W) \cup \mathtt{fv}_{aux}(T, W))$ then
$W\ \vdash_{\overline{a}} S <: T \Rightarrow C$ for some C, such that $\{\bot <:\ a_i <:\ \top\} \in C$.

Proof.
Straightforward. Induction on the `CG` constraint generation rules. In Pierce and Turner [2000] it is clearly stated that a type variable-free subtyping relation $S <: T$ resolves to subtype checking. $\square$

Proof of Lemma 3.10.

Proof.
By induction on the last inference rule used in the $(a_i, \psi_\pm, W \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$ judgment.

We note that the `CG` constraint generation rules keep track of the bounded type variables that can be *promoted*, $\Uparrow^W$, or *demoted*, $\Downarrow^W$. The corresponding rules of Θ`G` also carry the information about a set of bounded type variables, W. This allows for using the application of the $A \Uparrow^W B$ and $C \Downarrow^W D$ operations to implicitly *promote* and *demote bound* type variables in the *least upper bound* and *greatest lower bound* approximations.
Similarly to the formalization of Odersky et al. [2001], we assume that W is provided implicitly to both operations and the `fv` function (for example `fv`(S) means `fv`$_{aux}(S, W)$).

The soundness lemma is proved by proving the soundness of the translation for lower and upper bounds separately for each type variable a_i, where $a_i \in \overline{a}$.

Soundness of lower type bounds:

Case $\Theta\mathtt{G}_{(+,<)}$ a_i, +, $W \vdash_{gen}$ $a_i <: T \rightsquigarrow \epsilon$:
By definition of the constraint generation judgment, only the `(CG-Upper)` rule could apply for the $a_i <: T$ relation.
By definition of the `(CG-Upper)` rule, $W \vdash_{\overline{a}} a_i <: T \Rightarrow C$ and $\{\bot <: a_i <: R\} \in C$.

Since, `fv`$_{aux}(a_i, W) \cap \overline{a} \neq \emptyset$, the subcase is trivially satisfied.
If `fv`$_{aux}(T, W) \cap \overline{a} = \emptyset$, then $\bigvee \overline{\Theta}(T) = \bot$ by definition of the $\bigvee$ approximation for $\overline{\Theta} = \epsilon$.

Case $\Theta\mathtt{G}_{(+,>)}$ a_i, +, $W \vdash_{gen}$ $T <: a_i \rightsquigarrow \{[\,]\}$:
By definition of the constraint generation judgment, only the `(CG-Lower)` rule could apply for the $T <: a_i$ relation.
By definition of the `(CG-Lower)` rule, $W \vdash_{\overline{a}} T <: a_i \Rightarrow C$ and $\{R <: a_i <: \top\} \in C$, where $T \Uparrow^w R$.

If `fv`$_{aux}(T, W) \cap \overline{a} = \emptyset$, then $\bigvee \overline{\Theta}(T) = [\,](T)_{\mathtt{tpe}} \Uparrow^W = R$.
Since, `fv`$_{aux}(a_i, W) \cap \overline{a} \neq \emptyset$, the subcase is trivially satisfied.

Case $\Theta\mathtt{G}_{(\text{TOP})}\ a_i,\ +,\ W \vdash_{gen} S <: \top \rightsquigarrow \epsilon$:
By definition of the constraint generation judgment, rules (CG-Top) and (CG-Upper) could apply for the $S <: \top$ relation.
By definition of the (CG-Top) rule, $(W \vdash_{\overline{a}} S <: \top \Rightarrow C)$ and $\{\bot <: a_i <: \top\} \in C$.

If $\mathtt{fv}_{aux}(S, W) \cap \overline{a} = \emptyset$, then $\bigvee\overline{\Theta}(S) = \bot$ by definition of the $\bigvee$ approximation for $\overline{\Theta} = \epsilon$.
Since, $\mathtt{fv}(\top) \cap \overline{a} = \emptyset$, the subcase is trivially satisfied by the same argument as the previous subcase.

By definition of the (CG-Upper) rule, $(W \vdash_{\overline{a}} S <: \top \Rightarrow C)$ for $\{\bot <: a_i <: T\} \in C$ and $T = \top$. The condition is trivially satisfied by the same argument as for the (CG-Top) rule.

Case $\Theta\mathtt{G}_{(\text{BOT})}\ a_i,\ +,\ W \vdash_{gen} \bot <: S \rightsquigarrow \epsilon$:
By definition of the constraint generation judgment, rules (CG-Bot) and (CG-Lower) could apply for the $\bot <: S$ relation.
By definition of the (CG-Bot) rule, $(W \vdash_{\overline{a}} \bot <: S \Rightarrow C)$ and $\{\bot <: a_i <: \top\} \in C$.

If $\mathtt{fv}_{aux}(S, W) \cap \overline{a} = \emptyset$, then $\bigvee\overline{\Theta}(S) = \bot$ by definition of the $\bigvee$ approximation for $\overline{\Theta} = \epsilon$.
Since, $\mathtt{fv}(\bot) \cap \overline{a} = \emptyset$, the subcase is trivially satisfied by the same argument as the previous subcase.

By definition of the (CG-Lower) rule, $(W \vdash_{\overline{a}} \bot <: S \Rightarrow C)$ for $\{T <: a_i <: \top\} \in C$ and $T = \bot$. The condition is trivially satisfied by the same argument as for the (CG-Bot) rule.

Case $\Theta\mathtt{G}_{(\emptyset)}\ a_i,\ +,\ W \vdash_{gen} S <: T \rightsquigarrow \epsilon$:
where $a_i \notin (\mathtt{fv}_{aux}(S, W) \cup \mathtt{fv}_{aux}(T, W))$.

By defininition of the constraint generation judgment, only rules (CG-Top), (CG-Bot), (CG-Refl), (CG-Fun) and (CG-Rec) could apply for the subtyping relation. Out of the 5 rules, only (CG-Fun) and (CG-Rec) could potentially return a non-default constraint set. By Lemma E.3, $\{\bot <: a_i <: \top\} \in C$ for all the rules.

The result from such default constraint set is immediate, using the analogous argument to the previous cases:

- If $\mathtt{fv}_{aux}(S, W) \cap \overline{a} = \emptyset$, then $\bigvee\overline{\Theta}(S) = \bot$ by definition of the $\bigvee$ approximation for $\overline{\Theta} = \epsilon$.
- If $\mathtt{fv}_{aux}(T, W) \cap \overline{a} = \emptyset$, then $\bigvee\overline{\Theta}(T) = \bot$ by definition of the $\bigvee$ approximation for $\overline{\Theta} = \epsilon$.

Case $\Theta\mathtt{G}_{(\text{FUN})}\ a_i,\ +, W \vdash_{gen} \forall\overline{b}.R \rightarrow S <: \forall\overline{b}.T \rightarrow U \rightsquigarrow \left\{\phi_{\mathtt{fun\text{-}param}} :: \overline{\Theta}'\right\} \cup \left\{\phi_{\mathtt{fun\text{-}res}} :: \overline{\Theta}''\right\}$:

where $a_i,\ +, W \cup \overline{b} \vdash_{gen}\ T <:\ R \rightsquigarrow \overline{\Theta}'$ and $a_i,\ +, W \cup \overline{b} \vdash_{gen}\ S <:\ U \rightsquigarrow \overline{\Theta}''$ and $a_i \in (\mathtt{fv}_{aux}(\forall \overline{b}.R \to S, W) \cup \mathtt{fv}_{aux}(\forall \overline{b}.T \to U, W))$.

By defininition of the constraint generation judgment, only the `(CG-Fun)` rule can apply, such that $(W \vdash_{\overline{a}} \forall \overline{b}.R \to S <: \forall \overline{b}.T \to U \Rightarrow C)$ where $\{A <:\ a_i <:\ B\} \in C$. From the premises of the `(CG-Fun)` rule:

- $(W \cup \overline{b} \vdash_{\overline{a}} T <: R \Rightarrow C')$ and $\{A' <:\ a_i <:\ B'\} \in C'$.
- $(W \cup \overline{b} \vdash_{\overline{a}} S <: U \Rightarrow C'')$ and $\{A'' <:\ a_i <:\ B''\} \in C''$.

Therefore, $\{A' \vee A'' <:\ a_i <:\ B' \wedge B''\} \in (C' \wedge C'') = C$.

By I.H. (twice) on the premises of the $\Theta\mathtt{G}_{(\text{FUN})}$ rule:

$$\begin{aligned}
&a_i,\ +, W \cup \overline{b} \vdash_{gen}\ T <:\ R \rightsquigarrow \overline{\Theta}' \implies\\
&W \cup \overline{b} \vdash_{\overline{a}} T <: R \Rightarrow C' \ \wedge\ \{A' <:\ a_i <:\ B'\} \in C' \ \wedge\\
&\quad (\mathtt{fv}_{aux}(T, W \cup \overline{b}) \cap \overline{a} = \emptyset \implies \bigvee \overline{\Theta}'(T) = A') \ \wedge\\
&\qquad\quad (\mathtt{fv}_{aux}(R, W \cup \overline{b}) = \emptyset \implies \bigvee \overline{\Theta}'(R) = A')
\end{aligned}$$

$$\begin{aligned}
&a_i,\ +, W \cup \overline{b} \vdash_{gen}\ S <:\ U \rightsquigarrow \overline{\Theta}'' \implies\\
&W \cup \overline{b} \vdash_{\overline{a}} S <: U \Rightarrow C'' \ \wedge\ \{A'' <:\ a_i <:\ B''\} \in C'' \ \wedge\\
&\quad (\mathtt{fv}_{aux}(S, W \cup \overline{b}) \cap \overline{a} = \emptyset \implies \bigvee \overline{\Theta}''(S) = A'') \ \wedge\\
&\qquad\quad (\mathtt{fv}_{aux}(U, W \cup \overline{b}) = \emptyset \implies \bigvee \overline{\Theta}''(U) = A'')
\end{aligned}$$

From the precondition of $(a_i,\ \psi_{\pm} \vdash_{gen}\ \forall \overline{b}.R \to S <:\ \forall \overline{b}.T \to U \rightsquigarrow \overline{\Theta})$, we know that either $\mathtt{fv}(\forall \overline{b}.R \to S) \cap \overline{a} = \emptyset$ or $\mathtt{fv}(\forall \overline{b}.T \to U) \cap \overline{a} = \emptyset$, as well as $\mathtt{fv}(\forall \overline{b}.R \to S) \cap \overline{a} = \emptyset$ and $\mathtt{fv}(\forall \overline{b}.T \to U) \cap \overline{a} = \emptyset$. By Lemma E.3, the latter case leads to $\{\bot <:\ a_i <:\ \top\} \in C$, which is trivially satisfied.

