IPSJ Transactions on Programming Vol. 3 No. 4 43–56 (Sep. 2010)

Regular Paper

Improving Error Messages in Type System

Cynthia Kustanto[†1] and Yukiyoshi Kameyama[†1]

We propose a type inference algorithm for a polymorphic type system which provides improved error messages. While the standard type inference algorithms often produce unnecessarily long or incomplete error messages, our algorithm provides relevant and complete information for type errors. It is relevant in the sense that all the program points and types in the output of our algorithm contribute to some type error, and is complete in the sense that, for each type error, our algorithm identifies not only two conflicting types, but also all types which conflict with each other. The latter property is particularly useful for debugging programs with lists or case branches. Our algorithm keeps track of the set of program points that are relevant to each type. To achieve completeness, we introduce a new type variable which represents a conflict among two or more incompatible types, and extend the unification algorithm to handle the special type variable appropriately. Finally, we argue that our algorithm is more efficient than those in the literature when there are more than two conflicting types in the given expression.

1. Introduction

Type error messages should be correct and informative to help programmers fix the error. However, when one designs a type inference algorithm, the main goal is usually its correctness and efficiency, and error reporting is just a subsidiary goal. As a result, error messages are often hard to understand, unnecessarily long, or spotting program fragments which are far away from the actual error sources [1)].

For instance, Algorithm $\mathcal{W}$ [2)–4)] for the Hindley/Milner let-polymorphic type system works in the bottom-up manner and performs unification whenever it infers the type of application terms, and therefore it signifies a type error at a program point far from the actual conflicting subterms. Moreover, it stops at the first error, hence it has the so called left-to-right bias. Numerous approaches have been proposed to improve the quality of type errors.

Haack and Wells presented an algorithm to compute minimal program-slices for type errors in implicitly typed higher-order languages with let-polymorphism [5)]. A program-slice (slice, in short) is a collection of program points which are relevant to a type error. Their algorithm satisfies two desirable properties, minimality and completeness. The slices obtained by their algorithm are minimal in the sense that no program points that are irrelevant to a type error are included in the slices. Their algorithm produces a complete set of slices in the sense that all type errors in the source expression can be explained in some slice. The problem with their algorithm is that it may produce unnecessarily long output when there are more than two conflicting types, which we will argue below.

A type error is often seen as a conflict between the types of two subexpressions, for example, a conflict between the argument type of a function and the type of its actual argument. However, there are cases when more than two subexpressions and types are involved in a conflict, such as the error in the elements of a list as described below [⋆1]:

$$\underline{[1; [1]; \mathit{true}]}$$

This list contains three elements each of which has a different type. Therefore, it has three type errors if we regard a type error as a conflict between two types. Clearly it is unnatural to produce three messages for this expression, and we think the conflict in this example should be reported as a single error message with all three subexpressions pinpointed as the error locations.

A similar phenomenon occurs in distinct subtrees of a source expression, for instance:

$$\underline{(\lambda x\,.\,x{+}4)}\ \underline{(\text{if}\ \mathit{true}\ \text{then}\ \mathit{true}\ \text{else}\ 1)}$$

This example has a type conflict between *true* and 1 because the if-expression requires its two arms be of the same type. Moreover, it should be the same type as the type of x which ought to be integer. Hence it has another type

†1 Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

⋆1 We use PCF-like syntax here. The precise syntax will be introduced in Section 3.

conflict between *true* and x. As the type inference system does not know the programmer's intention, it should output an error message which contains all three subexpressions (and the corresponding types) as "error locations".

Finding all error locations may be even more complex if a variable x has more than one occurrence, as in the following example:

$$\underline{(\lambda x\,.\,x + (x\ 1))}\ \underline{(\text{if}\ \mathit{true}\ \underline{\text{then}\ \mathit{true}\ \text{else}\ 1)}}$$

As the third occurrence of the variable x is used in application $(x\ 1)$, its type is the function type integer $\rightarrow$ integer. Hence there is yet another conflict of types, and in total, the expression above contains the conflict among four locations and three types. Apparently, we would get too many error messages if they are generated in a pairwise manner.

In this paper we propose a novel type inference algorithm which reports all type errors in the input program. More precisely, it identifies, for each type error, all the program points (locations) that contribute to the type error without reporting irrelevant locations.

We construct our algorithm as a constraint type inference algorithm with the addition of ψ type variables and the set of labels.

A ψ type variable is associated with a list of types, which represents an equation that involves an arbitrary number of types, instead of exactly two. When this equation is not satisfied, then there is a type error among the elements of the list of types.

A label uniquely determines a program point, and in our algorithm each type has the set of labels relevant to the type. Thus, when a conflict is found, we can recover all the relevant program points using the recorded set of labels.

We argue our algorithm improves Haack and Wells' algorithm in conceptual simplicity and expected efficiency. Their algorithm first generates error slices based on pairwise conflicts of types, then tries to find and merge overlapping slices, and finally, minimizes the output slices. Clearly, their algorithm may produce too big intermediate data. Suppose the given expression is a list of 10 elements whose types are mutually distinct. Their algorithm first generates 45 slices, since each slice corresponds to a conflict between two types, and then tries to merge and minimize these 45 slices, and the result will be only 1 slice with 10 conflicting types. On the contrary, our algorithm directly calculates this single slice when it solves the constraints by unification, and there is no need to merge or minimize slices. It is then natural to expect that our algorithm is simpler and runs more efficiently for such cases.

The rest of this paper is organized as follows. Section 2 describes the problems addressed in this paper and informally explains how the algorithm works, and Section 3 gives our type inference algorithm to solve the problems. We show some theoretical properties of the algorithm in Section 4. Section 5 gives an overview of previous work dedicated to improving type error messages and compares our approach with others. Section 6 concludes the paper and describes our plans for future work.

2. Identifying Error Locations

2.1 Detecting Multiple Type Errors

Many type inference systems stop at the first error they find. In the case where there are multiple errors, it is inconvenient for the programmer to keep re-executing the system to get explanations for each error. Information regarding the existence of other errors would be useful when fixing an error. However, there are drawbacks in detecting multiple errors. Even the manifesto for good type error reporting did not include multiple-error reporting because such goal comes with the risk of generating a cascade of bogus error messages [6]. In such cases, the number of error locations that need to be fixed is usually less than the number of errors reported. This issue is one of our motivations to identify all the relevant error locations in error messages. The errors reported in the error messages should be the actual errors that occurred in the program.

To find all the type errors in a program, the type inference algorithm must not fail at the first conflict found. Type inference basically stops when it fails to unify a constraint, so we have to delay the unification until we record all the constraints. To achieve this we use the constraint-based typing described by Pierce [7]. It differs from Algorithm $\mathcal{W}$ in the typing rules, where constraints are not *checked* immediately but *recorded* for later consideration.

However, the type inference of a term $let\ x = e_1\ in\ e_2$ strictly requires unification to be applied to the output of the inference of e_1 before inferring e_2. The

principal type scheme of e_1 is needed for instantiation each time an occurrence of x is found in e_2. Delay of unification will not yield a correct result.

The slicer proposed by Haack and Wells [5] uses Damas' type inference system which is used with Damas' algorithm $\mathcal{T}$ [8]. It differs from the Hindley/Milner system in the rule for let-expressions. The let-expressions rule of Damas is described below:

$$\frac{(S \neq \emptyset) \text{ and } \forall \tau_1 \in S \;.\; \Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \wedge S] \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

In the algorithm, if τ_1 and C_1 are, respectively, the type and the constraint set produced by the inference of e_1, each time an occurrence of x in e_2 is encountered, fresh variants of τ_1 and C_1 are generated. This let-expressions rule also copies C_1 for every instance of x in e_2.

Unification is used to calculate solutions to constraint sets [9]. The unification algorithm always terminates and fails when given a non-unifiable constraint set as input, and returns a principal unifier otherwise. In order to find all the errors in a program, we modify the standard unification algorithm so that it always terminates and returns a set of substitutions and an error set. If the error set is not empty, then the constraint set given as the input is not unifiable. If the error set is empty, then the substitution returned by the algorithm is a principal unifier.

In our algorithm, an error set is a set of lists of types. Each list corresponds to an error, and it contains two or more types which contribute to the error. We discuss the approach to record multiple conflicting types in the next subsection.

