Implementing a type debugger for Scala

Hubert Plociniczak Martin Odersky
École Polytechnique Fédérale de Lausanne
{first.last}@epfl.ch

Abstract

Statically-typed languages offer type systems that are less and less comprehensible for programmers as the language grows in complexity. In this paper, we present a type debugger, a tool that enables analysis of type-related problems as well as exploration of the typechecking process in general. We explain our findings on implementing a lightweight instrumentation mechanism for SCALA, as well as guide the reader through some typical debugging scenarios in which one can use our tool. The type debugger visualizes the internals of the typechecker which we believe increases its chances of being a successful educational tool, and which simplifies understanding of statically-typed languages in general.

Keywords type system visualization, debugger, compiler instrumentation

1. Introduction

Type systems offer a means to the programmers to abstract from concrete values and verify the high-level description of their program, with the aim of increasing their assurance that their code will do what it was designed to do. From the users' point of view, the typechecking process is a black box that they just have to trust. This state of matters is acceptable by the users as long as the decisions of the typechecker follow the intuition and knowledge of types of the programmer, and not reject programs that she considered type safe. When a program is rejected, the only feedback from the type system is given in the form of simple type error messages that do not reflect the complexity of the problem.

The dynamically-typed languages are better positioned than statically-typed ones to deal with this matter even though the error feedback is similarly uninformative as in the case of static typechecking. This is because they offer a large set of debugging tools ranging from primitive print statements via profilers to powerful debuggers. This way developers can analyze step-by-step the nature of any problem, understand the context in which their implementations took incorrect action and seek for potential solutions.

The approach towards improving error diagnosis had so far largely been focused on providing an automatic aid to the programmer [7, 9, 12]. This included heuristics for better error templates, suggestions for repairing mistakes by generating counter-examples or program slicing for better error isolation. However, as programming languages strive towards greater expressivity, making type systems more advanced, there is an increasing implicit discrepancy between the provided short error messages and the complex high-level concepts they refer to.

In order to eliminate that gap we present a type debugger, a tool used for the exploration of the typechecking process. When using our tool, each compiler decision is recorded, in order to be later presented in a form of an interactive proof, thus providing a better understanding of the context in which it was made. The current version of the tool does not attempt to provide immediate solutions to typing problems. Instead we promote the understanding of the type system in the incremental fashion typical for debuggers.

We based our implementation on SCALA that supports advanced type system features such as higher-kinded types [13], implicits [20], type members and path-dependent types [17] or type inference, which enable greater freedom during programming [18]. As a mature language, it exhibits the type system complexity issues shared with other popular languages like JAVA, C# or HASKELL.

In Section 2 we give a brief overview of SCALA, explain some aspects of its typechecking process and give an example of its visualization. Section 3 describes our mechanism for instrumenting the compiler, and techniques for analyzing and presenting the collected information in a simple user interface. Section 4 describes in detail three scenarios of dealing with non-trivial SCALA programming mistakes using the type debugger and evaluates the current implementation of our tool. In Section 5 we give an overview of related work and present our conclusions and future work in Section 6.

APPLC '12 June 14, Beijing.

2. Typechecking in SCALA: overview

Before we dive into the details of the SCALA compiler, we will give a very brief overview of the language. SCALA was born as a project that combines features of the object-oriented and functional programming world into one coherent language. SCALA shares some of its features with JAVA, like similar syntax and running on the JVM, and it provides interoperability with JAVA. It offers classes, objects (singleton pattern) and traits and shares common types with JAVA like `Boolean` or `Integer` but also provides higher-kinded types [13]. For a more comprehensive description we direct the reader to the literature [18, 19] but we will focus on two language features that will be often mentioned in the rest of our work - implicits and type inference.

Implicits is a type-driven mechanism for passing arguments, and it exists in two categories. The first category are implicit parameters that are just function or class parameters, whose arguments do not have to be applied explicitly by the programmer. Whenever such an argument is missing, the compiler searches for an implicit value in the scope that matches the type. The second category are implicit conversions (also known as views), in other words an implicit value of a function type. Whenever the term's type, say `A`, does not match the expected type, say `B`, it is still possible to succeed in typechecking. SCALA will search for an implicit function whose parameter type and the return type agrees with `A` and `B`, respectively. This allows for creating interesting language constructs [20].

SCALA's type inference mainly follows the design of GJ presented in [2, 14]. Its local type inference is a mixture of type propagation and solving local constraints, in contrast to the global constraints approach used in HINDLEY/MILNER type inference. The latter was hardly a feasible solution when dealing with subtyping constraints (size and principal types problems) and error localization issues [15, 16]. The end result of local type inference includes the reduction of the number of explicit type instantiations for polymorphic functions.

2.1 Compilation overview

The SCALA compiler runs multiple phases including a parser, name and type analysis, erasure, optimizations and bytecode generation. The parser provides initial abstract syntax trees (ASTs) that are successively transformed during each of the phases. As expected, the nodes of the SCALA trees mostly refer to their syntactic forms, such as values, methods or blocks of statements, and each of them has an associated offset position referring to its 'starting point' in the code.

For the purpose of type debugging we are only interested in the name and type analysis phases. The former associates symbols with ASTs, representing for example methods or classes, and enters each of them into enclosing scopes. The latter is responsible for type inference, implicits search and typechecking of the trees which results in assigning types to trees and symbols.

2.2 Typechecking = typing + adapting

SCALA's typechecking process for a given AST can be perceived as a two stage process. First, it attempts to assign a type to a currently considered tree depending on its kind and context. The expected type may have a considerable effect on inferring a more precise type but for brevity we omit the details. During the adaptation stage we adapt an already typed expression to the expected type. The latter involves, among other things verifying subtyping constraints, searching for an implicit conversion on failure, performing eta-expansions for partially applied functions or simply inferring any undetermined type parameters of the type. It is important to note that the typechecker will not always stop on the first failure. Instead, during both stages it can perform multiple fallbacks, for example by modifying expected type or performing implicit search in order to satisfy existing constraints.