Therefore, we consider the two subcase where

- either $\mathtt{fv}_{aux}(\forall \overline{b}.R \to S, W) \cap \overline{a} = \emptyset$
- or $\mathtt{fv}_{aux}(\forall \overline{b}.T \to U, W) \cap \overline{a} = \emptyset$.

Furthermore, by a property of *free variables*:

$$\begin{aligned}
&\forall A, B.\ \mathtt{fv}_{aux}(\forall \overline{a}.A \to B, W) = \emptyset \iff\\
&\qquad\qquad \mathtt{fv}_{aux}(A, W \cup \overline{a}) \cap \overline{a} = \emptyset \ \wedge\ \mathtt{fv}_{aux}(B, W \cup \overline{a}) \cap \overline{a} = \emptyset
\end{aligned}$$

we have that

$$\mathtt{fv}_{aux}(\forall \overline{b}.R \to S, W) \cap \overline{a} = \emptyset \implies \mathtt{fv}_{aux}(R, W \cup \overline{b}) \cap \overline{a} = \emptyset \wedge \mathtt{fv}_{aux}(S, W \cup \overline{b}) \cap \overline{a} = \emptyset$$
$$\mathtt{fv}_{aux}(\forall \overline{b}.T \to U, W) \cap \overline{a} = \emptyset \implies \mathtt{fv}_{aux}(T, W \cup \overline{b}) \cap \overline{a} = \emptyset \wedge \mathtt{fv}_{aux}(U, W \cup \overline{b}) \cap \overline{a} = \emptyset$$

Subcase $\mathtt{fv}_{aux}(\forall \overline{b}.R \to S, W) \cap \overline{a} = \emptyset$:

By Lemma E.2,
$\bigvee \overline{\Theta}'(R) = A' \implies \bigvee \left\{ \phi_{\mathtt{fun\text{-}param}} :: \overline{\Theta}' \right\} (\forall \overline{b}.R \to X) = A'$ for any type X. Similarly,

$\bigvee\overline{\Theta}'(S) = A'' \implies \bigvee\left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.Y \to S) = A''$ for any type Y.
In particular, for $X = S$ and $Y = R$:

- $\bigvee\overline{\Theta}'(R) = A' \implies \bigvee\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\}(\forall\overline{b}.R \to S) = A'$.
- $\bigvee\overline{\Theta}'(S) = A'' \implies \bigvee\left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.R \to S) = A''$.

$A' \vee A'' = \bigvee\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\}(\forall\overline{b}.R \to S) \vee \bigvee\left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.R \to S)$. By the disjoint type selection of $\phi_{\texttt{fun-param}}$ and $\phi_{\texttt{fun-res}}$ the latter is equivalent to
$\bigvee(\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\} \cup \left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\})(\forall\overline{b}.R \to S) = A$,
as required for the $\texttt{fv}_{aux}(\forall\overline{b}.R \to S, W) \cap \overline{a} = \emptyset$ case.

Subcase $\texttt{fv}_{aux}(\forall\overline{b}.T \to U, W) \cap \overline{a} = \emptyset$:

By the analogous argument as for $\texttt{fv}_{aux}(\forall\overline{b}.R \to S, W) \cap \overline{a} = \emptyset$ *subcase:*
$\bigvee\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\}(\forall\overline{b}.T \to U) = A'$ and $\bigvee\left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.T \to U) = A''$ and
$A' \vee A'' = \bigvee\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\}(\forall\overline{b}.T \to U) \vee \bigvee\left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.T \to U) =$
$\bigvee\left\{\phi_{\texttt{fun-param}} :: \overline{\Theta}'\right\} \cup \left\{\phi_{\texttt{fun-res}} :: \overline{\Theta}'\right\}(\forall\overline{b}.T \to U) = A$.

Since by the previous argument only the two options had to be considered, the case for the $\Theta\texttt{G}_{(\text{FUN})}$ rule is satisfied.

Case $\Theta\texttt{G}_{(\text{REC})}$ $a_i, +, W \vdash_{gen} \{x_1 : P_1, ..., x_m : P_m, ..., x_n : P_n\}$ <:
$\{x_1 : R_1, ..., x_m : R_m\} \rightsquigarrow \left\{\phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1\right\} \cup ... \cup \left\{\phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m\right\}$:
where $a_i, +, W \vdash_{gen} P_1 <: R_1 \rightsquigarrow \overline{\Theta}^1$ and ... and $a_i, +, W \vdash_{gen} P_m <: R_m \rightsquigarrow \overline{\Theta}^m$ and $a_i \in (\texttt{fv}_{aux}(\{x_1 : P_1, ..., x_m : P_m, ..., x_n : P_n\}, W) \cup \texttt{fv}_{aux}(\{x_1 : R_1, ..., x_m : R_m\}, W))$.

By defininition of the constraint generation judgment, only the `(CG-Rec)` rule can apply. By definition of the `(CG-Rec)` rule,
$(W \vdash_{\overline{a}} \{x_1 : P_1, ..., x_m : P_m, ..., x_n : P_n\} <: \{x_1 : R_1, ..., x_m : R_m\} \Rightarrow C)$ where $\{A <: a_i <: B\} \in C$.
From the premises of the rule
$(W \vdash_{\overline{a}} P_1 <: R_1 \Rightarrow C^1)$ where $\{A^1 <: a_i <: B^1\} \in C^1$ and ... and
$(W \vdash_{\overline{a}} P_m <: R_m \Rightarrow C^m)$ where $\{A^m <: a_i <: B^m\} \in C^m$.
Therefore, $\{A^1 \vee ... \vee A^m <: a_i <: B^1 \wedge ... \wedge B^m\} \in (C^1 \wedge ... \wedge C^m) = C$.

By I.H. on the premises of the $\Theta\texttt{G}_{(\text{REC})}$ rule for the record element x_k, where $1 \leq k \leq m$:

$$\begin{aligned}
&a_i, +, W \vdash_{gen} P_k <: R_k \rightsquigarrow \overline{\Theta}^k \implies \\
&\quad (W \vdash_{\overline{a}} P_k <: R_k \Rightarrow C^k) \wedge \{A^k <: a_i <: B^k\} \in C^k \wedge \\
&\quad\; (\texttt{fv}_{aux}(P_k, W) \cap \overline{a} = \emptyset \implies \bigvee\overline{\Theta}^k(P_k) = A^k) \wedge \\
&\quad\quad (\texttt{fv}_{aux}(R_k, W) \cap \overline{a} = \emptyset \implies \bigvee\overline{\Theta}^k(R_k) = A^k)
\end{aligned}$$

From the pre-condition of $(a_i, \psi_\pm \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta})$, we know that either $\texttt{fv}(S) \cap \overline{a} = \emptyset$ or $\texttt{fv}(T) \cap \overline{a} = \emptyset$, as well as $\texttt{fv}(S) \cap \overline{a} = \emptyset$ and $\texttt{fv}(T) \cap \overline{a} = \emptyset$. By Lemma E.3, the latter case leads to $\{\bot <: a_i <: \top\} \in C$, which is trivially satisfied. Therefore, we describe in detail only the cases when:

- either $\texttt{fv}_{aux}(\{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\}, W) \cap \overline{a} = \emptyset$,
- or $\texttt{fv}_{aux}(\{x_1 : R_1, \ldots, x_m : R_m\}, W) \cap \overline{a} = \emptyset$.

By a property of *free variables*:
$\forall S, T.\ \texttt{fv}_{aux}(\{x_1 : S_1, \ldots, x_n : S_n\}, W) \cap \overline{a} = \emptyset \iff$
$\texttt{fv}_{aux}(S_1, W) \cap \overline{a} = \emptyset \wedge \ldots \wedge \texttt{fv}_{aux}(S_n, W) \cap \overline{a} = \emptyset$
we know that

- $\texttt{fv}_{aux}(\{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\}, W) \cap \overline{a} = \emptyset \implies$
  $(\forall k.\ 1 \le k \le n \implies \texttt{fv}_{aux}(P_k, W) \cap \overline{a} = \emptyset)$.
- $\texttt{fv}_{aux}(\{x_1 : R_1, \ldots, x_m : R_m\}, W) \cap \overline{a} = \emptyset \implies$
  $(\forall k.\ 1 \le k \le m \implies \texttt{fv}_{aux}(R_k, W) \cap \overline{a} = \emptyset)$.

Case $\texttt{fv}_{aux}(\{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\}, W) \cap \overline{a} = \emptyset$*:*

By Lemma E.2,
$\bigvee \overline{\Theta}^k(P_k) = A^k \implies \bigvee \left\{ \phi_{\texttt{sel}_{x_k}} :: \overline{\Theta}' \right\}(\{x_1 : P_1, \ldots, x_k : P_k, \ldots, x_m : P_m, \ldots, x_n : P_n\}) = A^k$.