2.2 Sample Output of Our Algorithm

The type reconstruction algorithm in constraint-based typing [7] calculates a set of constraints which is a set of equations $\{S_i = T_i^{i \in 1..n}\}$. A constraint is inconsistent if there is a conflict between the types on the left side and the right side of the equation. Either or both of the types might be substitution instances or might also occur in other constraints. Thus, there are relations among the elements of a constraint set which could have contributed to the error. Similarly, there might be other inconsistencies that have (partly or completely) the same error locations. Our algorithm records these relations and related conflicts so that the error messages can present all the contributing error locations.

In order to explain our proposal, we use several examples from our simple prototype system.

```
# typecheck "((lambda x . [x;true;false;true]) (if true then 1 else 2))";;
((lambda x . [x;true;false;true]) (if _ then 1 else 2))
Type conflict: bool, int
```

The error in the above example is that there is a conflict between integer and boolean. In the function, the variable x is an element of a list in which elements are of type boolean. However, the function receives an argument of type integer. More detailed explanations:

- The list requires all its elements to be of the same type, as in (type of x = type of true = type of false = type of true).
- If-expression requires the argument of then-clause and else-clause to be of the same type, as in (type of 1 = type of 2).
- The function takes the output of if-expression as an argument, which create a constraint among the elements of the list and the type of the if-expression. Therefore, a conflict between type integer and type boolean arises.

Program points that are unrelated to the type error are omitted. To fix the type error, programmer doesn't need to modify any of the omitted points.

```
# typecheck "(lambda x . [1;x;true])";;
(___[1;_;true])
Type conflict: int, bool
```

In the above expression, the variable x does not contribute to the conflict although it is also an element of the list. Any change to it would not help fixing the error.

```
# typecheck "[[[1]];[[true]];[[]];[[false]];[[]]]";;
[[[1]];[[true]];_;[[false]];_]
Type conflict: int, bool
```

In the above example, the empty list also does not contribute to the error, as its element is not assigned with any type to be able to contribute anything to the conflict. However, the empty list itself (no matter what the elements are) can cause conflict as in the example below.

```
# typecheck "(lambda x . [1;[];x;true])";;
(___[1;[];_;true])
Type conflict: list of t3, int, bool
```

This implies that rather than just the relationship among subexpressions, it is the contribution of a subexpression that becomes the factor as to whether any types or locations are included in the error messages.

If there is any inconsistency in the inner list, it is considered as a different error.

```
# typecheck "[1;[34;false;95];true]";;
[1;[_;_;_];true]
Type conflict: list of t2, int, bool

[_;[34;false;95];_]
Type conflict: int, bool
```

In the case of the lambda expression and function application, the function type $\tau \to \tau$ is involved in the constraints, so there is a need to further examine the points that contribute to the error. It can be the body of the function, the output of the function, the parameter of the function, the function itself, the application, or any combination of the mentioned parts. The next example shows that only the function contributes to the conflict as the arrow type itself conflicts with boolean and integer.

```
# typecheck "[true;(lambda x . (x + 1));3]";;
[true;(lambda _ . ___);3]
Type conflict: (int -> int), bool, int
```

In the next example, the argument is a function, while inside the body of the function there is inconsistency among the types of the occurrences of x.

```
# typecheck "((lambda x . (x + (if x then 1 else 2)))
         (lambda y . (if true then y else (y + 1))))";;
((lambda x . (x + (if x then _ else _))) (lambda _ . ________))
Type conflict: (int -> int), bool, int
```

The next example illustrates a type error in the case of let-polymorphism. There are five occurrences of x; however, only two of them collide in the type error as x is a let-bound variable.

```
# typecheck "(let x = (lambda w . w) in
        (if (x true) then (x 1) else ((x 2) + (x x))))";;
(let x = (lambda w . w) in (__ ___ __(_ _ + (x x))))
Type conflict between: (t1.t11 -> t1.t11), int
```

The last example has two type errors of which error locations are overlapping. They are shown in different slices as there are two separate conflicts. The integer 1 conflicts with *true*, while 2 conflicts with *false*.

```
# typecheck "(if true then (lambda f . ((f 1) 2))
              else (lambda g . ((g true) false)))";;

(if _ then (lambda f . ((f 1) _)) else (lambda g . ((g true) _)))
Type conflict between: int, bool

(if _ then (lambda f . ((f _) 2)) else (lambda g . ((g _) false)))
Type conflict between: int, bool
```

3. Our Algorithm

3.1 Syntax and Notations

To describe our approach concretely, we use the language shown in **Fig. 1**. The syntax is similar to PCF with boolean and integer as the representatives of the base types. List is also included. $tp(\tau, C)$ is used for polymorphic types so that the algorithm can create new instances of the type and the constraint.

The type ψ is actually a type variable. It differs from ordinary type variables in that ψ forms a constraint with a list of types.

Figure 2 introduces some notations used in the algorithm. Labels $l_0, l_1, \ldots, l_n$ denote locations in a term. For a term e, e^{l_0} is a labeled term whose root location is labeled with l_0. An example of a fully labeled term is given as follows:

$$(\lambda x^{l_1} . (x^{l_3} + 1^{l_4})^{l_2})^{l_0}$$

For a type τ and a set of labels L, τ^L denotes a labeled type whose root position is labeled with L. Arrow types and list types have subtypes, each of which may also have labels, such as $(\tau_1^{L_1} \to \tau_2^{L_2})^{L_0}$ and $(tlist(\tau_1^{L_1})^{L_0})$. We often omit to write the label set L if L is empty. We use labeled types to keep track of the subterms which are relevant to the derivation of that type. For instance, the type of the term $(\lambda x^{l_1} . (x^{l_3} + 1^{l_4})^{l_2})^{l_0}$ is $(\psi_1^{[l_0;l_1]} \to int^{[l_0;l_2]})^{[l_0]}$. We notice that the arithmetic expression labeled with l_2 contributes to deriving the return type of the function as indicated by $int^{[l_0;l_2]}$.

Ψ denotes a set of multiple-type constraints, which are equations between the type variable ψ and a list of types. An example of Ψ is:

Term	e	::=	i | true | false	constants
		|	x	variable
		|	$(\lambda x\ .\ e)$	lambda abstraction
		|	$(e_1\ e_2)$	application
		|	(if e_1 then e_2 else e_3)	conditional
		|	$(e_1 + e_2)$	addition
		|	(let $x = e_1$ in e_2)	let-expression
		|	$[e_1;e_2;...;e_n]$	list
	i	::=	0 | 1 | -1 | ...	integer constants
Type	τ	::=	tb	base type
		|	$\tau \to \tau$	arrow type
		|	α	type variable
		|	ψ	multiple-type variable
		|	$tp(\tau, C)$	Damas' type scheme
		|	$tlist(\tau)$	list type
	tb	::=	int | bool	

Fig. 1 Syntax of the language.

Label	l		
Set of Labels	L		
Labeled term	e^l		
Labeled type	τ^L		
Constraint	c	::=	$(\tau_1^{L_1} = \tau_2^{L_2})$
Set of constraints	C		
Context	Γ	::=	List of $(x : \tau)$
Substitution	s	::=	$(\tau_1 \mapsto \tau_2^L)$
Set of substitutions	S		
Multiple-type constraints	c_m	::=	$(\psi =$ List of $\tau^L)$
Set of multiple-type constraints	Ψ	::=	List of c_m
Set of errors	err	::=	List of List of τ^L

Fig. 2 Definitions and notations.

$$
\begin{array}{lcl}
[\ \psi_1 & = & [bool^{L_1}]; \\
\ \psi_2 & = & [(int^{L_1} \to int^{L_2})^{L_3}]; \\
\ \psi_3 & = & [int^{L_1}; bool^{L_2}] \qquad]
\end{array}
$$

```
typecheck(Γ, e) =
    let (C, τ1) = recon(Γ, e) in
    let (S, err) = unify(C) in
    if err = [ ] then let τ2 = S ∘ (τ1) in Success(τ2)
    else Error(err)
```

Fig. 3 The main function of our algorithm.

