Support Tool and Strategy for Type Error Correction with Polymorphic Types

Hiroshi MARUYAMA1, Masaki MATSUYAMA11 and Keijiro ARAKI

Department of Computer Science and Communication Engineering
Kyushu University
Hakozaki, Higashi-ku, Fukuoka 812 Japan
E-mail: araki@csce.kyushu-u.ac.jp

Abstract

The notion of types is very important in programs. Type errors and correcting them are important and common issues in programming efforts, in general. In recent programming languages with polymorphic types, especially in strongly typed functional programming languages with polymorphic types, type inference facilities and higher order functions, the notion of types can be made full play. But in these languages, type error corrections become difficult mainly because of the polymorphic types. For efficient software development in these languages, a computer should support such tasks of type error correction. Therefore we focus on these languages, i.e., functional programming languages whose type checkers infer types in programs including polymorphic types and higher order functions' types from the context before execution of programs. And then, we examine new effective approaches to support type error corrections in these languages and two approaches to support them, following a general framework for debugging. We regard these two approaches as case studies to develop systematic debugging processes based on the general framework for debugging.

1 Introduction

The notion of types [1] is very important in programs because it improves program readability, reliability and run-time performance. Program annotated with type information could become more understandable. Type checker can detect type errors before run-time, so execution errors are reduced. Compiler provided type information can yield efficient object code in which no objects carry their types at run-time.

A type error occurs because of a type inconsistency in a program. Of course, when there are type errors in programs, programmers have to correct them. It is not a trivial task in several languages to correct type errors because there are some cases in which the type errors become difficult to understand. The following program is written in the functional programming language Miranda [2]. In this example, there is no type error in the first three lines, but a type error occurs at the fourth line.

\[
\text{twice x a = x (x a)} \quad x a = a \\
\text{f1 = twice x twice x} \\
\text{f2 x = twice x twice x}
\]

For efficient software development, a computer should support such tasks of type error correction. Therefore, in this paper, we examine new effective approaches to support type error corrections. Especially in recent programming languages [3], there are polymorphic types [1] [4] which correspond to more than one types and provide flexible and powerful typing. In strongly typed languages, type errors cannot occur in run-time because types are checked strictly before execution, so these languages are effective. But type checking systems in these languages have to know what such polymorphic types will be instantiated from the context in programs. To this end, type inference facilities [3] are used. More over, type inference facilities play a practical role: programmers need not to specify types for all objects that can have types, and thus they can write programs conveniently.

In functional programming languages, the notion of types is very clear and it is easy to treat types mathematically as their results. So many modern strongly typed functional programming languages, e.g., ML [3] and Miranda, have developed type systems with such polymorphic types and type inference facilities. Besides, there are higher order functions [3] which take functions as their arguments or return functions as their results. Thus especially in these languages, the notion of types can be in full play.

In the above example, a variable \(x\) in the second line has a polymorphic type \(* \to *\) (where * is a type variable and these two *s must be the same type when instantiated). In the third line, the previous \(x\) is instantiated in two ways, and these two \(x\)s are inferred that they have instantiated polymorphic types \((\alpha \to \alpha \to \alpha \to \alpha)\to (\alpha \to \alpha)\to \alpha \to \alpha\) and \(\alpha \to \alpha\) respectively, then the type of \(f1\) is inferred.

---

1Currently, NEC Software Kyushu, Ltd., Fukuoka, Japan

11Currently, Nippon Koukan K., Kawasaki, Japan
as \( \alpha \rightarrow \alpha \) (where \( \alpha \) is a type variable and the above 12 occurrences of \( \alpha \) must be the same type). In the fourth line, two \( x \)'s in the right hand side are the same object as an argument given to \( \text{f} \), so these \( x \)'s have the same type. But these \( x \)'s are inferred that they have polymorphic types \(( \beta \rightarrow \gamma ) \rightarrow \beta \rightarrow \beta \rightarrow \beta \rightarrow \beta \) respectively (where \( \beta \) is also a type variable), so the type error occurs.