For $\texttt{fv}_{aux}(S_k, W) \cap \overline{a} = \emptyset$, where $S_k = \{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\}$:
$\bigvee \left\{ \phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1 \right\}(S_1) = A^1$ and $\ldots$ and $\bigvee \left\{ \phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m \right\}(S_k) = A^m$ and
$A^1 \vee \ldots \vee A^m = \bigvee \left\{ \phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1 \right\}(S_k) \vee \ldots \vee \bigvee \left\{ \phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m \right\}(S_k) =$
$\bigvee \left\{ \phi_{\texttt{sel}_{x_1}} :: \overline{\Theta}^1 \right\} \cup \ldots \cup \left\{ \phi_{\texttt{sel}_{x_m}} :: \overline{\Theta}^m \right\}(S_k) = A.$

Case $\texttt{fv}_{aux}(\{x_1 : R_1, \ldots, x_m : R_m\}, W) \cap \overline{a} = \emptyset$*:*

Analogous argument as for the $\texttt{fv}_{aux}(\{x_1 : P_1, \ldots, x_m : P_m, \ldots, x_n : P_n\}, W) \cap \overline{a} = \emptyset$ *case.*

As required by the $\Theta\texttt{G}_{(\text{REC})}$ rule.

Soundness of upper type bounds:
(Omitted. Proof by the analogous argument as for the lower type bounds modulo the greatest lower bound approximation.) □

E.3 Proof of the completeness of the *TypeFocus* sequences inferred from the subtyping relation

Lemma 3.11 states a completness property for lower and upper bounds of the translation.

The completeness properly states that the inferred *TypeFocus* instances extract only those lower type bounds which are present in the $\overline{a}$-constraint set inferred from the same subtyping relation, modulo the *meet* operation involving the *least upper bound* approximation of the lower type bounds. Similarly the inferred *TypeFocus* instances extract only those upper type bounds which are present in the $\overline{a}$-constraint set inferred from the same subtyping relation, modulo the *meet* operation involving the *greatest lower bound* approximation of the upper type bounds.

We first provide a technical Lemma E.4 that will be used in the main proof.

Lemma E.4 *TypeFocus inclusion for the well-behaved, extended TypeFocus sequences.*
Let $\overline{\Theta}$ and $\overline{\Theta}'$ be two sequences of *TypeFocus* instances.
If $(\forall\Theta.\, \Theta \in \overline{\Theta} \implies \Theta \in \overline{\Theta}')$ then $(\forall\Theta,\Theta'.\, \Theta \in \left\{\Theta' :: \overline{\Theta}\right\} \implies \Theta \in \left\{\Theta' :: \overline{\Theta}'\right\})$

Proof.
Straightforward from the definition of *TypeFocus* composition. □

Proof of Lemma 3.11:

Proof.

We will define the proof for the analysis of the lower type bounds in the $(a_i, + \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta}^{+})$ implication. The case for upper type bounds in the $(a_i, - \vdash_{gen} S <: T \rightsquigarrow \overline{\Theta}^{-})$ implication is analogous, and omitted for space reasons.

By induction on the structure of S and T.

Case $S = a$ where $a \in \overline{a}$ and $a_i = a$:

- $T = b$ where $b \in \overline{a}$
  From the assumption that either $\mathtt{fv}(S) \cap \overline{a} = \emptyset$ or $\mathtt{fv}(T) \cap \overline{a} = \emptyset$, the case is invalid.

- $T = \bot$

  For $\texttt{fv}(a) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.
  For $\texttt{fv}(\bot) \cap \overline{a} = \emptyset$: by strict canonical forms $\forall \Theta'. \Theta'(a) = \texttt{inl}\ a \implies (\Theta' == [\,])$.
  Since $\texttt{pos}_a(a) = +$ we have a contradiction and the result for the case is immediate.

- $T = \top$

  Satisfied by the same argument as in the previous subcase.

- $T = \forall \overline{b}.P \to R$

  Satisfied by the same argument as in the previous subcase.

- $T = \{x_1 : P_1,\ ...,\ x_n : P_n\}$

  Satisfied by the same argument as in the previous subcase.

Case $S = a$ where $a \in \overline{a}$ and $a_i \neq a$:

- $T = b$ where $b \in \overline{a}$

  From the assumption that either $\texttt{fv}(S) \cap \overline{a} = \emptyset$ or $\texttt{fv}(T) \cap \overline{a} = \emptyset$, the case is invalid.

- $T = \bot$

  For $\texttt{fv}(a) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.
  For $\texttt{fv}(\bot) \cap \overline{a} = \emptyset$: since $\nexists \Theta'.\Theta'(\bot) = \texttt{inl}\ a_i$ the result for the case is immediate.

- $T = \top$

  Satisfied by the same argument as in the previous subcase.

- $T = \forall \overline{b}.P \to R$

  Satisfied by the same argument as in the previous subcase.

- $T = \{x_1 : P_1,\ ...,\ x_n : P_n\}$

  Satisfied by the same argument as in the previous subcase.

Case $S = \bot$:

- $T = b$ where $b \in \overline{a} \ \wedge\ a_i = b$

  For $\texttt{fv}(\bot) \cap \overline{a} = \emptyset$:
  By strict canonical forms $\forall \Theta'. \Theta'(b) = \texttt{inl}\ a_i \implies (\Theta' == [\,])$ and $\texttt{pos}_{a_i}(b) = +$.
  By the $\Theta\texttt{G}_{(+,>)}$ rule $\overline{\Theta}^{+} = \{[\,]\}$ and the result of the subcase is immediate ($\Theta' \in \overline{\Theta}^{+}$).
  For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = b$ where $b \in \overline{a} \ \wedge\ a_i \neq b$ For $\texttt{fv}(\bot) \cap \overline{a} = \emptyset$:
  since $\nexists \Theta'.\Theta'(T) = \texttt{inl}\ a_i$ the result for the case is immediate. For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = \bot$

  For both implications $\nexists\Theta'.\Theta'(S) = \texttt{inl}\ a_i$ and $\nexists\Theta'.\Theta'(T) = \texttt{inl}\ a_i$ and result for the case is immediate.

- $T = \top$

  Satisfied by the same argument as in the previous subcase.

- $T = \forall\overline{b}.P \to R$

  Satisfied by the same argument as in the previous subcase.

- $T = \{x_1 : P_1,\ ...,\ x_n : P_n\}$

  Satisfied by the same argument as in the previous subcase.

Case $S = \top$:

- $T = b$ where $b \in \overline{a}\ \wedge\ a_i = b$

  For $\texttt{fv}(\top) \cap \overline{a} = \emptyset$:
  By strict canonical forms $\forall\Theta'.\,\Theta'(b) = \texttt{inl}\ a_i \implies (\Theta'\ == [\,])$ and $\texttt{pos}_{a_i}(b) = +$.

  By the $\Theta\texttt{G}_{(+,\,>)}$ rule $\overline{\Theta}^{+} = \{[\,]\}$ and the result of the subcase is immediate ($\Theta' \in \overline{\Theta}^{+}$).

  For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = b$ where $b \in \overline{a}\ \wedge\ a_i \neq b$

  For $\texttt{fv}(\top) \cap \overline{a} = \emptyset$:
  since $\nexists\Theta'.\Theta'(T) = \texttt{inl}\ a_i$ the result for the case is immediate. For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = \bot$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

- $T = \top$

  For both implications $\nexists\Theta'.\Theta'(S) = \texttt{inl}\ a_i$ and $\nexists\Theta'.\Theta'(T) = \texttt{inl}\ a_i$ and result for the case is immediate.

- $T = \forall\overline{b}.P \to R$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

- $T = \{x_1 : P_1,\ ...,\ x_n : P_n\}$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

Case $S = \forall\overline{b}.P \to R$:

- $T = b$ where $b \in \overline{a}\ \wedge\ a_i = b$

  For $\texttt{fv}(\forall\overline{b}.P \to R) \cap \overline{a} = \emptyset$:
  By strict canonical forms $\forall\Theta'.\,\Theta'(b) = \texttt{inl}\ a_i \implies (\Theta'\ == [\,])$ and $\texttt{pos}_{a_i}(b) = +$.

By the $\Theta\mathtt{G}_{(+,\,>)}$ rule $\overline{\Theta}^{+} = \{\,[\,]\,\}$ and the result of the subcase is immediate ($\Theta' \in \overline{\Theta}^{+}$).

For $\mathtt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = b$ where $b \in \overline{a} \,\wedge\, a_i \neq b$

  For $\mathtt{fv}(\forall \overline{b}.P \to R) \cap \overline{a} = \emptyset$:
  since $\nexists\Theta'.\Theta'(T) = \mathtt{inl}\ a_i$ the result for the case is immediate. For $\mathtt{fv}(b) \cap \overline{a} \neq \emptyset$ the result for the case is immediate.

- $T = \bot$

  No $\Theta\mathtt{G}$ rule applies. The result is immediate.

- $T = \top$

  By definition of the $\phi_{\mathtt{fun\text{-}param}}$ and $\phi_{\mathtt{fun\text{-}res}}$ $\nexists\Theta'.\Theta'(S) = \mathtt{inl}\ a_i$ and $(\Theta', \emptyset \vdash^{\mathtt{WF}} \top)$ and $\nexists\Theta'.\Theta'(T) = \mathtt{inl}\ a_i$ and result for the case is immediate.

- $T = \forall \overline{b}.Q \to Z$

  For $(a_i, + \vdash_{gen}\ \forall \overline{b}.P \to R <:\ \forall \overline{b}.Q \to Z \rightsquigarrow \overline{\Theta}^{+})$.

  From the premises of the $\Theta\mathtt{G}_{(\text{FUN})}$ rule and by I.H. (twice):

  If $a_i, + \vdash_{gen}\ Q <:\ P \rightsquigarrow \overline{\Theta}^{+'}$ then

  By I.H.
  $(\mathtt{fv}(Q) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'.\ (\Theta'(P) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} Q)$ and $(\mathtt{pos}_{a_i}(P) = +) \implies \Theta' \in \overline{\Theta}^{+'})$
  and
  $(\mathtt{fv}(P) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'.\ (\Theta'(Q) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} P)$ and $(\mathtt{pos}_{a_i}(Q) = -) \implies \Theta' \in \overline{\Theta}^{+'})$

  If $a_i, + \vdash_{gen}\ R <:\ Z \rightsquigarrow \overline{\Theta}^{+''}$ then

  By I.H.
  $(\mathtt{fv}(R) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'.\ (\Theta'(Z) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} R)$ and $(\mathtt{pos}_{a_i}(Z) = +) \implies \Theta' \in \overline{\Theta}^{+''})$
  and
  $(\mathtt{fv}(Z) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'.\ (\Theta'(R) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} Z)$ and $(\mathtt{pos}_{a_i}(R) = -) \implies \Theta' \in \overline{\Theta}^{+''})$

  We will now consider the two (covariant) implications separately.

