An Embedded Type Debugger

Kanae Tsushima(✉) and Kenichi Asai

Ochanomizu University, Tokyo, Japan
tsushima.kanae@is.ocha.ac.jp, asai@is.ocha.ac.jp

Abstract. This paper presents how to build a type debugger *without* implementing any dedicated type inferencer. Previous type debuggers required their own type inferencers apart from the compiler's type inferencer. The advantage of our approach is threefold. First, by *not* implementing a type inferencer, it is guaranteed that the debugger's type inference never disagrees with the compiler's type inference. Secondly, we can avoid the pointless reproduction of a type inferencer that should work precisely as the compiler's type inferencer. Thirdly, our approach is robust to updates of the underlying language. The key observation of our approach is that the interactive type debugging, as proposed by Chitil, does not require a type inference tree but only a tree with a certain simple property. We identify the property and present how to construct a tree that satisfies this property using the compiler's type inferencer. The property guides us how to build a type debugger for various language constructs. In this paper, we describe our idea and first apply it to the simply-typed lambda calculus. After that, we extend it with let-polymorphism and objects to see how our technique scales.

1 Introduction

To write a well-typed program is not always easy. Although a compiler gives us an error message when a type error occurs, it is not straightforward to understand why the type error arose. Furthermore, the source of a type error can be far from the place reported by the compiler as a type error. In this paper, we define the source of a type error to be a part of an ill-typed program which programmers want to fix. Our purpose is to construct a way to find it in a strongly typed functional language. In this paper, we use OCaml's syntax for examples.

1.1 Locating the Source of a Type Error

Two Conflicting Expressions. A type error occurs when types of two expressions conflict with each other. Let us consider the following example:

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
(f 1 [2;3;4]) @ [5;6;7]
```

R. Hinze (Ed.): IFL 2012, LNCS 8241, pp. 190–206, 2013.
DOI: 10.1007/978-3-642-41582-1_12, © Springer-Verlag Berlin Heidelberg 2013

In this program, the two boxed expressions have a type conflict causing a type error. The first argument `g` of the function `f` is used as a function in `(g fst)`, but an integer `1` is passed as `g` in `(f 1 [2;3;4])`. Because a function type `'a -> 'b` cannot be unified with `int`, a type error occurs. To locate these two conflicting expression is useful when one of these conflicting expressions is the source of a type error. Unfortunately, it is not always the case.

The Source of a Type Error. The source of a type error cannot be determined solely from the conflict of types. For example, suppose that a call to `f` in the previous example is wrapped by a call to `h`.

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
let h n lst = f n lst in
(h 1 [2;3;4]) @ [5;6;7]
```

In this program, although `(g fst)` and `(h 1 [2;3;4])` are the conflicting expressions, the source of the type error may be in the definition of `h`: if we replace the boxed expression with `(f (fun x -> x + n) lst)`, the program is well-typed. Because which of these expressions is the source of the type error depends on the programmer's intention, we cannot locate the source of the type error automatically.

A Standard Type Inference Tree. To locate the source of a type error, we basically detect the difference between an inferred type and a programmer's intended type. Let us consider a small example:

```
(fun x -> x + x) true
```

This program is ill-typed, because `true` is passed to `x`, but `x` is consumed by an integer addition `+`. Let us assume that the programmer wrote this program, because he mistakenly thought that `+` was the logical *or* operator.[1] Since the logical *or* operator in OCaml is `||`, the programmer's intended program is `(fun x -> x || x) true`.

We show a standard type inference tree for this example constructed by the compiler in Fig. 1 and programmer's intended type tree in Fig. 2. By detecting the difference between these two type inference trees, we can locate an expression that includes the source of a type error. For example, since types of expressions in the boxed part differ in Figs. 1 and 2, the source of the type error resides in the expression `(fun x -> x + x)`. However, we cannot further identify which subexpression of this expression is the root cause of the type error, as long as we use a compiler's type inference tree.

The standard type inference tree is not suited for type debugging, because a type of an expression can depend on the types of other expressions. In the above

[1] This is an example of the source of this type error. If the programmer has a different intention, other fixes are possible, such as replacing `true` with `1`.

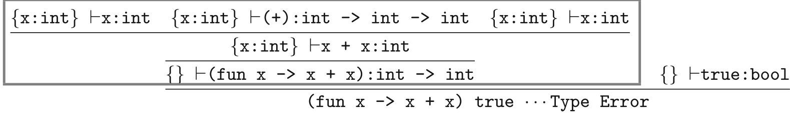

Fig. 1. A standard type inference tree

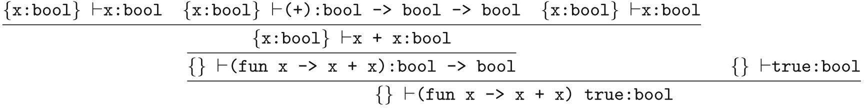

Fig. 2. Programmer's intended type tree

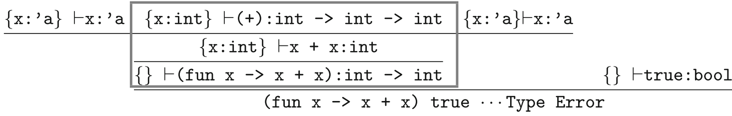

Fig. 3. The most general type tree

example, the type of `x` does not have to be `int` if it appears independently. It becomes `int`, because it is used as an argument of `+`. Such information is lost in the standard type inference tree, because the type of `x` becomes `int` throughout, once it is unified with the argument type of `+`.