An error is represented by a list of labeled types, and *err* is a list of errors. The labeled types in an error represent a type conflict among two or more types. For instance:

$$
\begin{array}{lll}
[\ [\tau_{1.1}^{L_{1.1}}; \tau_{1.2}^{L_{1.2}}; ...; \tau_{1.n_1}^{L_{1.n_1}}]; & & 1^{st} \text{ error} \\
\ [\tau_{2.1}^{L_{2.1}}; \tau_{2.2}^{L_{2.2}}; ...; \tau_{2.n_2}^{L_{2.n_2}}]; & & 2^{nd} \text{ error} \\
\ ... & & \\
\ [\tau_{m.1}^{L_{m.1}}; \tau_{m.2}^{L_{m.2}}; ...; \tau_{m.n_m}^{L_{m.n_m}}] \quad] & & \mathrm{m}^{th} \text{ error}
\end{array}
$$

Each error in *err* contains all the conflicting types of which labels represent all the locations contributing to the corresponding error.

3.2 Main Function

Figure 3 describes the main function of our algorithm. The inputs of the function *typecheck* are a context Γ and a term e. It returns either Success(τ) for a principal type τ of e, or Error(err) for errors.

First, the function *recon*, which is a constraint-based typing algorithm, receives a context Γ and a term e and returns a constraint set C and a type τ. Constraint set C will be passed as the input for the unification process. Function *unify* checks the consistency of constraint set C and returns a substitution set S and an error set *err* as the output.

If the error set *err* is an empty set, then the type inference succeeds. We can get the principal type of the input term by applying substitution S to τ_1. If the error set is not empty, then there are type error(s). The error set *err* contains a list of errors, each of which contains a list of conflicting types. The types are attached with the record of labels of the expression contributing to the errors, which can be used to pinpoint the possibly erroneous locations.

To get the error locations, we only have to get the labels of the types recorded in the error set. For example, given an erroneous expression as follows:

$(\lambda x\,.\,(\text{let } w\ =\ (x+1) \text{ in } (\text{if } w \text{ then } [1; true] \text{ else } [4])))$

Each subexpression is annotated with a unique label. Let the above expression be labeled as follows:

$(\lambda x^{l_2}\,.\,(\text{let } w^{l_4}\ =\ (x^{l_6}+1^{l_7})^{l_5} \text{ in } (\text{if } w^{l_9} \text{ then } [1^{l_{11}}; true^{l_{12}}]^{l_{10}} \text{ else } [4^{l_{14}}]^{l_{13}})^{l_8})^{l_3})^{l_1}$

The result of the algorithm for this labeled expression is an Error state with the following error set:

```
[   [int^[l3;l4;l5;l9]; bool^[l8]];
    [int^[l10;l11;l13;l14]; bool^[l10;l12]]   ]
```

According to the error set, there are two type errors in the given expression. The first error is a type conflict between integer and boolean. We retrieve the labels recorded in all the types in the list, $[l_3; l_4; l_5; l_8; l_9]$, to get the error locations as shown below:

```
(___(let w = (_ + _) in (if w then _ else _)))
```

The same goes for the second error. The second error is a type conflict between integer and boolean, with $[l_{10}; l_{11}; l_{12}; l_{13}; l_{14}]$ as the error labels. We use the labels to seek the error locations which are illustrated below:

```
(___(_______(___[1;true]_[4])))
```

The following subsections describe the functions *recon* and *unify* in more details.

3.3 Labeled Type Inference

Function *recon* generates a set of constraints. It receives a term e and a context Γ to compute the type of the term, τ, and a set of constraints C.

recon follows the standard algorithm, but keeps track of labels which are responsible for the derivation of each type. For its definition in **Fig. 4**, we assume that the given term is fully labeled, and the labels are mutually distinct. The label of each subexpression is recorded in its inferred type. The labels are passed along the inference and also when constraints are built. These labels are used for keeping track of the subexpressions contributed to the derivation of types, so they are very crucial for finding the location of an error.

In the let-expression rule, the *tp* type is introduced to handle polymorphism. It carries a type τ^L and a constraint set C. Every time an occurrence of x is found in the input term, the variable rule generates a fresh variant of τ^L and C, which are described in the algorithm as τ_1^L and C_1. We will use an example

recon: Context × Term → Set of constraints × Type

$$
\begin{array}{lcl}
\text{recon}(\Gamma,\ i^{l_0}) & = & ([\,],\ \text{int}^{[l_0]}) \\
\text{recon}(\Gamma,\ true^{l_0}) & = & ([\,],\ bool^{[l_0]}) \\
\text{recon}(\Gamma,\ false^{l_0}) & = & ([\,],\ bool^{[l_0]}) \\
\text{recon}(\Gamma,\ x^{l_0}) & = & \text{if } (x : \psi_0) \in \Gamma \text{ then } ([\,],\ \psi_0^{[l_0]}) \\
 & & \text{else if } (x : tp(\tau^{L_1}, C)) \in \Gamma \text{ then} \\
 & & \text{let } (C_1,\ \tau_1) \text{ be fresh variants of } (C, \tau) \text{ in} \\
 & & (C_1,\ \tau_1^{L_1 \cup [l_0]}) \\
\text{recon}(\Gamma,\ (\lambda x^{l_1}.e)^{l_0}) & = & \text{let } (C_2,\ \tau_2^{L_2}) = \text{recon}(\Gamma \cup [x : \psi_0],\ e^{l_2}) \text{ in} \\
 & & (C_2,\ (\psi_0^{[l_0;l_1]} \to \tau_2^{(L_2 \cup [l_0])})^{[l_0]}) \\
\text{recon}(\Gamma,\ (e_1\ e_2)^{l_0}) & = & \text{let } (C_1,\ \tau_1^{L_1}) = \text{recon}(\Gamma,\ e_1^{l_1}) \text{ in} \\
 & & \text{let } (C_2,\ \tau_2^{L_2}) = \text{recon}(\Gamma,\ e_2^{l_2}) \text{ in} \\
 & & (C_1 \cup C_2 \cup [\tau_1^{L_1} = (\tau_2^{L_2 \cup [l_0]} \to \alpha_0^{[l_0]})^{[l_0]}],\ \alpha_0^{[l_0]}) \\
\text{recon}(\Gamma,\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3)^{l_0}) & = & \text{let } (C_1,\ \tau_1^{L_1}) = \text{recon}(\Gamma,\ e_1^{l_1}) \text{ in} \\
 & & \text{let } (C_2,\ \tau_2^{L_2}) = \text{recon}(\Gamma,\ e_2^{l_2}) \text{ in} \\
 & & \text{let } (C_3,\ \tau_3^{L_3}) = \text{recon}(\Gamma,\ e_3^{l_3}) \text{ in} \\
 & & \text{let } C_0 = [\tau_1^{L_1} = bool^{[l_0]};\ \psi_0^{[l_0]} = \tau_2^{L_2};\ \psi_0^{[l_0]} = \tau_3^{L_3}] \text{ in} \\
 & & (C_1 \cup C_2 \cup C_3 \cup C_0,\ \psi_0^{[l_0]}) \\
\text{recon}(\Gamma,\ (e_1 + e_2)^{l_0}) & = & \text{let } (C_1,\ \tau_1^{L_1}) = \text{recon}(\Gamma,\ e_1^{l_1}) \text{ in} \\
 & & \text{let } (C_2,\ \tau_2^{L_2}) = \text{recon}(\Gamma,\ e_2^{l_2}) \text{ in} \\
 & & \text{let } C_0 = [\tau_1^{L_1} = int^{[l_0]};\ \tau_2^{L_2} = int^{[l_0]}] \text{ in} \\
 & & (C_1 \cup C_2 \cup C_0,\ \text{int}^{[l_0]}) \\
\text{recon}(\Gamma,\ (\text{let } x^{l_1} = e_2 \text{ in } e_3)^{l_0}) & = & \text{let } (C_2,\ \tau_2^{L_2}) = \text{recon}(\Gamma,\ e_2^{l_2}) \text{ in} \\
 & & \text{let } \Gamma_2 = \Gamma \cup [x : tp(\tau_2^{L_2 \cup [l_0;l_1]}, C_2)] \\
 & & \text{let } (C_3,\ \tau_3^{L_3}) = \text{recon}(\Gamma_2,\ e_3^{l_3}) \text{ in} \\
 & & (C_2 \cup C_3,\ \tau_3^{L_3 \cup [l_0]}) \\
\text{recon}(\Gamma,\ [e_1;e_2;..;e_n]^{l_0}) & = & \text{for } i = 1 \text{ to } n\text{: let } (C_i,\ \tau_i^{L_i}) = \text{recon}(\Gamma,\ e_i^{l_i}) \text{ in} \\
 & & \text{let } C' = [\tau_1^{L_1} = \psi_0^{[l_0]};..;\tau_n^{L_n} = \psi_0^{[l_0]}] \text{ in} \\
 & & \text{let } C_0 = C_1 \cup ... \cup C_n \cup C' \text{ in} \\
 & & (C_0,\ tlist(\psi_0)^{[l_0]})
\end{array}
$$

Fig. 4 Type inference algorithm.

where all labels of the types are omitted. If type τ is $\alpha_1 \to \alpha_1$, then its fresh variant would be $\alpha_{1.1} \to \alpha_{1.1}$, $\alpha_{1.2} \to \alpha_{1.2}$, $\alpha_{1.3} \to \alpha_{1.3}$, and so on, where $\alpha_{1.j}$ are new type variables. A fresh variant of the constraint C' is actually a copy

of C with fresh variants of its types. For example, if C is $[\alpha_1 = int]$, then its variant will be $[\alpha_{1.1} = int], [\alpha_{1.2} = int]$, and so on.

