Improving Human-Compiler Interaction Through Customizable Type Feedback

Hubert Plocliniczak  Heather Miller  Martin Odersky
Ecole Polytechnique Fédérale de Lausanne
firstname.lastname@epfl.ch

Abstract

Type errors reported by compilers can sometimes be cryptic, or difficult to understand. In this paper, we propose a type debugging framework that exposes a high-level representation of the typechecking decision-making process that users normally do not have access to in state-of-the-art compilers. This representation makes it easier for non-experts to analyze complex type errors. Our system is implemented by instrumenting the existing Scala typechecker, but without modifying it. We also provide generic search algorithms that can be used as basic building blocks to build a number of systems, from visual type debugging tools to customized error reporting for normal Scala users as well as to users of domain-specific languages. Using our framework, it is possible to overcome well-known limitations of local type inference by providing precise type error messages to mainline Scala users or Scala DSL users alike. In addition, the framework provides better user feedback for non-trivial type errors from existing Scala libraries and DSLs.

Keywords  type system, type errors, type debugging

1 Introduction

Writing software using statically-typed languages inherently involves fixing type errors. The subject of what constitutes a good error message, and therefore properly directs the user, has been well studied [14, 32]. Yet, in practice, it is hard to achieve both generality (i.e., typechecking rules should apply to as many conflicts as possible) and precision (i.e., type errors convey enough information for users to be able to easily navigate to and fix the error) without affecting the performance of regular compiler runs. Since language architects and compiler writers must provide enough information to assist both novice, as well as expert users (such as embedded DSL authors in Scala), achieving this trade-off is even more difficult.

An example of an innocuous code snippet which leads to a puzzling type error is shown in Listing 1. In this example, we simply have a list of integers that we want to increment using the foldRight method from the Scala standard library. The call to foldRight traverses the elements of a List from right to left, starting with the first argument as the first element, in this case Nil (an empty List), and applies the second argument, a function, to each element, which in this case appends the incremented elements.

Listing 1. Incrementing a list of integers gone wrong

val xs: List[Int] = List(1, 2, 3)
xs.foldRight(Nil)((x, ys) => (x + 1) :: ys)
// type mismatch;
// found : List[Int]
// required: Nil
// (x + 1) :: ys)
//

1 Source file, line and/or character position.
The framework is able to suggest the following helpful type inference example in Listing 1. In this case, building on the derivation tree with custom error feedback, accessible to users via an API for traversing, analyzing, and tree, and for adding custom type errors. This information is a set of generic routines for analyzing the type derivation high-level representations of typing decisions, we provide coming instantiations of typing rules. Built on top of this construction of a high-level representation of typing decisions that is obtained directly from derivation trees built from the actual type checking process, that (b) we show can be used as the foundation for many systems to interact with and customize type error feedback, from graphical type debugging tools, to domain-specific type error customizations. Crucially, our approach is based upon the construction of a high-level representation of typing decisions that is obtained directly from derivation trees built from the result of the compiler's type checking, all without modifying the logic of the existing typechecker or reducing its features. To achieve this representation, we map low-level, context-free typechecking information, extracted from instrumented compiler runs, into what essentially becomes instantiations of typing rules. Built on top of this high-level representations of typing decisions, we provide a set of generic routines for analyzing the type derivation tree, and for adding custom type errors. This information is accessible to users via an API for traversing, analyzing, and building on the derivation tree with custom error feedback.

We achieve a sweet spot between generality and precision, such as in the case of the cryptic error message from the type inference example above in Listing 1. In this case, the framework is able to suggest the following helpful type ascription:

```
// Inferred expected type using the following location(s):
// xs.foldRight(Nil)((x: Int, ys: List[Int]) => (x + 1) :: ys)
// ~~~
// You may try to annotate your code like: 'Nil:List[Int]
```

Beyond such type ascriptions, helpful user feedback can include program slicing, descriptions of the type derivation tree, suggestions on how to better guide type inference, or even graphical tools for exploring the type derivation tree.

Finally, our framework is not limited to users of the mainline Scala distribution. Domain-specific library (DSL) authors with shallow and deeply embedded DSLs are also able to take advantage of customized type error analysis and reporting since the framework doesn’t require or include a duplicate typechecking pass, it uses the normal typechecker.

### 1.1 Selected Related Work

Chameleon [30] translates Haskell’s global type inference into a constraint solving problem. During typechecking, the Haskell compiler will generate a large number of constraints, each of which carries source location information for debugging purposes. Debugging essentially involves finding a minimal set of constraints that can explain a typing error using a fixed set of constraint reduction rules. Our framework is focused on object-oriented languages, where it has been shown [27] that local type inference is more suitable in terms of providing better localized error messages and that handling subtyping constraints is still tractable [24].

The work of El Boustani and Hage [1] is the only work, to the best of our knowledge, which focuses on modern object-oriented programming languages that support parametric and subtype polymorphism. They present a number of heuristics that improve error messages for Java with Generics by implementing a typechecker which collects subtyping constraints for very localized problems, specifically only method invocations that have local type parameters and variance. Our approach crosses the boundaries of function application, and therefore performing a more global type error analysis. In fact, our approach can also be used for explaining typing decisions related to a much richer type inference mechanism, and other type system features such as overloading, higher-ranked types and implicit search. In our evaluation, we show that heuristics used in [1] can be implemented in our framework.

Tsushima et al. [33] attempt to build type derivation trees by treating an existing typechecker as an oracle that recursively typechecks subtrees of the original program. The reconstruction of this type derivation tree is performed by interacting with users. While the main idea of our framework is to reconstruct similar typing rules, we avoid any user interaction which would be infeasible when attempting to visualize and debug advanced type system features in a straightforward way [28].

### 1.2 Contributions

This paper makes the following contributions:

- (1) A novel approach to type error customization and debugging framework that is based on type derivation trees from the actual type checking process (achieved by instrumenting the Scala compiler in a lightweight way), and (2) a programmer-accessible high-level representation of typechecking decisions. Thus, our high-level type-
checking representation reflects the behavior of the real world Scala typechecker, while still supporting the full set of Scala language features.

- A set of programmer-facing generic routines that make it possible to traverse and analyze the high-level type-checking representation and to suggest and verify code modification for type errors. Such routines can be used to build a number of useful system for interacting with and customizing type error feedback, from graphical type debugging tools to type analysis plugins for IDEs.

- A plug-in architecture that can be used by the authors of libraries and embedded DSLs. This makes it possible for developers of such DSLs to provide additional domain-specific error analysis and type errors to provide their users with comprehensive and accurate domain-specific feedback, rather than relying on the basic error-reporting infrastructure of the host language.

- An extensive real world validation of these techniques: we integrate our framework into the full-fledged Scala compiler, and perform a detailed analysis of several advanced Scala libraries and frameworks which make advance use of the type system.

2 Type Derivation Trees: A Basis

Typechecking a program can be thought of as instantiating terms and types into the general inference rule, and then by analyzing it, to locate the typing error. We'll achieve this by first constructing a type derivation tree which we will use to illustrate how such trees can aid in type debugging.

2.1 Formalization

Since the cryptic type error shown in Listing 1 is caused by a known limitation of Scala’s local type inference, we adopt the formalism and type system described in Odersky et al.'s treatment of Scala’s colored local type inference [25] to build our type derivation tree.

For the sake of brevity and clarity, we present a fragment of the typing rules of local type inference, as defined in [25].

### Definition 1

Core language syntax, from [25, pg 3].

<table>
<thead>
<tr>
<th>Terms</th>
<th>$E, F = \begin{cases} x : E_x &amp; x \in \mathcal{E} \ \text{fun}[\pi](x : T)E &amp; \text{fun}(x)E \ P[E] &amp; \text{P}(E) \ E.x &amp; {x_1 = E_1, \ldots, x_n = E_n} \end{cases}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>$T, S, R = \begin{cases} \text{a}[\Gamma] &amp; {a \in \text{A} \downarrow } \ T \rightarrow \downarrow S &amp; {x_1 : T_1, \ldots, x_n : T_n} \end{cases}$</td>
</tr>
<tr>
<td>Environments</td>
<td>$\Gamma' = x : T</td>
</tr>
</tbody>
</table>

### Grammar

Definition 1 shows our core language syntax, which itself is based on $F_\leq$ extended with records, as per [25, 27].

