Scalad: An Interactive Type-level Debugger

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

ABSTRACT

Dealing with statically-typed languages and their complex type systems is problematic for programmers of all levels. Resolving confusing type errors is a time consuming and not always successful process.

In this tool demonstration we give an overview of Scalad, an interactive tool that can explain decisions made by the existing typechecker of a multi-paradigm programming language by visualizing the whole process in the form of a proof tree. The tool works for both type correct and incorrect programs, making it suitable for educational purposes as well as debugging. We provide examples on how the tool can be used to understand typing puzzles. The debugger comes with an expandable search mechanism that can precisely guide users in finding answers to the typechecking problems and improve exploration time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, error handling and recovery*

Keywords

type system, visualization, debugging, errors

1. INTRODUCTION

Statically-typed languages offer programmers an opportunity to verify their code before the execution. That reduces the number of things that *can go wrong* during runtime but at the same time limits their freedom in terms of writing code that corresponds to their desired logic. Often it also leads to unavoidable type complexity and non-trivial type error messages produced by the typechecking black box.

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

Listing 1: Incrementing a list using foldRight

Consider a trivial program presented in Listing 1 written in Scala. Using the higher-order method `foldRight` of `List` we attempt to increment all its values. The method call successively appends incremented elements, going right to left and starting with an empty list accumulator (`Nil`). Yet Scala reports a type error that hardly hints at the source of the irregularity:

```
found   : List[Int]
required: scala.collection.immutable.Nil.type
    (x + 1) :: ys)
            ^
```

Such typing problems lead to disappointment in Scala's local type inference and an excessive usage of type ascription in further declarations, even if not entirely practical. We analyze the problem in detail with the help of our tool in the next section.

Scalad is a type-level debugger built to help users understand non-trivial type system issues. The tool provides a visualization of the typechecking in the form of an interactive proof tree. We refer the reader to an earlier work [Plociniczak and Odersky(2012)] which provides the details of the design for efficient extraction of the compiler information as well as the initial presentation attempts. In this tool demonstration we focus on the usability side of the debugger through a brief tutorial.

2. TYPE DEBUGGER

The type debugger is implemented on top of a recent trunk version of the Scala compiler. The universality of the tool stems from the fact that it can handle various type system tasks, ranging from subtyping or overloading resolution to higher-kinded types and implicits.

Scalad comes with a simple user interface consisting of an initially blank page on the left (later filled with the complete proof tree) and user code on the right side. At this point the tool is no different from the front-end of the standard compiler. Investigation of any type-related problem starts by selecting a fragment of code to debug. This triggers a compilation run that gathers a minimal amount of information concerning only the selected region. The result of such an initial run on the erroneous location from Listing 1 can be seen in Figure 1(*selection 1*).

The typechecking proof is presented in the form of a derivation tree. Each node represents a single step that the compiler tries to satisfy. Initially we only expand the minimal subtree covering goals that refer to main errors and the requested code region. The non-exhaustive set of features that help with the proof navigation includes:

