Kent Academic Repository

Full text document (pdf)

Citation for published version

Tsushima, Kanae and Chitil, Olaf (2018) A Common Framework Using Expected Types for Several Type Debugging Approaches. In: FLOPS 2018: Fourteenth International Symposium on Functional and Logic Programming, May 9-11, 2018, Nagoya, Japan.

DOI

https://doi.org/10.1007/978-3-319-90686-7_15

Link to record in KAR

http://kar.kent.ac.uk/66352/

Document Version

Author's Accepted Manuscript

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

A Common Framework Using Expected Types for Several Type Debugging Approaches

Kanae Tsushima
National Institute of Informatics, Japan
k_tsushima@nii.ac.jp

Olaf Chitil
University of Kent, UK
O.Chitil@kent.ac.uk

Abstract. Many different approaches to type error debugging were developed independently. In this paper, we describe a new common framework for several type error debugging approaches. For this purpose, we introduce expected types from the outer context and propose a method for obtaining them. Using expected types, we develop three type error debugging approaches: enumeration of type error messages, type error slicing and (improved) interactive type error debugging. Based on our idea we implemented prototypes and confirm that the framework works well for type debugging.

1 Introduction

The Hindley-Milner type system is the core of the type systems of many statically typed functional programming languages such as ML, OCaml and Haskell. The programmer does not have to write any type annotations in the program; instead most general types are inferred automatically. Functions defined in the program can be used with many different types.

However, type error debugging is a well-known problem: When a program cannot be typed, the type error messages produced by the compiler often do not help much with locating and fixing the cause(s). Consider the following OCaml program:

```
let rec f lst n = match lst with
 | [] -> []
 | fst :: rest -> (fst ^ n) :: (f rest n) in
f [2]
```

We assume that the programmer intended to define the function that maps a list of numbers to a list of squared numbers (e.g., `f [3] 2 = [9]`). Hence the programmer's intended type of `f` is *int list* → *int* → *int list*. However, in OCaml `^` is the string concatenation function. Therefore, the type of the function `f` is inferred as *string list* → *string* → *string list*. The OCaml compiler returns the following type error message:

```
f [2]  ;;
Error: This expression has type int but
       an expression was expected of type string
```

The message states that the underlined expression `2` has type *int*, but it is expected to have type *string* because of other parts of this program. The message does not substantially help the programmer: they will wonder why `2` should have type *string*.

The starting point for our work is the principal typing tree that was introduced by Chitil [2] for interactive type error debugging.

A principal typing $\Gamma \vdash \tau$ of some expression e consists of a type τ and an environment Γ which gives types to all free variables of e. All typings for an expression are instances of its principal typing. For example, the principal typing for a variable `x` is $\{x : \alpha\} \vdash \alpha$, where α is a type variable. In contrast, the better known principal type τ is just the most general type for a given expression e and given environment Γ.

The principal typing tree is the syntax tree of a program where each node describes a subexpression of the program and includes the principal typing for this subexpression. Chitil applied algorithmic debugging [9] to the tree to locate the source of a type error. Tsushima and Asai proposed an approach to construct the principal typing tree using the compiler's type inferencer and implemented a type error debugger for OCaml [12]. The implemented type debugger has been used in classes at Ochanomizu University for the past 5 years by approximately 200 novice programmers [5].

In this paper we claim that **expected typings** are also useful for type debugging. Expected typings are duals to principal typings. Let us consider a program $C[e]$ that consists of an expression e and its context C. The expression e on its own has a principal typing. The expected typing of e is the type of the hole of the context C, disregarding e. In the previous example the principal type of `2` is *int*, but its expected type, according to the context, is *string*. The type error message of the OCaml compiler actually stated both types.

In this paper we introduce the **type debugging information tree**. The tree for our example program is shown in Figure 1. The tree includes for each subexpression both its principal and expected typing (Section 3).

We show how the tree can be used to realise the following type error debugging approaches:

1. Enumeration of type error messages.
   Type error messages show two conflicting types. We can obtain both types from the type debugging information tree (Section 4.1).
2. Type error slicing.
   Only well-typed contexts yield expected typings. Hence program parts that do not yield expected typings do not contribute to a conflict between principal and expected typing (Section 4.2).
3. Interactive type error debugging.
   We can use the programmer's intended types to derive expected types. With expected types, we can efficiently narrow the area to debug (Section 4.3).

