AN ABSTRACT OF THE DISSERTATION OF

Sheng Chen for the degree of Doctor of Philosophy in Computer Science presented on July 1, 2015.

Title: Variational Typing and Its Applications

Abstract approved: _____

Martin Erwig

The study of variational typing originated from the problem of type inference for variational programs, which encode numerous different but related plain programs. In this dissertation, I present a sound and complete type inference algorithm for inferring types of all plain programs encoded in variational programs. The proposed algorithm runs exponentially faster than the strategy of generating all plain programs and applying type inference to them separately. I also present an error-tolerant version of variational type inference to deliver better feedback in the presence of ill-typed plain programs. All presented algorithms require various kinds of variational unification. I prove that all these problems are decidable and unitary, and I develop sound and complete unification algorithms. The idea of variational typing has many applications. As one example, I present how variational typing can be employed to improve the diagnosis of type errors in functional programs, a problem that has been extensively studied.

Variational Typing and Its Applications

by

Sheng Chen

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented July 1, 2015
Commencement June 2016

Doctor of Philosophy dissertation of Sheng Chen presented on July 1, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Sheng Chen, Author

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Martin Erwig, who nurtured me as a researcher and thinker better than I ever expected. Over the past six years, his guidance and support improved my presentation and technical writing skills, deepened my understanding of programming languages and type systems, and positively shaped my way of viewing and solving problems. The discussions with him were enlightening, inspiring, and a lot of joy. I also want to thank him for providing a firm financial support for me through the graduate research assistant position so that I could focus on doing research and had many chances of attending conferences. His help of organizing a dry run of a job interview was invaluable to me.

I want to express a special thank you to my wife Keying Xu. Her true love, support, encouragement, and confidence in me helped me to be more focused and productive. Her love and devotion contributed to a smooth transition from being blank and not knowing what to do when I first moved to America to being more relaxed and confident now. I appreciate her support in many roles, as my wife, my children's Mom, and my coworker. I feel very lucky that she shares her life with me.

During my study at Oregon State University, I worked closely with Eric Walkingshaw and had a lot of interesting and rewarding discussions with him. I want to thank him for his support in many aspects. Thank you to him for driving three hours every day for two weeks to attend Oregon Programming Language Sum-

mer School in Summer 2012. Thank you to him for solving many puzzling LaTeX problems and sharing LaTeX sources for various documents with me.

I want to thank many faculty members here at Oregon State University. The members of my program committee, Glencora Borradaile, Alex Groce, Prasad Tadepalli, and Eric Walkingshaw, offered a lot of support during my study here. Thank you to the faculty members who did mock job interviews with me.

I want to thank my friends Duc Le and Chris Chambers, the conversations, debates, arguments, and jokes with them turned many boring moments into fun. Duc offered a lot of help when I was really in need. Thank you to him for all his rides and delicious noodle soups.

Thank you to my family for their unconditional love throughout my life. Without knowing what I was up to, they supported my decisions and interests completely. Thank you to my in-laws for their kind support. They, together with my Mom, went to the US several times to take care of my children. Without their help, the completion of this dissertation wouldn't have been possible.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF APPENDIX FIGURES

Chapter 1: Introduction

In this dissertation, I present my research on variational typing, which is a theoretically sound and principled solution to the practical problem of type checking and type inference of software product lines. Variational typing offers extra insights beyond its original problem territory, and it has been used to solve many long-standing problems. In the later part of this dissertation, I present one such application.

This chapter motivates the needs of variational typing by investigating the challenges of typing software product lines. It also outlines the structure of this dissertation and presents the contributions of this work.

1.1 Variational Programming and Challenges

The idea of writing a set of related programs, rather than a single program, at a time is widely adopted in implementing software systems. There are several reasons for this. First, it helps to maximize the reuse of data structures and algorithms. Second, it saves costs to develop and test programs. Finally, it significantly shortens time to market: picking a program from existing ones is much faster than developing a new one.

This idea can be implemented through the use of software product lines [Pohl

```
int y;
#ifdef A
int x = 3;
#else
char x = '3';
#endif

y = 7 + 5 + x;
```

(a) A simple VP.

```
int y;
int x = 3;
y = 7 + 5 + x;
```

(b) Variant when A defined.

```
int y;
char x = '3';
y = 7 + 5 + x;
```

(c) Variant when A undefined.

Figure 1.1: A variational program with its variants.

et al., 2005], generative programming [Czarnecki and Eisenecker, 2000], feature-oriented software development [Apel and Kästner, 2009], and variational programming [Erwig and Walkingshaw, 2013]. For example, Figure 1.1a presents a variational program (VP) implemented using C Preprocessor (CPP) [GNU Project, 2009]. This VP implements a set of two programs, which are called variants. Variants can be generated from VPs by making configurations. For example, when the macro `A` is defined, the variant in Figure 1.1b is generated. Otherwise, if `A` is undefined, the variant in Figure 1.1c is generated.

We can observe that with the single macro A the example VP can generate two variants. In general, with n independent macros, a VP can generate 2^n variants. While this means that with one VP we can potentially generate a lot of variants satisfying different requirements, it's also likely that the VP introduces unintended interactions [Abal et al., 2014; Nadi et al., 2013]. For example, consider the following code, which is a snippet from Busybox [Kästner et al., 2012b]. The problem here is that when the macro `ENABLE_FEATURE_PS_LONG` is undefined, the variables `now` and `uptime` will not be declared. However, the code later always

refers to them, whether ENABLE_FEATURE_PS_LONG is defined or not. As a result, when ENABLE_FEATURE_PS_LONG is undefined, the code will cause a type error since it refers to undeclared variables.

```
int ps_main(int argc, char **argv){
...
#if ENABLE_FEATURE_PS_LONG
  time_t now = now;
  long uptime;
#endif
...
  puts("S UID PID PPID VSZ RSS TTY STIME TIME CMD");

  now = time(NULL);
  uptime = get_uptime();

...
}
```

The question is, how can we detect this kind of problems in VPs, or better guarantee the correctness of all generated variants? A potential solution is to leverage the existing static analyses. While typically not equipped to deal with macros, any static analysis defined for single programs can be conceptually extended to variational programs by simply generating all program variants and analyzing each one individually. In practice, this is usually impossible due to the sheer number of variants that can be generated. As mentioned earlier, a variational program with n macros is likely to generate 2^n variants. For this reason, this brute-force approach already fails when the variational program contains about 30 macros. Moreover, it is common for variational program to have hundreds of macros, for

example, SQLite[1] contains about 300 macros, MySQL[2] contains some 900 macros, and Linux contains about 10,000 macros. Thus, this strategy works only for the most simple variational programs. Sampling techniques can be used to improve the situation, but do not solve the problem in general [Devroey et al., 2014].

Efficiency is not the only problem of the brute-force strategy; there is also the issue of how to *represent* the results of analyses on variational software. Using the brute-force approach, one type error in the variational program can cause type errors in a huge number of program variants. These errors will be reported in terms of particular variants rather than the actual variational source code. Similarly, the types inferred for a program variant will not correspond to the variational source code, limiting their use as a way to describe and understand the original variational program.

1.2 Solution with Variational Typing

This dissertation presents my solution to the aforementioned challenges through the idea of *variational typing*, which consists of both variational type checking and variational type inference. Many efforts have been made to address the efficiency problem by developing strategies that type check variational software directly, rather than individual program variants [Thaker et al., 2007; Kästner et al., 2012a; Kenner et al., 2010]. My work focuses on the more difficult problem of *type inference*, which differs from type checking in the following aspects.

[1] https://www.sqlite.org/
[2] https://www.mysql.com/

1. Type representations. While the purpose of type checking is to determine if variational programs are well typed, that of type inference is to additionally infer the result types of variational programs. Thus, we need a representation of typing result.

2. Underlying computations. While type checking requires only to check the equality of types, type inference involves the solving of type unification problems.

3. Error vulnerabilities. It is easier to recover from type errors in type checking than in type inference because there is more type information available in type checking.

An important result of variational type inference is that it assigns a type to a variational program if and only if it contains only well-typed variants. This has two important implications. First, in order to decide whether all variants of a VP are well typed, there is no need to generate all variants and type each individually. Instead, we can directly apply variational type inference to the VP. If the VP is well typed, then all the variants are well typed. Second, an unfortunate consequence of this is that variational type inference fails to reveal useful type information for VPs that may contain errors in only few variants. This implies that the inference algorithm can be employed only when the development of the whole VP is completed. However, a more natural way of developing a VP is first constructing the more important variants and gradually expanding the core parts to build the whole VP [Apel and Kästner, 2009].

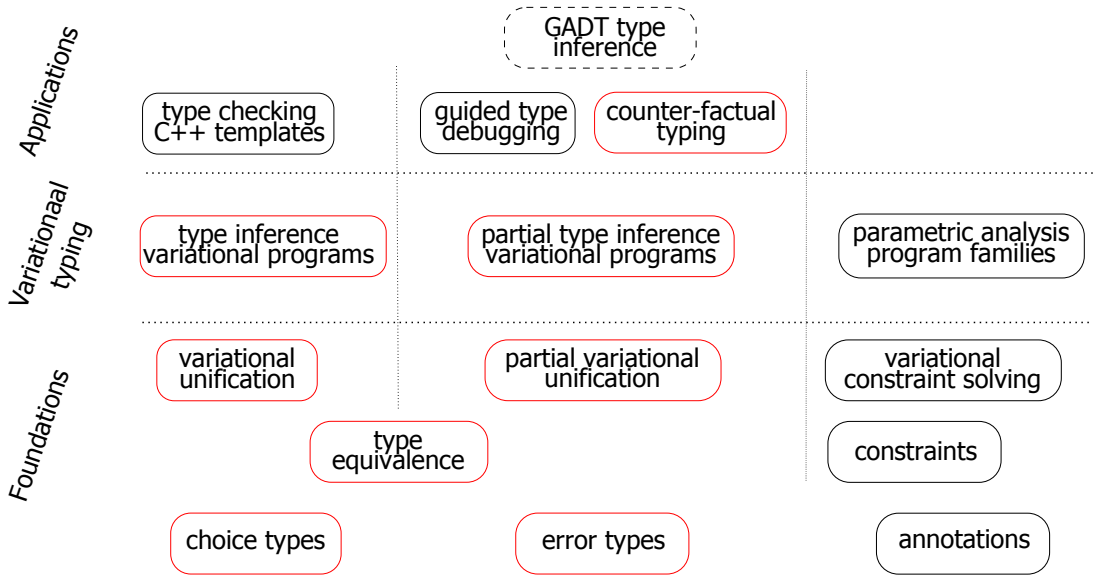


Figure 1.2: Foundations and applications of variational typing. This dissertation will focus on boxes with red lines.
Legend ———— : completed – – – – : in progress

To address this issue, we develop an error-tolerant type system by introducing error types to quarantine type errors such that the typing process is not terminated until the whole process is completed [Chen et al., 2012]. The inference result contains error types for variants that have type errors and plain types for other variants. The extended inference algorithm thus works for all VPs.

Both the variational type inference and partial type inference algorithms are based on some foundational work. In Figure 1.2, we observe that the variational type inference algorithm relies on a variational unification algorithm, a notion of choice types, and a type equivalence relation. The partial type inference algorithm relies on a partial unification algorithm, a notion of error types, and an extended type equivalence relation.

As Figure 1.2 shows, variational typing includes another component: type-based parametric analysis of VPs [Chen and Erwig, 2014a], which generalizes choice types and the variational unification algorithm to build a generalized type inference algorithm. More specifically, by annotating types with various kinds of information, we have designed a type-based static analysis lifting framework, which lifts static analyses for plain program to those for variational program. Designing specific static analyses for VPs has been an active research area. Many variational analysis algorithms have been proposed [Thüm et al., 2014], and each of them involves a considerable amount of work. Our lifting framework automates this lifting process for many static analyses.

Note that in Figure 1.2, the x-axis doesn't have a special meaning except for the "Foundations" category, where notions presented to the right are more powerful than those to the left. For example, both error types and annotated types subsume choice types.

1.3 Variational Typing Applications

As a relatively new research area, much of the work in variational analysis is driven by approaches and results from traditional program analysis. This can be seen, for example, in variational parsing [Kästner et al., 2011; Gazzillo and Grimm, 2012], type checking [Kästner et al., 2012b,a; Apel et al., 2010; Delaware et al., 2009b], dataflow analysis [Brabrand et al., 2012; Liebig et al., 2012], model checking [Classen et al., 2010, 2011; Cordy et al., 2012; Apel et al., 2013] and theo-

rem proving [Delaware et al., 2011; Thüm et al., 2012]. Similarly, variational type inference is obtained by extending the machineries for traditional type inference to deal with variations.

Interestingly, it turns out that the representations and methods developed for variational analysis can in turn be applied to improve traditional program analysis. Examples are shown in Figure 1.2 in the "Applications" category. In the following, I will briefly describe some of the applications that are completed.

Type checking C++ Templates [Chen and Erwig, 2014d]. Despite the numerous efforts in improving type checking of C++ Templates [Miao and Siek, 2010; Gregor et al., 2006; Järvi et al., 2006; Siek and Taha, 2006a; Reis and Stroustrup, 2005; Garcia and Lumsdaine, 2009; Dos Reis and Stroustrup, 2006], no type safety guarantees are available of C++ Templates until they are instantiated and object programs are generated. We partially addressed this problem by developing a type system for a calculus that captures the essential capabilities of C++ Templates. With choice types, our solution uses a more precise characterization of types and thus a better utilization of type information within template definitions. Our type system guarantees that well-typed templates generate only well-typed object programs.

Counter-factual typing [Chen and Erwig, 2014e]. In terms of error localization, type checking and type inference algorithms suffer from an inherent precision problem. Debugging type errors for functional programs has been an active research area for the past three decades [Wand, 1986; Johnson and Walz, 1986; McAdam, 2002b; Lee and Yi, 1998, 2000; Yang, 2001; Wazny, 2006; Lerner et al.,

2007; Choppella, 2002; McAdam, 2002a; Tip and Dinesh, 2001; Haack and Wells, 2003; Heeren, 2005; Schilling, 2012], yet effectively locating and reporting type errors remains a problem. Based on variational typing, we have developed counter-factual typing that significantly increases error reporting precisions.

Guided type debugging [Chen and Erwig, 2014c]. A distinctive feature of guided type debugging is that it allows users to specify a desired type for an ill-typed expression. It then suggests changes that match the desired type while all the previous approaches propose changes based only on guessing. Compared to counter-factual typing, guided type debugging further increases error debugging efficiency.

This dissertation will present the application of counter-factual typing in more detail.

1.4 Contributions and Outline of This Dissertation

In this section, I present the structure of the remainder of this dissertation and the contributions of this work along the way.

Chapter 2 (*Background*) introduces some basic notions that will be used through out the dissertation, the basis upon which variational typing will be developed, and the constructs for representing variations. This chapter contains material from [Chen et al., 2014b].

Chapter 3 (*Literature Review*) collects research related to variational type inference, partial type inference, and type error debugging. This chapter contains

material from [Chen et al., 2014b] and [Chen and Erwig, 2014e].

Chapter 4 (*Variational Type Checking*) introduces variational types and presents the rules for assigning variational types to variational expressions. This chapter contains material from [Chen et al., 2014b] and makes the following contributions.

1. A notion of variational types, which is an extension of type representations with variational constructs to express the result of variational type inference.

2. A type system that maps variational expressions to variational types and related theorems that show that the type system is sound and complete.

3. A type equivalence relation that underlies the proposed type system and a type rewriting relation that checks the equivalence of types, which is terminating and confluent.

Chapter 5 (*Variational Unification*) investigates the properties of variational unification problems and develops a variational unification algorithm. Variational unification problems are equational modulo the type equivalence relation introduced in Chapter 4. Consequently, they are hard to solve. Variational unification is also one of the most important theoretical contributions of this work as it is used in many other applications. This chapter contains material from [Chen et al., 2014b] and makes the following contributions.

1. A proof that the variational type unification problem is decidable and unitary.

2. A variational unification algorithm and a proof that the algorithm is sound, complete, and most general.

Chapter 6 (*Variational Type Inference*) develops a variational type inference algorithm based on the variational unification algorithm presented in Chapter 5 and studies its performance by comparing it with the brute-force approach. It also presents a qualitative analysis of the performance gain. This chapter contains material from [Chen et al., 2014b] and makes the following contributions.

1. A type inference algorithm and a proof that the type inference algorithm is sound, complete, and principal.

2. A performance evaluation that demonstrates the scalability of the variational type inference algorithm.

Chapter 7 (*Partial Variational Typing*) presents an extension to make variational type inference error tolerant. While variational type inference computes types for variational programs that contain well-typed variants only, partial variational typing assigns a type to any variational program. We realize this extension by explicitly representing, introducing, and propagating type errors. This chapter contains material from [Chen et al., 2012] and makes the following contributions.

1. An introduction of typing patterns that indicate which variants contain type errors. They also serve as a measurement for comparing result types and unifiers.

2. A proof about the existence of principal typing patterns for partial unification problems, which implies that for any unification problem, there is some mapping (partial unifier henceforth) that introduces fewest type errors. Also, it proves that partial unification is unitary. Moreover, these two aspects can be reconciled such that for any partial unification problem, there is a mapping (most general partial unifier henceforth) such that it introduces fewest errors and is most general.

3. A partial variational unification algorithm that computes most general partial unifiers, together with a proof about its soundness, completeness, and principality.

4. An inference algorithm that computes partial types for partial programs such that for type-correct variants, the partial type contains principal types and for ill-typed variants, it contains a special type indicating a type error.

Chapter 8 (*Counter-Factual Typing*) presents a method for improving type error debugging, an intensively studied research problem over the past three decades.

Based on generating and filtering a comprehensive set of type-change suggestions, counter-factual typing generates *all* (program-structure-preserving) type changes that can possibly fix the type error. These suggestions will be ranked and presented to the programmer in an iterative fashion. In some cases it also produces suggestions to change the program. In most situations, this strategy delivers the correct change suggestions quickly, and at the same time never misses

any rare suggestions. The computation of the potentially huge set of type-change suggestions is efficient since it is based on the variational type inference algorithm developed in Chapter 7, which efficiently reuses type information for shared parts. This chapter contains material from [Chen and Erwig, 2014e] and makes the following contributions.

1. A type debugging method that covers all potential error locations and generates a type change or expression change suggestion for each erroneous location.

2. A type system based on choice types plus a proof that the typing result encodes all potential type changes for any ill-typed expression.

3. An implementation of a type inference algorithm that, based on partial variational unification, computes all change suggestions efficiently.

4. A comparison of our approach with three other tools. Based on a large set of examples drawn from the literature, we show that our approach outperforms previous approaches.

Chapter 9 (*Conclusion*) closes this dissertation with a summary of other applications of choice types, the most important contributions of this work, and directions for future research.

Chapter 2: Background

This chapter introduces the foundations for presenting variational typing: the Hindley-Milner type system and the variation representation based on the choice calculus [Erwig and Walkingshaw, 2011].

2.1 Hindley-Milner Type System

Hindley-Milner (HM) is a type system that assigns types to expressions written in lambda calculus with parametric polymorphism [Hindley, 1969; Damas and Milner, 1982]. The syntax for constructing expressions is given below.

e	::=	x	*Variable*
	|	$\lambda x.e$	*Abstraction*
	|	$e\ e$	*Application*
	|	v	*Constant*
	|	**let** $x = e$ **in** e	*Polymorphism*

The first three constructs form the traditional lambda calculus [Barendregt et al., 1992]. Lambda calculus, though simple, is Turing complete in the sense that it is as powerful as Turing machines in expressing computations [Davis, 2004]. Thanks to its simplicity and expressivity, lambda calculus is widely used in programming language research.

To simplify the presentation, we introduce the construct v for referring to con-

stants, for example, numerical values (`1`), boolean values (`True`), and so on. Note that the introduction of v doesn't increase the expressiveness of the lambda calculus, which represents constants through encodings [Pierce, 2002]. The **let** construct is very useful for writing complicated expressions by giving names to subexpressions. Each let expression **let** x = e **in** e' consists of three parts, the variable x, the binding e, and the body e'.

The following table shows some expressions written in lambda calculus extended with pairs.

Expressions	Meanings of the expressions
`1`	The numeric constant 1
`not`	The constant function to negate boolean values
$\lambda x.x$	The identity function
$(\lambda x.x)$ `1`	Applying the identity function to 1
`not 1`	Computing the negation of 1
let `f` = $\lambda x.$`1` **in** (`f 1,f True`)	Applying `f` to arguments of different types

We observe that while `not 1` is well formed, its evaluation will fail[1]. Thus, we need a mechanism to distinguish "good" expressions from "bad" expressions, which when evaluated will cause runtime errors. A most widely used such mechanism is type checking, or type inference when type annotations may be omitted.

[1]The meaning of "evaluation" depends heavily on the semantics of reduction [Pierce, 2002]. Since the presentation of variational typing doesn't rely on a formal definition of evaluation, we will not discuss it in this dissertation.

2.1.1 Type System

The core part of a type checking algorithm is a type system that relates expressions to types. A type system usually consists of two parts: the language for types, or *types* for short and the *rules* that assign types to expressions. HM has the following type syntax.

$$\tau ::= \gamma \mid a \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall \overline{a}.\tau$$

The types are stratified into two layers: the monomorphic types, ranged over by τ, and the polymorphic types, ranged over by σ. We use γ to range over constant types, for example, `Int` and `Bool`. We use a to range over type variables. A type variable can be substituted with any monomorphic type. The function type $\tau_1 \to \tau_2$ characterizes a function that takes arguments of the type τ_1 and returns values of the type τ_2. The polymorphic types introduce type schemas $\forall \overline{a}.\tau$ that allow type variables to be universally quantified. We use the notation $\overline{a}$ to denote a list of type variables. We extend this notation to other objects and relations. While a type variable in a monomorphic type can be instantiated only once, a type variable in a type schema can be instantiated an arbitrary number of times. For example, the type $\forall a.a \to a$ can be instantiated to `Int` $\to$ `Int`, `Bool` $\to$ `Bool`, and (`Bool` $\to$ `Int`) $\to$ (`Bool` $\to$ `Int`). From time to time, we need to collect all the free type variables in a

type. The following function $FV(\sigma)$ implements this functionality.

$$\begin{aligned} FV(\gamma) &= \varnothing \\ FV(a) &= \{a\} \\ FV(\tau_1 \to \tau_2) &= FV(\tau_1) \cup FV(\tau_2) \\ FV(\forall \overline{a}.\tau) &= FV(\tau) - \overline{a} \end{aligned}$$

We use Γ to range over type assumptions, which are binary relations mapping expression variables to type schemas. The definition of FV extends naturally to Γ by collecting all the free type variables of the types in the codomain of Γ. We use θ to range over substitutions, which are binary relations mapping type variables to monomorphic types. The application of a type substitution to a type schema is written as $\theta(\sigma)$ and replaces free type variables in σ by the corresponding images in θ. The definition is as follows.

$$\begin{aligned} \theta(\gamma) &= \gamma \\ \theta(a) &= \begin{cases} a & \text{if } a \notin dom(\theta) \\ \tau & \text{if } a \mapsto \tau \in \theta \end{cases} \\ \theta(\tau_1 \to \tau_2) &= \theta(\tau_1) \to \theta(\tau_2) \\ \theta(\forall \overline{a}.\tau) &= \forall \overline{a}.\theta_{/\overline{a}}(\tau) \end{aligned}$$

The notation $\theta_{/\overline{a}}$ removes all the mappings $a \mapsto \tau$ in θ if a is in $dom(\theta)$.

The rules for assigning types to expressions are presented in Figure 2.1. The typing relation has the judgment $\Gamma \vdash^H e : \tau$, meaning that under the type environment Γ, the expression e has the type τ.

The rule HM-Con states that if v has the type γ, then the relation $\Gamma \vdash^H v : \gamma$ is satisfied. If the variable x has the type $\forall \overline{a}.\tau_1$ in Γ, then x is said to have any type by instantiating type variables $\overline{a}$ in τ_1. This fact is conveyed through the rule

$$\boxed{\Gamma \vdash^H e : \tau}$$

$$\text{HM-Con}\ \frac{\nu \text{ is of type } \gamma}{\Gamma \vdash^H \nu : \gamma} \qquad \text{HM-Var}\ \frac{\Gamma(x) = \forall \overline{a}.\tau_1 \qquad \tau = \{\overline{a \mapsto \tau'}\}(\tau_1)}{\Gamma \vdash^H x : \tau} \qquad \text{HM-Abs}\ \frac{\Gamma, x \mapsto \tau \vdash^H e : \tau'}{\Gamma \vdash^H \lambda x.e : \tau \to \tau'}$$

$$\text{HM-App}\ \frac{\Gamma \vdash^H e_1 : \tau_2 \to \tau \qquad \Gamma \vdash^H e_2 : \tau_2}{\Gamma \vdash^H e_1\ e_2 : \tau}$$

$$\text{HM-Let}\ \frac{\Gamma, x \mapsto \tau \vdash^H e : \tau \qquad \overline{a} = FV(\tau) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall \overline{a}.\tau \vdash^H e' : \tau'}{\Gamma \vdash^H \textbf{let}\ x = e\ \textbf{in}\ e' : \tau'}$$

Figure 2.1: Typing rules of HM.

HM-Var. Note that in the rule, we write $\{\overline{a \mapsto \tau'}\}$ to form a new type substitution. To determine the type of the function $\lambda x.e$, we first assume a type τ for x in e. If e has the type τ' under Γ and the assumption $x \mapsto \tau$, then $\lambda x.e$ has the type $\tau \to \tau'$. For example, to type $\lambda x.x$, we first assume that x has the type a. With that, the body of the function x obviously has the type a. Thus, the function $\lambda x.x$ has the type $a \to a$.

The rule HM-App deals with function applications. For a function application $e_1\ e_2$ to be well typed, two conditions need to be satisfied. First, the function e_1 must have a function type. In the rule, we write the function type $\tau_2 \to \tau$. We call τ_2 and τ the domain type and the return type of the function type, respectively. Second, the type of the argument e_2 must match the domain type of the function. In the rule, we use two occurrences of τ_2 to express this relation. If both conditions are satisfied, then the application has the type τ.

One specialty of **let** expressions is that they introduce parametric polymorphism, which means that one same piece of code may be used to deal with expressions of different types. Without parametric polymorphism, the following expression is rejected although its evaluation will not cause any error.

$$e = \textbf{let } \texttt{f} = \lambda x.\texttt{1} \textbf{ in } (\texttt{f 1}, \texttt{f True})$$

Without parametric polymorphism, the subexpression `f` will have the type $a \to$ `Int`. When it's first referred to in `f 1`, a needs to be instantiated to `Int` for `f 1` to be well typed. This essentially makes `f` have the type `Int` $\to$ `Int`. This instantiation is effective in the whole body of the let expression. Now what happens if we want to type `f True`? We observe that `f` has the type `Int` $\to$ `Int` but `True` has the type `Bool`. According to HM-APP, `f True` is ill typed because `Int` doesn't match `Bool`.

What prevents e from being well typed? The problem is that we want to instantiate the type variable a to two different types in the body of e. However, we can instantiate a type variable just once. Thus, to make e well typed, we need the ability of instantiating the type of `f` with different types. We can realize this by assigning `f` a polymorphic type $\forall a.a \to$ `Int`. Moreover, whenever we refer to `f`, we instantiate it independently. To type `f 1` we instantiate a with `Int`. To type `f True` we instantiate a with `Bool`. We assign `f` a polymorphic type by universally quantifying its type variable a. However, one caveat is that we shouldn't generalize all type variables of the binding expression. Only type variables that don't occur in the type environment can be generalized.

This idea of typing **let** expressions is captured in the rule HM-LET, which pro-

ceeds in three steps. First, retrieve the type of the binding. Second, generalize the retrieved type by universally quantifying certain type variables. Third, compute the type of the body with the extended assumption that the variable has the generalized type. The overall type of a **let** expression is the type of its body.

2.1.2 Type Inference

Type inference solves the problems of inferring types for expressions without type annotations. In type checking, we have type information for each part of the expression. For example, in the expression `succ 1`, we know that the type of `succ` is $\texttt{Int} \to \texttt{Int}$ and that of `1` is `Int`. Since `succ 1` is an application, we apply the HM-APP rule to type `succ 1`. We can check that both the premises of HM-APP are satisfied, thus `succ 1` is well typed and has the type `Int`. In type inference, type information for some program parts is missing. For example, in the problem of typing the expression $(e_1\ e_2)\ e_3$ under the following assumptions, the type of e_1 contains type variables, which need to be figured out to decide the type of the expression.

$$
\begin{aligned}
&e_1 : (a \to a) \to b \to b \\
&e_2 : \texttt{Int} \to \texttt{Int} \\
&e_3 : \texttt{Bool}
\end{aligned}
$$

In general, type inference poses the following challenges.

1. Solving equations involving unknown types. For the given example, we first need to decide whether $e_1\ e_2$, the function of the whole expression, is well typed. Since $e_1\ e_2$ is a function application, we try to apply the HM-APP rule

to derive its type. The problem is that the type of e_1 contains type variables a and b, and we need to figure them out before we can apply rule HM-APP. The premises of the HM-APP rule specify the relation between the types of e_1 and e_2, leading to the following unification problem

$$(a \to a) \to b \to b =^{?} (\texttt{Int} \to \texttt{Int}) \to a_1 \tag{2.1}$$

where the type variable a_1 is a placeholder denoting the result type of the function application $e_1\ e_2$ if it is well typed. Our goal of solving this unification problem is to find a mapping θ such that the following relation is satisfied.

$$\theta((a \to a) \to b \to b) = \theta((\texttt{Int} \to \texttt{Int}) \to a_1)$$

Many solutions exist, and a potential solution is given below.

$$\theta_1 = \{a \mapsto \texttt{Int}, b \mapsto \texttt{Int}, a_1 \mapsto \texttt{Int} \to \texttt{Int}\}$$

We say that θ_1 is a solution because when we apply it to both sides of the unification problem, they both become $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}$.

With θ_1, the result type of $e_1\ e_2$ is $\theta_1(a_1)$, which is $\texttt{Int} \to \texttt{Int}$. Now if we apply $e_1\ e_2$ to e_3, we encounter a type error because the domain type `Int` doesn't match the type of the argument, which is `Bool`. Thus, the expression $(e_1\ e_2)\ e_3$ is ill typed. However, the expression becomes well typed when using a different mapping. This leads to the second challenge below.

2. Maintaining generality. In general, given a unification problem, finding a solution is insufficient. What we need is the best solution. The reason that

θ_1 makes the expression $(e_1\ e_2)\ e_3$ ill-typed is that it is too restricted. Let's investigate the situation with the following less restricted mapping.

$$\theta_2 = \{a \mapsto \texttt{Int}, a_1 \mapsto b \to b\}$$

It's easy to verify that θ_2 is also a solution for the unification problem (2.1). Moreover, with θ_2, the result of $e_1\ e_2$ is $b \to b$. To type $(e_1\ e_2)\ e_3$, we have to solve the following additional unification problem.

$$b \to b =^? \texttt{Bool} \to a_2 \tag{2.2}$$

where the type a_2 denotes the result type of applying $e_1\ e_2$ to e_3. For this unification problem, there is a unique solution presented below.

$$\theta_3 = \{b \mapsto \texttt{Bool}, a_2 \mapsto \texttt{Bool}\}$$

With θ_2 and θ_3, the overall type of $(e_1\ e_2)\ e_3$ is `Bool`.

Robinson's unification algorithm addresses both challenges [Robinson, 1965]. Given two monomorphic types, it returns a most general unifier when the given types are unifiable. Given a unification problem, a unifier is a mapping that makes both sides of the unification problem the same. We use ξ to range over unifiers. Since this work has a close relation with Robinson's unification algorithm, I present it in Figure 2.2.

The operation $\circ$ in Figure 2.2 composes two unifiers into one and is defined as follows.

$$\xi_2 \circ \xi_1 = \xi_2 \cup \{a \mapsto \xi_2(\tau) | a \mapsto \tau \in \xi_1\}$$

It is easy to verify that for the unification problems (2.1) and (2.2), $\mathscr{U}_R$ returns θ_2

$$\mathcal{U}_R : (\tau_L, \tau_R) \rightharpoonup \xi$$

$$\mathcal{U}_R(a, a) = \varnothing$$

$$\mathcal{U}_R(a, \tau)$$
$$\quad \mid a \notin FV(\tau) = \{a \mapsto \tau\}$$
$$\quad \mid \text{otherwise} = \text{fail}$$

$$\mathcal{U}_R(\tau, a) = \mathcal{U}_R(a, \tau)$$

$$\mathcal{U}_R(\tau, \tau) = \varnothing$$

$$\mathcal{U}_R(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) = \xi \leftarrow \mathcal{U}_R(\tau_1, \tau_3)$$
$$\text{return } \mathcal{U}_R(\xi(\tau_2), \xi(\tau_4)) \circ \xi$$

$$\mathcal{U}_R(\tau_1, \tau_2) = \text{fail}$$

Figure 2.2: Robinson's unification algorithm.

and θ_3, respectively. In general, $\mathcal{U}_R$ is sound, complete, and most general [Robinson, 1965].

Based on Robinson's unification algorithm, Damas and Milner [1982] developed the type inference algorithm $\mathcal{W}$ for their Hindley-Milner type system. Algorithm $\mathcal{W}$ is the foundation for the variational type inference and partial variational type inference algorithms. We thus present it in Figure 2.3. The algorithm relies on the notion of fresh type variables, which are simply type variables that haven't been used before.

The algorithm is sound, complete, and principal for HM [Damas and Milner, 1982]. For the example expression $(e_1\ e_2)\ e_3$ introduced in the beginning of this section, it is easy to verify that $\mathcal{W}$ computes `Bool` as the result type.

We make two observations about algorithm $\mathcal{W}$. First, the inference result of the left part of an expression has an impact on the inference of the right part, but

$\mathcal{W} : \Gamma \times e \to \xi \times \tau$

$\mathcal{W}(\Gamma, x) =$
  $\forall \overline{a}.\tau \leftarrow \Gamma(x)$
  return $(\varnothing, \{\overline{a \mapsto b}\}(\tau))$ where $\overline{b}$ fresh

$\mathcal{W}(\Gamma, \lambda x.e) =$
  $(\xi, \tau) \leftarrow \mathcal{W}((\Gamma, x \mapsto a), e)$ where a fresh
  return $(\xi, \xi(a \to \tau))$

$\mathcal{W}(\Gamma, e_1\ e_2) =$
  $(\xi_1, \tau_1) \leftarrow \mathcal{W}(\Gamma, e_1)$
  $(\xi_2, \tau_2) \leftarrow \mathcal{W}(\xi_1(\Gamma), e_2)$
  $\xi \leftarrow \mathcal{U}_R(\xi_2(\tau_1), \tau_2 \to a)$ where a fresh
  return $(\xi \circ \xi_2 \circ \xi_1, \xi(a))$

$\mathcal{W}(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) =$
  $(\xi_1, \tau_1) \leftarrow \mathcal{W}(\Gamma, e)$
  $\overline{a} \leftarrow FV(\tau) - FV(\xi_1(\Gamma))$
  $(\xi_2, \tau_2) \leftarrow \mathcal{W}(\xi_1(\Gamma, x \mapsto \forall \overline{a}.\tau), e_2)$
  return $(\xi_2 \circ \xi_1, \tau_2)$

Figure 2.3: The type inference algorithm $\mathcal{W}$.

not vice versa. This is clear from the rule for the application $e_1\ e_2$. The algorithm first infers the type of e_1, and then infers the type for e_2 under the updated environment. Second, the algorithm uses Robinson's algorithm to solve unification problems. As Robinson's algorithm is most general, it means that algorithm $\mathcal{W}$ places only necessary constraints on the subexpressions seen so far. These two facts have a huge impact on error localization when expressions are ill typed, as we shall see in Section 3.3.

2.2 Variation Representations

This section presents a choice-based variation representation, the other important component for presenting variational typing. Section 2.2.1 introduces the elements of the Choice Calculus. Section 2.2.2 presents lambda calculus with variation constructs. We call this language variational lambda calculus (VLC). Section 2.2.3 discusses the semantics of VLC by relating VLC expressions and the lambda calculus expressions they encode.

2.2.1 Elements of the Choice Calculus

The variation representation is based on Erwig and Walkingshaw's work on the *choice calculus* [Erwig and Walkingshaw, 2011], a fundamental representation of software variation designed to improve existing variation representations and serve as a foundation for theoretical research in the field. The fundamental concept in the choice calculus is the *choice*, a construct for introducing variation points in the code. Choices are associated with names, which are used to relate different choices. Choices with the same name are synchronized and those with different names are independent. In this dissertation, we assume that all choices are globally scoped, although choice calculus comes with a *dimension* construct to structure choices [Erwig and Walkingshaw, 2011].

As an example, suppose we have two different ways to represent some part of a program's state. In one version of the program the state is binary so we use a boolean representation; in another version of the program there are more possible

states, so we use integers. The decision of which state to use is *static*—that is, we make the decision before the program runs. In our code, when we initialize this part of the state, we have to somehow choose between these two different representations. This is expressed in the choice calculus as a choice.

```
x = Rep⟨True,1⟩
```

The two different alternatives are tagged with a name, *Rep*, that stands for the decision to be made about the representation.

Now suppose we have to inspect the value of `x` at some other place in the program. We have to make sure that we process the values with a comparison operator of the right type, as indicated in the example below.

```
if Rep⟨not x,x < 1⟩ then ...
```

This choice ensures that `not` is applied if `x` uses the boolean representation and the numeric comparison is used when `x` is an integer. Here the choice's name comes critically into play. Different choices in different parts of the program will be synchronized if they have the same name. This reflects the fact that we want to make the decision about the state representation once, then reuse this decision consistently throughout our code.

So what is the type of `x`? It can be either `Bool` or `Int`, depending on the decision made for the dimension *Rep*. We can therefore express the type of `x` also using a choice.

```
x : Rep⟨Bool,Int⟩
```

Choices can have more than two alternatives, but all choices with the same name must have the same number of alternatives. Of course, a variational program can have choices with many different names, and these can be arbitrarily nested.

Like CPP, the choice calculus is principally agnostic about the language that is being annotated by choices. However, the choice calculus is a much more structured representation of variability than CPP since:

(1) it annotates the abstract syntax tree, rather than the plain text of the concrete syntax, ensuring that all variants that can be generated are syntactically correct;

(2) choices provide a more restricted and regular form of variability than arbitrary boolean conditions on macros; and

(3) it ensures that variation points are consistent in the sense that all choices in the same dimension must have the same number of alternatives.

2.2.2 Lambda Calculus with Variations

The syntax of VLC is given in Figure 2.4. To simplify the presentation of typing and the equivalence rules, we restrict choices to the binary case. This is not a fundamental limitation since it is easy to simulate an n-ary choice by nesting $n-1$ binary choices.

The first five VLC constructs, *variable*, *abstraction*, *application*, *constant*, and *let* are as in HM. The *choice* construct is from the choice calculus and was explained in the previous subsection.

Neither VLC nor the choice calculus provide direct support for optional expressions. This is because the alternative model of variation is more general, although

e	::=	x	*Variable*
	|	$\lambda x.e$	*Abstraction*
	|	$e\ e$	*Application*
	|	v	*Constant*
	|	**let** $x = e$ **in** e	*Polymorphism*
	|	$D\langle e, e\rangle$	*Choice*

Figure 2.4: VLC syntax.

in the case of VLC, it can sometimes lead to redundancy. For example, suppose we want to represent a choice between the function `even` or the function `even` applied to the constant `3`. With explicit support for optionality via empty expressions, we might represent this as `even` $A\langle$`3`$,\epsilon\rangle$, where ϵ denotes an empty expression. However, notice that we can't just include ϵ anywhere we allow an expression—for example, the body of an abstraction cannot be empty. Therefore, rather than treat optionality specially, we require the scope of the choice to be expanded. So the above VLC expression would be represented as $A\langle$`even,even 3`$\rangle$.

If a VLC expression does not contain any choices (that is, it is a regular lambda calculus expression with constants), we say that the expression is *plain*.

2.2.3 Semantics of Variational Lambda Calculus

Conceptually, a VLC expression represents a set of related lambda calculus expressions. It is important to stress again that the choice calculus constructs in VLC describe *static* variation in lambda calculus expressions. That is, we will not extend the semantics of lambda calculus to incorporate choices. Rather, the

$$
\begin{aligned}
\lfloor v \rfloor_s &= v \\
\lfloor x \rfloor_s &= x \\
\lfloor \lambda x.e \rfloor_s &= \lambda x.\lfloor e \rfloor_s \\
\lfloor e_1\ e_2 \rfloor_s &= \lfloor e_1 \rfloor_s\ \lfloor e_2 \rfloor_s \\
\lfloor D\langle e_l, e_r \rangle \rfloor_s &= \begin{cases} \lfloor e_l \rfloor_s & \text{if } s = D \\ \lfloor e_r \rfloor_s & \text{if } s = \tilde{D} \\ D\langle \lfloor e_l \rfloor_s, \lfloor e_r \rfloor_s \rangle & \text{otherwise} \end{cases} \\
\lfloor \textbf{let } x = e \textbf{ in } e' \rfloor_s &= \textbf{let } x = \lfloor e \rfloor_s \textbf{ in } \lfloor e' \rfloor_s
\end{aligned}
$$

Figure 2.5: Selection / variation elimination.

semantics of a VLC expression is a mapping from configurations to plain lambda calculus expressions encoded in the VLC expression.

To produce a plain variant, we must repeatedly eliminate dimensions of variation until we obtain an expression with no choices. For each dimension D, we can select either left or right, which will replace each choice in dimension D with its left or right alternative, respectively. We write $\lfloor e \rfloor_D$ to indicate choosing the left alternatives of all choices in dimension D and $\lfloor e \rfloor_{\tilde{D}}$ to indicate choosing the right alternatives. We call D and $\tilde{D}$ *selectors*, and range over them with the metavariable s. Since each selection eliminates a dimension of variation, we also refer to this operation as *variation elimination*. The operation is defined in Figure 2.5. Most cases simply propagate the selection to subexpressions. The interesting case is for choices, where the choice is eliminated and replaced by one of its alternatives if the selector is of the corresponding dimension.

We call a set of selectors a *decision*, and range over decisions with the metavariable δ. The semantics of VLC is then defined as a mapping from deci-

sions to plain lambda calculus expressions. The definition is based on repeated selection with selectors taken from decisions, that is, $\lfloor e \rfloor_\delta = \lfloor\lfloor \ldots \lfloor e \rfloor_{s_1} \ldots \rfloor_{s_{n-1}} \rfloor_{s_n}$, where $\delta = \{s_1, s_2, \ldots, s_n\}$. A decision that eliminates all choices in e is called *complete*.

We want the semantics to be "robust" in the sense that selection with a dimension that does not occur in e is well defined but does not eliminate any choice in e. Thus, for any given expression e, a selector in a decision δ can play two different roles when used in a selection. It can either be *relevant* and eliminate at least one choice from e or *irrelevant* and not change e. Since there are infinitely many irrelevant selectors for any expression e, we define the semantics in two conceptual steps. First, we construct a mapping with decisions containing only relevant selectors, and then extend it to account for irrelevant selectors.

A decision δ is *minimally complete* with respect to an expression e if it satisfies the following two conditions:

(a) $\lfloor e \rfloor_\delta$ is plain,

(b) $\forall s \in \delta : \lfloor e \rfloor_{\delta - \{s\}}$ is not plain.

Condition (a) ensures that δ eliminates all variability in e (that is, δ is complete with respect to e), and condition (b) ensures that δ is minimal in the sense that it does not contain any irrelevant selectors.

The semantics can now be defined by mapping all minimally complete decisions to the plain expressions they select while allowing each decision to also include irrelevant dimensions. In the definition given below we use the function

$|\delta| = \{|s| \mid s \in \delta\}$ (where $|D| = |\tilde{D}| = D$) to extract the set of dimensions underlying a set of selectors.

$$[\![e]\!] = \{(\delta \cup \delta', \lfloor e \rfloor_\delta) \mid \delta \text{ is minimally complete wrt. } e \text{ and } |\delta| \cap |\delta'| = \varnothing\}$$

To illustrate this definition, we give the semantics of the expression $A\langle e_1, B\langle e_2, e_3\rangle\rangle$ where e_1, e_2, and e_3 are plain expressions, and where we use the shorthand notation $\{\delta \rightarrowtail e\}$ to denote the set $\{(\delta \cup \delta', e) \mid |\delta| \not\subseteq |\delta'|\}$.

$$[\![A\langle e_1, B\langle e_2, e_3\rangle\rangle]\!] = \{\{A\} \longmapsto e_1\} \cup \{\{\tilde{A}, B\} \rightarrowtail e_2\} \cup \{\{\tilde{A}, \tilde{B}\} \longmapsto e_3\}$$

Note that due to the minimality constraint, dimension B does not appear in the decision of the first entry since it is irrelevant.

Chapter 3: Literature Review

The work of variational typing and applications is related to many research activities: type system design, type checking/inference of generated programs, type checking software product lines, and debugging type errors. This chapter collects the work related to the latter three activities. The work related to type system design will be discussed once variational typing is presented in Chapters 4 and 7.

3.1 Type Checking Software Product Lines

In the context of software product lines (SPLs), a lot of work has been done to improve the type checking of generated products and avoid the brute-force strategy of typing each product individually. A main difference between my work and the work in this area is that I solve the type inference problem while previous work focused on the type checking problem. Variational type inference poses some particular challenges, for example, solving variational unification problems and locating variants containing type errors. We address these challenges in Chapters 5 and 7.

SPLs are usually implemented through two very different approaches: *annotative* and *compositional* [Walkingshaw, 2014]. The annotative approach usually involves two languages: an object language for implementing software function-

alities and a metalanguage for managing software variations. The name "annotative" implies that the metalanguage is used directly to annotate the object code. This approach is very flexible since annotations can be attached to virtually any object language construct. Given an SPL, products are generated by evaluating its annotations. When an annotation is evaluated to true, its associated code is included in the resulting product. Instead, code corresponding to annotations that are evaluated to false is not included. In any case, the annotations are not part of the product. C Preprocessor (CPP) [GNU Project, 2009], the most widely used metalanguage for annotating source code, uses macros and #ifdef and related constructs to manage variation. Each macro is often referred to as a feature. Thus, selecting a feature means to define that macro and include the related code into the generated product.

In the compositional approach, only one language is used to implement SPLs. However, such a language is often obtained by extending an existing language with certain constructs for achieving high compositionality, for example, mixins [Bracha and Cook, 1990; Batory et al., 2004], aspects [Elrad et al., 2001; Mezini and Ostermann, 2003, 2004], and refinements [Apel et al., 2010]. An SPL is usually decomposed into the mandatory code base and the optional component features. Products are generated by choosing certain features and applying them to the code base. The ordering of applying features is significant.

Usually, each SPL is also associated with a *feature model* specifying the relationships between its features. The feature model can be described through a use of feature diagrams [Kang et al., 1990] or logics [Batory, 2005; Schobbens et al.,

2006]. A feature model describes what products are valid. For example, a feature model that contains the constraint A ⇒ B says that any product that includes the feature A must also include the feature B. The purpose of type checking software product lines is to ensure that all valid products are type correct.