As in the above example, it may become very difficult to correct type errors in those functional programming languages. The types in programs tend to become hard to grasp mainly because the polymorphic types are instantiated with wide variety and higher order functions are of complicated (possibly polymorphic) types. Besides, type inference facilities would make the programmers' requirements about types ambiguous because they tend not to write down the type declarations into their programs. Without sufficient knowledge about a type checking activity, a programmer hardly knows how types in a program are given actually and where in the program to modify in order to correct a type error. Therefore, we think that the three issues are the main reason why type error correction in these languages becomes difficult, i.e.,

1. advanced type system with polymorphism and higher order functions,
2. ambiguous requirement for types in programs with such an advanced type system, and
3. lack of knowledge about type checking mechanism.

In this paper, we focus on strongly typed functional programming languages with polymorphic types, type inference facilities and higher order functions, i.e., functional programming languages whose type checkers infer types in programs including polymorphic types and higher order functions' types from the context before execution of programs. And then, we examine type error corrections and approaches to support those tasks, following a general framework for debugging [5].

We discuss type error corrections rigorously in section 2. And we show two type error correction support approaches: tool-based approach in section 3 and strategic approach in section 4. Finally in section 5, we mention concluding remarks.

2 Type checking and type errors

Type checking is to find inconsistency among inferred types. When type checking system infers that some two types must be the same but cannot unify those two types, the system raises a type error and stops type checking. Therefore a type error is regarded identical with an inconsistent pair of inferred types.

It is clear that at least one of such types must be wrong. Though a programmer has to change the wrong type in order to correct the type error, he/she cannot change the type directly. He/she can change only the program to be typed.

We selected a functional language Exp[4] in our study of the type error correction because it is strongly typed and it has type polymorphism and sufficient features for a functional programming language, i.e.,

higher order functions, a simple grammar and a type inference facility.

The expression of the language Exp is generated by the following grammar:

\[
\begin{align*}
  e &::= x \mid (e') \mid \text{if } e \text{ then } e' \text{ else } e'' \mid \lambda x.e \mid \text{fix } x.e \mid \text{let } z = e \text{ in } e',
\end{align*}
\]

where \( z \) ranges over identifiers, \( (e') \) means application, \( \text{fix } x.e \) stands for the least fixed point of \( \lambda x.e \), and the last clause binds \( z \) to the value of \( e \) throughout \( e' \).

Milner[4] presented a compile-time type checking algorithm \( W \) for Exp. If \( W \) finds no type error in a program at compile-time, the program never raises type error at run-time. \( W \) is defined in the form of a functional language:

\[
W(\overline{p}, f) = (T, f).
\]

\( W \) takes a typed prefix expression \( \overline{p} \) and a complete program \( f \), and returns a type substitution \( T \) and a typed program \( f \). A typed prefix expression is a finite sequence whose members are \( \lambda x. \), \( \text{fix } x. \), \( \text{let } x. \), and \( x \). Based on the unification algorithm of Robinson[6], \( W \) infers the type of each expression so as to hold the following conditions:

\[
\begin{align*}
  &((\overline{x} e')_\rho)_\sigma \quad \rho = \sigma \rightarrow \tau, \\
  &((\overline{x} e')_\rho \text{ then } e''_\rho \text{ else } e''')_\tau \quad \rho = \sigma_0 \text{ and } \sigma = \tau \rightarrow \tau', \\
  &((\lambda x_\rho e_\rho)_\sigma)_\tau \quad \tau = \rho \rightarrow \sigma, \\
  &((\text{fix } x_\rho e_\rho)_\sigma \text{ in } e'')_\tau \quad \rho = \sigma \rightarrow \tau, \\
  &((\text{let } x = e_\rho)_\sigma) \quad \sigma = \tau,
\end{align*}
\]

where \( \rho, \sigma, \tau \ldots \) are type variables and \( e_\rho \) means the expression typed \( \rho \). In other words, \( W \) assigns a type variable to each expression, and unifies type variables according to the above conditions. If and only if any of these unifications fails, \( W \) will fail.