The Most General Type Tree. To break the dependency between expressions, we introduce the most general type tree. We show the most general type tree for our example in Fig. 3. The most general type tree holds the most general type for each subexpression. For example, `x` has a typing {`x:'a`} ⊢`x:'a` for any type `'a`, because `x` alone does not require any constraints on its type. The type of `x` is constrained only when it is used in a context. For example, `x + x` has a typing {`x:int`} ⊢ `x + x:int`, because `+` requires that `x` has type `int`. Using this most general type tree, we can exactly locate the source of a type error by detecting difference between inferred types and intended types. By comparing Figs. 3 and 2, we find that the type conflict occurs in the boxed part of Fig. 3. We can then locate the source of the type error to be `+`. Note that the type of `x` (at the two leaves of the tree) does not contradict with programmer's intended type, because `bool` is an instance of `'a`.

Algorithmic Debugging. Of course, a tree with programmer's intended types exists only in programmer's mind. To extract programmer's intention, we use algorithmic debugging proposed by Shapiro [11]. Algorithmic debugging is used to identify the location of an error in a tree by traversing over the tree according to oracles. For oracles, questions for the programmer are often used. It is originally used for Prolog, but algorithmic debugging can be used for any tree structures and is applied to various areas, to locate run-time errors [9], semantic errors [12], etc. To debug Fig. 3 using algorithmic debugging, we start from the

root of the tree where a type error occurs. The type debugger first asks if the two child nodes are correctly typed according to programmer's intention. Since the programmer's intended type for `(fun x -> x + x)` is not `int -> int` but `bool -> bool`, the programmer answers no to the first question. From this answer, the type debugger determines that the source of the type error resides within this expression. Next, the type debugger asks whether the intended type of `x + x` is `int`. Again, the answer is no, and the type debugger moves into the subexpression. By repeating this process, the type debugger locates the source of the type error as `+`.

1.2 Problems

Chitil [1] constructed the most general type tree by inferring types compositionally, and located the source of a type error interactively using algorithmic debugging. Using his type debugger, one can locate the source of a type error by simply answering questions.

Following Chitil's work, we implemented a type debugger for a subset of OCaml together with some improvements [15] and used it in a course in our university. However, due to the need to implement a tailor-made type inferencer, we encountered at least three problems.

Implementation of a Type Inferencer. First, to implement a type inferencer that returns exactly the same type as the compiler's type inferencer is tedious and error-prone. Even for a small language, we had to fully understand the behavior of the compiler's type inferencer. For example, a compiler has an initial environment for typing. If a tailor-made type inferencer lacks a part of the initial environment, it cannot infer the same type as the compiler's type inferencer. Furthermore, the discrepancy between the two type inferencers becomes apparent only when we find unexpected debugging behavior. It makes it hard to detect errors in the tailor-made type inferencer.

Support for Advanced Features. Secondly, to implement a type inferencer for advanced features, such as objects and modules, is difficult and takes time. In our previous type debugger [15], we could implement the main subset of OCaml, including functions, lists, and pattern matching, but not the advanced features, such as user-defined data structures, objects, and modules. This is unfortunate: a type debugger would be particularly useful in the presence of such advanced features.

Compiler's Updates. Thirdly, to reimplement the type inferencer every time the compiler is updated is costly. In the last three years, the OCaml compiler had two major updates and two minor updates. It is not realistic to follow all these updates and reimplement the type inferencer.

To solve these problems, we propose *not* to implement a tailor-made type inferencer but to use the compiler's own type inferencer as is to construct the most general type tree.

1.3 Our Approach

Rather than implementing our own type inferencer, we use a compiler's type inferencer to construct the most general type tree. Construction consists of two stages. First, the erroneous program to be debugged is decomposed into subprograms. This decomposition determines the overall shape of the tree. Then, the type of each subprogram is inferred by passing the subprogram to the compiler's type inferencer. For example, if a program M is decomposed into subprograms, $M_1, \ldots, M_n$, we first construct the left tree below.

$$\frac{M_1 \nwarrow \quad \ldots \quad \nearrow M_n}{M} \quad \Rightarrow \quad \frac{M_1 : \tau_1 \quad \ldots \quad M_n : \tau_n}{M : \tau}$$

We then infer their types (possibly an error) by passing each of M_i (and M) to the compiler's type inferencer to obtain its type τ_i (and τ). Note that unlike the standard type inference, types of subexpressions are *not* determined by applying typing rules to the parent expression. Rather, they are determined by executing the compiler's type inferencer for each subexpression independently.

The above explanation is somewhat simplistic, because we did not consider bindings. To cope with bindings properly, we actually maintain a context C of an expression M, treating $C[M]$ as a complete closed program (where $C[M]$ is the expression C whose hole is filled with M, possibly capturing free variables of M). We call M in $C[M]$ the *focused* expression.

Overview. In the rest of this paper, we first show a type debugger for the simply-typed lambda calculus in Sect. 2 and a necessary property for decomposition in Sect. 3. We then extend it with let polymorphism (Sect. 4), and objects (Sect. 5) to see how our technique scales. Finally, we describe our implementation of a type debugger for OCaml that uses OCaml's own type inferencer (Sect. 6). We compare our work with related work in Sect. 7, and the paper concludes in Sect. 8.