The grammar gives terms, types and environments of the language. A term can be a variable $x$, a record constructor $\{x_1 = E_1, \ldots, x_n = E_n\}$ or record selection $E.x$. It also has two versions of function application and abstraction: ones with explicit type parameters and type arguments ($P[T]\{E\}$ and $\text{fun}[\pi](x : T)E$) and those that elide them, if possible, by conveniently inferring them from the context ($P[E]$ and $\text{fun}(x)E$), respectively. The overbar $\bar{\pi}$ signifies a finite sequence of local type parameters, and is equivalent to $a_1, \ldots, a_n$ for some $n$. A type is either a type variable $\text{a}$, a top or bottom type in the type hierarchy ($\top$ and $\bot$ respectively), a potentially polymorphic function type $T \xrightarrow{a} S$ (with type variables written over the arrow), or a record type.

### 2.1.1 Typing Rules

A fragment of the local type inference type system is presented in Figure 1.

The inference judgment, $P, \Gamma \vdash^w E : T$ consists of the term to be typed, $E$, that eventually is assigned type $T$ in a type environment $\Gamma$, and a prototype $P$ representing parts of the type of $E$ that are inherited from the context. $P$ can be treated as a normal type, potentially having some type constants $? \in \text{place of the missing parts}. For instance, given a judgment of $\text{Int} \rightarrow ?, \epsilon \vdash^w \text{fun}(x) : \text{Int} \rightarrow \text{Int}$ we know that the type of the only parameter $x$ is $\text{Int}$ and is inherited, whereas the return type is synthesized from the body of the function.

Since we are only interested in constructing a type derivation tree for the $\text{foldRight}$ example shown in Listing 1, we only discuss rule (app) for function application with elided type arguments. The first premise of (app) expresses typechecking of function $F$ without any prototype, and it requires the inferred type to be of a function type $S \rightarrow T$. Hence, in the second premise, the argument can be typed with an expected type coming from the just resolved function, with all the unknown type parameters substituted by constant type $?$. Matching between the expected prototype and synthesized type is expressed through operator $\nearrow$. $T \nearrow P$ means
that either \( T \) is structurally equal to \( P \), with \(?\) filled by some arbitrary types, or we can find the smallest supertype of \( T \) which is structurally equal to \( P \). The operation \( T \bowtie P \) is the dual of \( \nabla \), where the greatest subtype of \( T \) is structurally equal to \( P \).

2.2 Constructing the Type Derivation Tree

Consider the following definition of the \texttt{foldRight} function mentioned earlier:

\[
\texttt{foldRight} = \text{fun}[a] (\text{elems: List}[a]) \text{ fun}[b] (\text{acc: b})
\]

\[
\text{foldRight} \{ \begin{align*}
\text{caseNil}() &= \text{acc} \\
\text{caseCons}(x, y) &= f(x, \text{foldRight}(y)(\text{acc})(f))
\end{align*} \}
\]

This definition uses a straightforward encoding of lists (summarized in Appendix B).

\texttt{foldRight} simply traverses the initial list of elements, \texttt{elems}, until an end is reached, \texttt{caseNil}, where it returns the initial accumulator. Then it will recursively backtrack, and apply function \( f \) to the intermediate result of applying \texttt{foldRight} to the result of the suffix of the list. Using the typing rules from [25, pg. 6], function \texttt{foldRight} has type, \( \text{List}[a] \rightarrow \text{b} \rightarrow (\text{a, b} \rightarrow \text{b}) \rightarrow \text{b} \), which allows us to express the erroneous application from Listing 1 as:

\[
\text{foldRight}((\text{Cons}(1, \text{Nil}())(\text{Nil}()))((x, y) \rightarrow \text{Cons}(x+1, y)))
\]

By applying the typing rules from Figure 1 to this definition of \texttt{foldRight}, we can construct the type derivation tree shown in Figure 2.

2.3 Using Type Derivation Trees to Debug \texttt{foldRight}

The typing error from Listing 1 is highlighted in the derivation tree in Figure 2 as (\textit{type-mismatch}). It results from the structural inequality between the inferred type, \( \text{List}[\text{Int}] \), of the body of the function which actually performs the list manipulation, and an inferred expected type, \( \text{List} [\bot] \).

\texttt{List}[\text{Int}] is inferred because lists are covariant in their type parameter, and the least substitution of \( \text{Int} \bowtie a \) and \( \bot \bowtie a \) for \( a \) is \( \text{Int} \), which means that the conflicting type originates in a different location in the type derivation tree.

In order to debug this, one has to figure out where the \textit{prototype} \( \text{List} [\bot] \) comes from. Type derivation trees allow us to backtrack through the type inference process, and to see how individual premises of the typing rules were arrived at. For the sake of clarity, in the figure, we highlight the types which led to the type mismatch in gray boxes, and use numerical superscripts to illustrate the order of our explanation.

Function \( f \) is the argument of the application of \( \text{foldRight}(\text{xs})(\text{Nil}()) \), which inherits an expected type \( (\text{Int}, \text{List}[\bot]) \rightarrow \text{List}[\bot] \) due to the (abs) rule from Figure 1. This type comes from the inference of the type of the function \( \text{foldRight}(\text{xs})(\text{Nil}()) \) (superscript 4). There, we can see that the type responsible for the type mismatch was instantiated from the type parameter \( b \) (superscript 5), which in turn resulted from the application of the partially applied function \( \text{foldRight}(\text{xs}) \) to \( \text{Nil}() \). In this case, the type of the argument, \( \text{Nil}() \), is simply \( \text{List}[a] \). As previously alluded to, since variable \( a \) is in a covariant position, and carries no other constraints, the most appropriate solution coming from \( \bot \bowtie \) results in the substitution of \( a \) to \( \bot \). This, in effect, imposes the only constraint on type variable \( b \), which is \( \text{List} [\bot] \bowtie b \). This constraint results in the highlighted substitution (superscript 10) that eventually leads to a type mismatch.

Given such a simple code snippet, one might be surprised at the complexity of the fragment of the type derivation tree shown in Figure 2. While presenting raw type derivation trees to users is certainly impractical, we nonetheless argue that type debugging via these type derivation trees is powerful. Traversing and analyzing such trees can provide essentially any information about the typechecking process, from a detailed picture of how language features interact, to the revelation of limitations of local type inference. Furthermore, as one might be able to infer from the grayed-out types in Figure 2, type derivation trees can be traversed and pruned so as to simplify and speed up the localization of types of interest, as we will show in section 4.1. We will also show in later sections how this approach serves as a foundation for a number of different systems to customize or debug type errors, from interactive visualization tools for type debugging [28], to custom domain-specific type errors for embedded DSLs.

3 From Typechecker to Type Derivation Tree

A central novelty of our approach is that a high-level representation of typing decisions is obtained directly from the result of an existing compiler’s type checking process, all without modifying the logic of the existing typechecker or reducing its features. Throughout the remainder of the paper, in the context of the type debugging framework, we refer to typing rules as \textit{goals}, which can have some \textit{premises} that, by definition, must be fulfilled for the rule (or goal) to be applied.

\[
\frac{P, \Gamma ; x:T \Downarrow E : S}{T \Rightarrow P, \Gamma ; x : T \Downarrow E : S}
\]

\[
\frac{?; P, \Gamma ; w : T \Downarrow E : S}{?; P, \Gamma ; w : T \Downarrow E : S}
\]
Typechecking function argument \( f \)

\[
\begin{align*}
&\text{(var)} ?, \Gamma \vdash ^\omega \text{Cons} : \\
&\quad (\{a, \text{List}[a]\} \rightarrow \text{List}[a]) \\
&\text{(app)} ?/a \ a, \Gamma \vdash ^\omega (x + 1) : \text{Int} \Rightarrow a \\
&\quad (\text{var}) ?/a \ \text{List}[a], \Gamma \vdash ^\omega y : \text{List}[\bot] \Rightarrow \text{List}[\bot] \\
&\quad \Gamma_1 = \Gamma, x : \text{Int}, y : \text{List}[\bot] \\
&\text{List}[\bot] : \bot \\
&\quad \text{List}[\bot] < a \Rightarrow C_1 \\
&\quad \Gamma_1 \vdash ^\omega ?/\text{List}[\bot] \\
&\quad \text{List}[\bot] \vdash ^\omega ?/\text{List}[\bot] \\
&\quad \text{List}[\bot] \vdash ^\omega \text{fun}(x, y) \rightarrow \text{Cons}(x + 1, y) : \text{Error} \\
&\quad \text{Typechecking application of \text{foldRight}(xs)(\text{Nil}())} \\
&\quad \text{Typechecking function argument } f \\
&\quad \text{Typechecking information from within the body of methods is not possible} \\
&\quad \text{extract compiler data at the entry and exit point of method calls; obtaining} \\
&\quad \text{typechecking information from within the body of methods is not possible} \\
&\quad \text{to recover without reconstructing the logic of the mainline typechecker.}
\end{align*}
\]

Figure 2. Fragment of type derivation tree for application \( \text{foldRight}(xs)(\text{Nil}())(f) \), where \( xs = \text{Cons}(1, \text{Nil}()) \) and \( f = \text{fun}(x, y) \rightarrow \text{Cons}(x + 1, y) \)

Our type customization and debugging framework builds on top of the existing implementation of Scala [22], a hybrid object-oriented and functional language. Apart from having Java-like features and running on the JVM, Scala has an advanced type system which includes features such as higher-kind types [18], path-dependent types [21] and implicits [26], all of which are supported by our framework.