Listing 1. Application of implicit conversion

```
class Foo(val x: Int)
def withoutImplicit {
  val a = new Foo(1)
  val b: Int = a  // Error: Found Foo, expected Int.
}
def withImplicit {
 implicit def fromFooToInt(f: Foo):Int =f.x
 val a = new Foo(1)
 val b: Int = a
}
```

To briefly show the motivation for a visualization of the typechecking process, we give an informal explanation of typechecking the fragments of Listing 1. There, `class Foo` takes a single parameter of type `Int`. The typechecker traverses the body of `withoutImplicit` and first typechecks `a` by inferring its type to be `Foo` and continues with the declaration of `b`. The latter will first typecheck an explicit type annotation `Int`, type the body of the assignment and fail to satisfy the subtyping constraint `Foo <: Int` enforced by the context at the adaptation stage. On the other hand, typechecking `withImplicit` succeeds thanks to the valid implicit in scope i.e., `fromFooToInt`. The compiler will type the application involving the view, namely `fromFooToInt(a)` and will succeed during the adaption as `Int <: Int`.

Although some of the above information can be inferred directly from the language specification, it is not very approachable for users. Consider a snapshot of SCALA's type debugger in Figure 1 representing the compiler's attempt at verifying `withImplicit`. It consists of the typechecking visualization in a form of a proof tree on the left and the source code on the right. Each node is a goal that the typechecker needs to satisfy, which it in turn divides into multiple subgoals that can be revealed interactively. Goals store internally more descriptive information available to the user

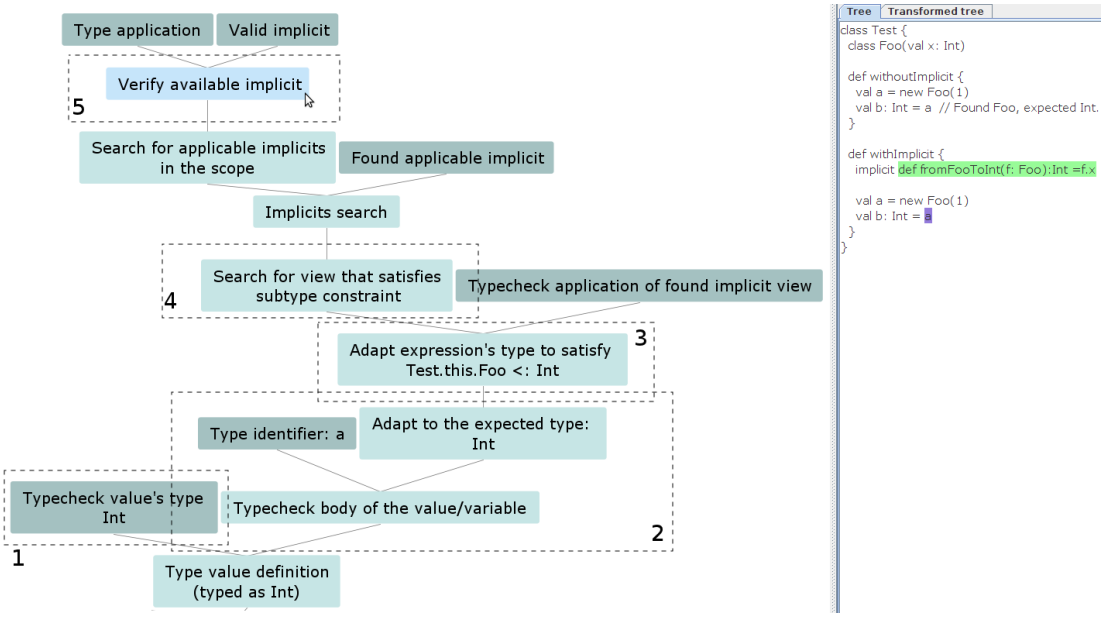

Figure 1. Typechecking definition of value `b` in method `withImplicit`

through UI. Having a closer look at how the proof is resolved for the value `b` in `withImplicit`, one can find direct correspondence to our informal explanation before: the checking of the explicit type annotation of the value (1), the typechecking of the body of the value (2), the failed subtyping constraint (3), the search for implicits (4) and the verification of `fromFooToInt` as a potential solution (5). Dashed boxes on the figure are used only for the presentation purposes of this paper, they do not appear in our tool.

3. Implementation

This section details vital changes to the compiler that allowed us to implement the type debugger efficiently. Initial attempts at storing all the typing information within the ASTs proved to be infeasible due to the large amount of redundant data and the memory burden. Instead, we developed a lightweight event system that allows us to instrument typechecker, hook up directly into the running instance of the compiler, processes the data and feed it into an UI.

3.1 Lightweight event mechanism

The `event` mechanism can be considered as an internal compiler plugin mechanism that allows the language architects to take a snapshot of pre-determined data at a specific execution point in the compiler. In fact its initial aim was to enable convenient extraction of information based on hard-coded filters i.e., it was useful for finding bugs in its original version but not for understanding the bigger picture. The system comes with a simple DSL used to emit instances of `Event` classes.

Listing 2 represents a typical situation where a programmer wants to be informed about a tree being typed. The method `typed` can be considered as the main entry point to the typechecking process, whereas the `EV` object represents a reference to our `event` plugin mechanism. The `<<` member of `EV` takes an argument of type `Event` (here used in infix notation) and pushes it further to any registered listeners. Class `TypedNode` stores references to the current typing context, the initial and the typed tree. The instance of `TypedNode` is a `case class` that allows for convenient filtering and further extraction of data.

Listing 2. Issuing `events` within the compiler

```
def typed(tree: Tree, expected: Type): Tree = {
  ... // perform typechecking
  EV << TypedNode(context, tree, typedTree)
  typedTree
}

case class TypedNode(ctx: Context, tree0: Tree, tree: Tree)
 extends TreeEvent { ... }
```

3.2 Instrumenting the compiler

Compiler instrumentation, necessary to visualize the problem presented in the overview, can require relatively tedious work as we have to handle at least all the functions that typecheck different kinds of trees in order to present the general picture of the process.