3.1.1 Type Checking Annotative Software Product Lines

Kästner et al. [2012a] describe a type system for SPLs implemented in Colored Featherweight Java (CFJ). In CFJ, parts of a Featherweight Java (FJ) program can be "colored", marking them as optional and associating them with a particular feature. FJ fulfills a role in object-oriented language research similar to lambda calculus in functional language research. The following code presents a minimal FJ program, which defines a class named `Database`. The class includes a field `storage` of the type `Storage`. It also includes a method named `insert`.

```
class Database{
    Storage storage;

    void insert(Object key, Object data, Txn txn){
        storage.set(key, data, txn.getLock()) ;
    }
}
```

CFJ programs are very similar but allow certain parts to be optional [Kästner et al., 2012a]. However, instead of adding annotations to code fragments directly, they use an *annotation table AT* to map each optional code fragment to an annotation. Annotations can be viewed as propositional formulas over features. Figure 3.1 presents a CFJ program, which we name `DB`. Instead of writing

```
class Database{
    Storage storage;

    void insert(Object key, Object data
                ,Txn txn                                          TRANSACTIONS
               ){
        storage.set(key, data
                    ,txn.getLock()                                TRANSACTIONS
                   );
    }
}

class Txn{...}                                                    TRANSACTIONS

class Storage{
    boolean set(Object key, Object data, Txn txn) {...}            PERSISTENT
}
```

Figure 3.1: A CFJ program named DB.

down an explicit annotation table, I directly attach annotations to optional code by putting them at the end of the corresponding lines. For example, the annotation PERSISTENT means that the method set of Storage is present only when the feature PERSISTENT is chosen in generating the product. If we wrote AT explicitly, it would contain the entry AT(Storage.set) = PERSISTENT.

Assume that the feature model associated with DB contains the single constraint TRANSACTIONS ⇒ PERSISTENT. Two valid products may be generated. The first product with TRANSACTIONS selected is presented in Figure 3.2. This product includes the code associated with either TRANSACTIONS or PERSISTENT.

The second product, given in Figure 3.3, contains the code associated only with PERSISTENT.

```
class Database{
    Storage storage;

    void insert(Object key, Object data
                ,Txn txn){
        storage.set(key, data
                    ,txn.getLock() );
    }
}

class Txn{...}

class Storage{
    boolean set(Object key, Object data, Txn txn) {...}
}
```

Figure 3.2: The product generated from DB with TRANSACTIONS selected.

We can observe that while the first product is well typed, the second product contains a type error. The reason is that the `insert` method invokes `set` with two arguments while `set` requires three arguments. Another reason is that the method `set` refers to the undefined class `Txn`. One of the goals of the type system developed by Kästner et al. [2012a] is to catch such kinds of errors.

Their type system thus is a combination of the type system for FJ and a path analysis that checks whether elements that are referred to are reachable. Besides normal type checking, each typing rule also deals with reachability checking and annotation propagation. For example, the rule for typing variable references in CFJ consists of two parts, denoted with different background colors. The first part that has white background is inherited from the rule for typing variables in FJ. The second part that has light gray background deals with path reachability. The

```
class Database{
    Storage storage;

    void insert(Object key, Object data){
        storage.set(key, data);
    }
}

class Storage{
    boolean set(Object key, Object data, Txn txn) {...}
}
```

Figure 3.3: The product generated from DB with TRANSACTIONS unselected.

A' in the first premise specifies the annotation that makes the variable x available. The second premise states that to access the variable x in the current context, its associated annotation A must be tighter than the annotation of x. This idea is expressed through the logical implication $A \rightarrow A'$. The conclusion says that with the current annotation A and the type environment Γ, x has the type C.

$$\frac{\mathrm{x}: \mathrm{C} \text{ with } A' \in \Gamma \qquad A \rightarrow A'}{A; \Gamma \vdash \mathrm{x}: \mathrm{C}}$$

There has also been some work on statically checking variational C programs containing CPP annotations. Initial work in this area was done by [Aversano et al., 2002] who demonstrate the widespread use of conditionally declared variables with potentially different types and the difficulty in ensuring that they are used correctly in all variants. As a solution, they propose the construction of an extended symbol table that contains conditions under which each symbol is defined and the corresponding type. [Kenner et al., 2010; Kästner et al., 2012b] provide

a working implementation of essentially this approach in TypeChef[1], although it currently ensures only that symbol references are satisfied in all variants and that no symbols are redefined. TypeChef's ultimate goal is to be able to efficiently ensure the type correctness of all variants of CPP-annotated C programs, which becomes promising with the work of variability-aware parsing [Kästner et al., 2011; Gazzillo and Grimm, 2012]. There is a huge amount of engineering overhead in such a project, not related to variational type systems, because of CPP's highly unstructured variation representation. A macro's setting can change several times throughout a single run of the C preprocessor, making it much more difficult to even determine which code corresponds to a particular variant.

3.1.2 Type Checking Compositional Software Product Lines

Thaker et al. [2007] present an approach for type checking SPLs based on the safe composition of type-correct modules [Delaware et al., 2009a,b]. This is given as a tool implemented in the AHEAD framework for feature-oriented software development [Batory et al., 2004], where each feature is implemented in a separate module. These modules can then be selectively composed into products, and the set of all such possible products forms an SPL. Safe composition of features is achieved in two steps. In the first, each module is compiled and checked to see whether it satisfies a lightweight global consistency property. After that, constraints between particular modules are checked.

[1] http://ckaestne.github.io/TypeChef/

Other work on ensuring the type correctness of generated products within the compositional approach include the work of [Apel and Hutchins, 2008], which describes feature composition formally with a new calculus, and the work of [Chae and Blume, 2008], which ensures that the types of composed features will match.

While CFJ provides type checking support for annotative variation in Featherweight Java, Feature Featherweight Java (FFJ) [Apel et al., 2008] and FFJ$_{PL}$ [Apel et al., 2010] provide type checking support for compositional variation in the language. The goal of these languages is to explore the flexibility of class refinement and module composition and to ensure the type correctness of whole software product lines.

In FFJ$_{PL}$, each feature has its own module implementing the functionalities of that feature. Figure 3.4 shows as an example the `EC` SPL in FFJ$_{PL}$. We observe that `EC` includes four features, and correspondingly, four modules. A key concept in FFJ is class refinement, indicated by the keyword `refine` in the code. When a class `C1` refines another class `C2`, all fields in `C1` will be merged into `C2` when no name conflicts occur during the merging. Otherwise, the refinement fails. The rule for merging methods consists of three parts. (1) Methods are merged together when they have pairwise different names. (2) If two classes share the same signature for some method and that method is specified with the `override` modifier in `C1`, the method in `C1` will replace the one in `C2` during merging. (3) Merging fails for all other cases.

EMAILCLIENT

```
class Msg extends Object{
    String serialize(){... }
}
class Trans extends Object{
    boolean send(Msg msg){... }
}
```

TEXT

```
refine class Trans{
    Timer tmr;
    Unit receive(Msg msg){
        return Display().render(msg);
    }
}
class Display{
    Unit render(Msg msg) {...}
}
```

MOZILLA

```
refine class Display{
    override Unit render(Msg msg) {...}
}
```

SAFARI

```
refine class Display{
    override Unit render(Msg msg) {...}
    override colorful(Msg msg) {...}
}
```

Feature model:

$$\begin{aligned} \textsc{EmailClient} &\Rightarrow \textsc{Text} \\ \textsc{Text} &\Rightarrow \textsc{Mozilla} \vee \textsc{Safari} \\ \textsc{Mozilla} &\Rightarrow \neg\textsc{Safari} \\ \textsc{Safari} &\Rightarrow \neg\textsc{Mozilla} \end{aligned}$$

Figure 3.4: An email client SPL named EC in FFJ$_{PL}$.

```
class Msg extends Object{
    String serialize(){... }
}

class Trans extends Object{
    boolean send(Msg msg){... }
    Timer tmr;
    Unit receive(Msg msg){
        return Display().render(msg);
    }

class Display{
    Unit render(Msg msg) {...} /*The method is from the MOZILLA module*/
}
```

Figure 3.5: The product generated from EC with MOZILLA selected.

According to the feature model in Figure 3.4, the SPL presented in that figure can generate two different products. The first product, with the presence of the MOZILLA feature, is given in Figure 3.5. We have merged fields and methods according to the refinement merging rule described above.

While the first product is well typed, the second product that includes the feature SAFARI is ill typed. We present part of the second product in Figure 3.6, which makes it clear why it's ill typed.

The problem is that while the module SAFARI tries to refine the class `Display` by overriding the method `colorful`, that method is missing in the original `Display` class defined in the module TEXT. The work of FFJ$_{PL}$ is to prevent such kinds of errors. For the SPL in Figure 3.4, FFJ$_{PL}$ reports a type error.

TEXT

```
...
class Display{
    Unit render(Msg msg) {...}
}
```

SAFARI

```
refine class Display{
    override Unit render(Msg msg) {...}
    override colorful(Msg msg) {...}
}
```

Figure 3.6: The product generated from EC with SAFARI selected.

3.2 Type Checking Generated Programs

In VLC, program variation is managed by using the annotative choice constructs. Program variation can also be expressed using program generation or metaprogramming techniques, for example, MetaML [Taha and Sheard, 2000], Template Haskell [Sheard and Peyton Jones, 2002], C++ Templates [Austern, 1998], and many others [Fähndrich et al., 2006; Huang and Smaragdakis, 2008, 2011]. In this section, we will focus on their type checking aspect.

While the original goal of MetaML is to optimize program execution [Taha and Sheard, 2000], its metaprogramming mechanisms also supports program customization and generation. In particular, MetaML provides three constructs for managing program generation.

1. The bracket operator (<>) prevents code reduction[2] inside brackets. For ex-

[2]Code reductions can be understood as computations.

ample, the subexpression `2+3+4` in `<2+3+4>` during reduction shouldn't be reduced to `9`. The expression `<e>` is called a piece of code.

2. The escape operator (`~`) splices a piece of code into its surrounding context. It works by enabling reductions of escaped expressions inside brackets. Specifically, if we have `e=<...~f ...>`, then the reduction of `e` will cause that of `f`. Moreover, if `~f` reduces to `<f1>`, then `f1` will be spliced together with the code surrounding `~f` inside `e`. For example, the reduction of

   ```
   <2+~(if true then <3> else <1>)+4>
   ```

   yields the expression `<2+3+4>`. The evaluation of the `if` expression produces `<3>`, and `3` is used to produce the larger resulting expression.

3. The `run` operator forces an execution of a delayed computation, which is represented as a code expression. For example, the expression `run <2+3+4>` evaluates to `9`.

The MetaML expressions are typed, and this ensures that all generated programs are well typed. Code types are introduced to assign types to expressions. For example, when `e` has the type `Int`, `<e>` will have the type `<Int>`. If we extend the type system defined in Section 2.1.1 with code types, the following relations

hold. We use the notation `\x.x` to denote the identity function $\lambda x.x$.

$$\Gamma \vdash^H \texttt{2 : Int}$$
$$\Gamma \vdash^H \texttt{<2> : <Int>}$$
$$\Gamma \vdash^H \texttt{<<2>> : <<Int>>}$$
$$\Gamma \vdash^H \texttt{run <<2>> : <Int>}$$
$$\Gamma \vdash^H \texttt{<\x.x> : <Int} \rightarrow \texttt{Int>}$$
$$\Gamma \vdash^H \texttt{run <\x.x> : Int} \rightarrow \texttt{Int}$$

With these three operators and the type system, we can build the following program, which has the type `<Bool>` → `<Int>`.

```
vp1 x = <~(if ~x then <succ> else <\x.x+2>) 1>
```

We observe that the argument to `vp1` can only be `<true>` or `<false>`[3]. When the argument is `<true>`, the expression `<succ 1>` is generated. Otherwise, the expression `<(\x.x+2) 1>` is generated. To make this process more explicit, we show the evaluation steps of `run (vp1 <false>)` as follows.

```
run (vp1 <false>) = run (<~(if ~<false> then <succ> else <\x.x+2>) 1>)
                  = run (<~(if false then <succ> else <\x.x+2>) 1>)
                  = run (<~(<\x.x+2>) 1>)
                  = run (<(\x.x+2) 1>)   Object program is generated
                  = (\x.x+2) 1
                  = 3
```

MetaML's type system is limited in the sense that an expression may be rejected by the type system although all generated expressions are well typed. For

[3]We use the MetaML convention of writing boolean values, which begin with lower case letters.

example the following very similar expression is not typable in MetaML.

```
vp2 x = <~(if ~x then <succ> else <even>) 1>
```

With `vp2`, we can potentially generate two programs. When the argument is `<true>` and `<false>`, the generated program is `<succ 1>` and `<even 1>`, respectively. We observe that both generated programs are well typed, although `vp2` is rejected by MetaML.

This limitation is, however, not due to the design of MetaML itself, but a reflection of the limitations of static type systems in general. In other words, this limitation of MetaML is inherited from that of ML, which is statically typed and requires the branches of `if` expressions to have the same type. We further observe that MetaML uses ML's `if` expressions to realize program generation. Therefore, code alternatives in MetaML must have the same type.

Template Haskell [Sheard and Peyton Jones, 2002] implements metaprogramming in a very different manner: it introduces more language constructs and uses a different type checking mechanism. In particular, type checking in Template Haskell is delayed until after object programs are generated, abandoning the static typing paradigm [Shields et al., 1998]. This model results in a language that is more flexible, yet has fewer type correctness guarantees. In Template Haskell, it is easy to implement the `vp2` program introduced earlier, but it is also easy to write programs that generate ill typed programs.

The design of Template Haskell is, to a large extent, inspired by C++ Templates [Austern, 1998; Abrahams and Gurtovoy, 2004]. Thus, they share many

properties: both of them enjoy great flexibility but delay type checking until after programs are generated. C++ Templates enable software variation through template instantiations. By instantiating templates with different arguments, different program variants can be generated.

For example, Figure 3.7 implements the earlier introduced `vp1` in C++ Templates. The main function `vp1c` returns the function `succ` if it is instantiated with `true` and returns `succ2` otherwise. The last line calls the returned functions. Note that in `vp1c<true>()(1)`, the first pair of parentheses is used to call the function `vp1c` and the subexpression `(1)` invokes the function returned in the first function call.

```
int succ(int v) { return v + 1 ; }
int succ2(int v){ return v + 2 ; }

typedef int (*PII)(int);

template<bool v>
PII vp1c(){
    if(v)
        return succ;
    else
        return succ2;
}

cout << vp1c<true>()(1) << vp1c<false>()(1);
```

Figure 3.7: A C++ Template program generates two programs with the same type.

While we can't implement `vp2` in the same way, we can implement it through a use of class templates and their specializations [Abrahams and Gurtovoy, 2004]. One of its potential implementation is given in Figure 3.8. The code first defines

a class template, which is then specialized with different template arguments. While the notation of the function call in the last line is peculiar, the code doesn't contain any typos.

```
int succ(int v) { return v + 1 ; }
bool even(int v) { return v%2 == 0; }

typedef int (*PII)(int);
typedef bool (*PIB)(int);

template<bool v>    //Class template for vp2c
struct vp2c{
    int operator()(){ return 1; }
};

template<>
struct vp2c<true>{  //Specialized for true
    PII operator()(){ return succ; }
};

template<>
struct vp2c<false>{ //Specialized for false
    PIB operator()(){ return even; }
};

cout << vp2c<true>()()(1) << vp2c<false>()()(1);
```

Figure 3.8: A C++ Template program generates two programs with different types.

While the code `vp2c<true>()()` is executed, the function `succ` is returned. Similarly, the execution of `vp2c<false>()()` returns the function `even`. The code in Figure 3.8 demonstrates the flexibility of C++ Templates. In addition, C++ Templates provide a static type reflection mechanism [Garcia, 2008] that allow metaprograms to query the type information of template parameters. These features allow

C++ to achieve both maximum reusability and efficiency.

However, since the complete type information of the template parameters is not available statically, full type checking is delayed until after a particular program variant is generated [Stroustrup, 1994]. This means that it's not possible to statically type check all program variants without generating each one.

In the following example, there is a type error in the expression `return powN<N-1>(m)*m` because `powN<N-1>(m)` is of type `int` and `m` is of type `string`. However, the statement `return powN<N-1>(m)*m` will not be type checked until the function `powN` is instantiated since the expression `powN<N-1>(m)` uses the template parameter `N`. Therefore, the detection of the type error will be unnecessarily delayed.

```
template<int N>
int powN(string m){
    return powN<N-1>(m)*m;
}
```

Thus, although `powN` is accepted by C++, no instantiation exists to generate well-typed programs.

The model of delayed type checking has led to serious usability problems [Gregor et al., 2006]. Despite huge efforts devoted to address this issue [Reis and Stroustrup, 2005; Dos Reis and Stroustrup, 2006; Gregor et al., 2006; Siek and Taha, 2006a; Garcia and Lumsdaine, 2009; Miao and Siek, 2010; Chen and Erwig, 2014d], no satisfying solution has been proposed. Järvi et al. [2006] conclude that to achieve modular type checking for C++ templates, either the use or implementation of specialization must be constricted.

SafeGen [Huang et al., 2005] and MorphJ [Huang and Smaragdakis, 2008, 2011] provide a way to create generic Java classes whose methods are generated

by iterating over the fields and methods of other classes. This is particularly suited for defining wrappers and proxies for existing classes. Method definition in MorphJ consists of two parts: (1) the iteration pattern that determines which methods will be matched, and the resulting method name, return type, and argument types, and (2) the method body, which may refer to pattern-matching variables defined in the iteration pattern. MorphJ checks that the operations applied to pattern-matching variables are supported by assumptions introduced in the iteration pattern. This is very similar to C++ *concepts*, which describe constraints on template parameters [Gregor et al., 2006]. Both approaches ensure that generated classes will be well-typed if they are instantiated with arguments that satisfy the assumptions/constraints.

Many other techniques have been developed for ensuring the generation of well-formed and well-typed programs in mainstream object-oriented languages. Fändrich et al. [2006] propose a pattern-matching and template-based approach for writing reflective code in C#. Work on Maven [Goldman et al., 2010] and subsequent work [Disenfeld and Katz, 2012] has addressed the problem of ensuring that specified behaviors are achieved after complex aspect-weaving operations. Finally, *expanders* [Warth et al., 2006] provide a new language construct for updating the methods and fields of an existing Java class in a non-invasive way. This work achieves modular type checking by statically scoping the usage of expanders.

3.3 Debugging Type Errors

Generating informative and helpful type error messages remains a challenge for implementing type inference algorithms. The difficulty of locating real error causes in $\mathcal{W}$ [Damas and Milner, 1982] attributes to the way type inference is implemented. As already mentioned in Section 2.1.2, $\mathcal{W}$ tends to create a left-to-right bias in locating type errors [Heeren et al., 2003a; Hage and Heeren, 2009]. Consider, for example, the following very simple expression `rank`[4]. This expression is ill typed because it tries to apply the parameter `f` to values of different types, which is prevented in HM[5].

```
one = 1::Int
rank f = (f True, f one)
```

The type inference algorithm in the Glasgow Haskell Compiler (GHC) 7.8[6] is an extension of algorithm $\mathcal{W}$. If we load the expression `rank` into GHC, we receive the following message. We observe that the identified error cause is `one`, rather than `True`, which is in fact equally likely a cause of the type error.

```
Couldn't match expected type 'Bool' with actual type 'Int'
In the first argument of 'f', namely 'one'
In the expression: f one
```

While this message demonstrates a potential problem with error localization, it is not the only problem with type error diagnosing in general. In particular, the generated error messages usually contain compiler jargon that are hard to digest

[4]The expression is so named because it is ill typed in type system with rank-1 polymorphism but is well typed with higher-rank polymorphism.

[5]We focus on the discussion of debugging type errors in HIndley-Milner type system.

[6]www.haskell.org/ghc/

by those that are unfamiliar with type inference implementation details. To illustrate, consider the following example `palin`, which checks whether a list is a palindrome [Stuckey et al., 2003]. The first equation for `fold` contains a type error and should return `z` instead of `[z]`.

```
fold f z []     = [z]
fold f z (x:xs) = fold f (f z x) xs
flip f x y = f y x
rev = fold (flip (:)) []
palin xs = rev xs == xs
```

For this example, GHC produces the following error message.[7]

```
Occurs check: cannot construct the infinite type: t ~ [t]
Expected type: [[t]]
  Actual type: [t]
Relevant bindings include
  xs :: [t] (bound at Palin.hs:5:7)
  palin :: [t] -> Bool (bound at Palin.hs:5:1)
In the second argument of '(==)', namely 'xs'
In the expression: rev xs == xs
```

Hugs98 [8], another modern Haskell compiler, displays the following message.

```
*** Expression     : rev xs == xs
*** Term           : rev xs
*** Type           : [[a]]
*** Does not match : [a]
*** Because        : unification would give infinite type
```

We observe similar error messages from both compilers since they implement the standard type inference algorithm. While technically accurate, both the error messages point to the wrong error cause and explain the type error in terms of

[7] For presentation purposes, we have slightly edited the outputs of some tools by changing their indentation and line breaks.

[8] http://www.haskell.org/hugs/

unification failure. In fact, the hardness of explaining type inference failure was recognized [Wand, 1986; Johnson and Walz, 1986] soon after algorithm $\mathscr{W}$ [Damas and Milner, 1982] had been developed. It has since prompted numerous research efforts from different perspectives [McAdam, 2002b; Lee and Yi, 1998, 2000; Yang, 2001; Wazny, 2006; Lerner et al., 2007; Choppella, 2002; McAdam, 2002a; Tip and Dinesh, 2001; Haack and Wells, 2003; Heeren, 2005; Schilling, 2012]. In the following, I will briefly review these efforts by grouping different approaches according to major features they share. Beyond the Hindley-Milner type system, the problem of debugging type errors has also been studied in security programming [Weijers et al., 2014], type classes [Heeren and Hage, 2005], and Generic Java [El Boustani and Hage, 2011, 2010].

3.3.1 Reporting Single Locations

Both the examples `rank` and `palin` demonstrate a left-to-right bias of inference algorithm $\mathscr{W}$. Many approaches have tried to eliminate this bias. Examples are algorithms $\mathscr{M}$ [Lee and Yi, 2000], $\mathscr{G}$ [Eo et al., 2004], $\mathscr{W}^{SYM}$ and $\mathscr{M}^{SYM}$ [McAdam, 2002b], and $\mathscr{UAE}$ and $\mathscr{IEI}$ [Yang et al., 2000]. While algorithm $\mathscr{W}$ processes nodes in tree representations from left to right, algorithm $\mathscr{M}$ [Lee and Yi, 2000] uses a top-down fashion. Algorithm $\mathscr{G}$ [Eo et al., 2004] is a generic one that is parameterized over the way the nodes are traversed. Therefore, $\mathscr{M}$ and $\mathscr{W}$ can be viewed as instances of $\mathscr{G}$. Algorithm $\mathscr{W}^{SYM}$ is a modification of $\mathscr{W}$ by changing its rule for processing applications. For both the function and the argument, $\mathscr{W}^{SYM}$

returns a type and a substitution. These two substitutions are then merged.

In the following, we show the locations reported by various algorithms for the `rank` example.

$$\texttt{rank f = (f True, } \underbrace{\texttt{f } \overbrace{\texttt{one}}^{\mathcal{M}}}_{\mathcal{W}} \texttt{)} \quad \underbrace{\qquad\qquad\qquad}_{\mathcal{W}^{SYM} \text{ and } \mathcal{M}^{SYM}}$$

All these algorithms interpret the place of unification failure as the source of the type error. A very different approach has been taken by Johnson and Walz [1986], who used a maximum-flow technique to determine the most likely cause of the type error when type conflicts arise in a set of type equations. Specifically, when a type variable has to be unified with two conflicting types, the type variable is mapped to a disjunction of these two types, and the typing process continues. The conflict is eventually resolved by something like "usage voting", that is, whatever type a variable is unified with most often, will be selected.

3.3.2 Slicing Type Errors

In contrast to approaches that blame type errors on single locations, many slicing approaches have been developed to identify a *set* of possible error locations [Tip and Dinesh, 2001; Haack and Wells, 2003; Schilling, 2012]. The basic idea is to find *all* program positions that contribute to a type error and exclude those that do not. For example, the Skalpel[9] type error slicer for SML [Haack and Wells, 2003] produces the following result. (We have translated the program into ML for

[9] www.macs.hw.ac.uk/ultra/skalpel/

Skalpel to work.)

```
fun fold f z  []      = [z] ;
  | fold f z  (x::xs) = fold f  (f (z,x)) xs ;
fun flip f (x,y) = f (y, x) ;
fun rev xs = fold ( (flip op ::)) [] xs ;
fun palin  xs = rev xs = xs  ;
```

For `rank`, it produces the following slice.

```
val one = 1 : int ;
fun rank f = (f true,f one) ;
```

Skalpel finds type error slices in two steps. First, it generates labeled constraints for programs such that constraint solving failure can be linked back to programs. Second, it finds a minimal unsolvable constraint set if the generated constraints couldn't be solved successfully. A significant difference of its constraint generation process compared with other inference algorithms is that to represent the result type of a subexpression, it creates a fresh variable and a constraint between that fresh variable and the result type. This ensures that all subexpressions involved in the type error will be included in the slice. However, this strategy also causes the slice to include unnecessary subexpressions. For example, the variables defined for passing around an erroneous expression will be included in the slice. As we can see from the example, Skalpel includes most program parts in the slice.

Some techniques have been developed to minimize the possible locations contributing to a type error. One example is the Chameleon Type Debugger,[10] which produces the following output.

[10] `ww2.cs.mu.oz.au/~sulzmann/chameleon/`. Since Chameleon doesn't offer a type diagnosis option anymore, the result is reproduced directly from [Stuckey et al., 2003].

```
fold f z []     = [z] ;
fold f z (x:xs)= fold f (f z x) xs ;
flip f x y = f y x ;
rev = fold (flip (:)) [] ;
palin xs = rev xs = xs ;
```

Chameleon is based on constraint handling rules and identifies a minimal set of unsatisfiable constraints, from which the corresponding places in the program contributing to the type error are derived.

Following this idea of showing fewer locations, a more aggressive strategy is taken by SHErrLoc[11], which analyzes all the constraints and identifies the constraint that is most likely to cause the error by using Bayesian methods [Zhang and Myers, 2014]. For the `palin` example, it outputs the following message.

```
One typing error is identified

Expressions in the source code that appear most likely
to be wrong (mouse over to highlight code):

x [loc: 4,7-8]
a [loc: 6,30-31]
x [loc: 4,28-29]
z [loc: 3,13-14]
[z] [loc: 3,15-20]
```

For the `rank` example, SHErrLoc produces the following message, which lists three potential locations that may cause the type error. These locations correspond to the code `one` (the second occurrence), `1 :: Int`, and `True`, respectively.

```
Constraints in the source code that appear most likely to be wrong:

(variable)t_aHE == (variable)t_aHS [loc: program.hs:4,20-23]
Int == (variable)t_aHE [loc: program.hs:3,6-12]
Bool == (variable)t_aHN [loc: program.hs:4,12-16]
```

[11] http://www.cs.cornell.edu/projects/SHErrLoc/

Neubauer and Thiemann [2003; 2004] developed a type system based on discriminative sum types to record the causes of type errors. Specifically, they place two non-unifiable types into a sum type. Their system returns a set of sources related to type errors. Thus, it can be viewed as an error slicing approach. However, compared to other slicing approaches, it is not guaranteed that the returned set of locations is minimal. On the other hand, error locations reported by their approach may contain program fragments that have nothing to do with type errors. For example, a variable used for passing type information will be reported as a source of type errors if it is unified once with some sum types during the type inference process.

3.3.3 Producing More Informative Messages

The approaches presented so far only show error locations. Approaches that produce more informative error messages have also been developed.

Many approaches have focused on identifying and explaining the causes of type conflicts. Wand [1986] records each unification step so that they can be tracked back to the failure point. Duggan and Bent [1995] on the other hand record the reason for each unification that is being performed. Beaven and Stansifer [1994] and Yang [2000] produce textual explanation for the cause of the type errors.

While these explanation techniques can be useful in many cases, there are also potential downsides. First, the explanation can become quite verbose and repetitive, and the size grows rapidly as the program size increases. Second, the

explanation is inherently coupled to the underlying algorithm that performs the inference. Thus, knowledge about how the algorithms work is often needed to understand the produced messages. Third, the explanations usually lead to the failure point, which is often the result of biased unification and not the true cause of the type error. Finally, although a potential fix for the type error may lurk in the middle of the explanation chain, it's not always clear about how to exploit it and change the program.

Helium[12] is developed to assist beginners to learn Haskell. A declared focus of Helium is to generate good error messages [Heeren et al., 2003c; Heeren, 2005]. Helium locates type errors in several steps. First, it builds a type graph to represent the constraints about typing relations among different parts of the program. When constraint solving fails, it will search through the type graph to discover a constraint whose removal will make the constraint solving successful. Helium finds the most suspicious constraint based on some heuristics [Hage and Heeren, 2007].

Another distinct feature of Helium is that it frequently makes use of what it calls sibling functions [Heeren et al., 2003b]. Siblings are pairs of functions which are in some way similar or offer related functionality. Example of this include `(:)` and `(++)`, as well as `max` and `maximum` which find the maximum of two values and the maximum in a list of values, respectively. Literals can also be considered siblings, such as the string and character versions of a single letter (`"c"` and `'c'`) or the floating point and integer versions of a number. On example of using the

[12] www.cs.uu.nl/wiki/bin/view/Helium/WebHome

sibling functions to produce error messages is shown below. If we type `"abc":"def"` into the Helium interactive window, we will receive the following message.

```
Type error in constructor
 expression       : :
   type           : a      -> [a]    -> [a]
   expected type  : String -> String -> b
 probable fix     : use ++ instead
```

We observe that Helium suggests to use `(++)` over `(:)` to fix the type error. For the `palin` example introduced earlier, Helium generates the following message. We observe that the reported error location is incorrect.

```
(5,19): Type error in infix application
 expression       : rev xs == xs
 operator         : ==
   type           : a     -> a   -> Bool
   does not match : [[b]] -> [b] -> c
 because          : unification would give infinite type
```

For the `rank` example, Helium, like algorithm $\mathcal{W}$, blames `f one` as the error cause.

Instead of changing type checkers or compilers, Seminal [Lerner et al., 2006, 2007] improves error reporting by searching for a well-typed program that is similar to the ill-typed program. Seminal mainly consists of two steps. First, it locates a type error through top-down removal. Specifically, if a program is ill typed, Seminal tries to remove each top-level expression to check whether the removal of that expression will eliminate the type error. If this succeeds for a top-level expression, then seminal recursively searches within that expression. Second, for the located expressions, it performs constructive changes at that location. Examples of such changes are the removal of an argument from a function call, swapping the ordering of arguments to function calls, and so on. After new expressions are constructed, each one is type checked. Seminal then suggests changes to the users

```
(* List.combine : 'a list -> 'b list -> ('a * 'b) list *)
(* List.map : ('a -> 'b) -> 'a list -> 'b list *)
(* List.filter : ('a -> bool) -> 'a list -> 'a list *)

let map2 f aList bList =
    List.map (fun (a, b) -> f a b)
        (List.combine aList bList)

let lst = map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6]

let ans = List.filter (fun x -> x==0) lst

-- The following message is produced by Seminal
Try replacing fun (x, y) -> x + y
with fun x y -> x + y
of type int -> int -> int
within context let lst =
    map2 (fun x y -> x + y)
        [1;2;3] [4;5;6]
```

Figure 3.9: An ill typed Ocaml program and the corresponding message produced by Seminal.

by ordering all the programs that passed type checking using some heuristics. Seminal tries to find as many errors as possible.

In Figure 3.9, we present an ill typed OCaml program together with the error message generated by Seminal. In the figure, the computation of `lst` contains a type error. The problem is that the first argument to `map2` has a wrong type. While defining `map2`, the parameter `f` is expected to be an uncurried function. However, at the call site, `map2` is passed in a curried function. We observe that Seminal generates a useful message for fixing the type error.

For the `palin` example introduced earlier, Seminal suggests the corrective

```
File "Palin.ml", line 8, characters 21-27:
This expression has type 'a list list but is
here used with type 'a list
Relevant code: rev xs
------------------------------------------------
File "Palin.ml", line 8, characters 15-17:
Try replacing
    xs == (rev xs)
with
    ( == ) (xs, (rev xs))
of type
    'b list * 'b list list -> bool
within context
    let palin xs = ( == ) (xs, (rev xs)) ;;
```

Figure 3.10: Seminal's error message for `palin`.

change shown in Figure 3.10. Unfortunately, the suggested error location is not correct (according to [Stuckey et al., 2003]), and although the suggested change will eliminate the type error, it changes the wrong code (the suggested change of partially applying == to the pair of differently typed lists turns `palin`'s type into `'a list -> 'a list * 'a list list -> Bool`.)

Compared to other approaches, interactive approaches give users a better understanding about the type error or why certain types have been inferred for certain expressions. Consequently, several approaches to interactive type debugging have been pursued.

The ability to infer types for unbound variables enable a type debugging paradigm that is based on the idea of replacing a suspicious program snippet by a fresh variable [Bernstein and Stark, 1995]. If such replacement leads to a type correct program, then the error location has been identified. However, the original

system proposed by Berstein and Stark requires users to do these steps manually. Later, Braßel [2004] automated this process by systematically commenting out parts of the program and running the type checker iteratively. Since type changing is based on unification, it can again introduce the bias problem. Also, it is unclear how to handle programs that contain more than one type error.

Through employing a number of different techniques, Chitil [2001], Neubauer and Thiemann [2003], and Stuckey [2003; 2006] have developed tools that allow users to explore a program and inspect the types for any subexpression. Chameleon [Stuckey et al., 2003; Wazny, 2006] also allows users to query how the types for specific expressions are inferred. All these approaches provide a mechanism for users to explore a program and view the type information. However, none of them provides direct support for finding or fixing type errors.

Chapter 4: Variational Type Checking

In this chapter I present a type system for VLC. I introduce a representation of variational types in Section 4.1 and give typing rules for relating variational types to VLC expressions in Section 4.2. The type system is based on the definition of an equivalence relation on variational types defined in Section 4.3. In order to test for type equivalence, we have to identify a representative instance from each equivalence class. This is achieved through a set of terminating and confluent rewrite rules. This technical aspect is provided in Section 4.4. Finally, in Section 4.5 I present one of the main results, which says that the typing of expressions is preserved over selection.

4.1 Variational Types

As the example in Section 2.2.1 demonstrates, describing the type of variational programs requires a similar notion of variational types. The representation of variational types for VLC is given in Figure 4.1. The meanings of constant types, type variables, and function types are similar to other type systems. We extend the notion of plainness to types, defining that *plain types* contain only these three constructs. Non-plain types contain *choice types* to represent variation in types, just as choices represent variation in expressions. Choice types often (but do

$$
\begin{array}{lcll}
\phi & ::= & \gamma & \textit{Constant Type} \\
 & | & a & \textit{Type Variable} \\
 & | & \phi \to \phi & \textit{Function Type} \\
 & | & D\langle \phi, \phi \rangle & \textit{Choice Type} \\
 & & & \\
\sigma & ::= & \phi & \textit{Variational Type} \\
 & | & \forall \overline{a}.\phi & \textit{Type schema}
\end{array}
$$

Figure 4.1: VLC types.

$$
\begin{array}{lcl}
\Gamma & ::= & \varnothing \mid x \mapsto \sigma, \Gamma \\
\theta & ::= & \varnothing \mid a \mapsto \phi, \theta
\end{array}
$$

Figure 4.2: VLC type environments and substitutions.

not always) correspond directly to choice expressions; for example, the expression $D\langle 2, \texttt{True} \rangle$ has the corresponding choice type $D\langle \texttt{Int}, \texttt{Bool} \rangle$.

The function $FV(\sigma)$ for collecting free type variables is the same as the one defined in Section 2.1.1 plus the following rule for dealing with choice types.

$$FV(D\langle \phi_1, \phi_2 \rangle) = FV(\phi_1) \cup FV(\phi_2)$$

We define type environments and type substitutions in Figure 4.2. The application of a type substitution to a type schema is the same as the one defined in Section 2.1.1 plus the following rule for dealing with choice types.

$$\theta(D\langle \phi_1, \phi_2 \rangle) = D\langle \theta(\phi_1), \theta(\phi_2) \rangle$$

Note that we do not consider variational polymorphic types. This is not a problem since we can always lift quantifiers out of choices. For instance, $D\langle \forall a.\phi_1, \forall \beta.\phi_2 \rangle$ can be transformed to $\forall a_1 \beta_1.D\langle \phi_1', \phi_2' \rangle$ with $a_1 \notin FV(\phi_1)$ and

$$\boxed{\Gamma \vdash^V e : \phi}$$

$$\text{VC-Con}\ \frac{\nu \text{ is a constant of type } \gamma}{\Gamma \vdash^V \nu : \gamma} \qquad \text{VC-Abs}\ \frac{\Gamma, x \mapsto \phi_1 \vdash^V e : \phi}{\Gamma \vdash^V \lambda x.e : \phi_1 \to \phi}$$

$$\text{VC-Var}\ \frac{\Gamma(x) = \forall \overline{a}.\phi_1 \qquad \phi = \{\overline{a \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash^V x : \phi} \qquad \text{VC-App}\ \frac{\Gamma \vdash^V e_1 : \phi_1 \qquad \Gamma \vdash^V e_2 : \phi_2 \qquad \phi_1 \equiv \phi_2 \to \phi}{\Gamma \vdash^V e_1\ e_2 : \phi}$$

$$\text{VC-Let}\ \frac{\Gamma, x \mapsto \phi \vdash^V e : \phi \qquad \overline{a} = FV(\phi) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall \overline{a}.\phi \vdash^V e' : \phi'}{\Gamma \vdash^V \textbf{let}\ x = e\ \textbf{in}\ e' : \phi'}$$

$$\text{VC-Chc}\ \frac{\Gamma \vdash^V e_1 : \phi_1 \qquad \Gamma \vdash^V e_2 : \phi_2}{\Gamma \vdash^V D\langle e_1, e_2 \rangle : D\langle \phi_1, \phi_2 \rangle}$$

Figure 4.3: VLC typing rules.

$\beta_1 \notin FV(\phi_2)$, and $\phi'_1 = \{a \mapsto a_1\}(\phi_1)$ and $\phi'_2 = \{\beta \mapsto \beta_1\}(\phi_2)$.

In the next section we define the mapping from VLC expressions to variational types.

4.2 Typing Rules

The association of types with expressions is determined by a set of typing rules, given in Figure 4.3.

The VC-Con rule is a trivial rule for mapping constant expressions to type constants. The VC-Abs, VC-Var, and VC-Let typing rules for typing abstractions,

variable references, **let** expressions, respectively, use the type environment Γ and are the same as in Figure 2.1 for HM.

The VC-APP rule, however, differs from the standard definition. Given an application $e_1\ e_2$, it is typically required that if $e_2 : \phi_2$ then $e_1 : \phi_2 \to \phi$. But this is too rigid in the presence of variation since there are many cases where e_1 or e_2 are not exactly equal but are still compatible due to the presence of choice types. Instead we require that the type of e_1 be *equivalent* to a function of the appropriate type, using the type equivalence relation $\equiv$. The definition of type equivalence, and a more concrete motivation for the more permissive VC-APP rule will be provided in Section 4.3.

The VC-CHC rule states that the type of a choice is a choice type in the same dimension, where each alternative in the choice type is the type of the corresponding alternative in the choice expression.

4.3 Type Equivalence

In this section we return to the discussion of the VC-APP rule from Section 4.2. This rule is similar to the standard rule for typing application in lambda calculus, except that requiring type equality between the type of the argument and the domain type of the function is too strict. We demonstrate this with the following example.

$\texttt{succ}\ A\langle \texttt{1}, \texttt{2} \rangle$

The LHS of the application, `succ`, has type `Int` → `Int`; the RHS, $A\langle 1,2\rangle$, has type $A\langle \texttt{Int},\texttt{Int}\rangle$. Since $\texttt{Int} \neq A\langle \texttt{Int},\texttt{Int}\rangle$, the VC-APP typing rule will fail under a type-equality definition of the ≡ relation. This suggests that equality is too strict a requirement since all of the individual variants generated by the above expression (`succ 1` and `succ 2`) are perfectly well typed (both have type `Int`).

Although the types `Int` and $A\langle \texttt{Int},\texttt{Int}\rangle$ are not equal, they are still in some sense compatible, and are in fact compatible with an infinite number of other types as well. In this section we formalize this notion by defining the ≡ type equivalence relation used to determine when function application is well typed. The example above can be transformed into a more general rule that states that any choice type $D\langle\phi_1,\phi_2\rangle$ is equivalent to type ϕ if both alternative types ϕ_1 and ϕ_2 are also equivalent to ϕ. This relationship is captured formally by the choice idempotency rule, C-IDEMP, one of several type equivalence rules given in Figure 4.4.

Besides idempotency, there are many other type equivalence rules concerning choice types. The F-C rule states that we can factor/distribute function types and choice types. The C-C-SWAP rules state that types that differ only in the nesting of their choice types are equivalent. The C-C-MERGE rules reveal the property that outer choices *dominate* inner choices. For example, $D\langle D\langle 1,2\rangle,3\rangle$ is semantically equivalent to $D\langle 1,3\rangle$ since the selection of the first alternative in the outer choice implies the selection of the first alternative in the inner choice.

The remaining rules are very straightforward. The FUN rule propagates equivalence across function types, defining that two function types are equivalent if their argument and result types are equivalent. Similarly, the CHOICE equiva-

$$\textsc{Fun}\ \frac{\phi_l' \equiv \phi_r' \qquad \phi_l \equiv \phi_r}{\phi_l' \to \phi_l \equiv \phi_r' \to \phi_r} \qquad \text{F-C}\ D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_1', \phi_2' \rangle \equiv D\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle$$

$$\text{C-C-Swap1}\ D'\langle D\langle \phi_1, \phi_2 \rangle, \phi_3 \rangle \equiv D\langle D'\langle \phi_1, \phi_3 \rangle, D'\langle \phi_2, \phi_3 \rangle \rangle$$

$$\text{C-C-Swap2}\ D'\langle \phi_1, D\langle \phi_2, \phi_3 \rangle \rangle \equiv D\langle D'\langle \phi_1, \phi_2 \rangle, D'\langle \phi_1, \phi_3 \rangle \rangle \qquad \text{C-C-Merge1}\ D\langle D\langle \phi_1, \phi_2 \rangle, \phi_3 \rangle \equiv D\langle \phi_1, \phi_3 \rangle$$

$$\text{C-C-Merge2}\ D\langle \phi_1, D\langle \phi_2, \phi_3 \rangle \rangle \equiv D\langle \phi_1, \phi_3 \rangle \qquad \textsc{Choice}\ \frac{\phi_1 \equiv \phi_1' \qquad \phi_2 \equiv \phi_2'}{D\langle \phi_1, \phi_2 \rangle \equiv D\langle \phi_1', \phi_2' \rangle}$$

$$\text{C-Idemp}\ \frac{\phi_1 \equiv \phi \qquad \phi_2 \equiv \phi}{D\langle \phi_1, \phi_2 \rangle \equiv \phi} \qquad \textsc{Refl}\ \phi \equiv \phi \qquad \textsc{Symm}\ \frac{\phi \equiv \phi'}{\phi' \equiv \phi} \qquad \textsc{Trans}\ \frac{\phi \equiv \phi' \qquad \phi' \equiv \phi''}{\phi \equiv \phi''}$$

Figure 4.4: Variational type equivalence.

lence rule propagates equivalence across choice types, defining that two choice types are equivalent if both of their alternatives are equivalent. The Refl, Symm, and Trans rules make type equivalence reflexive, symmetric, and transitive, respectively.

The important property of equivalent types is that they represent the same mapping from *super-complete* decisions to plain types. A super-complete decision on types ϕ_1 and ϕ_2 is a decision that is complete for both ϕ_1 and ϕ_2; that is, it resolves both potentially variational types into plain types. Making a selection on types is the same as making a selection on expressions. As with expressions, we define selection in irrelevant dimensions to be idempotent, for example, $\lfloor \texttt{Int} \rfloor_s =$

`Int`. This is crucial since super-complete decisions will often result in selection in dimensions that do not exist in one type or the other. The semantics function $[\![\cdot]\!]$ for types is defined in the same way as for expressions, as a mapping from all complete decisions to the plain types they produce through the process of selection.

The following lemma states that a super-complete decision on two equivalent types produces the same plain type.

Lemma 1 (Type equivalence property) $\phi_1 \equiv \phi_2 \implies \forall \delta \in S : (\delta, \phi_1') \in [\![\phi_1]\!] \wedge (\delta, \phi_2') \in [\![\phi_2]\!] \implies \phi_1' = \phi_2'$, *where* S *is the set of all super-complete decisions on* ϕ_1 *and* ϕ_2.

The proof of Lemma 1 relies on a simpler lemma that states that type equivalence is preserved over selection.

Lemma 2 (Type equivalence preservation) *If* $\phi_1 \equiv \phi_2$, *then* $\lfloor \phi_1 \rfloor_s \equiv \lfloor \phi_2 \rfloor_s$.

A proof sketch for this lemma is given in Appendix A. Using Lemma 2, we can now prove Lemma 1.

PROOF of Lemma 1. By induction over the size of δ, from Lemma 2 it follows that $\phi_1 \equiv \phi_2$ implies $\lfloor \phi_1 \rfloor_\delta \equiv \lfloor \phi_2 \rfloor_\delta$. Since δ is complete for both types, neither $\lfloor \phi_1 \rfloor_\delta$ nor $\lfloor \phi_2 \rfloor_\delta$ will have any choice types. By examining the four equivalence rules that do not include choice types (FUN, REFL, SYMM, TRANS), it is clear that these types must be structurally identical. □

$$\text{S-F-Arg}\quad \frac{\phi_l \rightsquigarrow \phi_l'}{\phi_l \to \phi_r \rightsquigarrow \phi_l' \to \phi_r} \qquad \text{S-F-Res}\quad \frac{\phi_r \rightsquigarrow \phi_r'}{\phi_l \to \phi_r \rightsquigarrow \phi_l \to \phi_r'}$$

$$\text{S-F-C-Arg}\quad D\langle \phi_1, \phi_2 \rangle \to \phi \rightsquigarrow D\langle \phi_1 \to \phi, \phi_2 \to \phi \rangle \qquad \text{S-F-C-Res}\quad \phi \to D\langle \phi_1, \phi_2 \rangle \rightsquigarrow D\langle \phi \to \phi_1, \phi \to \phi_2 \rangle$$

$$\text{S-C-Swap1}\quad \frac{D \preceq D'}{D'\langle D\langle \phi_1, \phi_2 \rangle, \phi_3 \rangle \rightsquigarrow D\langle D'\langle \phi_1, \phi_3 \rangle, D'\langle \phi_2, \phi_3 \rangle \rangle}$$

$$\text{S-C-Swap2}\quad \frac{D \preceq D'}{D'\langle \phi_1, D\langle \phi_2, \phi_3 \rangle \rangle \rightsquigarrow D\langle D'\langle \phi_1, \phi_2 \rangle, D'\langle \phi_1, \phi_3 \rangle \rangle}$$

$$\text{S-C-Dom1}\quad \frac{\lfloor \phi_1 \rfloor_D = \phi_1' \qquad \phi_1 \neq \phi_1'}{D\langle \phi_1, \phi_2 \rangle \rightsquigarrow D\langle \phi_1', \phi_2 \rangle} \qquad \text{S-C-Dom2}\quad \frac{\lfloor \phi_2 \rfloor_{\tilde{D}} = \phi_2' \qquad \phi_2 \neq \phi_2'}{D\langle \phi_1, \phi_2 \rangle \rightsquigarrow D\langle \phi_1, \phi_2' \rangle}$$

$$\text{S-C-Alt1}\quad \frac{\phi_1 \rightsquigarrow \phi_1'}{D\langle \phi_1, \phi_2 \rangle \rightsquigarrow D\langle \phi_1', \phi_2 \rangle} \qquad \text{S-C-Alt2}\quad \frac{\phi_2 \rightsquigarrow \phi_2'}{D\langle \phi_1, \phi_2 \rangle \rightsquigarrow D\langle \phi_1, \phi_2' \rangle} \qquad \text{S-C-Idemp}\quad D\langle \phi, \phi \rangle \rightsquigarrow \phi$$

Figure 4.5: Variational type simplification.