  The first implication.

  From the first implication we know that $\mathtt{fv}(\forall \overline{b}.P \to R) \cap \overline{a} = \emptyset$ and $\mathtt{fv}(\forall \overline{b}.Q \to Z) \cap \overline{a} \neq \emptyset$.

  It is now sufficient to consider just the three possible cases of the *free variables* sets, where the nested *TypeFocus* can extract type variable a_i:

  1. $\mathtt{fv}(Q) \cap \overline{a} \neq \emptyset$ and $\mathtt{fv}(Z) \cap \overline{a} = \emptyset$ and $\mathtt{fv}(P) \cap \overline{a} = \emptyset$ and $\mathtt{fv}(R) \cap \overline{a} = \emptyset$:

For $Q <: P$:
From the I.H. implication
$(\forall \Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$\Theta' \in \overline{\Theta}^{+'})$
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$

For $R <: Z$:
Since type a_i is not present in either R or Z,
$\nexists \Theta'.\Theta'(R) = \texttt{inl}\ a_i \vee \Theta'(Z) = \texttt{inl}\ a_i$ and $\overline{\Theta}^{+''} = \epsilon$.
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$
By combining the implications for the two subcases we have that
$\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = \overline{\Theta}^{+}$.

2. $\texttt{fv}(Q) \cap \overline{a} = \emptyset$ and $\texttt{fv}(Z) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(P) \cap \overline{a} = \emptyset$ and $\texttt{fv}(R) \cap \overline{a} = \emptyset$:

For $Q <: P$:
Since type a_i is not present in either Q or P,
$\nexists \Theta'.\Theta'(Q) = \texttt{inl}\ a_i \vee \Theta'(P) = \texttt{inl}\ a_i$ and $\overline{\Theta}^{+''} = \epsilon$.
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+''} \right\})$
For $R <: Z$:
From the I.H. implication
$(\forall \Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$\Theta' \in \overline{\Theta}^{+''})$
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

By combining the implications for the two subcases we have that
$\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = \overline{\Theta}^{+}$.

3. $\texttt{fv}(Q) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(Z) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(P) \cap \overline{a} = \emptyset$ and $\texttt{fv}(R) \cap \overline{a} = \emptyset$:

For $Q <: P$:
From the I.H. implication
$(\forall \Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$\Theta' \in \overline{\Theta}^{+'})$

By Lemma E.4,
$(\forall\Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$

For $R <: Z$:
From the I.H. implication
$(\forall\Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$\Theta' \in \overline{\Theta}^{+''})$
By Lemma E.4,
$(\forall\Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

By combining the implications for the two subcases we have that
$\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = \overline{\Theta}^{+}$.

By Canonical Forms Lemma and $\texttt{fv}(\forall\overline{b}.Q \to Z) \cap \overline{a} \neq \emptyset$:

$$\forall\Theta'.\ \Theta'(\forall\overline{b}.Q \to Z) = \texttt{inl}\ a_i \implies \texttt{head}(\Theta) = [\phi_{\texttt{fun-param}}] \lor \texttt{head}(\Theta) = [\phi_{\texttt{fun-res}}]$$

As a result, $\overline{\Theta}^{+}$ is equivalent to $\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\}$.

Since $\texttt{pos}_{a_i}(\forall\overline{b}.Q \to Z) = \texttt{+}$ iff $\texttt{pos}_{a_i}(Q) = \texttt{-}$ and $\texttt{pos}_{a_i}(Z) = \texttt{+}$ and we have shown that for all the possible combinations of the a_i type variable
$(\forall\Theta'.\ (\Theta'(Q) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} P)$ and $(\texttt{pos}_{a_i}(Q) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$ and
$(\forall\Theta'.\ (\Theta'(Z) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} R)$ and $(\texttt{pos}_{a_i}(Z) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

then for any types X and Y
$(\forall\Theta'.\ ((\phi_{\texttt{fun-param}} :: \Theta')(\forall\overline{b}.Q \to X) = \texttt{inl}\ a_i)$ and
$((\phi_{\texttt{fun-param}} :: \Theta'), \epsilon \vdash^{\texttt{WF}} \forall\overline{b}.P \to Y)$ and $(\texttt{pos}_{a_i}(Q \to X) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$
and for any types X and Y
$(\forall\Theta'.\ ((\phi_{\texttt{fun-res}} :: \Theta')(\forall\overline{b}.X \to Z) = \texttt{inl}\ a_i)$ and
$((\phi_{\texttt{fun-res}} :: \Theta'), \epsilon \vdash^{\texttt{WF}} \forall\overline{b}.Y \to R)$ and $(\texttt{pos}_{a_i}(\forall\overline{b}.X \to Z) = \texttt{+}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

as required for the first implication.

The second implication.

(By the analogous argument as in the previous case)

From the first implication we know that $\texttt{fv}(\forall\overline{b}.P \to R) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(\forall\overline{b}.Q \to Z) \cap \overline{a} = \emptyset$.

It is now sufficient to consider just the three possible cases of the *free variables* sets, where the nested *TypeFocus* can extract type variable a_i:

1. $\texttt{fv}(Q) \cap \overline{a} = \emptyset$ and $\texttt{fv}(Z) \cap \overline{a} = \emptyset$ and $\texttt{fv}(P) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(R) \cap \overline{a} = \emptyset$:

   For $Q <: P$:
   From the I.H. implication
   $(\forall\Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies \Theta' \in \overline{\Theta}^{+'})$
   By Lemma E.4,
   $(\forall\Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies (\phi_{\texttt{fun-param}} :: \Theta')) \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$

   For $R <: Z$:
   Since type a_i is not present in either R or Z,
   $\nexists\Theta'.\Theta'(R) = \texttt{inl}\ a_i \vee \Theta'(Z) = \texttt{inl}\ a_i$ and $\overline{\Theta}^{+''} = \epsilon$.
   By Lemma E.4,
   $(\forall\Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies (\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

   By combining the implications for the two subcases we have that
   $\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = \overline{\Theta}^{+}$.

2. $\texttt{fv}(Q) \cap \overline{a} = \emptyset$ and $\texttt{fv}(Z) \cap \overline{a} = \emptyset$ and $\texttt{fv}(P) \cap \overline{a} = \emptyset$ and $\texttt{fv}(R) \cap \overline{a} \neq \emptyset$:

   For $Q <: P$:
   Since type a_i is not present in either Q or P,
   $\nexists\Theta'.\Theta'(Q) = \texttt{inl}\ a_i \vee \Theta'(P) = \texttt{inl}\ a_i$ and $\overline{\Theta}^{+''} = \epsilon$.
   By Lemma E.4,
   $(\forall\Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies$
   $(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+''} \right\})$
   For $R <: Z$:
   From the I.H. implication
   $(\forall\Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies$
   $\Theta' \in \overline{\Theta}^{+''})$
   By Lemma E.4,
   $(\forall\Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies$
   $(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

By combining the implications for the two subcases we have that
$\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = \overline{\Theta}^{+}$.

3. $\texttt{fv}(Q) \cap \overline{a} = \emptyset$ and $\texttt{fv}(Z) \cap \overline{a} = \emptyset$ and $\texttt{fv}(P) \cap \overline{a} \neq \emptyset$ and $\texttt{fv}(R) \cap \overline{a} \neq \emptyset$:

For $Q <: P$:
From the I.H. implication
$(\forall \Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies \Theta' \in \overline{\Theta}^{+'})$
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies (\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$

For $R <: Z$:
From the I.H. implication
$(\forall \Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies \Theta' \in \overline{\Theta}^{+''})$
By Lemma E.4,
$(\forall \Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies (\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

By combining the implications for the two subcases we have that
$\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\} = -\Theta^{+}$.

By Canonical Forms Lemma and $\texttt{fv}(\forall \overline{b}.P \to R) \cap \overline{a} \neq \emptyset$:

$$\forall \Theta'.\ \Theta'(\forall \overline{b}.P \to R) = \texttt{inl}\ a_i \implies \texttt{head}(\Theta) = [\phi_{\texttt{fun-param}}] \lor \texttt{head}(\Theta) = [\phi_{\texttt{fun-res}}]$$

As a result, $\overline{\Theta}^{+}$ is equivalent to $\left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\} \cup \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\}$, where $\overline{\Theta}^{+'}$ and $\overline{\Theta}^{+''}$ result from the application of the `tail` function by Definition 8.

Since $\texttt{pos}_{a_i}(\forall \overline{b}.P \to R) = -$ iff $\texttt{pos}_{a_i}(P) = +$ and $\texttt{pos}_{a_i}(R) = -$ and we have shown that for all the possible combinations of the a_i type variable
$(\forall \Theta'.\ (\Theta'(P) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Q)$ and $(\texttt{pos}_{a_i}(P) = +) \implies (\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$
and
$(\forall \Theta'.\ (\Theta'(R) = \texttt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\texttt{WF}} Z)$ and $(\texttt{pos}_{a_i}(R) = -) \implies (\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

Then for all types X and Y
$(\forall\Theta'.\,((\phi_{\texttt{fun-param}} :: \Theta')(\forall\overline{b}.P \to X) = \texttt{inl}\ a_i)$ and
$((\phi_{\texttt{fun-param}} :: \Theta'), \epsilon \vdash^{\texttt{WF}} \forall\overline{b}.Q \to Y)$ and $(\texttt{pos}_{a_i}(P \to X) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-param}} :: \Theta') \in \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta}^{+'} \right\})$
and for all types X and Y
$(\forall\Theta'.\,((\phi_{\texttt{fun-res}} :: \Theta')(\forall\overline{b}.X \to R) = \texttt{inl}\ a_i)$ and
$((\phi_{\texttt{fun-res}} :: \Theta'), \epsilon \vdash^{\texttt{WF}} \forall\overline{b}.Y \to Z)$ and $(\texttt{pos}_{a_i}(\forall\overline{b}.X \to R) = \texttt{-}) \implies$
$(\phi_{\texttt{fun-res}} :: \Theta') \in \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta}^{+''} \right\})$

as required for the second implication.