The DSL also has `<<<` and `>>>` operators to deal with explicit *opening* and *closing* blocks of subgoals, respectively. In a sense, those operations emit the `events` as before except that they carry an additional goal-subgoal relationship information. Our first text-based prototype used that information to indent the events properly. In the end it allowed us to simplify the front-end that was processing the instrumented data so that it does not have to reverse engineer the whole logic behind the compiler. The additional advantage of generating such information on a compiler level is

that any future UI that can hook up into the compiler can immediately generate a graphical representation of the data.

The `event` mechanism allows us to emit classes representing the state of the execution of the compiler, but they typically do not carry exact context information. Consider function `def run(v: String): String` when used in an application `run("foo"): Int`. As we have learned already, the typechecker will attempt to type the application `run("foo")` as well as `toInt(run("foo"))` (in the adaptation stage, assuming an existence of a matching implicit `toInt`). From the typecheckers point of view, both involve value application and internally are represented through similarly shaped ASTs. Instead of providing even more instrumentation, which is a possible solution, we decided to add an additional implicit parameter to the main typechecker functions in order to carry high-level context information. A modified signature of the method `typed` from Listing 2 would now become:

```
def typed(tree: Tree, exp: Type) (implicit val expl: CInfo) : Tree
```

and can be used to carry more information while typing the members of the `Definition` AST (like its type parameters, parameters and body):

```
def typedDef(ddef: DefTree, expected: Type): Tree = {
  val tps = ddef.tparams.map(tp => typed(tp, ?) (DefTParam) )
  val ps = ddef.params.map(p => typed(p, ?) (DefParam) )
  val body = typed(ddef.body, ddef.pt) (DefBody)  ... }
```

3.3 Errors as exceptions

Type errors do not always get reported to the user, backtracking being one of the reasons for that. In such a situation instead of being reported, the typechecker throws an exception internally. The compiler catches the error and runs a different typechecking scenario. Throwing exceptions had an unfortunate consequence of ruining our logical blocks of instrumented data. A temporary solution was to wrap any of the block operations in a standard `try/catch` or analyze the stacktrace, ensure that blocks are properly closed and propagate the exception. Neither of them was elegant nor efficient not to mention the fact that exceptions started to influence the control flow of the compiler which by principle is a bad design. Instead we decided to store errors within the typing context and make explicit decisions within the compiler depending on the state of the context.

3.4 Visualization

Our initial basic Swing interface lacked *interactivity* and we decided on using a successful work on PREFUSE [6], a toolkit for creating rich interactive visualizations.

Apart from the obvious display of the nodes corresponding to the instrumented data, the type debugger UI includes the following features:

- Convenient mouse and/or keyboard navigation.
- Shortcuts that allow for automatic localization of the typechecking proof responsible for assigning a given type to the value.
- The initial typechecking tree expands only the type error and the closest goals leading to it since exploring the full tree is often infeasible due to a large number of nodes. Whenever more errors are reported, we expand the least spanning tree that covers them in order to speed up debugging (errors have a tendency to be correlated).
- Interacting with any nodes highlight their corresponding source code. This is possible due to the trees having range positions that record the beginning and the end point in the code. Due to the format of this paper we can only show one highlighting per figure.
- We can display full ASTs corresponding to the nodes, which is useful for more advanced SCALA programmers.
- The tool offers an advanced mode which visualizes among other things detailed information about implicits applicability, calculating least upper or greatest lower bound among types, as well as detailed subtyping checking and synthetic trees.
- Since node names hardly convey enough information for users not familiar with the typechecker, each of them has an associated tooltip description that shows detailed representation of types, symbols or ASTs or aims of the typechecking goals.

The type debugger can also work with error-free code. In that case users start their exploration on the top package level and can freely direct the expansion of the goals.

4. Evaluation

The next three examples present typical kinds of problems SCALA programmers have to deal with. In order to demonstrate and evaluate the type debugger's usefulness we will go through each of them in a form of a brief tutorial.

4.1 Inferring `Nothing`

SCALA's type inference allows for writing clear and compact code. But omitting type annotations sometimes leads to surprising types that get inferred. Consider the code presented in Listing 3.

Listing 3. `Nothing` inference

```
class A {
  def foo[T](a: Int)(b: T): T = b

  def bar {
    val par = foo(10) _
    par(2)  // found    : Int(2)
  }         // required: Nothing
}
```

Here, `foo` defines a method having two parameters (think currying functions), `a` and `b`, and a type parameter `T`. `bar`