ψ-types are introduced for the cases of if-expression, list, and abstraction. ψ-types are used for handling variable instances and elements of list because all the instances of the same variable and all elements of a list have to be of the same type, and there can be more than two instances and elements, respectively. Therefore, we need to make constraints connecting all the related types and locations. ψ is also used in if-expression because the type of then-clause, the type of else-clause, and the type of the whole if-expression have to be the same.

We use the previous example to illustrate this algorithm. If we apply *recon* to the following expression:

$(\lambda x^{l_2}\,.\,(\mathrm{let}\,w^{l_4} = (x^{l_6} + 1^{l_7})^{l_5}\,\mathrm{in}\,(\mathrm{if}\,w^{l_9}\,\mathrm{then}\,[1^{l_{11}}; true^{l_{12}}]^{l_{10}}\,\mathrm{else}\,[4^{l_{14}}]^{l_{13}})^{l_8})^{l_3})^{l_1}$

then *recon* generates outputs $\psi_1^{[l_1;l_2]} \rightarrow \psi_2^{[l_1;l_3;l_8]}$ as τ, and the constraint set C as follows:

$$
\begin{array}{[lll}
\psi_1^{[l_6]} = int^{[l_5]}; & int^{[l_7]} = int^{[l_5]}; & \psi_{1.1}^{[l_5]} = int^{[l_5]}; \\
int^{[l_7]} = int^{[l_5]}; & \psi_3^{[l_{10}]} = int^{[l_{11}]}; & \psi_3^{[l_{10}]} = bool^{[l_{12}]}; \\
\psi_4^{[l_{13}]} = int^{[l_{14}]}; & int^{[l_3;l_4;l_5;l_9]} = bool^{[l_{12}]}; & tlist(\psi_3)^{[l_{10}]} = \psi_2^{[l_8]}; \\
tlist(\psi_4)^{[l_{13}]} = \psi_2^{[l_8]} \;] & &
\end{array}
$$

3.4 Unification

If the labels are ignored and the conditions involving ψ-types are removed, our unification algorithm is just a standard unification algorithm modified to suit the syntax of our language. Our unification function terminates when all the constraints have been processed. Type error occurs when the unification of a constraint fails, and the conflicting types will be stored in the set of errors *err* for displaying error messages later.

The main function of the unification process is the function *unify* as described in **Fig. 5**. *unify* receives a constraint set C as the input and produces a substitution set and an error set as the output. If C is unifiable, the error set should be an empty list.

Two functions are executed inside *unify*. Function u puts away all the constraints involved with ψ-types in the list Ψ while unifying the other constraints. The list of multiple-type constraints Ψ is checked by the function *solvePsi*, which produces the final substitution set and error set.

```
unify: Set of constraints → Set of substitutions × Set of errors
unify(C)  =  let (S_temp, Ψ, err_temp) = u(C,[ ],[ ],[ ],[ ]) in
             solvePsi(S_temp, Ψ, err_temp)
```

Fig. 5 Unification algorithm.

3.5 Solving Equations Between a Pair of Types

Function u is defined in **Fig. 6**. u is a recursive function receiving 5 parameters, the first of which is a constraint set C which is being solved. The second parameter is a constraint set C_ψ for quarantining constraints in the form of $(\psi_1^{L_1} = \psi_2^{L_2})$. This parameter will be explained later. The third input is the substitution set S, which holds the record of all the generated substitutions. The fourth parameter is the list of multiple-type constraint Ψ, while the fifth one is the error set *err* which holds the record of failed unification. The three latter parameters will be returned as the output of the function. In the function, *addS* is used for applying a substitution to the elements of the parameters. We use OCaml syntax in the figures, for example, $h :: r$ is a list with head h and rest r.

Besides the treatment of labels, the difference of u and the standard unification algorithm is the special treatment for ψ-types. In the function u, all constraints in the form of $(\psi_1^{L_1} = \psi_2^{L_2})$ are quarantined in a different list C_ψ as the second parameter of the function u. For each constraint in the form of equation between a ψ-type and a type variable $(\psi^{L_1} = \alpha^{L_2})$, a substitution $(\alpha \mapsto \psi^{L_1 \cup L_2})$ is applied to all the parameters and then recorded in S. When a constraint in the form of equation between a ψ-type and any other type $(\psi^{L_1} = \tau^{L_2})$ is encountered, the τ^{L_2} is added to Ψ as one of the types assigned to ψ^{L_1}.

When all the constraints in C has been checked by u, we check the contents of the parameter C_ψ (see the first two clauses in Fig. 6). If C_ψ is empty, then the function terminates and returns the substitution set, the Ψ set, and the error set. If C_ψ is not empty, then the function *upsi* is called to solve C_ψ.

The function *upsi* defined in **Fig. 7** handles the constraints in the form $(\psi_1^{L_1} = \psi_2^{L_2})$ by building substitutions and arranging the list of types in Ψ. *upsi* receives 5 parameters which are exactly the same as the parameters received by u. It returns the constraint set, the substitution set, Ψ, and the error set as the output.

u: Set of constraints × Set of constraints × Set of substitutions
× Set of multiple-type constraints × Set of errors
→ Set of substitutions × Set of multiple-type constraints × Set of errors

u([ ],[ ], S, Ψ, err) = (S, Ψ, err)
u([ ], C_ψ, S, Ψ, err) = let $(C', S', \Psi', err') =$ upsi([ ], C_ψ, S, Ψ, err) in
u(C',[ ], S', Ψ', err')
u(($tb_1^{L_1} = tb_2^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) =
if $tb_1 \neq tb_2$ then u($rest$, C_ψ, S, Ψ, $err \cup [tb_1^{L_1}; tb_2^{L_2}]$)
else u($rest$, C_ψ, S, Ψ, err)
u(($tb^{L_1} = tlist(\tau_2)^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) = u($rest$, C_ψ, S, Ψ, $err \cup [tb^{L_1}; tlist(\tau_2)^{L_2}]$)
u(($\alpha^{L_1} = \tau^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) = let $(ocheck, olabel) =$ occurcheck(α, τ^{L_2}) in
if $ocheck$ then u($rest$, C_ψ, S, Ψ, $err \cup [\alpha^{L_1 \cup olabel}; \tau^{L_2}]$)
else u(addS($(\alpha \mapsto \tau^{L_1 \cup L_2})$, $rest$, C_ψ, S, Ψ, err))
u(($\psi^{L_1} = \alpha^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) =
u(addS($(\alpha \mapsto \psi^{L_1 \cup L_2})$, $rest$, C_ψ, S, Ψ, err))
u(($\psi_1^{L_1} = \psi_2^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) =
if $(\psi_1^{L_3} = \psi_2^{L_4}) \in C_\psi$ || $(\psi_2^{L_4} = \psi_1^{L_3}) \in C_\psi$
then let $C'_\psi = C_\psi - [(\psi_1^{L_3} = \psi_2^{L_4}); (\psi_2^{L_4} = \psi_1^{L_3})] in$
u($rest$, $C'_\psi \cup [\psi_1^{L_1 \cup L_3} = \psi_2^{L_2 \cup L_4}]$, S, Ψ', err)
else u($rest$, $C_\psi \cup [\psi_1^{L_1} = \psi_2^{L_2}]$, S, Ψ', err)
u(($\psi^{L_1} = \tau^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) = let $(\Psi', C') =$ addPsi($\psi, \tau^{L_1 \cup L_2}, \Psi$) in
u($C' \cup rest$, C_ψ, S, Ψ', err)
u((($\tau_1^{L_1} \to \tau_2^{L_2}$)^{L_5} = ($\tau_3^{L_3} \to \tau_4^{L_4}$)^{L_6}) :: $rest$, C_ψ, S, Ψ, err) =
u($[\tau_1^{L_1} = \tau_3^{L_3}; \tau_2^{L_2} = \tau_4^{L_4}] \cup rest$, C_ψ, S, Ψ, err)
u((($\tau_1^{L_1} \to \tau_2^{L_2}$)^{L_4} = $tlist(\tau_3)^{L_3}$) :: $rest$, C_ψ, S, Ψ, err) =
u($rest$, C_ψ, S, Ψ, $err \cup [(\tau_1^{L_1} \to \tau_2^{L_2})^{L_4}; tlist(\tau_3)^{L_3}]$)
u((($\tau_1^{L_1} \to \tau_2^{L_2}$)^{L_4} = tb^{L_3}) :: $rest$, C_ψ, S, Ψ, err) =
u($rest$, C_ψ, S, Ψ, $err \cup [(\tau_1^{L_1} \to \tau_2^{L_2})^{L_4}; tb^{L_3}]$)
u(($tlist(\tau_1)^{L_1}$ = $tlist(\tau_2)^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) = u(($\tau_1 = \tau_2$) :: $rest$, C_ψ, S, Ψ, err)
(* for any other pair of types, exchange left-hand side and right-hand side *)
u(($\tau_1^{L_1} = \tau_2^{L_2}$) :: $rest$, C_ψ, S, Ψ, err) = u(($\tau_2^{L_2} = \tau_1^{L_1}$) :: $rest$, C_ψ, S, Ψ, err)