In this section, we will step through the construction of this high-level representation, beginning in section 3.1 with details of our compiler instrumentation API. Section 3.2 discusses how goals are defined in our high-level representation. Finally, section 3.3 describes how we can transform low-level data obtained from instrumenting the Scala compiler to high-level premises and goals introduced in section 3.2.

3.1 Instrumenting the Compiler

The type debugging framework collects low-level typechecking information by instrumenting the existing Scala compiler using a minimal API, a set of low-level instrumentation classes, and an infrastructure for debugging. A small example is shown in Listing 2, where we instrument the body of a method that typechecks some AST (tree) given a prototype (tpe).

In this example 2, an EV object collects all instrumentation notifications. TyperTypecheck, AstTyped, and TyperTypecheckFinished of type Event are concrete instrumentation classes that carry raw typechecking information, such as type or symbol references.

We chose to manually instrument the Scala compiler since the alternative is to modify bytecode (e.g., http://eclipse.org/aspectj/), which is too coarse-grained. That is, bytecode modification provides means to extract compiler data at the entry and exit point of method calls; obtaining typechecking information from within the body of methods is not possible to recover without reconstructing the logic of the mainline typechecker.

def typecheckAst(ast: Tree, tpe: Type): Tree = {
  EV "<< TyperTypecheck(ast, expectedType)
  ... // instrumented typing of ast
  EV "<< AstTyped(...) 
  ... 
  EV ">> TyperTypecheckFinished(...) 
  ...
}

Listing 2. A brief look at the Instrumentation API.

Importantly, we introduce the notion of an instrumentation block, which makes it possible for structural information to be collected during instrumentation, which results in instrumentation data that more closely reflects the premises-conclusion relations in the typing rules, as opposed typical “flat” instrumentation data. These instrumentation blocks are delimited by the “<<” and “>>” operators, and typically also contain other (potentially nested) instrumentation blocks, delimited using the same operators, as well as single instrumentation points (defined using the “<” operator). As a result, the framework understands that direct instrumentation points between the “<<” and “>>” method calls can be considered as typechecking dependencies, without having direct references to them in the source code.

While stuctured, the type of information that instrumentation collects is quite low-level. Raw compiler data, such as ASTs that are being typechecked, expected types inherited from the context, or type parameters to be instantiated, are among the bits of low-level data collected. Indeed, this sort of data is a far cry from the high-level type derivation trees constructed from that we’re after, as it carries no knowledge of related premises that need to be satisfied, nor of enclosing typechecking context information.

Finally, while instrumentation does incur non-negligible compilation time performance penalties while collecting
data, our framework manages to sidestep noticeable slowdowns by selectively enabling instrumentation only when typechecking regions of the source code that directly affect the detected errors. We additionally save performance by making use of the fact that definitions in the Scala compiler are initialized in a lazy manner i.e., their type signatures are verified or inferred once per compilation, at the first point when they are referenced during typechecker’s run.

3.2 High-Level Typechecking Representation
In this section, we provide a brief overview of our high-level typechecking representation, based on type derivation trees. In section 3.3, we delve into the details of our high-level representation and how it maps to our low-level instrumentation data.

Listing 3 provides a class for a Goal (or typing rule) that has a reference to all its premises, and a conclusion, parent, which is also of type Goal. Subclasses of Goal and their members provide concrete requirements for each of the typechecking decision points. Nodes of a type derivation tree can be constructed from subclasses of Goals which represent numerous concepts such as typing of an abstraction or typing of function application.

Typecheck is an example of such a subclass. In order to satisfy this goal, for example, its members require that it first be typed and then adapted. Note that the process of typing and adaptation in this example are represented via the types of Typecheck’s members, which are also subclasses of Goal.

3.3 Mapping From Low-Level to High-Level
To understand how we map from low-level instrumentation data to high-level type derivation trees, we provide an explanation from two different angles. We first begin with a side-by-side example of instrumented code, and high-level goals in Figures 4 and 5, respectively, which gives an intuition for how our implementation achieves this mapping. We then provide a formalization of this mapping, in an effort to provide a complete and general depiction of how our framework translates from low-level instrumentation events to high-level goals and premises.

3.3.1 Instrumentation to Goals & Premises: An Example
Listing 4 presents a simplification of the actual Scala compiler’s implementation, and instrumentation, of typing function application. In this example, the method typeApplication takes an argument, ast, which is an Apply AST node (an AST node that represents function application). This AST has two parameters, the AST for the function, and the AST for the function’s arguments, both of which are extracted via a pattern match on line 2.

The first instrumentation block is delimited by instrumentation classes TypeApplication and TypeAppDone. This creates a logical block of instrumentation which essentially specifies that any instrumentation executed in between is part of the logic responsible for typing an function application. The function itself is typechecked via method typecheckAst, which is itself also instrumented (not shown).

The rest of Listing 4, beginning on line 4, represents the logic for actually typing function application. Lines 5, 11, and 14 illustrate different possible paths through this typechecking method; a function type having type parameters (PolymorphicType, line 14), a monomorphic function type (MethodType, line 11), and an erroneous function type (ErrorType, line 5). Importantly, each different path through this typeApplication method result in the instantiation of different sorts of instrumentation classes.

Now that we’ve seen an example of a typechecking method, we now look at the high-level representation that we’ll create from the low-level instrumentation events.

Listing 5 gives a high-level representation for typing function application. We use inheritance to abstract over different typechecker executions (one can think of traits in Scala as a simple Java interface in this context). These interfaces give users of the high-level representation a handle on the possible decisions the typechecker can make. As we will see in section 4, this also enables users to pattern match [7] on these decisions.

In Listing 5, a TypeApp trait states that any typing of function application has to always first typecheck the function (member typecheckFun) and directly corresponds to the first premise of typing rule app in Figure 1. Traits TypeAppFallback and TypeAppCorrect correspond to different typechecker executions involving erroneous and error-free results of typechecking a function. Trait TypeApplicationMain, whose type is listed as the type of a premise of TypeAppCorrect, represents the typing of an application given a (valid) function type. Its subtraits TypeAppMonomorphic and TypeAppPolymorphic correspond to executions involving typing of monomorphic and polymorphic applications, respectively. This distinction between subtraits of TypeApplicationMain directly reflects the presence, or lack thereof, of type variables in the inference rule (app). Member targsFromExpectedType corresponds to the potential inference of type arguments from the result type of the method and the expected type (recall InferTypeArguments in Listing 4), and member inferMethInstance corresponds to the act of inferring type arguments from the typechecked arguments and formal parameters.

In order to relate these low-level instrumentation events to this high-level representation, we return to the low-level in-
Listing 4. Example of instrumented typing of function application

stratification data. These low-level instrumentation events collected by instances of EV are essentially a sequence of type checking events, which carry low-level information like symbols and expected types. Importantly, these events are delimited by methods <<< and >>>, and instances of instrumentation classes. These delimiters enable us to reconstruct the nesting structure characteristic of type derivation trees.

In order to determine which high-level goal a given instrumentation event corresponds to, we pattern match on these sequences. If we succeed in matching, we create a corresponding high-level representation. In the following section, we’ll see how this matching works. These high-level representations refer back to the low-level events that they were created from. Subtle differences between low-level instrumentation, especially in the presence of type errors, result in different high-level type derivation trees.