2 The Simply-Typed Lambda Calculus

In this section, we introduce a type debugger for the simply-typed lambda calculus. Although simple, it is enough to explain the basic behavior of our type debugger.

The Language. We show the syntax of lambda calculus $\lambda_{\rightarrow}$ in Fig. 4. It includes constants, variables, abstractions, and applications. We assume that basic primitive operations (such as + that we will use in examples) are predefined as constants. Types include type variables, type constants, and function types.

Tree Structure Determined by Decomposition. Let us consider a type inference tree for $\lambda x.x + 1$. Since the only subprogram of $\lambda x.x + 1$ is $x + 1$ and it is further decomposed into three subprograms, x, $(+)$, and 1, the overall structure of the tree should look like:

$$\frac{\dfrac{\Gamma_0 \vdash x \quad \Gamma_0 \vdash (+) \quad \Gamma_0 \vdash 1}{\Gamma_0 \vdash x + 1}}{\Gamma_0 \vdash \lambda x.x + 1}$$

$$
\begin{array}{rcll}
(M : \mathit{term}) & ::= & c & \text{(constant)} \\
& | & x & \text{(variable)} \\
& | & \lambda x.M & \text{(abstraction)} \\
& | & M_1\ M_2 & \text{(application)} \\
(\tau : \mathit{typ}) & ::= & b & \text{(type variable)} \\
& | & \mathrm{int}, \mathrm{bool}, \ldots & \text{(type constants)} \\
& | & \tau_1 \rightarrow \tau_2 & \text{(function type)} \\
(C : \mathit{context}) & ::= & \square & \text{(empty context)} \\
& | & \lambda x.C & \text{(lambda context)}
\end{array}
$$

Fig. 4. The syntax of simply-typed lambda calculus $\lambda_{\rightarrow}$

where Γ_0 is the initial environment used by the type inferencer of the underlying compiler and contains all the bindings for the supported constants. However, the above subprograms are not directly typable using the compiler's type inferencer, because they include free variables (such as x).

Decomposition with Contexts. To make a subprogram typable, we enclose it with a context that supplies necessary bindings for free variables. In this language, a context is defined as either an empty context $\square$ or a lambda binding $\lambda x.C$ (Fig. 4). The most general type tree of $\lambda x.x + 1$ becomes as follows:

$$
\dfrac{\dfrac{\Gamma_0 \vdash \lambda x.[x] : \texttt{'a -> ['a]} \quad \Gamma_0 \vdash \lambda x.[(+)] : \texttt{'a -> [int -> int -> int]} \quad \Gamma_0 \vdash \lambda x.[1] : \texttt{'a -> [int]}}{\Gamma_0 \vdash \lambda x.[x+1] : \texttt{int -> [int]}}}{\Gamma_0 \vdash [\lambda x.x+1] : \texttt{[int -> int]}}
$$

Looking at the focused expressions filled in the context, we see that it has the same structure as the previous tree. Thanks to the contexts, all the subprograms are now typable under Γ_0. The types enclosed by `[...]` correspond to the types of focused expressions.

Although the above tree is similar to the standard type inference tree for $\lambda_{\rightarrow}$:

$$
\dfrac{\dfrac{\Gamma_0, x : \texttt{int} \vdash x : \texttt{int} \quad \Gamma_0, x : \texttt{int} \vdash (+) : \texttt{int -> int -> int} \quad \Gamma_0, x : \texttt{int} \vdash 1 : \texttt{int}}{\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}}}{\Gamma_0 \vdash \lambda x.x + 1 : \texttt{int -> int}}
$$

they have two important differences. First, the type of x is *not* constrained to `int` at the leaf nodes. Since we treat all the subderivations independently, each judgement depends only on its subexpressions. It enables us to locate where the type of x is first forced to `int`. Secondly, the type environment contains only the predefined constants. It enables us to use the compiler's type inferencer to infer the type of each expression. We simply pass it to the compiler's type inferencer and obtain its type. This is in contrast to the standard type inference tree where the environment contains free variables.

Other Approach. A compiler's type inferencer is usually designed to accept an open expression and an environment for its free variables. Although we could use this extra flexibility for the type debugger, it does not lead to a simpler type debugger. In this paper, we chose to use contexts, to avoid going into the underlying compiler implementation together with the representation of environments.

$$
\begin{aligned}
&Dec : context * term \to (context * term)\ list \\
&Dec[\![(C, c)]\!] = [] \\
&Dec[\![(C, x)]\!] = [] \\
&Dec[\![(C, \lambda x.M)]\!] = [(C[\lambda x.\Box], M)] \\
&Dec[\![(C, M_1\ M_2)]\!] = [(C, M_1); (C, M_2)]
\end{aligned}
$$

Fig. 5. The decomposition function *Dec* for $\lambda_{\to}$

$$
\begin{aligned}
&env = (var * typ)\ list \\
&Collect : context \to typ \to env \to (env * typ) \\
&Collect_{\Box}[\![\tau]\!]\mu = (\mu, \tau) \\
&Collect_{\lambda x.C}[\![\tau_1 \to \tau_2]\!]\mu = Collect_C[\![\tau_2]\!]\mu[x \mapsto \tau_1]
\end{aligned}
$$

Fig. 6. The function *Collect* to obtain types of free variables for $\lambda_{\to}$

If we want to implement type debuggers for various languages, it would require substantial investigation of the underlying compiler. The method proposed here has an advantage that we can treat the compiler's type inferencer completely as a black box that accepts an expression and returns its type.