We note that for $\texttt{fv}_{aux}(\forall\overline{b}.P \to R, W) \cap \overline{a} = \emptyset$ and $\texttt{fv}_{aux}(\forall\overline{b}.Q \to Z, W) \cap \overline{a} = \emptyset$, the case is trivial since $\forall\Theta'.\,\Theta' \in \overline{\Theta} \implies \Theta'(S) = \texttt{inl}\ a_i \iff \overline{\Theta} = \epsilon$.
Then the case is satisfied immediately for $\overline{\Theta} = \epsilon$.

- $T = \{x_1 : P_1,\ \ldots,\ x_n : P_n\}$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

Case $S = \{x_1 : P_1,\ \ldots,\ x_m : P_m,\ x_n : P_n\}$:

- $T = b$ where $b \in \overline{a} \wedge a_i = b$

  For $\texttt{fv}(\{x_1 : P_1,\ \ldots,\ x_m : P_m,\ x_n : P_n\}) \cap \overline{a} = \emptyset$:
  By strict canonical forms $\forall\Theta'.\,\Theta'(b) = \texttt{inl}\ a_i \implies (\Theta'\ \texttt{==}\ [\,])$ and $\texttt{pos}_{a_i}(b) = \texttt{+}$.

  By the $\Theta\texttt{G}_{(+,\,>)}$ rule $\overline{\Theta}^{+} = \{\,[\,]\,\}$ and the result of the subcase is immediate $(\Theta' \in \overline{\Theta}^{+})$.

  For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$: the result for the case is immediate.

- $T = b$ where $b \in \overline{a} \wedge a_i \neq b$

  For $\texttt{fv}(\{x_1 : P_1,\ \ldots,\ x_m : P_m,\ x_n : P_n\}) \cap \overline{a} = \emptyset$:
  since $\nexists\Theta'.\Theta'(T) = \texttt{inl}\ a_i$ the result for the case is immediate.
  For $\texttt{fv}(b) \cap \overline{a} \neq \emptyset$ the result for the case is immediate.

- $T = \bot$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

- $T = \top$

  By definition of the $\phi_{\texttt{sel}_{x_k}}$ $\nexists\Theta'.\Theta'(S) = \texttt{inl}\ a_i$ and $(\Theta', \emptyset \vdash^{\texttt{WF}} \top)$ and $\nexists\Theta'.\Theta'(T) = \texttt{inl}\ a_i$ and result for the case is immediate.

- $T = \forall\overline{b}.Q \to Z$

  No $\Theta\texttt{G}$ rule applies. The result is immediate.

- $T = \{x_1 : R_1, ..., x_m : R_m\}$

  For $a_i, + \vdash_{gen} \{x_1 : P_1, ..., x_m : P_m, x_n : P_n\} <: \{x_1 : R_1, ..., x_m : R_m\} \rightsquigarrow \overline{\Theta}^{+}$.

  From the premises of the $\Theta\mathtt{G}_{(\text{REC})}$ rule and by I.H. (m-times):

  For $1 \leq k \leq m$:
  If $a_i, + \vdash_{gen} P_k <: R_k \rightsquigarrow \overline{\Theta}^{+'}$ then

  By I.H.
  $(\mathtt{fv}(P_k) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'. (\Theta'(R_k) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} P_k)$ and $(\mathtt{pos}_{a_i}(R_k) = \mathtt{+}) \implies \Theta' \in \overline{\Theta}^{+'})$
  and
  $(\mathtt{fv}(R_k) \cap \overline{a} = \emptyset) \implies$
  $(\forall \Theta'. (\Theta'(P_k) = \mathtt{inl}\ a_i)$ and $(\Theta', \epsilon \vdash^{\mathtt{WF}} R_k)$ and $(\mathtt{pos}_{a_i}(P_k) = \mathtt{-}) \implies \Theta' \in \overline{\Theta}^{+'})$

  The proof for the case follows the same argument as for the two polymorphic function types and is omitted for brevity.

□

E.4 Proof of the completeness of the $\vdash^{sub}$ judgment

Proof of Lemma 4.3:

Proof.
By induction (twice) on the structure of types S and T. The proof omits an extension of the rules for the wildcard constant type for simplicity; the grayed-out conditions in the ΘSub rules apply to this proof in a straightforward manner.

Case $S = a$:

- $T = a$: Since $a <: a$ by the `(Var)` rule, the result is immediate.
- $T = \bot$: Only the rule $\Theta Sub_{(var1)}$ applies, and by definition of $[\,]$, $[\,](S) =$ `inl` a and $[\,](T) =$ `inl` $\bot$ and $a \not<: \bot$.
- $T = \top$: By the `(Top)` rule, the result is immediate.
- $T = \forall a.\ A \to B$: Only the rule $\Theta Sub_{(var1)}$ applies, and the argument is analogous as for $T = \bot$.
- $T = \{x_1 : P_1,\ \ldots,\ x_m : P_m\}$: Only the rule $\Theta Sub_{(var1)}$ applies, and the argument is analogous as for $T = \bot$.

Case $S = \bot$:
By the `(Bot)` rule, $\bot <: T$ for any T, and the result is immediate.

Case $S = \top$:

- $T = a$: Only the $\Theta Sub_{(var2)}$ applies, and by definition of $[\,]$, $[\,](S) =$ `inl` $\top$ and $[\,](T) =$ `inl` a, and $\top \not<: a$.
- $T = \bot$: Only $\Theta Sub_{(top-bot)}$ applies, and by definition of $[\,]$, $[\,](S) =$ `inl` $\top$ and $[\,](T) =$ `inl` $\bot$, and $\top \not<: \bot$.
- $T = \top$: Since $\top$ `<:` $\top$ by the `(Top)`, the result is immediate.
- $T = \forall \overline{a}.A \to B$: Only the rule $\Theta Sub_{(fun1)}$ applies, and by definition of $[\,]$, $[\,](S) =$ `inl` $\top$ and $[\,](T) =$ `inl` $\forall \overline{a}.A \to B$, and $\top \not<: \forall \overline{a}.A \to B$.
- $T = \{x_1 : P_1,\ \ldots,\ x_m : P_m\}$: Only the rule $\Theta Sub_{(rec2)}$ applies, and the argument is analogous as for the previous case.

Case $S = \forall \overline{a}.A \to B$:

- $T = a$: Only the rule $\Theta Sub_{(fun2)}$ applies, and by definition of $[\,]$, $[\,](S) =$ `inl` $\forall \overline{a}.A \to B$ and $[\,](T) =$ `inl` a, and $\forall \overline{a}.A \to B \not<: a$.

- $T = \bot$: Only the rule $\Theta Sub_{(fun2)}$ applies, and the argument is analogous as for the $T = a$ case.
- $T = \top$: By the `(Top)` rule, the result is immediate.
- $T = \forall a.\ C \to D$: Only the rule $\Theta Sub_{(fun)}$ applies.
  $\forall \overline{a}.A \to B \not<: \forall \overline{a}.C \to D$ implies that either $C \not<: A$ or $B \not<: D$, by definition of the `(Fun)` rule.
  By I.H. (twice) on the parameter and result type of the function.
  - For $\vdash^{sub} C \not<: A \to \overline{\Theta}$:
    By I.H.
    $(\forall \Theta.\ \Theta \in \overline{\Theta} \implies$
    $\exists C', A'.\ \Theta(C) = \texttt{inl}\ C' \wedge \Theta(A) = \texttt{inl}\ A' \ \wedge\ (C' \not<: A' \ \vee\ A' \not<: C'))$.

    By definition of $[\phi_{\texttt{fun-param}}]$,
    $\forall \Theta.\ \Theta \in \overline{\Theta} \implies$
    $\exists C', A'.\ (\phi_{\texttt{fun-param}} :: \Theta)(\forall \overline{a}.C \to X) = \texttt{inl}\ C' \wedge$
    $(\phi_{\texttt{fun-param}} :: \Theta)(\forall \overline{a}.A \to Y) = \texttt{inl}\ A'$
    for any types X and Y.
    In particular, for $X = B$ and $Y = D$ and by definition of *TypeFocus* composition, where $\overline{\Theta}' = \left\{ \phi_{\texttt{fun-param}} :: \overline{\Theta} \right\}$,
    $\vdash^{sub} \forall \overline{a}.A \to B \not<: \forall \overline{a}.C \to D \to \overline{\Theta}' \implies$
    $(\forall \Theta.\ \Theta \in \overline{\Theta}' \implies$
    $\exists C', A'.\ \Theta(\forall \overline{a}.A \to B) = \texttt{inl}\ A' \wedge \Theta(\forall \overline{a}.C \to D) = \texttt{inl}\ C' \wedge$
    $(C' \not<: A' \ \vee\ A' \not<: C')$
  - For $\vdash^{sub} B \not<: D \to \overline{\Theta}$:
    By I.H.
    $(\forall \Theta.\ \Theta \in \overline{\Theta} \implies$
    $\exists B', D'.\ \Theta(B) = \texttt{inl}\ B' \wedge \Theta(D) = \texttt{inl}\ D' \ \wedge\ (B' \not<: D' \ \vee\ D' \not<: B'))$.

