An Approach to Completing Variable Names for Implicitly Typed Functional Languages

Takumi Goto Isao Sasano

Shibaura Institute of Technology
Tokyo, Japan
{m110057, sasano}@sic.shibaura-it.ac.jp

Abstract

This paper presents an approach to completing variable names when writing programs in implicitly typed functional languages. As a first step toward developing practical systems, we considered a simple case: up to the cursor position the program text is given completely. With this assumption we specify a variable completion problem for an implicitly typed core functional language with let-polymorphism and show an algorithm for solving the problem. Based on the algorithm we have implemented a variable name completion system for the language as an Emacs-mode.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program editors

General Terms Reliability, Theory, Languages

Keywords polymorphic language, type inference, Emacs-mode, variable name completion

1. Introduction

Integrated development environments (IDE) play an important role in developing large software. IDEs provide functionalities including automatic indentation, keyword highlighting, variable name completion, and so on. Among them one of the most basic and convenient functionalities is variable name completion: when inputting a variable name, candidates for the variable names that start with the string which has been input are shown on a pop-up window for example. In developing large programs, we tend to use long names especially for variables with long scopes for enhancing the program readability, and in such cases variable name completion substantially decreases the time for recalling the variable names as well as the amount of keyboard typing or spelling errors. Up to now IDEs for commonly-used languages like Java, C, C++ have been well developed while not so much for functional programming languages. Functional languages are now getting to be used widely so that IDEs for them are strongly expected.

IDEs should be designed and implemented reliably as well as compilers, since otherwise people may not want to use them. Especially IDEs for statically typed functional languages are expected to utilize the feature of being statically typed. Reflecting typing information on variable name completion reduces the number of candidates for completion, as well as decreases type errors and spelling errors. Actually IDEs for Java, including Java IDE on Eclipse, provide member variable or method name completion, after typing dot, with type information (class definition) reflected. In explicitly typed languages like Java, type information can be directly computed from the program text. In implicitly typed languages, *i.e.*, those which allow either annotating or not annotating types to variable declarations, it is not so obvious how to design variable completion mechanism with type information reflected, since we need to do type inference in some way when completing variable names.

In developing small programs, purely syntactic variable completion may work well without using type information. Our intention is to complete variable names that are defined in some libraries or some other program modules. For example, let us consider the case where a programmer is writing a program fragment that prints some integer value. In Standard ML she might write the fragment as follows[1].

```
print (Int.
```

Here let us suppose that she does not recall the function name that converts an integer to the corresponding string but remembers the structure name `Int` inside which the function is declared. In this situation there are 29 candidates since the structure `Int` has 29 declarations. By using type information, the candidates are reduced to two since the function `print` takes a value of string type as its argument and the structure `Int` has only two functions that may return a value of string type: `fmt` and `toString`.

In this paper we present a basic mechanism of variable name completion for an implicitly typed ML-like core functional language. In order to filter candidates by type information, we need to do type inference for incomplete programs. While various settings can be considered, we consider a simple case: up to the cursor position, the program text is given completely. More precisely, we assume that the input programs before the cursor position do not have any syntax error or type error. When computing candidates for variable names to be completed we only use the program text before the cursor position and ignore the text after the cursor position. This simple approach works well for ML-like languages since all the variables are declared syntactically before their use in ML-like languages except for (mutually) recursive declarations. In the future we may extend the approach to use the program text after the cursor position for supporting Haskell-like languages where variables may appear syntactically before their declarations.

[1] Although our current system does not deal with programs with structures (or modules), here we use this example since it is a typical case where filtering with types effectively reduces the number of candidates.

If the system might sometimes eliminate valid candidates, it would be confusing since the programmer would have to consider whether or not some valid candidates may exist other than the candidates in the pop-up window. So it is strongly desired for the variable name completion system to have the property that all the possible variables are shown as candidates. On the other hand, variable name completion system is expected to eliminate all the variables that cause type error for whatever program text is input after the cursor position. In Section 7 we state as conjectures that our solution presented in this paper satisfies these two properties.

As for the cost of inferring types for partial programs, we construct an efficient partial type inference algorithm while retaining the above two properties. It is sufficient to do type inference for each variable that is within its scope at the cursor position, but it includes many redundant computation. As will be presented in Section 5, our partial type inference algorithm appropriately abstracts types of variables that are within their scope at the cursor position, which results in an efficient algorithm suitable for practical use.

Based on the mechanism we present, we have implemented variable name completion as an Emacs mode for the ML-like core language. Our experiment shows that our mechanism works well at least for the core language. Currently we apply the algorithm for computing candidates every time when some character is input on Emacs. We expect that the algorithm can be incrementalized by reusing the previous computation, which we leave as future work.

The rest of the paper is organized as follows. Section 2 specifies the variable completion problem we solve. Section 3 shows our basic ideas and outline of our solution. Section 4 describes the term completion phase by introducing two abstractions, dummy nodes and marked nodes. Section 5 describes the type inference algorithm. Section 6 describes the phase for filtering by type constraint and the (partial) spelling of the variable name being input at the cursor position. Section 7 summarizes the algorithm and gives the properties of our algorithm. Section 8 describes our implementation as an Emacs mode for a small subset of Standard ML. Section 9 describes some experimental results. Section 10 discusses about related work. Section 11 describes future work and concludes the paper.