3.3.2 Formalization

In this section, we formalize a mapping function which maps instances of low-level instrumentation into its high-

Listing 5. A fragment of the high-level representation corresponding to typechecker decisions necessary to type a function application.

level counterparts, in a one-to-many relation. The algorithm transforms low-level instrumentation recursively in a depth-first postfix manner. In other words, for any block of instrumentation data, we first map all low-level instrumentation contained within it, and then use that information as a context for the selection of a single high-level representation, based on the types of members of potential high-level goals.

Definition 2. High-Level Representation & Dependencies

Definition 3. Mapping: Type Signatures

Definition 4. Mapping: Function Definitions

2014/3/26
The premises function returns a sequence of all required premises for a given static type \( T \), respecting the order of declarations (\( T < \) Goal) and \( T \) is among super types of \( T \) and \( e \) represents an empty sequence) e.g., in the case of TypeAppCorrect it would be only type TypeApplicationMain. The linearization function returns the chain of super types up to type Goal. Function spec follows the linearization chain to retrieve all (declared and inherited) types of premises. targetT function declares possible high-level representations for every low-level instrumentation (it is defined by compiler experts). In order to express the accuracy of mapping a low-level instrumentation to some concrete high-level representation, we use an internal data structure of MatchR. MatchR represents the result of matching actual premises to required ones i.e., how already mapped low-level instructions within the block (recall the postfix order of mapping), fit with members of high-level Goal. It can be either Success, meaning that the mapping function has found real dependencies (\( S \)) for each of the required premises for the Goal of type \( T \), or Partial, meaning that only some (\( S \)) have been found.

The function matchF(ps)(ts)(g) attempts to map each required premise \( t \) to an actual premise \( p \) in \( ps \). This mapping respects the order of the actual and required premises. To express the properties of matchF more formally, we use a function match of type \( \text{MatchR, Int} \Rightarrow \top \). It extracts a sequence of actual premises, \( \overline{Y} \), that have been associated with a required premise at a provided index. Using match, we can express the order-preserving mapping of matchF as follows:

1. \( \forall i \forall j, j > i \land \text{match}(m, i) = \overline{Y} \land \text{match}(m, j) = \overline{Y} \Rightarrow (\forall k, l Y_k \in \overline{Y} \land Y'_l \in \overline{Y} \Rightarrow l > k) \)

2. \( \forall i, \text{match}(m, i) = \overline{Y} \Rightarrow (\forall a \forall b, a < b \land Y^a_k \in \overline{Y} \land Y^b_l \in \overline{Y} \Rightarrow k < l) \)

The subscript \( j \) in \( Y^j_k \) represents an index of the premise \( Y \) within an initial sequence of ps, while superscript \( k \) represents an index of an element within sequence \( \overline{Y} \). The first statement ensures that actual premises respect the order in which they were mapped to required premises, while the second ensures that the order within a sequence associated with a member is also preserved.

**Sorting Matching Results** The role of function matchF is only to deterministically associate required with actual premises for a single high-level representation. It does not guarantee that it will always return a single Success object if we apply it to different high-level Goals coming from targetT function. In fact, since we do not restrict in any way how high-level representations are defined, or what mapping is provided by compiler experts, we can have multiple Success results for a one-to-many mapping between the representations or none at all. Thus, in order to disambiguate, we sort the matching results using compare.

The first parameter of compare expresses the fact that the comparison is bound with a particular instance of low-level representation. The compare function returns a negative value, zero or a positive value as the second argument is less specific than, equal to, or more specific than the third one. The sorting function sort takes a sequence of (possibly) ambiguous results of matching the premises, and sorts them using this instance-bound compare.

**The Final Mapping Step** The mapping function map in Definition 4 extracts the first, most specific, matching result and returns the high-level representation associated with it using the head function. As a result, map is capable of reconstructing type derivation trees by mapping low-level instrumentation to a high-level representation. As (potentially) ambiguous mappings are detected during the head operation since it checks if the next matching result in sequence is strictly less specific.

## 4 Programatically Exploring Typechecking

Our framework implements a number of generic search algorithms on top of a high-level representation that can answer common typechecking questions in the form of located Goals. In section 4.1, we describe a generic technique that uses the shape of the type derivation tree and the types of terms to precisely prune the search space. Section 4.2 gives an overview on how to combine this searching and pruning with the high-level representation to understand decisions made by typechecker. Section 4.3 gives an example on how we can analyze type constraints used in type inference, and finally, in section 4.4 we show how our framework can suggest and verify code modification for type errors.

### 4.1 Pruning the Type Derivation Tree Search Space

Brute force navigation through the high-level representation is impractical for even small programs, while using heuristics that perform undirected depth-first search over type derivation trees is imprecise. We present a simple mechanism that we call TypeFocus, which allows us to gradually build knowledge about the evolution of a type of interest as we step through typing rules and at the same time prune the search space.

TypeFocus is a function that takes a type and extracts some part of it based on its shape. Definition 5 provides examples for function and polymorphic function type (Error simply means that type extraction was invalid). We create instances of TypeFocus to exploit the fact that parts of prototype are used in a consistent way in the premises of typing rules e.g., abstraction, (abs), uses only the result type of the prototype. More importantly, we can freely compose TypeFocus i.e., the composition of TypeFocus, say \( f \) and \( g \), applied to some type. \((f \circ g)(tp)\), is equivalent to \( f(g(tp)) \). This gives us an opportunity to create compositions of TypeFocus instances from multiple, potentially distant, typing rules and still extract specific portion of a type.

Applying TypeFocus to types of terms allows us decide whether a potential type inference, that is performed as part of a typing rule, should guide our type derivation tree exploration. During the first stage of type mismatch explo-
unlike type inference, the judgment infers a sequence of
is because a particular function type may contain type vari-

Definition 5. Type focus.

\[ \text{Focus} : T \Rightarrow T \]

\[ \text{foc-fun-res}(tp) = \begin{cases} 
S & \text{if } tp = T \Rightarrow S \\
\text{Error} & \text{otherwise}
\end{cases} \]

\[ \text{foc-fun-param}_n(tp) = \begin{cases} 
T_n & \text{if } tp = (T_1, \ldots, T_n, \ldots, T_m) \Rightarrow S \\
\text{Error} & \text{otherwise}
\end{cases} \]

\[ \text{foc-poly-res}(tp) = \begin{cases} 
T \Rightarrow S & \text{if } tp = T \Rightarrow S \\
\text{Error} & \text{otherwise}
\end{cases} \]

\[ \text{foc-all}(tp) = tp \]

ration, we want to understand how parts of the prototype
have been used to typecheck the argument so that we can
build an appropriate TypeFocus composition. Later, we can
engage TypeFocus, by applying it to types of terms, in order
to identify premises that perform type inference.

4.1.1 Building TypeFocus for Type Mismatch Errors

Definition 6 presents a recursive build function that back-
tracks from an actual error to function application. During
this process, we construct an appropriate composition of
TypeFocus instances based on the sort of typing rule used.
Such construction stops at the function application typing
rule because we know that its prototype does not affect the
prototype used for typing the erroneous argument. The func-
tion parent-rule returns a reference to the instance of a
parent typing rule it is part of in the derivation tree. Helper
functions, arg\_n\_error and result-type-error, allow us to de-
determine whether an error occurred in a premise that types an
argument or simply because of the method result type mismatch.

Definition 6. Building TypeFocus from type mismatch:

\[ \text{build}(\text{typing-rule} \Rightarrow \text{Focus} \Rightarrow \text{Focus}) = \text{build}(\text{parent-rule}(\text{abs}))(f \circ \text{foc-fun-res}) \]

\[ \text{build}(\text{app})(f) = \begin{cases} 
f \circ \text{foc-fun-param}_n & \text{if } \text{arg\_n\_error}(\text{app}) \\
f \circ \text{foc-fun-res} & \text{if } \text{result-type-error}(\text{app})
\end{cases} \]

\[ \text{build}(\text{var})(f) = \text{build}(\text{parent-rule}(\text{var}))(f) \]

build for a derivation tree that typechecks an argument
of type mismatch from Figure 2 will therefore construct
composition \(\text{foc-all} \circ \text{foc-fun-res} \circ \text{foc-fun-param}_0\).

4.1.2 Analysis of Type Derivation Trees Based on

TypeFocus

As each of the typing rules assigns a type to a term, we
can use TypeFocus in order to understand if typing rules
perform type inference that later leads to a type error. Figure
3 defines analysis for each of the typing rules present in the
local type inference formalization. The analysis judgment,
\(\text{Fc}, P, \Gamma \vdash \tau, S\), is similar to the type inference judgment,
eXcept for TypeFocus \text{Fc}, which guides exploration. Also,
unlike type inference, the judgment infers a sequence of
types, \(S\), that affect the inference of a type extracted by
TypeFocus.