Construction of the Most General Type Tree. The most general type tree is built as follows. A program to be debugged $C[M]$ is first decomposed into subprograms using the decomposition function *Dec* defined in Fig. 5. It basically decomposes M and returns a list of its subprograms, but it maintains its contexts properly so that the resulting subprograms (pairs of a context and a decomposed term) are always closed. When the decomposition of $C[M]$ is $[C_1[M_1]; \ldots; C_n[M_n]]$, all the subprograms become the children of $C[M]$ in the tree.

The type of each subprogram $C[M_i]$ is determined using the compiler's type inferencer by passing $C[M_i]$ to it. When the context C is empty $\Box$, the returned type is the type of the expression. When the context is not empty, we split the obtained type into two: types for free variables and the type for the focused expression. If we obtain the type of $\lambda x.[x+1]$ as `int -> int`, for example, we associate the type of x to be `int` (the argument part of `int -> int`) and the type of $x+1$ to be `int` (the body part of `int -> int`). This is done by the function *Collect* in Fig. 6.

Using *Dec* and *Collect*, we construct a judgement for $C[M]$ in the tree as shown in Fig. 7. First, we construct a closed term M' by plugging M into C. It is then passed to the compiler's type inferencer written as `typing` here. When we obtain a type τ of M', we split it into an environment γ holding types of variables in the context and a type τ' for M. Using them, we can construct a judgement for (possibly open) M (in the context C) as $\Gamma_0, \gamma \vdash M : \tau'$. For $\lambda x.[x+1]$, for example, we have $\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}$.[2]

[2] Before, we wrote $\Gamma_0 \vdash \lambda x.[x+1] : \texttt{int -> [int]}$ to emphasize that we are using the compiler's type inferencer to infer the type of M in C. Since we are interested in the type of M itself together with the types of its free variables, we also write it using the standard notation $\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}$.

$$
\begin{aligned}
Judge[\![(C, M)]\!] = &\text{ let } M' = C[M] \text{ in}\\
&\text{ let } \tau = \texttt{typing } M' \text{ in}\\
&\text{ let } (\gamma, \tau') = Collect_C[\![\tau]\!][] \text{ in}\\
&\ (\gamma, \tau')
\end{aligned}
$$

Fig. 7. The function *Judge* to obtain typing for $\lambda_{\rightarrow}$

3 The Decomposition Property

In our type debugger, the most general type tree is constructed by first decomposing an expression into subexpressions and then inferring their types using the compiler's type inferencer. The shape of the tree is determined by how we decompose an expression. However, it does not mean that we can use any arbitrary decomposition. We require that the decomposition satisfies the following necessary property:

Definition 1 (The Decomposition Property). *The decomposition function Dec should satisfy the following property for any context C and term M:*

$$T(C[M]) \Rightarrow \forall (C', M') \in Dec[\![(C, M)]\!], T(C'[M'])$$

where T is a predicate stating that a given expression is well typed (under the compiler's type inferencer).

The decomposition property states that if a program $C[M]$ is well typed, all of its decomposed subprograms are also well typed. Although this property looks trivial, it does preclude $x + 1$ as a decomposition of $\lambda x.x + 1$, because the latter is well typed, but the former is not typable with unbound x. In the next section, we will see how this property guides us to define decomposition that is suitable for type debugging.

This property is essential for our type debugger. Since the source of a type error is detected by tracking conflicts between inferred types and intended types, we can no longer continue type debugging into subexpressions if their inferred types are not available from the compiler's type inferencer. Therefore, we design decomposition carefully so that it satisfies the property and thus keeps the typability of expressions. In the following sections, we sketch why the presented decomposition satisfies this decomposition property. For the simply-typed lambda calculus, we reason as follows.

Decomposition for $\lambda_{\rightarrow}$ Satisfies the Decomposition Property. We need to show that for each case of the definition of *Dec* in Fig. 5, all the subexpressions in the right hand side are well typed if the left hand side is well typed. For constants and variables, it is satisfied vacuously. For abstraction, because the expression in the left hand side $C[\lambda x.M]$ is identical to the expression in the right hand side $C[\lambda x.[M]]$, the decomposition property is satisfied. For application, we notice

that if $C[M_1 M_2]$ is well typed, $M_1 M_2$ is also well typed in a type environment consistent with C (formally proven by induction on C). Hence, both M_1 and M_2 are well typed in the same environment. Since C has all the necessary bindings for M_1 and M_2 and C simply adds binding to them, both $C[M_1]$ and $C[M_2]$ are well typed as required.

4 Let Polymorphism

In this section, we extend our idea to let polymorphism.

The Language. We show the syntax of λ_{let} in Fig. 8. It extends the simply-typed lambda calculus with pairs, fixed points, and let expressions. Types are also extended accordingly. Unlike the standard let-polymorphic calculus, we do not introduce type schemes. Type schemes are required only for inferring types. Once the type inference is done (in the compiler), all the expressions in the most general type tree are given mono types (possibly containing type variables).