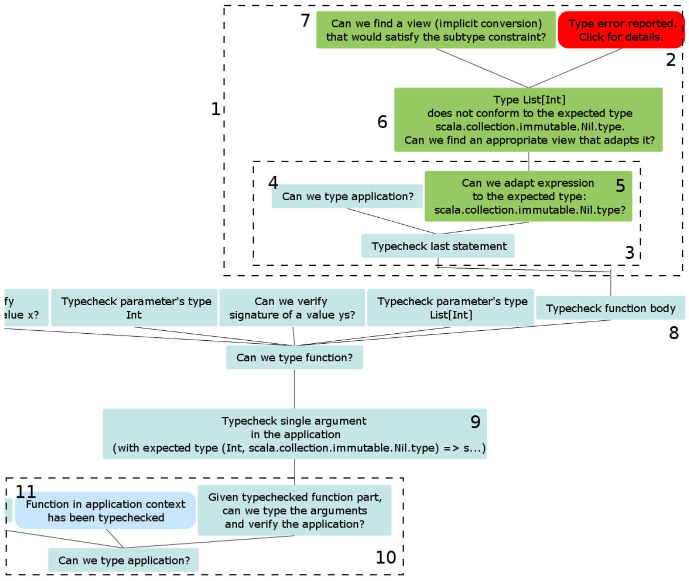

7
Can we find a view (implicit conversion) that would satisfy the subtype constraint?
Type error reported. Click for details.
2
1
6
Type List[Int] does not conform to the expected type scala.collection.immutable.Nil.type. Can we find an appropriate view that adapts it?
4
Can we type application?
Can we adapt expression to the expected type: scala.collection.immutable.Nil.type?
5
Typecheck last statement
3
Typecheck parameter's type Int
Can we verify signature of a value ys?
Typecheck parameter's type List[Int]
Typecheck function body
8
Can we type function?
Typecheck single argument in the application (with expected type (Int, scala.collection.immutable.Nil.type) => s...)
9
11
Function in application context has been typechecked
Given typechecked function part, can we type the arguments and verify the application?
Can we type application?
10

Figure 1: Finding location of expected type (part 1)

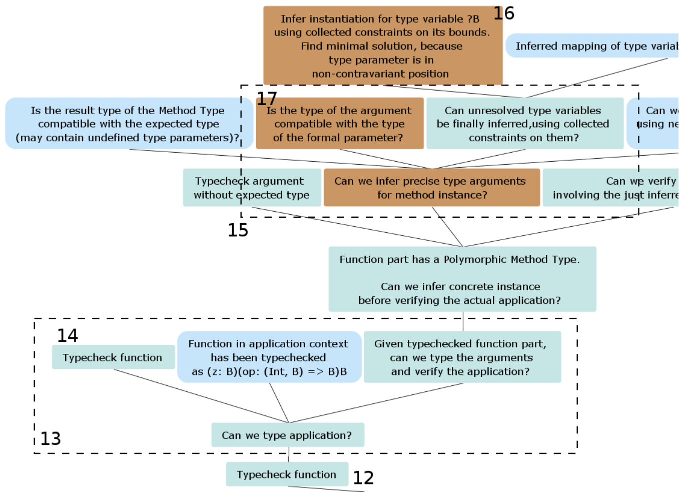

Infer instantiation for type variable ?B using collected constraints on its bounds. Find minimal solution, because type parameter is in non-contravariant position
16
Inferred mapping of type varial
17
Is the result type of the Method Type compatible with the expected type (may contain undefined type parameters)?
Is the type of the argument compatible with the type of the formal parameter?
Can unresolved type variables be finally inferred,using collected constraints on them?
Typecheck argument without expected type
Can we infer precise type arguments for method instance?
Can we verify involving the just inferre
15
Function part has a Polymorphic Method Type.
Can we infer concrete instance before verifying the actual application?
14
Typecheck function
Function in application context has been typechecked as (z: B)(op: (Int, B) => B)B
Given typechecked function part, can we type the arguments and verify the application?
13
Can we type application?
Typecheck function
12

Figure 2: Finding location of expected type (part 2)

- Selective debugging, primarily used for triggering targeted debugging, can also be used for coarse-grained navigation.
- Goals can be freely expanded and collapsed.
- Navigating through goals highlights corresponding code fragments.
- Goals are presented in the form of questions that will be solved. Details, like full type signatures, specification references and explanations, are available as tooltips.
- Goals can provide short answers to the questions to inform about the state of a particular branch.

The reported type error (*goal 2*) occurred while typechecking the last statement of an anonymous function (*selection 3*). Its immediate subgoals consist of typing and adaptation. Typing involves typechecking all the components of the abstract syntax tree (AST), and then assigning the type based on its kind and context. The typing goal (*4*) verifies the application `(x + 1)::ys`. The adaptation stage makes sure that the inferred type conforms to the expected one. This may involve inferring still undetermined type parameters, applying implicit arguments or performing necessary conversions. Therefore (*goal 5*) involves adapting `List[Int]` to `Nil`. Typechecking may also require an additional step only performed on the first definition usage attempt - verification of the type signature (if provided explicitly by the user) or its inference from the body of the definition (if type ascription was missing). The three stages are the only background information about the compiler initially required from the user.