4.4 Type Simplification

Each equivalence class contains an infinite number of types. For example, we can always trivially expand a type ϕ into an equivalent choice type $D\langle \phi, \phi \rangle$. In order to facilitate checking whether two types are equivalent, we identify one representative from each equivalence class as a normal form and define rewriting rules to achieve this normal form. Two types are then equivalent if they have the same normal form.

We define the *type simplification* relation $\rightsquigarrow$ to rewrite a type into a simpler one and use the reflexive, transitive closure of this relation $\rightsquigarrow^*$ to transform a type into normal form. The type simplification rules, shown in Figure 4.5, are derived from the type equivalence rules in Figure 4.4.

The FUN equivalence rule leads to two rewriting rules, S-F-ARG and S-F-RES, which simplify the domain and result of a function type, respectively. Likewise, the S-F-C-ARG and S-F-C-RES rules are derived from the F-C equivalence rule, and distribute a choice type over the domain or result of a function type, respectively.

The two S-C-SWAP simplification rules are derived from the two C-C-SWAP equivalence rules, but each adds an additional premise that ensures choice types will be nested according to a known ordering relation $\preceq$ on dimension names. Thus, if ϕ is in normal form and $A \preceq B$, then no choice type $A\langle\ldots\rangle$ will appear within an alternative of a choice type $B\langle\ldots\rangle$ in ϕ.

Picking a good ordering relation is important for efficiency since the S-C-SWAP rules each duplicate part of the type. In the worst case, the normalization process could lead to an exponential blow up in the size of the type. For example, if we assume a lexicographic ordering for $\preceq$, then normalizing the type $D\langle C\langle B\langle A\langle\cdots\rangle,\cdots\rangle,\cdots\rangle,\phi\rangle$, will lead to ϕ being duplicated eight times. One way to deal with this problem is to determine $\preceq$ heuristically, for example, by initially performing a depth-first traversal of one of the types we are comparing (before converting it into normal form), then using its nesting of choices as $\preceq$. This will ensure that we only perform the necessary choice-swaps in the other type. Another approach (which can be combined with the heuristic approach) is to share, rather

than duplicate, the common part when swapping choice types. This requires solving the common subexpression problem, for which there are some efficient algorithms [Downey et al., 1980; Nelson and Oppen, 1980]. Since performance is not our primary focus here, our unification algorithm uses a variant of the first approach, only swapping choices as needed during the decomposition process.

The remaining rewriting rules are straightforward. Like the S-F-ARG and S-F-RES rules, the S-C-ALT rules are focused adaptations of the CHOICE equivalence rule, one simplifies the left alternative of a choice type, the other simplifies the right. The S-C-DOM rules follow less directly from the corresponding C-C-MERGE equivalence rules. They reuse the selection operation from the semantics to more immediately eliminate any dominated choice types. Finally, the S-C-IDEMP rewriting rule is derived from the C-IDEMP equivalence rule, but is somewhat stricter since it requires the two alternatives to be structurally equal.

By repeatedly applying type simplification until no rewrite rule matches, we achieve a type in normal form. Types in normal form satisfy the following criteria:

1. Choice types are maximally lifted over function types. For example, the type $A\langle \texttt{Int} \to a, a \to \texttt{Bool} \rangle$ is in normal form, while $A\langle \texttt{Int}, a \rangle \to A\langle a, \texttt{Bool} \rangle$ is not.

2. The type does not contain dominated choices. For example, the type $A\langle A\langle \texttt{Int}, \texttt{Bool} \rangle, a \rangle$ is not in normal form. It can be simplified to $A\langle \texttt{Int}, a \rangle$, which is in normal form.

3. The nesting of choices adheres to the ordering relation $\preceq$ on their dimension names.

$$
\begin{array}{ll}
B\langle A\langle \phi_1, \phi_2\rangle, A\langle \phi_1, \phi_2\rangle\rangle & \\
\rightsquigarrow A\langle B\langle \phi_1, A\langle \phi_1, \phi_2\rangle\rangle, B\langle \phi_2, A\langle \phi_1, \phi_2\rangle\rangle\rangle & \text{(S-C-SWAP1)} \\
\rightsquigarrow A\langle B\langle \phi_1, \phi_1\rangle, B\langle \phi_2, \phi_2\rangle\rangle & \text{(S-C-DOM)} \\
\rightsquigarrow A\langle \phi_1, B\langle \phi_2, \phi_2\rangle\rangle & \text{(S-C-IDEMP/S-C-ALT1)} \\
\rightsquigarrow A\langle \phi_1, \phi_2\rangle & \text{(S-C-IDEMP/S-C-ALT2)}
\end{array}
$$

Figure 4.6: Example transformation into normal form.

4. The type contains no choice types with equivalent alternatives.

5. Finally, a function type is in normal form if both its domain and result types are in normal form; a choice type is in normal form if all its alternatives are in normal form.

Figure 4.6 shows an example transformation of the type $B\langle A\langle \phi_1, \phi_2\rangle, A\langle \phi_1, \phi_2\rangle\rangle$ into its corresponding normal form $A\langle \phi_1, \phi_2\rangle$ (we assume $A \preceq B$). Note that in the application of the S-C-SWAP1 rule, we arbitrarily chose to swap the nested choice in the first alternative. We could have also applied S-C-SWAP2, or applied S-C-IDEMP to the alternatives of the A choice type. An important property of the $\rightsquigarrow^*$ relation, however, is that our decisions at these points do not matter. No matter which rule we apply, we will still achieve the same normal form. This is the property of *confluence*, expressed in Theorem 1 below.

Theorem 1 (Confluence) *If $\phi \rightsquigarrow^* \phi_1$ and $\phi \rightsquigarrow^* \phi_2$, then there exists a ϕ' such that $\phi_1 \rightsquigarrow^* \phi'$ and $\phi_2 \rightsquigarrow^* \phi'$.*

A rewriting relation is confluent if it is both *locally confluent* and *terminating*. These properties are expressed for the $\rightsquigarrow^*$ relation below, in Lemmas 3 and 4.

From these two lemmas, Thorem 1 follows directly.

Lemma 3 (Local confluence) *For any type ϕ, if $\phi \rightsquigarrow \phi_1$ and $\phi \rightsquigarrow \phi_2$, then there exists some type ϕ' such that $\phi_1 \rightsquigarrow^* \phi'$ and $\phi_2 \rightsquigarrow^* \phi'$.*

The proof of this lemma is given in Appendix A.

Lemma 4 (Termination) *Given any type ϕ, $\phi \rightsquigarrow^* \phi'$ is terminating.*

We give an informal proof of termination here to convince the reader. A formal proof based on a counting mechanism is presented in Appendix A.

PROOF SKETCH. The $\rightsquigarrow^*$ relation will terminate when we reach a normal form (as defined by the criteria listed above) because an expression satisfying these criteria will not match any rule in the $\rightsquigarrow$ relation, by construction. Therefore, we must show that these criteria will be satisfied in a finite number of steps.

Trivially, the two S-C-DOM rules eliminate dominated choices, and the S-C-IDEMP rule eliminates equivalent alternatives, in a finite number of steps. The S-F-C-RES and S-F-C-ARG lift choice types over function types, and no rule can lift function types back out. The S-C-SWAP1 and S-C-SWAP2 rules define a similar one-way relation for choice nestings, according to the $\preceq$ relation on choice names. Thus, we can see that all rules make progress toward satisfying one of the criteria, and that, in isolation they can achieve this in a finite number of steps.

A potential challenge to termination arises via the duplication of type subexpressions in the S-F-C and S-C-SWAP rules. For example, the right alternative ϕ_3 of the original choice type is duplicated in the application of the S-C-SWAP1 rule.

However, observe that these can only create a finite amount of additional work since the rules otherwise make progress as described above. □

A terminating rewriting relation is by definition *normalizing*. Since rewriting is both confluent and normalizing, any variational type can be transformed into a unique normal form [Baader and Nipkow, 1998, p. 12]. We write $norm(\phi)$ for the unique normal form of ϕ. We capture the fact that a normal form represents an equivalence class by stating in the following theorem that two types are equivalent if and only if they have the same normal form.

Theorem 2 $\phi \equiv \phi' \Leftrightarrow norm(\phi) = norm(\phi')$.

PROOF. This follows from Theorem 1, the fact that $\leadsto^*$ is normalizing, and the observation that the $\equiv$ relation is the symmetric, reflexive, and transitive closure of $\leadsto$. □

This is the essential result needed for checking type equivalence.

4.5 Type Preservation

An important property of the type system is that any plain expression that can be selected from a well-typed variational expression is itself well typed, and that the plain type of the variant can be obtained by the same selection on the variational type. This result can be proved with the help of the following lemma, which states that variational typing is preserved over a single selection.

Lemma 5 $\Gamma \vdash^V e : \phi \implies e$ *is plain or* $\forall s : \Gamma \vdash^V \lfloor e \rfloor_s : \lfloor \phi \rfloor_s$.

PROOF. The proof is based on induction over the typing rules. We show only the cases for the VC-APP rule and the VC-CHC rule. The cases for the other rules can be constructed similarly. Also, we write the typing judgment $\Gamma \vdash^V e : \phi$ more succinctly as $e : \phi$ when the environments are not significant.

We consider the VC-APP rule first. Assume that $e\ e' : \phi$, then we must show that $\lfloor e\ e' \rfloor_s : \lfloor \phi \rfloor_s$. We do this through the following sequence of observations.

1.	$e : \phi''$, $e' : \phi'$, and $\phi'' \equiv \phi' \to \phi$	by the definition of VC-APP
2.	$\lfloor e\ e' \rfloor_s = \lfloor e \rfloor_s\ \lfloor e' \rfloor_s$	by the definition of $\lfloor \cdot \rfloor_s$
3.	$\lfloor e \rfloor_s : \lfloor \phi'' \rfloor_s$ and $\lfloor e' \rfloor_s : \lfloor \phi' \rfloor_s$	by the induction hypothesis
4.	$\lfloor \phi'' \rfloor_s \equiv \lfloor \phi' \to \phi \rfloor_s$	by 1 and Lemma 2
5.	$\lfloor \phi'' \rfloor_s \equiv \lfloor \phi' \rfloor_s \to \lfloor \phi \rfloor_s$	by 4 and the definition of $\lfloor \cdot \rfloor_s$
6.	Therefore, $\lfloor e\ e' \rfloor_s : \lfloor \phi \rfloor_s$	by 2, 3, 5, and the definition of VC-APP

For the VC-CHC rule, assume that $D\langle e_1, e_2 \rangle : D\langle \phi_1, \phi_2 \rangle$. Then we must show that $\lfloor D\langle e_1, e_2 \rangle \rfloor_s : \lfloor D\langle \phi_1, \phi_2 \rangle \rfloor_s$. There are two cases to consider: either s represents a selection in choice D, or it does not.

If s represents a selection in choice D, the proof follows directly from the induction hypothesis and the definitions of selection on expressions and types. For example, if s selects the first alternative in D, then selecting the first alternative on both sides of the typing relation leaves us with $\lfloor e_1 \rfloor_s : \lfloor \phi_1 \rfloor_s$, which is the induction hypothesis.

If s does not represent a selection in D, then applying selection to each side of the typing relation yields $D\langle \lfloor e_1 \rfloor_s, \lfloor e_2 \rfloor_s \rangle : D\langle \lfloor \phi_1 \rfloor_s, \lfloor \phi_2 \rfloor_s \rangle$. Since $\lfloor e_1 \rfloor_s : \lfloor \phi_1 \rfloor_s$ and

$\lfloor e_2 \rfloor_s : \lfloor \phi_2 \rfloor_s$ by the induction hypothesis, the claim follows through a direct application of the VC-CHC rule. □

By induction it follows that for any set of selectors δ that yields a plain expression from e, δ also selects the corresponding plain type from ϕ. Therefore, the following theorem, which captures the type preservation property described at the beginning of this section, follows directly from Lemma 5.

Theorem 3 (Type preservation) *If* $\Gamma \vdash^V e : \phi$ *and* $(\delta, e') \in [\![e]\!]$, *then* $\Gamma \vdash^V e' : \phi'$ *where* $(\delta, \phi') \in [\![\phi]\!]$.

With Theorem 3, the type of any particular variant of e can be easily selected from its inferred variational type ϕ. For example, suppose $\varnothing, \Gamma \vdash^V e : \phi$ with $\phi = A\langle B\langle \phi_1, \phi_2 \rangle, \phi_3 \rangle$, then the type of $e' = \lfloor \lfloor e \rfloor_A \rfloor_{\vec{B}}$ is $\lfloor \lfloor \phi \rfloor_A \rfloor_{\vec{B}} = \phi_2$.

Type preservation demonstrates that the type system is correct. We must also show that it is complete. That is, if every plain variant encoded by a variational expression e is type correct, then our type system will assign a variational type to e. The completeness property is stated in the following theorem.

Theorem 4 (Completeness) *If* $\forall (\delta, e') \in [\![e]\!]$, $\exists \phi'$ *such that* $\Gamma \vdash^V e' : \phi'$, *then* $\exists \phi$ *such that* $\Gamma \vdash^V e : \phi$.

This theorem can be proved by a simple induction over the structure of e, with the help of the following lemma.

Lemma 6 *If D is a dimension used in a choice in e, and $\exists \phi_1$ such that $\Gamma \vdash^V \lfloor e \rfloor_D : \phi_1$, and $\exists \phi_2$ such that $\Gamma \vdash^V \lfloor e \rfloor_{\tilde{D}} : \phi_2$, then $\exists \phi$ such that $\Gamma \vdash^V e : \phi$.*

PROOF. This follows from induction over the structure of e and the typing derivations of $\lfloor e \rfloor_D$ and $\lfloor e \rfloor_{\tilde{D}}$. □

4.6 Related Work

This section addresses work related to choice types, type normalization, and the implementation of variational type checking. Work related to other aspects of this dissertation is presented in Chapter 3.

Choice types are in some ways similar to variant types [Kagawa, 2006]. Variant types support the uniform manipulation of a heterogeneous collection of types. A significant difference between the two is that choices (at the expression level) contain all of the information needed for inferring their corresponding choice types. Values of variant types, on the other hand, are associated with just one label, representing one branch of the larger variant type. This makes type inference very difficult. A common solution is to use explicit type annotations; whenever a variant value is used, it must be annotated with a corresponding variant type. Typing VLC expressions does not require such annotations.

Choice types are also somewhat similar to union types [Dezani-Ciancaglini et al., 1997]. A union type, as its name suggests, is a union of simpler types. For example, a function `f` might accept as arguments the union of types `Int` and `Bool`. Function application is then well typed if the argument's type is an element of the

union type. Thus, `f` could accept arguments of type `Int` or type `Bool`. The biggest difference between union types and choice types is that union types are comparatively unstructured. In VLC, choices can be synchronized, allowing functions to provide different implementations for different argument types, or for different sets of functions to be defined in the context of different argument types. With union types, an applied function must be able to operate on all possible values of an argument with a union type. A major challenge in type inference with union types is union elimination, which is not syntax directed and makes type inference intractable. Therefore, as with variant types, syntactic markers are needed to support type inference.

Type conditions are an extension to parametric polymorphism in the presence of subtyping that have been studied in the contexts of both the Java generics system [Huang et al., 2007] and C++ templates [Dos Reis and Stroustrup, 2006]. They can be used to conditionally include data members and methods into a class only when the type parameters are instantiated with types that satisfy the given conditions (for example, that the type is a subtype of a certain class). Often this can be used to produce similar effects to the C Preprocessor, but in a way that can be statically typed. Type conditions differ from VLC in that they capture a much more specific type of variation, namely, conditional inclusion of code depending on the type of a class's type parameters; in contrast, VLC can represent arbitrary variation. Type conditions also have a quite coarse granularity, varying only top-level methods and fields. A feature, relative to VLC, is that different variants of the same code (class) can be used within the same program (by instantiating the

class's type parameters differently).

When a new type system is designed or when new features are added to an existing system, a new unification algorithm and type inference algorithm must be coined for the new system, and the correctness of the new system and algorithms have to be demonstrated. As evidenced by this dissertation, this is quite a lot of work. To reduce this burden and promote reuse, Odersky and Sulzmann [1999; 2001; 2008] have proposed HM(X), a general framework for type systems with constraints, including a type inference algorithm that computes principal types that satisfy these constraints. By instantiating X to different extensions, different type systems can be generated from HM(X). For example, X can be instantiated to polymorphic records, equational theories, and subtypes. Variational type inference cannot be implemented within HM(X), however, and we cannot therefore reuse its algorithms and proofs. This is because HM(X) requires constraints to satisfy a regularity property that does not hold in variational type inference. The regularity property states that two sides of any equational theory must have the same free variables, but this is not true in VLC's type system because of choice domination. For example, $A\langle A\langle a,b\rangle,c\rangle \equiv A\langle a,c\rangle$ but $\{a,b,c\} \neq \{a,c\}$.

Some aspects of the type system presented in this chapter can be simulated by dependent types [Xi and Pfenning, 1999]. However, there are limitations of this approach. For one, type inference with dependent types is undecidable. Most dependent type systems also require programmers to supply complex type annotations or construct proof terms to support type checking [Paulin-Mohring, 1993; Xi and Pfenning, 1999; Fogarty et al., 2007; Norell, 2007]. This is a significant

burden that our type system does not impose. A more restricted version of choice types could also be implemented with phantom types and GADTs [Johann and Ghani, 2008]. However, GADTs cannot express arbitrary choice types since the type of each alternative would be constrained by the requirement that the result type of each branch of a GADT must refine the data type being defined.

Related to our process of variation type normalization, [Balat et al., 2004] present a powerful normalizer for terms in lambda calculus with sums. They make use of a similar transformation for eliminating dead alternatives. Our type normalizer differs from theirs in two technical details. First, choices in VLC are named, and choices with different names are treated differently. Their normalizer makes no such distinction among sums, making it essentially equivalent to VLC in which all choices are in the same dimension. Second, the order of choice nesting is significant in our normalization, whereas the order of sum nesting is not in theirs.

Chapter 5: Variational Unification

A type inference algorithm relies heavily on its underlying unification algorithm, for example, algorithm $\mathscr{W}$ relies on Robinson's unification algorithm $\mathscr{U}_R$. For variational type inference, we need to solve unification problems of variational types that respect the semantics of choice types and allow a less strict typing for function applications. The equational theory is defined by the type equivalence relation in Figure 4.4. We call this unification problem *choice type* (CT). The properties of the CT-unification problem are described in Section 5.1, while the unification algorithm that solves it is presented in Section 5.2. In Section 5.3 we formally evaluate the correctness of the unification algorithm, and we analyze its time complexity in Section 5.4.

5.1 The Choice Type Unification Problem

If we view a choice as a binary operator on its two subexpressions, then CT's equational theory contains both distributivity (introduced by the C-C-SWAP rule) and associativity (which follows from the C-C-MERGE rules). Usually, this yields a unification problem that is undecidable [Anantharaman et al., 2004]. CT-unification, however, *is* decidable. The key insight is that a normalized choice type cannot contain nested choice types in the same dimension, effectively bounding the number

of choice types a variational type can contain.

To get a sense for CT-unification, consider the following unification problem.

$$A\langle \texttt{Int}, a\rangle \equiv^{?} B\langle b, c\rangle \tag{5.1}$$

Several potential unifiers for this problem are given below. In each mapping, type variables other than a, b, and c are assumed to be fresh.

$$\xi_1 = \{a \mapsto \texttt{Int},\ b \mapsto \texttt{Int},\ c \mapsto \texttt{Int}\}$$
$$\xi_2 = \{b \mapsto A\langle \texttt{Int}, a\rangle,\ c \mapsto A\langle \texttt{Int}, a\rangle\}$$
$$\xi_3 = \{a \mapsto B\langle \texttt{Int}, d_2\rangle,\ b \mapsto \texttt{Int},\ c \mapsto A\langle \texttt{Int}, d_2\rangle\}$$
$$\xi_4 = \{a \mapsto B\langle d_2, \texttt{Int}\rangle,\ b \mapsto A\langle \texttt{Int}, d_2\rangle,\ c \mapsto \texttt{Int}\}$$
$$\xi_5 = \{a \mapsto B\langle d_1, d_2\rangle,\ b \mapsto A\langle \texttt{Int}, d_1\rangle,\ c \mapsto A\langle \texttt{Int}, d_2\rangle\}$$
$$\xi_6 = \{a \mapsto B\langle A\langle d_5, d_1\rangle, A\langle d_6, d_2\rangle\rangle,\ b \mapsto B\langle A\langle \texttt{Int}, d_1\rangle, d_3\rangle, c \mapsto B\langle d_4, A\langle \texttt{Int}, d_2\rangle\rangle\}$$

These mappings are unifiers since, after applying any one of these mappings to the types in problem (5.1), the types of the LHS and RHS of the problem are equivalent. We observe that ξ_6 is the most general of these unifiers. In fact, it is the most general unifier (mgu) for this CT-unification problem. This means that by assigning appropriate types to the type variables in ξ_6, we can produce any other unifier. For example, composing ξ_6 with

$$\{d_5 \mapsto d_1, d_6 \mapsto d_2, d_3 \mapsto A\langle \texttt{Int}, d_1\rangle, d_4 \mapsto A\langle \texttt{Int}, d_2\rangle\}$$

yields ξ_5, which is in turn the most general among the first five unifiers.

Although ξ_6 is more general than ξ_5, if we apply either one to the types in problem (5.1), then simplify dominated choices, we will get the same result. Therefore, it may seem that the generality provided by ξ_6 is superficial. But in fact, ξ_6 solves strictly more unification problems than ξ_5. For instance, assume

$e_1 : A\langle \texttt{Int}, a\rangle \to c \to c$, $e_2 : B\langle b, c\rangle$, and $e_3 : B\langle \texttt{Bool}, \texttt{Int}\rangle$. Using ξ_5 the expression $e_1\ e_2$ has type $A\langle \texttt{Int}, d_2\rangle \to A\langle \texttt{Int}, d_2\rangle$, so the expression $e_1\ e_2\ e_3$ will be ill typed since $A\langle \texttt{Int}, d_2\rangle$ does not unify with $B\langle \texttt{Bool}, \texttt{Int}\rangle$. On the other hand, if we use ξ_6, then $e_1\ e_2$ has type $B\langle d_4, A\langle \texttt{Int}, d_2\rangle\rangle \to B\langle d_4, A\langle \texttt{Int}, d_2\rangle\rangle$, so the expression $e_1\ e_2\ e_3$ is type correct since the unification problem $B\langle d_4, A\langle \texttt{Int}, d_2\rangle\rangle \equiv^? B\langle \texttt{Bool}, \texttt{Int}\rangle$ has the mgu $\{d_2 \mapsto B\langle l, A\langle m, \texttt{Int}\rangle\rangle, d_4 \mapsto B\langle \texttt{Bool}, k\rangle\}$, where k, l and m are fresh type variables.

An equational unification problem is said to be *unitary* if there is a unique unifier that is more general than all other unifiers [Baader and Snyder, 2001]. This is important to make type inference feasible since we need only maintain the unique mgu throughout the inference process.

It is not immediately obvious that CT-unification is unitary. Usually, equational unification problems with associativity and distributivity are not unitary. However, the same bounds that make CT-unification decidable (that is, the normalization process that ensures, via the S-C-DOM rules that there are no nested choices in the same dimension) also make the problem unitary. Specifically, choice dominance ensures that a CT-unification problem can be decomposed into a finite number of simpler unification problems that are known to be unitary. Furthermore, the mgus of these subproblems can be used to construct the unique mgu of the original CT-unification problem.

That the CT-unification problem is unitary is captured in the following theorem.

Theorem 5 *Given a CT-unification problem U, there is a unifier ξ such that for*

any unifier ξ', there exists a mapping θ such that $\xi' = \theta \circ \xi$.

The proof of this theorem relies on definitions from the rest of this section and so is delayed until Appendix B. We give a high-level description of the argument here.

A CT-unification problem encodes a finite number of plain subproblems, where a plain unification problem is between two plain types. For example, problem (5.1) encodes the plain subproblems $\texttt{Int} \equiv^? b$, $\texttt{Int} \equiv^? c$, $a \equiv^? b$, and $a \equiv^? c$ since A and B are independent dimensions that can be selected from independently. In principle, solving a variational unification problem requires solving all of the plain unification problems it encodes. One challenge of CT-unification is that different plain subproblems may share the same type variables, and these may be incorrectly mapped to different types when solving the different subproblems. However, CT-unification problems whose plain subproblems share no common type variables are easy to solve. We just generate all of the plain subproblems, solve each of them using the traditional Robinson unification algorithm [Robinson, 1965], then take the union of the resulting set of unifiers as the solution to the original problem.

The basic structure of the argument that CT-unification is unitary is therefore to demonstrate that:

1. We can transform any CT-unification problem U into an equivalent unification problem U', such that the plain subproblems of U' share no type variables. This can be done through the process of type variable qualification, described

in Section 5.2.

2. These subproblems are plain and therefore themselves unitary.

3. We can construct a unique mgu for U from the mgus of the individual subproblems of U'. This is achieved through the process of completion, also described in Section 5.2.

Of course, we do not actually solve CT-unification problems by solving all of the corresponding plain subproblems separately since it would be very inefficient. However, type variable qualification and completion do play a role in the actual algorithm, which is developed and presented in the next subsection.

5.2 Variational Unification with Qualifications

This section will present our approach to unifying variational types. Since there is no general algorithm or strategy for equational unification problems [Baader and Snyder, 2001], we begin by motivating our approach. Consider the following example unification problem.

$$A\langle \texttt{Int}, a\rangle \equiv^? A\langle a, \texttt{Bool}\rangle \tag{5.2}$$

We might attempt to solve this problem through simple decomposition, by unifying the corresponding alternatives of the choice types. This leads to the unification problem $\{\texttt{Int} \equiv^? a,\ a \equiv^? \texttt{Bool}\}$, which is unsatisfiable. However, notice that $\{a \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}$ is a unifier for the original problem (through choice domination), so this approach to decomposition must be incorrect.

The key insight is that there is a fundamental difference between the type variables in the types a, $A\langle a,\phi\rangle$, and $A\langle\phi,a\rangle$, even though all three are named a. A type variable in one alternative of a choice type is *partial* in the sense that it applies only to a subset of the type variants. In particular, it is independent of type variables of the same name in the other alternative of that choice type. In example (5.2), the two occurrences of a can denote two different types because they cannot be selected at the same time. The important fact that a appears in two different alternatives of the A choice type is lost in the decomposition by alternatives.

We address this problem with a notion of *qualified type variables*, where each type variable is marked by the alternatives in which it is nested. A qualified type variable a is denoted by a_q, where q is the qualification and is given by a set of selectors (see Section 4.3), rendered as a lexicographically sorted sequence. For example, the type variable a in $B\langle\phi_1,A\langle a,\phi_2\rangle\rangle$ corresponds to the qualified type variable $a_{A\tilde{B}}$. Likewise, the (non-qualified) unification problem in example (5.1) can be transformed into the *qualified unification problem* $A\langle\texttt{Int},a_{\tilde{A}}\rangle \equiv^?_q B\langle b_B,c_{\tilde{B}}\rangle$, and the problem in example (5.2) can be transformed into $A\langle\texttt{Int},a_{\tilde{A}}\rangle \equiv^?_q A\langle a_A,\texttt{Bool}\rangle$.

In addition to the traditional operations of matching and decomposition used in equational unification, our unification algorithm uses two other operations: choice type *hoisting* and type variable *splitting*. These are needed to transform the types being unified into more similar structures that can then be matched or decomposed.

Hoisting is applied when unifying two types that have top-level choice types

with different dimension names. To illustrate, consider the following unification problem.

$$A\langle B\langle \texttt{Int}, \texttt{Bool}\rangle, a_{\tilde{A}}\rangle \equiv_q^? B\langle a_B, \texttt{Bool}\rangle$$

We cannot immediately decompose this problem by alternatives since the dimensions of the top-level choice types do not match. However, this problem can be solved by applying the C-C-SWAP1 rule to the LHS, thereby hoisting the B choice type to the top.

$$B\langle A\langle \texttt{Int}, a_{\tilde{A}B}\rangle, A\langle \texttt{Bool}, a_{\tilde{A}\tilde{B}}\rangle\rangle \equiv_q^? B\langle a_B, \texttt{Bool}\rangle$$

Notice that we must add a qualification to all of the duplicated type variables that were originally in the alternative opposite the hoisted choice type, but are now nested beneath it, such as the $a_{\tilde{A}}$ variable in the example. Now we can decompose the problem by unifying the corresponding alternatives of the top-level choice type.

Splitting is the expansion of a type variable into a choice type between two qualified versions of that variable. It is used whenever decomposition cannot proceed and the problem cannot be solved by hoisting. For example, to decompose the problem $a \equiv_q^? A\langle a_A, \texttt{Int}\rangle$, we first split a into the choice type $A\langle a_A, a_{\tilde{A}}\rangle$, then decompose by alternatives. To decompose the problem $A\langle \texttt{Int}, a_{\tilde{A}}\rangle \equiv_q^? B\langle \texttt{Int}, b_{\tilde{B}}\rangle$, we can split either $a_{\tilde{A}}$ into a choice in B or $b_{\tilde{B}}$ into a choice in A. In either case, we must then apply hoisting once before the problem can be decomposed by alternatives.

Figure 5.1 presents an example in which split and hoist are used to prepare a qualified unification problem for decomposition. Note that after decomposition, we

$$
\begin{array}{r}
A\langle \texttt{Int}, \underline{a_{\tilde{A}}} \rangle \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle \\
\underline{A\langle \texttt{Int}, B\langle a_{\tilde{A}B}, a_{\tilde{A}\tilde{B}} \rangle \rangle} \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle \\
B\langle A\langle \texttt{Int}, a_{\tilde{A}B} \rangle, A\langle \texttt{Int}, a_{\tilde{A}\tilde{B}} \rangle \rangle \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle
\end{array}
\quad \begin{array}{l} \scriptstyle split \downarrow \\ \scriptstyle hoist \downarrow \end{array}
$$

$$
\begin{array}{r}
A\langle \texttt{Int}, a_{\tilde{A}B} \rangle \equiv_q^? b_B \\
A\langle \texttt{Int}, a_{\tilde{A}\tilde{B}} \rangle \equiv_q^? c_{\tilde{B}}
\end{array}
$$

Figure 5.1: Example of qualified unification.

do not need to split b_B into a choice in A because b_B is isolated and occurs on only one side of the subtask; instead we can return the substitution $\{b_B \mapsto A\langle \texttt{Int}, a_{\tilde{A}B} \rangle\}$ for this subtask directly. Likewise for $c_{\tilde{B}}$ in the second subtask.

To solve a unification problem U, we solve the corresponding qualified unification problem Q, then transform the solution of Q, σ_Q, into a solution for U, σ_U. Each mapping $a \mapsto \phi$ in σ_U is derived through a process called *completion* from the related subset of mappings in σ_Q, $\{a_{q_1} \mapsto \phi_1, \ldots, a_{q_n} \mapsto \phi_n\}$. Each qualified mapping $a_{q_i} \mapsto \phi_i$ describes a leaf in a tree of nested choice types that makes up the resulting type ϕ. Building and populating this tree is the goal of completion. For example, given the qualified mappings $\{a_A \mapsto \texttt{Int}, a_{\tilde{A}B} \mapsto b, a_{\tilde{A}\tilde{B}} \mapsto \texttt{Bool}\}$, completion yields the unqualified mapping $a \mapsto A\langle \texttt{Int}, B\langle b, \texttt{Bool} \rangle \rangle$. When the qualified mappings do not describe a complete tree, the completion process introduces fresh type variables to represent the unconstrained parts of the type. For example, given the qualified mappings $\{a_A \mapsto \texttt{Int}, a_{\tilde{A}\tilde{B}} \mapsto \texttt{Bool}\}$, completion yields the unqualified mapping $a \mapsto A\langle \texttt{Int}, B\langle c, \texttt{Bool} \rangle \rangle$, where c is a fresh type variable.

Formally, we define completion by folding the helper function *comp*, defined

$$
\begin{aligned}
comp(D\,q, \phi, D\,\langle \phi_1, \phi_2 \rangle) &= D\,\langle comp(q, \phi, \phi_1), \phi_2 \rangle \\
comp(\tilde{D}\,q, \phi, D\,\langle \phi_1, \phi_2 \rangle) &= D\,\langle \phi_1, comp(q, \phi, \phi_2) \rangle \\
comp(D\,q, \phi, \phi') &= D\,\langle comp(q, \phi, \phi'), fresh(\phi') \rangle \\
comp(\tilde{D}\,q, \phi, \phi') &= D\,\langle fresh(\phi'), comp(q, \phi, \phi') \rangle \\
comp(\epsilon, \phi, \alpha) &= \phi
\end{aligned}
$$

Figure 5.2: Helper function used in the completion process.

in Figure 5.2, across the mappings in σ_Q. This function produces a partially completed type given (1) a type variable qualification q, (2) the type to store at the path described by q, and (3) the type that is being completed. The definition of *comp* relies on top-down pattern matching on the first and third arguments (ϵ matches the empty qualification), and on a second helper function *fresh* that renames every type variable in its argument type to a new, fresh type variable.

In the first two cases of *comp*, if the partially completed type already contains a choice type in the dimension D referred to by the first selector in the qualification, the function consumes the selector and propagates the completion into the appropriate alternative. Note that these choice types will have been created by a previous invocation of *comp* on a different qualification, as we'll see below. In the third and fourth cases, the partially completed type does not already contain a choice type in D, so we create a new one and propagate the completion into the appropriate branch, freshening the type variables in the duplicated alternative. In these first four cases, we traverse and create a tree structure of choice types. This relies on the fact that selectors are sorted in the qualification q, avoiding the creation of choice types in the same dimension. However, it is possible that types

stored at the leaves of this tree will contain choice types in dimensions created by *comp*; these can be eliminated by a subsequent normalization step.

Finally, the completion of $a \mapsto \phi$ from $\{a_{q_1} \mapsto \phi_1, \ldots, a_{q_n} \mapsto \phi_n\}$ is defined as follows.

$$\phi = comp(q_1, \phi_1, comp(q_2, \phi_2, \ldots\ comp(q_n, \phi_n, b)\ \ldots))$$

The initial argument to the folded completion function is a fresh type variable b, and the order in which we process the qualifications $q_1, \ldots, q_n$ does not matter. Also note that, although *comp* is not specified for all argument patterns, the completion process cannot fail on any unifier produced by our unification algorithm. This is because we do not produce mappings for "overlapping" qualified type variables (see the discussion of *occurs* later in this section).

The final and most important piece of the variational-type-unification puzzle is the algorithm for solving qualified unification problems. The definition of this algorithm, *vunify*, is given in Figure 5.3. In this definition, we use p to range over plain types (which do not contain choice types), and g to range over *ground plain types*, which do not contain choice types or type variables. We also assume that $D_1 \neq D_2$ and use ϕ_L and ϕ_R to refer to the entire LHS and RHS of the unification problem, respectively. Cases marked with an asterisk represent two symmetric cases. That is, the definition of $vunify^*(\phi, \phi')$ implies the definition of both $vunify(\phi, \phi')$, as written, and a dual case $vunify(\phi', \phi) = vunify(\phi, \phi')$.

The definition of *vunify* will be explained in detail below. The algorithm relies on several helper functions. The function *hoist* implements a deep form

$$vunify : (\phi_L, \phi_R) \to \xi$$

$$vunify(p, p') = \mathscr{U}_R(p, p') \tag{1}$$

$$vunify^*(a_q, D\langle\phi_1, \phi_2\rangle) = vunify(D\langle a_{Dq}, a_{\tilde{D}q}\rangle, D\langle\phi_1, \phi_2\rangle) \tag{2}$$

$$vunify(D\langle\phi_1, \phi_2\rangle, D\langle\phi'_1, \phi'_2\rangle) = \xi_1 \leftarrow vunify(\phi_1, \phi'_1) \quad \xi_2 \leftarrow vunify(\xi_1(\phi_2), \xi_1(\phi'_2)) \quad \text{return } \xi_1 \circ \xi_2 \tag{3}$$

$$vunify^*(D_1\langle\phi_1, \phi_2\rangle, D_2\langle\phi'_1, \phi'_2\rangle) \mid D_2 \in chcs(\phi_L) = vunify(hoist(\phi_L, D_2), \phi_R) \tag{4}$$

$$vunify^*(D_1\langle\phi_1, \phi_2\rangle, D_2\langle\phi'_1, \phi'_2\rangle) \tag{5}$$
$$\mid splittable(\phi_L) \neq \varnothing \wedge D_2 \notin chcs(\phi_L) = a_q \leftarrow splittable(\phi_L) \quad \xi \leftarrow \{a_q \mapsto D_2\langle a_{D_2q}, a_{\tilde{D}_2q}\rangle\} \quad \text{return } vunify(hoist(\xi(\phi_L), D_2), \phi_R)$$

$$vunify(D_1\langle\phi_1, \phi_2\rangle, D_2\langle\phi'_1, \phi'_2\rangle) \tag{6}$$
$$\mid splittable(\phi_L) = \varnothing \wedge D_2 \notin chcs(\phi_L) \wedge splittable(\phi_R) = \varnothing \wedge D_1 \notin chcs(\phi_R) = vunify(D_2\langle\phi_L, \phi_L\rangle, \phi_R)$$

$$vunify^*(g, D\langle\phi_1, \phi_2\rangle) = \xi_1 \leftarrow vunify(g, \phi_1) \quad \text{return } \xi_1 \circ vunify(g, \xi_1(\phi_2)) \tag{7}$$

$$vunify^*(\phi \to \phi', D\langle\phi_1, \phi_2\rangle) = vunify(D\langle\phi \to \phi', \phi \to \phi'\rangle, D\langle\phi_1, \phi_2\rangle) \tag{8}$$

$$vunify(\phi_1 \to \phi_2, \phi'_1 \to \phi'_2) = \xi \leftarrow vunify(\phi_1, \phi'_1) \quad \text{return } \xi \circ vunify(\xi(\phi_2), \xi(\phi'_2)) \tag{9}$$

$$vunify^*(a_q, \phi \to \phi') \mid occurs(a_q, \phi_R) = \text{fail} \quad \mid \text{otherwise} = \{a_q \mapsto \phi_R\} \tag{10}$$

Figure 5.3: The qualified unification algorithm.

of the C-C-SWAP rule. It takes as arguments a choice type ϕ and a dimension name D of a (possibly nested) choice type in ϕ, returning a type equivalent to ϕ but with a D choice type at the root. For example, $hoist(A\langle B\langle a, \texttt{Int}\rangle, \texttt{Bool}\rangle, B)$ yields $B\langle A\langle a, \texttt{Bool}\rangle, A\langle \texttt{Int}, \texttt{Bool}\rangle\rangle$. The function $chcs(\phi)$ returns the set of dimension names of all choice types in ϕ. The function *splittable* returns the type vari-

ables that can be split into a choice type. A variable is splittable if the path from itself to the root consists only of choice types; that is, there are no function types between the root of the type and the type variable. For example, $\mathit{splittable}(A\langle \texttt{Int} \to b, c\rangle) = \{c\}$. The final helper function, *occurs*, performs an operation similar to an occurs check, described in the final case below.

Finally, we can describe each case of the *vunify* algorithm as follows.

(1) When unifying two plain types, we defer to Robinson's unification algorithm $\mathcal{U}_R$ presented in Figure 2.2.

(2) To unify a type variable with a choice type, we split the type variable as described earlier in this section.

(3) To unify two choice types in the same dimension, we decompose the problem and unify their corresponding alternatives.

(4) To unify two choice types in different dimensions, we try to hoist a choice type so that both types are rooted by a choice in the same dimension.

(5) If this is impossible, then a splittable type variable is split into a choice in that dimension, which can then be hoisted.

(6) To unify two choice types in different dimensions, where there is no splittable type variable, we partially decompose the problem by unifying each alternative of one choice type with the other choice type.

(7) To unify a ground plain type g with a choice type, we again decompose the problem, unifying g with each alternative of the choice type.

(8) To unify a function type with a choice type in dimension D, we first expand the function type into a choice type in D, similar to the splitting operation on type variables. We then decompose the problem by alternatives.

(9) To unify two function types, we unify their corresponding argument types and return types, composing the results.

(10) Finally, to unify a type variable with a function type, a process similar to an *occurs check* is needed. The operation $occurs(a_q, \phi)$ returns true if there exists a type variable $a_{q'}$ in ϕ such that $q \subseteq q'$. This ensures that we do not assign overlapping type variables to different types, supporting the completion of a qualified unifier back into an unqualified unifier.

5.3 Correctness of the Unification Algorithm

In this subsection we collect several results to demonstrate the correctness of the qualified unification algorithm.

We begin by observing that the operations of decomposition, splitting, and hoisting form the core of the algorithm. In the following lemmas we establish the correctness of these operations. First, we show that the decomposition by alternatives of a qualified unification problem is correct.

Lemma 7 (Decomposition) *Let $\phi_L = D\langle\phi_1, \phi_2\rangle$ and $\phi_R = D\langle\phi'_1, \phi'_2\rangle$. Then $\phi_L \equiv^?_q \phi_R$ is unifiable iff $\phi_1 \equiv^?_q \phi'_1$ and $\phi_2 \equiv^?_q \phi'_2$ are unifiable. Moreover, if the problem is unifiable, then $\xi_1 \cup \xi_2$ is a unifier for $\phi_L \equiv^?_q \phi_R$, where ξ_1 and ξ_2 are unifiers for*

$\phi_1 \equiv^?_q \phi'_1$ *and* $\phi_2 \equiv^?_q \phi'_2$*, respectively.*

PROOF. Observe that qualified type variables with the same variable name but different qualifiers are treated as different type variables. Therefore, given a choice type $D\langle\phi_l, \phi_r\rangle$, it is always the case that $FV(\phi_l) \cap FV(\phi_r) = \varnothing$, by the definition of qualification. Specifically, the qualifier of every type variable in ϕ_l will contain the selector D, while the qualifier of every type variable in ϕ_r will contain the selector $\tilde{D}$. Since this property holds for both choice types in the lemma, the unification subproblems do not share any common type variables and are therefore independent. □

Next we show that splitting a type variable is variable independent. This means that when more than one type variable is splittable, we will achieve an equivalent unifier no matter which type variable we choose to split.

Lemma 8 (Variable independence) *Let* $\phi_L = D_1\langle\phi_1, \phi_2\rangle$ *and* $\phi_R = D_2\langle\phi'_1, \phi'_2\rangle$*. Assume* $a_q, b_r \in splittable(\phi_L)$*, where qualifiers* q *and* r *do not contain selectors in dimension* D_2*. Let* $\theta_a = \{a_q \mapsto D_2\langle a_{D_2 q}, a_{\tilde{D}_2 q}\rangle\}$ *and* $\theta_b = \{b_r \mapsto D_2\langle b_{D_2 r}, b_{\tilde{D}_2 r}\rangle\}$*. Then* $vunify(\theta_a(\phi_L), \phi_R) = vunify(\theta_b(\phi_L), \phi_R)$*.*

PROOF. After splitting a type variable a_q (or b_r) in ϕ_L, the newly formed choice type must be hoisted to the top so that the unification problem can be decomposed by alternatives. After applying this series of hoists, we obtain a new type ϕ'_L with a choice type in dimension D_2 at the top level. The lemma depends crucially on the fact that no matter which type variable we split, ϕ'_L will be the same, and therefore the result of the unfication will be the same. This is true because the

$$
\begin{array}{r}
D_1\langle \underset{}{a_q}, D_3\langle \phi_1, b_r\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle \\
\underset{split\downarrow}{} \\
D_1\langle \underline{D_2}\langle a_{D_2 q}, a_{\tilde{D}_2 q}\rangle, D_3\langle \phi_1, b_r\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle \\
\underset{hoist\downarrow}{} \\
D_2\langle D_1\langle a_{D_2 q}, D_3\langle \phi_1, b_{D_2 r}\rangle\rangle, D_1\langle a_{\tilde{D}_2 q}, D_3\langle \phi_1, b_{\tilde{D}_2 r}\rangle\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle
\end{array}
$$

$$
\begin{array}{r}
D_1\langle a_q, D_3\langle \phi_1, \underline{b_r}\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle \\
\underset{split\downarrow}{} \\
D_1\langle a_q, D_3\langle \phi_1, \underline{D_2}\langle b_{D_2 r}, b_{\tilde{D}_2 r}\rangle\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle \\
\underset{hoist\downarrow}{} \\
D_1\langle a_q, \underline{D_2}\langle D_3\langle \phi_1, b_{D_2 r}\rangle, D_3\langle \phi_1, b_{\tilde{D}_2 r}\rangle\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle \\
\underset{hoist\downarrow}{} \\
D_2\langle D_1\langle a_{D_2 q}, D_3\langle \phi_1, b_{D_2 r}\rangle\rangle, D_1\langle a_{\tilde{D}_2 q}, D_3\langle \phi_1, b_{\tilde{D}_2 r}\rangle\rangle\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle
\end{array}
$$

Figure 5.4: Demonstration of variable independence.

process of hoisting the new D_2 choice type will essentially cause every other type variable in ϕ_L to be split in dimension D_2. This process is described below.

By the definition of *splittable*, the path from the top of ϕ_L to a_q (or b_r) consists of only choice types. For each choice type $D_i\langle \ldots \rangle$ along this path, we must apply *hoist* once in order to lift the D_2 choice type outward one level. Without loss of generality, assume that the D_2 choice type is in the left alternative, so $D_i\langle D_2\langle \phi_1, \phi_2\rangle, \phi_3\rangle$. After applying *hoist*, we have $D_2\langle D_i\langle \phi_1, \phi_3\rangle, D_i\langle \phi_2, \phi_3\rangle\rangle$. Since ϕ_3 was copied into both alternatives of the D_2 choice type, every type variable in the first ϕ_3 will be qualified by D_2 while every type variable in the second ϕ_3 will be qualified by $\tilde{D}_2$. In this way, the process of hoisting the D_2 choice type to the top level will cause every other type variable to be split into two type variables qualified by selectors for dimension D_2. □

The process described in the proof of Lemma 8 is illustrated in Figure 5.4 with a small example. In this example, $\phi_L = D_1\langle a_q, D_3\langle \phi_1, b_r\rangle\rangle$ and $\phi_R = D_2\langle \phi_2, \phi_3\rangle$, where q does not contain qualifiers in D_2 or D_3 and r does not contain a qualifier in D_2. In the top case, we split a_q in dimension D_2, while in the bottom, we split b_r. Observe how type variables are copied and qualified when a choice type is hoisted over them.

Just as it does not matter which splittable type variable we choose, it does not matter which dimension we choose to split it in, as long as the type variable is not already qualified by that dimension.

Lemma 9 (Choice independence) *Let* $\phi_L = D_1\langle \phi_1, \phi_2\rangle$ *and* $\phi_R = D_2\langle \phi_1', \phi_2'\rangle$. *Assume* $D_m, D_n \in \mathit{chcs}(\phi_R)$ *and* $a_q \in \mathit{splittable}(\phi_L)$, *where qualifier* q *does not contain selectors in* D_m *or* D_n. *Let* $\theta_1 = \{a_q \mapsto D_m\langle a_{D_m q}, a_{\tilde{D_m} q}\rangle\}$ *and* $\theta_2 = \{a_q \mapsto D_n\langle a_{D_n q}, a_{\tilde{D_n} q}\rangle\}$. *Then* $\mathit{vunify}(\theta_1(\phi_L), \phi_R) = \mathit{vunify}(\theta_2(\phi_L), \phi_R)$.

PROOF. Assume without loss of generality that we split a_q in dimension D_m. If $D_m = D_2$, we can make progress by hoisting the newly created choice type in D_m to the top of ϕ_L and decomposing the resulting unification problem by alternatives. Otherwise, we will have to also hoist the choice in D_m to the top of ϕ_R, then decompose. Either way, this will result in at least two new type variables, $a_{D_m q'}$ (in the left decomposition of ϕ_L) and $a_{\tilde{D_m} q''}$ (in the right decomposition of ϕ_L). Since q did not contain a selector in D_n, there is no choice type on the path from a_q to the root of ϕ_L, so neither q' nor q'' will contain a selector in D_n, which might otherwise have been introduced during the hoisting process. However, since a

choice type in D_n still exists in at least one of the two subproblems, we will have to split one or both of $a_{D_m q'}$ and $a_{\tilde{D_m} q''}$ in dimension D_n in order to complete the unification. Therefore, since we must eventually split the original a_q in both D_m and D_n, proving choice independence is equivalent to proving that the order in which we perform these splits does not affect the unification result.