Naive Decomposition. To support a let expression in the type debugger, we first need to define its decomposition. Because a let expression contains two subexpressions, the let-bound expression and the main body, we are tempted to define its decomposition as these two subexpressions. However, straightforward decomposition leads to violation of the decomposition property (Sect. 3). Let us consider the following program:

$$1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)$$

Since id in the second subexpression $(id\ id)\ 2.0$ is free, we need to supply its context. If we naively follow the previous section, however, we end up with the following tree:

$$\dfrac{\vdash [1] : \texttt{int} \quad [+] : \texttt{int}\to\texttt{int}\to\texttt{int} \quad \boxed{\dfrac{\vdash [\lambda x.x] : \texttt{'a}\to\texttt{'a} \quad \vdash (\lambda id.[(id\ id)\ 2.0]) \cdots \texttt{Type Error}}{\vdash [let\ id = \lambda x.x\ in\ (id\ id)\ 2.0] : \texttt{float}}}}{\vdash [1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)] \cdots \mathit{Type\ Error}}$$

Although the bottom expression in the boxed part is well typed, one of its subexpressions is not well typed. Thus, this decomposition does not satisfy the decomposition property.

$$\begin{array}{rcll}
(M : term) & ::= & ...\ \mid\ (M_1, ..., M_n) & \text{(tuple)}\\
 & \mid & fix\ f\ x \to M & \text{(fixed point)}\\
 & \mid & let\ x = M_1\ in\ M_2 & \text{(let expression)}\\
(\tau : typ) & ::= & ...\ \mid\ \tau_1 * ... * \tau_n & \text{(product type)}\\
(C : context) & ::= & ...\ \mid\ fix\ f\ x \to C & \text{(fix context)}\\
 & \mid & let\ x = M\ in\ C & \text{(let context)}
\end{array}$$

Fig. 8. The syntax of the let-polymorphic language λ_{let} (new cases only)

The reason why $(\lambda id.[(id\ id)\ 2.0])$ is not typable is clear. In the original expression, id is used polymorphically, while in the decomposed subexpression, id is bound by λ and thus monomorphic. From this example, we observe that we need to preserve the polymorphic types of let-bound variables, when decomposing expressions.

Decomposition with let Context. To preserve polymorphic types of let-bound variables, we extend the context with a let context (Fig. 8). We also extend it with a fix context since it is a (monomorphic) binder. Using the let context, the above tree changes as follows, satisfying the decomposition property:

$$\dfrac{\vdash [1] : \texttt{int} \quad [+] : \texttt{int}\rightarrow\texttt{int}\rightarrow\texttt{int} \quad \boxed{\dfrac{\vdash [\lambda x.x] : \texttt{'a}\rightarrow\texttt{'a} \quad \vdash (let\ id = \lambda x.x\ in\ [(id\ id)\ 2.0]) : \texttt{float}}{\vdash [let\ id = \lambda x.x\ in\ (id\ id)\ 2.0] : \texttt{float}}}}{\vdash [1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)] \cdots \texttt{Type Error}}$$

Construction of the most General Type Tree. To enable inspection of the definition of let-bound variables, we change the decomposition function as shown in Fig. 9. The definition is the straightforward extension of the previous definition except for the variable case. When we decompose a variable, we search for its definition using *Get* defined in Fig. 10. When the variable is bound by a let expression, *Get* returns its (inner-most) definition as the decomposition of the variable. Otherwise, the variable is bound by lambda or fix, so *Get* returns no decomposition. Using this decomposition function, we can further debug into the definition of let-expressions to identify the source of a type error.

Since the context is extended with a let context and a fix context, the definition of *Collect* is extended accordingly as shown in Fig. 11. It collects types for

$$\begin{aligned}
Dec : context * term &\rightarrow (context * term)\ list \\
Dec[\![(C, x)]\!] &= Get(C, x, \square, \mathrm{None}) \\
Dec[\![(C, (M_1, ..., M_n))]\!] &= [(C, M_1); ...; (C, M_n)] \\
Dec[\![(C, fix\ f\ x \rightarrow M)]\!] &= [(C[fix\ f\ x \rightarrow \square], M)] \\
Dec[\![(C, let\ x = M_1\ in\ M_2)]\!] &= [(C, M_1); (C[let\ x = M_1\ in\ \square], M_2)]
\end{aligned}$$

Fig. 9. *Dec* for λ_{let} (new cases only)

$$\begin{aligned}
&Get : context * var * context * \\
&\quad (context * term)\ option \rightarrow (context * term)\ list \\
Get(\square, v, C, p) &= \begin{cases} [] & \text{if } p = \mathrm{None} \\ [(C', M)] & \text{if } p = \mathrm{Some}(C', M) \end{cases} \\
Get(\lambda x.C', v, C, p) &= \begin{cases} Get(C', v, C[\lambda x.\square], \mathrm{None}) & \text{if } x = v \\ Get(C', v, C[\lambda x.\square], p) & \text{if } x \neq v \end{cases} \\
Get(fix\ f\ x \rightarrow C', v, C, p) &= \begin{cases} Get(C', v, C[fix\ f\ x \rightarrow \square], \mathrm{None}) & \text{if } v \in \{f, x\} \\ Get(C', v, C[fix\ f\ x \rightarrow \square], p) & \text{if } v \notin \{f, x\} \end{cases} \\
Get(let\ x = M\ in\ C', v, C, p) &= \begin{cases} Get(C', v, C[let\ x = M\ in\ \square], \mathrm{Some}(C, M)) & \text{if } x = v \\ Get(C', v, C[let\ x = M\ in\ \square], p) & \text{if } x \neq v \end{cases}
\end{aligned}$$