Fig. 6 Unification function for checking consistency of equations between a pair of types.

If the ψ_1 and ψ_2 in the constraint $(\psi_1^{L_1} = \psi_2^{L_2})$ are the same, then we only need to record the labels in $L_1 \cup L_2$ in all occurrences of ψ_1 in the parameters. If ψ_1 is inside Ψ and has non-empty list of types, then the labels are also recorded in those types. If the ψ_1 and ψ_2 in the constraint $(\psi_1^{L_1} = \psi_2^{L_2})$ are different and both already exist in Ψ, then we have to merge the list of types of ψ_1 and the list

upsi: Set of constraints × Set of constraints × Set of substitutions
× Set of multiple-type constraints × Set of errors → Set of constraints
× Set of substitutions × Set of multiple-type constraints × Set of errors

upsi(C,[ ],S, Ψ, err) = (C, S, Ψ, err)
(* If the ψ-type in the left hand side is the same as the right hand side *)
upsi(C, ($\psi_1^{L_1} = \psi_1^{L_2}$) :: $rest$, S, Ψ, err) =
let $L' = L_1 \cup L_2$ in
let (C', $rest'$, S', Ψ', err') = addS(($\psi_1 \mapsto \psi_1^{L'}$), C, $rest$, S, Ψ, err) in
if $(\psi_1, [\tau_1^{L_1}; ...; \tau_n^{L_n}]) \in \Psi'$
then upsi(C', $rest'$, S', $(\psi_1, [\tau_1^{L_1 \cup L'}; ...; \tau_n^{L_n \cup L'}]) :: rest\Psi'$, err')
else upsi(C', $rest'$, S', $\Psi' \cup [(\psi_1, [\,])]$, err')
(* If the ψ-type in the left hand side is different from the right hand side *)
upsi(C, ($\psi_1^{L_1} = \psi_2^{L_2}$) :: $rest$, S, Ψ, err) =
if $(\psi_1, [\tau_{a1}^{L_{a1}}; ...; \tau_{an}^{L_{an}}]) \in \Psi$ and $(\psi_2, [\tau_{b1}^{L_{b1}}; ...; \tau_{bn}^{L_{bn}}]) \in \Psi_a$
then let $\Psi' =$ (for $i = 1$ to n: addPsi($\psi_1, \tau_{bi}^{L_{bi} \cup L'}, \Psi$)) in
let $\Psi'' =$ delete ψ_2 from Ψ' in
upsi(addS(($\psi_2 \mapsto \psi_1^{L'}$), C, $rest$, S, Ψ'', err))
else if $(\psi_1, [\tau_{a1}^{L_{a1}}; ...; \tau_{an}^{L_{an}}]) \in \Psi$
upsi(addS(($\psi_2 \mapsto \psi_1^{L'}$), C, $rest$, S, Ψ, err))
else upsi(addS(($\psi_1 \mapsto \psi_2^{L'}$), C, $rest$, S, Ψ, err))

Fig. 7 Unification function for handling equations between a pair of ψ types.

of types of ψ_2, and then substitute one of the ψ-type to the other one. $L_1 \cup L_2$ must be recorded as well.

3.6 Adding Types to Multiple-type Constraints

Figure 8 describes the function *addPsi* for adding a type to the set of multiple-type constraints Ψ. The function is applied in the function u and also in the function *upsi* when merging lists of types. *addPsi* has three parameters; a type ψ, a type τ, and the list Ψ. The function returns the Ψ list which has been updated by adding τ to the list of types assigned to ψ.

To make it easier to check the consistency of multiple-type constraints afterwards, there can only be one occurrence of the same type in a Ψ element. Therefore, when adding a new type to Ψ as an element, we must check whether the same type already exists or not. For example, when adding a new type int^{L_1} to $(\psi_1, [int^{L_2}])$, we only have to pass the labels so that it becomes $(\psi_1, [int^{L_2 \cup L_1}])$.

addPsi: Multiple-type variable × Labeled Type × Set of multiple-type constraints
→ Set of multiple-type constraints × Constraints

addPsi(ψ, τ, Ψ) =
  let (Ψ', *ltype*) = if $(\psi, [\tau_1^{L_1}; ...; \tau_n^{L_n}]) \in \Psi$
    then ($\Psi - (\psi, [\tau_1^{L_1}; ...; \tau_n^{L_n}]), [\tau_1^{L_1}; ...; \tau_n^{L_n}]$) else ($\Psi$, [ ])
  in match τ with
      $tb_1^{L_1}$: if $tb_2^{L_2} \in ltype$ then ($\Psi' \cup (\psi, ltype - [tb_2^{L_2}] \cup [tb_2^{L_1 \cup L_2}])$, [ ])
        else ($\Psi' \cup (\psi, ltype \cup [tb^{L_1}])$, [ ])
    | $(\tau_1^{L_1} \to \tau_2^{L_2})^{L_{12}}$: if $(\psi_3 \to \psi_4)^{L_{34}} \in ltype$
        then ($\Psi' \cup (\psi, ltype - [(\psi_3 \to \psi_4)^{L_{34}}] \cup [(\psi_3 \to \psi_4)^{L_{34} \cup L_{12}}])$,
          $[\tau_1^{L_1} = \psi_3; \tau_2^{L_2} = \psi_4]$)
        else let ψ_3, ψ_4 = new variables in
          ($\Psi' \cup (\psi, ltype \cup (\psi_3 \to \psi_4)^{L_{12}})$, $[\tau_1^{L_1} = \psi_3; \tau_2^{L_2} = \psi_4]$)
    | $tlist(\tau_1)^{L_1}$: if $tlist(\psi_2)^{L_2} \in ltype$
        then($\Psi' \cup (\psi, ltype - [tlist(\psi_2)^{L_2}] \cup [tlist(\psi_2)^{L_2 \cup L_1}])$, $[\tau_1^{L_1} = \psi_2]$)
        else let ψ_2 = new variable in ($\Psi' \cup (\psi, ltype \cup [tlist(\psi_2)^{L_1}])$, $[\tau_1^{L_1} = \psi_2]$)

Fig. 8 Function for adding a type to the set of multiple-type constraints.

If we add $bool^{L_1}$ to this Ψ, it becomes (ψ_1, [int^{L_2}; $bool^{L_1}$]). With this, we can see that a multiple-type constraint is consistent when its list is singleton.