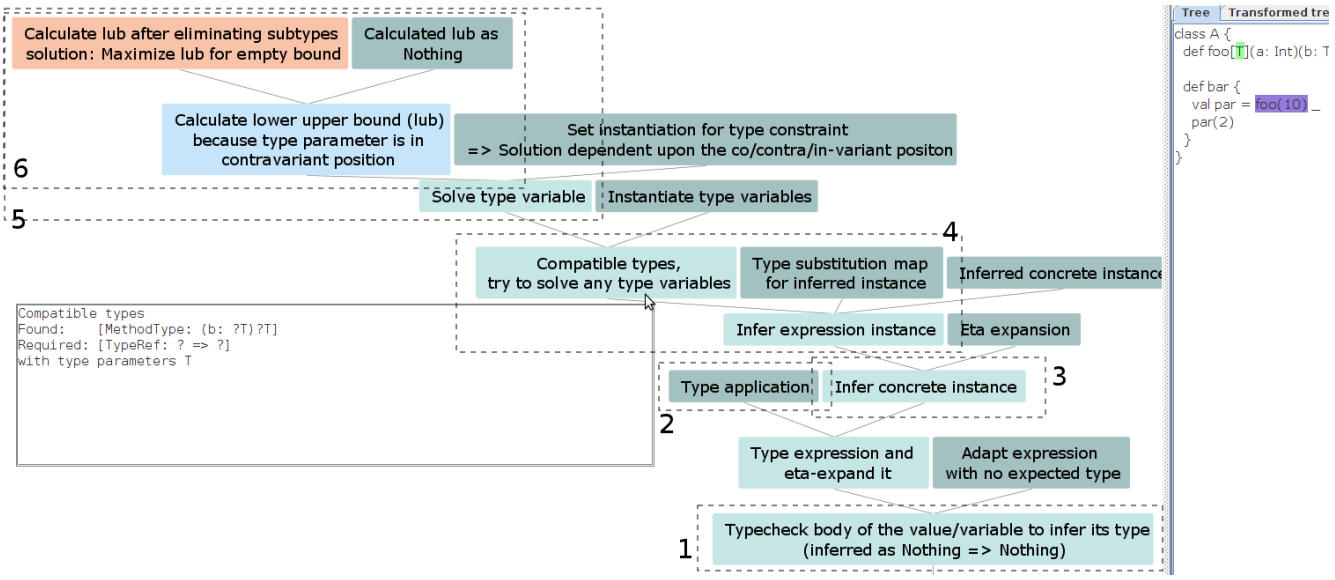

Figure 2. `Nothing` inference

method partially applies `foo` using the underscore `_` keyword. Later we want to finish the application to `par` using `2` as an argument. A typical SCALA user would assume that a method `bar` should compile without any problems. Instead she will get a mysterious type error saying *Found Int, Expected Nothing* when applying `2` to `par` (`Any` and `Nothing` are at the top and bottom of the SCALA's type hierarchy, respectively). This can be frustrating since `Nothing` is hardly ever the type expected by the users.

Consider now a fragment of a type debugger's session in Figure 2 which focuses on the process of typechecking the `par` definition (box 1). After typechecking the partial application of `10` to `foo` (box 2), the compiler still needs to instantiate the type parameter `T` (box 3) because SCALA does not support higher-ranked types. Since neither the function parameter `b` nor the return type is known, it infers the expected type to be `? => ?`, where `?` refers to a wildcard. This is compared with the current method type `(b: ?T)?T` where `?T` underlines the usage of a fresh type variable instead of just `T` (box 4). Since that is valid, one can proceed with solving any remaining type variables i.e., `?T` (box 5). Later we are calculating the least upper bound of all the constraints collected while solving the type variable due to the contravariant position of the type parameter `T` (box 6). Since the list of constraints is empty, the inferencer will maximize the solution and pick the most suitable type, namely `Nothing`. The problem can be fixed by explicitly providing the `par`'s type. A possibly more elegant and valid solution would be to reuse our findings from the type debugger and change the signature of `foo` to `foo[T >: Int](a: Int)(b: T): T` to guide the inference of a least upper bound solution for the type parameter `T`. Although the current implementation does not support such suggestions, we believe it will be feasible to implement a mechanism where user can interactively request inference of some more precise type, and the type debugger will subsequently suggest code location(s) and/or manipulation(s) that could possibly satisfy it (based on the location of the original type constraint).

4.2 Implicits and ambiguity

Function overloading is a popular paradigm available in many programming languages. Yet, when excessively used by programmers, it can lead to code that is hard to understand and maintain. Such situations may also come up when dealing with certain SCALA features like implicit resolution or type inference.

Listing 4. Ambiguity for overloaded methods

```
class Base[T]
class NatA[T] extends Base[T]
class NatB[T] extends Base[T]
class AA[A]

object Ambiguity {
 implicit def conv1(i: Int) = new NatA[Int]
 implicit def conv2(i: Int) = new NatB[Int]
 implicit def conv3(op: AA[String]) = new Base[String]
 implicit def conv4(op: AA[Int])    = new Base[Int]

 def aFunc[A](a: NatA[A])  = new AA[String] // (1)
 def aFunc[A](a: NatB[A])  = new AA[Int]    // (2)

 def bFunc[T](e1: Base[T]): Base[T] = e1

 def convertToBase1(p: Int): Base[Int] = {
   val x = aFunc(p)  // Fails
   bFunc(x)
 }
 def convertToBase2(p: Int): Base[Int] =
   bFunc(aFunc(p))   // OK
}
```

Consider the program presented in Listing 4. The code itself is a simplified version of a real DSL library and a bug report within SCALA itself. We define a simple class hierarchy involving `Base[T]` as a supertype for `NatA[T]` and `NatB[T]`. Class `AA` is not a subtype of `Base` but there is an implicit conversion from `AA` to `Base` (see `conv3` and `conv4`). The `Ambiguity` test is self-contained and we encourage the reader to try to perform manually an ambiguity resolution and explain the type error for `convertToBase1`:

```
both method aFunc of type [A](a: NatB[A])AA[Int]
and  method aFunc of type [A](a: NatA[A])AA[String]
match argument types (Int)
   val x = aFunc(p)  // Fails
           ^
```

In fact a difference between the two methods can be a result of a simple refactoring gone wrong.

Figure 3 represents a snapshot of the type debugger presenting a path to the error (box 8) in more detail. The ambiguity occurs while we are trying to typecheck the right-hand side of the value assignment in `convertToBase1`, which in turn is necessary for inferring the type of `x` (box 1). Typechecking `aFunc` in the application returns an overloaded type involving two alternatives (box 2). Since argument `p` is of type `Int` (box 3) and there is a mismatch between the type of the argument and the parameter, inference of a correct alternative fails without implicits (box 4). With implicits turned on (box 5), the alternative (1) will succeed because of a successful implicit search involving the expected type `Int => NatA[Nothing]`, namely `conv1` (box 6). Similarly the alternative (2) of type `[A](a: NatB[A]): AA[Int]` will succeed on the implicit search with expected type `Int => NatB[?]` due to `conv2` (box 7). In this particular case the comparison of the two alternatives does not lead to any clear winner and we are forced to report an ambiguity type error for the overloaded method `aFunc` (box 8).