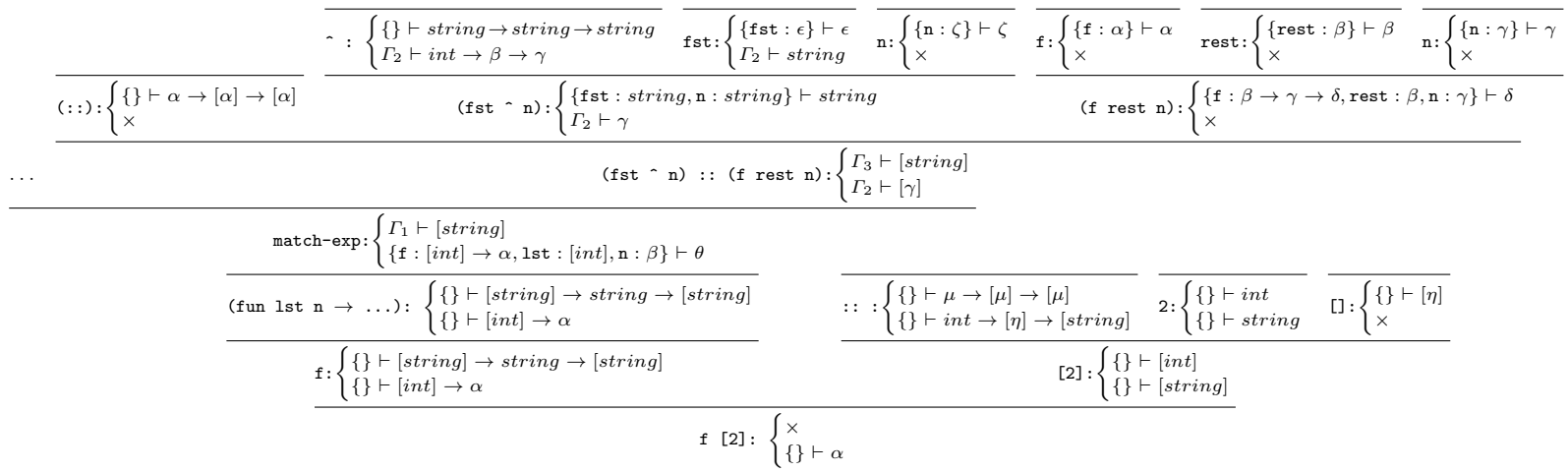


Fig. 1. Type debugging information tree for the program from the Introduction

$$
\begin{array}{rll}
p ::= & \texttt{c} & \text{(constant)} \\
| & \texttt{v} & \text{(variable)} \\
| & p \ \texttt{::} \ p & \text{(cons pattern)} \\
| & \texttt{(}p\texttt{,}\ p\texttt{)} & \text{(tuple pattern)} \\
\\
M ::= & \texttt{c}^l & \text{(constant)} \\
| & \texttt{v}^l & \text{(variable)} \\
| & \texttt{fun}^l \ \texttt{x} \rightarrow M & \text{(function)} \\
| & \texttt{(}M\ M\texttt{)}^l & \text{(application)} \\
| & \texttt{(}M, M\texttt{)}^l & \text{(tuple)} \\
| & \texttt{if}^l \ M \ \texttt{then} \ M \ \texttt{else} \ M & \text{(if expression)} \\
| & \texttt{match}^l \ M \ \texttt{with} \mid p \rightarrow M & \text{(match expression)} \\
\\
l ::= & \textit{location number} & \\
\\
S ::= & \texttt{let rec x =} \ M \ \texttt{in} \ S & \text{(let expression)} \\
| & M & \\
\\
\tau ::= & \textit{int} \mid \textit{bool} \mid ... & \text{(constant type)} \\
| & \alpha & \text{(type variable)} \\
| & \tau \rightarrow \tau & \text{(function type)} \\
| & \tau * \tau & \text{(tuple type)} \\
| & [\tau] & \text{(list type)}
\end{array}
$$

Fig. 2. OCaml-like functional language

2 Language and Principal Typing Tree (PTT)

We describe our framework for a core functional language, a small subset of OCaml, as defined in Figure 2. Every expression construct is annotated with some unique location information l. To save space, we use Haskell's notation for list types (e.g., instead of *int list* we write $[int]$). A program S consists of a sequence of function definitions defined by a recursive let construct. However, this let-rec construct does not appear in any of the trees that we define in the following sections. Instead, we unfold the definition of a let-rec defined variable at every use occurrence in the tree. Note that this 'unfolding' is only conceptual to obtain a simple tree structure; the actual implementation handles each let-rec construct only once and actually constructs a directed graph, which, unfolded, yields a tree.

Figure 1 demonstrates this unfolding for the program in the Introduction. It shows part of the type debugging information tree where the tree node for `f` has the definition of `f` as child node.

A typing consists of a type environment Γ, a finite mapping from free variables to types, and a type τ. Instead of the more common $\Gamma \vdash M : \tau$ we write $M : \Gamma \vdash \tau$ to state that expression M has typing $\Gamma \vdash \tau$. The special typing $\times$ indicates that no typing of the form $\Gamma \vdash \tau$ exists; so $\times$ indicates a type error.

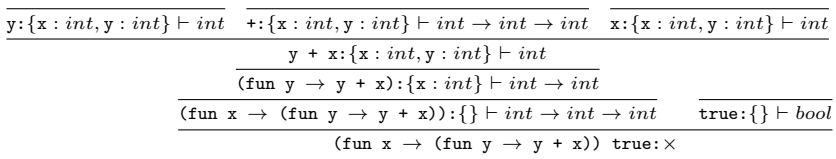


Fig. 3. Standard type inference tree of `(fun x -> (fun y -> y + x)) true`

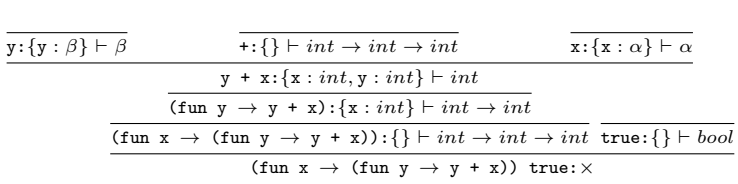


Fig. 4. Most general type tree of `(fun x -> (fun y -> y + x)) true`

We obtain expected typings from principal typings. For a program a syntax tree where each node is annotated with a principal typing can be constructed[1]; details are given by Chitil, Tsushima and Asai [2, 12].