In the application rule, \((\text{APP–FOC})\), we use the informa-
tion extracted from the function type to determine if type
inference should affect our derivation tree exploration. This
is because a particular function type may contain type vari-
ables that are of no importance to us. If that is the case,
we continue exploration of the function with an updated
TypeFocus.

In the case that our derivation tree for \text{foldRight}(\text{xs})(\text{Nil}())
represents function application, the composition of TypeFocus
would be \(\text{foc-all} \circ \text{foc-fun-res} \circ \text{foc-fun-param}_0 \circ \text{foc-fun-res} \circ \text{foc-poly-res}\). Applying it to the type
of the function, \(b_1 \rightarrow ((\text{Int}, b_3) \rightarrow b_2) \rightarrow b_5\), results in a
still uninstantiated type variable \(b_4\) (indexes in type variable
are only for presentational reasons). Since we have extracted
type variables from the type using the TypeFocus, we know
that the type inference that will be performed as part of that
typing rule will also instantiate them, end eventually lead to
type mismatch.

Rule \((\text{APP\_fo-FOC})\), gives an example where applying
TypeFocus identifies explicit type arguments that will lead
to type errors.

The framework provides also a more refined TypeFocus
that is built from a high-level representation of failed algo-
rithmic subtyping operations. This allows for even more pre-
cise error reporting but also requires some refinements in
existing judgment, hence we describe the details only in Ap-
pendix C.

4.2 Exploring Type Derivation Trees

In this section, we show that our high-level representation
and TypeFocus do a good job of implementing exploration
techniques in a generic way, i.e., without making any as-
sumptions about the source code or user input.

In order to express analysis rules our framework conve-
niently allows for pattern-matching on instances of \text{Goals} to
extract their premises\(^3\).

Listing 6 shows that we can conveniently enumerate
through a list of typing rules by pattern matching on \text{goal}
in line 2. While we match on \text{Goals} of typing conditionals,
member selection, variables, and application, respectively,
we can integrate analysis for further typing rules by simply
adding new pattern matching cases. An example of such ex-
tension is given in method \text{applicationContext}, which
essentially implements the analysis rule \text{APP–FOC}. Apart
from the already-discussed function applications, we now
also pattern match on \text{TypeAppOverloaded}, and as a result
can support method overloading.

Listing 6 shows that we can conveniently enumerate
through a list of typing rules by pattern matching on \text{goal}
in line 2. While we match on Goals of typing conditionals,
member selection, variables and application, respectively,
we can integrate analysis for further typing rules by pattern
matching on their high-level representation. An example of
such extension is given in method \text{applicationContext}
which essentially implements analysis rule \text{APP–FOC}. Apart
from the already discussed polymorphic and monomor-
phic function applications, we now also pattern match on

\(^3\) Scala allows for providing custom extractors \[7\] for pattern matching and
the framework provides them for every \text{Goal} subtype
TypeAppOverloaded, and as a result can support method overloading.

```
0 def dispatch(goal: TypeGoal, focus: TypeFocus): List[Goal] =
1   goal match
2 case TypeIfElse(_, thenP, elseP, leastUpperBound) => ...
3 case TypeMemberSelection(qualifier, memberSel) => ...
4 case TypeVariable() => ...
5 case TypeApp() => ...
6 }
7 def applicationContext(app: TypeApp, focus: TypeFocus) =
8   app match {
9 case TypeAppCorrect(fun, app@TypeAppPolymorphic(_, infer, _)) =>
10   val focu1 = focus compose FocFunRes compose FocPolyRes
11   val typePart = focu1(typeOFun(fun))
12   if (typePart == Error) app :: Nil
13   else if (hasTypeVariables(typePart)) inferInstance(infer, focu1)
14   else app :: dispatch(fun, focu1)
15 case TypeAppCorrect(fun, app: TypeAppNonomorphic) => ...
16 case TypeAppCorrect(fun, app: TypeAppOverloaded) => ...
17 }
```

Listing 6. Fragment of a generic routine that finds the source of the type, based on TypeFocus.

While the high-level representation is clearly necessary for navigating derivation trees, it is not sufficient. TypeFocus operates on low-level type information in order to guide type exploration. While in line 11, we extract low-level information using function typeOFun, we therefore make our exploration implementation dependent.

4.3 Reconstructing Local Type Inference Decisions Using the High-Level Representation

An advantage of local type inference is that all inference decisions are local, in the sense of the proximity of Goals, and are based on type constraints that are also locally collected. In this section, we exploit this fact by describing how, in using only TypeFocus and a type inference Goal, we can recreate specific source code locations that affected type inference.

```
trait InferMethodInstance extends InferInstance {
  def subargsTypesWithFormals: List[SubtypeCheck]
  def inferInstantiations: List[SolveTVar]
}
```

Listing 7. Instantiation of a polymorphic method

In Section 4.1, we have shown that by using TypeFocus we can precisely identify typing rules where inference will instantiate type variables from the extracted portion of a type. We therefore assume that we can identify InferMethodInstance Goals, presented in Listing 7, which are high-level representations of type inference decision process.

The interface of InferMethodInstance clearly states that before performing any instantiation of type variables in Goals of type SolveTVar, two typechecking decisions will have to be made, both of which involve a Goal of type SubtypeCheck. More importantly, SubtypeCheck Goals represent a high-level representation for subtype relation checks between the two types. Hence, the first two members, give a reference to derivation trees that perform subtyping checks between the function result type and the expected type, and types of arguments and formal parameters, respectively. In a type inference sense, derivations represented by the two members represent primary sources of type constraints.

The main concern is to identify only those type constraint Goals that affected the instantiation of particular type variables. Here, again, we can make use of TypeFocus to select only those SolveTVar Goals from the member of inferInstantiations that have been in the extracted type part of the polymorphic function type. As each type variable instantiation Goal, SolveTVar, also has a low-level reference of used type constraints, we can identify correct SubtypeCheck Goals by simply traversing derivation trees they represent in search for those constraints.

Knowing which subtyping relation checks led to type constraints is still far from the actual identification of source code locations used in type inference. This is because a corresponding low-level information of SubtypeCheck Goals only has information about types rather than terms. From the discussion in previous sections, we may recall that InferMethodInstance is itself a premise to a polymorphic function application goal, TypeAppPolymorphic. The latter has a member typecheckArgs of type List[TypeGoal], which represents the decision process for typechecking the arguments, and more importantly has low-level information about the expressions that were being typechecked at that point. Subtyping checks performed in the member subArgsTypesWithFormals of the type inference Goal, come directly from typechecking those arguments. As a result, we are able to identify expressions that affected type inference of selected type variables, without encoding that
information directly in either the low-level or high-level representation.

4.4 Providing Type-Correct Code Modifications

The high-level representation provides a convenient environment for compiler experts and library authors to use their experience to suggest direct code modifications.

Since many of the errors related to limited local type inference, the framework exposes two of the typechecker’s internal methods responsible for calculating least upper bound (lub) and greatest lower bound (glb) from a list of types (both take simply arguments of type List[Type]). This allows us to investigate a more suitable type instantiation for local type parameters that is not limited by the boundaries of parameters’ list i.e., in the motivating example of foldRight application we provide original type constraint List[Nothing] coming from Nil (discovered using technique from Section 4.3), as well as an inferred type List[Int] to lub, where typechecker returns better instantiation, namely List[Int].

As a programmer, we still want to make sure that such change is precise and type correct, so the framework has to provide an API that can typecheck the modifications. The listing above presents a subset of API provided in the framework — typecheckAscription verifies an explicit type ascription of lub type for an expression at location exprPos, typecheckTypeArguments verifies an application of type arguments targs for a polymorphic function application at position appPos, and typecheckMethodSignature which modifies a type of a polymorphic method methodDef to a new type newTypeSig and typechecks its usage in function application at position appPos. This way programmers can experiment with different heuristics for code modifications both in generic and plugin approach and still remain conservative in terms of type feedback.

5 Domain-Specific Type Errors

Generic exploration algorithms and code modifications might still reveal internal details of the data structures in the debugging feedback, which is correct but not always desirable. This section describes a complementary feature of the framework infrastructure—type debugging plugins—that allows for further type error customization for eDSLs and general purpose libraries.

trait DebuggerPlugin {
  def definedFor: PartialFunction[Goal, Option[Fix]]
}

Any domain specific plugin has to implement a DebuggerPlugin trait presented in listing above. A single method defines a handler for the erroneous case by means of a partial function from Goal to an optional fix, Fix, that carries information necessary to generate an additional feedback (we leave out the definition of Fix for irrelevance).