    By definition of $[\phi_{\texttt{fun-res}}]$,
    $\forall \Theta.\ \Theta \in \overline{\Theta} \implies$
    $\exists B', D'.\ (\phi_{\texttt{fun-res}} :: \Theta)(\forall \overline{a}.X \to B) = \texttt{inl}\ B' \wedge$
    $(\phi_{\texttt{fun-res}} :: \Theta)(\forall \overline{a}.Y \to D) = \texttt{inl}\ D'$
    for any types X and Y.
    In particular, for $X = A$ and $Y = C$ and by definition of *TypeFocus* composition, where $\overline{\Theta}'' = \left\{ \phi_{\texttt{fun-res}} :: \overline{\Theta} \right\}$,
    $\vdash^{sub} \forall \overline{a}.A \to B \not<: \forall \overline{a}.C \to D \to \overline{\Theta}'' \implies$
    $(\forall \Theta.\ \Theta \in \overline{\Theta}'' \implies$
    $\exists B', D'.\ \Theta(\forall \overline{a}.A \to B) = \texttt{inl}\ B' \wedge \Theta(\forall \overline{a}.C \to D) = \texttt{inl}\ D' \wedge$
    $(B' \not<: D' \ \vee\ D' \not<: B')$

  By the definition of `head`, $\overline{\Theta}'$ and $\overline{\Theta}''$ extract distinct type elements of the $\forall \overline{a}.A \to B$ and $\forall \overline{a}.C \to D$ function types, and their composition makes the result immediate.

- $T = \{x_1 : P_1, \dots, x_m : P_m\}$: Only the rule $\Theta Sub_{(fun2)}$ applies, and the argument is analogous as for the $T = a$ case.

Case $S = \{x_1 : P_1, \dots, x_m : P_m, \dots, x_n : P_n\}$:

- $T = a$: Only the rule $\Theta Sub_{(rec1)}$ applies, and by definition of $[\,]$, $[\,](S) = \texttt{inl}\ \{x_1 : P_1, \dots, x_m : P_m, \dots, x_n : P_n\}$ and $[\,](T) = \texttt{inl}\ a$, and $\{x_1 : P_1, \dots, x_m : P_m, \dots, x_n : P_n\} \not<: a$.
- $T = \bot$: Only the rule $\Theta Sub_{(rec1)}$ applies, and the argument is analogous as for the $T = a$ case.
- $T = \top$: Since $\{x_1 : P_1, \dots, x_m : P_m, \dots, x_n : P_n\} <: \top$ Only $\Theta Sub_{(top)}$, the result is immediate.
- $T = \forall a.\ A \to B$: Only the rule $\Theta Sub_{(rec1)}$ applies, and the argument is analogous as for the $T = a$ case.
- $T = \{x_1 : R_1, \dots, x_k : R_k\}$ where $k > n$: Only the $\Theta Sub_{(rec1)}$ applies, and the argument is analogous as for the $T = a$ case.
- $T = \{x_1 : R_1, \dots, x_m : R_m\}$ where $n \geq m$:
  - For $m = 0$: The condition trivially holds for $\overline{\Theta} = \epsilon$.
  - For i, where $1 \leq i \leq m$:
    $\{x_1 : P_1, \dots, x_m : P_m, \dots, x_n : P_n\} \not<: \{x_1 : R_1, \dots, x_m : R_m\}$ implies that for at least one record member $P_i \not<: R_i$, by definition of the `(Rec)` rule.
    By I.H. and the definition of $[\phi_{\mathtt{sel}_{x_i}}]$,
    $\forall \Theta.\ \Theta \in \overline{\Theta} \implies$
    $\exists P', R'.\ (\phi_{\mathtt{sel}_{x_i}} :: \Theta)(\{x_1 : X_1, \dots, x_m : X_m, \dots, x_n : X_n\}) = \texttt{inl}\ P' \wedge$
    $(\phi_{\mathtt{sel}_{x_i}} :: \Theta)(\{x_1 : Y_1, \dots, x_n : Y_n\} = \texttt{inl}\ R'$
    for any type X_j and Y_j where $1 \leq j \leq m$ and $j \neq i$.
    In particular for $X_1 = P_1, \dots, X_m = P_m, \dots, X_n = P_n$ and $Y_1 = R_1, \dots, Y_m = R_m$ and by definition of *TypeFocus* composition,
    where $\overline{\Theta}^i = \left\{ \phi_{\mathtt{sel}_{x_i}} :: \overline{\Theta} \right\}$,
    $\vdash^{sub} S \not<: T \to \overline{\Theta}^i \implies$
    $(\forall \Theta.\ \Theta \in \overline{\Theta}^i \implies \exists P', R'.\ \Theta(S) = \texttt{inl}\ P' \wedge \Theta(T) = \texttt{inl}\ R' \wedge$
    $(P' \not<: R' \vee R' \not<: P')$
    By the definition of `head`, $\overline{\Theta}^1, \dots, \overline{\Theta}^m$ extract distinct type elements of the record types, and their composition makes the result immediate.

□

Appendix F

Proofs on mapping between the low-level instrumentation data and its high-level representation

A translation between a low-level instrumentation data and its high-level representation involves designing a class hierarchy that correctly reflects the individual dependencies of the typing decisions as well as different type checker executions. The ability to support the mapping in a one-to-many fashion allows to reduce the boilerplate of both of the representations but at the same time introduces the possibility of ambiguous mappings. Lemma 5.1 formally states the guarantee of inferring a unique mapping from a number of possible definitions, under certain circumstances.

Before we provide the main proof, we first state a few technical definitions.

The 20 condition ensures that the correct, order-preserving matching assigns a high-level instance to the member if and only if the latter is not preceded by some other non-*optional* member.

Definition 20 *Finding a match between an instance of a high-level representation and some high-level specification that respects the order of the specification*
Let T_1 represent an instance of a high-level class hierarchy, such that $\texttt{sub}(\texttt{runtimeTpe}(T_1), Goal)$, and let $\overline{M}_C$ represent a fragment of the specification for some high-level representation of type C, in which a mapping for T_1 is to be found.

Any inferred mapping, $\sigma_A^{\overline{T}}$, that respects the order of the specification and assigns T_1, where $\overline{T} = \{ T_1, \ldots, T_n \}$ for some n, to one of the members of the provided specification, say $M_x = \langle x, S_x \rangle$, such that $M_x \in \overline{M}_C$ and $T_1 \in \sigma_A^{\overline{T}}(x)$, then
$\texttt{sub}(\texttt{runtimeTpe}(T_1), \texttt{underlying}(S_x)) \wedge S_x \in \overline{S}_C \wedge (\forall S_y .\ S_y \in \overline{S}_C \wedge \texttt{idx}(S_y, \overline{S}_C) < \texttt{idx}(S_x, \overline{S}_C) \implies \texttt{opt}(S_y))$.

Appendix F. Proofs on mapping between the low-level instrumentation data and its high-level representation

Lemma F.1 *Uniqueness of matching against a single high-level representation*
Let $\overline{T}'$ represent a sequence of high-level instances, which have been already mapped from low-level instrumentation data, E be the low-level instrumentation event representing an instrumentation block they are part of, and Z a possible, *safe*, high-level type to which E can map to, *i.e.,* $Z = \left\langle E, Z_{super}, \overline{M}_Z \right\rangle$.
If $\sigma_Z^{\overline{T}'}$ represents some inferred mapping, then
$\forall \sigma'^{\overline{T}'}_Z . \sigma'^{\overline{T}'}_Z$ **is defined** $\wedge\ dom(\sigma'^{\overline{T}'}_Z) = dom(\sigma^{\overline{T}'}_Z) \implies \forall x. \sigma^{\overline{T}'}_Z(x) = \sigma'^{\overline{T}'}_Z(x)$

Proof.
Sketch.
Proof by contradiction. We assume that it is possible to infer two different, but still correct, mappings that satisfy the same context, consisting of the same sequence of high-level instances, for the same high-level type.

By the order-preservation property of the inferred mapping and Definition 20, we notice that the different mappings can only occur in the presence of members, either of which is *optional*, which share the same type and are only separated by a zero or more *optional* members. By the *uniqueness* property, the latter are prohibited, hence contradiction. □

The single inheritance nature of the high-level class hierarchy allows to state a subtyping relation between the two, different, types that are super types of the same, third type, as formally stated in Lemma F.2. We notice that our approach of defining the high-level representation implicitly limits Scala's capabilities - the language allows for the multiple inheritance of traits.

Lemma F.2 *A relation between the two high-level types that are subtypes of the same type.*
Let T_i represent a type of an instance of a high-level class, in some high-level representation hierarchy, and T_x and T_y stand for two, potentially distinct, types of two high-level classes in the same high-level representation hierarchy.

If $\texttt{sub}(T_i, T_x) \wedge \texttt{sub}(T_i, T_y) \implies \texttt{sub}(T_x, T_y) \vee \texttt{sub}(T_y, T_x)$

Proof.
Sketch

From the definition of the *linearization* and the definition of subtyping for types of the high-level class hierarchy, we have that:

- $\texttt{linearization}(T_i, Goal) = \{\, Goal, \,...,\, T_x, \,...,\, T_i \,\}$
- $\texttt{linearization}(T_i, Goal) = \left\{\, Goal, \,...,\, T_y, \,...,\, T_i \,\right\}$

In order for both of the subtyping relations to be satisfied at the same time, we have to have that $\texttt{linearization}(T_i, Goal)$ is either $\left\{\, Goal, \,...,\, T_y, \,...,\, T_x, \,...,\, T_i \,\right\}$ or $\left\{\, Goal, \,...,\, T_x, \,...,\, T_y, \,...,\, T_i \,\right\}$, which is sufficient to satisfy the right hand side of the stated implication. □

Proof of Lemma 5.1 on the correctness of the unique one-to-many mapping.

Proof.
Proof by contradiction.

We assume the existence of a pair of high-level *safe* types, A and B, the low-level event E can map to, for a context sequence $\overline{T}$, and the corresponding, correct, inferred mappings $\sigma_A^{\overline{T}}$ and $\sigma_B^{\overline{T}}$, respectively, that completely matched the provided context.

Case $\texttt{sub}(A, B)$:

By Lemma F.1 we notice that the shared members of A and B, *i.e.,* in the case of $\texttt{sub}(A, B)$ the members of B, are matched with some sequence of members $\overline{T}'$, such that $\overline{T}'' = \overline{T} \setminus \overline{T}'$ and $\overline{T}' \subseteq \overline{T}$.