Suppose we perform both splits before doing any decompositions. If we split a_q in D_m first and then D_n, a_q will be replaced by the type ϕ_a below. If we split a_q in D_n and then D_m, it will be replaced by the type ϕ'_a.

$$\phi_a = D_m\langle D_n\langle a_{D_m D_n q}, a_{D_m \tilde{D_n} q}\rangle, D_n\langle a_{\tilde{D_m} D_n q}, a_{\tilde{D_m}\tilde{D_n} q}\rangle\rangle$$
$$\phi'_a = D_n\langle D_m\langle a_{D_m D_n q}, a_{\tilde{D_m} D_n q}\rangle, D_m\langle a_{D_m \tilde{D_n} q}, a_{\tilde{D_m}\tilde{D_n} q}\rangle\rangle$$

It is easy to see that $\phi_a \equiv \phi'_a$ by the equivalence rules in Figure 4.4. We can transform ϕ_a into ϕ'_a by applying the C-C-SWAP rules to each alternative, then applying the C-C-MERGE rules to eliminate the dominated choice types. Since ϕ_a and ϕ'_a are equivalent, then $\phi'_L = \{a_q \mapsto \phi_a\}(\phi_L)$ and $\phi''_L = \{a_q \mapsto \phi'_a\}(\phi_L)$ are also equivalent, so the results of unifying $\phi'_L \equiv^?_q \phi_R$ and $\phi''_L \equiv^?_q \phi_R$ will be the same. □

The process described in the proof of Lemma 9 is illustrated in Figure 5.5. In this example, $\phi_L = D_1\langle a_q, \phi_1\rangle$ and $\phi_R = D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle$, where the qualifier q does not contain qualifiers in dimensions D_2 or D_3. In the top case we split a_q into a choice type in dimension D_2, eventually yielding three unification subproblems. In the bottom case we split a_q in dimension D_3, eventually yielding four subproblems. (The vertical ellipses in each of these derivations represent further splitting the type variable in dimension D_3, hoisting this choice to the top level, and decom-

$$
\begin{array}{rl}
D_1\langle \underline{a_q}, \phi_1\rangle \equiv_q^? & D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle \\
\quad \underline{\quad}_{split}\downarrow & \\
D_1\langle \underline{D_2}\langle a_{D_2 q}, a_{\tilde{D}_2 q}\rangle, \phi_1\rangle \equiv_q^? & D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle \\
\quad \underline{\quad}_{hoist}\downarrow & \\
D_2\langle D_1\langle a_{D_2 q}, \phi_1\rangle, D_1\langle a_{\tilde{D}_2 q}, \phi_1\rangle\rangle \equiv_q^? & D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle \\
\quad \swarrow\!\!\searrow & \\
D_1\langle a_{D_2 q}, \phi_1\rangle \equiv_q^? \phi_2 & D_1\langle a_{\tilde{D}_2 q}, \phi_1\rangle \equiv_q^? D_3\langle \phi_3, \phi_4\rangle \\
 & \quad \vdots \\
 & D_1\langle a_{\tilde{D}_2 D_3 q}, \phi_1\rangle \equiv_q^? \phi_3 \\
 & D_1\langle a_{\tilde{D}_2 \tilde{D}_3 q}, \phi_1\rangle \equiv_q^? \phi_4
\end{array}
$$

$$
\begin{array}{rl}
D_1\langle \underline{a_q}, \phi_1\rangle \equiv_q^? & D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle \\
\quad \underline{\quad}_{split}\downarrow & \\
D_1\langle \underline{D_3}\langle a_{D_3 q}, a_{\tilde{D}_3 q}\rangle, \phi_1\rangle \equiv_q^? & D_2\langle \phi_2, D_3\langle \phi_3, \phi_4\rangle\rangle \\
\quad \underline{\quad}_{hoist}\downarrow & \\
D_3\langle D_1\langle a_{D_3 q}, \phi_1\rangle, D_1\langle a_{\tilde{D}_3 q}, \phi_1\rangle\rangle \equiv_q^? & D_2\langle \phi_2, \underline{D_3}\langle \phi_3, \phi_4\rangle\rangle \\
 & \quad \underline{\quad}_{hoist}\downarrow \\
D_3\langle D_1\langle a_{D_3 q}, \phi_1\rangle, D_1\langle a_{\tilde{D}_3 q}, \phi_1\rangle\rangle \equiv_q^? & D_3\langle D_2\langle \phi_2, \phi_3\rangle, D_2\langle \phi_2, \phi_4\rangle\rangle \\
\quad \swarrow\!\!\searrow & \\
D_1\langle a_{D_3 q}, \phi_1\rangle \equiv_q^? D_2\langle \phi_2, \phi_3\rangle & D_1\langle a_{\tilde{D}_3 q}, \phi_1\rangle \equiv_q^? D_2\langle \phi_2, \phi_4\rangle \\
\quad \vdots & \quad \vdots \\
D_1\langle a_{D_2 D_3 q}, \phi_1\rangle \equiv_q^? \phi_2 & D_1\langle a_{D_2 \tilde{D}_3 q}, \phi_1\rangle \equiv_q^? \phi_2 \\
D_1\langle a_{\tilde{D}_2 D_3 q}, \phi_1\rangle \equiv_q^? \phi_3 & D_1\langle a_{\tilde{D}_2 \tilde{D}_3 q}, \phi_1\rangle \equiv_q^? \phi_4
\end{array}
$$

Figure 5.5: Demonstration of choice independence.

posing by alternatives.) However, observe that the subproblem on the left branch of the top case is equivalent to the two subproblems in the bottom case that have ϕ_2 on their RHS. In order to obtain the subproblems in the bottom case, we can use choice idempotency to rewrite ϕ_2 to $D_3\langle \phi_2, \phi_2\rangle$, then decompose by alternatives.

Since the hoisting operation only restructures a type in a semantics-preserving way, its correctness is obvious.

Our unification algorithm is terminating through decomposition that eventually produces calls to Robinson's unification algorithm (which is terminating). The only challenge to termination is that the splitting of type variables introduces new choice types to the types that are being unified. However, two facts ensure that this does not prevent termination: (1) a variable can only be split into a choice type whose dimension occurs in the type being unified against and (2) immediately after a split is performed the new choice type is hoisted and decomposed, producing two subtasks that are each smaller than the original task.

The qualified unification algorithm is sound, meaning the mappings it produces always unify its arguments. It is also complete and most general, which means that if the two types are unifiable, then the algorithm will return the most general mapping that unifies them. We express each of these results in the following theorems.

Theorem 6 (Soundness) *If* $vunify(\phi_1, \phi_2) = \xi$*, then* $\xi(\phi_1) \equiv \xi(\phi_2)$.

Theorem 7 (Complete and most general)
If $\xi(\phi_1) \equiv \xi(\phi_2)$*, then* $vunify(\phi_1, \phi_2) = \xi'$ *where* $\xi = \xi'' \circ \xi'$ *for some* ξ''.

A proof of Theorem 6 is given in Appendix B.

In order to prove the correctness of CT-unification, we must relate the above theorems on qualified unification to the problem of variational unification. To do this, we must first establish the relationships between the *comp* function, the qualifiers, the unification problem, and the qualified unification problem.

The following lemma states the expectations for the *comp* function, which transforms a mapping from a single qualified type variable into a mapping from an unqualified type variable to a partially completed type. The lemma is proved in Appendix B, demonstrating that *comp* is correct.

Lemma 10 *Given a mapping* $\{a_q \mapsto \phi'\}$*, if* $\phi = comp(q, \phi', b)$ *is the completed type (where* b *is fresh), then* $\lfloor \phi \rfloor_q = \lfloor \phi' \rfloor_q$*. More generally, given* $\{a_{q_1} \mapsto \phi_1, \ldots, a_{q_n} \mapsto \phi_n\}$*, if* $\phi = comp(q_1, \phi_1, comp(q_2, \phi_2, \ldots comp(q_n, \phi_n, b)\ldots))$*, then for every* $q_i \in \{q_1 \ldots q_n\}$*, we have* $\lfloor \phi \rfloor_{q_i} = \lfloor \phi_i \rfloor_{q_i}$.

PROOF. We can prove the first part of this theorem by structural induction on the qualifier q. The base case, where q is the empty qualifier ϵ, is trivial since $comp(\epsilon, \phi', b) = \phi'$. We show the inductive case below for $q = Dq'$ (the case for $q = \tilde{D}q'$ is a dual). Note that the induction hypothesis is $\lfloor comp(q', \phi', b) \rfloor_{q'} = \lfloor \phi' \rfloor_{q'}$.

$$
\begin{array}{lll}
\lfloor \phi \rfloor_q = \lfloor comp(Dq', \phi', b) \rfloor_{Dq'} & & \text{by assumption} \\
= \lfloor D\langle comp(q', \phi', b), fresh(b) \rangle \rfloor_{Dq'} & & \text{definition of } comp \\
= \lfloor \lfloor comp(q', \phi', b) \rfloor_D \rfloor_{q'} & & \text{definition of repeated selection} \\
= \lfloor \lfloor comp(q', \phi', b) \rfloor_{q'} \rfloor_D & & \text{selector ordering is irrelevant} \\
= \lfloor \lfloor \phi' \rfloor_{q'} \rfloor_D & & \text{induction hypothesis} \\
= \lfloor \phi' \rfloor_q & & \text{selector ordering is irrelevant}
\end{array}
$$

We can prove the second part by induction on the mapping of qualified type variables, using the result from the first part and the observation that *comp* is commutative, for example, $comp(q_1, \phi_1, comp(q_2, \phi_2, b)) \equiv comp(q_2, \phi_2, comp(q_1, \phi_1, b))$.

□

Using this result, we can prove the correctness of the completion process. Given a

solution to a qualified unification problem, completion produces a solution to the original unqualified version. The lemma below states the expectation of completion with respect to the selection semantics. It is also proved in Appendix B.

Lemma 11 (Completion) *Given a CT-unification problem $\phi_L \equiv^? \phi_R$ and the corresponding qualified unification problem $\phi'_L \equiv^?_q \phi'_R$, if σ_Q is a unifier for $\phi'_L \equiv^?_q \phi'_R$ and σ_U is the unifier attained by completing σ_Q, then for any super-complete decision δ, $\lfloor \sigma_U(\phi_L) \rfloor_\delta \equiv \lfloor \sigma_Q(\phi'_L) \rfloor_\delta$ and $\lfloor \sigma_U(\phi_R) \rfloor_\delta \equiv \lfloor \sigma_Q(\phi'_R) \rfloor_\delta$.*

Completion also preserves principality since the *comp* function adds fresh type variables everywhere except at the leaf addressed by the path q (maximizing generality), and the principal type inferred during qualified unification is inserted directly at q.

The following theorem generalizes Lemma 11, stating that through qualification and completion, we can solve CT-unification problems. We call this process *variational unification*.

Theorem 8 *Given a CT-unification problem U and the corresponding qualified unification problem Q, if σ_Q is a unifier for Q, then we can attain a unifier σ_U for U through the process of completion.*

Variational unification is sound, complete, and most general since the underlying qualified unification algorithm has these properties and since completion preserves principality.

5.4 Time Complexity

Solving the unification problem U consists of three steps: (1) transforming U into the corresponding qualified unification problem Q, (2) solving Q with the qualified unification algorithm *vunify*, and (3) transforming the qualified unifier into the variational unifier through completion. To determine the time needed to solve U, we will consider the time complexity of each step in turn. As before, we use ϕ_L and ϕ_R to denote the LHS and RHS of U. We use σ_U and σ_Q to denote the unifier for U and Q, respectively. The size of a type ϕ, denoted by $|\phi|$, is the number of nodes in its AST (as defined by the grammar in Section 4.1). We assume that $|\phi_L| = l$ and $|\phi_R| = r$. The size of a unification problem is the sum of the sizes of the types being unified.

The process of transforming U to Q qualifies each type variable in U. This can be achieved by a top-down traversal of the ASTs of ϕ_L and ϕ_R. Thus, the complexity of this process is $O(l + r)$. Note that the resulting qualified problem Q is the same size as the original U.

For the second step of solving Q, we do a worst-case complexity analysis. For simplicity, assume that the internal nodes of ϕ_L and ϕ_R are all choice types. Then the worst case for unification is when $chcs(\phi_L) \cap chcs(\phi_R) = \varnothing$. When ϕ_L and ϕ_R have no choices in common, we proceed by (1) splitting a type variable in one of the types, say ϕ_L, into a choice type in the dimension of the root choice of the other type, ϕ_R; (2) hoisting the new choice type to the root of ϕ_L; and (3) decomposing the problem by alternatives. Splitting and hoisting the new choice type increases the

size of the LHS to $1+2l$: 1 for the new choice type plus $2l$ for the copy of ϕ_L in each alternative with extended qualifications on its type variables. The splitting and hoisting process can be performed in $O(l)$ time by introducing the new choice type, copying ϕ_L into each alternative, and then traversing each alternative, qualifying the type variables accordingly.

After decomposing the problem by alternatives, we are left with two smaller subproblems $\phi_{L_1} \equiv_q^? \phi_{R_1}$ and $\phi_{L_2} \equiv_q^? \phi_{R_2}$. We know that $|\phi_{L_1}| = |\phi_{L_2}| = |\phi_L|$ since ϕ_{L_1} and ϕ_{L_2} are just copies of ϕ_L with different type variable qualifiers. Moreover, $|\phi_{R_1}| + |\phi_{R_2}| = |\phi_R| - 1$ since ϕ_{R_1} and ϕ_{R_2} are the left and right branch of the root node of ϕ_R, respectively. The split-hoist-decompose process will be recursively applied to the subproblems $\phi_{L_1} \equiv_q^? \phi_{R_1}$ and $\phi_{L_2} \equiv_q^? \phi_{R_2}$. After two more decompositions, there will be four unification subproblems. Since there are $(r-1)/2$ internal nodes, there will be $(r-1)/2$ decompositions, and since each decomposition takes $O(l)$ time, the whole decomposition takes $O(l \cdot (r-1)/2)$ time.

We can also observe that each decomposition by alternatives creates two subproblems from one. This will result in $(r+1)/2$ subproblems, one from the decomposition corresponding to each choice node in the tree. Based on the decomposition process, each resulting subproblem is either of the form $\phi'_L \equiv_q^? g$ or $\phi'_L \equiv_q^? a_q$, where ϕ'_L differs from ϕ_L only in type variable qualifications, g is a ground type, and a_q is a qualified type variable. Based on cases (2) and (7) of the unification algorithm, each final subproblem therefore takes $O(|\phi'_L|) = O(|\phi_L|) = O(l)$ to solve. Thus the time needed to solve all subproblems is $O(l \cdot (r+1)/2)$.

Summing the time needed for decomposition, $O(l \cdot (r-1)/2)$, and the time needed

for solving the resulting unification problems, $O(l \cdot (r+1)/2)$, the time complexity of solving Q is $O(lr)$.

Finally, we consider the complexity of the third step of the unification process, transforming the solution σ_Q for Q into a solution σ_U for U through the process of completion. Again, we perform a worst-case analysis. Completion is performed by folding the mappings in σ_Q with the function *comp*. We can establish an upper bound on the number of mappings in σ_Q by following the decomposition process in the previous step and counting the number of potential type variables. At the end of this process we have at most $(r-1)/2$ subproblems of the form $\phi'_L \equiv^?_q \phi$. Each ϕ'_L is of size l and contains at most $(l+1)/2$ type variables at the leaves; each ϕ is either a type variable or a ground plain type. Therefore, after some simplification, σ_Q contains at most $(1/2)(r-1)(l+1)$ mappings.

If we think of the completion process as incrementally building up a tree of nested choices that describe the result type ϕ, then each mapping $a_{q_i} \mapsto \phi_i \in \sigma_Q$ essentially describes a leaf in that tree. Applying *comp* to such a mapping constitutes traversing ϕ according to the path described by q_i, possibly generating at most one new choice type and one new type variable (if this is the first traversal along this path) at each step of the way; this takes $O(|q_i|)$ time, where $|q_i|$ is the length of the qualifier. The length of the qualifier is in turn bounded by the total number n of dimensions present in the unification problem. An upper bound on n can be expressed in terms of l and r as $(l-1)/2 + (r-1)/2$ since there is at most one dimension name for each internal node in the original types ϕ_L and ϕ_R. Finally, since a single completion step takes time $O(n) = O(l+r)$ and we will perform

$O(lr)$ completion steps (one for each mapping), the total time for completion is $O(l^2r + lr^2)$.

Summing these three steps, we see that the completion step dominates the others, so the unification of variational types takes cubic time, in the worst case, with respect to the size of the types. When unifying types that contain choice types in the same dimension, we can expect the complexity of unification to be much lower.

Chapter 6: Variational Type Inference

This chapter presents a variational type inference algorithm in Section 6.1 and a study of performance gain of variational type inference over the brute-force strategy of type inference in Section 6.2.

6.1 An Inference Algorithm

Although the unification algorithm for VLC differs significantly from Robinson's unification algorithm, the type inference algorithm is only a simple extension of algorithm $\mathscr{W}$ for HM [Damas and Milner, 1982]. We call this algorithm *vinfer* and its type is given below.

$$vinfer : \Gamma \times e \rightarrow \xi \times \phi$$

The function takes two arguments: the type environment $\Gamma$and the expression to type. It returns a type substitution and the inferred type. The cases of the *vinfer* algorithm can be derived from the typing rules in Section 4.2. The cases for choices and applications are given below.

On a choice, we determine the alternative types in the result by inferring the type of each alternative expression. Note that we apply the mapping produced by inferring the type of e_1 to the typing environment used to infer the type of e_2. This ensures that the result types and mappings will be consistent. Finally, the result

$vinfer(\Gamma, D\langle e_1, e_2\rangle) =$
  $(\xi_1, \phi_1) \leftarrow vinfer(\Gamma, e_1)$
  $(\xi_2, \phi_2) \leftarrow vinfer(\xi_1(\Gamma), e_2)$
  return $(\xi_2 \circ \xi_1, D\langle \phi_1, \phi_2\rangle)$

$vinfer(\Gamma, e_1\ e_2) =$
  $(\xi_1, \phi_1) \leftarrow vinfer(\Gamma, e_1)$
  $(\xi_2, \phi_2) \leftarrow vinfer(\xi_1(\Gamma), e_2)$
  $\xi \leftarrow vunify'(\xi_2(\phi_1), \phi_2 \to a)$ where a fresh
  return $(\xi \circ \xi_2 \circ \xi_1, \xi(a))$

Figure 6.1: The variational type inference algorithm.

mapping is just a composition of the mappings produced during type inference of the two alternatives.

The *vinfer* algorithm types applications as in $\mathcal{W}$, except replacing the unification algorithm with our own variational unification algorithm (and propagating the additional environments). We use $vunify'$ to represent the combined qualification, unification, and completion process. That is, first the type variables in ϕ_1 and ϕ_2 are qualified, then *vunify* is invoked on the transformed types, and finally the resulting mapping is completed to produce ξ, the solution to the original unqualified unification problem. The remaining cases are similarly straightforward. Abstractions, λ-bound variables, and **let** expressions are exactly as in $\mathcal{W}$.

The following theorem expresses the standard property of soundness for the variational type inference algorithm.

Theorem 9 (Type inference is sound) $vinfer(\Gamma, e) = (\xi, \phi) \implies \xi(\Gamma) \vdash^V e : \phi$.

PROOF. The *vinfer* algorithm is directly derived from the typing rules and based on algorithm $\mathcal{W}$, which is sound. The only challenge to soundness comes from

the divergence from $\mathcal{W}$ on applications, where we replace the standard unification algorithm with *vunify*′. However, since variational unification is also sound per Theorem 8, this property is preserved. □

The type inference algorithm also has the principal type property, which follows from Theorems 7 and 8.

Theorem 10 (Type inference is complete and principal) *For every mapping* ξ *and type* ϕ *such that* $\xi(\Gamma) \vdash^V e : \phi$*, there exists a* ξ' *and* ϕ' *such that vinfer*$(\Gamma, e) = (\xi', \phi')$ *where* $\xi = \xi'' \circ \xi'$ *for some* ξ'' *and* $\phi = \xi'''(\phi')$ *for some* ξ'''.

These results are important because they demonstrate that properties from other type systems can be preserved in the context of variational typing.

6.2 Performance Evaluation

For any static variation representation (such as the choice calculus) applied to a statically-typed object language, there exists a trivial typing algorithm: generate every program variant, then type each one individually using the non-variational type system of the object language. We call this the "brute-force" strategy. There are two significant advantages of a more integrated approach using variational types. The first is that we can characterize the variational structure of types present in variational software—this is useful for aiding understanding of variational software and informing decisions about which program variant to select. The second is that we can gain significant efficiency improvements over the brute-force strategy. Due to the combinatorial explosion of program variants as we add

new dimensions of variation, separately inferring or checking the types of all program variants quickly becomes infeasible. In this section we describe how variational type systems, and our type system for VLC in particular, can increase the efficiency of type inference for variational programs, making typing possible for massively variable systems. We do this in two ways: by analytically characterizing the opportunities for efficiency gains, and by demonstrating these gains experimentally.

6.2.1 Analytical Characterization of Efficiency Gains

Although we have considered only binary dimensions so far, we assume in this discussion that the variational type system has been extended to support arbitrary n-ary dimensions. While this extension is not interesting from a technical perspective, it is important for practical use and accentuates the potential for efficiency gains.

An important observation is that the *worst-case* performance of any variational type system is guaranteed to be no better than the brute-force strategy, assuming the variation representation is sufficiently general. Consider the following VLC expression.

$$A\langle B\langle e_1, e_2\rangle, B\langle e_3, e_4\rangle\rangle$$

If e_1, e_2, e_3, and e_4 contain no common parts that can be factored out, there is simply no improvement to be made over the brute-force strategy. We must type each of the four expressions separately, and the type of each one provides no

insight into the types of others. Fortunately, we expect there to be many more opportunities for improvement in actual software. In this section, we describe the two basic ways that variational typing can save over the brute-force strategy, characterizing the efficiency gains by each. Since these patterns are expected to be ubiquitous in practice, variational typing can likewise be expected to be much more efficient.

The first opportunity for efficiency gains arises because choices capture variation *locally*. This allows the type system to reuse the types inferred for the common context of the alternatives in a choice. Suppose we have a choice $D\langle e_1,\ldots,e_n\rangle$ in a non-variational context C. Conceptually, a context is an expression with a hole; we can fill that hole with the choice above to produce the overall expression, which we write as $C[D\langle e_1,\ldots,e_n\rangle]$. Our algorithm types the contents of C only once, whereas the brute-force strategy would type each $C[e_i]$ separately, typing C a total of n times. While the work performed on C by our algorithm is constant, the extra work performed by the brute-force strategy obviously grows multiplicatively with the size of each new dimension of variation. We can maximize the benefits of choice locality gains by ensuring that choices are *maximally factored*. Erwig and Walkingshaw [2011] say that such expressions are in *choice normal form*, and they provide a semantics-preserving transformation to achieve this desirable state.

The second, more subtle opportunity involves the typing of applications between two choices, for example, $A\langle e_1,\ldots,e_n\rangle\ B\langle e'_1,\ldots,e'_m\rangle$. Since the brute-force strategy considers every variant individually, it must unify the type of every alternative in the first choice with the type of every alternative in the second choice,

for a total of $n \cdot m$ unifications. The ability to see all variants together provides substantial opportunity for speed-up if several alternatives in either choice have the same type. For example, if the alternatives of the choice in dimension A have $k < n$ unique types and the alternatives of the choice in B have $l < m$ unique types, then the type inference algorithm must invoke unification at most $k \cdot l$ times. Since we expect it to often be the case that all alternatives of a choice have the *same* type (consider varying the values of constants, the names of variables, or only the implementation of a function), this offers a dramatic opportunity for efficiency gains.

6.2.2 Experimental Demonstration

In this section we continue the efficiency discussion through a series of three experimental demonstrations of the performance of our variational type inference algorithm, *vinfer*. First, we illustrate the savings described in the analytical evaluation. Second, we describe a degenerate scenario that induces poor performance in *vinfer*, but show that it can still exploit sharing to perform better than the brute-force algorithm. Third, we demonstrate the performance of *vinfer* on large and complex, randomly generated expressions. These experiments are not intended as a rigorous or real-world experimental evaluation of the variational type inference algorithm, but as a vehicle to further the discussion.

The experiments are based on a Haskell prototype that implements the ideas and algorithms presented in Chapters 4, 5, and 6. The prototype consists of three parts: the variational type normalizer, the equational unification algorithm, and

the type inference algorithm. The prototype implements most of the features described in these three chapters. One exception is that the second opportunity for efficiency gains, described in the previous subsection, is exploited only in the special case when all variants of a choice have the same type.

6.2.2.1 Illustration of Efficiency Gains

In this part, we will be referring to the expressions and results in Figure 6.2. The leftmost column names each expression and the next column defines it. The *dims* column indicates the number of different choice names in the expression. The *variants* column indicates the total number of variants, which can be calculated by multiplying the arity of each of the dimensions. The timing results are given in the final three columns. The *one* column indicates the time needed to infer the type of a *single* program variant. This is intended as a reference point for comparison with the other two timing results. The *brute* column gives the time to infer the type of each variant separately using the brute-force strategy, and *vlc* gives the time taken by *vinfer* to infer a variational type for the expression.

All times are calculated within our prototype. In the absence of variation (when inferring types for *one* and *brute*), the prototype reduces to a standard implementation of algorithm $\mathcal{W}$. The typing environment is seeded with boolean and integer values that map to constant types `Bool` and `Int`, and several simple functions like `id`, `not`, `succ`, and `even` that map to the expected types. The function `if` has type $\texttt{Bool} \to \alpha \to \alpha \to \alpha$.

	expression	dims	variants	one	brute	vlc
e_1	$A\langle\lambda\mathtt{x}.3, \lambda\mathtt{x}.\mathtt{true}\rangle\ B\langle 3,5\rangle$	2	4	2.6	10.2	10.1
e_2	$A\langle\lambda\mathtt{x}.3, \lambda\mathtt{x}.\mathtt{true}\rangle\ B\langle 3,5,7,9\rangle$	2	8	2.5	20.4	10.2
e_3	$\mathtt{if}\ \mathtt{true}\ e_1\ e_1'$	4	16	21.2	334.7	34.3
e_4	$A\langle B\langle \mathtt{succ}, \lambda\mathtt{x}.\mathtt{true}\rangle, B\langle \lambda\mathtt{x}.3, \mathtt{not}\rangle\rangle\ B\langle 3, \mathtt{true}\rangle$	2	4	2.6	10.1	15.7
e_5	$\mathtt{id}\ \mathtt{id}\ \mathtt{id}\ e_4$	2	4	31.9	125.1	63.0
e_6	$\mathtt{if}\ \mathtt{true}\ e_4\ e_4'$	4	16	24.0	380.5	55.0

Figure 6.2: Running times of type inference strategies on several examples. Each test was run 200,000 times on a 2.8GHz dual core processor with 3GB of RAM. All times are in seconds.

The first group of expressions demonstrates some basic relationships between the number and arity of dimensions and the potential efficiency gains of variational type inference. In e_1 we present a simple unification problem with an opportunity for sharing (both alternatives of the B choice have type `Int`). Since the number of variants is so small, the overhead of *vinfer* negates the gains made by sharing, and the algorithm performs equivalently to the brute-force strategy. However, this quickly changes as we add variants and additional context. In e_2 we have doubled the number of variants by increasing the number of alternatives in the B dimension from 2 to 4. While the running time for the brute-force strategy correspondingly doubles, variational type inference does not since the new alternatives can also be shared. Finally, in e_3 we double the number of dimensions to increase the number of variants by a factor of four. The expression e_1' is identical to e_1, except with unique dimension names. This example also adds some additional, unvaried context (`if True`). Now we can see the exponential explosion of the brute-force strategy (which must also type the common context 16 times),

while *vinfer* scales essentially linearly with respect to the size of the expression. We can also observe that the ratio of overhead for *vinfer*, relative to the reference single-variant inference time, decreases as we increase the size of the expression.

In the second group we begin with a more complex variational structure with no opportunities for sharing, e_4. As expected, *vinfer* performs worse than brute-force due to the overhead. With e_5, however, we demonstrate how even a very small amount of common context can tip the scale back in *vinfer*'s favor. If we again duplicate the initial expression and rename the dimension names, as in e_6, we introduce an opportunity for sharing, allowing *vinfer* to scale nicely while brute-force does not.

6.2.2.2 Cascading Choice Problem

In this part, we analyze the impact of a difficult case for our algorithm that we call *cascading choices*. This occurs when we have a long sequence of choices, each in a different dimension, connected by applications. If there are few opportunities for sharing, the result type produced from the first unification can be expanded by the second, third, and so on, potentially building up a result type exponentially and making each successive unification more expensive than the last.

Figure 6.3 demonstrates the performance of *vinfer* and the brute-force algorithm on expressions designed to induce the cascading choice problem. In the left graph, the x-axis indicates the number of dimensions in the expression, and the y-axis gives the running time on a logarithmic scale. The leftmost expression with

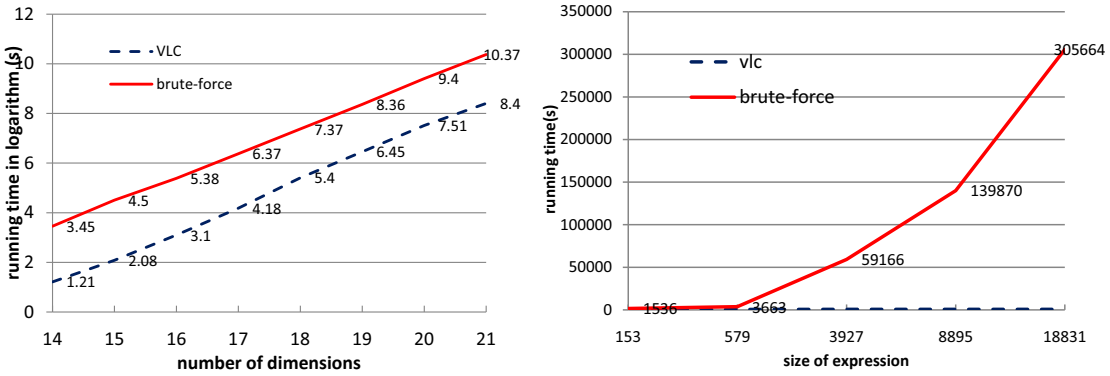


Figure 6.3: Performance comparison for the cascading choice problem.

14 dimensions has 16384 variants and produces a result type with 14335 different variants. We can observe that the running time of *vinfer* is exponential with regard to the number of dimensions. However, it still performs slightly better than the brute-force strategy because it takes advantage of the few opportunities for sharing available.

While *vinfer* is sensitive to the number of dimensions in expressions inducing the cascading choice problem, it is less sensitive to the overall size of the expression. This is in stark contrast to the brute-force strategy, as illustrated in the right graph in Figure 6.3. Here, we fix the length of cascading choices at 21 but increase the size of the expression by making the alternatives in each choice more complex. The x-axis shows this size (in number of AST nodes) and the y-axis shows the running time in seconds. We observe that the brute-force strategy grows sharply as the size of the alternatives increases since each will be typed several times. This additional work will be shared in *vinfer*, however, and so the running time grows much less (increasing from 338 seconds to 461).

size	dims	nc/nd	na/s	nesting	cascading	one	brute	vlc
569	27	5.38	0.070	11(1)	11(1)	0.0011	148504	0.52
3505	57	6.61	0.279	12(1)	11(2)	0.0236	-	0.57
8153	168	6.21	0.250	12(4)	12(4)	0.0583	-	2.14
9429	215	6.67	0.218	12(7)	12(5)	0.0510	-	3.16
29481	681	6.87	0.210	12(25)	12(17)	0.154	-	10.16
61345	1434	7.05	0.203	12(56)	12(37)	0.321	-	21.67
213521	4983	7.03	0.203	12(183)	12(119),13(3)	1.10	-	76.98
429586	10002	7.08	0.202	17(2),12(287)	19(1),12(229)	2.17	-	142.33

Figure 6.4: Running times of type inference for large expressions (in seconds).

6.2.2.3 Performance on Large Expressions

Finally, in Figure 6.4, we demonstrate the efficiency of type inference on several large, randomly generated expressions. These expressions are generated in several steps. First, we add several functions and their corresponding types to an initial environment. Then we manually build up a library of small (potentially variational) expressions and add these to the environment. We use these seeded expressions as building blocks for randomly constructing larger expressions by picking a random function from the environment, picking random arguments that will satisfy its type, possibly changing the dimension names, then joining these expressions with applications. Finally, we add this new expression back into the environment and repeat until the environment contains expressions of the desired size and complexity.

The table reports the running time for many large expressions. It gives the size of each expression as the number of AST nodes and the number of contained dimensions. Each dimension is binary, so an expression with nd dimensions will

describe 2^{nd} total variants. The next four columns characterize the composition and structure of the expression. We indicate the ratio nc/nd of choices to dimensions, and the ratio na/s of application nodes to the size of the expression. In general, we would expect a higher ratio of application nodes to present a greater challenge for the inference algorithm (since unification must be invoked more often). The column *nesting* indicates the deepest choice-nestings in the expression, where *dp(n)* indicates that the nesting depth *dp* occurs *n* times. Similarly, the column *cascade* indicates the longest occurrences of cascading choices, as described above. In the last three columns we give the time required to infer the type of a single variant, to infer the types of all variants using the brute-force strategy, and to infer a variational type using *vinfer*. Note that it is impossible to apply the brute-force approach to all but the first of these expressions.

These results demonstrate the feasibility of variational type inference on very large expressions. Our results for type inference are consistent with those for type checking demonstrated by [Thaker et al., 2007]. While usually much larger in size, we would expect real-world software to be considerably less complex. For example, in an analysis of real variational software implemented with the AHEAD framework [Batory et al., 2004], [Kim et al., 2008] found a maximum nesting depth of just 3 and an average depth of 1.5.

Chapter 7: Partial Variational Typing

The type system presented in Chapter 4 possesses an important desirable result: well-typed variational programs generate only well-typed plain programs. This result also holds in the reverse order. If all plain programs of a variational program are well typed, then the variational program itself is well typed. However, what can be said about variational programs that contain ill-typed program variants?

For an illustration, assume we want to type the expression $A\langle \texttt{even}, \texttt{not} \rangle\ \texttt{1}$. Figure 7.1 attempts to show the derivation tree of typing this expression. However, it fails because there is no ϕ such that the type of the function, $A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool} \rangle$ is equivalent to $\texttt{Int} \to \phi$. Thus, no typing rule in Figure 4.3 applies, and we fail to derive any type information for this expression. We observe that the variational type system presented in Chapter 4 can be improved. Although part of $A\langle \texttt{even}, \texttt{not} \rangle\ \texttt{1}$ is ill typed, one of its plain expressions $\texttt{even 1}$ is well typed.

This chapter presents an extension of the type system introduced in Chapter 4 to deliver desirable results for partially well-typed variational programs by introducing an *error type* written as $\bot$. In the case of the expression $A\langle \texttt{even}, \texttt{not} \rangle\ \texttt{1}$, it returns the variational type $A\langle \texttt{Bool}, \bot \rangle$, denoting that the variant $\texttt{even 1}$ has the type $\texttt{Bool}$ and the variant $\texttt{not 1}$ contains a type error.

The addition of error types is a non-trivial extension to the type system and in-

$$\text{VC-App}\ \dfrac{\dfrac{\dfrac{\text{VC-Var}\ \overline{(\texttt{even},\texttt{Int}\to\texttt{Bool})\in\Gamma}}{\Gamma\vdash \texttt{even}:\texttt{Int}\to\texttt{Bool}} \quad \dfrac{\text{VC-Var}\ \overline{(\texttt{not},\texttt{Bool}\to\texttt{Bool})\in\Gamma}}{\Gamma\vdash \texttt{not}:\texttt{Bool}\to\texttt{Bool}}}{\Gamma\vdash A\langle\texttt{even},\texttt{not}\rangle : A\langle\texttt{Int}\to\texttt{Bool},\texttt{Bool}\to\texttt{Bool}\rangle}\ \text{VC-Chc} \\ \vdots \qquad\qquad \Gamma\vdash \texttt{1}:\texttt{Int} \qquad A\langle\texttt{Int}\to\texttt{Bool},\texttt{Bool}\to\texttt{Bool}\rangle \not\equiv \texttt{Int}\to\phi}{\Gamma\vdash A\langle\texttt{even},\texttt{not}\rangle\ \texttt{1}:?}$$

Figure 7.1: Fail to type the expression $A\langle\texttt{even},\texttt{not}\rangle\ \texttt{1}$ under the assumption $\Gamma = \{(\texttt{even},\texttt{Int}\to\texttt{Bool}),(\texttt{not},\texttt{Bool}\to\texttt{Bool})\}$.

ference algorithm presented in Chapter 4 through Chapter 6. In particular, there are many subtle implications for the unification of variational types. In the case Figure 7.1, the location of the error is obvious. However, often there are many possible candidates for the type error, depending on how we infer the surrounding types. The goal is to assign errors such that as few variants as possible are considered ill typed, that is, to find a type that is *most defined*. This goal is in addition to the usual goal of inferring the *most general* type possible. It is not obvious whether these two qualities of types are orthogonal. In this chapter we will show that they are, and we present an inference algorithm that identifies most-defined, most-general types.

The rest of this chapter is organized as follows.

1. In Section 7.1, I introduce the notion of an *error type* so that we can assign a type to any variational expression, although it may contain variants that are ill typed. Section 7.1 also presents the concept of *typing patterns*, which indicate which variants of a variational program are well typed, and an as-

sociated *more-defined* relation for comparing them.

2. In Section 7.2, I present typing rules for dealing with type errors. In particular, I have to deal with unbound variables and type mismatches in function applications. I also present the type preservation theorem (Theorem 11), which formally establishes the relationship between a variational type identified by the type system and the set of types or type errors produced by the brute-force approach.

3. In Section 7.3, I study the properties about the problem of unifying variational types containing type errors. Most significantly, I show that for any unification problem, there is a mapping that produces the most-defined result type (Theorem 12), and that among such mappings, there is a unique mapping that produces the most-general result type (Theorem 13). In the same section, I present a partial variational unification algorithm that produces unifiers that result in most-defined, most-general types. I show that the unification algorithm is sound and complete (Theorems 14 and 15).

4. In Section 7.4, I present an error-tolerant variational type inference algorithm, which is sound and complete (Theorems 16 and 17).

5. Section 7.5 presents experiments that demonstrate that the overhead to support error-tolerant type inference is minor and that the algorithm offers significant performance improvements over the brute-force approach. The evaluation results also reveal an interesting relationship between the dis-

$$
\begin{array}{rcll}
\phi & ::= & \gamma & \textit{Constant Type} \\
 & | & a & \textit{Type Variable} \\
 & | & \phi \to \phi & \textit{Function Type} \\
 & | & D\langle \phi, \phi \rangle & \textit{Choice Type} \\
 & | & \bot & \textit{Error Type} \\
 & | & \top & \textit{OK Type} \\
 & & & \\
\sigma & ::= & \phi & \textit{Partial Type} \\
 & | & \forall \overline{a}.a & \textit{Type Schemas}
\end{array}
$$

Figure 7.2: Partial variational types for VLC.

$$\pi ::= \bot \mid \top \mid D\langle \pi, \pi \rangle$$

Figure 7.3: Syntax of typing patterns.

tribution of type errors in an expression and the time it takes to infer a type for that expression.

7.1 Error Types and Typing Patterns

The type syntax extended with error types is presented in Figure 7.2. Compared to the types for variational programs in Figure 4.1, we have two new type constructs. The *error type*, $\bot$, represents a type error and can appear anywhere in a variational type. We say that a variational type is *partial* if it contains one or more error types and *complete* otherwise. The symbol $\top$ is used to represent an arbitrary complete type that also contains no type variables, that is, a type that is monomorphic and error free. This abstraction is only used in *typing patterns*, which are described in Figure 7.3.

A typing pattern is a variation type consisting only of $\bot$, $\top$, and choice types and is used to describe *which variants* of an expression are well typed and which contain type errors. For example, the typing pattern $\pi = A\langle\top, B\langle\top,\bot\rangle\rangle$ indicates a type error in the variant corresponding to the decision $\{\tilde{A},\tilde{B}\}$, and not in any other variants. A single typing pattern corresponds to an infinite number of partial variational types. Some types corresponding to π include: $A\langle\texttt{Int}, B\langle\texttt{Bool},\bot\rangle\rangle$, $A\langle\texttt{Int},\texttt{Bool}\rangle \to B\langle\texttt{Int}, A\langle\texttt{Bool},\bot\rangle\rangle$, and $A\langle\texttt{Int}, B\langle\texttt{Bool},\bot\rangle \to B\langle\texttt{Int},\bot\rangle\rangle$. In these examples, the constant and function types are irrelevant—all that matters is that selecting $\{\tilde{A},\tilde{B}\}$ produces a type containing errors, and that all other type variants are complete.

Typing patterns are not really types in the traditional sense, but rather an *abstraction* of variation types that indicate where the errors are in the variation space. They are useful for determining which types are *more defined* than others (that is, which contain errors in fewer variants) and play a crucial role in the unification of partial types (see Section 7.3.2). We conflate the representation of variational types and typing patterns because they behave similarly and doing so allows us to reuse a lot of technical machinery. In the rest of this section, we employ typing patterns to define a few operations that will be used throughout this chapter.

We begin by defining a reflexive, transitive relation for determining which typing patterns are *more defined* than others, given in Figure 7.4. All typing patterns are more defined than $\bot$ and less defined than $\top$. Note that one typing pattern is not more defined than another by simply having fewer occurrences of error types.

$$\pi \leq \pi \qquad \pi \leq \bot \qquad \top \leq \pi \qquad \frac{\pi \leq \pi_1 \qquad \pi \leq \pi_2}{\pi \leq D\langle \pi_1, \pi_2 \rangle}$$

$$\frac{\pi_1 \leq \pi \qquad \pi_2 \leq \pi}{D\langle \pi_1, \pi_2 \rangle \leq \pi} \qquad \frac{\pi_1 \leq \pi_1' \qquad \pi_2 \leq \pi_2'}{D\langle \pi_1, \pi_2 \rangle \leq D\langle \pi_1', \pi_2' \rangle}$$

Figure 7.4: The more-defined relation on typing patterns.

For example, the pattern $A\langle B\langle \bot, \top \rangle, B\langle \top, \bot \rangle \rangle$ is trivially more defined than $\bot$.

Next, we consider the *masking* of types with patterns. Given a pattern π and a type ϕ, masking $\pi \lhd \phi$ potentially adds error types to ϕ according to the position of error types in π. We have only three cases for this operation since typing patterns have only three constructs, as shown in Figure 7.3.

$$\begin{aligned} \top \lhd \phi &= \phi \\ \bot \lhd \phi &= \bot \\ D\langle \pi_1, \pi_2 \rangle \lhd \phi &= D\langle \pi_1 \lhd \lfloor \phi \rfloor_D, \pi_2 \lhd \lfloor \phi \rfloor_{\tilde{D}} \rangle \end{aligned}$$

For example, masking type $\texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int} \rangle$ with the typing pattern $A\langle \top, \bot \rangle$ yields the type $A\langle \texttt{Int} \to \texttt{Bool}, \bot \rangle$.

The *intersection* of two typing patterns π and π', written $\pi \otimes \pi'$, is a pattern that is well typed in exactly those variants that are well typed in both π and π'. For example, given patterns $A\langle \top, \bot \rangle$ and $B\langle \bot, \top \rangle$, their intersection is $A\langle B\langle \bot, \top \rangle, \bot \rangle$, which indicates that the only well-typed variant corresponds to the decision $\{\tilde{A}, \tilde{B}\}$. Intersection is just a special case of masking, where the masked type is a typing pattern: $\pi \otimes \pi' = \pi \lhd \pi'$.

The dual of intersection is pattern *union*. The union of two typing patterns π

and π', written $\pi \oplus \pi'$, is well typed in those variants that are well typed in either π or π', or both.

$$\begin{aligned} \top \oplus \pi &= \top \\ \bot \oplus \pi &= \pi \\ D\langle \pi_1, \pi_2 \rangle \oplus \pi &= D\langle \pi_1 \oplus \lfloor \pi \rfloor_D, \pi_2 \oplus \lfloor \pi \rfloor_{\tilde{D}} \rangle \end{aligned}$$

For example, the union of $A\langle \top, \bot \rangle$ and $B\langle \bot, \top \rangle$ is $A\langle \top, B\langle \bot, \top \rangle \rangle$.

Note that the above definitions are all left biased with regard to the nesting order of choices and the structure of the resulting type. This bias can be eliminated through the normalization process described in Chapter 4, which can be applied unaltered to typing patterns.

In the typing process we often need to check whether two types *match*, for example, to check that the domain type of a function matches the type of the argument it is applied to. Rather than a simple boolean response, we can use typing patterns to provide a more precise account, indicating in which variants the types match ($\top$) and in which they do not ($\bot$). In Figure 7.5, we define this matching operation, which has the type $\bowtie : \phi \times \phi \to \pi$. In the definition, we assume both arguments are in normal form. However, this is only assumed for presentation purpose; in our type checker $\bowtie$ is implemented as part of the unification algorithm, and the arguments do not need to be in normal form. For example, matching $\texttt{Int} \to A\langle \texttt{Bool}, \bot \rangle \bowtie B\langle \texttt{Int}, \bot \rangle \to \texttt{Bool}$ produces the typing pattern $A\langle B\langle \top, \bot \rangle, \bot \rangle$. This operation is used in the typing of applications, as we'll see in the next section.

$$
\begin{aligned}
\phi \bowtie \phi &= \top \\
\phi_1 \to \phi_1' \bowtie \phi_2 \to \phi_2' &= \phi_1 \bowtie \phi_2 \otimes \phi_1' \bowtie \phi_2' \\
D\langle \phi_1, \phi_2 \rangle \bowtie D\langle \phi_1', \phi_2' \rangle &= D\langle \phi_1 \bowtie \phi_1', \phi_2 \bowtie \phi_2' \rangle \\
D\langle \phi_1, \phi_2 \rangle \bowtie \phi &= D\langle \phi_1 \bowtie \phi, \phi_2 \bowtie \phi \rangle \\
\phi \bowtie D\langle \phi_1, \phi_2 \rangle &= D\langle \phi_1, \phi_2 \rangle \bowtie \phi \\
\bot \bowtie \phi = \phi \bowtie \bot &= \bot \\
\phi \bowtie \phi' &= \bot \qquad \textit{(otherwise)}
\end{aligned}
$$

Figure 7.5: The operation of matching two types.

7.2 Partial Variational Type Checking

The association of partial variational types with VLC expressions is determined by a set of typing rules, given in Figure 7.6. A VLC typing judgment has the form $\Gamma \vdash^{\bot} e : \phi$, which states that expression e has type ϕ in the context of Γ. We observe that most rules are the same as in Figure 4.3 for typing variational expressions. One specialty here is that we explicitly assign an error type to any unbound variables so that we don't terminate the typing process. The focus here is on the E-APP rule for typing applications, extending it to support partial types. Previously this rule required that the left argument be equivalent to a function type whose argument type is unifiable with the type of the parameter value. In the presence of partial types, we can relax these requirements, introducing error types (rather than failing) when they are not satisfied.