The error occurred as `List[Int]` does not conform to `Nil` (*goal 6*) and the implicit conversion fallback was unsuccessful (*goal 7*). One understands the context of the error by tracking back the typechecking process, starting with *Typechecking the last statement*, *Typechecking function body* (*goal 8*) and arriving at *Typechecking argument with expected type (Int, Nil) => Nil* (*goal 9*). At that point, we reached typechecking the full application `x.foldRight(Nil)(...)` (*selection 10*). Its function part has already been typechecked and has a concrete instance of a method type `(op: (Int, Nil) => Nil)Nil` (*goal 11*). Since the searched type `Nil` is clearly related to the type of the function part, the user would expand the goal responsible for typechecking `x.foldRight(Nil)` (*goal 12*, Figure 2) for which the compiler poses the question *Can we type application?*. Since we are again in the application context (*selection 13*), the derivation tree involves verifying the function part `x.foldRight` which ends up with a generic method type `(z: B)(op: (Int, B) => B)B` (*goal 14*). Hence the user can reasonably expect that the type inference will take place in the next subgoal *[...]can we type arguments and verify the application?*. Further expansion proves the existence of type inference: *Can we infer precise type argument for method instance?* (*selection 15*). The latter contains two interesting subgoals: *Is the type of the argument compatible with the type of the formal parameter?* and *Can unresolved type variables be finally inferred...?*. The latter reveals the exact details of how the type parameter `B` is instantiated from the constraints collected by the compiler therefore giving the source of the expected type in the error message. We delay the discussion on the source of the constraints until the next section.

Even though we support incremental exploration, the disadvantage of mirroring full typechecking is that the initial usage can be challenging for users without sufficient experience with type system implementation and will likely involve some amount of derivation tree guessing. Fortunately as developers of the debugger we can provide automatic search that guides users towards problematic goals.

3. TYPE INFERENCE AND TYPE MISMATCH ERRORS

It has been shown that the larger the number of type variables, the harder it is for users to understand typing problems [Jun et al.(2002)Jun, Michaelson, and Trinder]. Therefore to demonstrate how the tool deals with more a more involved example, we complicated the definition of `foldRight` in our own implementation of the list, `MyList`, as presented in Listing 2. `myFoldRight`, takes an additional type parameter `C` and value parameter `y` and uses it to separate the values of `MyList`.

The type error from Listing 1 remains and reports a type mismatch between `MyList[Int]` and `MyListNil.type`. Selecting the problematic operation again results in a view similar to Figure 1. This time however we right-click the initial adaptation and select **`Where does the expected type come from?`** from the list of available questions. The tool performs a precise search of the derivation tree, taking into consideration how expected type evolved in the context, and expand two goals (rather than randomly picking ones with matching types in a naive approach): *[...] can we type arguments and verify the application?* and *Infer instantiation for type variable ?B ?* that are similar to the ones found in section 2. In short, the debugger will explain that the type of the function part `xs.myFoldRight` still has a generic type `(x: B, y: C)(f: (Int, B, C) => B)C` at the entry point of the goal and that the expected type originates from the `B` result type in the type of the `f` parameter. The second goal directs at the point where type parameter `B` is being instantiated. The search itself can be incrementally expanded and can provide different levels of detail, making it suitable for large typechecking derivations.

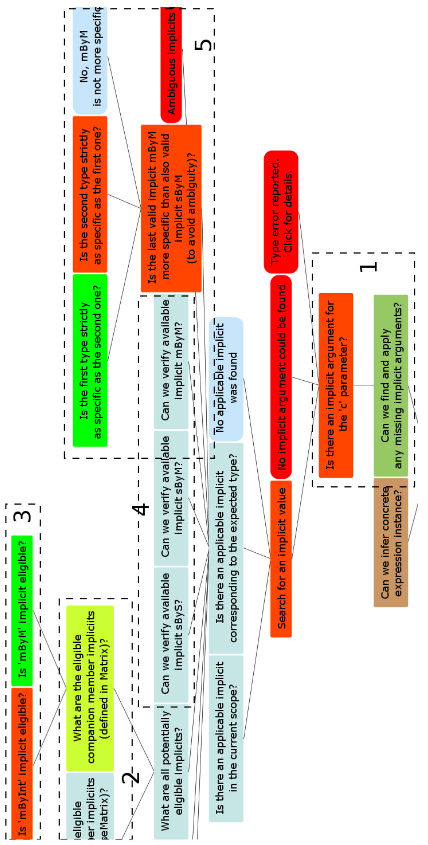
```
class MyList[+T](elems: T*) {
 def myFoldRight[B, C](x: B, y: C)(
        f: (T, B, C) => B): B = ...
 def ::[S >: T](elem: S): MyList[S]   = ...
}
object MyListNil extends MyList[Nothing]
val xs = new MyList[Int]()
xs.myFoldRight(MyListNil, MyListNil)((elem: Int,
 ls: MyList[Int], s: MyListNil.type) =>
       (elem+1)::(s ++ ls))
```