The typechecking of method `convertToBase2` leads to a type debugger visualization presented in Figure 4. One can immediately see that the argument of `bFunc` is now typed with an expected type of `Base[Int]` (box 2) due to the constraints coming from `bFunc`'s parameter type and `convertToBase2`'s return type (box 1). Due to the expected type imposed on the overloaded type involving the two alternatives (box 3), the typechecker is able to successfully filter out the alternative (1) (boxes 4 and 5). After dealing with the invalid alternative, typechecking resolves to a normal typechecking of the application of argument `p` to `aFunc` (box 6) where adaptation needs to satisfy the `Int <: NatB[?]` subtyping constraint (not shown on the figure).

We believe that exposing the internal workings of the compiler is essential, especially to library architects. Mixing advanced features leads usually to non-trivial problems but programmers should not be discouraged by that since it allows for a creation of interesting designs like human-readable DSL libraries[1].

4.3 Invariance and refined type

TYPE CLASSES [24] is a widely accepted type system construct to support ad-hoc polymorphism. It was first introduced by HASKELL but SCALA has been shown [20] to offer similar capabilities through implicit parameters. The example from Listing 5 discusses usage of a generic ordering function inspired by the TYPE CLASSES pattern in SCALA. Consider function `universalComp` that does generic comparison between the two values of the same type. Since there is no way to compare generic types without any additional information or constraint, we require an implicit value of type `Ordering[T]` that bounds the type of `T` (more commonly known as a context bound in SCALA). `Ordering` is a trait in the standard SCALA library that defines abstract member `compare` which communicates how the two values of type `T` should compare. We also provide an implicit value `AOrdering` in the scope that gives an implementation of `Ordering` for values of type `A`.

Listing 5. Implicit values and variance

```
class A { def f: Any }
class B extends A { def f: Int = 5 }
class C extends A { def f: Long = 5L }

def universalComp[T](t1: T, t2: T)
  (implicit evidence: Ordering[T]) = 1
implicit val AOrdering: Ordering[A] = ...

universalComp(new B, new C)
// error: No implicit Ordering defined for A{def f: AnyVal}.
//    universalComp(new B, new C)
//                  ^
universalComp[A](new B, new C) // works
```

Therefore it may seem surprising that an application involving two values of type `B` and `C`, where both are subtypes of `A`, fails when trusting the type inference in inferring an implicit value (box 1 in Figure 5) but typechecks when we explicitly apply the type. Figure 5 shows that the typechecker verifies default implicit values defined in the standard SCALA library (box 3) that could potentially satisfy the implicit evidence. None of it provide an ordering or comparator that matches our newly defined `class A` due to the expected type (box 4). A more surprising fact is that it did not even try to typecheck our `AOrdering` implicit value as a potential argument for `universalComp` (box 2). A further exploration of the type debugger shown in Figure 6 reveals that the (correctly) inferred refined type `A{def f: AnyVal}` (least upper bound of `B` and `C`) is simply too precise. This is because applying value `AOrdering` as an argument for parameter `evidence` (box 1) requires constraint `Ordering[A] <: Ordering[A{def f: AnyVal}]` (box 2). Since the documentation for `Ordering` clearly states

[1] SCALA's testing frameworks, like Specs, ScalaCheck or ScalaTest, are a prime example of intuitive libraries