Fig. 10. The function *Get* to search definition of variables for λ_{let}

$$env = (var * typ)\ list$$
$$Collect : context \to typ \to env \to (env * typ)$$
$$Collect_{fix\ f\ x \to C}[\![\tau_1 \to \tau_2]\!]\mu = Collect_C[\![\tau_2]\!]\mu[f \mapsto (\tau_1 \to \tau_2); x \mapsto \tau_1]$$
$$Collect_{let\ x=M\ in\ C}[\![\tau]\!]\mu = Collect_C[\![\tau]\!]\mu$$

Fig. 11. *Collect* for λ_{let} (new cases only)

lambda- and fix-bound variables and discards let-bound variables since they do not appear in the type returned by the compiler. (We assume that the compiler's type inferencer returns $\tau_1 \to \tau_2$ as the type of $fix\ f\ x \to M$ (and hence of f) where τ_1 is the type of x and τ_2 is the type of M.)

As the program to be debugged becomes larger, the number of let-bound variables increases. Since we can debug into the definition of let-bound variables when their types conflict with the programmer's intention, we can skip asking for the type of let-bound variables as an oracle each time. (For example, in the previous tree, the type debugger can skip the node $\vdash [\lambda x.x]$: `'a` $\to$`'a`). Rather, we only ask for variables in a context that are bound by lambda or fix. This is consistent with Chitil's approach that maintains an environment for polymorphic variables separately.

Decomposition for λ_{let} Satisfies the Decomposition Property. We can confirm that the decomposition property is still satisfied. The interesting case is for variables. (Other cases are similar to the reasoning shown for $\lambda_{\to}$.) Assume that $C[x]$ is well typed. We first observe that $Get(C_1, x, C_2, p)$ maintains an invariant that $C_2[C_1]$ is always the same across the recursive call, because at each recursive call, the topmost frame of C_1 is simply moved to the hole of C_2. This ensures that all the contexts appearing in the definition of *Get* are well typed (as contexts), because the initial context $[C[x]]$ with which *Get* is called from *Dec* is well typed. Next, the returned expression $C[M]$ is collected only from the let case. Because $C[let\ x = M\ in\ C']$ is well typed, we hence have that $C[M]$ is also well typed as required.

Observe how the decomposition property serves as a guideline for what we have to do and what we can do to incorporate let expressions. We have to define the decomposition function so that the let polymorphism is preserved. On the other hand, as long as the decomposition property is satisfied, we have the liberty of defining the decomposition in a way the debugging process becomes easier for programmers to understand. By defining the decomposition of let-bound variables as their definition, the debugger's focus moves from the use of variables to their definition.

5 Objects

So far, we have seen that interactive debugging is possible for various language constructs by suitably defining a *Dec* function that satisfies the decompositon property. This idea extends to advanced language constructs. In this section, we introduce objects and see how they can be supported in a similar way.

$$
\begin{array}{rll}
(L : classobj) ::= & inherit\ x & \text{(inheritance declaration)} \\
| & method\ x = M & \text{(method declaration)} \\
| & val\ x = M & \text{(value declaration)} \\
(M : term) ::= & ... \mid x_1 \# x_2 & \text{(method invocation)} \\
| & new\ x & \text{(object creation)} \\
| & class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ M & \text{(class definition)} \\
(\tau : typ) ::= & ... \mid obj & \text{(object type)} \\
(C : context) ::= & ... \mid class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ C & \text{(class context)}
\end{array}
$$

Fig. 12. The syntax of the object language λ_{obj} (new cases only)

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, x_1 \# x_2)]\!] & = & SearchObj(C, x_1, \Box, []) \\
Dec[\![(C, new\ x)]\!] & = & SearchObj(C, x, \Box, []) \\
Dec[\![(C, class\ x\ v_1 ... v_n = object(v') & = & [(C[class\ x\ v_1 ... v_n = object(v') \\
L_1 ... L_n\ end\ in\ M)]\!] & & L_1 ... L_n\ end\ in\ \Box], M)]
\end{array}
$$

Fig. 13. *Dec* for λ_{obj} (new cases only)

The Language. We show the syntax of the object language λ_{obj} in Fig. 12. It models OCaml-style objects where an object is defined using a class (in which single inheritance is allowed) and is created by the *new* construct. Besides the inheritance declaration, an object can contain method and value declarations. In OCaml, class names (to be more precise, the object structures denoted by the class names) are used as types. We use them as is in our type debugger, abbreviated as *obj* in Fig. 12.

Construction of the most General Type Tree. The decomposition function *Dec* is extended with the new constructs and the *Get* function used in the variable case is extended with the class context (Figs. 13, 14). The interesting cases are for *new* and method invocation of *Dec*. In both cases, we need to identify the object mentioned in the expressions (in case their types contradict with intended types, so that we can debug into the object). For this purpose, the function *SearchObj* in Fig. 15 is used. Its behavior is similar to that of *Get*, but differs in that *SearchObj* collects *all* the method declarations in the designated object. In particular, if the object contains inheritance declaration, those method declarations are collected, too (see $SearchObj'$).