Listing 2: Type-involving definition of foldRight

By asking the tool **`What is the source of constraints used in the type inference?`** on the inference goal (*16* in Figure 2), the type debugger, being aware of the type-checking decision process, can again trace back through the derivation tree. The tool selects a minimal set of affected goals, taking into account such properties as variance of the type variable or elimination of constraints that are subtypes. In the case of the two analyzed programs, the tool expands goals that check compatibility between the type of the argument and the formal parameter (*17* in Figure 2) i.e. `Nil.type <: B` and `MyListNil.type <: B`, respectively. Full explanation emphasizes the limitation of Scala's type inference which eagerly resolves type parameters at the border of consecutive list of arguments. Hence the user finally finds out that the expected type in both cases comes from the first argument in the application. Since real-world examples are rarely so short, one can continue the search until enough information is revealed, for example by asking what affected the type inference of the argument.

Using the preciseness of the search, the tool can propose:

```
Inferred expected type for comes from
xs.myFoldRight(MyListNil, MyListNil)((elem: Int
               ~~~~~~~~~
You may try to annotate the term(s) as:
MyListNil: MyList[Int]
```



No, mByM is not more specific
Ambiguous implicits
5
Is the second type strictly as specific as the first one?
Is the last valid implicit mByM more specific than also valid implicit sByM (to avoid ambiguity)?
Is the first type strictly as specific as the second one?
Type error reported. Click for details.
1
No implicit argument could be found
Is there an implicit argument for the 'c' parameter?
Can we find and apply any missing implicit arguments?
Can we infer concrete expression instance?
Search for an implicit value
No applicable implicit was found
Is there an applicable implicit corresponding to the expected type?
Is there an applicable implicit in the current scope?
Can we verify available implicit mByM?
Can we verify available implicit sByM?
Can we verify available implicit sByS?
4
What are all potentially eligible implicits?
What are the eligible companion member implicits (defined in Matrix)?
2
3
Is 'mByM' implicit eligible?
Is 'mByInt' implicit eligible?

Figure 3: Understanding implicit search ambiguity

The annotation is verified by typechecking `MyListNil` with the *found* type from the error message. When the existing heuristics do not provide an automatic localization of the problem, the users have an advantage that they can always fallback to the (slower) proof exploration technique.

4. UNDERSTANDING IMPLICIT SEARCH

Scala's implicits [Oliveira et al.(2010)Oliveira, Moors, and Odersky] have been shown to allow creation of interesting design patterns. At the same time they are considered as unpredictable due to the number of potential scopes where they are searched for and incomprehensible rules that govern their precedence. Listing 3 presents an example of matrix multiplication by providing the implementation through specialized implicit values, selected by the typechecker based on the type of the arguments and context.

Since implicits are not explicitly imported, beginner users, unaware of implicit scope search may find type correct examples 1 and 2 interesting. It becomes less obvious why calculations involving sparse matrix, **`SMx`** (3 and 4), report ambiguity between **`mByM`** and **`sByS`** implicit values. Figure 3 presents a fragment of implicit search debugging only for function 3, the other case being analogous. The typechecker needs to find an implicit argument for the missing parameter `c` (*selection 1*) and searches for the implicits in the companion objects of the argument types (*selection 3*), since the current lexical context contains none (*selection 2*). The user can analyze how and which implicits are filtered first based on the shape of the expected type (*selection 4*) and later by typing them (*selection 5*). Type debugger can explain in detail how overloading resolution fails to resolve the ambiguity, details are omitted for brevity.

```
trait Mx // Matrix
object Mx {
 implicit val mByM:   Multipl[Mx,Mx,Mx] = ...
 implicit val mByInt: Multipl[Mx,Int,Mx] = ...
}
trait SMx extends Mx // Sparse Matrix
object SMx {
 implicit val sByM: Multipl[SMx,Mx,Mx] = ...
 implicit val sByS: Multipl[SMx,SMx,Mx] = ...
}
trait Multipl[-A,-B,+R]{def comp(a: A, b: B): R}
def multiplyMe[A, B, C](x: A, b: B)
  (implicit c: Multipl[A, B, C]): C = ...

val m  = new Mx {}
val sp = new SMx {}
multiplyMe(m, m): Mx       // 1
multiplyMe(m, m)           // 2
multiplyMe(sp, sp)         // 3
multiplyMe(sp, sp): Mx     // 4
```