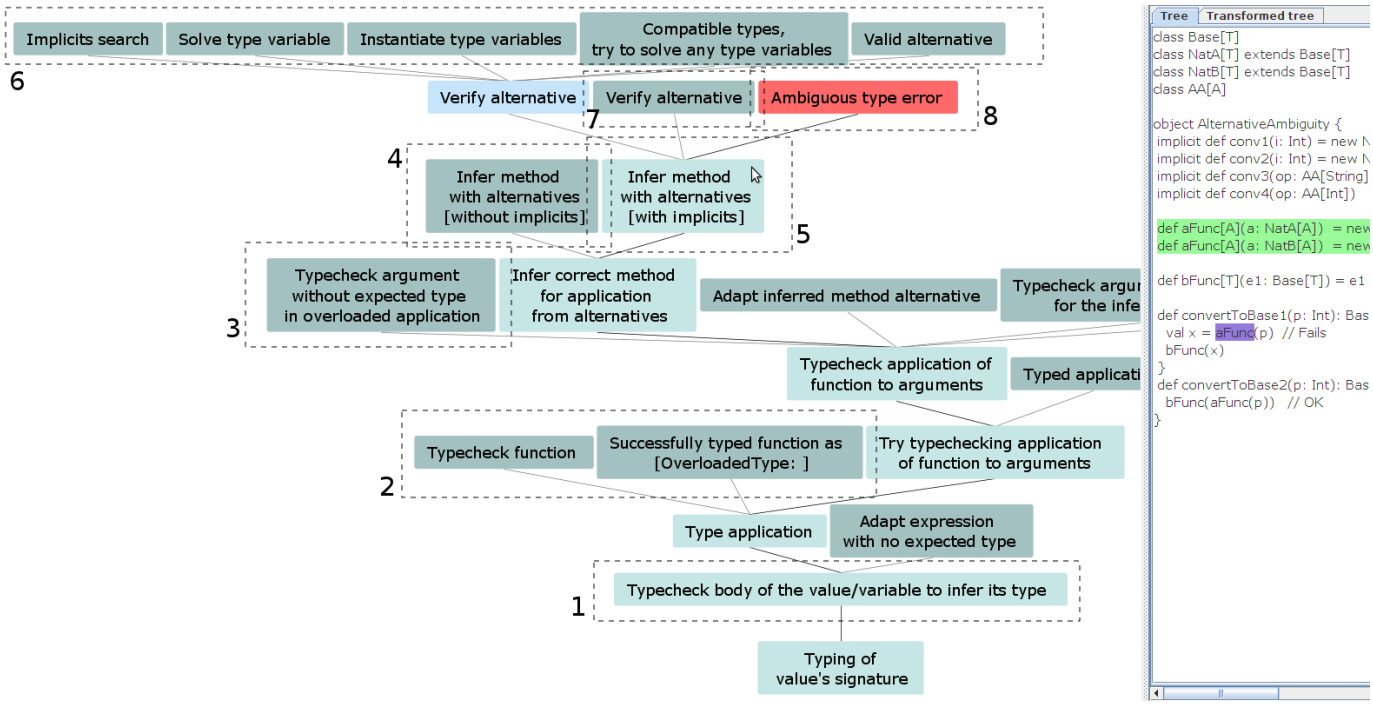

Figure 3. Typechecking `convertToBase1`

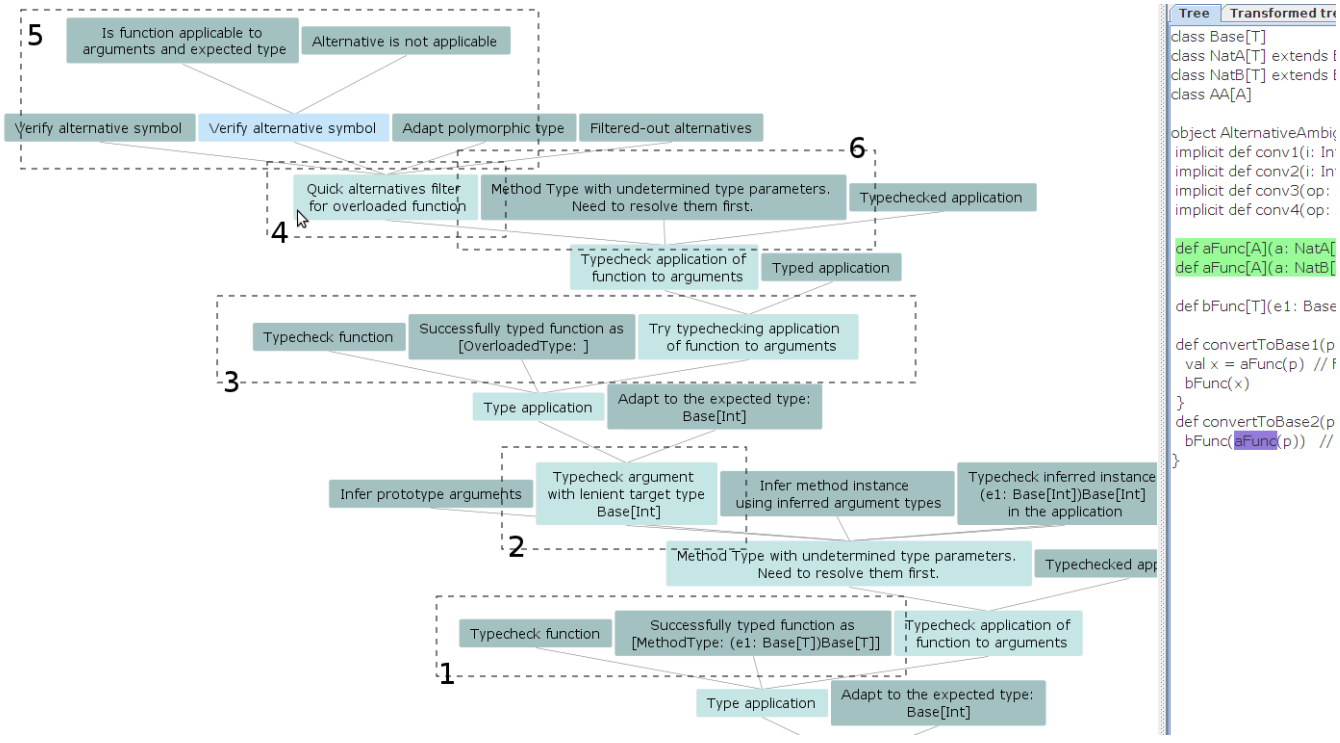

Figure 4. Typechecking `convertToBase2`

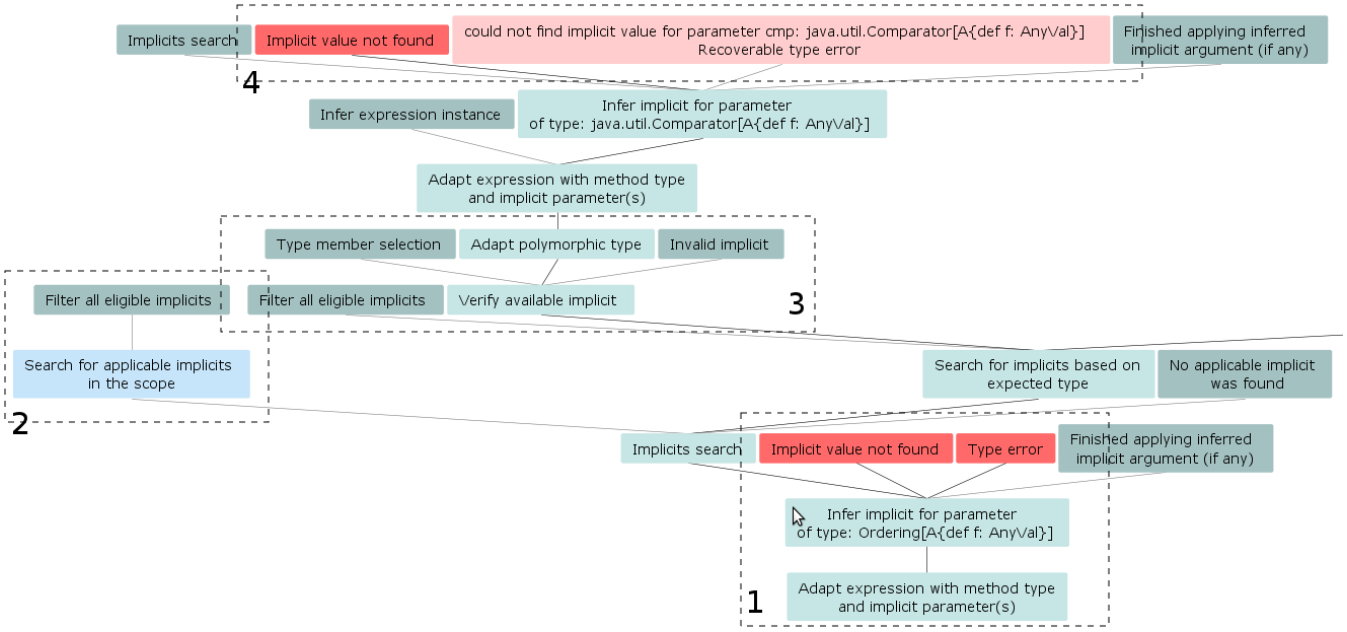

Figure 5. Failed implicit search for `universalComp` application (source code panel omitted)

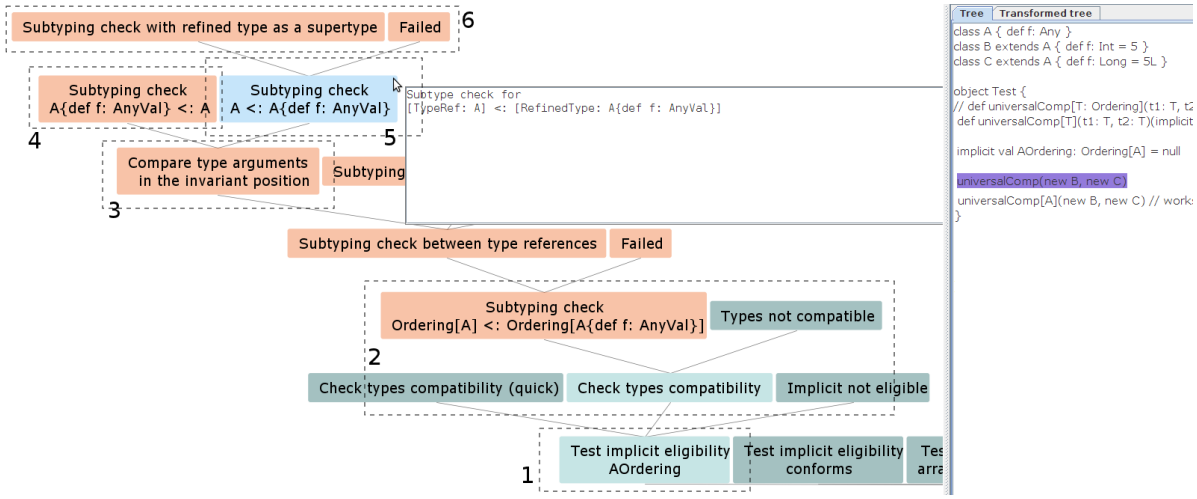

Figure 6. Testing eligibility of an implicit value for `Ordering` type parameter at an invariant position

that its only type parameter is at an invariant position (box 3), it implies that `A{def f: AnyVal} <: A` (box 4) and `A <: A{def f: AnyVal}` (box 5) has to hold. It is clear that the latter condition will always be false (box 6). In the future when the user will want to understand the applicability of a specific value as an argument to the implicit parameter, the tool will offer quick navigation to a corresponding proof tree in the visualization, rather than requiring a manual search from the user.

Bugs involving variance in type parameters belong to some of the most common problems encountered by SCALA programmers and which lead to long hours of debugging on examples similar to the one just presented.

4.4 Performance and usability

The type debugger was designed for a mature language, so it was obvious from the beginning that the tool will have to be tightly integrated with its compiler implementation. At the same time it should only have a minimal effect during normal runs as we do not intend to maintain two separate typechecker implementations. Instrumenting involves simple method calls that are ignored during non-debugging runs.

Although a large part of SCALA's typechecker is written with immutability in mind in a functional way, some parts critical to performance are not. Since instructions store references to trees, types or symbols we cannot rely on the fact that their state taken during the instantiation of the `Event`

is the same as during the visualization of the node in type debugger. To avoid such a discrepancy we clone trees, types and symbols when necessary which is obviously expensive.