Most type inference algorithms try to construct a tree based on the Hindley-Milner rules. To see how the PTT differs, consider the following ill-typed OCaml program: `(fun x -> (fun y -> y + x)) true`

The function `+` adds two integers; its type is $int \to int \to int$. The constant `true` has type *bool*. Hence the program is ill-typed.

Figure 3 shows a Hindley-Milner type inference tree and Figure 4 shows the PTT for this program. In the PTT each subexpression appears with its principal typing. For example, in the left upper part of Figure 4, `y` has type β not *int* because there is no constraint to have type *int* in `y` itself; also the variable `x` does not appear in this type environment at all. In contrast in Figure 3, the subexpression `y` has the type environment $\{\mathtt{x} : int, \mathtt{y} : int\}$, because of constraints in other parts of the tree.

3 Expected Typings & Type Debugging Information Tree

We obtain the type debugging information tree from the PTT by adding an expected typing to every node of the tree. So in every tree node an expression is annotated with both its principal and its expected typing.

At first we focus on programs that have only a single type error. Afterwards we show why we have to obtain expected typings from principal typings, not the typings of a standard Hindley-Milner type inference tree. Finally we show how

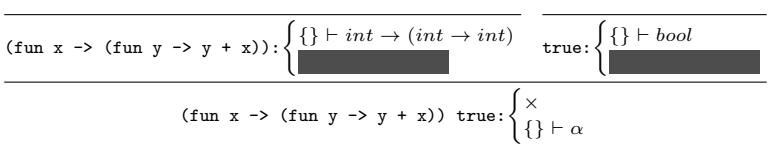
[1] It is actually a tree of principal monomorphic typings; let-bound potentially polymorphic variables do not appear in the type environment [2].

we can obtain expected typings from programs that have multiple type errors, the normal case in practice.

3.1 How to Obtain Expected Types

For simplicity we assume that we have an ill-typed program with a PTT that has only one type error node. Let us reconsider the example from the preceding section: `(fun x -> (fun y -> y + x)) true` The PTT of this program is shown in Figure 4.

Inference of expected typings starts from the root of the tree. We assume that the expected type of the whole program is some type variable α, which means that there is no type constraint. Because the whole program has no free variables, its type environment is empty. Our next goal is to infer the expected typings shown as black boxes below. The first box is the expected typing of the function `(fun x -> (fun y -> y + x))` and the second box is the expected typing of `true`.



The idea for obtaining the expected typing of an expression M is that we do not use the typing of M itself but use the typing of its sibling nodes and parent node.

Here the type environments are empty and we are solely concerned with types. We obtain the expected type of `(fun x -> (fun y -> y + x))` from the principal type of the argument `true` and the expected type of the whole program. We can visualize the constraint of the function application as follows:

$$\texttt{(fun x -> (fun y -> y + x)):}\blacksquare\texttt{) (true:}\mathit{bool}\texttt{)} \;:\; \alpha$$

Thus we see that the black box must be $\mathit{bool} \to \alpha$. Similarly we can obtain the expected type of `true`, namely int.

3.2 Inferring Expected Typings

Figure 5 shows our inference rules for expected typings. We assume that all principal typings and the expected typing of an expression at the bottom of a rule are known. The rules define the expected typings for the subexpressions on top of the rules. In the figure these inferred expected typings are boxed.

For many language constructs we need to compose several type environments and solve a set of equational type constraints. Hence we define and use a most general unifier function called mgu. The function call

$$\mathrm{mgu}(\{\Gamma_1, \ldots, \Gamma_n\}, \{\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'\}, \tau)$$

$$\frac{}{\mathtt{n}^l : \begin{cases}(\Gamma_i \vdash \tau_i, \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}} \qquad \frac{}{\mathtt{v}^l : \begin{cases}(\Gamma_i \vdash \tau_i, \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

$$\frac{M : \begin{cases}(\Gamma_{i'} \vdash \tau_{i'}, L_0)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e\}, \{\tau_e = \alpha \to \beta\}, \beta), L_e \cup \{l\})}\end{cases}}{\mathtt{fun}^l\ \mathtt{x} \to M : \begin{cases}(\Gamma_i \vdash \tau_i, L_0 \cup \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

$$\frac{\begin{array}{l}M_0 : \begin{cases}(\Gamma_0 \vdash \tau_0, L_0)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e, \Gamma_1\}, \{\}, \tau_1 \to \tau_e), L_e \cup L_1 \cup \{l\})}\end{cases}\\ M_1 : \begin{cases}(\Gamma_1 \vdash \tau_1, L_1)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e, \Gamma_0\}, \{\tau_0 = \alpha \to \tau_e\}, \alpha), L_e \cup L_0 \cup \{l\})}\end{cases}\end{array}}{(M_0\ \ M_1)^l : \begin{cases}(\Gamma_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

$$\frac{\begin{array}{l}M_0 : \begin{cases}(\Gamma_0 \vdash \tau_0, L_0)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e, \Gamma_1\}, \{\tau_e = \alpha * \beta\}, \alpha), L_e \cup L_1 \cup \{l\})}\end{cases}\\ M_1 : \begin{cases}(\Gamma_1 \vdash \tau_1, L_1)\\ \boxed{\mathrm{mgu}(\{\Gamma_e, \Gamma_0\}, \{\tau_e = \alpha * \beta\}, \beta), L_e \cup L_0 \cup \{l\})}\end{cases}\end{array}}{\mathtt{(}M_0\mathtt{,}\ M_1\mathtt{)}^l : \begin{cases}(\Gamma_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