However, we need to do more for the arrow type and the list type. The arrow type and the list type contain subtypes so we have to make new ψ-types and new constraints for every type included. For example, when a type $(\tau_1^{L_1} \to \tau_2^{L_2})^{L_{12}}$ is added to an element of Ψ, we need to build new constraints $(\psi_{new1} = \tau_1^{L_1})$ and $(\psi_{new2} = \tau_2^{L_2})$. The type listed in the Ψ would be the $(\psi_{new1} \to \psi_{new2})^{L_{12}}$, so that when there are other arrow types forming constraints with the corresponding ψ, the constraints between the nested types are created and their relationships are recorded.

3.7 Solving Multiple-type Constraints

The final consistency check of the multiple-type constraint set Ψ is done by the algorithm solvePsi described in **Fig. 9**. *solvePsi* takes the Ψ set and the temporary set of substitutions and set of errors as the input, and checks the consistency of the Ψ set.

Because elements of the list of types in Ψ have unique values, we only have to

solvePsi: Set of multiple-type constraints × Set of substitutions × Set of errors
→ Set of substitutions × Set of errors

solvePsi(S, [ ], err) = (S, err)
solvePsi(S, (ψ_0, [ ]):: $restpsi$, err) =
  let (eC, eC_ψ, S', $restpsi'$, err') = addS(($\psi_0 \mapsto \alpha_0$), [ ], [ ], S, $restpsi$, err) in
  solvePsi($restpsi'$, S', err')
solvePsi(S, (ψ_0, $\tau_1^{L_1}$::[ ]):: $restpsi$, err) =
  let ($ocheck, olabel$) = occurcheck($\psi_0, \tau_1^{L_1}$) in
  if $ocheck$ then solvePsi($restpsi$, S, $err \cup [\psi_0^{olabel}; \tau_1^{L_1}]$)
  else let (eC, eC_ψ, S', $restpsi'$, err') = addS(($\psi_0 \mapsto \tau_1^{L_1}$), [ ], [ ], S, $restpsi$, err) in
    solvePsi($restpsi'$, S', err')
solvePsi(S, (ψ_0, $[\tau_1^{L_1}; ...; \tau_n^{L_n}]$) :: $restpsi$, err) =
  let $olabel$ = [ ] in
  for $i = 1$ to n :
    let ($ocheck_i$, $olabel_i$) = occurcheck($\psi_0, \tau_i^{L_i}$) in
    if $ocheck_i$ then add $olabel_i$ to $olabel$
  in
  if $olabel \neq$ [ ] then
    solvePsi($restpsi$, S, $err \cup [\psi_0^{olabel}; \tau_1^{L_1}; ...; \tau_n^{L_n}]$)
  else solvePsi($restpsi$, S, $err \cup [\tau_1^{L_1}; ...; \tau_n^{L_n}]$)

Fig. 9 Unification function for checking consistency of the constraints assigned to ψ types.

check the number of elements in the list of types assigned to each ψ. If there is only one type in it, for example $(\psi_1, [int^{L_1}]) :: rest$, then the constraint is consistent and the substitution $(\psi_1 \mapsto int^{L_1})$ is applied. Otherwise, there is an error and the corresponding list of types is added to the error set. In the case of error involving arrow types and/or list types, occurs check is applied as well.

After all the elements of Ψ have been checked, the function returns the final set of substitutions S and the set of errors err.

3.8 Other Necessary Functions

Figure 10 describes the function for occurs check. The function *occurcheck* takes a type τ_1 and a labeled type $\tau_2^{L_2}$ as the input. τ_1 can be either a type variable or a ψ-type. If any occurrence of τ_1 is found inside $\tau_2^{L_2}$, then the function returns *true* and the labels of τ_1 that occured in τ_2. If τ_1 is not found in $\tau_2^{L_2}$, then the function returns $false$ and an empty set.

occurcheck: Type × Labeled type → bool × Set of labels

$$
\begin{array}{lcl}
\mathrm{occurcheck}(\tau_1, (\tau_2^{L_2} \to \tau_3^{L_3})^{L_1}) & = & \text{let } (o_1, lo_1) = \mathrm{occurcheck}(\tau_1, \tau_2^{L_2}) \text{ in} \\
 & & \text{let } (o_2, lo_2) = \mathrm{occurcheck}(\tau_1, \tau_3^{L_3}) \text{ in} \\
 & & (o_1 || o_2,\ lo_1 \cup lo_2) \\
\mathrm{occurcheck}(\tau_1, tb_2^{L_2}) & = & (\mathit{false}, [\,]) \\
\mathrm{occurcheck}(\tau_1, \alpha_2^{L_2}) & = & \text{if } \tau_1 = \alpha_2 \text{ then } (\mathit{true}, L_2) \text{ else } (\mathit{false}, [\,]) \\
\mathrm{occurcheck}(\tau_1, \psi_2^{L_2}) & = & \text{if } \tau_1 = \psi_2 \text{ then } (\mathit{true}, L_2) \text{ else } (\mathit{false}, [\,]) \\
\mathrm{occurcheck}(\tau_1, \mathit{tlist}(\tau_2)^{L_2}) & = & \text{let } (o, lo) = \mathrm{occurcheck}(\tau_1, \tau_2) \text{ in} \\
 & & \text{if } o \text{ then } (\mathit{true}, lo \cup L_2) \text{ else } (\mathit{false}, [\,])
\end{array}
$$

Fig. 10 Occurs check.

In our unification process, labels are passed every time a substitution is applied. For example, if a substitution $(\alpha_1 \mapsto \mathit{int}^{L_2})$ is applied to a type $\alpha_1^{L_3}$, then the substitution instance will be $\mathit{int}^{L_3 \cup L_2}$.

The application of the function *addS* in the figures represents the substitution process. The first parameter of *addS* is the substitution $(\tau_1 \mapsto \tau_2^{L_2})$, while the five other parameters are the lists to which the substitution is applied. After $(\tau_1 \mapsto \tau_2^{L_2})$ is applied to the five lists, it will be added to the fourth parameter, substitution set S. The function *addS* returns the five lists as the output.

3.9 Example of the Unification Process

To explain the unification process, we will use an example. We consider the following expression:

$(\lambda x\,.\,(\mathrm{let}\ w\ =\ (x+1)\ \mathrm{in}\ (\mathrm{if}\ w\ \mathrm{then}\ [1; \mathit{true}]\ \mathrm{else}\ [4])))$

From the function *recon* we obtain a constraint set C as follows:

$$
\begin{array}{lll}
[\quad \psi_1^{[l_6]} = \mathit{int}^{[l_5]}; & \mathit{int}^{[l_7]} = \mathit{int}^{[l_5]}; & \psi_{1.1}^{[l_5]} = \mathit{int}^{[l_5]}; \\
\quad \mathit{int}^{[l_7]} = \mathit{int}^{[l_5]}; & \psi_3^{[l_{10}]} = \mathit{int}^{[l_{11}]}; & \psi_3^{[l_{10}]} = \mathit{bool}^{[l_{12}]}; \\
\quad \psi_4^{[l_{13}]} = \mathit{int}^{[l_{14}]}; & \mathit{int}^{[l_3;l_4;l_5;l_9]} = \mathit{bool}^{[l_{12}]}; & \mathit{tlist}(\psi_3)^{[l_{10}]} = \psi_2^{[l_8]}; \\
\quad \mathit{tlist}(\psi_4)^{[l_{13}]} = \psi_2^{[l_8]}\ ] & &
\end{array}
$$

The above constraint set is given as the input for the function *unify*. In *unify*, $u(C,[\,],[\,],[\,],[\,])$ is called first, and we get a temporary substitution set S_{temp}, Ψ, and a temporary error set err_{temp}. The content of each set is as follows:

$$
\begin{array}{lcl}
S & : & [\psi_5 \mapsto \psi_4;\ \psi_3 \mapsto \psi_4] \\
\Psi & : & [(\psi_1 = [\mathit{int}^{[l_5;l_6]}]);\ (\psi_{1.3} = [\mathit{int}^{[l_5;l_6]}]);\ (\psi_2 = [\mathit{tlist}(\psi_4)^{[l_8;l_{10};l_{13}]}]); \\
 & & (\psi_4 = [\mathit{int}^{[l_{10};l_{11};l_{13};l_{14}]}; \mathit{bool}^{[l_{10};l_{12}]}])] \\
\mathit{err} & : & [\ [\mathit{int}^{[l_3;l_4;l_5;l_9]}; \mathit{bool}^{[l_8]}]\ ]
\end{array}
$$

These three sets are then passed to the function *solvePsi*. The output of $\mathit{solvePsi}(S, \Psi, \mathit{err})$ is as follows:

$$
\begin{array}{lcl}
S & : & [\psi_5 \mapsto \psi_4;\ \psi_3 \mapsto \psi_4;\ \psi_1 \mapsto \mathit{int}^{[l_5;l_6]}; \\
 & & \psi_{1.3} \mapsto \mathit{int}^{[l_5;l_6]};\ \psi_2 \mapsto \mathit{tlist}(\psi_4)^{[l_8;l_{10};l_{13}]}] \\
\mathit{err} & : & [\ [\mathit{int}^{[l_3;l_4;l_5;l_9]}; \mathit{bool}^{[l_8]}];\ [\mathit{int}^{[l_{10};l_{11};l_{13};l_{14}]}; \mathit{bool}^{[l_{10};l_{12}]}]\ ]
\end{array}
$$

The error set is not empty; therefore, C is not unifiable and there are two type errors.