We collect all the declarations in an object because types of declared methods in an object are mutually dependent. Thus, we need to ask for the types of

$$
\begin{array}{rcl}
Get : context * var * context * & & \\
(context * term)\ option & \rightarrow & (context * term)\ list \\
Get(class\ x\ v_1 ... v_n = object(v') & = & Get(C', v, C[class\ x\ v_1 ... v_n = object(v') \\
L_1 ... L_n\ end\ in\ C', v, C, p) & & L_1 ... L_n\ end\ in\ \Box], p)
\end{array}
$$

Fig. 14. *Get* for λ_{obj} (new cases only)

$$
\begin{aligned}
SearchObj' &: classobj\ list * context \to (context * term)\ list \\
SearchObj'([], C) &= [] \\
SearchObj'((inherit\ x) :: r, C) &= SearchObj(C, x, \Box, [])@SearchObj'(r, C) \\
SearchObj'((method\ x = M) :: r, C) &= (C, M) :: SearchObj'(r, C) \\
SearchObj'((val\ x = M) :: r, C) &= SarchObj'(r, C[let\ x = M\ in\ \Box])
\end{aligned}
$$

$$
\begin{aligned}
SearchObj &: context * var * context * \\
&\quad (context * term)\ list \to (context * term)\ list \\
SearchObj(\Box, v, C, p) &= p \\
SearchObj(\lambda x.C', v, C, p) &= SearchObj(C', v, C[\lambda x.\Box], p) \\
SearchObj(fix\ f\ x \to C', v, C, p) &= SearchObj(C', v, C[fix\ f\ x \to \Box], p) \\
SearchObj(class\ x\ v_1...v_n = object(v')\ L_1...L_n\ end\ in\ C', v, C, p) &= \\
&\text{if } x = v \text{ then} \\
&\quad SearchObj(C', v, C[class\ x\ v_1...v_n = object(v')\ L_1...L_n\ end\ in\ \Box], \\
&\qquad SearchObj'(L_1...L_n, C[\lambda v_1...\lambda v_n.\lambda v'_v.\Box])) \\
&\text{else } SearchObj(C', v, C[class\ x\ v_1...v_n = object(v')\ L_1...L_n\ end\ in\ \Box], p)
\end{aligned}
$$

Fig. 15. The function *SearchObj* to search for the definition of objects for λ_{obj}

all these method declarations to locate the source of type errors. For example, consider the following program:

```
class counter = object (self)
                  val mutable n = 0
                  method incr = n <- n+1
                  method get = n
                end
let t = (new counter) in
t#incr; ("now, the conter is" ^ t#get)
```

The last line results in a type error, because `t#get` returns an integer, which is in conflict with the intended type (i.e., `string`). To find the source of this type error, we first look up `t`'s class definition `counter` and search for the definition of the `get` method. However, we find here that the `get` method itself does not force the type of `n` as an integer. It simply returns a value of `n`. Instead, `n` is an integer because it is assigned `0` and `n+1` elsewhere in the class. Thus, we need to examine all the declarations in an object to find the source of type errors.

Since any method declarations can be the source of type errors, we collect all the method declarations in a class definition, and return them as decomposition of the object reference. Although this strategy is necessary in general, it could lead to a large number of questions. Its practical implementation is future work.

Decomposition for λ_{obj} Satisfies the Decomposition Property. We can confirm that *Dec* satisfies the decomposition property as follows. First, *Get* will return a list of well-typed subexpressions only, using the similar argument we described in Sect. 4. For *new* and method invocation, we have to show that *SearchObj* returns a list of well-typed subexpressions. It can be proved by observing that *SearchObj* simply collects subexpressions in an object in a suitable context. The

only interesting case is for a class declaration, where we have to properly insert bindings for the arguments to the class and the self variable v'. Note that declared values are put into let contexts in $SearchObj'$.

6 Implementation

We have implemented a type debugger for OCaml 3.12.1. To minimize the implementation efforts, we utilize the following components from OCaml as is:

- the abstract syntax tree for structures, expressions, and types (together with the lexer, the parser, and the pretty printer)
- the type inferencer `typing` (that accepts an expression and returns its type, both expressed using the above abstract syntax tree)
- the `is_expansive` function (that accepts an expression and returns a boolean to judge whether the given expression needs to be kept monomorphic or not)

By using exactly the same abstract syntax as OCaml, we can not only avoid reproducing the same abstract syntax but also utilize OCaml's own lexer, parser, and pretty printer. In addition to the type inferencer, we utilize the `is_expansive` function. Although OCaml has its own criteria for weak polymorphism [2], we can stay away from it by using OCaml's `is_expansive` function as is. Furthermore, this approach is robust to updates of OCaml: if the syntax and the interface of the two functions are the same, we can use the same debugger.

A slight complication is that OCaml treats a let expression without `in` differently from the one with `in`: the former is a structure, while the latter is an expression. We support both styles by splitting the context into two: the structure part and the expression part.

Another complication is the use of patterns in place of a variable declaration. For example, instead of `fun lst ->`, one can write `fun (first :: rest) ->`. Because patterns have type constraints, they may be the source of a type error. To make such an error detectable, we included patterns as the decomposition of the expression.

The rest of the language constructs are supported without requiring any special treatment. For each new construct, we define its decomposition and show that it satisfies the decomposition property. Our type debugger supports all features of OCaml including weak polymorphism and modules.

To construct the most general type tree, we use the compiler's type inferencer many times. Although it appears that our type debugger incurs significant overhead, this is not the case, because we do not have to construct the whole tree beforehand. Instead, the most general type tree is constructed as we debug: after the root node is constructed, the rest of the tree can be constructed during the interaction with the programmer.