There are essentially two ways that error types can be introduced: (1) if we cannot convert the type of the left argument ϕ_1 into a function type $\phi_2' \to \phi'$, and (2) if ϕ_2' does not match the type of the parameter ϕ_2. The introduction of errors

$$\boxed{\Gamma \vdash^{\perp} e : \phi}$$

$$\text{E-Con}\ \frac{v \text{ is a constant of type } \gamma}{\Gamma \vdash^{\perp} v : \gamma} \qquad \text{E-Var}\ \frac{\Gamma(x) = \forall \overline{a}.\phi_1 \qquad \phi = \{\overline{a \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash^{\perp} x : \phi} \qquad \text{E-Unbound}\ \frac{x \notin dom(\Gamma)}{\Gamma \vdash^{\perp} x : \perp}$$

$$\text{E-Abs}\ \frac{\Gamma, x \mapsto \phi' \vdash^{\perp} e : \phi}{\Gamma \vdash^{\perp} \lambda x.e : \phi' \to \phi} \qquad \text{E-Chc}\ \frac{\Gamma \vdash^{\perp} e_1 : \phi_1 \qquad \Gamma \vdash^{\perp} e_2 : \phi_2}{\Gamma \vdash^{\perp} D\langle e_1, e_2\rangle : D\langle \phi_1, \phi_2\rangle}$$

$$\text{E-App}\ \frac{\Gamma \vdash^{\perp} e_1 : \phi_1 \qquad \Gamma \vdash^{\perp} e_2 : \phi_2 \qquad \phi_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash^{\perp} e_1\ e_2 : \phi}$$

$$\text{E-Let}\ \frac{\Gamma, x \mapsto \phi \vdash^{\perp} e : \phi \qquad \overline{a} = FV(\phi) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall \overline{a}.\phi \vdash^{\perp} e' : \phi'}{\Gamma \vdash^{\perp} \textbf{let}\ x = e\ \textbf{in}\ e' : \phi'}$$

Figure 7.6: Typing rules mapping VLC expressions to partial types.

in the second case is handled by matching the two types using the $\bowtie$ operation to produce a typing pattern π, then masking the result type ϕ with π. In the first case, we employ a helper function $\uparrow$, which lifts a function type to the top level, introducing error types as needed.

$$\begin{aligned} \uparrow(\phi_1 \to \phi_2) &= \phi_1 \to \phi_2 \\ \uparrow(D\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle) &= D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_1', \phi_2' \rangle \\ \uparrow(D\langle \phi_1, \phi_2 \rangle) &= \uparrow(D\langle \uparrow(\phi_1), \uparrow(\phi_2) \rangle) \\ \uparrow(\phi) &= \perp \to \perp \qquad \textit{(otherwise)} \end{aligned}$$

For example, $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Int} \rangle) = A\langle \texttt{Int}, \texttt{Bool} \rangle \to A\langle \texttt{Bool}, \texttt{Int} \rangle$, while $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \rangle)$ must introduce error types to lift the function type to the top: $A\langle \texttt{Int}, \perp \rangle \to A\langle \texttt{Bool}, \perp \rangle$.

To illustrate the typing of an application, consider the expression $e_1\ e_2$, where $e_1 : A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle$ and $e_2 : \texttt{Int}$. Applying $\uparrow$ to the type of e_1 and simplifying the result type yields the type $A\langle \texttt{Int}, \texttt{Bool}\rangle \to \texttt{Bool}$. Matching $A\langle \texttt{Int}, \texttt{Bool}\rangle \bowtie \texttt{Int}$ produces the typing pattern $A\langle \top, \bot\rangle$, which we use to mask the result, $A\langle \top, \bot\rangle \lhd \texttt{Bool}$, producing the type of the application: $A\langle \texttt{Bool}, \bot\rangle$.

The previous VC-APP rule emerges as a special case of E-APP. When e_1 is a function type whose argument type matches the type of e_2, then matching returns $\top$ and masking doesn't alter the return type.

The correspondence between variational types and VLC expressions is established inductively through the process of selection. Given that $e : \phi$, if e is plain, then ϕ is a plain type or $\bot$. If e is not plain, then we can select a tag from e to produce $e' : \phi'$, and ϕ' can be obtained by a corresponding selection from ϕ. The inductive step is captured in the following lemma.

Lemma 12 (Variation elimination)

$\Gamma \vdash^{\bot} e : \phi \implies \forall\ s : \Gamma \vdash^{\bot} \lfloor e \rfloor_s : \lfloor \phi \rfloor_s$

PROOF. The proof for this lemma is very similar to that for Lemma 5 and is omitted here. □

By induction it follows that a sequence of selections that produces a plain expression can be used to select a corresponding plain or error type. This results in the following theorem.

Theorem 11 (Type preservation) *If* $\Gamma \vdash^{\bot} e : \phi$ *and* $(\delta, e') \in [\![e]\!]$, *then* $\Gamma \vdash^{\bot} e' : \phi'$ *where* $(\delta, \phi') \in [\![\phi]\!]$.

This theorem demonstrates the soundness of the type systems since it establishes that from the type of a variational program we can obtain the type of each program variant it contains. We had similar type preservation results in Chapter 4, but they applied to only well-typed variational programs. The results here are stronger since they apply to *any* variational programs.

7.3 Partial Variational Unification

Having extended the type system to work with and produce partial types, we now turn to the more challenging problem of inferring variational types containing type errors. By far the most difficult piece is partial type unification. In Section 7.3.1 we will describe the specific challenges posed. In particular, the unification algorithm must yield unifiers that produce types that are both most general *and* most defined, two qualities that are not obviously orthogonal. In Section 7.3.2 we show that such unifiers exist, and in Section 7.3.3 we present an algorithm for computing unifiers.

7.3.1 Reconciling Type Partiality and Generality

To support partial type inference, we must extend variational type unification to produce and extend mappings containing error types, and to identify mappings that are somehow best.

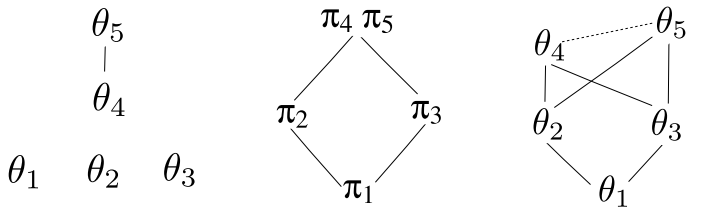
As a running example, consider the application $e\ e'$ where $e : \phi =$

i	θ_i	$\phi_i = \theta_i(\phi)$	$\phi'_i = \theta_i(\phi')$
1	$\{a \mapsto \texttt{Ch}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{Ch}$	$B\langle \texttt{In}, \texttt{Ch}\rangle$
2	$\{a \mapsto \texttt{In}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{In}$	$B\langle \texttt{In}, \texttt{In}\rangle$
3	$\{a \mapsto \texttt{Bo}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{Bo}$	$B\langle \texttt{In}, \texttt{Bo}\rangle$
4	$\{a \mapsto A\langle \texttt{In}, \texttt{Bo}\rangle\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to A\langle \texttt{In}, \texttt{Bo}\rangle$	$B\langle \texttt{In}, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$
5	$\{a \mapsto B\langle b, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to B\langle b, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$	$B\langle \texttt{In}, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$
i	θ_i	$\pi_i = arg(\phi_i) \bowtie \phi'_i$	$R_i = \pi_i \lhd res(\phi_i)$
1	$\{a \mapsto \texttt{Ch}\}$	$A\langle B\langle \top, \bot\rangle, \bot\rangle$	$A\langle B\langle \texttt{Ch}, \bot\rangle, \bot\rangle$
2	$\{a \mapsto \texttt{In}\}$	$A\langle \top, \bot\rangle$	$A\langle \texttt{In}, \bot\rangle$
3	$\{a \mapsto \texttt{Bo}\}$	$A\langle B\langle \top, \bot\rangle, B\langle \bot, \top\rangle\rangle$	$A\langle B\langle \texttt{Bo}, \bot\rangle, B\langle \bot, \texttt{Bo}\rangle\rangle$
4	$\{a \mapsto A\langle \texttt{In}, \texttt{Bo}\rangle\}$	$A\langle \top, B\langle \bot, \top\rangle\rangle$	$A\langle \texttt{In}, B\langle \bot, \texttt{Bo}\rangle\rangle$
5	$\{a \mapsto B\langle b, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle\}$	$A\langle \top, B\langle \bot, \top\rangle\rangle$	$A\langle B\langle b, \texttt{In}\rangle, B\langle \bot, \texttt{Bo}\rangle\rangle$

Figure 7.7: Some mappings for $\phi = A\langle \texttt{Int}, \texttt{Bool}\rangle \to a$ and $\phi' = B\langle \texttt{Int}, a\rangle$, with the typing patterns and result types (R_i) they produce.

$A\langle \texttt{Int}, \texttt{Bool}\rangle \to a$ and $e' : \phi' = B\langle \texttt{Int}, a\rangle$. Usually we would find the most general unifier (mgu) for the problem $A\langle \texttt{Int}, \texttt{Bool}\rangle \equiv^? B\langle \texttt{Int}, a\rangle$, but in this case the two types are not unifiable since there is a type error in the $\{\tilde{A}, B\}$ variant. So what should we map a to? The mapping we choose should be *most general* in the usual sense, but it should also be *most defined*, yielding types with type errors in as few variants as possible. In this subsection we will explore the interaction of these two properties.

In Figure 7.7 we list several mappings we might choose to partially unify ϕ and ϕ' in our example. In the table, the type constants `Bool`, `Char`, and `Int` are shortened for space reasons. Each mapping is identified by a θ_i, for example, $\theta_2 = \{a \mapsto \texttt{Int}\}$. We also give the result of applying each mapping to each of the two types as ϕ_i and ϕ'_i, the typing pattern π_i that results from matching the argument type of ϕ_i to ϕ'_i, and the result type generated by masking the result type of ϕ_i



(a) θ_i by $\sqsubseteq$ (b) π_i by $\trianglelefteq$ (c) θ_i by $\trianglelefteq, \sqsubseteq$

Figure 7.8: Orderings among patterns, result types, and mappings.

with π_i. Note that we use the functions *arg* and *res* to access, respectively, the argument and result types of a function type.

Figure 7.8 visualizes the more-general and more-defined relationships among mappings and typing patterns. The relations are defined for elements connected by lines, and the element higher in the graph is considered more general or more defined.

The first thing to note is that the standard more-general relation, $\sqsubseteq$, is not very helpful in selecting a mapping. A mapping θ is more general than θ', written $\theta \sqsubseteq \theta'$ if $\exists\theta''$ such that $\theta' = \theta'' \circ \theta$. But this relationship is only defined on one pair of our five mappings: $\theta_5 \sqsubseteq \theta_4$ (since $\{b \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\} \circ \theta_5 = \theta_4$). Since we are not restricted to mappings that are valid unifiers, there are many more possibilities, and many will not be ordered by the more-general relation.

More useful is the more-defined relation (see Section 7.1) on the match-produced typing patterns, for which many relationships are defined, as seen in Figure 7.8b. Using this metric, we can rule out mappings θ_1, θ_2, and θ_3 because they will produce types with errors in more variants than the mappings θ_4 and θ_5. The problem is that θ_4 and θ_5 produce the same pattern.

The solution, of course, is to use both metrics together, as demonstrated in Figure 7.8c. The solid lines between mappings correspond to more-defined relations between the generated typing patterns, and the dotted line corresponds to the more-general relation between the mappings directly. This reveals θ_5 as the most-defined, most-general mapping. At this point it is not clear whether this convergence was a quirk of our example or whether these properties will always converge in this way. In the next section we will tackle the general case, and show that a most-defined, most-general mapping always exists.

7.3.2 Most-General Partial Unifiers

In Section 7.3.1, we have illustrated how unification with partial types requires the integration of two partial orderings of types, $\preceq$ and $\sqsubseteq$. In this section, we introduce the necessary machinery that enables unification to deal with this situation in general and produce most general partial unifiers.

In the following we consider a general unification problem of the form $U = \phi_L \equiv^? \phi_R$. For a given mapping θ, we write $U :: \theta$ for the typing pattern $\theta(\phi_L) \bowtie \theta(\phi_R)$ that results from θ and U. When we say that π is a typing pattern for U, we mean that there is some θ such that $\pi = U :: \theta$. With $FV(U)$ we refer to all type variables in U, and we use $dom(\theta)$ to denote the domain of θ. We use $\|\theta\|_U$ to normalize θ with respect to the variables in U, that is, $\|\theta\|_U$ is obtained from θ by renaming type variables such that $dom(\|\theta\|_U) = FV(U)$.

Finally, we extend selection to apply to unification problems and mappings,

that is, $\lfloor U \rfloor_s = \lfloor \phi_L \rfloor_s \equiv^? \lfloor \phi_R \rfloor_s$ and $\lfloor \theta \rfloor_s = \{(a, \lfloor \phi \rfloor_s) \mid (a, \phi) \in \theta\}$. We write $\theta|_V$ for the restriction of θ by a set of variables V, which is defined as $\theta|_V = \{(a, \theta(a)) \mid a \in V\}$.

The first three lemmas state that selection extends in a homomorphic way across several operations.

Lemma 13 $\lfloor \phi_L \bowtie \phi_R \rfloor_s = \lfloor \phi_L \rfloor_s \bowtie \lfloor \phi_R \rfloor_s$

The proofs for this and the following lemmas are very simple and similar to that for Lemma 2 and are omitted here.

Lemma 14
$$\lfloor \phi_L \oplus \phi_R \rfloor_s = \lfloor \phi_L \rfloor_s \oplus \lfloor \phi_R \rfloor_s$$
$$\lfloor \phi_L \otimes \phi_R \rfloor_s = \lfloor \phi_L \rfloor_s \otimes \lfloor \phi_R \rfloor_s$$
$$\lfloor \pi \lhd \phi \rfloor_s = \lfloor \pi \rfloor_s \lhd \lfloor \phi \rfloor_s$$
$$\lfloor \phi_L \multimap \phi_R \rfloor_s = \lfloor \phi_L \rfloor_s \multimap \lfloor \phi_R \rfloor_s$$

We also have a similar result for type substitution.

Lemma 15 $\lfloor \theta(\phi) \rfloor_s = \lfloor \theta \rfloor_s (\lfloor \phi \rfloor_s)$

The next lemma says that the computation of typing patterns can be decomposed by using selection.

Lemma 16 $\lfloor U :: \theta \rfloor_s = \lfloor U :: \lfloor \theta \rfloor_s \rfloor_s = \lfloor U \rfloor_s :: \lfloor \theta \rfloor_s$

PROOF. The proof for the first part is as follows. Let $\pi = U :: \theta$ and $\pi' = U :: \lfloor \theta \rfloor_s$,

then

$$\begin{aligned}
\lfloor \pi \rfloor_s &= \lfloor \theta(\phi_L) \bowtie \theta(\phi_R) \rfloor_s \\
&= \lfloor \theta(\phi_L) \rfloor_s \bowtie \lfloor \theta(\phi_R) \rfloor_s && \text{by Lemma 13} \\
&= \lfloor \theta \rfloor_s(\lfloor \phi_L \rfloor_s) \bowtie \lfloor \theta \rfloor_s(\lfloor \phi_R \rfloor_s) && \text{by Lemma 15} \\
\lfloor \pi' \rfloor_s &= \lfloor \lfloor \theta \rfloor_s(\phi_L) \bowtie \lfloor \theta \rfloor_s(\phi_R) \rfloor_s \\
&= \lfloor \lfloor \theta \rfloor_s(\phi_L) \rfloor_s \bowtie \lfloor \lfloor \theta \rfloor_s(\phi_R) \rfloor_s && \text{by Lemma 13} \\
&= \lfloor \lfloor \theta \rfloor_s \rfloor_s(\lfloor \phi_L \rfloor_s) \bowtie \lfloor \lfloor \theta \rfloor_s \rfloor_s(\lfloor \phi_R \rfloor_s) && \text{by Lemma 15} \\
&= \lfloor \theta \rfloor_s(\lfloor \phi_L \rfloor_s) \bowtie \lfloor \theta \rfloor_s(\lfloor \phi_R \rfloor_s)
\end{aligned}$$

The proof for the second part is analogous. □

Lemma 17 (Typing patterns have a join) *If π_1 and π_2 are typing patterns for U, then so is $\pi_1 \oplus \pi_2$.*

PROOF. Assume θ_1 and θ_2 are the mappings such that $\pi_1 = U :: \theta_1$ and $\pi_2 = U :: \theta_2$. The proof consists of several cases. For each case, we construct a mapping θ_3 such that $U :: \theta_3 = \pi_1 \oplus \pi_2$, which we denote as π_3. We show the proof for the case where $\pi_1 = D\langle \pi_{11}, \pi_{12} \rangle$ and $\pi_2 = D\langle \pi_{21}, \pi_{22} \rangle$ and there is no $\leq$ relation between π_1 and π_2. The proofs for other cases are simpler or can be transformed into this case. We assume that θ_1 and θ_2 are already normalized with respect to U. We can consider several cases.

First, if we assume $\pi_{21} \leq \pi_{11}$ and $\pi_{12} \leq \pi_{22}$, we let

$$\theta_3 = \{(a, D\langle \lfloor \theta_2(a) \rfloor_D, \lfloor \theta_1(a) \rfloor_{\tilde{D}} \rangle) \mid a \in FV(U)\}$$

for which we observe the following.

$$\begin{aligned}
U :: \theta_3 &= D\langle \lfloor U :: \theta_3 \rfloor_D, \lfloor U :: \theta_3 \rfloor_{\tilde{D}} \rangle \\
&= D\langle \lfloor U \rfloor_D :: \lfloor \theta_3 \rfloor_D, \lfloor U \rfloor_{\tilde{D}} :: \lfloor \theta_3 \rfloor_{\tilde{D}} \rangle && \text{Lemma 16} \\
&= D\langle \lfloor U \rfloor_D :: \lfloor \theta_1 \rfloor_D, \lfloor U \rfloor_{\tilde{D}} :: \lfloor \theta_2 \rfloor_{\tilde{D}} \rangle && \text{construction} \\
&= D\langle \lfloor U :: \theta_1 \rfloor_D, \lfloor U :: \theta_2 \rfloor_{\tilde{D}} \rangle && \text{Lemma 16} \\
&= D\langle \pi_{21}, \pi_{12} \rangle \\
&= \pi_1 \oplus \pi_2 && \text{def. of } \oplus
\end{aligned}$$

Second, the case for $\pi_{11} \trianglelefteq \pi_{21}$ and $\pi_{22} \trianglelefteq \pi_{12}$ is analogous.

Third, if there is no $\trianglelefteq$ relation between π_{21} and π_{11} or π_{12} and π_{22}, we let $U_1 = \lfloor U \rfloor_D$, $U_2 = \lfloor U \rfloor_{\tilde{D}}$, $\theta_{11} = \theta_1|_{FV(U_1)}$, $\theta_{12} = \theta_1|_{FV(U_2)}$, $\theta_{21} = \theta_2|_{FV(U_1)}$ and $\theta_{22} = \theta_2|_{FV(U_2)}$. By induction, we can construct a mapping θ_{31} from θ_{11} and θ_{21} for U_1 such that $U_1 :: \theta_{31} = \pi_{11} \oplus \pi_{21}$. Likewise, we can construct a mapping θ_{32} from θ_{12} and θ_{22} for U_2 such that $U_2 :: \theta_{32} = \pi_{12} \oplus \pi_{22}$. We can now build θ_3 based on θ_{31} and θ_{32} as follows. For each type variable $a \in FV(U)$ we define θ_3 as follows.

$$\theta_3(a) = \begin{cases} D\langle \theta_{31}(a), \theta_{32}(a) \rangle & \text{if } a \in FV(U_1) \wedge a \in FV(U_2) \\ \theta_{31}(a) & \text{if } a \in FV(U_1) \\ \theta_{32}(a) & \text{if } a \in FV(U_2) \end{cases}$$

Proving that $U :: \theta_3 = D\langle \pi_{31}, \pi_{32} \rangle = D\langle \pi_{11}, \pi_{12} \rangle \oplus D\langle \pi_{21}, \pi_{22} \rangle$ is similar to the proof for the previous case. □

Combining this lemma with the rule $\top \trianglelefteq \pi$ we can conclude that for any unification problem, there is an upper-bound typing pattern, which we call the *principal typing pattern*.

Theorem 12 (Existence of principal typing patterns) *For every unification*

problem U there is a mapping θ with $\pi = U :: \theta$, such that $\pi \leq \pi'$ for any other mapping θ' with $\pi' = U :: \theta'$.

We call a mapping that leads to the principal typing pattern a *partial unifier* and use η to denote partial unifiers. We call mappings that are not partial unifiers "non-unifiers" for short. Based on these definitions, the first example in Section 7.3.1 has the principal typing pattern π_4 and partial unifiers θ_4 and θ_5.

Theorem 12 only shows the existence of partial unifiers, but does not say anything about how many partial unifiers exist and how they are possibly related. It turns out that partial unifiers can be compared with respect to their generality and for each unification problem there is a *most general partial unifier* (mgpu) of which all other partial unifiers are instances.

Theorem 13 (Partial unification is unitary) *For every unification problem U there is one partial unifier η of such that any other partial unifier η' for U is an instance of it, that is, $\eta \sqsubseteq \eta'$.*

The proof strategy is similar to that for Theorem 12, although more complex. Given any two partial unifiers, we can construct a new partial unifier that is more general than the old ones.

Figure 7.9 summarizes the notions presented in this section, where we show the mappings (θ), typing patterns (π), partial unifiers (η) and their relation for the unification problem $A\langle \texttt{Int}, a\rangle \equiv^? B\langle \texttt{Bool}, b\rangle$. We use $+_\pi$ to denote the composition of mappings defined by Lemma 17 and use $+_\eta$ to denote the composition of partial unifiers to get a more general partial unifiers. There are three layers of mappings

partitioned by their corresponding typing patterns. The layers that are higher have more defined typing patterns. For the typing patterns that are less defined than the principal typing pattern, only one mapping is shown for each typing pattern.

The highest layer depicts the relations between partial unifiers, where we use dashed line to denote the more general relation and use solid lines to denote the composition of partial unifiers to get a more general unifier. For example, from η_2 and η_3 we get η_7, which can also be reached by composing η_3 and η_4. By composing η_7 and η_6, we get η_8, the mgpu of the unification problem. Note that for space reasons the figure doesn't include all partial unifiers. Also, for simplicity, some relations are omitted. For example, the more general relation between η_3 and η_6 and the $+_\eta$ relation between η_4 and η_5 to get η_6 are omitted.

7.3.3 A Partial Variational Unification Algorithm

In Section 7.3.2 we showed that for each partial unification problem, there is a unique mgpu that produces the corresponding principal typing pattern. In this section, we show how to compute each of these by extending the process described in Section 5.2. We do this first by example, then give the algorithm directly.

Consider the unification problem $A\langle \texttt{Int}, a\rangle \equiv^? B\langle \texttt{Bool}, b\rangle$. We begin, as described in Section 5.2, by transforming this into the corresponding qualified unification problem shown at the top of Figure 7.10. Since the top-level choice names don't match, we choose a type variable and apply the split-hoist strategy (first two

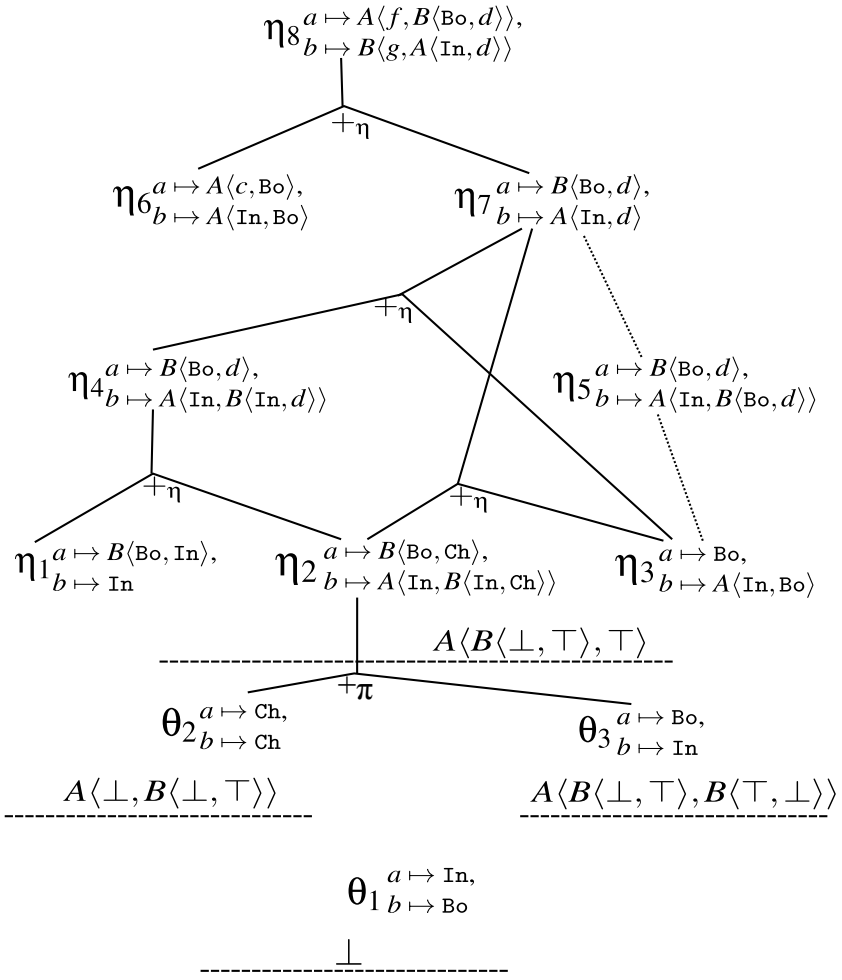


Figure 7.9: Typing pattern, mappings, partial unifiers and their relations for the unification problem $A\langle \mathtt{Int}, a\rangle \equiv^? B\langle \mathtt{Bool}, b\rangle$.

steps) in order to decompose by alternatives (third step). This gives us the two subproblems at the fourth level from the top. When a plain type is unified with a choice type, we can decompose it by unifying the plain type with each alternative. This is demonstrated in the left branch, which yields two smaller subproblems, one of which, $\mathtt{Int} \equiv^? \mathtt{Bool}$, reveals a type error.

$$A\langle \texttt{Int}, a_{\tilde{A}} \rangle \equiv^? B\langle \texttt{Bool}, \underline{b_{\tilde{B}}} \rangle$$
$$\downarrow_{split}$$
$$A\langle \texttt{Int}, a_{\tilde{A}} \rangle \equiv^? \underline{B\langle \texttt{Bool}, A\langle b_{A\tilde{B}}, b_{\tilde{A}\tilde{B}} \rangle \rangle}$$
$$\downarrow_{hoist}$$
$$A\langle \texttt{Int}, a_{\tilde{A}} \rangle \equiv^? A\langle B\langle \texttt{Bool}, b_{A\tilde{B}} \rangle, B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}} \rangle \rangle$$

$$\frac{\texttt{Int} \equiv^? B\langle \texttt{Bool}, b_{A\tilde{B}} \rangle}{* \texttt{Int} \equiv^? \texttt{Bool} *} \qquad a_{\tilde{A}} \equiv^? B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}} \rangle$$

$$\texttt{Int} \equiv^? b_{A\tilde{B}}$$

Figure 7.10: Qualified unification resulting in a type error.

This decomposition contains all of the information needed to construct both the mgpu and the principal typing pattern. We construct the mgpu by composing the mappings generated at the end of every successful branch of the unification process. In this case, there were two successful branches, giving the following mgpu.

$$\{a_{\tilde{A}} \mapsto B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}} \rangle, b_{A\tilde{B}} \mapsto \texttt{Int}\}$$

We construct the principal typing pattern by observing which branches of the decomposition fail and succeed. In this case, the branch corresponding to the first alternative in both A and B failed, yielding the principal error pattern $A\langle B\langle \bot, \top \rangle, \top \rangle$.

As the final step, we use completion to produce the solution to the original (unqualified) unification problem.

$$\{a \mapsto A\langle c, B\langle \texttt{Bool}, d \rangle \rangle, b \mapsto B\langle f, A\langle \texttt{Int}, d \rangle \rangle\}$$

Figure 7.11 gives the partial unification algorithm. It accepts a qualified unification problem $\phi_L \equiv^? \phi_R$ and returns a principal typing pattern π and a mgpu η. We show only the cases that differ significantly from the qualified unification algorithm presented in Section 5.2.

The algorithm relies on several helper functions. The functions $chcs(\phi)$ and *splittable* are introduced in Section 5.2. The function $sdims(v_q, \phi)$ returns the set of dimension names not present in q but present in the qualifications of type variables that are more *specific* than v_q. We say that u_p is more specific than v_q if $u = v$ and p can be written as qp' for some nonempty p'. For example, $sdims(a_A, a_{A\tilde{B}} \to \texttt{Int}) = \{B\}$.

We will work through the cases of the *punify* algorithm, from top to bottom. In the body of the algorithm and in these descriptions, ϕ_L and ϕ_R are used to refer to the first and second arguments of *punify*, respectively. We first consider a couple of base cases. Attempting to unify any type and an error type yields an empty mapping and the fully undefined typing pattern $\bot$. This defines the propagation of errors. When unifying two plain types, we defer to Robinson's unification algorithm $\mathcal{U}_R$ introduced in Figure 2.2. If it succeeds, we return the unifier and the fully defined typing pattern $\top$. If it fails, we return the empty mapping and $\bot$.

When unifying a *ground plain type* g (a type that does not contain choice types or type variables) with a choice type, we just unify g with both alternatives. This is seen in the second decomposition in Figure 7.10. The first decomposition is by alternatives, which is performed when unifying two choices in the same dimension; this is captured in the fourth case of *punify*. Note that we do

$$
\begin{aligned}
&\mathit{punify} : \phi \times \phi \to \pi \times \eta \\
&\mathit{punify}(\bot, \phi) = (\bot, \varnothing) \\
&\mathit{punify}(p, p') \\
&\quad \mid \mathscr{U}_R(p, p') = \bot = (\bot, \varnothing) \\
&\quad \mid \text{otherwise} = (\top, \mathscr{U}_R(p, p')) \\
&\mathit{punify}(g, D\langle \phi_1, \phi_2 \rangle) = \mathit{punify}(D\langle g, g \rangle, D\langle \phi_1, \phi_2 \rangle) \\
&\mathit{punify}(D\langle \phi_1, \phi_2 \rangle, D\langle \phi_1', \phi_2' \rangle) = \\
&\quad (\pi_1, \eta_1) \leftarrow \mathit{punify}(\phi_1, \phi_1') \\
&\quad (\pi_2, \eta_2) \leftarrow \mathit{punify}(\phi_2, \phi_2') \\
&\quad \text{return } (D\langle \pi_1, \pi_2 \rangle, \eta_1 \circ \eta_2) \\
&\mathit{punify}(D_1\langle \phi_1, \phi_2 \rangle, D_2\langle \phi_1', \phi_2' \rangle) \\
&\quad \mid D_2 \notin \mathit{chcs}(\phi_L) \wedge \mathit{splittable}(\phi_L) = \varnothing \wedge \\
&\qquad D_1 \notin \mathit{chcs}(\phi_R) \wedge \mathit{splittable}(\phi_R) = \varnothing \\
&\quad = \mathit{punify}(\phi_L, D_1\langle \phi_R, \phi_R \rangle) \\
&\mathit{punify}(v_q, \phi_1' \to \phi_2') \\
&\quad \mid v_q \in \mathit{FV}(\phi_R) = (\bot, \varnothing) \\
&\quad \mid D \in \mathit{sdims}(v_q, \phi_R) = \mathit{punify}(D\langle v_{Dq}, v_{\tilde{D}q} \rangle, \phi_R) \\
&\quad \mid \text{otherwise} = (\top, \{v_q \mapsto \phi_R\}) \\
&\mathit{punify}(\phi_1 \to \phi_2, \phi_1' \to \phi_2') = \\
&\quad (\pi_1, \eta_1) \leftarrow \mathit{punify}(\phi_1, \phi_1') \\
&\quad (\pi_2, \eta_2) \leftarrow \mathit{punify}(\eta_1(\phi_2), \eta_1(\phi_2')) \\
&\quad \pi \leftarrow \pi_1 \otimes \pi_2 \\
&\quad \text{return } (\pi, \eta_1 \circ \eta_2)
\end{aligned}
$$

Figure 7.11: Partial unification algorithm.

not need to apply the mapping η_1 to ϕ_2 and ϕ_2', as we might expect, because $(\mathit{FV}(\phi_1) \cup \mathit{FV}(\phi_1')) \cap (\mathit{FV}(\phi_2) \cup \mathit{FV}(\phi_2')) = \varnothing$ due to type variable qualification. We then compose the corresponding unifiers and combine the error patterns with a choice type.

The fifth case considers the unification of two choice types in different dimen-

sions with no splittable type variables. This is not fully unifiable and so would usually represent failure. However, with partial unification we can proceed by attempting to unify all combinations of alternatives in order to locate the variants that contain errors. For example, $A\langle \texttt{Int}, \texttt{Bool} \rangle \equiv^? B\langle \texttt{Int}, \texttt{Bool} \rangle$ produces the typing pattern $A\langle B\langle \top, \bot \rangle, B\langle \bot, \top \rangle \rangle$. We reuse our existing machinery by duplicating ϕ_R and putting it in a choice type that will be decomposed by alternatives in the recursive execution of *punify*.

Although I don't show all the cases of unifying a qualified type variable against other types, I do show the trickiest case of unifying a type variable with a function type in the sixth case in Figure 7.11. There are three sub-cases to consider: (1) If v_q occurs in ϕ_R, the unification fails. (2) If v_q does not occur in ϕ_R but a more specific type variable v_{qr} does, then some variants may still be well typed. So, we create a new unification problem by adding a dimension D from r to the qualification of v_q, then splitting the new variable v_{Dq} in the D dimension. (3) Finally, if v does not appear in any form in ϕ_R, then we simply map v_q to ϕ_R. Note that the decomposition of the unification problem is such that if there is any v_p in ϕ_R, then either $v_p = v_q$ or v_p is more specific than v_q.

Finally, we consider the unification of two function types. We unify the corresponding argument types and result types and compose the mappings. The resulting typing pattern is the intersection of the patterns of the two subproblems since the result will be well typed only if both the argument and result types agree.

We conclude by presenting some important properties of the unification algorithm. The first result is that the partial unification algorithm is terminating

through decomposition that eventually results in either the propagation of type errors, or calls to the $\mathscr{U}_R$ algorithm, which is terminating. There are two cases that do not decompose, but rather grow the size of the types being unified, and so pose a threat to termination. The first is the splitting of type variables. The second is the fifth case shown in Figure 7.11. Both of these cases introduce a new choice type and duplicate one of their arguments. These cases do not prevent termination, however, for two reasons. First, both cases are followed immediately by a decomposition that produces two subproblems smaller than the original problem. Second, the number of new choice types that can be introduced is bounded by the overall number of dimensions in the unification problem. This follows from the property of choice domination and the fact that we eliminate a dimension from consideration with each decomposition by alternatives.

In Section 5.4, we did an in-depth time complexity analysis of the variational unification algorithm. We showed that if the size of ϕ_L and ϕ_R are l and r respectively, then the time complexity of variational unification is $O(lr(l+r))$. Since the computation of typing patterns in the unification algorithm does not exceed the time for computing partial unifiers, the run-time complexity is still $O(lr(l+r))$ for the partial unification algorithm.

The partial unification algorithm is also sound and complete. These facts are expressed in the following theorems. We use *punify*$'$ to refer to the entire three-part unification process described in Section 5.3 (qualification, qualified unification, completion).

Theorem 14 (Partial unification is sound) *Given the unification problem*

$\phi_1 \equiv^? \phi_2$, *if* $punify'(\phi_1, \phi_2) = (\pi, \eta)$, *then* $\eta(\phi_1) \bowtie \eta(\phi_2) = \pi$.

Theorem 15 (Partial unification is complete, most defined, and most general) *Given the unification problem* $\phi_1 \equiv^? \phi_2$, *if* $\theta(\phi_1) \bowtie \theta(\phi_2) = \pi$, *then* $punify'(\phi_1, \phi_2) = (\pi', \eta)$ *such that* $\pi' \leq \pi$ *and if* $\pi' \equiv \pi$ *then there exists some* θ' *such that* $\theta = \theta' \circ \eta$.

The proofs for these two theorems are omitted here since they are very similar to those for Theorems 6 and 7, respectively.

7.4 Partial Variational Type Inference

Although the partial unification algorithm is quite complicated, the inference algorithm itself is simple. We define it as an extension of algorithm $\mathcal{W}$ (Figure 2.3) and show the most interesting case below.

$$
\begin{array}{l}
pinfer : \Gamma \times e \to \eta \times \phi \\
pinfer(\Gamma, e_1\ e_2) = \\
\quad (\eta_1, \phi_1) \leftarrow pinfer(\Gamma, e_1) \\
\quad (\eta_2, \phi_2) \leftarrow pinfer(\eta_1(\Gamma), e_2) \\
\quad (\pi, \eta) \leftarrow punify'(\eta_2(\phi_1), \eta_2(\phi_2) \to a) \qquad \text{where } a \text{ fresh} \\
\quad \phi \leftarrow \pi \lhd \eta(a) \\
\quad \text{return } (\eta \circ \eta_2 \circ \eta_1, \phi)
\end{array}
$$

The algorithm takes two arguments: a type environment and an expression. It returns a partial unifier and the inferred partial type. Traditionally, inferring the result of a function application consists of four steps: (1) infer the type of the function, (2) infer the type of the argument, (3) unify the argument type of the function

with the type of the argument, and (4) instantiate the result type of the function with the returned unifier. Our algorithm adds just one more step: we must mask the result type according to the typing pattern returned by partial unification, in order to introduce error types for the cases where traditional unification would fail.

The remaining cases can be derived from the typing rules in Section 7.2. Variables and abstractions are treated as in $\mathcal{W}$. For a choice we infer the type of each alternative and build a corresponding choice type.

The following theorems state that our type inference algorithm is sound and complete and has the principal typing property. In the following, the symbol $\preceq$ represents a more-defined, more-general relation on variational types. That is, $\phi' \preceq \phi$ means that for every corresponding pair of (plain) variants V' and V from ϕ' and ϕ, respectively, either $V' \sqsubseteq V$ or $V = \bot$.

Theorem 16 (Partial Type inference is sound)

If $\mathit{pinfer}(\Gamma, e) = (\eta, \phi)$*, then* $\eta(\Gamma) \vdash^{\bot} e : \phi$.

Theorem 17 (Partial Type inference is complete and principal)

If $\eta(\Gamma) \vdash^{\bot} e : \phi$*, then* $\mathit{pinfer}(\Gamma, e) = (\eta', \phi')$ *such that* $\eta = \theta \circ \eta'$ *for some* θ *and* $\phi' \preceq \phi$.

Due to the close relation of *pinfer* and *vinfer*, the proofs for these theorems closely resemble those for Theorems 9 and 10. Thus, we omit the proofs for them here.

These results mean that, for any syntactically correct VLC expression, we can infer the most general type, containing type errors in as few variants as possible, all without type annotations.

7.5 Performance Evaluation

In this section we empirically evaluate the efficiency of partial type inference in a variety of ways. To do this, we have developed a prototype in Haskell that implements the contents of this chapter. The prototype consists of three parts: a normalizer for variational types, the equational partial unification algorithm described in Section 7.3.3, and the type inference algorithm described in Section 7.4.

In the first experiment, we measure the additional cost of the extensions described in this chapter, relative to Chapter 6. To measure this overhead effectively, we intentionally induced our worst-case performance through the *cascading choice* problem. Cascading choices are long sequences of applications, where each expression is a choice in a different dimension. If the types of all alternatives are different, no solution can perform better than the brute-force strategy. The overhead of our prototype on such examples (all well-typed, with between 14 and 21 dimensions) was about 30% of the running time of the non-error-tolerant prototype described in Chapter 6.2.

In the second experiment, we study how the distribution of errors in an expression affects the efficiency of partial type inference. The graph in Figure 7.12 shows the running time of the prototype on a cascading choice problem with 21 dimensions, seeded with errors. The horizontal axis indicates the percentage of variants that were seeded with type errors, and the different lines represent different distributions of these errors. Errors can be *spread* evenly throughout the expression, *clustered* together, distributed *randomly*, or introduced at the *end* of

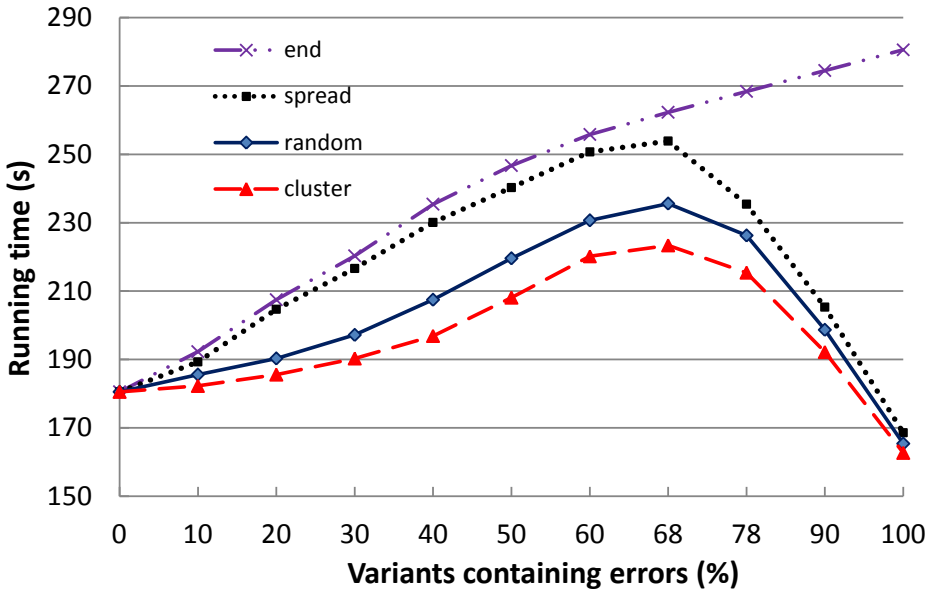


Figure 7.12: Running time of prototype by error distribution.

the expression. An interesting phenomenon is that, while the running time at first increases as we introduce errors (due to the costs of maintaining and applying error patterns), in three of the four curves the running time decreases sharply as the error density increases. This is because additional errors introduce opportunities for reduction through choice idempotency ($D\langle\bot,\bot\rangle = \bot$) that are usually denied in cascading choice expressions. As expected, this feature is most pronounced when errors are clustered and least pronounced when they are spread evenly. When errors are introduced at the end of the expression, this opportunity never arises since all the work has already been done.

Finally, in the third experiment, we demonstrate the efficiency and effectiveness of partial type inference in finding type errors, relative to the brute-force approach (implemented as a prototype in the same way as our own). The results

size	dims	variants	errors	spread		clustered	
				brute (%)	vlc (s)	brute (%)	vlc (s)
702	22	2^{16}	100	100♣	0.62	100♡	0.57
3719	22	2^{16}	100	14.90	1.09	4.10	1.02
976	24	2^{17}	200	100♢	0.71	100♠	0.68
5327	24	2^{17}	200	0.50	2.26	39.85	2.17
8412	24	2^{17}	200	0.05	3.79	0.00	3.65
1163	27	2^{21}	400	27.05	0.76	4.90	0.71
1745	33	2^{25}	500	0.42	1.31	0.00	1.19
2079	37	2^{29}	500	0.04	1.44	2.98	1.33
3505	57	2^{40}	1000	0.08	1.82	0.21	1.74
9429	215	2^{165}	1000	0.00	4.31	0.01	4.16
61345	1434	2^{892}	2000	0.00	31.45	0.99	29.44
213521	4983	2^{3073}	5000	0.02	104.61	0.00	99.37
429586	10002	2^{7455}	10000	0.00	183.75	0.10	172.52

♣ 648 s ♢ 2700 s ♡ 639 s ♠ 2645 s

Figure 7.13: A comparison of the performance of the brute-force approach and our inference algorithm on large expressions containing seeded type errors. The errors are either distributed evenly or clustered within the expression. Our inference algorithm caught 100% of the errors in all cases, so we show only the time taken to do so. The running time of the brute-force approach was capped at one hour (3600 s). For the cases that completed before this cap was reached, we give the running time as a footnote. For cases that did not complete, we ran each test 10 times starting from a random variant, and averaged the results.

are presented in the table in Figure 7.13. Each row represents an artificially constructed expression that varies in the indicated number of dimensions. The size of each expression is given by the number of AST nodes. The expressions are constructed such that not all dimensions are independent (some dimensions are nested within choices), so the number of variants each expression represents is also given.

In each expression, we manually seeded the indicated number of errors ac-

cording to two different distributions: errors may be *spread* evenly throughout the expression or *clustered* together. Thus, each row actually represents two expressions with different error distributions that are otherwise identical. Errors are counted relative to the variational expression, not the variants they occur in. For example, if the expression *err* produces a type error, then $A\langle err, B\langle 1,2\rangle\rangle$ is considered to contain just one type error even though that error is expressed in two variants ($\{A,B\}$ and $\{A,\tilde{B}\}$).

Finally, for each expression we give the percentage of errors caught and total running time in seconds (run on a 2.8GHz dual core processor with 3GB RAM) of the brute-force approach and our inference algorithm, respectively. Often the problem is intractable for the brute-force approach, so we cap the running-time at one hour and count the number of errors caught up to this point. Because of this cap, and especially when errors are clustered, there is a potential for bias in which variants the brute-force algorithm looks before the time limit is reached. To mitigate this, we ran each brute-force test 10 times, starting from random variants, and averaged the results. Note that for presentation reasons we do not list the percentage of errors found for our algorithm since this value is always 100%. Similarly, we do not list the running time of the brute-force approach since this is the full 3600 seconds in all but a few cases, which are indicated by footnotes.

From the results in Figure 7.13 we observe that our algorithm scales well as the size, variability, and number of errors in an expression increases. Our algorithm is also more reliable for detecting errors since the ability to completely type the expressions means that it is not sensitive (in this regard) to the distribution

of errors, and we do not have to consider issues like which variant the algorithm starts with.

Collectively, these results demonstrate the feasibility of error-tolerant type inference on large, complex expressions. In practice, we expect real software to be considerably less complex (from a variational perspective) than the expressions examined in this section, and very unlikely to induce worst-case scenarios. As already mentioned, some real-world studies have suggested an average choice nesting depth of just 1.5 [Kim et al., 2008]. However, it is possible that variational complexity is artificially limited by the inadequacy of current tools, which this work directly addresses.

7.6 Related Work

This section collects work related to error types. Chapter 3 presents a more comprehensive discussion of the work related to other aspects of this dissertation.

Although they share a name, our notion of partial types differs from the work of Thatte [1988]. Thatte's partial types provide a way to type certain objects that are not typable with simple types in lambda calculus, such as heterogeneous lists and persistent data. They are more similar to our typing patterns. Thatte's "untyped" type Ω represents an arbitrary well-typed expression, similar to our $\top$ type, while his inclusion relationship on partial types ($\leq$) is similar to our more-defined relationship on patterns ($\leq$). Type inference with Thatte's partial types was proved decidable [O'Keefe and Wand, 1992; Kozen et al., 1992], a property that holds for

our type system also.

Top and bottom types in subtyping [Pierce, 2002] are also similar to the types $\top$ and $\bot$ used in typing patterns. Moreover, the subtyping relationship plays a similar role to that of $\leq$ on typing patterns. For example, all types are subtypes of the top type, which corresponds to the fact that all typing patterns are less or equally defined as $\top$ (similar for $\bot$ and the bottom type). However, the purpose of these type bounds is quite different. The top and bottom types are introduced to facilitate the proofs of certain properties and the design of type systems, for example, in bounded quantification [Pierce, 1997], whereas the $\top$ and $\bot$ types are used as parts of larger patterns to track which variants are well- and ill-typed, and to mask result types accordingly.