In the general framework for debugging, an 'error' means the difference between a program and its specification—the difference between the behavior requested by the specification and the behavior performed by the program—and a 'bug' means the cause of the error. Figure 2.1 shows the process model of the general framework for debugging[5]. Debugging is the process of locating and correcting errors in a program in which errors have been detected. In locating errors and grasping their causes, programmers develop hypotheses about the errors and their causes, and verify or refute these hypotheses by examining the program. In correcting the errors, programmer again develop hypotheses about how to modify the program and verify or refute them.

In this paper, we focus on 'type errors' found by the type checking and 'bug' is the cause of a type error. Our goal is to get a program without type errors. As in the general framework for debugging process, type error correction process consists of two phases: the bug locating phase and the fixing phase. Two approaches described in the following sections mainly support the bug locating phase.
3 Tool-based approach

One of the approaches supporting type error correction is to provide software tools for programmers to correct type errors effectively [7] like general debugging tools in correcting execution errors. We followed traditional debugging tools and implemented a prototype of support tool for correcting type errors. According to the general framework for debugging, the prototype is regarded as a type error debugging tool which provides available information for programmers to develop, select, verify and modify their hypotheses about causes of type errors and its corrections.

We must pay attention to a main difference between type error correction and execution error correction: the former depends on how a program is typed in its type checking system, whereas the latter depends on how the program behaves in its execution system. We must also pay attention to a difficulty of type error correction which we mentioned before in section 1.

We selected the functional programming language Exp [4] in our investigation of the type error correction. On the other hand, we selected the functional programming language Miranda in implementing our support tool.

Our implementation consists of two phases for the sake of efficiency and simplicity: a type checking phase and an analysis phase. The type checking phase takes an Exp program file and checks types of the program and makes a type checking information file. The analysis phase uses the type checking information file and analyzes it according to a programmer's intention and supplies him/her a useful information for type error correction. The analysis phase provides the following functions, and we explain these four functions of our tool.

1. Type checking reporter.
2. Static program analyzer.
3. Type checking tracer.
4. Type checking slicer.

In the function 1, for a given Exp program, the result of checking types is reported. When the type checking succeeds, a programmer gets the types of the program. On the other hand, when it fails in type checking, he/she gets a type error report which mentions not only what types fails in their unification but also from where these types were and why the unification was needed. From this type error report, he/she...
If the function 3, type checking activity is traced. To know how an Exp program is typed, we traced the type checking program, and thus especially here, we paid attention to the difference between type error correction and execution error correction. Therefore this function provides information with which a programmer can easily find which part of his/her program is typed. Reading this information, he/she can find in detail how our program is typed by the type checking system. Figure 3.5 shows a trace data for the same program in Figure 3.2. In this trace data, we can know as follows why the variable \( n \) is typed \( \text{Nat} \) instead of \( \text{Bool} \). We wrote \( \text{True} \) of the reference number 5, so \( \text{is1} \) of the reference number 2 is typed \( \text{Boo1} \). The variable \( n \) of the reference number 20 is typed \( \text{t7} \), and function application of the reference number 22 requires that the first part of \( \text{is1}'s \) type and the type of the variable \( n \) are equal. This unification constructs a type substitution which substitutes \( \text{Boo1} \) for \( \text{t7} \). As a result, the variable \( n \) is typed \( \text{Boo1} \) and the type error causes at the reference number 29.

In the function 4, using a program slicing [8][9] which is applicable to debugging [10]. A type checking slice is extracted. The type checking slice is a part of the program (or the entire program) in which a typing to designated part of the program is equivalent to the typing as in case of original program. Generally it is defined that slices are executable, but they have not to be executable when used in debugging [11]. Similarly, a slice found by this function is not necessarily typed by itself. Instead we intended to reduce the size of slices. We applied not a static program slicing [8] but a dynamic one [9] to this type checking slicing. It is because we have to execute the type checking actually in order to know values of type substitutions constructed in the type checking activity. Using this function, we can narrow the area in which the cause of type error exists, so correcting type error becomes easier. Figure 3.3 shows a slice of the same program in Figure 3.2. Of course, the slice includes the cause of the type error of the reference number 5.