4. Properties of the Algorithm

In this section, we show several properties of the algorithm in Section 3.

The first theorem states the correctness of our algorithm as a (normal) type inference algorithm.

Theorem 4.1 (Correctness). Given a context Γ and a term e, if e is typable under Γ, the algorithm returns $\mathit{Success}(\tau)$ for a principal type τ of e; otherwise, it returns $\mathit{Error}(\mathit{err})$ for some err.

This theorem can be proved straightforwardly; if we ignore all the labels, and treat ψ-types as normal type variables, our algorithm coincides with the algorithm $\mathcal{W}$, with the rule of let-expressions being modified based on Damas' algorithm $\mathcal{T}$.

The next theorem states that, if the input term is not typable under the given context, our algorithm returns all the type errors in the term. To precisely state the theorem, we need the notion of replacement.

For a labeled term e, its label l, and a term constructor g, we define the replacement $\mathtt{repl}(e; l; g)$ as the term e in which the constructor at the l-position is replaced by g. Its precise definition can be given as follows:

$$
\begin{array}{lcl}
\mathtt{repl}((f(a_1, \ldots, a_n))^l; l; g) & = & g(a_1, \ldots, a_n)^l \\
\mathtt{repl}((f(a_1, \ldots, a_n))^{l_0}; l; g) & = & (f(\mathtt{repl}(a_1; l; g), \ldots, \mathtt{repl}(a_n; l; g)))^{l_0} \quad \text{if } l_0 \neq l
\end{array}
$$

where $f(a_1, \ldots, a_n)$ denotes a generic form of expressions with an n-ary constructor f and arguments $a_1, \cdots, a_n$. We implicitly assume that the arity of

f and that of g are the same. f ranges over the set of constants and functions symbols such as +, if-then-else, let-in, list, and application. For instance, $\texttt{repl}(((1+2)^{l_1}+3)^{l_0}; l_0; g) = (g((1+2)^{l_1}, 3))^{l_0}$, and $\texttt{repl}(((1+2)^{l_1}+3)^{l_0}; l_1; g) = ((g(1,2))^{l_1}+3)^{l_0}$. The definition of replacement is naturally extended to the sets of labels: $\texttt{repl}(e; l_1, \cdots, l_m; g_1, \cdots, g_m)$.

We can now state the next theorem.

Theorem 4.2 (Completeness with respect to error locations). If our algorithm returns $Error(err)$ given a context Γ and a term e, err contains all error locations.

Namely, let L be the error locations in err, m be the number of locations in L, $x_1, \cdots, x_m$ be fresh variables, and $\phi_1, \cdots, \phi_m$ be fresh type variables. Then the term $\texttt{repl}(e; L; x_1, \cdots, x_m)$ is typable under the context $\Gamma \cup \{(x_1 : \phi_1), \cdots, (x_m : \phi_m)\}$.

In the theorem, we use fresh variables x_i as term constructors which have arbitrary arities. These term constructors have the types ϕ_i.

We illustrate it using the example below:

$$(\text{if } (1^{l_2} + 2^{l_3})^{l_1} \text{ then } 3^{l_4} \text{ else } 5^{l_5})^{l_0}$$

Given the expression above and an arbitrary context Γ, our algorithm returns $Error(err)$ with err being $[[int^{[l_1]}; bool^{[l_0]}]]$, which means that there is a type error between integer and boolean. According to the recorded labels, the contributing constructors for this error are those with the label l_0 (the if-expression) and l_1 (the arithmetic function +). Therefore we have two ways to fix the type error as follows:

- If we replace the function symbol at l_0 (i.e., *if-then-else*) by g and add $(g : \phi_1)$ to Γ, we get the term $(g\ (1+2)\ 3\ 5)$, which is typable.
- If we replace the function symbol at l_1 (i.e., $+$) by f and add $(f : \phi_2)$ to Γ, we get the term (*if* $(f\ 1\ 2)$ *then* 3 *else* 5), which is typable.

In both cases, we get typable expressions provided f and g are fresh variables and ϕ_1 and ϕ_2 are fresh types. It is clear that subterms other than the if-expression and the + expression are irrelevant to the type error. Thus, the algorithm finds the *complete* set of error locations.

The theorem can be proven without big problems: suppose a type constraint $\tau^L = \sigma^{L'}$ is generated at the label l before the replacement $\texttt{repl}(e; l; x)$. Then by the replacement, the generated constraint would become $\tau^L = \phi$ for a fresh ϕ. Since ϕ does not appear elsewhere in the constraints, the replacement, in effect, removes away this constraint. It is then a bookkeeping task to check that the remaining constraints are solvable, thus the term after the replacement is typable.

Our algorithm is also expected to return a *minimal* error when the given term contains type errors. All of the subterms whose labels are recorded in err are relevant to the error.

To illustrate this, we use the expression below as an example.

$$(\lambda x^{l_1} . [(1^{l_4} + 1^{l_5})^{l_3}; (x^{l_7} 1^{l_8})^{l_6}; true^{l_9}]^{l_2})^{l_0}$$

Typechecking the above expression will produce an err whose content is $[int^{[l_2;l_3]}; bool^{[l_2;l_9]}]$. Provided that f, g, h are fresh variables and ϕ_1, ϕ_2, ϕ_3 are fresh type variables, we show that all the constructors whose labels are recorded in err contribute to the error, namely:

- The list (labeled l_2) is relevant to the error. If we replace the list by f and add $(f : \phi_1)$ to Γ, then we get $(\lambda x\ .\ (f\ (1+1)\ (x\ 1)\ true))$, which is typable. Thus, the replaced list contributes to the type error.
- The arithmetic function + (labeled l_3) is relevant to the error. If we replace + by g and add $(g : \phi_2)$ to Γ, we get $(\lambda x\ .\ [(g\ 1\ 1);(x\ 1);true])$, which is typable. Thus, the replaced + contributes to the type error.
- The constant *true* (labeled l_9) is relevant to the error. If we replace *true* by h and add $(h : \phi_3)$ to Γ, we get $(\lambda x\ .\ [(1+1);(x\ 1);h])$, which is typable. Thus, the replaced *true* contributes to the type error.

Although it seems possible to formally state the minimality property, we do not do so in this paper, because there is no generally agreeable notion of minimality for type errors if there are multiple errors. We leave the elaboration on the minimality property for future work.

5. Related Work

There are numerous different approaches that have been proposed to improve the quality of type error messages. Heeren gives a summary of the suggested

approaches [10] and presents a classification based on the following categories; order of unification, explanation systems, reparation systems, program slicing, and interactive systems. There are many variations in the form of error messages and the quality comparison of each system remains subjective. Yang and others propose a manifesto for good type error reporting [6]. According to the manifesto, good error reports should meet these criteria: correct, precise, succinct, amechanical, source-based, unbiased, and comprehensive.

Modifying the order of unification is the most straightforward approach to change the behavior of error reporting systems. The top-down Algorithm $\mathcal{M}$ detects type inconsistency earlier than Algorithm $\mathcal{W}$, and there are hybrid algorithms which try to combine the properties of both algorithms which result in different behaviors for different kinds of type errors. However, for any fixed unification order, there are always cases where the error location reported is far from the actual location. This has motivated works aimed at proving better error reporting by explaining how types are inferred. Some approaches provide detailed explanations on how a type is inferred, while some extract only the information that are crucial for explaining the conflict.