Our work is also related to the work of Siek et al. on gradual typing [Siek and Vachharajani, 2008; Siek and Taha, 2006b]. The goal of that work is to integrate static and dynamic typing into a single type system. They use the symbol ? to represent a type that is not known statically (that is, it is a dynamic type). This is similar to our $\bot$ type in partial types and typing patterns, particularly in the way it is used to determine a notion of *informativeness*. A type is less informative if it contains more ? types (or rather, if more of the type is subsumed by ? types). This relation is similar to an inverse of our more-defined relation on typing patterns, where a pattern becomes less defined as it is subsumed by $\bot$ types. The biggest difference between this work and our own is that $\bot$ types represent parts of a variational program that are statically known to be type incorrect, whereas the parts of a program annotated with ? types may still be dynamically type correct.

Also, their system isolates ? types as much as possible with respect to a plain type, while we allow $\bot$ types to propagate outward in plain types, but contain $\bot$ types to as few *variants* as possible. This is best demonstrated by the fact that (if we extend the notion of definedness to partial types) $\bot$ is equally defined as $\bot \to \texttt{Int}$, but ? is strictly less informative than $? \to \texttt{Int}$.

Chapter 8: Counter-Factual Typing

Changing a program in response to a type error plays an important part in modern software development. However, the generation of good type error messages remains a problem for highly expressive type systems. Existing approaches often suffer from a lack of precision in locating errors and proposing remedies. In Section 3.3, I reviewed numerous efforts aiming at improving type error debugging, including reordering of unification [Yang et al., 2000; Lee and Yi, 2000; Eo et al., 2004; McAdam, 2002b], type error slicing [Tip and Dinesh, 2001; Haack and Wells, 2003; Neubauer and Thiemann, 2003; Schilling, 2012; Zhang and Myers, 2014; Pavlinovic et al., 2014], and suggesting changes [Lerner et al., 2006, 2007; Heeren et al., 2003c; Heeren, 2005].

The shortcomings with approaches based on reordering unifications are that they tend to miss the real cause of type errors, convey the errors in compiler jargon, and don't tell the user how to fix errors. While giving reasons for the failure of unification might be useful for experienced programmers and type system experts, such error messages still require some effort to manually reconstruct some of the types and solve unification problems.

One of the problems of the approaches based on reordering unifications is that they commit to a *single* error location, because in some cases the program text does not contain enough information to confidently make the right decision about

the correct error location. This has led to a number of program slicing approaches that try to identify a *set* of possible error locations instead. However, showing too many program locations involved in the type error diminishes the value of the slicing approach because of the cognitive burden put on the programmer to work through all marked code and to single out the proper error location.

On the other end of the single-vs.-many location spectrum we find approaches that, like GHC or Hugs, follow Johnson and Walz's idea of finding the most likely erroneous location and try to add explanations or suggestions for how to correct the error. Offering change suggestions is, however, a double-edged sword: While it can be very helpful in simplifying the task of fixing type errors, it can also be sometimes very misleading, and frustrating when the suggested change doesn't work. As shown in Section 3.3, both Helium [Heeren, 2005] and Seminal [Lerner et al., 2007], two state-of-the-art change-suggesting tools, fail to correctly locate the error location for the `palin` example.

The task of debugging type errors seems to be an inherently ambiguous undertaking, because in some situations there is just not enough information present in the program to generate a correct change suggestion. Consider, for example, the expression `not 1`. The error in this expression is either `not` or `1`,[1] but without any additional knowledge about the purpose of the expression, there is no way to decide whether to replace the function or the argument. This is why it is generally impossible to isolate one point in the program as the source of a type error.

[1] It could also be the case that the whole expression is incorrect and should be replaced by something else, but we ignore this case for now.

This fact provides a strong justification for slicing approaches that try to provide an unbiased account of error situations. On the other hand, in many cases some locations are more likely than others, and specifically in larger programs, information about the context of an erroneous expression can go a long way of isolating a single location for a type error.

Thus, a reasonable compromise between slicing and single-error-reporting approaches could be a method to principally compute *all* possible type error locations (together with possible change suggestions) and present them ranked and in small portions to the programmer. At the core of such an approach has to be a type checker that produces a complete set of type changes that would make the program type correct.

In this chapter I present a method for *counter-factual change inference*, whose core is a technique to answer the question "What type should a particular subexpression have to remove type errors in a program". In Sections 8.1 and 8.2, I give a high-level view of this method, called *counter-factual typing*, from a conceptual and technical perspective, respectively. In Section 8.3, I discuss the core of the proposed approach, a type system that infers a set of type-change suggestions. Section 8.4 and 8.5 deal with the implementation aspects, including an inference algorithm that implements the type system described in Section 8.3, a set of heuristics for ranking potential change suggestions, and a method for deducing expression changes from type changes. Note that the implementation relies on partial variational unification discussed in Section 7.3. I have evaluated a prototype implementation by comparing it with three closely related tools and found

that counter-factual typing can generate correct change suggestions more often than the other approaches. This evaluation is described in Section 8.6.

8.1 Systematic Identification of Type Errors

It involves a lot of computations to find all potential error causes and find a change suggestion for each identified location. To keep the complexity manageable we only produce so-called *atomic* type changes, that is, type changes for the leaves of the program's abstract syntax tree. This helps avoid the introduction of too exotic or too extreme changes. Consider, for example, the non-atomic type change suggested by Seminal for the `palin` program (page 60), which seems to be not realistic. Or consider changing a whole program to a value of type `Bool` or `Int`, which always works but is hardly ever correct.

However, errors that are best fixed by non-atomic expression changes are quite common. Examples are the swapping of function arguments or the addition of missing function arguments. The identification of such non-atomic program changes is not ruled out by the approach taken and can actually often be achieved by deducing expression changes from type changes.

To increase the readability, I reproduce the `palin` example below, which was first introduced in Section 3.3.

```
fold f z []     = [z]
fold f z (x:xs) = fold f (f z x) xs
flip f x y = f y x
rev = fold (flip (:)) []
palin xs = rev xs == xs
```

Rank	Loc	Code	Change code of type	To new type	Result type
1	(1,19)	`(:)`	`a -> [a] -> [a]` `[z]`	`a -> [b] -> a` `z`	`[a] -> Bool`
2	(5,22)	`xs`	`[a]`	`[[a]]`	`[a] -> Bool`
3	(5,12)	`rev`	`[a] -> [[a]]`	`[a] -> [a]`	`[a] -> Bool`
4	(5,19)	`(==)`	`[a] -> [a] -> Bool`	`[[a]] -> [a] -> b`	`[a] -> b`
5	(4,7)	`fold`	`t1`	`t2`	`[a] -> Bool`

Figure 8.1: Ranked list of single-location type and expression change suggestions inferred for the `palin` example. Note that our prototype represents `[z]` as `(:) z []`. The types `t1` and `t2` in the table are `(a -> b -> a) -> a -> [b] -> [a]` and `([a] -> a -> [a]) -> [b] -> [a] -> [a]`, respectively.

For this example, Figure 8.1 shows a ranked list of all (single-location) type changes, computed by our prototype, that can fix the type error. The correct change ranks first in our method. Note that this is not a representation intended to be given to end users. We rather envision an integration into a user interface in which locations are underlined and hovering over those locations with the mouse will pop up windows with individual change suggestions. In this dissertation I focus on the technical foundation to compute the information required for implementing such a user interface.

Each suggestion is essentially represented by the expression that requires a change together with the inferred actual and expected type of that expression. (Since we are only considering atomic type changes, this expression will always be a constant or variable in case of a type change, but it can be a more complicated expression in case of deduced expression change.) We also show the position of the code in the program[2] and the result types of the program if the corresponding

[2]We have added the line and column numbers by hand since our prototype currently works on

change is adopted. This information is meant as an additional guide for programmers to select among suggestions.

The list of shown suggestions is produced in several steps. First, we generate (lazily) all possible type changes, that is, even those that involve several locations. Note that sometimes the suggested types are unexpected. For example, the suggested type for `fold` is `([a] -> a -> [a]) -> [b] -> [a] -> [a]` although `(a -> b -> a) -> a -> [b] -> a` would be preferable. This phenomenon can be generally attributed to the context of the expression. On the one hand, the given context can be too restrictive and coerce the inferred type to be more specific than it has to be, just as in this example the first argument `flip (:)` forces `fold` to have `[a] -> a -> [a]` as the type of its first argument. On the other hand, the context could also be too unrestrictive. There is no information about how `fold` is related to `[]` in the fourth line of the program. Thus, the type of the second argument of `fold` is inferred as `[b]`. This imprecision can't be remedied by exploiting type information of the program.

Second, we select those type changes that involve only one location. We present those first to the programmer since these are generally easier to understand and to adopt than multi-location change suggestions. Should the programmer reject all these single-location suggestions, two-location suggestions will be presented next, and so on.

Third, in addition to type-change suggestions, we also try to infer some non-atomic expression changes from type changes. In general, only the programmer

---

abstract syntax and doesn't have access to the information from the parser.

who wrote the program knows how to translate required type changes into expression changes. However, there are a number of common programming mistakes, such as swapping or forgetting arguments, that are indicated by type-change suggestions. Similar to Seminal, our prototype identifies these kind of changes that are mechanical and do not require a deep understanding of the program semantics. In our example, we infer the replacement of `[z]` by `z`, because the expected type requires that the return type be the same as the first argument type. We thus suggest to use the first argument, that is `z`, to replace the application `(:) z []`.

Note however, that we don't infer a similar change for the fifth type change because `fold` is partially applied in the definition, and we have no access to the third argument of `fold`. Had the `rev` function been implemented using an eta-expanded list argument, say `xs`, we would have also inferred the suggestion to change `fold (flip (:)) [] xs` to `xs`.

Note also that we do not supplement type-change suggestions with atomic expression changes. For example, in the second suggestion, we do *not* suggest to replace `xs` by `[xs]`. There are two reasons for this. On the one hand, we believe that, given the very specific term to change, the inferred type, and the expected type, the corresponding required expression change is often easy to deduce for a programmer. On the other hand, suggesting specific expression changes requires knowledge about program semantics that is in many cases not readily available in the program. Thus, such suggestions can often be misleading.

Finally, all the type-change suggestions are ranked according to a few simple, but effective complexity heuristics.

8.2 Identifying Type Errors Through Variational Typing

The main idea behind counter-factual typing is to systematically vary parts of the ill-typed program to find changes that can eliminate the corresponding type error(s) from the program. It is infeasible to apply this strategy directly on the expression level since there are generally infinitely many changes that one could consider. Therefore, we perform the variation on the type level. Basically, we ask for each atomic expression e the counter-factual question: *What type should e have to make the program well typed?*

The counter-factual reasoning is built into the type checking process in the following way. To determine the type of an expression e we first infer e's type, say ϕ. But then, instead of fixing this type, we leave the decision open and assume e to have the type $D\langle\phi, a\rangle$, where D is a fresh name and a is a fresh type variable. By leaving the type of e open to revision we account for the fact that e may, in fact, be the source of a type error. By choosing a fresh type variable for e's alternative type, we enable type information to flow from the context of e to forge an alternative type ϕ' that fits into the context in case ϕ doesn't. If ϕ does fit the context, it is unifiable with ϕ', and the choice could in principle be removed. However, this is not really necessary (and we, in fact, don't do this) since in case of a type-correct program, we can find the type at the end of the typing process by simply selecting the first option from all generated choices.

Let us illustrate this idea with a simple example. Consider the expression $e =$ `not 1`. If we vary the types of both `not` and `1`, we obtain the following typing

judgments.

$$\begin{aligned} \texttt{not} &: A\langle \texttt{Bool} \to \texttt{Bool}, a_1 \rangle \\ \texttt{1} &: B\langle \texttt{Int}, a_2 \rangle \end{aligned}$$

where a_1 and a_2 represent the expected types of `not` and `1` according to their respective contexts. To find the types a_1 and a_2, we have to solve the following unification problem.

$$A\langle \texttt{Bool} \to \texttt{Bool}, a_1 \rangle \equiv^? B\langle \texttt{Int}, a_2 \rangle \to a_3$$

where a_3 denotes the result type of the application and $\equiv^?$ denotes that the unification problem is solved modulo the type equivalence relation mentioned in Section 4.3 rather than the usual syntactical identity.

Another subtlety of the unification problem is that two types may not be unifiable. In that case a solution to the unification problem consists of a so-called *partial unifier*, which is both most general and introduces as few errors as possible. The unification algorithm developed in Section 7.3 achieves both these goals.

For the above unification problem, the following unifier is computed. The generality introduced by a_6 and a_7 ensures that only the second alternatives of choices A and B are constrained.

$$\begin{aligned} \{ & a_1 \mapsto A\langle a_6, B\langle \texttt{Int}, a_4 \rangle \to a_5 \rangle, \\ & a_2 \mapsto B\langle a_7, A\langle \texttt{Bool}, a_4 \rangle \rangle, \\ & a_3 \mapsto A\langle \texttt{Bool}, a_5 \rangle \} \end{aligned}$$

Additionally, the unification algorithm returns a typing pattern that characterizes all the viable variants and helps to compute the result type of the varied expres-

sion. In this case we obtain $A\langle B\langle \bot, \top\rangle, \top\rangle$. Based on the unifier and the typing pattern, we can compute that the result type of the varied expression of `not 1` is $\phi = A\langle B\langle \bot, \texttt{Bool}\rangle, a_5\rangle$. From the result type and the unifier, we can draw the following conclusions.

- If we don't change e, that is, we select A and B from the varied expression, the type of the expression is $\bot$ (the variant corresponds to A and B in the result type), which reflects the fact that the original expression is ill typed.
- If we vary `not` to some other expression f, that is, if we select variant $\tilde{A}$ and B from the variational result type, the result type will be a_5. Moreover, the type of f is obtained by selecting $\tilde{A}$ and B from the type that a_1 is mapped to, which yields $\texttt{Int} \to a_5$. In other words, by changing `not` to an expression of type $\texttt{Int} \to a_5$, `not 1` becomes well typed. In the larger context, a_5 may be further constrained to have some other type.
- If we vary `1` to some expression g, that is, if we select A and $\tilde{B}$ from the variational type, then the result type becomes `Bool`.
- If we vary both `not` to f and `1` to g, which means to select $\tilde{A}$ and $\tilde{B}$, the result type is a_5. Moreover, from the unifier we know that f and g should have the types $a_4 \to a_5$ and a_4, respectively.

This gives us all atomic type changes for the expression `not 1`. The combination of creating variations at the type level and variational typing provides an efficient way of finding all possible type changes.

8.3 Type-Change Inference

This section presents the type system that generates a complete set of atomic corrective type changes. After defining the syntax for expressions and types in Section 8.3.1, I present the typing rules for type-change inference in Section 8.3.2. In Section 8.3.3 I investigate some important properties of the type-change inference system.

8.3.1 Syntax

We consider a type checker for lambda calculus with let-polymorphism. To simplify the presentation, I collect the syntax for expressions, types, and meta environments for the type system in Figure 8.2. Note that we use the same notations as we did in earlier chapters.

We use l to denote program locations, in particular, leaves in ASTs. We assume that there is a function $\ell_e(f)$ that returns l for f in e. For presentation purposes, we assume that f uniquely determines a location. We may omit the subscript e when the context is clear. The exact definition of $\ell(\cdot)$ does not matter.

We use the choice environment Δ to associate choice types that were generated during the typing process with the corresponding location in the program. Operations on types can be lifted to Δ by applying them to the types in Δ.

Expressions	e	::=	v	*Constant*
		|	x	*Variable*
		|	$\lambda x.e$	*Abstraction*
		|	$e\ e$	*Application*
		|	**let** $x = e$ **in** e	*Polymorphism*
		|	**if** e **then** e **else** e	*Conditional*
Monotypes	τ	::=	γ	*Type constant*
		|	a	*Type variable*
		|	$\tau \to \tau$	*Function type*
Variational types	ϕ	::=	τ	*Monotype*
		|	$\bot$	*Error type*
		|	$D\langle\phi,\phi\rangle$	*Choice type*
		|	$\phi \to \phi$	*Function type*
Polymorphic types	σ	::=	ϕ	*Variational type*
		|	$\forall\overline{a}.\phi$	*Polymorphic type*
Type environments	Γ	::=	$\varnothing \mid \Gamma, x \mapsto \sigma$	
Substitutions	θ	::=	$\varnothing \mid \theta, a \mapsto \phi$	
Choice environments	Δ	::=	$\varnothing \mid \Delta, (l, D\langle\phi,\phi\rangle)$	

Figure 8.2: Syntax of expressions, types, and environments.

8.3.2 Typing Rules

Figure 8.3 presents the typing rules for inferring type changes. The typing judgment is of the form $\Gamma \vdash e : \phi | \Delta$ and produces as a result a variational type ϕ that represents all the typing "potential" for e plus a set of type changes Δ for the atomic subexpressions of e that will lead to the types in ϕ.

Since we are only interested in atomic changes during this phase, we only vary the leaves in the AST of programs, which are constants and variable references.

$$\boxed{\Gamma \vdash e : \phi | \Delta}$$

$$\textsc{Con}\ \frac{v \text{ is of type } \gamma \qquad D \text{ fresh}}{\Gamma \vdash v : D\langle \gamma, \phi \rangle | \{(\ell(v), D\langle \gamma, \phi \rangle)\}} \qquad \textsc{Var}\ \frac{\Gamma(x) = \forall \overline{a}.\phi_1 \qquad D \text{ fresh} \qquad \phi = \{\overline{a \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash x : D\langle \phi, \phi_2 \rangle | \{(\ell(x), D\langle \phi, \phi_2 \rangle)\}}$$

$$\textsc{Unbound}\ \frac{x \notin dom(\Gamma) \qquad D \text{ fresh}}{\Gamma \vdash x : D\langle \bot, \phi \rangle | \{(\ell(x), D\langle \bot, \phi \rangle)\}} \qquad \textsc{Abs}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi' | \Delta}{\Gamma \vdash \lambda x.e : \phi \to \phi' | \Delta}$$

$$\textsc{Let}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi | \Delta \qquad \overline{a} = FV(\phi) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall \overline{a}.\phi \vdash e' : \phi' | \Delta'}{\Gamma \vdash \textbf{let } x = e \textbf{ in } e' : \phi' | \Delta \cup \Delta'}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \phi_1 | \Delta_1 \qquad \Gamma \vdash e_2 : \phi_2 | \Delta_2 \qquad \phi_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash e_1\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

$$\textsc{If}\ \frac{(\Gamma \vdash e_i : \phi_i | \Delta_i)^{i:1..3} \qquad \pi_1 = \phi_1 \bowtie \texttt{Bool} \qquad \pi_2 = \phi_2 \bowtie \phi_3 \qquad \phi = \pi_1 \lhd (\pi_2 \lhd \phi_2)}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \phi | \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

Figure 8.3: Rules for type-change inference.

This is reflected in the typing rules as we generate fresh choices in rules CON, VAR, and UNBOUND. In each case, we place the actual type in the first alternative and an arbitrary type in the second alternative of the choice. When an unbound variable is accessed, it causes a type error. We thus put $\bot$ in the first alternative of the choice.

The rules ABS and LET for abstractions and let-expressions are very similar to those in other type systems except that variables are bound to variational types.

The rule APP for typing applications is very similar to the application rule discussed in Section 7.2. The only difference is that the rule here keeps track of the information for Δ.

The IF rule employs the same machinery as the APP rule for the potential introduction of type errors and partially correct types. In particular, the condition e_1 is not strictly required to have the type `Bool`. However, only the variants that are equivalent to `Bool` are type correct. Likewise, only the variants in which both branches are equivalent are type correct.

8.3.3 Properties

In this section I investigate some important properties of the type-change inference system. I show that it is consistent in the sense that any type selected from the result variational type can be obtained by applying the changes as indicated by that selection. I also show that the type-change inference is complete in finding all corrective atomic type changes. Based on this result I also show that the type-change inference system is a conservative extension of HM.

We start with the observation that type-change inference always succeeds in deriving a type for any given expression and type environment.

Lemma 18 *Given e and Γ, there exist ϕ and Δ such that $\Gamma \vdash e : \phi | \Delta$.*

The proof of this lemma is obvious because for any construct in the language, even for unbound variables, there is a corresponding typing rule in Figure 8.3 that is applicable and returns a type.

$$\boxed{\Delta \Downarrow \Delta'}$$

$$\varnothing \Downarrow \varnothing \qquad \frac{\Delta \Downarrow \Delta' \qquad \exists \tau : D\langle \overline{\phi} \rangle \equiv \tau}{\Delta, (l, D\langle \overline{\phi} \rangle) \Downarrow \Delta'} \qquad \frac{\Delta \Downarrow \Delta' \qquad \neg \exists \tau : D\langle \overline{\phi} \rangle \equiv \tau}{\Delta, (l, D\langle \overline{\phi} \rangle) \Downarrow \Delta', (l, D\langle \overline{\phi} \rangle)}$$

$$\boxed{\lfloor \rfloor_{\_} : \phi \times s \to \phi}$$

$$\begin{aligned}
\lfloor \tau \rfloor_s &= \tau \\
\lfloor \phi_1 \to \phi_2 \rfloor_s &= \lfloor \phi_1 \rfloor_s \to \lfloor \phi_2 \rfloor_s \\
\lfloor \bot \rfloor_s &= \bot \\
\lfloor B\langle \overline{\phi} \rangle \rfloor_A &= B\langle \overline{\lfloor \phi \rfloor_A} \rangle \quad \text{if } A \neq B \\
\lfloor B\langle \overline{\phi} \rangle \rfloor_{\tilde{A}} &= B\langle \overline{\lfloor \phi \rfloor_{\tilde{A}}} \rangle \quad \text{if } A \neq B \\
\lfloor B\langle \phi_1, \phi_2 \rangle \rfloor_B &= \lfloor \phi_1 \rfloor_B \\
\lfloor B\langle \phi_1, \phi_2 \rangle \rfloor_{\tilde{B}} &= \lfloor \phi_2 \rfloor_{\tilde{B}}
\end{aligned}$$

Figure 8.4: Simplifications and selection.

Next, we need to simplify Δ in the judgment $\Gamma \vdash e : \phi | \Delta$ to investigate the properties of the type system. Specifically, we define a simplification relation $\Downarrow$ in Figure 8.4 that eliminates idempotent choices from Δ. Note that the sole purpose of simplification is to eliminate choice types that are equivalent to monotypes, or equivalently, remove all positions that don't contribute to type errors. Thus, there is no need to simplify types nested in choice D in Figure 8.4. Also, we formally define the selection operation $\lfloor \phi \rfloor_s$ in Figure 8.4.

Next we want to establish the correctness of the inferred type changes. Formally, a *type update* is a mapping from program locations to monotypes. The intended meaning of one particular type update $l \mapsto \tau$ is to change the expression at l to an expression of type τ. We use ω to range over type updates. A type update is given by the locations and the second component of the corresponding choice

$$\textsc{Con-C}\ \frac{\nu \text{ is of type } \gamma}{\Gamma;\omega \vdash^C \nu : \omega(\nu)||\gamma} \qquad \textsc{Var-C}\ \Gamma;\omega \vdash^C x : \omega(x)||\{\overline{\alpha \mapsto \tau}\}(\Gamma(x))$$

$$\textsc{App-C}\ \frac{\Gamma;\omega \vdash^C e_1 : \tau_1 \to \tau \qquad \Gamma;\omega \vdash^C e_2 : \tau_1}{\Gamma;\omega \vdash^C e_1\ e_2 : \tau}$$

$$\textsc{If-C}\ \frac{\Gamma;\omega \vdash^C e_1 : \texttt{Bool} \qquad \Gamma;\omega \vdash^C e_2 : \tau \qquad \Gamma;\omega \vdash^C e_3 : \tau}{\Gamma;\omega \vdash^C \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

Figure 8.5: Rules for the type-update system.

types in the choice environment. We use $\downarrow\cdot$ to extract that mapping from Δ. The definition is $\downarrow\Delta = \{l \mapsto \tau_2 \mid (l, D\langle\tau_1,\tau_2\rangle) \in \Delta\}$. (For the time being we assume that all the alternatives of choices in Δ are monotypes; we will lift this restriction later.) For example, with $\Delta = \{(l, A\langle\texttt{Int},\texttt{Bool}\rangle)\}$ we have $\downarrow\Delta = \{l \mapsto \texttt{Bool}\}$.

The application of a type update is part of a *type update system* that is defined by the set of typing rules shown in Figure 8.5. These typing rules are identical to an ordinary Hindley-Milner type system, except that they allow to "override" the types of atomic expressions according to a type update ω that is a parameter for the rules. We only show the rules for constants, variables, applications, and conditionals since those for abstractions and **let** expressions are obtained from the HM ones in the same way as the application rule by simply adding the ω parameter. We write more shortly $\omega(e)$ for $\omega(\ell(e))$, and we use the "orelse" notation $\omega(e)||\tau$ to pick the type $\omega(e)$ if $\omega(e)$ is defined and τ otherwise.

The rules CON-C and VAR-C employ a type update if it exists. Otherwise, the usual typing rules apply. Rule APP-C delegates the application of change updates to subexpressions since we are considering atomic change suggestions only.

We can now show that by applying any of the inferred type changes (using the rules in Figure 8.5), we obtain the same types that are encoded in the variational type potential computed by type-change inference. We employ the following additional notation. We write $\Delta.2$ for the decision of selectors $\tilde{D}$ for each choice $D\langle\rangle$ in Δ. For example, $\{(\ell_1, A\langle \texttt{Int}, \texttt{Bool}\rangle), (\ell_2, B\langle \texttt{Bool}, \texttt{Int}\rangle)\}.2 = \{\tilde{A}, \tilde{B}\}$. Formally, we have the following result. (We assume that Δ has been simplified by $\Downarrow$ in Figure 8.4 and the alternatives of choices in Δ are plain, as mentioned before.)

Theorem 18 (Type-change inference is consistent) *For any given e and Γ, if $\Gamma \vdash e : \phi | \Delta$ and there is some τ such that $\lfloor \phi \rfloor_{\Delta.2} = \tau$, then $\Gamma; \downarrow\Delta \vdash^C e : \tau$.*

PROOF. A detailed proof is given in Appendix C. □

Moreover, the type-change inference is complete since it can generate a set of type changes for any desired type.

Theorem 19 (Type-change inference is complete) *For any e, Γ and ω, if $\Gamma; \omega \vdash^C e : \tau$, then there exist ϕ, Δ, and a typing derivation for $\Gamma \vdash e : \phi | \Delta$ such that $\downarrow\Delta = \omega$ and $\lfloor \phi \rfloor_{\Delta.2} = \tau$.*

PROOF. A detailed proof is given in Appendix C. □

The introduction of arbitrary alternative types in rules CON, VAR, and UNBOUND are the reason that type-change inference is highly non-deterministic, that is, for

any expression e we can generate an arbitrary number of type derivations with different type potentials and corresponding type changes.

Many of those derivations don't make much sense. For example, we can derive $\Gamma \vdash \texttt{5} : A\langle \texttt{Int}, \texttt{Bool}\rangle | \Delta$ where $\Delta = \{(\ell_5(\texttt{5}), A\langle \texttt{Int}, \texttt{Bool}\rangle)\}$. However, since the expression `5` is type correct, it doesn't make sense to suggest a change for it.

On the other hand, the ill-typed expression $e = \texttt{not (succ 5)}$ can be typed in two different ways that can correct the error, yielding two different type potentials and type changes. We can either suggest to change `succ` to an expression of type $\texttt{Int} \to \texttt{Bool}$, or we can suggest to change `not` into something of type $\texttt{Int} \to a_1$. The first suggestion is obtained by a derivation for $\Gamma \vdash e : A\langle \bot, a_1\rangle | \Delta_1$ with $\Delta_1 = \{(\ell_e(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to a_1\rangle)\}$. The second suggestion is obtained by a derivation for $\Gamma \vdash e : B\langle \bot, \texttt{Bool}\rangle | \Delta_2$ with $\Delta_2 = \{(\ell_e(\texttt{succ}), B\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle)\}$.

Interestingly, we can combine both suggestions by deriving a more general typing statement, that is, we can derive the judgment $\Gamma \vdash e : A\langle B\langle \bot, \texttt{Bool}\rangle, B\langle a_1, a_2\rangle\rangle | \Delta_3$ where

$$\begin{aligned}\Delta_3 = \{&(\ell_e(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, B\langle \texttt{Int} \to a_1, a_3 \to a_2\rangle\rangle),\\ &(\ell_e(\texttt{succ}), B\langle \texttt{Int} \to \texttt{Int}, A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to a_3\rangle\rangle)\}\end{aligned}$$

We can show that the third typing is better than the first two in the sense that its result type (a) contains fewer type errors than either of the result types and (b) is more general. For example, by selecting $\{A, \tilde{B}\}$ from both result types, we obtain $\bot$ and `Bool`, respectively. Making the same selection into the third result type, we obtain `Bool`. Likewise, when we select with $\{\tilde{A}, B\}$, we get the types a_1, $\bot$, and a_1, respectively. For each selection, the third result type is better than either one of

the first two.

In the following we show that this is not an accident, but that we can, in fact, always find a most general change suggestion from which all other suggestions can be instantiated.

First, we extend the function $\downarrow$ to take as an additional parameter a set of selectors δ. We also extend the definition to work with general variational types (and not just monotypes).

$$\downarrow_\delta \Delta = \{l \mapsto \lfloor \phi_2 \rfloor_\delta \mid (l, D\langle \phi_1, \phi_2 \rangle) \in \Delta \wedge \tilde{D} \in \delta\}$$

Intuitively, we consider all the locations for which the second alternative of the corresponding choices are chosen. We need to apply the selection $\lfloor \phi_2 \rfloor_\delta$ because each variational type may include other choice types that are subject to selection by δ.

Next we will show that type-change inference produces most general type changes from which any individual type change can be instantiated. We observe that type potentials and type changes can be compared in principally two different ways. First, the result of type-change inference $\phi|\Delta$ can be *more defined* than another result $\phi'|\Delta'$, which means that for any δ for which $\lfloor \phi' \rfloor_\delta$ yields a monotype then so does $\lfloor \phi \rfloor_\delta$. Second, a result $\phi|\Delta$ can be *more general* than another result $\phi'|\Delta'$, written as $\phi \leq \phi'$, if there is some type substitution θ such that $\phi' = \theta(\phi)$. (Similarly, we call a type update ω_1 more general than another type update ω_2, written as $\omega_1 \leq \omega_2$, if $dom(\omega_1) = dom(\omega_2)$ and there is some θ such that for all l $\omega_2(l) = \theta(\omega_1(l))$.

Since we have these two different relationships between type changes, we have to show the generality of type-change inference in several steps.

First, we show that we can generalize any type change that produces a type error in the resulting variational type for a particular selection when there is another type change that does not produce a type error for the same selection. In the following lemma, we stipulate that the two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$ assign the same choice name to the same program location.

Lemma 19 (Most defined type changes) *Given e and Γ and two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$, if $\lfloor \phi_1 \rfloor_\delta = \bot$ and $\lfloor \phi_2 \rfloor_\delta = \tau$, then there is a typing $\Gamma \vdash e : \phi_3 | \Delta_3$ such that*

- $\lfloor \phi_3 \rfloor_\delta = \lfloor \phi_2 \rfloor_\delta$ *and for all other* δ' $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$.
- $\downarrow_\delta \Delta_3 = \downarrow_\delta \Delta_2$ *and* $\downarrow_{\delta'} \Delta_3 = \downarrow_{\delta'} \Delta_1$ *for all other* δ'.

PROOF. A detailed proof is given in Appendix C. □

Next we show that given any two type changes, we can always find a type change that generalizes the two.

Lemma 20 (Generalizability of type changes) *For any two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$, if neither $\lfloor \phi_1 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$ nor $\lfloor \phi_2 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta$ holds, there is a typing $\Gamma \vdash e : \phi_3 | \Delta_3$ such that*

- $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta$, $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$ *and for all other* δ', $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$.
- $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_1$, $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_2$ *and for all other* δ', $\downarrow_{\delta'} \Delta_3 = \downarrow_{\delta'} \Delta_1$.

PROOF. A detailed proof is given in Appendix C. □

We can now combine and generalize Lemmas 19 and 20 and see that type-change inference can always produce most general typings with minimal change sets. This is an important result, captured in the following theorem.

Theorem 20 (Most general and error-free type changes) *Given e and Γ and two typings $\Gamma \vdash e : \phi_1 | \Delta_1$ and $\Gamma \vdash e : \phi_2 | \Delta_2$, there is a typing $\Gamma \vdash e : \phi_3 | \Delta_3$ such that for any δ,*

- *if $\lfloor \phi_1 \rfloor_\delta = \bot$ and $\lfloor \phi_2 \rfloor_\delta = \tau$, then $\lfloor \phi_3 \rfloor_\delta = \tau$ and $\downarrow_\delta \Delta_3 = \downarrow_\delta \Delta_2$.*
- *if $\lfloor \phi_2 \rfloor_\delta = \bot$ and $\lfloor \phi_1 \rfloor_\delta = \tau$, then $\lfloor \phi_3 \rfloor_\delta = \tau$ and $\downarrow_\delta \Delta_3 = \downarrow_\delta \Delta_1$.*
- *if $\lfloor \phi_1 \rfloor_\delta = \tau_1$ and $\lfloor \phi_2 \rfloor_\delta = \tau_2$, then $\lfloor \phi_3 \rfloor_\delta \leq \tau_1$ and $\lfloor \phi_3 \rfloor_\delta \leq \tau_2$. Moreover, $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_1$ and $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_2$.*

PROOF. By construction, we delegate the actual construction process to the one described in the proof for Lemma 19 or that for Lemma 20. In particular, given two typings, only one of the three cases as mentioned in the theorem can occur. First, if $\lfloor \phi_1 \rfloor_\delta = \bot$ and $\lfloor \phi_2 \rfloor_\delta = \tau$, we use the idea presented in the proof for Lemma 19 to construct the new typing. The second case is a dual case of the first case, where $\lfloor \phi_2 \rfloor_\delta = \bot$ and $\lfloor \phi_1 \rfloor_\delta = \tau$. We proceed as we do in the first case but swap ϕ_1 and ϕ_2, and also Δ_1 and Δ_2. Finally, if $\lfloor \phi_1 \rfloor_\delta = \tau_1$ and $\lfloor \phi_2 \rfloor_\delta = \tau_2$, we use the construction process described in the proof for Lemma 20 to construct the new typing. In each case, the proof follows directly from Lemma 19 or Lemma 20. □

From Theorems 19 and 20 it follows that there is a typing for complete and principal type changes. We express this in the following theorem, where we write

$\omega_1 \leq \omega_2$ if $dom(\omega_1) = dom(\omega_2)$ and $\forall l : l \in dom(\omega_1) \Rightarrow \omega_1(l) \leq \omega_2(l)$.

Theorem 21 (Complete and principal type changes) *Given e and Γ, there is a typing $\Gamma \vdash e : \phi|\Delta$ such that for any ω if $\Gamma;\omega \vdash^C e : \tau$, then there is some δ such that $\lfloor\phi\rfloor_\delta \leq \tau$ and $\downarrow_\delta \Delta \leq \omega$.*

PROOF. Based on Theorem 19, if $\Gamma;\omega_i \vdash^C e : \tau_i$ then there is a typing and some δ such that $\Gamma \vdash e : \phi_i|\Delta_i$ with $\lfloor\phi_i\rfloor_\delta = \tau_i$ and $\downarrow_\delta \Delta_i = \omega_i$. For different τ_is, we may get different ϕ_is and Δ_is. Based on Theorem 20, there is a typing $\Gamma \vdash e : \phi|\Delta$ that is better than all typings with ϕ_is and Δ_is. The result holds. □

Finally, there is a close relationship between type-change inference and the HM type system. When type-change inference succeeds with an empty set of type changes, it produces a non-variational type that is identical to the one derived by HM. This result is captured in the following theorem, where the judgment $\Gamma \vdash^H e : \tau$ is introduced in Section 2.1.1 expressing that expression e has the type τ under Γ in HM type system.

Theorem 22 *For any given e and Γ, $\Gamma;\varnothing \vdash^C e : \tau \iff \Gamma \vdash^H e : \tau$.*

PROOF. The proof is a straightforward induction over the typing relation in Figure 8.5 and Figure 2.1. □

Based on Theorem 18, Theorem 19, Theorem 22 and the fact that $\downarrow\varnothing = \varnothing$, we can infer that when a program is well typed, the type change-inference system and the HM system produce the same result.

Theorem 23 $\Gamma \vdash e : \tau | \varnothing$ *if and only if* $\Gamma \vdash^H e : \tau$.

PROOF. Based on Theorem 18, $\Gamma \vdash e : \tau | \varnothing$ implies that $\Gamma; \varnothing \vdash^C e : \tau$, which implies $\Gamma \vdash^H e : \tau$ according to Theorem 22. Meanwhile, $\Gamma \vdash^H e : \tau$ implies $\Gamma; \varnothing \vdash^C e : \tau$ according to Theorem 22. Based on Theorem 19, $\Gamma \vdash e : \tau | \varnothing$ holds. □

Note that $\Gamma \vdash e : \tau | \varnothing$ implies that $\Gamma \vdash e : \phi | \Delta$, $\phi \equiv \tau$, and $\Delta \Downarrow \varnothing$. This theorem also implies that type-change inference will never assign a monotype to a type-incorrect program.

8.4 Implementation

This section presents an algorithm for inferring type changes. We will discuss properties of the algorithm as well as strategies to bound its complexity.

Given the partial type unification algorithm presented in Section 7.3, the inference algorithm is obtained by a straightforward translation of the typing rules presented in Figure 8.3. We present this algorithm in Figure 8.6, where we show the cases for variable references, applications, and **if** expressions. The cases for abstractions and **let** expressions can be derived from $\mathcal{W}$ by simply adding the threading of Δ.

For variable reference, the algorithm first tries to find the type of the variable in Γ and either instantiates the found type schema with fresh type variables or returns $\perp$ if the variable is unbound. After that, a fresh choice containing a fresh

$cft : \Gamma \times e \to \eta \times \phi \times \Delta$
$cft(\Gamma, x) =$
  $\phi' \leftarrow inst(\Gamma(x))$ return $\bot$ when x is unbound
  $\phi \leftarrow D\langle \phi', a \rangle$ where D and a fresh
  return $(\varnothing, \phi, \{(\ell(x), \phi)\})$

$cft(\Gamma, e_1\ e_2) =$
  $(\eta_1, \phi_1, \Delta_1) \leftarrow cft(\Gamma, e_1)$
  $(\eta_2, \phi_2, \Delta_2) \leftarrow cft(\eta_1(\Gamma), e_2)$
  $(\eta_3, \pi) \leftarrow punify(\eta_2(\phi_1), \eta_2(\phi_2) \to a)$ where a fresh
  $\phi \leftarrow \pi \lhd \eta_3(a)$
  $\eta \leftarrow \eta_3 \circ \eta_2 \circ \eta_1$
  return $(\eta, \phi, \eta(\Delta_1 \cup \Delta_2))$

$cft(\Gamma, \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) =$
  $(\eta_1, \phi_1, \Delta_1) \leftarrow cft(\Gamma, e_1)$
  $(\eta', \pi') \leftarrow punify(\phi_1, \texttt{Bool})$
  $(\eta_2, \phi_2, \Delta_2) \leftarrow cft(\eta' \circ \eta_1(\Gamma), e_2)$
  $(\eta_3, \phi_3, \Delta_3) \leftarrow cft(\eta_2 \circ \eta' \eta_1(\Gamma), e_3)$
  $(\eta_4, \pi_4) \leftarrow punify(\eta_3(\phi_2), \phi_3,)$
  $\eta \leftarrow \eta_4 \circ \eta_3 \circ \eta_2 \circ \eta' \circ \eta_1$
  return $(\eta, \pi' \lhd (\pi_4 \lhd \eta_4(\phi_3)), \eta(\Delta_1 \cup \Delta_2 \cup \Delta_3))$

Figure 8.6: An inference algorithm implementing counter-factual typing.

type variable is returned. The variable then has the returned choice type with the inferred type in the first alternative and the type variable in the second.

For typing **if** expressions, we use the algorithm $punify(\phi_1, \phi_2)$ developed in Section 7.3 for partial unification. In addition to a partial unifier a typing pattern is generated to describe which variants are unified successfully and which aren't (see Section 7.1). Otherwise, the algorithm follows in a straightforward way the usual strategy for type inference.

We can prove that the algorithm *cft* correctly implements the typing rules in Figure 8.3, as expressed in the following theorems.

Theorem 24 (Type-change inference is sound) *Given any* e *and* Γ*, if* $cft(\Gamma, e) = (\eta, \phi, \Delta)$*, then* $\eta(\Gamma) \vdash e : \phi | \Delta$.

At the same time, the type inference is complete and principal. We use the auxiliary relation $\phi_1 \preceq \phi_2$ to express that for any δ, either $\lfloor \phi_2 \rfloor_\delta = \bot$ or $\lfloor \phi_1 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$. Intuitively, this expresses that either the corresponding variant in ϕ_1 is more general or more correct. We also define $\Delta_1 \preceq \Delta_2$ if for any $(l, \phi_1) \in \Delta_1$ and $(l, \phi_2) \in \Delta_2$ the condition $\phi_1 \preceq \phi_2$ holds.

Theorem 25 (Type-change inference is complete and principal) *If* $\eta(\Gamma) \vdash e : \phi | \Delta$*, then* $cft(\Gamma, e) = (\eta_1, \phi_1, \Delta_1)$ *such that* $\eta = \theta_1 \circ \eta_1$ *for some* θ_1*,* $\Delta_1 \preceq \Delta$*, and* $\phi_1 \preceq \phi$.

From Theorems 20 and 25 it follows that our type-change inference algorithm correctly computes all type changes for a given expression in one single run.

During the type-change inference process, choice types can become deeply nested and the size of types can become exponential in the nesting levels. Fortunately, this occurs only with deep nestings of function applications where each argument type is required to be the same. For example, the function $f : a \to a \to \ldots \to a$ is more likely to cause this problem than the functions $g : a_1 \to a_2 \to \ldots \to a_n$ and $h : \gamma \to \gamma \to \ldots \to \gamma$ because only the function f requires all argument types to be unified, which causes choice nesting to happen.

To keep the run-time complexity of our inference algorithm under control, we eliminate choices beyond an adjustable nesting level that satisfy one of the following conditions: (A) choices whose alternatives are unifiable, and (B) choices whose

alternatives contain errors in the same places. These two conditions ensure that the eliminated choices are unlikely to contribute to type errors. There are cases in which this strategy fails to eliminate choices, but this happens only when there are already too many type errors in the program, and we therefore stop the inference process and report type errors and change suggestions found so far.

This strategy allows us to maintain choices whose corresponding locations are likely sources of type errors and discard those that aren't. Note, however, that this strategy sacrifices the completeness property captured in Theorem 25. We have evaluated the running time and the precision of error diagnosis against the choice nesting levels (see Section 8.6). We observed that only in very rare cases will the choice nesting level reach 17, a value that variational typing is able to deal with decently (Section 6.2).

Finally, we briefly describe a set of simple heuristics that define the ranking of type and expression changes.

(1) We prefer places that have deduced expression changes (see Section 8.5) because these changes reflect common editing mistakes [Lerner et al., 2007].

(2) We favor changes that are lower in the abstract syntax trees because changes at those places have least effect on the context and are least likely to introduce exotic results.

(3) We prefer changes that have minimal shape difference between the inferred type and the expected type. For example, a change that doesn't influence the arities of function types is ranked higher than a change that does change ari-

ties.

8.5 Deducing Expression Changes

While it is generally impossible to deduce expression changes from type changes, there are several idiosyncratic situations in which type changes do point to likely expression changes. These situations can be identified by unifying both types of a type change where the unification is performed modulo a set of axioms that represent the pattern inherent in the expression change.

As an example, consider the following expression.[3]

```
zipWith (\(x,y) -> x+y) [1,2] [3,4]
```

Our type change inference suggests to change `zipWith` from its original type `(a -> b -> c) -> [a] -> [b] -> [c]` to something of type `((Int,Int) -> Int) -> [Int] -> [Int] -> d`. Given these two types, we can deduce to curry the first argument to the function `zipWith` to remove the type error. (At the same time, we substitute `d` in the result type with `Int`.) Overall, our approach suggests to change `\(x,y) -> x+y` to `\x y -> x+y`.

By employing unification modulo different theories, McAdam [2002a] has developed a theory and an algorithm to systematically deduce changes of this sort. We have adopted this approach (and extended it slightly) for deducing expression changes, such as swapping the arguments of function calls, currying and uncurrying of functions, or adding and removing arguments of function calls.

[3] This example is adapted from [Lerner et al., 2007], where `zipWith` is called `map2`.

The extension is based on a simple form of identifying non-arity-preserving type changes. Such a change is used to modify the types, then McAdam's approach is applied, and the result is then interpreted in light of the non-arity-preserving type change as a new form of expression change. As an example, here is the method of identifying the addition or removal of arguments to function calls. In this case, the differences in the two types to be unified will lead to a second-level type change that pads one of the types with an extra type variable. For example, given the inferred type $\tau_1 \to \tau_3$ and the expected type $\tau_1 \to \tau_2 \to \tau_3$, we turn the first type into $a \to \tau_1 \to \tau_3$. The application of McAdam's approach suggests to swap the arguments. Also, a is mapped to τ_2. Interpreting the swapping suggestion through the second-level type change of padding, we deduce the removal of the second argument.

Besides these systematic change deductions, we also support some ad-hoc expression changes. Specifically, we infer changes by inspecting the expected type only. For example, if the inferred type for `f` in `f g e` is `b -> c` while the expected type is `(a -> b) -> a -> c`, we suggest to change `f g e` to `f (g e)`.

Another example are situations in which the result type of an expected type matches exactly one of its (several) argument types. In that case we suggest to replace the whole expression with the corresponding argument. This case applies, in fact, to the `palin` example, where the type change for `(:)` is to replace `a -> [a] -> [a]` by `a -> [b] -> a`. We therefore infer to replace `(:) z []`, which is `[z]`, by `z` because the first argument type is the same as the return type. Another case is when in expression `f g h` the expected type for `f` is `(a -> b) -> a -> b`. Then

we suggest to remove `f` from the expression. There are more such ad-hoc changes that are useful in some situations, but we will not discuss them here.

In Section 8.7 we will compare our method with McAdam's original. Here we only note that the success of the method in our prototype depends to a large degree on the additional information provided by type-change inference, specifically, the more precise and less biased expected types that are used for the unification.

8.6 Evaluation

To evaluate the usefulness and efficiency of the counter-factual typing approach, we have implemented a prototype of type-change inference and expression-change deduction in Haskell. (In addition to the constructs shown in Section 8.3.1 the prototype also supports some minor, straightforward extensions, such as data types and case expressions.) We compare the results produced by our CF typing tool to Seminal [Lerner et al., 2006, 2007], Helium [Heeren et al., 2003c; Heeren, 2005], and GHC. There are several reasons for selecting this group of tools. First, they provide currently running implementations. Second, these tools provide a similar functionality as CF typing, namely, locating type errors and presenting change suggestions, both at the type and the expression level. We have deliberately excluded slicing tools from the comparison because they only show all possible locations, and don't suggest changes.

For evaluating the applicability and accuracy of the tools we have gathered a collection of 121 examples from 22 publications about type-error diagnosis. These

papers include recent Ph.D. theses [Yang, 2001; McAdam, 2002a; Heeren, 2005; Wazny, 2006] and papers that represent most recent and older work [Schilling, 2012; Lerner et al., 2007; Johnson and Walz, 1986]. These papers cover many different perspectives of the type-error debugging problem, including error slicing, explanation systems, reordering of unification, automatic repairing, and interactive debugging. Since the examples presented in each paper have been carefully chosen or designed to illustrate important problem cases for type-error debugging, we have included them all, except for examples that involve type classes since our tool (as well as Seminal) doesn't currently support type classes. This exclusion did not have a significant effect. We gathered 8 unique examples regarding type classes involved in type errors discussed in [Stuckey et al., 2003; Wazny, 2006]. Both GHC and Helium were able to produce a helpful error message in only 1 case. Otherwise, the examples range from very simple, such as `test = map [1,10] even` to very complex ones, such as the `plot` example introduced in [Wazny, 2006].