Whenever an error Goal is encountered, the debugging framework will first check if there exists a plugin that can potentially handle it. If the custom plugin produces some user feedback, then it will be reported. On failure, the mechanism will find other potential plugins, or fallback to generic-purpose type error debugging algorithms. Hence we allow for initial lightweight matching that is later verified through, potentially expensive, more broad type derivation tree analysis. Lightweight error matching ranges from simple regular expression matching on the error message to basic overview of the derivation subtree in the vicinity of the error.

```scala
Listing 8. ScalaTest error for 'should' operator
Consider a fragment of a simple test specification written using a popular testing framework, ScalaTest4, in Listing 8. Programmer has defined a correct expectation regarding an instance of class Foo in a human-readable style where 'status("OK")' represents an expectation on the member status to return a concrete string value. The error message, that spans over multiple lines, reveals intimate details of the testing DSL.

```

Listing 9. Fragment of ScalaTest plugin that handles overloaded 'should' operator
Listing 9 presents a fragment of a ScalaTest-specific plugin that identifies such operator overloading problem. Plugins do not enforce any rules on how particular errors should be identified and analyzed. The typical approach identifies the vicinity of the error location by pattern matching on type derivation trees and identifying used low-level symbols and types.

In line 1 we pattern-match on the parent of the error and expect high-level representation for method overloading, TypeAppOverloaded. Clearly reported error mentioned multiple alternatives in function application.

As the library author we also know that function is a member selection and should operator is provided through a DSL-specific implicit conversion. Our error handler ensures that this is the case in lines 4-6 by pattern-matching on

http://www.scalatest.org
corresponding typing rule using high-level representation for member selection, TypeMemberSelection. The second argument, TypeMemberInAdaptedQual, specifies that typechecker had to adapt the qualifier to a particular member. In line 6 we use low-level symbol information from the adapted member selection, to make sure that the selected member is the method provided in the DSL.

The DSL-author can provide specialized, human-readable information about each of the possible alternatives, which are expressed in premise inferAlts (line 2), in order to provide a more acceptable error message, such as:

'`should' misses an operator to handle property 'status("OK")'.
Providing one of the operators like 'should not exist', 'should exist', 'should contain', 'should have', 'should be'
... or an implicit conversion to 'ShouldMatcher' is enough.

Since specialized type error debugging relies on identical high-level representation as regular debugging, plugins’ programmers can reuse elements of the infrastructure to explore type derivations in a uniform way. This includes provided algorithms that identify the source of the expected type, locate constraints that affect type inference or even analysis of implicit search.

6 Real World Validation

In this section, we present several scenarios where we have applied our framework in practice, from the mainline Scala compiler, to a visual type debugging tool, to embedded DSLs. Additional validation is in the appendix; we apply our framework to the experimental validation found in [6], and show that our framework applies appropriate feedback in all cases (Appendix E), and we apply it to Scala Virtualized [19] to provide custom type errors (Appendix F).

6.1 Instrumenting the Mainline Scala Compiler

The framework supports two versions of the Scala compiler: 2.11.x trunk branch (Scala 2.11.0-M2) and 2.10.x stable branch (Scala 2.10.2). With 230 instrumentation classes and no logical modifications to the compiler, we covered most of the typechecker implementation that is typically exercised by libraries and DSL programmers. The framework supports different compilers with the same set of instrumentation (modulo code changes) as well as algorithms that analyze type derivation trees. This follows our intuition that while the compiler is actively developed, the logic of typechecker remains mostly intact, and therefore hardly ever alters our instrumentation.

6.2 Visualizing Typechecking

Our type error customization and debugging framework is also capable of visualizing type derivations trees in an incremental way, in the spirit of (and in many ways subsuming) [28]. We’ve found that such a tool is advantageous to those with only a cursory knowledge of type systems. All of the generic search algorithms described in Section 4 can be run and visualized, therefore enabling interactive exploration of the erroneous scenarios. A screenshot of our tool is shown in Figure 4 in Appendix D.

6.3 Type Debugging Existing eDSLs and Libraries

For the evaluation of the applicability of our framework in explaining non-trivial type errors we have looked at popular libraries and DSLs:

- Testing libraries\(^5\) (Specs2, ScalaTest, ScalaCheck)
- Shapeless\(^6\) – a type class and dependent type based generic programming library that uses advanced type system features (more elaborate discussion is available in Appendix K).
- Standard Scala library – an official collections framework that uses higher-order functions and higher-rank polymorphism.
- Lightweight Modular Staging (LMS) [29] – a framework for type-driven staging to allow for deep embedding. We tested LMS with OptiML, an eDSL for machine learning written in Scala and based on LMS.

Customization offered by library- and DSL-specific plugins mostly resolves to correct identification of the erroneous scenario and analysis of results provided by the generic algorithms in order to give even more specialized user feedback. In many cases we have relied purely on generic algorithms to analyze derivation tree and provide appropriate error messages.

Table 1 presents an overview of type-system features encountered in non-trivial problems in the given libraries, and for which our framework was able to identify the root of the problem and provide better feedback to users. Type errors were collected from real code examples posted on mailing lists, forums like StackOverflow, public repositories, as well as directly from authors of the libraries. As a result we collected 92 self-contained examples representing various non-trivial problems. Many of the examples exercise the typechecker’s ability to infer correct types. This involves debugging type errors related to type inference:

- Implementation limitations related to inference composed of only local type constraints, including for higher-kinded types.
- For type parameters having lower or upper type bounds (either concrete or parametrized) (we also compare related work in that are in Appendix E).
- That is directed by the parametrized result type of the method or where term is part of an assignment.
- For member selection where we deal with type parameters belonging to methods and class declarations.
- For path-dependent types and partially applied function applications.

\(^5\)etorreborre.github.io/specs2, scalatest.org and scalacheck.org
\(^6\)github.com/milessabin/shapeless
Type inference for higher-kindred types in Scala is very limited\(^7\) and often manifests itself in cryptic type errors involving bottom type Nothing. While this is a known limitation of Scala’s implementation that can be often solved by modifying type signatures of methods, related questions regularly appear on Stackoverflow or mailing lists. We have implemented heuristics in our framework that identify such situations based solely on high-level representation exposed by our framework (examples are provided in Appendix 1). Type Debugger’s approach was tested on 14 real world examples, where it provided code modifications that correctly fix those type errors.

Shapeless also uses Scala macros [2] that allow for convenient compile-time metaprogramming. Since macro expansion is part of regular typechecking and performs type-checking as well, our framework was still able to expose type derivations trees to library-specific plugin and eventually produce better error feedback.

While the framework has proved to be precise when explaining local type inference limitations, it failed to provide comprehensive user feedback in situations when source of term’s type has been distributed among different type inference locations \textit{i.e.}, if a type depends on an instantiation of a type parameter that itself depends on an instantiation of another type parameter at some other location, the framework will not automatically identify such dependency between the two locations. This is not caused by any technical limitation but rather our aim to keep type error feedback succinct, as we could very easily overwhelm users with the amount of information.

Implicits [26] are used in the design of flexible eDSLs and libraries with convenient APIs. LMS and OptiML use implicit conversions to perform lifting of values to their staged equivalents, and operations on staged values through type-classes. Many of the methods in Shapeless have an implicit parameter that allows for \textit{type-level computation}, whereas Standard Scala collections use them to avoid code duplication [22].

Our framework can deal with typical type errors related to implicit search \textit{i.e.}, diverging implicits, ambiguous implicits coming from the same as well as different scopes or simply no implicits found. Library-specific plugins can analyze Go4ls representing implicit search information and present it in a user-friendly way.

Our type debugging framework is the first one to analyze a popular design pattern used in all of the analyzed libraries where implicit definition takes implicit parameters themselves, essentially triggering a chain of implicit searches on application. While convenient to use, failures carry little or none useful information to debug the problem. Since our framework instruments implicit search implementation, we can provide better feedback (we provide a simple example in Appendix J).