Listing 3: Matrix multiplication with implicits

5. RELATED AND FUTURE WORK

The related research has been mostly focused on providing more elaborate and better localized error messages [Chitil(2001)] by modifying the existing type inference foundations. Typically one carries context along the constraints and uses it for specializing messages. Unfortunately more elaborate information does not necessarily guarantee a successful outcome nor better understanding. In case of work on Java Generics messages [Boustani and Hage(2011)], the authors covered only general patterns related to variance, which the type debugger can already explain at the least to the same level of detail.

Chameleon [Sulzmann(2002)] is a type-level debugger for a subset of Haskell. It resembles the capabilities of our tool: users are able to ask for the type of the specific term and its explanation. The latter is performed by highlighting locations that contributed to the inference of a specific type but without any explanation. Chameleon relies on a modified typechecking mechanism that solves constraints through rewrite rules. Scalad, though highly influenced by Chameleon, focuses on explaining a full, existing type system implementation. Translating typechecking to constraint solving becomes impractical in our case due to the existence of at least subtyping tests [Odersky et al.(1999)Odersky, Sulzmann, and Wehr]. Scala users are often surprised by not a single type system feature but an interaction of multiple ones (as given in section 4). We believe that the visualization aspect, which unifies the explanation, is much better suited for this task and leaves a lot of room for improvement relative to Chameleon. Although still limited, our search guides provide comparable exploration flexibility to Chameleon.

Helium [Heeren et al.(2003)Heeren, Leijen, and van IJzendoorn] offers a language similar to Haskell but was built with the focus on providing better feedback to the beginner programmers. It utilizes a database of common user mistakes to produce more specialized messages. Scalad would definitely benefit from having such a database - it would allow to better cover most problematic cases that are currently missing in our heuristics.

The current guiding mechanism has been extended to deal with typical scenarios involving overloading resolution or eta-expansion but a more thorough coverage of the type system capabilities is needed. Only due to temporary limitations the current command-line suggestion mechanism offers suggestions for inference problems consisting of at most one constraint. This is still enough for beginner users who initially only strive to quickly fix their typing problems. Better visualization of more complicated type system features like pattern matching, existential or path-dependent types relies on having more coarse grained instrumentation. The latter is an incremental process and we successively extend the tool for that purpose.

6. CONCLUSIONS

Scalad is an interactive type-level debugger that allows to debug incomprehensible errors and better understand type constructs that were until now only reserved for experienced programmers. The tool relies on an unmodified type system implementation therefore enabling the programmers to understand current type system limitations. Scalad allows for selective debugging of type correct and erroneous code fragments and visualizes the information in a convenient proof tree form. We have presented a small number of available helpers that precisely locate the goals and answer questions typical for typechecking. For less sophisticated scenarios programmers can rely on the command-line suggestions from the type debugger. A more thorough tutorial for the tool as well as the latest snapshot is available at http://lampwww.epfl.ch/∼plocinic/type-debugger-tutorial/.

7. REFERENCES

[Boustani and Hage(2011)] N. Boustani and J. Hage. Improving type error messages for generic java. *Higher Order Symbol. Comput.*, June 2011.

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

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

[Jun et al.(2002)Jun, Michaelson, and Trinder] Y. Jun, G. Michaelson, and P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45:2002, 2002.

[Odersky et al.(1999)Odersky, Sulzmann, and Wehr] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

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

[Plociniczak and Odersky(2012)] H. Plociniczak and M. Odersky. Implementing a type debugger for scala. In *APPLC*, 2012.

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

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