$$\frac{\begin{array}{l}M_0 : \begin{cases}(\Gamma_0 \vdash \tau_0, L_0)\\ \boxed{(\mathrm{mgu}(\{\Gamma_1, \Gamma_2, \Gamma_e\}, \{\tau_1 = \tau_e, \tau_2 = \tau_e\}, \mathit{bool}), L_e \cup L_1 \cup L_2 \cup \{l\})}\end{cases}\\ M_1 : \begin{cases}(\Gamma_1 \vdash \tau_1, L_1)\\ \boxed{(\mathrm{mgu}(\{\Gamma_0, \Gamma_2, \Gamma_e\}, \{\tau_0 = \mathit{bool}, \tau_2 = \tau_e\}, \tau_e), L_e \cup L_0 \cup L_2 \cup \{l\})}\end{cases}\\ M_2 : \begin{cases}(\Gamma_2 \vdash \tau_2, L_2)\\ \boxed{(\mathrm{mgu}(\{\Gamma_0, \Gamma_2, \Gamma_e\}, \{\tau_0 = \mathit{bool}, \tau_1 = \tau_e\}, \tau_e), L_e \cup L_0 \cup L_1 \cup \{l\})}\end{cases}\end{array}}{\mathtt{if}^l\ M_0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2\mathtt{:}\ \begin{cases}(\Gamma_i \vdash \tau_i, L_0 \cup L_1 \cup L_2 \cup \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

$$\frac{\begin{array}{l}M_0 : \begin{cases}((\Gamma_0, \tau_0), L_0)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e, \Gamma_1, \Gamma_p\}, \{\tau_e = \tau_1\}, \tau_p) \setminus (\mathrm{fv}(p)), L_e \cup L_1 \cup \{l\})}\end{cases}\\ M_1 : \begin{cases}((\Gamma_1, \tau_1), L_1)\\ \boxed{(\mathrm{mgu}(\{\Gamma_e, \Gamma_0, \Gamma_p\}, \{\tau_0 = \tau_p\}, \tau_e), L_e \cup L_0 \cup \{l\})}\end{cases}\\ \text{where } \Gamma_p \vdash \tau_p \text{ is the principal typing of } p\end{array}}{\mathtt{match}^l\ M_0\ \mathtt{with}\ \mathtt{|}\ \ p \to M_1\ \mathtt{:}\ \begin{cases}(\Gamma_i \vdash \tau_i, L_0 \cup L_1 \cup \{l\})\\(\Gamma_e \vdash \tau_e, L_e)\end{cases}}$$

Fig. 5. Inference rules for expected typings

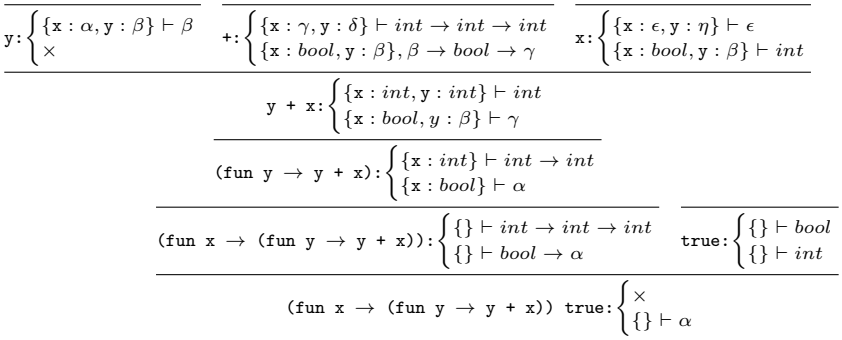


Fig. 6. Type debugging information tree of `(fun x -> (fun y -> y + x)) true`

computes a most general type substitution σ with

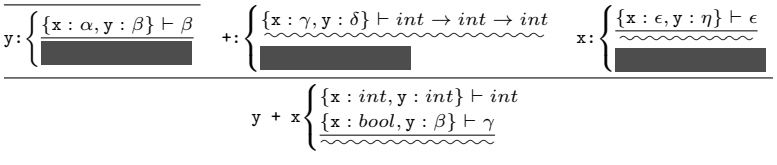
$$\begin{aligned} &\Gamma_1\sigma = \ldots = \Gamma_n\sigma \\ &\tau_1\sigma = \tau'_1\sigma,\ \ \ldots,\ \tau_n\sigma = \tau'_n\sigma \end{aligned}$$

and returns the typing

$$\Gamma_1\sigma \vdash \tau\sigma$$

If no such substitution exists, it returns $\times$.