By the *non-ambiguity* property, there exists a non-*optional* member that is declared in A.
If $\overline{T}'' = \epsilon$, then $\sigma_A^{\overline{T}}$ does not represent a complete matching, since the declared, non-*optional* member of A can be satisfied, and if $\overline{T}'' \neq \epsilon$ then $\sigma_B^{\overline{T}}$ does not represent a complete matching since $\overline{T}''$ instances are not represented in the mapping. A contradiction.

The case of $\texttt{sub}$(B, A) is analogous to the above argument.

Case $\neg\texttt{sub}(A, B)$:

We notice that the least upper bound of the two possible high-level types, *i.e.,* $\texttt{lub}(A, B) = C$, always exists. Moreover, by Lemma F.1 the shared members of C are matched with the same sequence of high-level instances of the context, denoted as $\overline{T}_{A,B}$. Hence, the difference in the inferred mappings $\sigma_A^{\overline{T}}$ and $\sigma_B^{\overline{T}}$ can only occur in the matching of the remaining sequence $\overline{T}'$, such that $\overline{T}' = \overline{T} \setminus \overline{T}_{A,B}$, against the specifications of $\texttt{spec}(A, C)$ and $\texttt{spec}(B, C)$.

By the (double) induction on the possible types of members of the remaining specifications, $\texttt{spec}(A, C)$ and $\texttt{spec}(B, C)$, with the same remaining $\overline{T}'$ sequence, we show that mappings from high-level types A and B cannot be both satisfied at the same time.

For $\overline{T}' = \epsilon$:

Since the inferred mapping preserves the order of the matching with respect to the $\overline{T}'$ sequence, it is sufficient to show that none of the possible combinations of *heads* of the remaining specifications, *i.e.,* $\texttt{spec}(A, C)$ and $\texttt{spec}(B, C)$, can be satisfied with it at the same time (thus violating the completeness property):

- $\texttt{spec}(A, C) = \langle x, X \rangle$ and $\texttt{spec}(B, C) = \langle y, Y \rangle$ for some x, X, y, and Y:
  Both of the remaining specifications have at least one, non-*optional* member, and both cannot satisfy the completeness property of the mapping with the remaining empty sequence $\overline{T}'$. A contradiction.

- $\texttt{spec}(A,C) = \langle x, X\rangle$ and $\texttt{spec}(B,C) = \langle y, \textsc{List}[Y]\rangle$ for some x, X, y, and Y:
  $\texttt{spec}(A,C)$ has a non-*optional* member. The mapping for A cannot be defined, because it will not be possible satisfy the required completeness property with the remaining sequence $\overline{T}'$. A contradiction.
- $\texttt{spec}(A,C) = \langle x, \textsc{List}[X]\rangle$ and $\texttt{spec}(B,C) = \langle y, \textsc{List}[Y]\rangle$ for some x, X, y, and Y:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A,C) = \langle x, \textsc{List}[X]\rangle$ and $\texttt{spec}(B,C) = \langle y, \textsc{List}[Y]\rangle$ for some x, X, y, and Y:
  The matching for members x and y is satisfied in both cases through an empty sequence. By induction, on the *tails* of the specifications, a contradiction.
- $\texttt{spec}(A,C) = \langle x, X\rangle$ and $\texttt{spec}(B,C) = \epsilon$ for some x, X:
  $\texttt{spec}(A,C)$ has a non-*optional* member. The mapping for A cannot be defined, because it will not satisfy the required completeness property with the remaining empty sequence $\overline{T}'$. A contradiction.
- $\texttt{spec}(A,C) = \epsilon$ and $\texttt{spec}(B,C) = \langle y, Y\rangle$ for some y, Y:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A,C) = \langle x, \textsc{List}[X]\rangle$ and $\texttt{spec}(B,C) = \epsilon$ for some x, and X:
  By the non-ambiguity property, $\texttt{spec}(A,C)$ has to have at least a single non-*optional* member. The latter can never be satisfied with an empty sequence $\overline{T}'$, therefore at least one of the inferred mappings will never satisfy the completeness property. A contradiction.
- $\texttt{spec}(A,C) = \epsilon$ and $\texttt{spec}(B,C) = \langle x, \textsc{List}[X]\rangle$ for some y, and Y:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A,C) = \epsilon$ and $\texttt{spec}(B,C) = \epsilon$:
  A case is not possible by the non-ambiguity property and $\neg\texttt{sub}(A,B)$.

For $\overline{T}' = \{\, T'_1, \ldots, T'_n \,\}$*:*

Since the inferred mapping preserves the order of the matching for sequence $\overline{T}'$, it is sufficient to show that none of the possible combinations of *heads* of the remaining specifications, *i.e.,* $\texttt{spec}(A,C)$ and $\texttt{spec}(B,C)$, can be matched against it at the same time:

- $\texttt{spec}(A,C) = \langle x, X\rangle$ and $\texttt{spec}(B,C) = \epsilon$ for some x, X:
  Since $\texttt{spec}(B,C)$ has no further required, or *optional* members, the mapping for B will never represent the required complete matching. A contradiction.
- $\texttt{spec}(A,C) = \epsilon$ and $\texttt{spec}(B,C) = \langle y, Y\rangle$ for some y, Y:
  By the analogous argument as in the previous case.

- $\texttt{spec}(A, C) = \langle x, \textsc{List}[X] \rangle$ and $\texttt{spec}(B, C) = \epsilon$ for some x, X:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A, C) = \epsilon$ and $\texttt{spec}(B, C) = \left\langle y, \textsc{List}[Y] \right\rangle$ for some y, Y:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A, C) = \epsilon$ and $\texttt{spec}(B, C) = \epsilon$:
  By the analogous argument as in the previous case.
- $\texttt{spec}(A, C) = \langle x, X \rangle$ and $\texttt{spec}(B, C) = \left\langle y, Y \right\rangle$ for some x,y, X, and Y:
  We now consider four possible subcases, based on the type of T_1:
  - $\texttt{sub}(T_1', X)$ and $\texttt{sub}(T_1', Y)$:
    By Lemma F.2, either $\texttt{sub}(X, Y)$ or $\texttt{sub}(Y, X)$. By the non-ambiguity property, the remaining specifications cannot coincide on their prefixes, so the class hierarchy is not possible. A contradiction.
  - $\texttt{sub}(T_1', X)$ and $\neg\texttt{sub}(T_1', Y)$:
    Since both required members are non-*optional*, and the mapping has to represent a complete, order-preserving matching, a contradiction.
  - $\neg\texttt{sub}(T_1', X)$ and $\texttt{sub}(T_1', Y)$:
    By the analogous argument as in the previous case.
  - $\neg\texttt{sub}(T_1', X)$ and $\neg\texttt{sub}(T_1', Y)$:
    By the analogous argument as in the previous case.
- $\texttt{spec}(A, C) = \langle x, X \rangle$ and $\texttt{spec}(B, C) = \left\langle y, \textsc{List}[Y] \right\rangle$ for some x,y, X, and Y:
  We now consider four possible subcases, based on the type of T_1:
  - $\texttt{sub}(T_1', X)$ and $\texttt{sub}(T_1', Y)$:
    By Lemma F.2, either $\texttt{sub}(X, Y)$ or $\texttt{sub}(Y, X)$. This in turn violates the non-ambiguity property, since $\exists y.\ y \in \texttt{prefix}(\texttt{spec}(B, C)) \ \wedge \texttt{sub}_2(x, y)$. A contradiction.
  - $\texttt{sub}(T_1', X)$ and $\neg\texttt{sub}(T_1', Y)$:
    By Definition 20, in order for the matching to be complete, there exists at least one member $\left\langle y', S_y' \right\rangle$ such that $\texttt{underlying}(S_y') = T_y'$, $\texttt{sub}(T_1', T_y')$. This in turn means that y' belongs to the prefix of $\texttt{spec}(B, C)$ and that violates the non-ambiguity because either $\texttt{sub}(X, Y')$ or $\texttt{sub}(Y', X)$ (by Lemma F.2). A contradiction.
  - $\neg\texttt{sub}(T_1', X)$ and $\texttt{sub}(T_1', Y)$:
    Since $\neg\texttt{sub}(T_1', X)$, the completeness property of the inferred mapping $\sigma_A^{\overline{T}}$ can never be satisfied, given that the inferred mappings always represent a complete matching.
  - $\neg\texttt{sub}(T_1', X)$ and $\neg\texttt{sub}(T_1', Y)$:
    Since neither $\texttt{sub}(T_1', X)$ nor $\texttt{sub}(T_1', Y)$, the case can be simply ignored, as no matching will take place for both types, and by induction on their *tails* both remaining specifications cannot be satisfied at the same time.

- `spec`$(A,C) = \langle x, \text{LIST}[X]\rangle$ and `spec`$(B,C) = \langle y, Y\rangle$ for some x,y, X, and Y:
  By the analogous argument as in the previous case.
- `spec`$(A,C) = \langle x, \text{LIST}[X]\rangle$ and `spec`$(B,C) = \langle y, \text{LIST}[Y]\rangle$ for some x,y, X, and Y:
  We now consider four possible subcases, based on the type of T_1:
  - `sub`(T_1', X) and `sub`(T_1', Y):
    Since `sub`(T_1', X) and `sub`(T_1', Y), the case can be simply ignored, as matching will take place in both cases, and by induction on the same specifications, but the tail of $\overline{T}'$, both specifications cannot be satisfied at the same time.
  - $\neg$`sub`(T_1', X) and `sub`(T_1', Y):
    By Definition 20, in order for the matching to be complete, there exists at least one member $\langle x', S_x'\rangle$ such that `underlying`$(S_x') = T_x'$, `sub`(T_1', T_x'). If $S_x' = \text{LIST}[T_x']$, then by the analagous argument as in the previous case, a contradiction. If $S_x' = T_x'$, then by Lemma F.2, either `sub`(T_x', Y) or `sub`(Y, T_x'). The latter, in turn, violates the non-ambiguity property since Y appears in the *prefix* of `spec`(B,C). A contradiction.
  - `sub`(T_1', X) and $\neg$`sub`(T_1', Y):
    By the analogous argument as in the previous case.
  - $\neg$`sub`(T_1', X) and $\neg$`sub`(T_1', Y):
    Since neither `sub`(T_1', X) nor `sub`(T_1', Y), the case can be simply ignored, as no matching will take place for both types, and by induction on their *tails* both remaining specifications cannot be satisfied at the same time.