From the viewpoint of the general framework for debugging, the type checking reporter supports programmers to develop hypotheses about the causes of type error and their correction, the static program analyzer and the type checking tracer support to verify the hypotheses and the type checking slice supports its selection.
We found that our tool is useful to correct type errors and to gain better comprehension of types in programs. But in the first three functions, there are mainly two defects: provided information is too much and a fair knowledge about the type checking activity is expected to programmers. On the other hand, even they have little knowledge about it, we found the type checking slicer effective. We do not provide a method to locate the causes of type error and to correct them.

Therefore, the strategic approach is required.

4 Strategic approach

To support type error correction, especially to support the bug locating phase, we will provide a strategy which suggests the candidates for bugs using the type error report and the history of type checking.

It is reasonable to express the history of the type checking in the form of a tree because the algorithm W is applied to an expression and its subexpressions recursively. We will call this tree W-tree and define as follows:

Given a typed prefix expression $F_0$ and a complete program $f_0$,

1. define a root $w_0(F_0, f_0)$,
2. on each node $w_i(F_i, f_i)$,
   (a) if $f_i$ is an identifier, $w_i$ is a leaf.
   (b) otherwise, for each subexpression $f_k$ of $f_i$, define $w_k(F_k, f_k)$ as a subtree of $f_i$. If a unification failed during type checking, the tree expands no further.

Figure 3.5 Type Checking Tracer Output for the Same Program in Figure 3.2

Figure 3.6 Type Checking Slicer Output for the Same Program in Figure 3.2
is defined and its left brother node returns mutative. Here we assume eq is a predefined function following example:

\[
\text{eq}(\text{true}, \text{false}) \Rightarrow \text{true} \\
\text{eq}(\text{false}, \text{true}) \Rightarrow \text{false}
\]

For example, Figure 4.1 shows a W-tree for the following example:

\[
\lambda f \lambda g . \lambda x . (\text{if } (\text{eq } (f \, g), x) \Rightarrow (g \, f) \, x) \text{ then True else False})
\]

which takes two functions f and g and a variable x as its parameters, and examines whether f and g are commutative. Here we assume eq is a predefined function typed

\[
\text{eq} :: \text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Bool})
\]

On the node \(w_{13}\), W tried to unify \(\beta\) and \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\) and failed because \(\beta\) is included in \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\). W cannot keep on the type checking any more, and the system raise a type error. Therefore we call this node 'type error node.' We call such a type that make unifications impossible 'inconsistent factor,' and we call all the terms including the inconsistent factor 'inconsistent terms.' In this case, the inconsistent factor is \(\beta\), and the inconsistent terms are \(\beta\), \(\beta \rightarrow (\text{f} \rightarrow \text{Nat})\), and \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\). Moreover, we can know that \(\beta\) is the type of \(g\), and that \(\beta \rightarrow (\text{f} \rightarrow \text{Nat})\) is the type of \(f\), and that \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\) is the type of \(g\).

To correct a type error, we must change at least one of these inconsistent terms into another one that enable the unification to succeed. It seems natural to consider the expression of the node which generated the inconsistent terms as candidates for bug. Now we will provide a strategy to suggest candidate nodes from a type error node on a W-tree:

For each inconsistent term,

**Initialization:** set a pivot on the type error node.

**Iteration:** If the inconsistent term or a more general term of it is included in

1. \(\beta\) of the parent node, then move the pivot onto it.
2. \(T\) of the left brother node, then move the pivot onto it.
3. \(f\) of a child node, then move the pivot onto it.
4. otherwise, stop iteration and the pivot node is a candidate node.