To understand the function mgu and the inference rule for application, let us consider the top of Figure 6:



First, let us determine the expected typing of `+`. We use the underlined typings: the expected typing of `+`'s parent node `y + x` and the principal typings of `y` and `x`. Because these type constraints must be satisfied simultaneously, their environments must be composed. Because `+` is a function applied to the two arguments, we have the additional constraint that its expected type is $\beta \to \epsilon \to \gamma$. In summary, `+` has the expected typing

$$\begin{aligned} &\mathrm{mgu}(\{\{\mathtt{x} : \alpha, \mathtt{y} : \beta\}, \{\mathtt{x} : \epsilon, \mathtt{y} : \eta\}, \{\mathtt{x} : \mathit{bool}, \mathtt{y} : \beta\}\}, \{\}, \beta \to \epsilon \to \gamma) \\ &\quad = \{\mathtt{x} : \mathit{bool}, \mathtt{y} : \beta\} \vdash \beta \to \mathit{bool} \to \gamma \end{aligned}$$

Second, let us determine the expected typing of `y`. in the top of Figure 6. We use the curve-underlined typings: the expected typing of `y + x` and the principal typings of `+` and `x`. We need to combine the type environments of the

three typings and express the constraint that `+` is a function with `y` and `x` as arguments. In summary, `y` has the expected typing

$$\begin{aligned}&\mathrm{mgu}(\{\{\mathtt{x}:\gamma,\mathtt{y}:\delta\},\{\mathtt{x}:\epsilon,\mathtt{y}:\eta\},\\&\qquad\{\mathtt{x}:\mathit{bool},\mathtt{y}:\beta\}\},\{\mathit{int}\to\mathit{int}\to\mathit{int}=\kappa\to\epsilon\to\gamma\},\kappa)\\&\quad=\times\end{aligned}$$

We obtain $\times$, because there does not exist any unifying substitution.

Nearly all our inference rules in Figure 5 use the mgu function. The only exception are the rules for constants and variables, because they have no smaller components; their own expected typings are determined by their contexts. The pattern match construct binds new variables in the pattern p. Hence these variables have to be removed from the typing for the term M_0.

3.3 Type Annotations / Signatures

Unlike all real functional programming languages, our core language does not have type annotations, which allows the user to specify that a function should have a certain type. However, if we do consider type annotations, then these naturally contribute to our expected typings. Expected typings propagate the user's annotations towards the leaves of the type debugging information tree. Thus expected typings become more informative and we also have more expected typings $\times$.

3.4 Why Obtain Expected Typings from Principal Typings?

We cannot obtain expected typings by means of the standard type inference tree. Consider again the expression `(fun x -> (fun y -> y + x)) true`. We obtain the following tree from the Hindley-Milner type rules:

$$\dfrac{\mathtt{y}:\left\{\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\}\vdash\mathit{int}\right.\quad\mathtt{+}:\left\{\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\}\vdash\mathit{int}\to\mathit{int}\to\mathit{int}\right.\quad\mathtt{x}:\left\{\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\}\vdash\mathit{int}\right.}{\mathtt{y + x}:\begin{cases}\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\}\vdash\mathit{int}\\\{\mathtt{x}:\mathit{bool},\mathtt{y}:\beta\}\vdash\gamma\end{cases}}$$

We want to determine the expected typing of `+`. However, if we now compose the type environments for `y`, `x` and `y + x`, we get

$$\begin{aligned}&\mathrm{mgu}(\{\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\},\{\mathtt{x}:\mathit{int},\mathtt{y}:\mathit{int}\},\{\mathtt{x}:\mathit{bool},\mathtt{y}:\beta\}\},\{\},\mathit{int}\to\mathit{int}\to\gamma)\\&\quad=\times\end{aligned}$$

because the types of `x` in the type environments are already in conflict. We cannot use the standard type inference tree, because type information in type environments includes type constraints of many parts of the program.

3.5 Expected Typings in the Presence of Multiple Type Errors

If a program has only a single type error, then the expected typing of an expression is derived from all other parts of the program except the expression. In the presence of multiple type errors, we should not derive the expected typing of an expression from all other parts, because these other parts contain type errors and hence many typings are $\times$. Note that even in the presence of a single type error several principal and expected typings are already $\times$, but we simply exclude these when inferring expected typings. However, if we applied the same method in the presence of multiple type errors, then we would lose too much type information and our expected typings would become useless.

To control which type constraints to include and which not, we need to record the program parts that contribute to the expected typings, which we represent with the notation $(\Gamma \vdash \tau, \{l_1, \ldots, l_n\})$. This means that typing $\Gamma \vdash \tau$ is derived from type information of the subexpressions with the locations $l_1, .., l_n$.

Let us consider the following multiple type error example: `((x 1, y 2), (x false, y true))`. This program includes multiple type errors, because there are two type conflicts about `x` and `y`. Let us focus on `(x 1, y 2)` in this program. In the first step we obtain the following PTT.

$$\frac{\texttt{x 1} : (\{\texttt{x} : int \to \alpha\} \vdash \alpha, L_0) \qquad \texttt{y 2} : (\{\texttt{y} : int \to \beta\} \vdash \beta, L_1)}{\texttt{(x 1, y 2)}^{l} \colon\ (\{\texttt{x} : int \to \alpha; \texttt{y} : int \to \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\})}$$

In the second step we obtain the following tree by the rules in Figure 5.

$$\frac{\texttt{x 1} : \begin{cases} (\{\texttt{x} : int \to \alpha\} \vdash \alpha, L_0) \\ \times \end{cases} \qquad \texttt{y 2} : \begin{cases} (\{\texttt{y} : int \to \beta\} \vdash \beta, L_1) \\ \times \end{cases}}{\texttt{(x 1, y 2)}^{l} \colon\ \begin{cases} (\{\texttt{x} : int \to \alpha; \texttt{y} : int \to \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\}) \\ \boxed{(\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\} \vdash \epsilon, L_e)} \end{cases}}$$

The expected typing of `(x 1, y 2)` is obtained from its sibling `(x false, y true)`'s principal typing. We successfully obtain the expected typing for `(x 1, y 2)` (the boxed part). Because the principal typings of `x 1` and `y 2` have conflicts with the expected typing of their parent node, each child has expected typing $\times$. We need more expected typings that we could not obtain in this step. In the third step, we reconstruct the following abstracted PTT using the upper tree information.