Listing 6. Optimizing `event` emission

```
@inline final def <<(ev: => Event): EventResponse = {
  if (eventsOn) eventHooks foreach (_ applyIfDefined ev)
  NoResponse
}
```

In order to still keep our lightweight instrumenting, all the methods involved in the emission of `events` (Listing 6 presents one example from our DSL) needed to be inlined as well as hidden behind the flag that enables our debugging mode. Finally, in order for the optimizer to do its job, all the `events`' DSL methods needed to have by-name parameters (by-name arguments are internally transformed as expensive closures). Otherwise they would deteriorate the performance by being instantiated outside of the `if (eventsOn)` condition.

Compiling the SCALA compiler itself (133k LOC) gave us 41% and 37% slowdown for the non-optimized and optimized instrumented compiler, respectively [2]. The results are not satisfying mostly due to the failure of the inliner to optimize some of the most critical parts, caused by an internal compiler bug, and as a result some of the costs mentioned above could not be mitigated. However for small to medium sized applications the performance degradation was not noticeable.

The current implementation has already proved to be helpful in an investigation of real bugs within the compiler itself. We realized that problem location is much quicker (given the proper instrumentation already in place) than debugging using existing manual techniques. That is a significant progress given that typechecker bugs are usually categorized as hard to fix. Admittedly, the type debugger is not yet feasible to be used on large applications due to the size of the typechecking proof graph generated during a single run (when shown in full). It is also not yet designed for beginner programmers since it requires some prior knowledge about type systems and programming languages concepts and lack of more fine grained mechanism for filtering and managing the typechecking proof.