Here \(\beta \rightarrow \delta\) is more general than \(\beta \rightarrow (\text{f} \rightarrow \text{Nat})\) because \(\beta \rightarrow \delta\) can be instantiated to \(\beta \rightarrow (\text{f} \rightarrow \text{Nat})\) with a substitution \(\delta := \text{f} \rightarrow \text{Nat}\), and a more general term of some inconsistent term must include the inconsistent factor.

We will illustrate the strategy with the above example. First, selecting \(\beta\) as an inconsistent term, the pivot moves from the type error node \(w_{13}\) to \(w_3\), \(w_4\), \(w_5\), and \(w_2\) and then stops. Secondly, selecting \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) as an inconsistent term, the pivot moves from \(w_{13}\) to \(w_{12}\), \(w_3\), \(w_7\), and \(w_9\). Thirdly, selecting \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\), the pivot remans on \(w_{13}\). Thus, the candidate nodes are \(w_2\), \(w_3\), and \(w_{13}\).

Although the programmer may be able to select the most likely candidate with his/her knowledge about the program or its type specification. Here we take into account each candidate in turn.

First, we easily understand that it has no meaning to select \(w_3\) because the type variable \(\beta\) was assigned to \(g\) in \(w_2\) and has not been changed.

Secondly, we select \(w_4\), where \(\beta \rightarrow \delta\) is more general than \(\beta \rightarrow (\text{f} \rightarrow \text{Nat})\) and it is the type of \(f\). \(\beta \rightarrow \delta\) includes the inconsistent factor \(\beta\) because of applying \(g\) to \(f\). The programmer can change the type of \(f\) by changing the expression applied to \(f\). For example, if \(he/she\) changes \((g \, f)\, x)\) into \((g \, (f \, x))\), the type of new \(f\) would expected to be \(\text{f} \rightarrow \text{f}\).

Thirdly, we select \(w_{13}\), which is the only node where \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) \(\Rightarrow\) \(\sigma\) appears. This term includes \((\beta \rightarrow (\text{f} \rightarrow \text{Nat}))\) which is the type of \(f\) because of applying \(g\) to \(f\). For example, if the programmer can changes \((g \, f)\, x)\) into \((g \, (f \, x))\), the type of new \(f\) would expected to be \(\text{f} \rightarrow \text{f}\).

After these corrections, the new types of \(f\), \(g\), and \(x\) would be \(\text{f} \rightarrow \text{f}\), \(\text{f} \rightarrow \text{f}\), \(\text{f} \rightarrow \text{f}\), and \(\text{f} \rightarrow \text{f}\), and the system would raise no type error.
To summarize this approach, we show a model of the type error correction process in Figure 4.2 based on the general framework for debugging. When the type checking system raises a type error, we can get both a set of inconsistent terms and the history of type checking as a type error report. According to the strategy, we can make a set of candidates. Then the programmer selects the most likely candidate with his/her knowledge about the program, and verify the selected candidate. If he/she decides that the candidate is the bug, the bug locating phase is over. Otherwise, he/she should select another candidate. At present, this approach does not directly support either selection or verification of a candidate. They require the knowledge about the type specification of the program but we assume that the programmers generally have only ambiguous knowledge about type specification as discussed in section 1.

### 5 Concluding remarks

Type errors and correcting them are important and common issues in programming efforts, in general. We focus on strongly typed functional programming languages with polymorphic types, type inference facilities and higher order functions. And then, we examined type error corrections and two approaches to support them, following the general framework for debugging. We regard these two approaches as instances of systematic debugging processes based on the general framework for debugging.

The tool-based approach provides available information for programmers when they locate and correct the causes of type errors, and in this approach we suggested a new idea of the type checking slicing. The strategic approach provides a strategy to locate them. To integrate these two approaches, we will get a more helpful support tool for type error correction.

We should improve the functions and user-interface of our tool, and we will implement our support tool built into an actual programming system such as ML or Miranda, and show practical effectiveness of our approaches.

### Acknowledgements

The authors thank Prof. Jingde Cheng and Prof. Zengo Furukawa for their valuable comments and Mr. Tomohiro Oda for his help in making the final version of this paper.

### References