We have grouped the examples into two categories. The first group ("with Oracle") contains 86 examples for which the correct version is known (because it either is mentioned in the paper or is obvious from the context). The other group ("ambiguous") contains the remaining 35 examples that can be reasonably fixed by several different single-location changes. For the examples in the "with Oracle" group, we have recorded how many correct suggestions each tool can find with at most n attempts. For the examples in the "ambiguous" group, we have determined how often a tool produces a complete, partial, or incorrect set of suggestions. For example, for the expression `\f g a -> (f a, f 1, g a, g True)`, which is given

	86 examples with Oracle					35 ambiguous examples		
	1	2	3	≥ 4	never	complete	partial	incorrect
CF typing	67.4	80.2	88.4	91.9	8.1	100.0	0.0	0.0
Seminal	47.7	54.7	58.1	59.3	40.7	40.0	25.7	34.3
Helium	61.6	-	-	61.6	38.4	0.0	100.0	0.0
GHC	17.4	-	-	17.4	82.6	0.0	34.3	65.7

Figure 8.7: Evaluation results for different approaches over 121 collected examples (in %).

in [Bernstein and Stark, 1995], Helium suggests to change `True` to something of type `Int`. While this is correct, there are also other changes possible, for example, changing `f 1` to `f True`. Since these are not mentioned, the result is categorized as partial.

Figure 8.7 presents the results for the different tools and examples with unconstrained choice nesting level for CF typing. Note that GHC's output is considered correct only when it points to the correct location and produces an error message that is not simply reporting a unification failure or some other compiler-centric point of view. We have included GHC only as a baseline since it is widely known. The comparison of effectiveness is meant to be between CF typing, Seminal, and Helium.

The numbers show that CF typing performs overall best. Even if we only consider the first change suggestion, it outperforms Helium which comes in second. Taking into account second and third suggestions, Seminal catches up, but CF typing performs even better.

In cases where Helium produces multiple suggestions, all suggestions are wrong. For CF typing 21 out of the 58 correct suggestions (that is, 36%) are ex-

pression changes. For Seminal the numbers are 20 out of 41 (or 51%), and for Helium it is 15 out of 52 (or 29%). This shows that Seminal produces a higher rate of expression change suggestions at a lower overall correctness rate.

Most of Helium and Seminal's failures are due to incorrectly identified change locations. Another main reason for Seminal's incorrect suggestions is that it introduces too extreme changes. In several cases, Seminal's change suggestion doesn't fix the type error.

Most cases for which CF typing fails are caused by missing parentheses. For example, for the expression `print "a" ++ "b"` [Lerner et al., 2007], our approach suggests to change `print` from the inferred type `a -> IO ()` to the type `String -> String` or change `(++)` from the expected type `[a] -> [a] -> [a]` to the inferred type `IO () -> String -> String`. Neither of the suggestions allows us to deduce the regrouping of the expression.

To summarize, since the examples that we used have been designed to test very specific cases, the numbers do not tell much about how the systems would perform in everyday practice. They provide more like a stress test for the tools, but the direct comparison shows that CF typing performs very well compared with other tools and thus presents a viable alternative to type debugging.

With the help of variational typing, we can generate all the potential changes very efficiently. The running time for all the collected examples is within 2 seconds. Figure 8.8 shows the running time for both our approach and Seminal for processing the reported examples. For each point (x, y) on the curve, it means that $x\%$ of all examples are processed in y seconds. The running time for our ap-

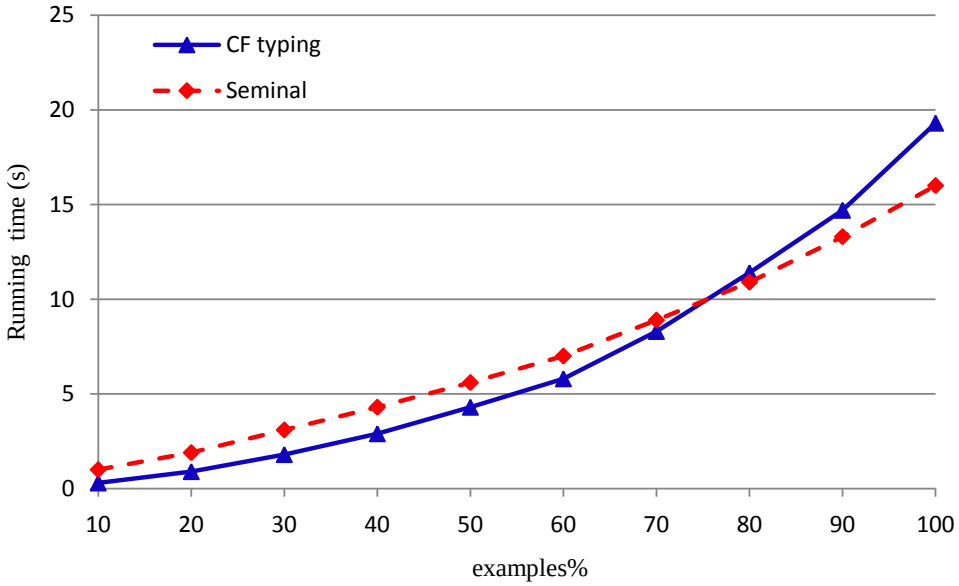


Figure 8.8: Running time for typing $x\%$ of the examples 10 times.

proach is measured on a laptop with a 2.8GHz dual core processor and 3GB RAM running Windows XP and GHC 7.0.2. The running time for Seminal is measured on the same machine with Cygwin 5.1. The purpose of the graph is simply to demonstrate the feasibility of our approach.

Second, we have evaluated how increasing levels of choice nestings affect the efficiency of the inference algorithm and how putting a limit on maximum nesting levels as described in Section 8.4 can regain efficiency at the cost of precision. For this purpose, we have automatically generated large examples, and we use functions of types like $a \to a \to \ldots \to a$ to trigger the choice elimination strategies discussed in Section 8.4. We first generated 200 type correct examples and then introduced one or two type errors in each example by changing the leaves,

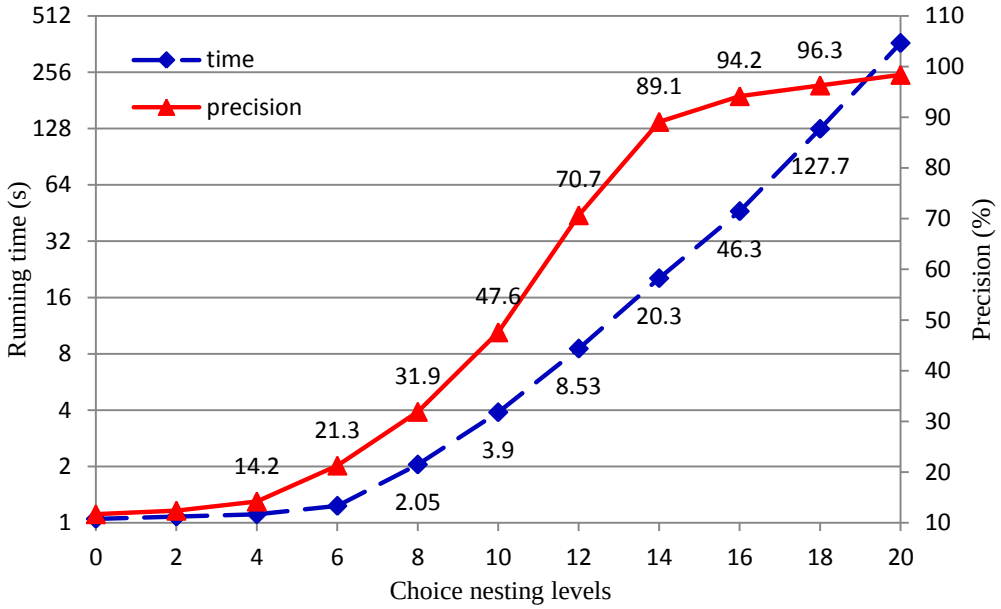


Figure 8.9: Limits on choice nesting trade efficiency for precision.

swapping arguments and so on. Each example contains about 60000 nodes in its abstract syntax tree representation.

Figure 8.9 presents the running time and precision against choice nesting levels for these generated examples. A change suggestion is considered correct if it fixes a type error and appears among the first four changes for that example. Precision is measured by dividing the number of examples that have correct change suggestions over the number of all examples. From the figure we observe that a nesting level cut-off between 12 and 18 achieves both high precision and efficiency.

8.7 Related Work

We presented a large body of work related to error debugging in Section 3.3. This section discusses the work that shares certain commonalities with counter-factual typing.

In deducing expression changes from type changes, we have used (an extension of) McAdam's technique [McAdam, 2002a]. Since his approach is based on the algorithm $\mathcal{W}$, it suffers from the bias of error locating mentioned above. Moreover, his approach doesn't have access to the precise expected type, which helps in our approach to ensure that deduced expression changes will not have an impact on the program as a whole.

Like error slicing approaches [Tip and Dinesh, 2001; Haack and Wells, 2003; Schilling, 2012], counter-factual typing is complete in not missing any potential change. However, the changes we present to users involve fewer locations. Usually, users have to focus on only one location. Another important difference is that our approach provides a change suggestion for each identified potential error source.

Counter-factual typing relies on the fact that each choice type can represent many types, allowing us to reason about many changes simultaneously. Similar to choice types, sum types can also encode many types. Neubauer and Thiemann [2003] developed a type system based on discriminative sum types to record the causes of type errors. Specifically, they place two non-unifiable types into a sum type. Technically, named choice types provide more fine-grained control over

variations in types than discriminative sum types. While sum types are unified component-wise, this is only the case for choice types of the same name. Each alternative in a choice type is unified with all the alternatives in other choices with different names. Also, their system returns a set of sources related to type errors. Thus, it can be viewed as an error slicing approach. Moreover, the approach doesn't provide specific change locations or change suggestions.

CF typing and Seminal [Lerner et al., 2006, 2007] could both be called "search based", although the search happens at different levels. While CF typing explores changes on the type level, Seminal works on the expression level directly, which makes it impossible for Seminal to generate a complete set of type-change suggestions. Given an ill-typed program, Seminal first has to decide where the type error is. Seminal uses a binary search to locate the erroneous place. This way of searching causes Seminal to make mistakes in locating errors when the first part of the program itself doesn't contain a type error but actually triggers type errors because it's too constrained. For example, the cause of the type error in the `palin` example discussed in Section 3.3 is the `fold` function, which is itself well typed. As a result, Seminal fails to find a correct suggestion.

Once the problematic expression is found, Seminal searches for a type-corrected program by creating mutations of the original program. For example, by swapping the arguments to functions, currying or uncurrying function calls, and so on. Compared to our change deduction approach, this has both advantages and disadvantages. In some cases, it can find a correct change while our approach fails to do so, as, for example, in the missing-parentheses problem discussed in

Section 8.6. On the other hand, its power to generate arbitrarily complicated changes can lead to bizarre suggestions, such as the suggestion to change `xs == (rev xs)` to `(==) (xs,(rev xs))`.

Chapter 9: Conclusions and Future Work

Besides the work presented in this dissertation, I have also explored other applications of choice types. I briefly present these applications in Section 9.1. Section 9.2 summarizes main contributions of my work at a high level and discusses directions for future work.

9.1 Other Applications

The concept of choice types has applications in a variety of areas. I will briefly discuss three of those, guided type debugging (Section 9.1.1), lazy typing (Section 9.1.2), and an analysis lifting framework (Section 9.1.3).

9.1.1 Guided Type Debugging

An important shortcoming with previous change-suggesting approaches, including our counter-factual typing approach discussed in Chapter 8, is that they focus solely on removing type errors but disregard users' intended result types. Consider, for example, the expression `e = foldl (:) []`. There is a type error in this expression because the type of `(:)` is `a->[a]->[a]`, but to make the program well-typed it should have the type `a->b->a`.

For this expression, Helium suggests to change `(:)` to `(++)`, which is perfect

when the intended type is `[[a]]->[a]`. However, it is very likely that the expression is intended to compute the reverse of the input list, and thus it should be of type `[a]->[a]`. In this case, the error message can't help. Worse, following the change suggestion will cause more problems in the expression.

Counter-factual typing suggests two potential changes to remove the type error. First, it suggests to change `(:)` to something of type `[a]->b->[a]`, so that the result type will be `[b]->[a]`. Second, it suggests to change `foldl` to something of type `(a->[a]->[a])->[b]->c`, so that the result type will be `c`. Given the expected type `[a]->b->[a]` of `(:)` and the result type `[b]->[a]` of `e`, it is still unclear about how to perform the change such that the result program has the type `[a] -> [a]`. The user has to solve the unification problem of `[b]->[a]` against `[a]->[a]` and derive that the type of `(:)` should be `[a]->a->[a]`. From there, the user has to further deduce to change `(:)` to `flip (:)`, which is again nontrivial.

We have developed a method, called *guided type debugging*, to automate this process [Chen and Erwig, 2014c]. Figure 9.1 presents two debugging sessions with guided type debugging for the expression `e`. We observe that the messages generated are guided by the user input.

To make the debugging process simple, guided type debugging only requires users to declaratively specify their intended result types, which is no harder than writing type annotations, something they do very often. Based on their intentions, we compute all erroneous locations and their expected types, rank them, and present the suggestions to users in that order.

Guided type debugging can improve the precision of suggesting type changes

```
What is the expected type of e?
[a] -> [a]

Potential fixes:
1  change (:) to (flip (:)).
2  change foldl
     from type (a -> b -> a) ->
               a -> [b] -> a
     to type (a -> [a] -> [a]) ->
               b -> [a] -> [a]
There are no other one-change fixes.
Show two-change fixes? (y/n)
```

```
What is the expected type of e?
[[a]] -> [a]

Potential fixes:
1  change (:)
     from type a -> [a] -> [a]
     to type [a] -> [a] -> [a]
2  change foldl
     from type (a -> b -> a) ->
               a -> [b] -> a
     to type (a -> [a] -> [a]) ->
               b -> [[a]] -> [a]
There are no other one-change fixes.
Show two-change fixes? (y/n)
```

Figure 9.1: Two examples of guided type debugging. The target type for `e` is `[a] -> [a]` (left) and `[[a]] -> [a]` (right). User inputs are shown in *italics*.

for ill-typed programs at a low cost. We have tested the method and compared it with CF typing on 86 programs, which were collected from 22 publications (see Section 8.6 for details). With guided type debugging we can now find the correct suggestions with the first attempt in 83% of the cases, which was 67% in CF typing. We can fix 90% of the cases with at most two attempts, an improvement of 10% over CF typing. At the same time guided type debugging adds never more than 0.5 seconds to the computing time.

9.1.2 Lazy Typing

Prior to counter-factual typing, we explored the idea of lazy typing [Chen and Erwig, 2014b]. We exploited the fact that choice types can represent uncertainty to delay typing decisions. Specifically, if branches (in `if` and `case` expressions)

have incompatible types, then we generally don't know which type to favor. By placing the conflicting types for the branches into a choice type and by continuing the typing process we can avoid a premature, uninformed decision and gather information about the context to decide which branch has the correct type. We then can report other branches as type incorrect.

As an example, consider the following function to compute Fibonacci numbers. This program contains a type error since the return types of case alternatives for the function `f` are different.

```
f x = case x of
       0 -> [0]
       1 -> 1

fib x = case x of
      0 -> f x
      1 -> f x
      n -> fib (n-1) + fib (n-2)
```

For this example, lazy typing reports the following error and corresponding type change suggestion.

```
(2,13): Type error in expression:
  [0]
Of type:  [Int]
Should have type:  Int
```

Lazy typing works the best when (1) at least one branch is compatible with the context, (2) the context consistently favors the same branch, and (3) there is enough information from the context to resolve the uncertainty in choice types.

Lazy typing and CF typing share the common idea of representing uncertainty about types using choice types. Other than that, the two approaches are fundamentally very different. First, CF typing generates a complete set of changes,

while lazy typing only tries to identify the most likely change. Moreover, CF typing suggests the expected type for the expression identified. Lazy typing only does this in some rare cases. Second, CF typing points to very specific change location while lazy typing has only branch-level granularity. Third, CF typing deduces expression-level changes, lazy typing works on the type level only.

Based on the observation that lazy typing and Helium [Heeren, 2005] perform well in different situations, we have looked into the question of combing both approaches to produce better error messages [Chen et al., 2014a]. By analyzing their respective strengths and weakness, we have identified a strategy of combing them to increase the precision of error reporting. Our evaluation of 1069 unique ill-typed programs out of a total of 11256 Haskell programs reveals that this combination strategy enjoys a correctness rate of 79%, which is an improvement of 22%/17% compared to using lazy typing/Helium alone.

9.1.3 Type-Based Parametric Analyses

We have also used choice types to develop a program analysis lifting framework that transforms type-based static analyses for plain programs into those for variational programs [Chen and Erwig, 2014a]. In contrast, most previous variational analyses are created by manually lifting analyses for plain programs. For example, the variability-aware module system [Kästner et al., 2012b] is the result of lifting the module system proposed by Cardelli [1997], the variability-aware dataflow analysis [Brabrand et al., 2012] is the result of lifting the traditional intraproce-

dural dataflow analysis [Nielson et al., 1999], and our variational type inference algorithm is the result of lifting the traditional inference algorithm $\mathcal{W}$ [Damas and Milner, 1982].

Each lifting consists of the following steps, all of which are nontrivial.

1. Introduce variability to data structures used in the traditional analysis to represent the value for a set of programs. In type checking feature-oriented product lines [Apel et al., 2010], plain types are expanded to sets of types. In variational dataflow analysis [Brabrand et al., 2012], value sets for the traditional dataflow analysis are expanded to functions from features to value sets. In our variational type inference algorithm, types are made variational through the use of choice types. The notion of choice is also adopted in [Kästner et al., 2012b] to perform variational type checking for C programs and in [Liebig et al., 2013, 2012] to conduct type checking and dataflow analysis for large-scale variational programs.

2. Adapt the traditional analysis to work with variationalized data structures. In variational type inference, typing rules, the unification algorithm, and the type inference algorithm are made variational. In variational model checking [Classen et al., 2010, 2011], symbolic encodings are made variational to deal with featured transition systems. In variational type checking [Kästner et al., 2012b] and variational dataflow analysis [Liebig et al., 2012], analyses are extended to deal with the variability introduced through choices.

3. Prove the correctness of the lifted analysis. Among different properties, the

most important one is that variation elimination commutes with the analysis. More specifically, if a variational analysis yields the variational result vr for the variational program vp, and if applying a selection to vp and vr yields the plain program p and the plain result r, then the plain analysis will produce r when applied to p. This property is presented and proved in most papers. A similar proof was absent in [Brabrand et al., 2012], but was later presented in [Brabrand et al., 2013].

4. Demonstrate the scalability of the lifted analysis by comparing its performance with the brute-force approach, which generates and analyzes each program individually. This part is present in almost all contributions.

This process is not only complicated, but also error prone. For example, the implementation of FFJ$_{PL}$, which is described in more depth in Section 3.1, contains a bug [Chen and Erwig, 2014a]. Instead, our analysis lifting framework transforms static analyses to those for variational programs with correctness guarantees and a bounded performance slowdown.

9.2 Main Contributions and Future Directions

This dissertation presents my work on variational typing and one of its applications. Compared to other more tool-oriented solutions, my work makes the following contributions.

1. It presents two type inference algorithms for solving the problem of typing

variational programs. The first algorithm works for variational programs when all encoded plain programs are well typed, and the second algorithm is a generalization of the first one and works for all variational programs. Experimental results show that both algorithms run exponentially faster than the brute-force strategy.

2. It studies the properties of the variational unification problem and presents a sound, complete, and most general unification algorithm. Similar contributions are also made for the partial variational unification problem. Both algorithms have many applications other than typing variational programs.

3. It presents solutions to many problems beyond typing variational programs, including increasing type safety guarantees of C++ Templates and improving the precision of type error debugging.

4. It creates a synergy between traditional analysis and variational analysis. Previous research has generalized traditional analyses to variational analyses. This thesis is the first to utilize techniques from variational analyses for traditional analyses, which led to improved type error debugging methods.

The work presented in this dissertation represents two very different lines of research. The goal of the first line is to improve the efficiency of activities related to creating, maintaining, and using variational programs. The second line, instead, focuses on using knowledge of variational analyses to improve the precision and usability of static analyses. Here are some possible directions for future

work, some are common to both lines of research while others are relevant to a particular line.

Both variational typing and its applications have focused on the Hindley-Milner type system so far. In recent years, we have seen many development of more advanced type system features, for example, qualified types, rank-N polymorphism [Peyton Jones et al., 2007], generalized algebraic data types [Vytiniotis et al., 2011], dependent types [Xi and Pfenning, 1999], and so on. It is interesting to investigate how choice types interact with them. This research has many practical implications. First, the heavy use of CPP macros has raised many concerns about useability and stability of Haskell code.[1] Extending the full-fledged Haskell type system with variations allows us to reason about the type correctness of Haskell libraries and improve their stability. Second, the introduction of these advanced features complicate the already difficulty problem of debugging type errors. It is interesting to see how counter-factual typing generalizes to type systems with these features.

While numerous approaches have been proposed to analyze and understand variational programs, the support for changing them is still primitive [Atkins, 1998; Kästner, 2010; Walkingshaw and Ostermann, 2014]. They either don't allow programmers to change views of certain configurations or have a limited strategy of propagating changes of views to source code. A potential application of variational typing here is to support more advanced strategies for propagating changes, for example, maximizing sharing while maintaining type correctness of all the

[1] https://mail.haskell.org/pipermail/haskell-cafe/2015-January/117683.html

plain programs.

While static analyses are good at checking whether programs satisfy certain properties, they are limited at telling where the sources of errors are and how to fix errors. It seems promising to generalize counter-factual typing to deliver more informative messages when static analyses fail. Research questions in this direction include locating parsing errors, fixing security violations, and blaming contract and effect violations.

Compared to other tool-oriented approaches targeting the problem of typing software product lines, my language-oriented solution is more fundamental. While practical approaches have the advantage of being immediately useful, a theoretical solution is likely to reveal the underlying principles of an area and have the benefit of applying these principles to solve problems in other areas. For example, without variational unification (Chapter 5) and error-tolerant variational type inference (Chapter 7), the work of counter-factual typing (Chapter 8) wouldn't have been possible. Beyond solving the problem of typing variational programs, the more fundamental contribution of this work is initiating a new research area for improving usability of static analyses through variational analyses.

Bibliography

I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, 2014.

D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

S. Anantharaman, P. Narendran, and M. Rusinowitch. Unification Modulo ACUI Plus Homomorphisms/Distributivity. *Journal of Automated Reasoning*, 33:1–28, 2004.

S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM Trans. Program. Lang. Syst.*, 32(5):19:1–19:33, May 2008. ISSN 0164-0925.

S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

S. Apel and C. Kästner. An overview of feature-oriented software development, July/August 2009. URL http://www.jot.fm/issues/issue_2009_07/column5/index.html. Refereed Column.

S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Int. Conf. on Generative Programming and Component Engineering*, pages 101–112, 2008.

S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, Sept. 2010.

S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Software Engineering*, 2013. To appear.

D. Atkins. Version sensitive editing: Change history as a programming tool. In *System Configuration Management*, pages 146–157. 1998.

M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman, 1998.

L. Aversano, M. D. Penta, and I. D. Baxter. Handling Preprocessor-Conditioned Declarations. In *Int. Workshop on Source Code Analysis and Manipulation*, pages 83–92, 2002.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

F. Baader and W. Snyder. Unification Theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. North Holland, 2001.

V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional Normalisation and Type-Directed Partial Evaluation for Typed Lambda Calculus with Sums. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 64–76, 2004.

H. Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309, 1992.

D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.*, volume 3714 of *LNCS*, pages 7–20, 2005.

D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2:17–30, 1994.

K. L. Bernstein and E. W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, 1995.

C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Int. Conf. on Aspect-Oriented Software Development*, AOSD '12, pages 13–24, 2012.

C. Brabrand, M. Ribeiro, T. Tolĺždo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X*, pages 73–108. 2013.

G. Bracha and W. Cook. Mixin-Based Inheritance. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 303–311, 1990.

B. Braßel. Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*, 2004.

L. Cardelli. Program fragments, linking, and modularization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 1997.

W. Chae and M. Blume. Building a Family of Compilers. In *Int. Software Product Line Conf.*, SPLC '08, pages 307–316, 2008.

S. Chen and M. Erwig. Type-based parametric analysis of program families. In *ACM SIGPLAN International Conference on Functional Programming*, pages 39–51, 2014a.

S. Chen and M. Erwig. Better Type-Error Messages Through Lazy Typing. Technical Report, Oregon State University, 2014b. URL http://web.engr.oregonstate.edu/~chensh/Docs/tr-lt.pdf.

S. Chen and M. Erwig. Guided type debugging. In *Functional and Logic Programming*, LNCS 8475, pages 35–51. 2014c.

S. Chen and M. Erwig. Early Detection of Type Errors in C++ Templates. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 133–144, 2014d.

S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 583–594, 2014e.

S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, pages 29–40, 2012.

S. Chen, M. Erwig, and K. Smeltzer. Let's Hear Both Sides: On Combining Type-Error Reporting Tools. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 145–152, 2014a.

S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *ACM Trans. Program. Lang. Syst.*, 36(1):1:1–1:54, Mar. 2014b.

O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ACM Int. Conf. on Functional Programming*, pages 193–204, September 2001.

V. Choppella. *Unification Source-Tracking with Application To Diagnosis of Type Inference*. PhD thesis, Indiana University, 2002.

A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, ICSE '10, pages 335–344, 2010.

A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, ICSE '11, pages 321–330, 2011.

M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-based Abstractions for Software Product-Line Model Checking. In *IEEE Int. Conf. on Software Engineering*, ICSE 2012, pages 672–682, 2012.

K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212, 1982.

M. Davis. *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Courier Corporation, 2004.

B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Workshop on Foundations of Aspect-Oriented Languages*, pages 31–35, 2009a.

B. Delaware, W. R. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 243–252, 2009b.

B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 595–608, 2011.

X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 10:1–10:7, 2014.

M. Dezani-Ciancaglini, S. Ghilezan, and B. Venneri. The "Relevance" of Intersection and Union Types. *Notre Dame Journal of Formal Logic*, 38(2):246–269, 1997.

C. Disenfeld and S. Katz. A Closer Look at Aspect Interference and Cooperation. In *Int. Conf. on Aspect-Oriented Software Development*, AOSD '12, pages 107–118, 2012.

G. Dos Reis and B. Stroustrup. Specifying C++ Concepts. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 295–308, 2006.

P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpression Problem. *J. ACM*, 27(4):758–771, Oct. 1980.

D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1995.

N. El Boustani and J. 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, 2010.

N. El Boustani and J. Hage. Improving type error messages for generic java. *Higher-Order and Symbolic Computation*, 24(1-2):3–39, 2011. ISSN 1388-3690.

T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. of the ACM*, 44(10):28–32, 2001.

H. Eo, O. Lee, and K. Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004.

M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, Dec. 2011.

M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, LNCS 7680, pages 55–100, 2013.

M. Fähndrich, M. Carbin, and J. R. Larus. Reflective Program Generation with Patterns. In *Int. Conf. on Generative Programming and Component Engineering*, GPCE '06, pages 275–284, 2006.

S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: Indexed Types Now! In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 112–121, 2007.

R. Garcia. *Static Computation and Reflection*. PhD thesis, Indiana University, September 2008.

R. Garcia and A. Lumsdaine. Toward Foundations for Type-Reflective Metaprogramming. In *Int. Conf. on Generative Programming and Component Engineering*, GPCE '09, pages 25–34, 2009.

P. Gazzillo and R. Grimm. SuperC: Parsing all of C by Taming the Preprocessor. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*, PLDI '12, pages 323–334, 2012.

GNU Project. *The C Preprocessor*. Free Software Foundation, 2009. http://gcc.gnu.org/onlinedocs/cpp/.

M. Goldman, E. Katz, and S. Katz. Maven: Modular Aspect Verification and Interference Analysis. *Formal Methods in System Design*, 37(1):61–92, 2010.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, 2006.

C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003. ISBN 3-540-00886-1.

J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. 2007.

J. Hage and B. Heeren. Strategies for solving constraints in type and effect systems. *Electronic Notes in Theoretical Computer Science*, 236:163–183, 2009.

B. Heeren and J. Hage. Type class directives. In *Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 253–267. 2005.

B. Heeren, J. Hage, and S. D. Swierstra. Constraint based type inferencing in helium. *Immediate Applications of Constraint Programming (ACP)*, pages 59–80, 2003a.

B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 3–13, 2003b.

B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 62–71, 2003c.

B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.

J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

S. S. Huang and Y. Smaragdakis. Expressive and Safe Static Reflection with MorphJ. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*, PLDI '08, pages 79–89, 2008.

S. S. Huang and Y. Smaragdakis. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Transactions on Programming Languages and Systems*, 33:6:1–6:44, 2011.

S. S. Huang, D. Zook, and Y. Smaragdakis. Statically Safe Program Generation with Safegen. In *Int. Conf. on Generative Programming and Component Engineering*, GPCE'05, pages 309–326, 2005.

S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *Int. Conf. on Aspect-Oriented Software Development*, pages 185–198, 2007.

J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*, PLDI '06, pages 272–282, 2006.

P. Johann and N. Ghani. Foundations for structured programming with gadts. pages 297–308, 2008.

G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986.

K. Kagawa. Polymorphic Variants in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 37–47, 2006.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

C. Kästner. Virtual separation of concerns: Toward preprocessors 2.0, 5 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9.

C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 10 2011.

C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21(3): 14:1–14:39, July 2012a. ISSN 1049-331X.

C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 773–792, 2012b.

A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Workshop on Feature-Oriented Software Development*, pages 25–32, 2010.

C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Int. Conf. on Generative Programming and Component Engineering*, pages 19–23, 2008.

D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient Inference of Partial Types. In *Journal of Computer and System Sciences*, pages 363–371, 1992.

O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. on Programming Languages and Systems*, 20(4):707–723, July 1998.

O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *Workshop on ML*, pages 63–73, 2006.

B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*, pages 425–434, 2007.

J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Large-Scale Variability-Aware Type Checking and Dataflow Analysis. Number MIP-1212. 2012.

J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Foundations of Software Engineering*, pages 81–91, 2013.

B. J. McAdam. *Reporting Type Errors in Functional Programs*. PhD thesis, Larboratory for Foundations of Computer Science, The University of Edinburgh, 2002a.

B. J. McAdam. *Repairing type errors in functional programs*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2002b.

M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Int. Conf. on Aspect-Oriented Software Development*, pages 90–99, 2003.

M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6): 127–136, 2004.

W. Miao and J. G. Siek. Incremental Type-checking for Type-Reflective Metaprograms. In *Int. Conf. on Generative Programming and Component Engineering*, GPCE '10, pages 167–176, 2010.

S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 111–120, 2013.

G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, Apr. 1980.

M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*, pages 15–26, 2003.

M. Neubauer and P. Thiemann. Haskell type browser. In *ACM SIGPLAN Workshop on Haskell*, pages 92–93, 2004.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Sceince and Engineering, Chalmers University of Technology and Göteborg University, 2007.

M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

P. M. O'Keefe and M. Wand. Type Inference for Partial Types is Decidable. In *European Symposium on Programming*, pages 408–417, 1992.

C. Paulin-Mohring. Inductive Definitions in the System Coq Rules and Properties. *Typed Lambda Calculi and Applications*, pages 328–345, 1993.

Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, 2014.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, Jan. 2007.

B. C. Pierce. Bounded Quantification with Bottom. Technical report, Computer Science Department, Indiana University, 1997.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

G. D. Reis and B. Stroustrup. A Formalism for C++. Technical report, ISO/IEC SC22/JTC1/WG21, October 2005.

J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

T. Schilling. Constraint-free type error slicing. In *Trends in Functional Programming*, pages 1–16. Springer, 2012.

P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *IEEE Int. Requirements Engineering Conf.*, pages 139–148, 2006.

T. Sheard and S. Peyton Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.

M. Shields, T. Sheard, and S. Peyton Jones. Dynamic Typing as Staged Type Inference. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 289–302, 1998.

J. Siek and W. Taha. A Semantic Analysis of C++ Templates. In *European Conf. on Object-Oriented Programming*, ECOOP'06, pages 304–327, 2006a.

J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006b.

J. G. Siek and M. Vachharajani. Gradual Typing with Unification-Based Inference. In *Symp. on Dynamic Languages*, pages 7:1–7:12, 2008.

B. Stroustrup. *The Design and Evolution of C++*. ACM Press, 1994.

P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 72–83, 2003.

M. Sulzmann. A General Type Inference Framework for Hindley/Milner Style Systems. In *Int. Symp. on Functional and Logic Programming*, pages 248–263, 2001.

M. Sulzmann and P. j. Stuckey. Hm(x) type inference is clp(x) solving. *J. Funct. Program.*, 18:251–283, March 2008.

W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Int. Conf. on Generative Programming and Component Engineering*, pages 95–104, 2007.

S. Thatte. Type Inference with Partial Types. In *Int. Colloq. on Automata, Languages and Programming*, pages 615–629, 1988.

T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *International Conference on Generative Programming and Component Engineering*, pages 11–20, 2012.

T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. 47(1):6:1–6:45, 6 2014.

F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, Jan. 2001.

D. Vytiniotis, S. Peyton jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, Sept. 2011.

E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, June 2014.

E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 29–38, 2014.

M. Wand. Finding the Source of Type Errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986.

A. Warth, M. Stanojević, and T. Millstein. Statically Scoped Object Adaptation with Expanders. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 37–56, 2006.

J. R. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The University of Melbourne, January 2006.

J. Weijers, J. Hage, and S. Holdermans. Security type error diagnosis for higher-order, polymorphic languages. *Science of Computer Programming*, 95:200–218, 2014.

H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, 1999.

J. Yang. Explaining type errors by finding the source of a type conflict. In *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

J. Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, May 2001.

J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Int. Workshop on Implementation of Functional Languages*, pages 71–86, 2000.

D. Zhang and A. C. Myers. Toward General Diagnosis of Static Errors. In *ACM Symp. on Principles of Programming Languages*, pages 569–581, 2014.

APPENDICES

Appendix A: Proofs for Chapter 4

Lemma 2 (Type equivalence preservation) *If* $\phi_1 \equiv \phi_2$*, then* $\lfloor \phi_1 \rfloor_s \equiv \lfloor \phi_2 \rfloor_s$.

PROOF. The proof of this lemma proceeds by case, demonstrating that for each equivalence rule defined in Figure 4.4, if we apply the same selector s to both the LHS and the RHS of the rule, the resulting expressions are still equivalent. We demonstrate this for only a few cases; but the other cases can be treated similarly.

First, we consider the F-C rule. There are two sub-cases to consider: Either the dimension of the choice type matches that of the selector, or it does not. We consider the sub-case where the dimension name does not match first.

$$
\begin{array}{rll}
\lfloor D\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle \rfloor_s &= D\langle \lfloor \phi_1 \rfloor_s \to \lfloor \phi_1' \rfloor_s, \lfloor \phi_2 \rfloor_s \to \lfloor \phi_2' \rfloor_s \rangle & \text{selection in LHS} \\
&\equiv D\langle \lfloor \phi_1 \rfloor_s, \lfloor \phi_2 \rfloor_s \rangle \to D\langle \lfloor \phi_1' \rfloor_s, \lfloor \phi_2' \rfloor_s \rangle & \text{by the rule F-C} \\
\lfloor D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_1', \phi_2' \rangle \rfloor_s &= \lfloor D\langle \phi_1, \phi_2 \rangle \rfloor_s \to \lfloor D\langle \phi_1', \phi_2' \rangle \rfloor_s & \text{selection in RHS} \\
&= D\langle \lfloor \phi_1 \rfloor_s, \lfloor \phi_2 \rfloor_s \rangle \to D\langle \lfloor \phi_1' \rfloor_s, \lfloor \phi_2' \rfloor_s \rangle & \text{by definition}
\end{array}
$$

For the sub-case where the dimension name matches, there are two further sub-cases, depending on whether we are selecting the first or second alternatives in dimension D. Below we show the case for $s = \tilde{D}$ (selecting the second alternatives). The case for $s = D$ is dual to this.

$$
\begin{array}{rll}
\lfloor D\langle \phi_1 \to \phi_1', \phi_2 \to \phi_2' \rangle \rfloor_{\tilde{D}} &= \lfloor \phi_2 \to \phi_2' \rfloor_{\tilde{D}} & \text{selection in LHS} \\
&= \lfloor \phi_2 \rfloor_{\tilde{D}} \to \lfloor \phi_2 \rfloor_{\tilde{D}} & \text{by definition} \\
\lfloor D\langle \phi_1, \phi_2 \rangle \to D\langle \phi_1', \phi_2' \rangle \rfloor_{\tilde{D}} &= \lfloor D\langle \phi_1, \phi_2 \rangle \rfloor_{\tilde{D}} \to \lfloor D\langle \phi_1', \phi_2' \rangle \rfloor_{\tilde{D}} & \text{selection in RHS} \\
&= \lfloor \phi_2 \rfloor_{\tilde{D}} \to \lfloor \phi_2' \rfloor_{\tilde{D}} & \text{by definition}
\end{array}
$$

Next we consider the C-C-SWAP2 rule. Here there are three cases to consider: s makes a selection in dimension D, in dimension D', or in some other dimension. The case for when s makes a selection in D follows.

$$\begin{aligned}
\lfloor D'\langle \phi_1, D\langle \phi_2, \phi_3\rangle\rangle \rfloor_{\tilde{D}} &= D'\langle \lfloor \phi_1 \rfloor_{\tilde{D}}, \lfloor D\langle \phi_2, \phi_3\rangle \rfloor_{\tilde{D}}\rangle && \text{selection in LHS} \\
&= D'\langle \lfloor \phi_1 \rfloor_{\tilde{D}}, \lfloor \phi_3 \rfloor_{\tilde{D}}\rangle && \text{by definition} \\
\lfloor D\langle D'\langle \phi_1, \phi_2\rangle, D'\langle \phi_1, \phi_3\rangle\rangle \rfloor_{\tilde{D}} &= \lfloor D'\langle \phi_1, \phi_3\rangle \rfloor_{\tilde{D}} && \text{selection in RHS} \\
&= D'\langle \lfloor \phi_1 \rfloor_{\tilde{D}}, \lfloor \phi_3 \rfloor_{\tilde{D}}\rangle && \text{by definition}
\end{aligned}$$

The second case is a dual to this, and the third case can be proved in a similar way as the first case for the F-C rule. The proofs of the remaining rules proceed in a similar fashion. □

Lemma 3 (Local confluence) *For any type ϕ, if $\phi \rightsquigarrow \phi_1$ and $\phi \rightsquigarrow \phi_2$, then there exists some type ϕ' such that $\phi_1 \rightsquigarrow^* \phi'$ and $\phi_2 \rightsquigarrow^* \phi'$.*

The proof requires the ability to address specific *positions* in a variational type. A position p is given by a path from the root of the type to a particular node, where a path is represented by a sequence of values L and R, indicating whether to enter the left or right branch of a function or choice type. The root type is addressed by the empty path ϵ. We use $\phi|_p$ to refer to the type at position p in type ϕ. For example, given $\phi = \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle$, we can refer to the component types of ϕ in

the following way.

$$\begin{aligned}\phi|_\epsilon &= \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle \\ \phi|_L &= \texttt{Int} \\ \phi|_R &= A\langle \texttt{Bool}, \texttt{Int}\rangle \\ \phi|_{RL} &= \texttt{Bool} \\ \phi|_{RR} &= \texttt{Int}\end{aligned}$$

We use $\phi[\phi']_p$ to indicate the substitution of type ϕ' at position p in type ϕ. For example, given the same ϕ as above, $\phi[\texttt{Bool}]_R = \texttt{Int} \to \texttt{Bool}$. We use $\mathscr{P}(\phi)$ to refer to the set of all positions in ϕ.

We also need a way to abstractly represent the application of a simplification rule. We use $l \rightsquigarrow r$ to represent an arbitrary simplification rule from Figure 4.5. We represent applying that rule somewhere in type ϕ by giving a position p and a substitution ξ indicating how to instantiate it. Before we apply the rule, it must be the case that $\phi|_p = \xi(l)$. The result of applying the rule will be $\phi[\xi(r)]_p$. As example, given $\phi = \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Bool}\rangle$, we can apply the S-C-IDEMP rule ($l = A\langle x, x\rangle$, $r = x$) at $p = R$ with the substitution $\xi = \{x \mapsto \texttt{Bool}\}$, resulting in $\phi' = \texttt{Int} \to \texttt{Bool}$ (note that we assume the dimension name in the simplification rule is instantiated automatically).

PROOF. Given type ϕ, assume that some rewrite rule $l_1 \rightsquigarrow r_1$ can be applied at position p_1 with substitution ξ_1, and another rewrite rule $l_2 \rightsquigarrow r_2$ can be applied at position p_2 with substitution ξ_2. Then $\phi_1 = \phi[\xi_1(r_1)]_{p_1}$ and $\phi_2 = \phi[\xi_2(r_2)]_{p_2}$. Then we must show that there is always a ϕ' such that $\phi_1 \rightsquigarrow^* \phi'$ and $\phi_2 \rightsquigarrow^* \phi'$. There are three cases to consider.

First, the two simplifications are *parallel*. This occurs when neither p_1 or p_2 is

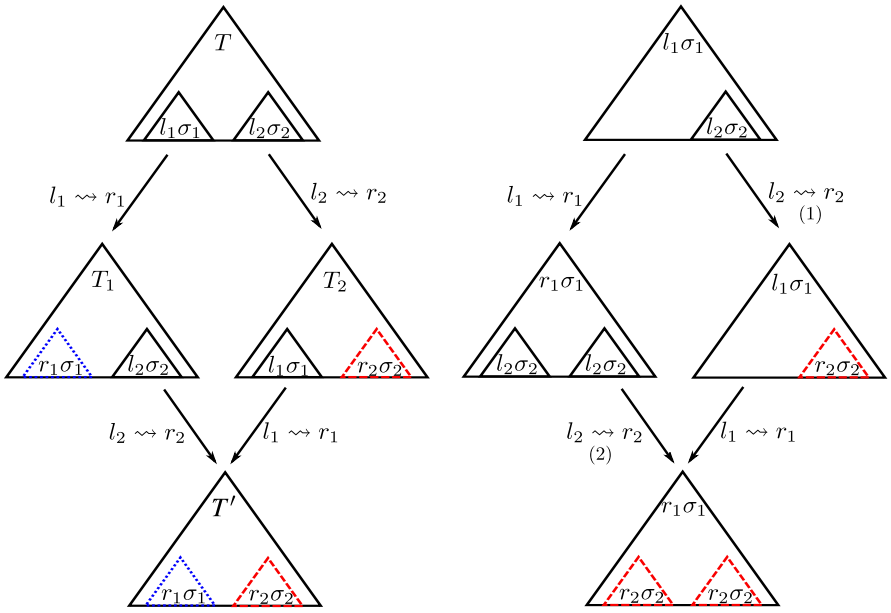


Figure A.1: Proof of local confluence.

a prefix of the other. It represents simplifications that are in different parts of the type and therefore independent. The proof for this case is shown in the left graph in Figure A.1. If we apply $l_1 \rightsquigarrow r_1$ first, we can reach ϕ' by next applying $l_2 \rightsquigarrow r_2$, and vice versa. This situation is encountered, for example, when we must choose between the S-F-ARG and S-F-RES rules. Intuitively, it does not matter whether we first simplify the argument type or result type of a function type. The situation is also encountered when choosing between the S-C-DOM1 and S-C-DOM2 rules, and the S-C-ALT1 and S-C-ALT2 rules.

Second, one simplification may *contain* the other. This occurs when p_1 is a prefix of p_2 and $\xi_2(l_2)$ is contained in the range of ξ_1 (the case where p_2 is a prefix

of p_1 is dual). For example, consider the following type ϕ.

$$\phi = A\langle B\langle \texttt{Int}, \texttt{Bool}\rangle, C\langle \texttt{Int}, \texttt{Int}\rangle\rangle$$

We can apply the S-C-SWAP1 rule, $A\langle B\langle x, y\rangle, z\rangle \rightsquigarrow A\langle B\langle x, z\rangle, B\langle y, z\rangle\rangle$, at position $p_1 = \epsilon$ with the substitution $\xi_1 = \{x \mapsto \texttt{Int}, y \mapsto \texttt{Bool}, z \mapsto C\langle \texttt{Int}, \texttt{Int}\rangle\}$. Or we can apply the S-C-IDEMP rule, $C\langle w, w\rangle \rightsquigarrow w$, at position $p_2 = R$ with the substitution $\xi_2 = \{w \mapsto \texttt{Int}\}$. Note that $\xi_2(l_2) = D\langle \texttt{Int}, \texttt{Int}\rangle$, which is in the range of ξ_1. The proof for this example is illustrated in the right graph of Figure A.1, the labels (1) and (2) indicate the number of times the associated rule must be applied. In general, if the variable at $l_1|_{p_2}$ occurs m times in l_1 and n times in r_1, then we need to apply the $l_2 \rightsquigarrow r_2$ rule n times in the left branch of the graph and m times in the right branch. Intuitively, this case arises when the simplifications are conceptually independent, but one is nested within the other. If we apply the outer simplification first, it may increase or decrease the number of times we must apply the inner one (and vice versa). Many combinations of rules can lead to this situation.

Third, the simplifications may *critically overlap*. This occurs when p_1 is a prefix of p_2 and there is some $p \in \mathscr{P}(l_1)$ such that $l_1|_p$ is not a variable and $\xi_1(l_1|_p) = \xi_2(l_2)$. For example, consider the following type ϕ.

$$\phi = C\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, B\langle \texttt{Bool}, \texttt{Int}\rangle\rangle$$

Then both S-C-SWAP1 and S-C-SWAP2 are applicable at $p_1 = p_2 = \epsilon$. To prove that our choice between these rules doesn't matter, we need to compute the critical pairs between the two rules and decide the joinability of all such critical pairs

[Baader and Nipkow, 1998]. If all critical pairs are joinable, then the two rules are locally confluent, otherwise they are not. The critical pairs are computed as follows. Given any $p \in \mathscr{P}(l_1)$ such that $l_1|_p$ is not a type variable, compute the mgu for $l_1|_p \equiv^? l_2$ as ξ, then $\xi(r_1)$ and $\xi(l_1)[\xi(r_2)]_p$ form a critical pair. We show the proof process for the two S-C-SWAP rules below.

First, we rewrite these rules in the following way, so they do not share any type variables. (We also instantiate the dimension names and eliminate the premises.)

S-C-SWAP1 $\quad C\langle A\langle x,y\rangle,z\rangle \rightsquigarrow A\langle C\langle x,z\rangle,C\langle y,z\rangle\rangle$

S-C-SWAP2 $\quad C\langle k_5,B\langle k_4,k_6\rangle\rangle \rightsquigarrow B\langle C\langle k_5,k_4\rangle,C\langle k_5,k_6\rangle\rangle$

When $p=\epsilon$, the unification problem is $C\langle A\langle x,y\rangle,z\rangle \equiv^? C\langle k_5,B\langle k_4,k_6\rangle\rangle$. The computed mgu ξ is given below, where all previously undefined type variables are fresh.

$$\xi = \{x \mapsto C\langle A\langle k_5,b\rangle,c\rangle, y \mapsto C\langle A\langle d,k_5\rangle,k_1\rangle, z \mapsto C\langle k_2,B\langle k_4,k_6\rangle\rangle\}$$

The critical pair consists of the following two types.