The process of instrumenting the compiler is rather mechanical thanks to the lightweight `event system` and rewritten error reporting mechanism. Nevertheless identifying important spots and providing meaningful descriptions does require a good knowledge of SCALA's typechecker. During the course of developing the tool we instrumented the most critical parts of the typechecker to get a general overview of the process, as well as focused on parts just presented. This process is by no means exhausted and will be continued as the tool gets used for solving different problems.

[2] tests performed on a load-free Intel Core 2 Duo machine with 3.2 GHz processor and 4GB of RAM

5. Related work

CHAMELEON [21, 23], a HASKELL-like language, was one of the few promising attempts aimed at solving the debugging problem. Instead of having plain type error messages it was offering means to explore them in detail. Unlike our attempt it relied on a system based on solving constraints using Constrained Handling Rules [22](CHR). Therefore rather than dealing with an existing language they focused on explaining an admittedly large subset of HASKELL. Whenever a type error was encountered the tool enabled the user to see what constraints were in conflict or could not be satisfied. Users could also explore the process of building the constraints using CHR from a source code in a real-time manner for both type correct and erroneous programs. In comparison to our work CHAMELEON was a text-based debugger and offered help for possible error localization. Also at the moment we only describe the reasons for the type constraints existence at the point when they are generated but we will work towards better navigation between them. Apart from the above HASKELL has seen several attempts [3, 7] at improving lack of debugging tools however most of them resolved around improving type error messages by giving only slightly better context or using heuristics that were supposed to better locate the real source of the problem.

Work by McAdam [11] describes techniques for presenting type information in the form of a graph. The author's research is based on a variation of a simply-typed λ-calculus. One can assume that in a sense our tool exercises the author's initial idea when dealing with a mature programming language. Unlike our visualization, the graphs do not offer any incremental exploration, making it impractical for even medium sized terms. Similar scalability issues have been experienced when recording directly the decisions made during the type inference process [1, 5].

PLT REDEX [10] is an embedded domain-specific language developed for designing term-rewriting systems. What clearly distinguishes it from the similar solutions in that domain is the existence of the visualizing environment where users can create and debug their models in an interactive manner. What is more important is that the tool has proven useful in preparing its host language, RACKET, for a wider adoption. Further work on REDEX [8] presents a possibility for defining and debugging formalizations of proofs. One can immediately see that that there is a close similarity between visualizing operational semantics and typechecking. Although formalization of SCALA's type system could possibly be done using REDEX, visualization of the actual implementation has to deal with a different set of problems.

Recent work on the JAVA compiler [4] has revealed plans on including support for debugging overload resolution. Similarly as with SCALA's type debugger, the JAVA compiler is being instrumented and each step of the decision process can be shown to the user. In contrast to our work `javac` will hardly put a structure to the instrumented data.

6. Conclusions and future work

The initial implementation of the type debugger has met our requirements of being able to assist the programmers in understanding the typecheckers logic and finding non-trivial bugs. Although not yet feature complete, the interactive aspect of the visualization is an appealing way of exploring the internals of the compiler. We presented three scenarios where the type debugger offers clear advantage for finding sources of type errors. We are confident that the reader will experience positively the more interactive side of the evaluation of the tool by simply running it on their own.

We plan to equip the type debugger with more background information related to the programming languages concepts, like references to the language specification or relevant documentation. The type debugger is already appreciated by compiler hackers in the localization of bugs (three fixed already, more identified) but we plan to implement interactive filters for presenting instrumented information that would make it more customizable and as a result useful with larger programs that generate even more data. Apart from planning to integrate the type debugger with the existing SCALA plugin for Eclipse[3] we want to allow for selective instrumenting of trees and as a result targeted debugging. Large typechecking proofs create a lot of noise but they also provide a good basis for further specialization of the views. Therefore we plan to explore the idea of building a set of templates associated with type error messages and their contexts in order to guide the user in the debugging exploration. For error-free scenarios we want to generalize visualizations that filter out detailed internals of the compiler or synthetic constructs, possibly creating another layer of building blocks to reduce the number of typechecking goals that users have to comprehend at a specific point. With this infrastructure in place we will be able to perform extensive user studies.

The source code of the type debugger, along with presented examples, is available online:

https://github.com/hubertp/prefuse-type-debugger

Acknowledgments

We would like to thank Paul Phillips for his initial work on the `event` system as well as anonymous reviewers for their detailed and helpful comments.

References

[1] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.*, 2:17–30, March 1993.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA*, pages 183–200, 1998.

[3] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP*, pages 193–204, 2001.

[4] M. Cimadamore. Testing overload resolution, 2011. URL `https://blogs.oracle.com/mcimadamore/entry/testing_overload_resolution`.

[5] D. Duggan and F. Bent. Explaining type inference. *Sci. Comput. Program.*, 27:37–83, July 1996.

[6] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI*, pages 421–430, 2005.

[7] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning haskell. In *Haskell*, pages 62–71, 2003.

[8] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *POPL*, pages 285–296, 2012.

[9] B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *ML*, pages 63–73, 2006.

[10] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *RTA*, pages 301–311, 2004.

[11] B. McAdam. Generalising techniques for type debugging. In *TFP*, pages 49–57, 2000.

[12] B. McAdam. How to repair type errors automatically. In *TFP*, pages 87–98, 2002.

[13] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *OOPSLA*, pages 423–438, 2008.

[14] M. Odersky. Inferred type instantiation for gj, 2002. URL `http://lampwww.epfl.ch/~odersky/papers/localti02.html`.

[15] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5 (1):35–55, 1999.

[16] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

[17] M. Odersky, V. Cremet, C. Rckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*, pages 201–224, 2003.

[18] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, EPFL, 2006.

[19] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2nd edition, 2011.

[20] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360, 2010.

[21] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27:1216–1269, November 2005.

[22] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type processing by constraint reasoning. In *APLAS*, pages 1–25, 2006.

[23] M. Sulzmann. An overview of the chameleon system. In *APLAS*, pages 16–30, 2002.

[24] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, pages 60–76, 1989.

[3] http://scala-ide.org/