The case of $\neg$`sub`(B,A) is analogous to the above argument.

$\square$

Bibliography

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8.

Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013a. ACM. ISBN 978-1-4503-2064-1.

Eugene Burmako. Applied materialization, 06 2013b. URL http://scalamacros.org/paperstalks/2013-06-13-AppliedMaterialization.pdf.

Sheng Chen and Martin Erwig. Guided type debugging. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 35–51. Springer International Publishing, 2014a.

Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014b. ACM.

Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 193–204, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.

Maurizio Cimadamore. Jep 101: Generalized target-type inference, 02 2015. URL http://openjdk.java.net/jeps/101.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6.

Mathias Doenitz. Magnet pattern, 12 2012. URL http://spray.io/blog/2012-12-13-the-magnet-pattern/.

Iulian Dragos. Optimizing Higher-Order Functions in Scala. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008.

Gilles Dubochet and Donna Malayeri. Improving api documentation for java-like languages. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 3:1–3:1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1.

Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, July 1996.

Nabil El Boustani and Jurriaan Hage. Corrective hints for type incorrect generic java programs. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 5–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1.

Nabil el Boustani and Jurriaan Hage. Improving type error messages for generic java. *Higher-Order and Symbolic Computation*, 24(1-2):3–39, 2011. ISSN 1388-3690.

Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In Erik Ernst, editor, *ECOOP 2007 Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 273–298. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73588-5.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 383–396, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

Tihomir Gvero and Viktor Kuncak. Interactive synthesis using free-form queries (tool demonstration). In *International Conference on Software Engineering (ICSE)*, 2015.

Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, March 2004. ISSN 0167-6423.

Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages*, IFL'06, pages 199–216, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-74129-9.

Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*, ESOP '94, pages 241–256, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

Thomas Hallgren. Fun with functional dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, 2000.

Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Constraint based type inferencing in helium. *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59–80, 2003a.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 3–13, New York, NY, USA, 2003b. ACM. ISBN 1-58113-756-7.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003c. ACM. ISBN 1-58113-758-3.

Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 09 2005. URL http://www.cs.uu.nl/people/bastiaan/phdthesis.

Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical report, University of Pennsylvania, 1999.

Alan Jeffrey. Generic java type inference is unsound. note sent to the types mailing list., 2001. URL http://www.cis.upenn.edu/~bcpierce/types/archives/current/msg00849.html.

Yang Jun, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45:2002, 2002.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and WilliamG. Griswold. An overview of aspectj. In JørgenLindskov Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42206-8.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 301–331. Springer London, 2010. ISBN 978-1-84882-911-4.

Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Phantm: Php analyzer for type mismatch. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 373–374, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2.

Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.React. Technical report, EPFL, 2012. URL http://infoscience.epfl.ch/record/176887/.

Bruce McAdam. Trends in functional programming. In Kevin Hammond and Sharon Curtis, editors, *TFP*, chapter How to Repair Type Errors Automatically, pages 87–98. Intellect Books, Exeter, UK, UK, 2002. ISBN 1-84150-070-4.

Bruce J. McAdam. Generalising techniques for type debugging. In *Selected Papers from the 1st Scottish Functional Programming Workshop (SFP99)*, SFP '99, pages 50–58, Exeter, UK, UK, 2000. Intellect Books. ISBN 1-84150-024-0.

Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1.

Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 423–438, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3.

Martin Odersky. Inferred type instantiation for GJ, 2002. URL http://lampwww.epfl.ch/$\sim$odersky/papers/localti02.html.

Martin Odersky. The Scala language specification, 2015. URL http://www.scala-lang.org/docu/files/ScalaReference.pdf.

Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS*, 2009.

Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, January 1999.

Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA, 2001. ACM. ISBN 1-58113-336-7.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP 2003 Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40531-3.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, EPFL, 2006.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: A new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. ISSN 0164-0925.

Hubert Plociniczak. Scalad: An interactive type-level debugger. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 8:1–8:4, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1.

Hubert Plociniczak and Martin Odersky. Implementing a type debugger for Scala. In *APPLC*, 2012.

Hubert Plociniczak, Heather Miller, and Martin Odersky. Improving Human-Compiler Interaction Through Customizable Type Feedback. Technical report, EPFL, 2014.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012. ISSN 1388-3690.

Claudio V. Russo and Dimitrios Vytiniotis. Qml: Explicit first-class polymorphism for ml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3.

Matthew Sackman and Susan Eisenbach. Errors for the Common Man: Hiding the unintelligable in Haskell. Technical report, Imperial College London, September 2008. URL http://pubs.doc.ic.ac.uk/error-handling-for-Haskell/.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596599. URL http://doi.acm.org/10.1145/1596550.1596599.

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, November 2005. ISSN 0164-0925.

Martin Sulzmann. An overview of the Chameleon System. In *APLAS*, 2002.

Kanae Tsushima and Kenichi Asai. An embedded type debugger. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 190–206. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4.

Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, September 2011. ISSN 0956-7968.

Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. Security type error diagnosis for higher-order, polymorphic languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 3–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1842-6.

J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1-3):111–156, 1999.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 12–21, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6.

Hubert Plociniczak - Résumé

Current address	Lausanne, Switzerland	**Phone**	+41 792562005
Date of Birth	6th November 1986	**Email**	hplociniczak@gmail.com
Nationality	Polish		

Education

2009-2015 PhD Student - EPFL, Switzerland

Member of the Scala Lab at EPFL
Thesis subject: *Debugging Local Type Inference*
Research interests: *Programming languages, type systems, type inference, debugging type systems*
Supervised by Prof. Martin Odersky.

2005-2009 MEng Hons Computing (Software Engineering) - Imperial College London, UK

First class degree, finished in the top of the class.
Specialization: *Software Engineering*
Four year course leading directly to MEng.
Distinguished final individual project: *JErlang: Erlang with Joins*

Employment History

Sep 2009 - Present Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
Research Assistant/Teaching Assistant/Compiler hacker

- Member of the Scala Core Team. Scala Compiler hacking/bug-fixing.
- Worked on techniques for debugging decisions of the existing Scala type checker, with a particular interest in providing a more informative error feedback and developing an interactive type debugger.
- Co-author of Scala Records. Used by Databricks.
- Google Summer of Code mentor and administrator (3 consecutive years) for the Scala Team. Supervised student projects and coordinated mentors' work for all Scala projects.
- Head Teaching Assistant for Foundations for Software (for 4 years), TA for Functional Programming and Object-oriented Programming courses.
- Presenter at industrial (e.g., Scala Days) and academic conferences.

Dec 2010 - Dec 2011 Typesafe Inc., Lausanne, Switzerland
Software developer

Software company providing commercial support for Scala, Akka and Play!

- Worked mainly on the ScalaIDE open-source project - Eclipse plugin for Scala. Stability improvements to the Scala Presentation Compiler and general bug-fixing.
- Worked on the improvements to the build mechanism, including integration with sbt build tool.
- Main technologies used: Scala, sbt, Eclipse, Java.

Part-time work

Jan 2007 - Jan 2009 LShift Ltd, London, UK
Software Developer

Hi-tech software development company

- Worked on further development of distributed subscription licensing system used by over 300 000 customers. Improved innovative licensing model, a messaging system and a profile management system. Additionally responsible for re-writing the security model and reporting section.
- Improved internal open-source projects Timetracker and Timezilla that were used by the developers for project management/resourcing.
- Worked on the Java and C♯ clients as well as the server for the RabbitMQ product (open-source implementation of Advanced Message Queuing Protocol). Involved in integrating the packing system with Debian/Ubuntu and Fedora standards.
- Main technologies used: Java, J2EE, Tomcat, Apache, Python, Jython, Perl, XML, XSLT, Jini, Hibernate, Struts, .Net/C♯, Erlang, PostgreSQL.

Worked part-time during term-time, full-time in summer and 6-month placement.

06/2006 - 01/2007 Applied Modelling and Computation Group (AMCG), Imperial College London, UK
Software Developer

- Created complex application (including logic and GUI) used in pre-processing stage for simulation modelling of Computational Fluid Dynamics (CFD) data and developed system for storing and managing over 500 simulation options with multiple dependencies and meta-data.
- Successfully implemented CFD General Notation System (CGNS) library in open source project in the given time.
- Technologies used: Python, C/C++, PyGTK, XML.

Worked part-time during term-time, full-time for 4 months in summer.

Skills

- **Programming Languages**
  Scala, Java technologies (including Java SE, J2EE, Struts, Jini), Python, Jython, PHP, C/C++, Prolog, Haskell, Erlang
- **Web technologies**
  HTML, XHTML, XML, XSLT, CSS, Javascript, AJAX, Flash
- **Miscellaneous**
  PostgreSQL, MySQL, Berkeley DB, Hibernate, Tomcat, Apache, RPM and Debian Packages, Jenkins, sbt, Akka, networking, Git, SVN, Mercurial, JIRA, Bugzilla, Trac, JVM, ScalaTest, ScalaCheck
  Linux: Ubuntu, Debian, openSuse, Fedora
  Windows: XP/Vista/7
- **Languages**
  Proficient English, Native Polish, Intermediate German, Basic French

Awards, Certificates

- June 2009, The Centenary Prize
  Awarded for the final individual project at the Department of Computing

- June 2008, Dynamics System Development Method (DSDM) Certified Practitioner
  Successfully passed agile software development course
- October 2007, Gloucester Research Prize
  Awarded to top ten students at the Department of Computing for academic excellence
- May 2006, MISYS Charitable Foundation Scholarship
  Awarded for excellent results in subjects at the Department of Computing

Interests

- **Road Cycling, Climbing, Blues Dancing**
- **Blues/Jazz Music**
- **Scientific literature, Politics**

References available upon request