1. $A\langle C\langle C\langle A\langle k_5,b\rangle,c\rangle,C\langle k_2,B\langle k_4,k_6\rangle\rangle\rangle,C\langle C\langle A\langle d,k_5\rangle,k_1\rangle,C\langle k_2,B\langle k_4,k_6\rangle\rangle\rangle\rangle$
2. $B\langle C\langle k_5,k_4\rangle,C\langle k_5,k_6\rangle\rangle$

This pair is joinable by simplifying the first component of the pair into the second, as demonstrated below.

$$\begin{aligned}
&A\langle C\langle C\langle A\langle k_5,b\rangle,c\rangle,C\langle k_2,B\langle k_4,k_6\rangle\rangle\rangle,\\
&\quad C\langle C\langle A\langle d,k_5\rangle,k_1\rangle,C\langle k_2,B\langle k_4,k_6\rangle\rangle\rangle\\
&= A\langle C\langle k_5,B\langle k_4,k_6\rangle\rangle,C\langle k_5,B\langle k_4,k_6\rangle\rangle\rangle && \text{S-C-DOM1 and S-C-DOM2}\\
&= C\langle k_5,B\langle k_4,k_6\rangle\rangle && \text{S-C-IDEMP}\\
&= B\langle C\langle k_5,k_4\rangle,C\langle k_5,k_6\rangle\rangle && \text{S-C-SWAP2}
\end{aligned}$$

When $p = L$, the unification problem is $A\langle x, y\rangle \equiv^? C\langle k_5, B\langle k_4, k_6\rangle\rangle$, and the computed mgu ξ is given below.

$$\xi = \{x \mapsto A\langle C\langle k_5, B\langle k_4, k_6\rangle, a\rangle\rangle, y \mapsto A\langle b, C\langle k_5, B\langle k_4, k_6\rangle\rangle\rangle\}$$

The critical pair consists of the following two types.

1. $A\langle C\langle A\langle C\langle k_5, B\langle k_4, k_6\rangle, a\rangle\rangle, z\rangle, C\langle A\langle b, C\langle k_5, B\langle k_4, k_6\rangle\rangle\rangle, z\rangle\rangle$
2. $C\langle B\langle C\langle k_5, k_4\rangle, C\langle k_5, k_6\rangle\rangle, z\rangle$

This pair is joinable by simplifying both components into the type $C\langle k_5, z\rangle$, as demonstrated below.

$$\begin{array}{ll}
& A\langle C\langle A\langle C\langle k_5, B\langle k_4, k_6\rangle, a\rangle\rangle, z\rangle, \\
& \quad C\langle A\langle b, C\langle k_5, B\langle k_4, k_6\rangle\rangle\rangle, z\rangle \\
= & A\langle C\langle k_5, z\rangle, C\langle k_5, z\rangle\rangle \qquad \text{S-C-DOM1 and S-C-DOM2} \\
= & C\langle k_5, z\rangle \qquad \text{S-C-IDEMP} \\
\hline
& C\langle B\langle C\langle k_5, k_4\rangle, C\langle k_5, k_6\rangle\rangle, z\rangle \\
= & C\langle B\langle k_5, k_5\rangle, z\rangle \qquad \text{S-C-DOM1 and S-C-DOM2} \\
= & C\langle k_5, z\rangle \qquad \text{S-C-IDEMP}
\end{array}$$

The proofs for other critically overlapping rules can be constructed similarly. □

Lemma 4 (Termination) *Given any type ϕ, $\phi \rightsquigarrow^* \phi'$ is terminating.*

PROOF. To support this proof, we use a tuple $(ad, (ur, ra), dd, ic)$ to measure how normalized a type is. When this tuple is $(0, (0, 0), 0, 0)$, the type is fully normalized and no rule in Figure 4.5 can be applied. Otherwise, the type is not fully normalized and some rule applies. We then divide all the rules in Figure 4.5 into four groups and show that the first group decreases ad, the second group decreases the

pair (ur, ra) without increasing ad, the third group decreases dd without increasing ad or (ur, ra), and the fourth group decreases ic without increasing any other component of the tuple.

The components of $(ad, (ur, ra), dd, ic)$ are defined as follows.

(1) The component ad denotes the number of choice types that are nested directly or indirectly in arrow types. For example, $ad = 3$ for the type $A\langle B\langle \tau_1, \tau_2\rangle, \tau_3\rangle \to C\langle \tau_4, \tau_5\rangle$.

(2) The pair (ur, ra) captures nested choice types that violate the ordering constraint $\preceq$ on dimension names. The component ur denotes the *unique* pairs of inverted dimension names, while ra is the *total* number of nested choices that violate this constraint. There are some subtleties to computing these values, which are described below.

(3) The component dd denotes the number of dead alternatives—alternatives that cannot be selected because of choice domination.

(4) Finally, ic denotes number of idempotent choice types—choice types where both alternatives are the same.

To compute the pair (ur, ra), we cannot just count the number of inverted choice types directly since the process of hoisting can mask intermediate progress if represented in this way. For example, given the type $\phi = C\langle B\langle \phi_1, \phi_2\rangle, A\langle \phi_3, \phi_4\rangle\rangle$, an application of S-C-SWAP1 yields $\phi' = B\langle C\langle \phi_1, A\langle \phi_3, \phi_4\rangle\rangle, C\langle \phi_2, A\langle \phi_3, \phi_4\rangle\rangle\rangle$, in which the choice type in B has been correctly hoisted above the choice type in C. However, ϕ contains just 2 inverted dimension names ($C \not\preceq B$ and $C \not\preceq A$) while ϕ'

contains 2 unique inversions ($B \not\leq A$ and $C \not\leq A$) and 4 total inversions (since both unique inversions appear twice). Thus, the naive measure fails to capture the progress made transforming ϕ to ϕ' and in fact suggests that we regressed despite having resolved one of the original inversions. Therefore, we instead compute ur as (a) the number of unique inversions, plus (b) the binomial coefficient $\binom{n}{2}$ where n is the number of unique inversions involving the root choice type. This additional component captures the combinations of inversions that could be created during the hoisting process. We compute ra in the same way but without limiting ourselves to unique inversions. Using this metric, the transformation of ϕ to ϕ' reduces (ur, ra) from $(3,3)$ to $(2,4)$.

Now we divide the rules in Figure 4.5 into four groups. Since the rules S-F-ARG, S-F-RES, S-C-ALT1 and S-C-ALT2 are congruence rules, they have no direct effect on the normalization metric. We divide the remaining rules as follows.

1. The first group contains the rules S-F-C-ARG and S-F-C-RES. Whenever one of these rules is applied, an arrow type is pushed into a choice type, reducing ad.

2. The second group contains the rules S-C-SWAP1 and S-C-SWAP2. Applying these rules will not increase ad since swapping choices will not generate new choice types and will not push choice types into arrow types. Applying these rules will either decrease ur or leave ur unchanged and decrease ra. For example, applying S-C-SWAP1 to $\phi = C\langle B\langle\cdots\rangle, A\langle\cdots\rangle\rangle$ leads to the type

$\phi' = B\langle C\langle\cdots, A\langle\cdots\rangle\rangle, C\langle\cdots, A\langle\cdots\rangle\rangle\rangle$. Assume that when computing *ur* for ϕ, parts (a) and (b) are n_1 and n_2, respectively, where $n_2 = \binom{n_3}{2}$. This means that n_1 is the total number of unique inversions in ϕ and n_3 is the number of unique inversions involving dimension C. Similarly, when computing *ur* for ϕ', assume parts (a) and (b) are n'_1 and n'_2, respectively, where $n'_2 = \binom{n'_3}{2}$. Then n'_1 can be computed as follows:

$n'_1 = -1$	previously $C \not\preceq B$, now B is hoisted out
$+ n_{23}$	number of choices nested in A that were inverted with B
$+ n_1$	remaining nestings are unchanged

Since $B \preceq C$, the choices inverted with C in ϕ may be no longer inverted with B in ϕ'. Thus, $n'_3 \leq n_3 - 1$ since B is no longer nested in C. Therefore, $n'_2 \leq \binom{n_3}{2}$. Moreover, since there are n_3 choices in ϕ inverted with C, there are no more than $n_3 - 1$ choices in A that are inverted with C. Combining this with the fact that $B \preceq C$, there are then fewer than $n_3 - 1$ choices that are inverted with C in ϕ'. So we have $n_{23} \leq n_3 - 1$.

The change of *ur* from ϕ to ϕ', denoted as ρ, can be computed as follows,

$$\begin{aligned}
\rho &= n_1 + n_2 - n'_1 - n'_2 \\
&\geq n_1 + C^2_{n_3} - n'_1 - C^2_{n_3-1} \\
&= (n_1 - n_1 + 1 - n_{23}) + (C^2_{n_3} - C^2_{n_3-1}) \\
&= (1 - n_{23}) + (n_3 - 1) \\
&= n_3 - n_{23} \\
&\geq 1
\end{aligned}$$

Thus, after hoisting, *ur* decreases at least by 1. The proof for the case that

$n_2 \leq 2$ is simple and is omitted here.

When ϕ is a part of a larger type, then there are two cases. First, if the larger type does not have B nested in C elsewhere, then it is clear that ur will decrease by at least 1 for the larger type. Otherwise, if B is nested in C elsewhere, then ur may stay the same. However, we can prove that ra decreases at least by 1 similarly to the case for ur shown in detail above. Intuitively, ra decreases because swapping B out of C has removed the reversed pair between C and B.

3. The third group includes the rules S-C-DOM1 and S-C-DOM2. Applying the rules in this group will not increase ad since they don't create new choices and they don't push down choice types into arrows. Applying them also doesn't increase (ur, ra) since choice orderings are not swapped. Whenever one of the two rules is applied, at least one dead alternative is removed. Thus, the rules will only decrease dd.

4. The fourth and final group contains the rule S-C-IDEMP. Applying this rule will remove a choice whose alternatives are the same, therefore it may decrease ad, (ur, ra), or dd, but it can never increase these values. Whenever this rule is applied, at least one idempotent choice will be eliminated. Therefore, it strictly decreases ic.

If we define an ordering relation on $(ad, (ur, ra), dd, ic)$ based on the ordering relation of each component, where the components are ordered from most-significant (ad) to least (ic), then we have demonstrated that the metric $(ad, (ur, ra), dd, ic)$ is

strictly decreasing by applying the simplification rules. This process terminates when $(ad,(ur,ra),dd,ic)$ reaches $(0,(0,0),0,0)$, completing the proof. □

Appendix B: Proofs for Chapter 5

Theorem 5 *Given a CT-unification problem U, there is a unifier ξ such that for any unifier ξ', there exists a mapping θ such that $\xi' = \theta \circ \xi$.*

In the following, we use U and Q to denote unification problems. We use ϕ_L and ϕ_R to denote the LHS and RHS of U, and ϕ'_L and ϕ'_R to denote the LHS and RHS of Q. We use $FV(U)$ to refer to all of the type variables in U. We also extend the notion of selection to unification problems and mappings by propagating the selection to the types they contain, as defined below.

$$\lfloor \phi_L \equiv^? \phi_R \rfloor_s = \lfloor \phi_L \rfloor_s \equiv^? \lfloor \phi_R \rfloor_s$$
$$\lfloor \xi \rfloor_s = \{(a, \lfloor \phi \rfloor_s) \mid (a, \phi) \in \xi\}$$

The following lemma states that selection further extends over type substitution in a homomorphic way.

Lemma 21 (Selection extends over substitution) $\lfloor \xi(\phi) \rfloor_s = \lfloor \xi \rfloor_s(\lfloor \phi \rfloor_s)$

PROOF of Lemma 21. The proof is based on induction over the structure of ϕ and ξ. We show the proof only for the most interesting cases where ϕ is a choice type, and where ϕ is a type variable mapped to a choice type in ξ.

1. Given $\phi = D\langle\phi_1, \phi_2\rangle$, assume $s = \tilde{D}$ (the case for $s = D$ is dual).

$$\begin{aligned}
\lfloor\xi(\phi)\rfloor_s &= \lfloor\xi(D\langle\phi_1,\phi_2\rangle)\rfloor_{\tilde{D}} && \text{by definition}\\
&= \lfloor D\langle\xi(\phi_1),\xi(\phi_2)\rangle\rfloor_{\tilde{D}} && \text{type substitution}\\
&= \lfloor\xi(\phi_2)\rfloor_{\tilde{D}} && \text{selection}\\
&= \lfloor\xi\rfloor_{\tilde{D}}(\lfloor\phi_2\rfloor_{\tilde{D}}) && \text{induction hypopethesis}\\
&= \lfloor\xi\rfloor_{\tilde{D}}(\lfloor D\langle\phi_1,\phi_2\rangle\rfloor_{\tilde{D}}) && \text{selection}\\
&= \lfloor\xi\rfloor_s(\lfloor\phi\rfloor_s)
\end{aligned}$$

2. Given $\phi = a$, assume $\xi(a) = D\langle\phi_1, \phi_2\rangle$ and $s = \tilde{D}$ (again $s = D$ is dual). Given ξ, we write $\xi[a = \phi]$ to denote that ξ maps a to ϕ.

$$\begin{aligned}
\lfloor\xi\rfloor_s(\lfloor\phi\rfloor_s) &= \lfloor\xi\rfloor_{\tilde{D}}(\lfloor a\rfloor_{\tilde{D}}) && \text{by definition}\\
&= \lfloor\xi\rfloor_{\tilde{D}}[a = \lfloor\phi_2\rfloor_{\tilde{D}}](a) && \text{selection}\\
&= \lfloor\phi_2\rfloor_{\tilde{D}}\\
\lfloor\xi(\phi)\rfloor_s &= \lfloor\xi(a)\rfloor_{\tilde{D}} && \text{by definition}\\
&= \lfloor D\langle\phi_1,\phi_2\rangle\rfloor_{\tilde{D}} && \text{by assumption}\\
&= \lfloor\phi_2\rfloor_{\tilde{D}} = \lfloor\xi\rfloor_s(\lfloor\phi\rfloor_s)
\end{aligned}$$

The remaining cases can be constructed similarly. □

From Lemma 21, it follows by induction that the same result holds for decisions as for single selectors: $\lfloor\xi(\phi)\rfloor_\delta = \lfloor\xi\rfloor_\delta(\lfloor\phi\rfloor_\delta)$. Combining this with Lemma 2 (selection preserves type equivalence), we see that if ξ is a unifier for U, then $\lfloor\xi(\phi_L)\rfloor_\delta \equiv \lfloor\xi(\phi_R)\rfloor_\delta$ for any δ. This is the same as saying that if $\xi(\phi_L) \not\equiv \xi(\phi_R)$, then $\exists\delta : \lfloor\xi(\phi_L)\rfloor_\delta \not\equiv \lfloor\xi(\phi_R)\rfloor_\delta$. A direct consequence of this result is that if δ is super-complete (it eliminates all choice types in ϕ_L, ϕ_R, and ξ) and $\lfloor\xi\rfloor_\delta(\lfloor\phi_L\rfloor_\delta) \equiv \lfloor\xi\rfloor_\delta(\lfloor\phi_R\rfloor_\delta)$, then ξ is a unifier for U.

PROOF of Theorem 5. Using the type splitting algorithm described in Section 5.2, we can transform U into Q such that for all super-complete decisions $\delta_1, \delta_2, \cdots, \delta_n$,

if $\delta_i \neq \delta_j$, then $FV(\lfloor Q \rfloor_{\delta_i}) \cap FV(\lfloor Q \rfloor_{\delta_j}) = \varnothing$.

Each subproblem $\lfloor Q \rfloor_{\delta_i}$ corresponding to a super-complete decision δ_i is plain. Therefore, we can obtain (via Robinson's algorithm) an mgu ξ_i such that $\lfloor \xi_i(\phi'_L) \rfloor_{\delta_i} \equiv \lfloor \xi_i(\phi'_R) \rfloor_{\delta_i}$. Let ξ be the disjoint union of all of these mgus, that is, $\xi = \bigcup_{i \in \{1..n\}} \xi_i$.

Since the type variables in each subproblem are different, for each subproblem we have $\lfloor \xi(\phi'_L) \rfloor_{\delta_i} \equiv \lfloor \xi(\phi'_R) \rfloor_{\delta_i}$. Then based on the discussion after Lemma 21, ξ is a unifier for $\phi'_L \equiv^?_q \phi'_R$. Moreover, it is most general by construction since each ξ_i is most general. Based on Theorem 8 (variational unification is sound) and Lemma 10 (*comp* is correct and preserves principality), the completion of ξ is the mgu for U, which proves that variational unification is unitary. □

Theorem 6 (Soundness) *If* $vunify(\phi_1, \phi_2) = \xi$, *then* $\xi(\phi_1) \equiv \xi(\phi_2)$.

PROOF. The proof is by induction on the structure of ϕ_1 and ϕ_2. To make the proof easier to follow, we step through each case of the *vunify* algorithm, briefly describing why the theorem holds for each base case, or why it is preserved for recursive cases. For many cases, correctness is preserved by *vunify* being recursively invoked on semantically equivalent arguments.

1. Both types are plain. The result is determined by the Robinson unification algorithm, which is known to be correct [Robinson, 1965].

2. A qualified type variable a_q and a choice type. Correctness is preserved since $a_q \equiv D\langle a_{Dq}, a_{\bar{D}q} \rangle$.

3. Two choice types in the same dimension. Decomposition by alternatives is correct by the inductive hypothesis and Lemma 7.

4. The next three cases consider choice types in different dimensions. They preserve correctness for the following reasons.

   (a) Hoisting is semantics preserving.

   (b) Splitting is variable independent by Lemma 8.

   (c) Splitting is choice independent by Lemma 9.

5. The next two cases consider unifying a choice type with a non-choice type. Correctness is preserved in both cases since the recursive calls are on semantically equivalent arguments, by choice idempotency.

6. Two function types. Given the inductive hypotheses, $\xi(\phi_1) \equiv \xi(\phi_1')$ and $\xi(\phi_2) \equiv \xi(\phi_2')$, we can construct $\xi((\phi_1 \to \phi_2)) \equiv \xi((\phi_1' \to \phi_2'))$ by an application of the FUN equivalence rule in Figure 4.4.

7. The last case considers a qualified type variable a_q and a function type $\phi \to \phi'$. If the occurs check fails, the theorem is trivially satisfied since the condition of the implication is not met. If it succeeds, the theorem is satisfied by the definition of substitution.

□

Appendix C: Proofs for Chapter 8

Theorem 18 (Type-change inference is consistent) *For any given e and Γ, if $\Gamma \vdash e : \phi | \Delta$ and there is some τ such that $\lfloor \phi \rfloor_{\Delta.2} = \tau$, then $\Gamma; \downarrow \Delta \vdash^C e : \tau$.*

PROOF. We consider two different cases. In the first case, $\Delta = \varnothing$. According to the definition of $\Downarrow$, this indicates that no changes have been made in the typing process for $\Gamma \vdash e : \phi | \Delta$. In this case, the assumption $\lfloor \phi \rfloor_{\Delta.2} = \tau$ simplifies to $\phi = \tau$. Thus, the theorem itself simplifies to $\Gamma \vdash e : \tau | \varnothing \Rightarrow \Gamma; \varnothing \vdash^C e : \tau$. This holds trivially since both type systems are the same as HM when no changes happen.

In the second case, $\Delta \neq \varnothing$. This indicates that changes happened during the typing process. We prove this case through a structural induction over the typing relation defined in Figure 8.3.

Case CON: We have $e = v$ and $\Gamma \vdash v : D\langle \gamma, \tau \rangle | \{(\ell(v), D\langle \gamma, \tau \rangle)\}$ for a fresh choice D since alternatives of choices are plain. The type update in this case is

$$\omega = \downarrow \Delta = \downarrow \{(\ell(v), D\langle \gamma, \tau \rangle)\} = \{v \mapsto \tau\}.$$

Our goal is to prove $\Gamma; \omega \vdash^C v : \tau$. Based on the structure of e, the only rule that applies in Figure 8.5 is CON-C. The proof follows directly as ω changes v to τ.

Case VAR: The proof for this case is very similar to that for case CON and is omitted here. Note that the instantiation of ϕ_1 in rule VAR is irrelevant as the

instantiation is overridden by the change.

Case APP: We need to show that

$$\Gamma \vdash e_1\ e_2 : \phi | \Delta \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \lfloor \phi \rfloor_{\Delta.2}$$

with the following induction hypotheses.

$$\Gamma \vdash e_1 : \phi_1 | \Delta_1 \Rightarrow \Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$$
$$\Gamma \vdash e_2 : \phi_2 | \Delta_2 \Rightarrow \Gamma; \downarrow\Delta_2 \vdash^C e_2 : \lfloor \phi_2 \rfloor_{\Delta_2.2}$$

Additionally, ϕ is computed by ϕ_1 and ϕ_2 through a use of the APP rule. Let $\uparrow(\phi_1) = \phi_{1l} \rightarrow \phi_{1r}$, then we have the following relation.

$$\begin{aligned}
\tau &= \lfloor \phi \rfloor_{\Delta.2} \\
&= \lfloor \phi_{1l} \bowtie \phi_2 \lhd \phi_{1r} \rfloor_{\Delta.2} \\
&= \lfloor \phi_{1l} \rfloor_{\Delta.2} \bowtie \lfloor \phi_2 \rfloor_{\Delta.2} \lhd \lfloor \phi_{1r} \rfloor_{\Delta.2} && \text{By Lemma 14}
\end{aligned}$$

According to the definitions of $\bowtie$ and $\lhd$ (see Section 7.1), we have

$$\lfloor \phi_{1l} \rfloor_{\Delta.2} = \lfloor \phi_2 \rfloor_{\Delta.2} \tag{C.1}$$
$$\lfloor \phi_{1r} \rfloor_{\Delta.2} = \tau. \tag{C.2}$$

Combing the fact that $\uparrow(\phi_1) = \phi_{1l} \rightarrow \phi_{1r}$ with the equations (C.1) and (C.2), we have $\lfloor \uparrow(\phi_1) \rfloor_{\Delta.2} = \lfloor \phi_2 \rfloor_{\Delta.2} \rightarrow \tau$. Based on the definition of $\uparrow$, we have the following relation.

$$\lfloor \phi_1 \rfloor_{\Delta.2} = \lfloor \phi_2 \rfloor_{\Delta.2} \rightarrow \tau \tag{C.3}$$

From $\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$ because compared to Δ_1, Δ contains additional information that only has impact on typing e_2. From $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta_1.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor \phi_1 \rfloor_{\Delta.2}$ because $\lfloor \phi_1 \rfloor_{\Delta_1.2}$ al-

ready yields a plain type, and expanding the decision $\Delta_1.2$ to $\Delta.2$ will not change the result. Overall, we have

$$\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor\phi_1\rfloor_{\Delta_1.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor\phi_1\rfloor_{\Delta.2} \tag{C.4}$$

Similarly, we have

$$\Gamma; \downarrow\Delta_2 \vdash^C e_2 : \lfloor\phi_2\rfloor_{\Delta_2.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_2 : \lfloor\phi_2\rfloor_{\Delta.2} \tag{C.5}$$

From equations (C.3) and (C.4), we have

$$\Gamma; \downarrow\Delta_1 \vdash^C e_1 : \lfloor\phi_1\rfloor_{\Delta_1.2} \Rightarrow \Gamma; \downarrow\Delta \vdash^C e_1 : \lfloor\phi_2\rfloor_{\Delta.2} \rightarrow \tau \tag{C.6}$$

From equations (C.5) and (C.6), the typing rule APP-C, and the induction hypotheses, we have $\Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \tau$. Since $\tau = \lfloor\phi\rfloor_{\Delta.2}$, we have $\Gamma; \downarrow\Delta \vdash^C e_1\ e_2 : \lfloor\phi\rfloor_{\Delta.2}$, completing the proof.

The case for the rule ABS is straightforward and those for rules LET and IF are similar to that for rule APP and are therefore omitted here. □

Theorem 19 (Type-change inference is complete) *For any e, Γ and ω, if $\Gamma; \omega \vdash^C e : \tau$, then there exist ϕ, Δ, and a typing derivation for $\Gamma \vdash e : \phi|\Delta$ such that $\downarrow\Delta = \omega$ and $\lfloor\phi\rfloor_{\Delta.2} = \tau$.*

PROOF. Again, we consider two cases. In the first case, $\omega = \varnothing$, which means no changes have been applied. Thus, $\Gamma; \varnothing \vdash^C e : \tau$ implies that the expression e under Γ is well typed. When typing using the rules in Figure 8.3, we make the second alternative the same as the first alternative for all choices. In other words, we create idempotent choices only. The theorem holds trivially in this case.

In the second case, $\omega \neq \varnothing$. We prove the lemma through a structural induction over the typing relation in Figure 8.5.

Case CON-C: We have $e = v$. Since $\omega \neq \varnothing$, it must be of the form $\omega = \{v \mapsto \tau\}$. We construct the typing relation as $\Gamma \vdash v : D\langle\gamma,\tau\rangle | \{(\ell(v), D\langle\gamma,\tau\rangle)\}$, where γ is the type of v and D is a fresh choice. It's simple to verify that $\downarrow\{(\ell(v), D\langle\gamma,\tau\rangle)\} = \{v \mapsto \tau\}$ and $\lfloor D\langle\gamma,\tau\rangle\rfloor_{\tilde{D}} = \tau$.

Case VAR-C: The proof of this case is the same as that for case CON-C and is omitted here. The only difference is that we are dealing with a variable reference rather than a constant.

Case APP-C: We need to show that

$$\Gamma;\omega \vdash^C e_1\ e_2 : \tau \Rightarrow \Gamma \vdash e_1\ e_2 : \phi|\Delta \text{ with } \downarrow\Delta = \omega \text{ and } \lfloor\phi\rfloor_{\Delta.2} = \tau$$

with the following induction hypotheses.

$$\Gamma;\omega \vdash^C e_1 : \tau_1 \to \tau \Rightarrow \Gamma \vdash e_1 : \phi_1|\Delta \text{ with } \downarrow\Delta = \omega \text{ and } \lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$$
$$\Gamma;\omega \vdash^C e_2 : \tau_1 \Rightarrow \Gamma \vdash e_2 : \phi_2|\Delta \text{ with } \downarrow\Delta = \omega \text{ and } \lfloor\phi_2\rfloor_{\Delta.2} = \tau_1$$

We can split Δ into Δ_1 and Δ_2 such that they contain the change information for e_1 and e_2, respectively. With that, we have the following relations.

$$\Gamma;\omega \vdash^C e_1 : \tau_1 \to \tau \Rightarrow \Gamma \vdash e_1 : \phi_1|\Delta_1 \text{ with } \lfloor\phi_1\rfloor_{\Delta_1.2} = \tau_1 \to \tau$$
$$\Gamma;\omega \vdash^C e_2 : \tau_1 \Rightarrow \Gamma \vdash e_2 : \phi_2|\Delta_2 \text{ with } \lfloor\phi_2\rfloor_{\Delta_2.2} = \tau_1$$

Since $\Delta_1.2 \subseteq \Delta.2$ and $\lfloor\phi_1\rfloor_{\Delta_1.2} = \tau_1 \to \tau$, we have $\lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$. We have a similar result regarding ϕ_2. Based on them, we arrive at the following

relations.

$$\Gamma;\omega \vdash^{C} e_1 : \tau_1 \to \tau \Rightarrow \Gamma \vdash e_1 : \phi_1|\Delta_1 \text{ with } \lfloor\phi_1\rfloor_{\Delta.2} = \tau_1 \to \tau$$
$$\Gamma;\omega \vdash^{C} e_2 : \tau_1 \Rightarrow \Gamma \vdash e_2 : \phi_2|\Delta_2 \text{ with } \lfloor\phi_2\rfloor_{\Delta.2} = \tau_1$$

Let $\uparrow(\phi_1) = \phi_{1l} \to \phi_{1r}$. We compute $\lfloor\phi\rfloor_{\Delta.2}$ as follows.

$$\begin{aligned}
\lfloor\phi\rfloor_{\Delta.2} &= \lfloor\phi_{1l} \bowtie \phi_2 \lhd \phi_{1r}\rfloor_{\Delta.2} \\
&= \lfloor\phi_{1l}\rfloor_{\Delta.2} \bowtie \lfloor\phi_2\rfloor_{\Delta.2} \lhd \lfloor\phi_{1r}\rfloor_{\Delta.2} && \text{By Lemma 14} \\
&= \tau_1 \bowtie \tau_1 \lhd \tau && \text{By relation between } \lfloor\phi_1\rfloor_{\Delta.2}, \phi_{1l}, \text{and } \phi_{1r} \\
&= \top \lhd \tau \\
&= \tau
\end{aligned}$$

This shows that $\lfloor\phi\rfloor_{\Delta.2} = \tau$. Based on induction hypotheses, $\downarrow\Delta = \omega$, completing the proof.

We can prove the cases for other rules in Figure 8.5 similarly. □

Lemma 19 (Most defined type changes) *Given e and Γ and two typings $\Gamma \vdash e : \phi_1|\Delta_1$ and $\Gamma \vdash e : \phi_2|\Delta_2$, if $\lfloor\phi_1\rfloor_\delta = \bot$ and $\lfloor\phi_2\rfloor_\delta = \tau$, then there is a typing $\Gamma \vdash e : \phi_3|\Delta_3$ such that*

- $\lfloor\phi_3\rfloor_\delta = \lfloor\phi_2\rfloor_\delta$ *and for all other* δ' $\lfloor\phi_3\rfloor_{\delta'} = \lfloor\phi_1\rfloor_{\delta'}$.
- $\downarrow_\delta\Delta_3 = \downarrow_\delta\Delta_2$ *and* $\downarrow_{\delta'}\Delta_3 = \downarrow_{\delta'}\Delta_1$ *for all other* δ'.

PROOF. We construct a typing $\Gamma \vdash e : \phi_3|\Delta_3$ so that both items in the lemma are satisfied. To construct a typing using the rules in Figure 8.3, we need to designate the second alternatives of the freshly created choices and the mappings for instantiating type schemas. Once they are decided, the typing is determined for the given e and Γ. The construction is based on a structural induction over the typing

relation in Figure 8.3. We prove a stronger lemma by dropping the conditions that $\lfloor \phi_1 \rfloor_\delta = \bot$ and $\lfloor \phi_2 \rfloor_\delta = \tau$.

Case CON: We need to further consider two sub-cases. Assume $\phi_1 = D\langle \gamma, \phi_1' \rangle$ and $\phi_2 = D\langle \gamma, \phi_2' \rangle$, where γ is the type of the constant v. Note that ϕ_1 and ϕ_2 determine the contents of Δ_1 and Δ_2, respectively.

(1) $\tilde{D} \in \delta$. Let $\phi_3 = D\langle \gamma, \phi_3' \rangle$ where $\phi_3' = expand(\delta, \phi_1', \phi_2')$. Here $expand(\delta, \phi_1', \phi_2')$ builds a type ϕ such that $\lfloor \phi \rfloor_\delta = \phi_2'$ and $\lfloor \phi \rfloor_{\delta'} = \phi_1'$ for all other δ'. This function is formally defined as follows.

$$
\begin{aligned}
expand(D\delta, \phi_1', \phi_2') &= D\langle expand(\delta, \phi_1', \phi_2'), \phi_1' \rangle \\
expand(\tilde{D}\delta, \phi_1', \phi_2') &= D\langle \phi_1', expand(\delta, \phi_1', \phi_2') \rangle \\
expand(D, \phi_1', \phi_2') &= D\langle \phi_2', \phi_1' \rangle \\
expand(\tilde{D}, \phi_1', \phi_2') &= D\langle \phi_1', \phi_2' \rangle
\end{aligned}
$$

Given a decision, we write $s\delta$ to single out an arbitrary selector s from that decision and bind the remaining to δ.

With the constructed ϕ_3 we can verify that $\lfloor \phi_3 \rfloor_\delta = \lfloor \phi_3' \rfloor_\delta = \phi_2' = \lfloor \phi_2 \rfloor_\delta$ and $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_3' \rfloor_{\delta'} = \phi_1' = \lfloor \phi_1 \rfloor_{\delta'}$ for all other δ'. Since ϕ_3 determines Δ_3, verifying the second item of the lemma follows directly from that of the first item.

(2) $\tilde{D} \notin \delta$. Let $\phi_3 = \phi_1$. When $\tilde{D} \notin \delta$, we have both $\lfloor \delta \rfloor_{\phi_1} = \gamma$ and $\lfloor \delta \rfloor_{\phi_2} = \gamma$, which means that the change doesn't affect the decision δ. Thus, we don't need to change the type of v. We can verify that $\lfloor \phi_3 \rfloor_\delta = v = \lfloor \phi_2 \rfloor_\delta$ and $\lfloor \phi_3 \rfloor_{\delta'} = v = \lfloor \phi_1 \rfloor_{\delta'}$ for all other δ'.

Case VAR: The proof is similar to that of case CON and is omitted here.

Case APP: For this case we don't need to construct anything but only have to show that the lemma is preserved over applying the APP rule. The induction hypotheses are

$$\Gamma \vdash e_1 : \phi_{11} | \Delta_{11} \quad \Gamma \vdash e_2 : \phi_{12} | \Delta_{12} \quad \Gamma \vdash e_1\ e_2 : \phi_1 | \Delta_1$$
$$\Gamma \vdash e_1 : \phi_{21} | \Delta_{21} \quad \Gamma \vdash e_2 : \phi_{22} | \Delta_{22} \quad \Gamma \vdash e_1\ e_2 : \phi_2 | \Delta_2$$
$$\Gamma \vdash e_1 : \phi_{31} | \Delta_{31} \quad \Gamma \vdash e_2 : \phi_{32} | \Delta_{32} \quad \Gamma \vdash e_1\ e_2 : \phi_3 | \Delta_3$$
$$\lfloor \phi_{31} \rfloor_\delta = \lfloor \phi_{21} \rfloor_\delta \quad \lfloor \phi_{31} \rfloor_{\delta'} = \lfloor \phi_{11} \rfloor_{\delta'} \quad \downarrow_\delta \Delta_{31} = \downarrow_\delta \Delta_{21} \quad \downarrow_{\delta'} \Delta_{31} = \downarrow_{\delta'} \Delta_{11}$$
$$\lfloor \phi_{32} \rfloor_\delta = \lfloor \phi_{22} \rfloor_\delta \quad \lfloor \phi_{32} \rfloor_{\delta'} = \lfloor \phi_{12} \rfloor_{\delta'} \quad \downarrow_\delta \Delta_{32} = \downarrow_\delta \Delta_{22} \quad \downarrow_{\delta'} \Delta_{32} = \downarrow_{\delta'} \Delta_{12}$$

We need to show that

$$\lfloor \phi_3 \rfloor_\delta = \lfloor \phi_2 \rfloor_\delta \quad \lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'} \quad \downarrow_\delta \Delta_3 = \downarrow_\delta \Delta_2 \quad \downarrow_{\delta'} \Delta_3 = \downarrow_{\delta'} \Delta_1$$

In the following we show $\lfloor \phi_3 \rfloor_\delta = \lfloor \phi_2 \rfloor_\delta$ with the assumptions that $\uparrow(\phi_{31}) = \phi_{31l} \to \phi_{31r}$ and $\uparrow(\phi_{21}) = \phi_{21l} \to \phi_{21r}$.

$$\begin{aligned}
\lfloor \phi_3 \rfloor_\delta &= \lfloor \phi_{31l} \bowtie \phi_{32} \lhd \phi_{31r} \rfloor_\delta \\
&= \lfloor \phi_{31l} \rfloor_\delta \bowtie \lfloor \phi_{32} \rfloor_\delta \lhd \lfloor \phi_{31r} \rfloor_\delta && \text{By Lemma 14} \\
&= \lfloor \phi_{31l} \rfloor_\delta \bowtie \lfloor \phi_{22} \rfloor_\delta \lhd \lfloor \phi_{31r} \rfloor_\delta && \text{By induction hypothesis} \\
&= \lfloor \phi_{21l} \rfloor_\delta \bowtie \lfloor \phi_{22} \rfloor_\delta \lhd \lfloor \phi_{21r} \rfloor_\delta && \text{By Lemma 14 and induction hypothesis} \\
&= \lfloor \phi_{21l} \bowtie \phi_{22} \lhd \phi_{21r} \rfloor_\delta && \text{By Lemma 14} \\
&= \lfloor \phi_2 \rfloor_\delta
\end{aligned}$$

We can prove $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$ similarly. Since $\Delta_3 = \Delta_{31} \cup \Delta_{32} = \Delta_{21} \cup \Delta_{22} = \Delta_2$, we have $\downarrow_\delta \Delta_3 = \downarrow_\delta \Delta_2$.

The proofs for other cases are very similar to the case APP and are omitted here.

□

We use an example to illustrate the proof process. We consider constructing the new typing for the example expression $e = \texttt{not (succ 5)}$ under the following typings.

$$\Gamma \vdash e : \phi_1 | \Delta_1 \quad \phi_1 = A\langle \bot, a_1 \rangle \quad \Delta_1 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to a_1 \rangle) \; (\ell(\texttt{succ}), B\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Int} \rangle)\}$$

$$\Gamma \vdash e : \phi_2 | \Delta_2 \quad \phi_2 = B\langle \bot, \texttt{Bool} \rangle \quad \Delta_2 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool} \rangle) \; (\ell(\texttt{succ}), B\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool} \rangle)\}$$

We consider $\delta = \{A, \tilde{B}\}$ and observe that $\lfloor \phi_1 \rfloor_\delta = \bot$ and $\lfloor \phi_2 \rfloor_\delta = \texttt{Bool}$. We construct $\Gamma \vdash e : \phi_3 | \Delta_3$ as follows. For `not`, the choice created is A. Since $\tilde{A} \notin \{A, \tilde{B}\}$, the type for `not` is $A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to a_1 \rangle$, the type for `not` in Δ_1. For `succ`, the created choice is B. Since $\tilde{B} \in \{A, \tilde{B}\}$, the type of `succ`, written as $\phi_{\texttt{succ}}$, can be computed as follows.

$$\begin{aligned}
\phi_{\texttt{succ}} &= B\langle \texttt{Int} \to \texttt{Int}, \mathit{expand}(\{A, \tilde{B}\}, \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}) \rangle \\
&= B\langle \texttt{Int} \to \texttt{Int}, A\langle \mathit{expand}(\tilde{B}, \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}), \texttt{Int} \to \texttt{Int} \rangle \rangle \\
&= B\langle \texttt{Int} \to \texttt{Int}, A\langle B\langle \texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool} \rangle, \texttt{Int} \to \texttt{Int} \rangle \rangle \\
&= B\langle \texttt{Int} \to \texttt{Int}, A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int} \rangle \rangle
\end{aligned}$$

The type for `5` is always `Int`. Now that we have specified types for `not`, `succ`, and `5`, we compute $\phi_3 = A\langle B\langle \bot, \texttt{Bool} \rangle, a_1 \rangle$. The content for Δ_3 is easy to construct, and we omit it here. It is easy to verify that $\lfloor \phi_3 \rfloor_{\{A,\tilde{B}\}} = \texttt{Bool} = \lfloor \phi_2 \rfloor_{\{A,\tilde{B}\}}$ and that for all other δ' we have $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$. We can verify that the relation given in the lemma holds for Δ_1, Δ_2, and Δ_3.

Lemma 20 (Generalizability of type changes) *For any two typings* $\Gamma \vdash e : \phi_1 | \Delta_1$ *and* $\Gamma \vdash e : \phi_2 | \Delta_2$*, if neither* $\lfloor \phi_1 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$ *nor* $\lfloor \phi_2 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta$ *holds, there is a typing* $\Gamma \vdash e : \phi_3 | \Delta_3$ *such that*

- $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta$, $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$ *and for all other* δ', $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$.
- $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_1$, $\downarrow_\delta \Delta_3 \leq \downarrow_\delta \Delta_2$ *and for all other* δ', $\downarrow_{\delta'} \Delta_3 = \downarrow_{\delta'} \Delta_1$.

The proof strategy is similar to that for proving Lemma 19. However, there is a subtlety here compared to that proof. In proving Lemma 19 we take something directly from the second typing and merge it into the first to get the third without any changes. This strategy is insufficient here. We use two examples to illustrate the problem, both regarding the expression `not 1`.

In the first example, we remove the type error with the following potential typings.

$$\Gamma \vdash \texttt{not 1} : \phi_1 | \Delta_1 \qquad \phi_1 = A\langle \bot, \texttt{Int} \rangle \qquad \Delta_1 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int} \rangle)\}$$

$$\Gamma \vdash \texttt{not 1} : \phi_2 | \Delta_2 \qquad \phi_2 = A\langle \bot, \texttt{Bool} \rangle \qquad \Delta_2 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Bool} \rangle)\}$$

We observe that these typings have different result types and neither is more general than the other. We know that there must exist a third typing that gives a more general result type. The type that is more general than both `Int` and `Bool` is a type variable, say a. We can achieve this result type by changing `not` to something of type $\texttt{Int} \to a$. This change can be derived by looking at the types that `not` is changed to in both typings. The type of `not` is changed to $\texttt{Int} \to \texttt{Int}$ and $\texttt{Int} \to \texttt{Bool}$, respectively. A type that is more general than the both is $\texttt{Int} \to a$.

In general, however, we need to accommodate the impact of changing the type for some subexpression on the typing of the whole expression. In the second example, we use the following typings to remove the type error in `not 1`.

$$\Gamma \vdash \texttt{not 1} : \phi_1 | \Delta_1 \quad \phi_1 = A\langle \bot, \texttt{Int} \rangle \quad \Delta_1 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int} \rangle), (\ell(\texttt{1}), B\langle \texttt{Int}, \texttt{Int} \rangle)\}$$

$$\Gamma \vdash \texttt{not 1} : \phi_2 | \Delta_2 \quad \phi_2 = B\langle \bot, \texttt{Bool} \rangle \quad \Delta_2 = \{(\ell(\texttt{not}), A\langle \texttt{Bool} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool} \rangle) (\ell(\texttt{1}), B\langle \texttt{Int}, \texttt{Bool} \rangle)\}$$

We consider $\delta = \{\tilde{A}, \tilde{B}\}$ and observe that $\lfloor \phi_1 \rfloor_\delta = \texttt{Int}$ and $\lfloor \phi_2 \rfloor_\delta = \texttt{Bool}$. Now in order to get a more general typing for the expression with a being the result type, we need to change the types assigned to subexpressions `not` and `1`. First, how should we change the type for `not`? Since these two typings assign it $\texttt{Int} \to \texttt{Int}$ and $\texttt{Bool} \to \texttt{Bool}$, respectively, a more general type is of the form $a_1 \to a$. In other words, we assign $a_1 \to a$ to `not`. Now how about the type for `1`? These two typings change it to `Int` and `Bool`, respectively. We may be tempted to assign it an arbitrarily more general type, for example a_3. However, this change will make `not 1` ill typed because the domain type of the function, a_1, doesn't match the type of the argument, a_3. We should instead assign `1` the type a_1, taking the fact that `not` has changed to $a_1 \to a$ into account.

In summary, while we need to generalize the types for certain subexpressions during the construction process, we should perform it consistently among all subexpressions. To simplify the presentation, we assume that there exists a function $postgen_e(l, \tau)$ that returns the type for the location l in e after generalization. We usually omit the subscript when the context makes it clear what e is. When no constraints have been seen so far for l in e, $postgen_e(l, \tau)$ returns a type that has the same structure as τ except that primitive types are replaced by fresh type variables. Otherwise, it returns the type that satisfies the type constraints

among subexpressions. For example, consider again $e = \texttt{not 1}$. For e, we have $postgen(\ell(\texttt{not}), \texttt{Int} \to \texttt{Int}) = a_1 \to a_2$. Since we haven't seen any constraints for `not`, we assign a fresh type variable to each `Int`. For `1`, we need to consider the constraint between `not` and `1`, and we have $postgen(\ell(\texttt{1}), \texttt{Int}) = a_1$. In static typing, the constraints among all subexpressions are easy to derive. Thus, the function $postgen_e(l, \tau)$ is easy to compute and the definition is omitted here.

PROOF of Lemma 20. The proof is by constructing a new typing based on the given typings so that both conditions of the lemma are satisfied.

Case CON: Assume $e = v$, $\phi_1 = D\langle\gamma, \phi_1'\rangle$, and $\phi_2 = D\langle\gamma, \phi_2'\rangle$, where γ is the type of v. We further consider two sub-cases.

(a) $\tilde{D} \in \delta$. Let $\phi_3 = D\langle\gamma, expand(\delta, \phi_1', postgen(\ell(v)))\rangle$ and $\Delta_3 = \{(\ell(v), \phi_3)\}$. We can easily verify that both conditions of the lemma hold.

(b) $\tilde{D} \notin \delta$. Let $\phi_3 = \phi_1$ and $\Delta_3 = \Delta_1$. We simply don't make any change because changing v will not affect the result selected with δ.

Case VAR: The proof is similar to that for CON and is omitted here.

Case APP: We show the proof for the first condition about the relation among ϕ_1, ϕ_2, and ϕ_3. Since the proof about relations among Δ_1, Δ_2, and Δ_3 is almost the same as in proof for Lemma 19 and is rather simple, we omit it here. We have the following induction hypotheses.

$$\Gamma \vdash e_1 : \phi_{11} | \Delta_{11} \quad \Gamma \vdash e_2 : \phi_{12} | \Delta_{12} \quad \Gamma \vdash e_1\ e_2 : \phi_1 | \Delta_1$$
$$\Gamma \vdash e_1 : \phi_{21} | \Delta_{21} \quad \Gamma \vdash e_2 : \phi_{22} | \Delta_{22} \quad \Gamma \vdash e_1\ e_2 : \phi_2 | \Delta_2$$
$$\Gamma \vdash e_1 : \phi_{31} | \Delta_{31} \quad \Gamma \vdash e_2 : \phi_{32} | \Delta_{32} \quad \Gamma \vdash e_1\ e_2 : \phi_3 | \Delta_3$$
$$\lfloor \phi_{31} \rfloor_\delta \leq \lfloor \phi_{11} \rfloor_\delta \quad \lfloor \phi_{31} \rfloor_\delta \leq \lfloor \phi_{21} \rfloor_\delta \quad \lfloor \phi_{31} \rfloor_{\delta'} = \lfloor \phi_{11} \rfloor_{\delta'}$$
$$\lfloor \phi_{32} \rfloor_\delta \leq \lfloor \phi_{12} \rfloor_\delta \quad \lfloor \phi_{32} \rfloor_\delta \leq \lfloor \phi_{22} \rfloor_\delta \quad \lfloor \phi_{32} \rfloor_{\delta'} = \lfloor \phi_{12} \rfloor_{\delta'}$$

We need to show that

$$\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta \quad \lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta \quad \lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$$

In the following, we show $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_1 \rfloor_\delta$ with the assumptions that $\uparrow(\phi_{31}) = \phi_{31l} \to \phi_{31r}$ and $\uparrow(\phi_{21}) = \phi_{21l} \to \phi_{21r}$.

$$\begin{aligned}
\lfloor \phi_3 \rfloor_\delta &= \lfloor \phi_{31l} \bowtie \phi_{32} \lhd \phi_{31r} \rfloor_\delta \\
&= \lfloor \phi_{31l} \rfloor_\delta \bowtie \lfloor \phi_{32} \rfloor_\delta \lhd \lfloor \phi_{31r} \rfloor_\delta && \text{By Lemma 13} \\
&= \top \lhd \lfloor \phi_{31r} \rfloor_\delta && \text{By definition of } \textit{postgen} \\
&= \lfloor \phi_{31r} \rfloor_\delta \\
&\leq \lfloor \phi_{11r} \rfloor_\delta && \text{By induction hypothesis} \\
&= \lfloor \phi_1 \rfloor_\delta && \text{By a similar reasoning for } \phi_{11r}
\end{aligned}$$

We can prove $\lfloor \phi_3 \rfloor_\delta \leq \lfloor \phi_2 \rfloor_\delta$ and $\lfloor \phi_3 \rfloor_{\delta'} = \lfloor \phi_1 \rfloor_{\delta'}$ similarly.

The proof for other rules is similar to that for APP and is omitted here. □