$$\frac{\texttt{x 1} : (\{\} \vdash \alpha', \{\}) \qquad \texttt{y 2} : (\{\} \vdash \beta', \{\})}{\texttt{(x 1, y 2)}^{l} \colon\ (\{\} \vdash \alpha' * \beta', \{l\})}$$

The reconstruction rules of PTT is the following. If an expression has an expected typing $\times$ in the past steps, we put the abstracted typing $(\{\} \vdash \gamma, \{\})$ (γ is a new type variable that does not appear anywhere. The second $\{\}$ means that this typing does not include any constraints) for it. Otherwise we use standard method in [2, 12] for obtaining their principal typings. So in this case, the principal typings of `x 1` and `y 2` are abstracted typings. On the other hand, the principal typing of `(x 1, y 2)` is not abstracted and inferred by its children. This is because `(x 1, y 2)` has expected typing in the second step. In the fourth step we apply our rules in Figure 5 to this tree and obtain the following tree.

$$\texttt{x 1}: \begin{cases} (\{\} \vdash \alpha', \{\}) \\ \boxed{(\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\}, \zeta, L_e \cup \{l\})} \end{cases}$$
$$\texttt{y 2}: \begin{cases} (\{\} \vdash \beta', \{\}) \\ \boxed{(\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\}, \eta, L_e \cup \{l\})} \end{cases}$$
$$\overline{\texttt{(x 1, y 2)}^{l}: \begin{cases} (\{\} \vdash \alpha' * \beta', \{l\}) \\ (\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\} \vdash \epsilon, L_e) \end{cases}}$$

In the fourth step we could obtain the expected typings of `x 1` and `y 2` (the boxed parts). Thanks to these expected typings, we can obtain their children's expected typings subsequently. Finally we can obtain the following type debugging information tree.

$$\texttt{x 1}: \begin{cases} (\{\texttt{x} : int \to \alpha\} \vdash \alpha, L_0) \\ (\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\}, \zeta, L_e \cup \{l\}) \end{cases}$$
$$\texttt{y 2}: \begin{cases} (\{\texttt{y} : int \to \beta\} \vdash \beta, L_1) \\ (\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\}, \eta, L_e \cup \{l\}) \end{cases}$$
$$\overline{\texttt{(x 1, y 2)}^{l}: \begin{cases} (\{\texttt{x} : int \to \alpha; \texttt{y} : int \to \beta\} \vdash \alpha * \beta, L_0 \cup L_1 \cup \{l\}) \\ (\{\texttt{x} : bool \to \gamma; \texttt{y} : bool \to \delta\} \vdash \epsilon, L_e) \end{cases}}$$

The algorithm in presence of multiple type errors is the following.

1 (Re)construct PTT. If the expected typing of an expression in the past steps is $\times$, we use abstracted typing $(\{\} \vdash \gamma, \{\})$ (γ is a new type variable that does not appear anywhere). Otherwise we use standard method to infer its principal typing in [2, 12].
2 By the rules in Figure 5. Infer expected typings that we could not obtain yet.
3 If we obtain new expected typings in step 2, we repeat step 1. Otherwise construction of type debugging information tree is finished.

For doing this, we need a restriction: each tree node has at most two children. Thanks to this restriction we can determine if the sibling node information is needed or not. So before inferring principal and expected typings we transform the tree such that every node has at most two children.

4 Using the Type Debugging Information Tree

In this section we show how the type debugging information tree and its typings can be used to develop different type debugging tools.

4.1 Enumeration of Type Error Messages

The type error messages of a compiler are most familiar to programmers. For an ill-typed program the OCaml compiler stops after producing one type error message. From our type error debugging tree we can easily produce many type error messages: Every tree node where the principal typing and expected typing

are in conflict, that is, cannot be unified, is a type error node and thus yields a type error message.

For example, the tree in Figure 1 yields four error messages about leaves (`^`, `fst`, `2`, `:: (in [2])`) and six error messages about inner nodes.

For each type error node we can produce a type error message as follows:

```
(fst ^ n)
This expression has type string -> string -> string
but an expression was expected of type int -> 'b -> 'c
```

In practice the order in which the type error messages are shown is important. The most likely causes, based on some heuristics, should be shown first [1].

4.2 Type Error Slicing

A type error slice is a slice of a program that on its own is ill-typed. A type error slicer receives an ill-typed program and returns one or many type error slices. For our example

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

the type debugging information tree enables us to produce the type error slice

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

In a slice `..` denotes any subexpression that does not belong to the slice.

We obtained the type error slice by simply removing any subexpression that has an expected typing $\times$, here just the subexpression `y`. An expected typing $\times$ indicates that the types of the surrounding program, from which the expected typing is determined, are already in conflict. This simple method works well when there is just a single type error slice. When there are several type errors, we can potentially obtain many slices using the following algorithm:

1. Each node of the type debugging information tree has
   - a location l of the programming construct of the node,
   - a set of locations L_i from which the principal typing was, determined
   - a set of locations L_e from which the expected typing was determined.
2. We remove elements in L_i and L_e that are not needed in opposite directions, obtaining smaller sets $L_i' \subseteq L_i$ and $L_e' \subseteq L_e$.
3. If $\{l\} \cup L_i' \cup L_e'$ is not subset of a type error slice that we already have, it is a new type error slice.