Method overloading is commonly used in the tested libraries and LMS’s DLSs. We provide more user-friendly error messages by analyzing type signatures of the available alternatives. We can also explain type inference limitations in the presence of method overloading, such as not inferring types of parameters for closures. While our framework certainly exposes enough information to explain why alternatives were not selected according to language specification or implementation, we do not perform it. Unless controlled by users, which we currently do not support, the debugging process would produce large amount of information that is hard to parse visually.

To summarize, mature libraries and DLSs used for evaluation of our type error customization and debugging framework present non-trivial type errors that are encountered in every day programming. Our results have shown that the framework exposes enough of typechecking information in a format suitable enough to handle advanced type features. Finally, thanks to plugins infrastructure we were able to specialize error feedback for libraries and DLSs, that otherwise would reveal internal data structures.

### 7 Other Related Work

**Type inference algorithm modification.** Research on type error analysis has mostly focused on improving global type inference limitations in languages like Haskell or ML. Many implementations use a variation of the \textit{W} algorithm [4] which is known to produce biased type errors. Different variations have been proposed such as \textit{M}\(^SYM\) [17] or \textit{UAQ} and \textit{IEI} [14] that help with error localization. Apart from the work done in [34], which mixes local and global type inference elements, none of the solutions improve the state of local type inference typically encountered in object-oriented languages with subtyping. While modifications to the type inference algorithm may improve the location information they do not explain the decision process that led to type error.

**Program slicing.** Typechecking systems that have been translated into constraint-based systems led to the development of program slicing. Instead of generating an error message, all locations that affected the inference of a type error [10, 30] are provided. Unfortunately program locations need to come with an additional information in order to be useful. Our framework is capable of providing both. Furthermore, it is unclear how these systems would behave in the context of more advanced type features such as implicit or path-dependent types where an error can not always be explained with the use of a simple location.

**Interactive debugging.** Chameleon [31] offers a more interactive approach to error messages in a subset of Haskell. The type inference problem is translated into a constraint solving problem, where constraints keep source location information. Therefore, the tool is able to respond precisely to type inference questions from the user. Our framework provides a visualization of derivation trees, based on work done in [28], which provides similar but more more powerful capabilities, as it visualizes every decision.

**Automatic repair systems.** Existing automatic repair systems [3, 17] typically attack the problem of ill-typed expressions by generating a large amount of plausible solutions which could potentially provide a quick solution. In order to reduce the amount of inevitable false-positives results, one has to devise a ranking mechanism that allows to quickly judge how the repair relates to others [15]. [9] presents a technique that explores a number of structure modifications that would fix an ill-typed term. This is limited to problems based only on the simply typed lambda calculus. In [3, 15, 16] the typechecker is used as an oracle for verifying program modifications which reduces the amount of false-positives. None of the approaches support type-classes (encoded in Scala as implicit [26]), which involve expensive context of more advanced type features such as implicits or path-dependent types where an error can not always be explained with the use of a simple location.

**Explanation systems.** Explanation systems [5, 11, 14] typically collect information during typechecking to provide additional feedback to the users regarding problematic error. The process involves modifying the existing typechecker implementation to provide debugging capabilities or collecting constraints in a separate phase that essentially implements a simplified version of the typechecker. Generating a good message based on the type of constraint is challenging. Our framework, does not modify or replicate the typechecker and provides type derivations that can explored in conveniently.

8 Conclusions

We have presented a new, unified way of improving type error feedback for regular programmers as well as DSL authors. We provide a type debugging framework which instruments the Scala compiler in a lightweight way and exposes typechecker’s decision process in a high-level representation. Relying on high-level representation renders an opportunity to easier navigate and reason about typechecking from the programmer’s point of view. We have exercised our framework by generating useful corrective code suggestions in order to overcome well-known limitations of local type inference. A complementary plugin mechanism is useful for library and DSL designers to customize type errors to generate better user feedback.

In future work, we plan to explore the interactive aspect of our framework. Users should be able to query every typing of their code, a natural progression of work, since that information is available in the high-level representation already. Finally the high-level representation offers an interesting opportunity for testing for regressions, and to verify the stability of the compiler during regular development.

References
A Instrumentation Output for foldRight Application

To give a better idea of the structure of the collected low-level instrumented data, we present a simplified output for typechecking application `xs.foldRight(Nil())` (from our motivating example) in Listing 10. For presentation reasons instrumentation blocks increase the level of indentation, and ... represents omitted instrumentation output. Extracted information about a typechecked function, `xs.foldRight` having type \((z : B) \rightarrow (\text{op} : (\text{Int}, B) \rightarrow B) \rightarrow B\), is within lines 1 and 6, type arguments are potentially inferred in lines 7-9, `Nil()` term is typechecked in lines 10-12, inference of type instantiation for type parameter \(B\) is output in lines 13-15, and finally typechecking of an application (with the just inferred type argument `List[Nothing]`) gives output in lines 16-18. While this particular fragment of typechecking execution is type correct, the true complexity of understanding such low-level output is particularly problematic when an erroneous case occurs and one of, potentially many, fallback mechanisms in the compiler is executed. Since the primary aim of our tool is to help with with understanding the decisions of typechecker in the presence of cryptic type errors, the latter is a common situation.

```
0  > TypeApplication      // xs.foldRight(Nil())
1   > TypeMemberSelection // xs.foldRight
2       ...             
3   > TypeSelection       
4   > TypeApp
5   > InferTypeArguments  // B from type "(z: B) => ..." // (op: (Int, B) => B) => B'
6   > ...               
7   > Inferred
8   > TypeGoal           // Nil()
9   ...               
10  > TypeGoalFinished    // For type "(z: B) => ..." // (op: (Int, B) => B) => B'
11   > ...               
12  > InferInstance       // For (z: List[Nothing]) => ...
13   > ...               
14  > TypeApplication     // List[Nothing] => List[Nothing]
15   > ...               
16  > TypeAppDone         // List[Nothing] => List[Nothing]
17   > ...               
18  > TypeAppDone
```

Listing 10. Simplified instrumentation output for `xs.foldRight(Nil())`

B Encoding Lists in the Formalism from [25]

Using the minimal language we can provide intuitive definitions for lists using records with a single, `match`, method that uses visitor pattern for inspection (in the same manner as in [25]).

```
type List[a] = {
  match[b](v: List[a,b]) : b
}
type ListVis[a,b] = {
  caseNil(): a,
  caseCons(x: a, xs: List[a])
}
```

Type `List` takes a single type parameter, \(a\), that represents the elements of the list, while the local type parameter of match, \(b\), represents the type of the result of matching on such a list with a list visitor. `ListVis` is a record type requiring two functions that are called either when an unde-

C TypeFocus Built From Algorithmic Subtyping

TypeFocus that is created solely from typing rules is not able to provide enough precision for complex types, such as type constructors having multiple type parameters. For instance, let us assume that the type constructor `Map` has two type parameters \(A\) and \(B\), and we encounter a type error caused by the type mismatch between `Map[X,Y]` and `Map[T,J]`, where the type argument \(Y\) is not a subtype of \(J\). Using the provided analysis, we would only identify types that affect type `Map[A, B]`, rather than just the type parameter \(B\).

The framework provides an orthogonal TypeFocus that is built from the high-level representation of algorithmic subtyping. Such subtyping `TypeFocus` allows us to achieve a greater level of precision when identifying parts of types that lead to errors. At the same time, it requires minor refinements in the formalization of the analysis presented in Figure 3. To illustrate the problem, we consider TypeFocus that selects a first type argument in type application involving `List` type constructor (i.e., `foc-targ-0-list(List[a]) = a` for some type variable \(a\)) in

```
foc-all o foc-targ-0-list o foc-fun-res o foc-fun-param-0 o foc-fun-res o foc-poly-res
```

Applying full `TypeFocus` composition to the function type will be invalid, since \(b_4\) is an uninstantiated type variable, rather than an expected type application. Therefore `TypeFocus` objects provide `partialFocus` method of type \(T \Rightarrow (T, Focus)\). It returns a tuple consisting of a type that has been extracted before an `Error`, and the remaining, `failed`, part of the `TypeFocus` composition. Analysis rules can be easily updated to reflect that change, and we use the remaining part of `TypeFocus` composition to analyze type constraints in type inference more precisely.