7 Related Work

The typical approach to improving type error messages is to design a new type inference algorithm. Wand [16] keeps track of the history how type variables

are instantiated and shows the conflicting history when a type error arises. Lee and Yi [6] present the algorithm M that finds conflict of types earlier than the algorithm W and thus reports a narrower expression as an error. Heeren and Hage [5] use a constraint-based type inference for improving type error messages. Although these improved type error messages are useful for programmers, it is in general not possible to identify the source of type errors by a single error message.

To locate the source of type errors, Chitil [1] uses compositional type inference and constructs an interactive type debugger for a subset of Haskell. Based on his work, we designed a type debugger for OCaml using the compiler's own type inferencer rather than a tailor-made type inferencer. The use of the compiler's type inferencer enables us to build a type debugger for a larger language easily. Stuckey, Sulzmann, and Wazny [14] find the source of type errors using type inference via CHR solving. They implement a type debugger called Chameleon, which can explain why an inferenced type is derived by searching. Tailor-made type inference is used for this purpose.

As different approaches, Haack and Wells [3] use slicing with respect to types to narrow the possibly erroneous parts of programs. By extracting the slice related to type errors, they help the programmer to identify the source of type errors. The advantage of this approach is that the process is automatic and the programmer does not have to answer questions. Schilling [10] obtains slices using the compiler's type inferencer. We share the goal of reusing the available resources in the compiler.

Lerner et al. [7] propose automatic type-error correction. They replace the erroneous part with various syntactically correct similar expressions, and see if they type check. If they do, they are displayed as the candidates for fixing the type error. Since the system automatically shows us possible fixes without intervention, the system is useful if the programmer's intended fix is shown. Unfortunately, it does not always produce the intended program.

As visualizing tools of types, Simon, Chitil, and Huch [13] show TypeView that allows programmers to browse through the source code and to query the types of each expression. McAdam [8] displays types as graphs and extracts various facts from them that are useful for debugging. Our previous Emacs interface [15] inspired by these works, and we will continue to build such interface.

8 Conclusion

In this paper, we have fleshed out our thesis that it is possible and also practical to write a type debugger by piggy-backing on the built-in type inferencer of an existing compiler. The key observation is that we only need the most general type tree with the decomposition property; such a tree can be constructed using the compiler's type inferencer. The decomposition property guided the design of our type debugger: we maintained contexts so that the property is satisfied all the time. We have illustrated the thesis with OCaml, and we have described how to handle a number of issues: simple types, let polymorphism, and objects. Our design is in use in our laboratory and in our classrooms.

We plan to continue the present line of work as follows. First, we want to explore how far the idea presented in this paper scales. In particular, we are interested in supporting type classes [4] in Haskell and GADTs introduced in OCaml 4.0. We will investigate how we can define decomposition of a program with type classes or GADTs and see if it satisfies the property (Sect. 3). Secondly, we want to perform thorough user tests. We have built an Emacs interface based on our previous work [15] and the type debugger is in use in several courses in our university. From user tests, we plan to obtain various feedback including usefulness and how to effectively show the type information to novices. Finally, we want to establish some kind of correctness criteria of the type debugger. By considering the most general type tree, it might become possible to formally state a property such as the type debugger would always find the source of a type error.

Aknowledgements. We would like to thank Olaf Chitil, Olivier Danvy, Ian Zerny, IFL participants, and anonymous reviewers for valuable comments and discussions.

References

1. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01), pp. 193–204 (2001)
2. Garrigue, J.: Relaxing the value restriction. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 196–213. Springer, Heidelberg (2004)
3. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 284–301. Springer, Heidelberg (2003)
4. Hall, C., Hammond, K., Jones, S.P., Wadler, P.: Type classes in Haskell. ACM Trans. Program. Lang. Syst. (TOPLAS) **18**(2), 241–256 (1996)
5. Heeren, B., Hage, J.: Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, Utrecht University (2002)
6. Lee, O., Yi, K.: Proofs about a folklore let-polymorphic type inference algorithm. ACM Trans. Program. Lang. Syst. **20**(4), 707–723 (1998)
7. Lerner, B.S., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), pp. 425–434 (2007)
8. McAdam, B.J.: Generalising techniques for type debugging, chapter 6. In: Trinder, P., Michaelson, G., Loidl, H.-W. (eds.) Trends in Functional Programming, pp. 49–57. Intellect, Portland (2000)
9. Nilsson, H.: Declarative debugging for lazy functional languages. Ph.D. thesis, Linköping, Sweden (1998)
10. Schilling, T.: Constraint-free type error slicing. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 1–16. Springer, Heidelberg (2012)
11. Shapiro, E.Y.: Algorithmic program debugging. MIT Press, Cambridge (1983)
12. Silva, J., Chitil, O.: Combining algorithmic debugging and program slicing. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06), pp. 157–166 (2006)

13. Simon, A., Chitil, O., Huch, F.: Typeview: a tool for understanding type errors. In: Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages, pp. 63–69 (2000)
14. Stuckey, P. J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell'03), pp. 72–83 (2003)
15. Tsushima, K., Asai, K.: Report on an OCaml type debugger. In: ACM SIGPLAN Workshop on ML, 3 p. (2011)
16. Wand, M.: Finding the source of type errors. In: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'86), pp. 38–43 (1986)