4.3 Improved Interactive Type Error Debugging

An interactive type debugger asks the user for information to determine the source of a type error. The PTT was originally defined to support algorithmic debugging of type errors. Here is a short example session, with the user's input underlined, demonstrating algorithmic debugging using the PTT of Figure 1:

```
1. Is your intended type of f [string] -> string -> [string]?
> No
...
6. Is your intended type of ^ string -> string -> string?
> No
Type error debugger locates the source of a type error: ^
Against intentions, its type is string -> string -> string
```

The source of the type error is located after six questions.

The questions of algorithmic debugging are based on a walk through the PTT. The sesssion starts at a node that has a principal typing $\times$, but all its child nodes have a principal typing unequal $\times$. Here the session starts at the root of the PTT.

Using the type error informatin tree, we start at a type error node instead. Because leaves of the tree are often sources of type errors and their typings are easier to understand, we start at a leaf type error node. So in our example we might start with the node for `^` and thus successful finish algorithmic debugging after one question.

Adding type annotations (signatures) to a program adds information to the expected typings. More of them become $\times$, which excludes the nodes from being asked about in an algorithm debugging session. If in our example we add that `f` should have type $[int] \to int \to [int]$, then `^` is the only leaf type error node of the type debugging information tree and hence the algorithmic debugging session has to start with it.

5 Evaluation

We evaluate our prototype using fifteen programs, each of which has one type error. Most of them are from the functional programming courses in Ochanomizu University. Some examples are from the online demonstration of Skalpel [13]. Three of them include multiple type conflicts. The OCaml compiler's error message locates the cause of a type error correctly for Test 5, 7, 9 and 11.

We rank error messages by giving higher priority to leaves than inner nodes, and higher priorty to large location sets.

How do expected typings reduce the search space of type debugging? Figure 7 shows the number of lines of each program and its number of expected typings. The sum of the expected typings are one third of the sum of the program sizes. Basically large programs have more locations that do not contribute to type errors; therefore expected typing will be more effective in large programs.

Are user's type annotation useful? The Figure 8 shows the number of expected typings of leaves and inner nodes, respectively. The blue line and green line show the expected typings without user's annotations. The other two lines are with user's annotations. From this figure we see that user's annotations reduce the search space of type debugging. This reduction could not be achieved without expected type inference.

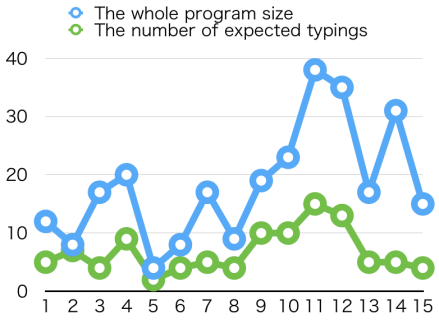


Fig. 7. The program sizes and the number of expected typings

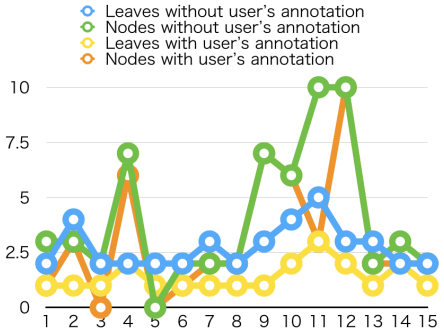


Fig. 8. The number of expected typings (leaves and inner nodes)

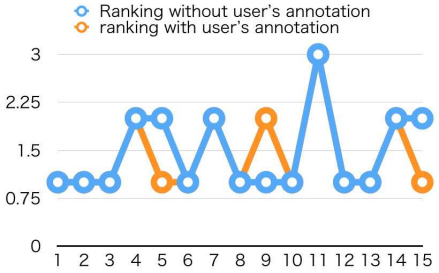


Fig. 9. The numbers of question to anwer (without and with user's annotation)

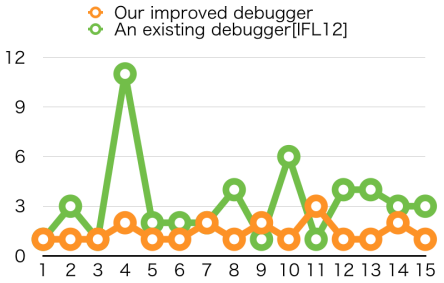


Fig. 10. Comparison with the existing debugger

How many questions does algorithmic debugging ask? Figure 9 shows the numbers of questions until we locate the type error source. Because the numbers are small, our strategy looks effective for type debugging. There are few expected typings; therefore there is no significant difference.

How much does our system improve interactive debugging? In Figure 10 we compare our debugger (adding a type annotation for the whole program) with an existing implementation [12] (for convenience we call this the IFLdebugger). In most cases our debugger can locate the source of a type error faster than the IFLdebugger. In some cases the IFLdebugger locates faster; however, this depends on the bias, from the bottom of the tree. Especially in the cases that the source of a type error is far from the type conflicted part (starting point of debugging) our method is effective (Test 4 and Test 10).