D Visual Type Debugger: Screen Shot

(See Figure 4)

E Debugging type errors related to Generics

The work of El Boustani and Hage [6] focuses on providing more informative error messages for problems related to Java Generics. In that sense they are the closest to our framework that authors are aware of, in terms of type debugging modern language that supports polymorphism and subtyping. We have translated all 22 examples that were used in the evaluation of the Java framework into Scala, 4 of which have been excluded as they were type correct in Scala.
The examples handle method invocations that have, potentially many, local type parameters or type constructors with various type bounds, and arguments are either variables or functions which result type is a concrete type. Type Debugger for Scala provides comparable error messages for such type errors. The quality of feedback has been assessed in terms of localizing the source of error and highlighting type constraints that led to a type mismatch. Only examples involving wildcard types, that were translated to existential types in Scala, proved to be problematic, as they do not always follow the same instantiation semantics and locations as regular type parameters. More importantly our framework does not limit error reporting to just invocations of polymorphic methods.

Code modifications suggested by the Java framework only modify type arguments or types for already type annotated variables without actually verifying those code changes. We believe that if type annotation is provided by the user, it expresses a firm conviction that a term is of a particular type and there is a much smaller chance that a simple type argument change will be type correct for the definition of the variable. To illustrate the difference we present a small Java program taken from [6], Scala translation of the method and error produced at function application:

```
<T> List<T> foo(Map<T, ? super T> a){...}
Map<Number, Integer> m = ...;
List<Integer> ret = foo(m);
```

Java framework generates type error:

Method <T>foo(Map<T, ? super T>) is not applicable for the argument of type (Map<Number, Integer>), because:

[*] The type Integer in Map<Number, Integer> on 5:9(5:21) is not a supertype of the inferred type for T: Number. However, replacing Number on 5:13 with Integer may solve the type conflict.

while our Type Debugger would produce:

Type mismatch has been partially caused by the type of the value ret that has an explicit type List[Integer]. Locations on the RHS of the assignment that directly led to type mismatch:

```
val m: Map[Number, Integer] = ???
val ret: List[Integer] = foo(m)
```

Type parameter T in method foo has been instantiated to Number using the least upper bound of:

```
val m: Map[Number, Integer] = ???
val ret: List[Integer] = foo(m)
```

By following Java’s suggestion and modifying the type annotation of m we will most likely lead to a type mismatch between the new type annotation of value m and its RHS. Hence, a more conservative approach in our framework.

F Virtualized Scala

Virtualized Scala [19] is an experimental branch of the Scala compiler that enables DSL authors to override standard language constructs (conditionals, variables, loops) by essentially transforming them into regular method calls. Meth-
ods prefixed with infix_ are treated as infix operators and require even more modifications in typechecking function application in order to support them. Whenever function in function application is a member selection, such typechecker will always try to find an infix method in the scope for the member i.e., \( x \cdot y(z) \) will essentially trigger a search and typechecking of application infix_\( y(x, z) \).

Our framework supports experimental compiler in order to improve the general DSL experience. Although Virtualized Scala modifies the existing Scala typechecker we had to add only 8 new instrumentation classes and 12 high-level Goals to regain the ability to correctly expose the same type-checking decision process as in a regular compiler. If type-checking of infix function fails, which would be most of the time, we fallback to regular typechecking of function application, for which a high-level representation already exists. For completeness we provide an example of high-level representation for virtualized function application in Appendix G.

Naturally, our exploration algorithms needed to be updated to reflect such change. Since analysis is expressed by pattern matching on high-level representation, we only had to add new pattern matching cases, rather than redesigning the whole framework. More importantly any changes are statically verified.

To evaluate our approach we have provided a custom debugger plugin specific to Scala Virtualized and libraries based on it. We were able to precisely identify common problems, involving type inference or operator overloading, and provide better error feedback.

G High-Level Representation for Virtualized Function Application

```scala
trait TypeVirtApply extends TypeGoal {
  def typeInfixMethod: TypeGoal
}
trait TypeVirtApplyFallback extends TypeVirtApply {
  def typeApply: TypeApply
}
trait TypeVirtApplyCorrect extends TypeVirtApply {
  def typeQual: TypeGoal
def typeArg: TypeGoal
}
```

Listing 11. High-level representation for virtualized function application

Goal TypeVirtApply expresses that in the typechecker of Virtualized Scala, we first attempt to type some method infix_\( y \) when encountering function application AST. Through member typeApply in TypeVirtApplyFallback we express the constraint that on failure, we fallback to the usual typing of function application, while on success our type debugger would return TypeVirtApplyCorrect, where premises mean that we need to typecheck (old) qualifier, \( x \), and a single argument, \( z \).

H Providing Customized Type Errors for Virtualized Scala

```scala
val x: Rep[Array[Int]] = ...
```

Listing 12. An error in eDSL that uses LMS and Virtualized Scala

Using type debugging plugins described in Section 5, we can provide custom feedback to libraries that make use of Virtualized Scala. Lightweight Modular Staging (LMS) [29], being one example, often exhibits confusing type errors that reveal internals of the staging process. Currently there exists no technique to mitigate or customize such errors. Listing 12 presents a trivial case where a user attempted to apply some function to all elements of a staged Array value, \( x \). Type constructor Rep tells LMS that the value is supposed to be staged. Both, regular and staged, types of Array provide method foreach through an implicit coercion, therefore DSL user could assume that the presented code is valid (without type Rep such code is a regular type-correct Scala code). Instead, an error is reported. In an ideal situation programmers would identify this problem by pattern matching on some function application where the function is foreach and it’s a member of qualifier of type Rep[Array[...]]. Such scenario is therefore abstract enough to be expressed in terms of typing rules. Fortunately, our framework was perfectly able to do. LMS-specific plugin would pattern match on the type derivation tree using case

```scala
case TypeAppCorrect(TypecheckGoal(_,
    TypeMemberSelection(qual, mem)), _),
  typeAppMain) => ...
```

to identify erroneous function application. Without going into details, TypeAppCorrect represents an already encountered typing rule for function application, with TypecheckGoal representing typechecking of function, premise TypeMemberSelection refers to regular high-level representation of a member selection typing rule where qual premise refers to typechecking of a qualifier, and mem to typing of a member of that qualifier. More importantly qual has low-level data about the type of the qualifier, and mem about the member that is accessed. If LMS plugin successfully matches them with Rep[Array[T]] (for some \( T \)) and foreach on them, respectively, then it has identified that particular problem. We were able to exploit that knowledge to produce more useful message:

Unlike the regular Scala compiler, LMS does not insert implicitly ‘(‘) value in the body of the ‘foreach’ closure. ‘foreach’ on ‘Array’ differs from the one in Standard library.

I Limitations of Type Inference for Higher-Kinded Types in Scala

Our test-suite contains 14 real examples that specifically target compiler’s inability to infer types for higher-kindred types. We illustrate the problem with a simple problem,
where one of the type parameters, $T$, is used within the bounds of another, $U$, and is in higher-order position:

```scala
abstract class A;
abstract class B[T <: A]
class XA extends A; class XB extends B[XA]
def foo[U <: B[T], T <: A]( resolver: U ): Unit = ()
foo(new XB)
```

Example below presents an un-informative type error from the regular Scala compiler, and the Type Debugger feedback.

Inferred type arguments [XB,Nothing] do not conform to method foo's type parameter bounds [U <: B[T], T <: A]

```scala
foo(new XB)
```

Type Debugger feedback:
The current type signature of method foo
(of type [U <: B[T], T <: A](resolver: U): Unit)
limits the current’s implementation ability to infer
an appropriate type argument for the type parameter $T$.
Inferred type argument for $U$, $XB$, is not within the upper bound $B[Nothing]$. In order to track appropriately the constraints
for the type parameter $T$ you can modify the type signature to:
```scala
def foo[U[_ <: A] <: B[_], T <: A](resolver: U[T]): Unit
```

J  Errors related to implicit search

The example above defines three implicits in the scope - b1, b2 and foo, and two classes. More importantly lack of an argument in function application `test()` triggers an implicit search. Type Debugger provides not only a reason for a type error, but also concrete arguments that led to it. Such feedback is crucial since Scala libraries typically define large number of potentially ambiguous implicits in them.

K  Errors in Shapeless

Exotic combinations of advanced type system features, such implicitss and higher-kindred types, that are present in Shapeless lead to type errors that are hard to parse for users. Nevertheless, we found Type Debugger to remain reasonably precise even without specialized debugger plugin.

```scala
def foo(x: Int :: String :: Nil) {}
foo(1 :: 2 :: Nil)
```

A function application example above uses data structure for heterogeneous lists, HList. Type mismatch reported by the compiler blames full type for an error and reveals internal details of the implementation, whereas Type Debugger is able to identify exact conflicting locations using subtyping `TypeFocus`. 