An example of the approaches which give error report without providing much textual explanation is the two algorithms $\mathcal{U}_{AE}$ and $\mathcal{IEI}$ [11]. $\mathcal{U}_{AE}$ (unification of assumption environments) is based on Algorithm $\mathcal{W}$. To eliminate the effect of unification order, it treats subexpressions equally by typing each of them independently. The consistency of the assumption environments returned by the typing of the subexpressions is checked at the root of subexpressions by unification. Because it is based on algorithm $\mathcal{W}$, algorithm $\mathcal{U}_{AE}$ also only reports application expressions as error location. To tackle this issue, $\mathcal{IEI}$ (incremental error inference) was developed in order to find more precise error location. The technique is to switch to Algorithm $\mathcal{M}$ when the unification algorithm fails at application expression.

Other alternatives also have been suggested, for example, reparation systems [12),13)] which use heuristics to find the most relevant error location, and interactive systems which use the programmers' intention and assumption to give better support. The effectiveness of every approach is subjected to the target's preferences. Investigation of the common error cases and requirements is needed to choose a suitable approach for certain environment.

The category most suitable for our algorithm is program slicing, where type error is reported as a set of program slices that contributes to the conflict. Dinesh and Tip observed that the tracking of positional information is very crucial for assisting the programmer to determine program locations that need to be changed in order to fix type errors [14]. Their rewriting technique is language independent but it is more applicable to explicitly typed languages. Haack and Wells present an algorithm to compute minimal type error slices for implicitly typed higher-order languages with let-polymorphism [5]. The extension of the algorithm for SML language [15] has a web implementation demonstrating their approach.

Compared to Haack and Wells' approach, our algorithm computes all the error locations simultaneously, without artificially breaking them into several slices for every pair of conflicting types. For example, given the list $[1; 2; 3; 4; 5; 6; \mathit{true}]$, our algorithm produces one slice $[1; 2; 3; 4; 5; 6; \mathit{true}]$ for the type error, while (the first phase of) Haack and Wells' algorithm produces six slices, the first of which identifies n and *true* as the error regions for each $n = 1, 2, \ldots, 6$. As a matter of fact, their algorithm needs a post-processing phase to find overlapping errors, and merge all overlapping errors if any. In the previous example, *true* is common to the six slices obtained in the first phase, hence these six slices are overlapping. Their algorithm then merges the six slices into one, and will produce an error similar to ours.

We argue that, although the outputs of the two algorithms are virtually the same, our algorithm is expected to run more efficiently since their algorithm needs an extra phase for finding and merging overlapping errors. If the given expression has a type conflict among N subexpressions whose types are mutually distinct [⋆1], then there are $N(N-1)/2$ pairwise slices, hence the cost of merging is at best $O(N^2)$. Although our unification algorithm may take slightly longer time than the time needed by the standard unification algorithm (because of the treatment of ψ-type variables), the extra cost is expected to be $O(N)$ or less. Hence, if N is large such as 100, then our algorithm is advantageous in efficiency.

The multiple-type constraint introduced in our unification algorithm is similar

⋆1 A typical example for this is a list of length N whose elements have mutually distinct types.

to the multi-equations used for unification algorithm described by Pottier and Remy [16]. Although the idea of using equations between multiple types for constraint is applied in both unification algorithms, the nature and the workings of the algorithms are very different. Our algorithm is specially developed to identify error location for improving error report.

6. Conclusion and Future Work

We present an algorithm to identify all the type errors that occur in a program. The output is shown in the form of error slices, each of which has the information regarding the program points that contribute to each type error. Therefore, we can fix a type error without referring to other parts of the program that are not identified in the corresponding slice. A type error can involve any number of conflicting types and any number of program points.

We also have implemented the algorithm in Section 3 in a simple prototype. All the error messages shown in Section 2.2 were essentially obtained as the outputs of our prototype implementation. For future improvement, we can extract useful information from the output to form error messages. Although identifying program points that contribute to the type errors is important, error explanation is also crucial. For example, attaching the conflicting types to the erroneous points can make it easier for programmers to understand the type errors. Deciding the right level of detail to form a concise but sufficient type error explanation is also necessary for improving the quality of error messages.

Although error-reports are mostly text-based, graphical user interface also has a role in improving the quality of the message, for example, in showing the level of contribution of every slice. Some techniques such as highlighting can help pinpointing locations based on their contributions. For example, it can indicate which parts are the end points, slices that produce the conflicting types, and which parts are the slices that participate in the relationships among the conflicting types. Another issue is how to deal with terms that have no special syntax to be marked, such as a function application. Generally a function application only has space between expressions, so it is important to consider ways to convey the information when a function application contributes to an error while its subexpressions do not. Our algorithm also can be improved and used for various application, for example, incremental typechecking.

In the future, we want to apply our approach in a computer-assisted learning system for computation and logic [17]. With students as the target audience, it is necessary to form error messages that match the level of their expertise. Moreover, for such e-learning system that is generic and not bound to any specific formal system, we need to examine ways to make maximum use of the techniques we have researched. By using the improved error messages, our goal is to make it easier and more convenient for students to understand formal systems and learn from their mistakes.

Acknowledgments The authors would like to thank anonymous reviewers and the participants of IPSJ SIG-PRO meeting on March, 2010 for constructive comments. The second author is supported in part by Graint-in-Aid for Scientific Research, No. 20650003 and No. 21300005.

References

1) Wand, M.: Finding the source of type errors, *POPL '86: Proc. 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, pp.38–43, ACM (1986).
2) Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic, *Transactions of the American Mathematical Society*, Vol.146, pp.29–60 (1969).
3) Milner, R.: A theory of type polymorphism in programming, *J. Comput. Syst. Sci.*, Vol.17, pp.348–375 (1978).
4) Damas, L. and Milner, R.: Principal type-schemes for functional programs, *POPL '82: Proc. 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.207–212, ACM (1982).
5) Haack, C. and Wells, J.B.: Type error slicing in implicitly typed higher-order languages, *Sci. Comput. Program.*, Vol.50, No.1-3, pp.189–224 (2004).
6) Yang, J., Michaelson, G., Trinder, P. and Wells, J.B.: Improved Type Error Reporting, *Proc. 12th International Workshop on Implementation of Functional Languages*, pp.71–86 (2000).
7) Pierce, B.C.: *Types and programming languages*, MIT Press, Cambridge, MA, USA (2002).
8) Damas, L.: Type Assignment in Programming Languages, PhD Thesis, University of Edinburgh (1985).
9) Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM*, Vol.12, No.1, pp.23–41 (1965).
10) Heeren, B.J.: Top Quality Type Error Messages, PhD Thesis, Universiteit Utrecht,

The Netherlands (2005).
11) Yang, J.: Explaining Type Errors by Finding the Source of a Type Conflict, *Trends in Functional Programming*, Michaelson, G., Trindler, P. and Loidl, H.-W. (Eds.), pp.58–66, Intellect Books (2000).
12) Lerner, B., Grossman, D. and Chambers, C.: Seminal: searching for ML type-error messages, *ML '06: Proc. 2006 workshop on ML*, New York, NY, USA, pp.63–73, ACM (2006).
13) Lerner, B.S., Flower, M., Grossman, D. and Chambers, C.: Searching for type-error messages, *SIGPLAN Not.*, Vol.42, No.6, pp.425–434 (2007).
14) Tip, F. and Dinesh, T.B.: A slicing-based approach for locating type errors, *ACM Trans. Softw. Eng. Methodol.*, Vol.10, No.1, pp.5–55 (2001).
15) Rahli, V., Wells, J.B. and Kamareddine, F.: Challenges of a type error slicer for the SML language, Technical Report HW-MACS-TR-0071, Heriot-Watt University (2009).
16) Pottier, F. and Remy, D.: The essence of ML type inference, *Advanced Topics in Types and Programming Languages*, Pierce, B.C. (Ed.), chapter 10, pp.389–489, MIT Press (2005).
17) Kameyama, Y. and Sato, M.: E-learning of Foundation of Computer Science, *Proc. AEARU Workshop on Network Education*, pp.169–181 (2006).

(Received February 15, 2010)
(Accepted May 1, 2010)

Cynthia Kustanto is a Graduate Student of the Master's Program at the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba. She is interested in programming languages and human-computer interaction.

Yukiyoshi Kameyama is a Professor of Computer Science at the Graduate School of Systems and Information Engineering, University of Tsukuba. He is interested in programming logic and software verification. He is a member of ACM, JSSST, and IEICE.