6 Related Work

New Type Inference Algorithms. Many researchers designed new type inference algorithms which are better than Milner's algorithm $\mathcal{W}$ for type error

debugging. For example, Lee and Yi [6] presented algorithm $\mathcal{M}$ which finds type conflicts earlier than algorithms $\mathcal{W}$. Like $\mathcal{W}$, algorithm $\mathcal{M}$ is biased: when there is a type conflict between two subexpressions in a program, it always blames the subexpression on the right. Both algorithms stop at the first type conflict that they find. In contrast, Neubauer and Thiemann [7], and Chen and Erwig [1] define type inference algorithms that succeed for any program. They extend the type system in different ways to allow an expression to have multiple types. Otherwise ill-typed expressions are typed with the new 'multi-types'. As part of presenting type debugging information to the user, Chen and Erwig formalise the notion of expected type.

Interactive Type Error Debugging. Chitil [2] emphasises that the cause of a type error depends on the intentions of the programmer. Hence type error debugging should be an interactive process involving the programmer. He shows how to apply algorithmic debugging [9] to type error debugging: the programmer has to answer a series of questions by the debugger. Chitil defines the PTT as foundation for algorithmic debugging. Tsushima and Asai [11] implement a type debugger for OCaml based on Chitil's idea.

Type Inference as Constraint Solving Problem. Every programming language construct puts a constraint on the type of itself and its direct subexpressions. So a program can first be translated into a set of constraints which are solved in a separate phase. A minimal unsatisfiable subset of constraints explains a type error. If each constraint is associated with the program locations that gave rise to it, then a minimal unsatisfiable subset of constraints defines a slice of the program that explains the type error to the user. Heeren, Hage and Swierstra [4] annotate a syntax tree of the program with type constraints such that different algorithms can solve these constraints in different orders. The Chameleon system [10] fully implemented the type-inference-as-constraint-solving approach for an expressive constraint language that supports many extensions of the Hindley-Milner type system, especially Haskell's classes. Besides presenting slices, Chameleon also provides interactive type error debugging. Concurrently Haack and Wells [3] applied the same idea to the Hindley-Milner type system. They later developed their method into the tool Skalpel for type error slicing of ML programs [13].

Reusing the Compiler's Type Inference as Black Box. Instead of inventing a new type inference algorithm, several researchers developed methods for reusing the existing type inference implementation of a compiler for type error debugging. The central motivation is that implementing type inference for a real programming language is a major investment already made; besides, the type systems of some languages such as Haskell continuously evolve. Schilling [8] developed a type error slicer for a subset of Haskell that produces program slices similar to Skalpel. Tsushima and Asai [12] built an interactive debugger for OCaml that provides the user experience of Chitil's approach (cf. Section 4.3).

Comparison. To produce the PTT of a program, our implementation uses the method of Tsushima and Asai, reusing the existing compiler's type inference. Hence, even though we have to add expected typings, unlike many other methods we do not require a reimplementation of type inference. Our framework supports several type debugging approaches with different user experiences, including the program slices of constraint solving methods and interactive type error debugging. Although constraints support formalising type systems and type debugging, they are not directly suitable for being shown to the user.

7 Conclusion

In this paper we argue that expected typings, which provide type information from the context of an expression, are useful for type error debugging. We describe a common framework for several type error debugging approaches. Our prototype supports the following claims

- Expected typings reduce the search space of type debugging.
- Propagating user's type annotations additionally reduces the search space of type debugging.

Our framework allows for a synergy of previously independent methods. than with existing type debuggers.

When a program contains many type errors, our expected typings are often not informative enough. We believe that this is because we use a coarse typing $\times$ to express any kind of type conflict. In the future we want to explore how we can extend our framework with a more fine-grained language of typings, where the language of types is extended by a construct $\times$ and instead of a typing $\times$ we have a typing like $\{\mathtt{x} : \times, \mathtt{y} : \beta\} \vdash \kappa$.

References

1. Chen, S., and Erwig, M. "Counter-factual typing for debugging type errors," POPL'14, pp. 583–594.
2. Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," ICFP'01, pp. 193–204 (2001).
3. Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," Science of Computer Programming - Special issue ESOP'03, Volume 50, Issues 1-3, pp. 189–224 (2004).
4. Heeren, B., J. Hage, and S. Swierstra. "Constraint based type inferencing in Helium," Immediate Applications of Constraint Programming (ACP), pp. 57–78 (2003).
5. Ishii, Y., and K. Asai. "Report on a User Test and Extension of a Type Debugger for Novice Programmers," Post conference proceedings of International Workshop on Trends in Functional Programming in Education., pp. 1–18 (2014).
6. Lee, O., K. Yi. "Proofs about a Folklore let-polymorphic Type Inference Algorithm," ACM Transactions on Programming Languages and Systems, pp. 707–723 (1998).
7. Neubauer, M., and Thiemann, P. "Discriminative sum types locate the source of type errors," ICFP'03, pp. 15–26 (2003)
8. Schilling, T. "Constraint Free Type Error Slicing," TFP'11, pp. 1–16 (2012).
9. Shapiro, E. Y. Algorithmic Program Debugging, MIT Press (1983).
10. Stuckey, P. J., M. Sulzmann, J. Wazny, "Interactive type debugging in Haskell," Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell'03), pp. 72–83 (2003).
11. Tsushima, K., and K. Asai. "Report on an OCaml type debugger," ML Workshop, 3 pages, (2011).
12. Tsushima, K., and Asai, K. "An Embedded Type Debugger," IFL'12, Springer, LNCS 8241, pp. 190–206 (2013).
13. Rahli, V., Wells, J., Pirie, J. and Kamareddine, F. "Skalpel," Journal of Symbolic Computation, Volume 80, Issue P1, pp. 164–208 (2017).