2. Specification of variable completion problem

In this section we specify the problem that we solve in this paper. We use the following core language on which we specify the problem.

$$\begin{array}{rcl} M & ::= & x \mid c \mid \lambda x.M \mid M\ M \mid (M) \\ & \mid & \mathbf{let}\ x = M\ \mathbf{in}\ M\ \mathbf{end} \mid \mathbf{fix}\ x.M \end{array}$$

Here x represents a variable, c represents a constant, $\lambda x.M$ represents a function abstraction, $M\ M$ represents a function application, $\mathbf{let}\ x = M\ \mathbf{in}\ M\ \mathbf{end}$ represents a let expression, and $\mathbf{fix}\ x.M$ represents a fix expression. For simplicity we suppose constants are + and − of type $int \to int \to int$ and natural numbers of type int. We have explicitly included the parentheses in the core language in order to express situations where there are open parentheses not yet closed. For simplicity we only consider the program without type annotations.

We consider the following types for the core language.

$$\sigma := \forall\alpha_1 \ldots \alpha_n.\tau \qquad \tau := int \mid \alpha \mid \tau \to \tau$$

The types consist of the integer type int, type variables, function types, and polymorphic types. Here we used σ as a meta variable for representing a polymorphic type and τ as a meta variable for representing a monomorphic type. When $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau_0$ and there exists some $\tau_1, \ldots, \tau_n$ such that $\tau = [\tau_1/\alpha_1\ \ldots\ \tau_n/\alpha_n]\tau_0$, we call τ as an instance of the polymorphic type σ and write $\tau < \sigma$.

$$\begin{array}{rl} \text{(const)} & \Gamma \rhd c : \tau\ \ (c : \tau \in \mathit{Const}) \\ \text{(var)} & \Gamma\{x : \sigma\} \rhd x : \tau\ \text{ if } \tau < \sigma \\ \text{(app)} & \dfrac{\Gamma \rhd M_1 : \tau_1 \to \tau_2\ \ \Gamma \rhd M_2 : \tau_1}{\Gamma \rhd M_1\ M_2 : \tau_2} \\ \text{(abs)} & \dfrac{\Gamma\{x : \tau_1\} \rhd M : \tau_2}{\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2} \\ \text{(let)} & \dfrac{\Gamma \rhd M_1 : \tau_1\ \ \Gamma\{x_1 : Cls(\Gamma, \tau_1)\} \rhd M_2 : \tau_2}{\Gamma \rhd \mathbf{let}\ x_1 = M_1 \mathbf{in}\ M_2\ \mathbf{end} : \tau_2} \\ \text{(fix)} & \dfrac{\Gamma\{x : \tau\} \rhd M : \tau}{\Gamma \rhd \mathbf{fix}\ x.M : \tau} \end{array}$$

Figure 1. Type system for the core language

$$\begin{array}{l} \mathit{pre} : M \to \{(P,\ x)\} \\ \mathit{pre}\ c = \{\} \\ \mathit{pre}\ x = \{(\llcorner_s,\ x) \mid s \text{ is a prefix of } x\} \\ \mathit{pre}\ (M_1\ M_2) = \{(M_1\ P_2,\ x_2) \mid (P_2,\ x_2) \in \mathit{pre}\ M_2\} \cup \\ \qquad\qquad \{(P_1,\ x_1) \mid (P_1,\ x_1) \in \mathit{pre}\ M_1\} \\ \mathit{pre}\ (\lambda x.M) = \{(\lambda x.P,\ x_1) \mid (P,\ x_1) \in \mathit{pre}\ M\} \\ \mathit{pre}\ (\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2\ \mathbf{end}) = \\ \qquad \{(\mathbf{let}\ x = M_1\ \mathbf{in}\ P_2,\ x_2) \mid (P_2,\ x_2) \in \mathit{pre}\ M_2\} \cup \\ \qquad \{(\mathbf{let}\ x = P_1,\ x_1) \mid (P_1,\ x_1) \in \mathit{pre}\ M_1\} \\ \mathit{pre}\ (\mathbf{fix}\ x.M) = \{(\mathbf{fix}\ x.P,\ x_1) \mid (P,\ x_1) \in \mathit{pre}\ M\} \\ \mathit{pre}\ ((M)) = \{((P,\ x_1) \mid (P,\ x_1) \in \mathit{pre}\ M\} \end{array}$$

Figure 2. Function *pre* for prefix relation

We use a usual ML-style type system with let-polymorphism given in Figure 1, where we omit the case for (M). As usual we write $\Gamma \rhd M : \tau$ for a type judgment that a term M has a type τ under a type environment Γ. A type environment is a mapping from variables to types. When we add a mapping from a variable x to a type τ to a type environment Γ, we write $\Gamma\{x : \tau\}$. We can obtain the type of a variable x under a type environment Γ by $\Gamma(x)$. Cls is a function that takes a type environment Γ and a type τ and returns $\forall\alpha_1 \cdots \alpha_n.\ \tau$ when $FTV(\tau) \setminus FTV(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$, where FTV takes a type τ or a type environment Γ and returns a set of free type variables in it. *Const* is a set of types of constants.

As we mentioned in Section 1, we only use the program text before the cursor position. In order to represent such an incomplete program text we introduce the following *prefix* of the core language.

$$\begin{array}{rcl} P & ::= & \llcorner \mid \lambda x.P \mid M\ P \mid (P \\ & \mid & \mathbf{let}\ x = M\ \mathbf{in}\ P \mid \mathbf{let}\ x = P \mid \mathbf{fix}\ x.P \end{array}$$

Here we introduced a cursor node $\llcorner$, which corresponds to the cursor position in the program being written. For simplicity we only complete variable names and do not complete keywords or constants, so the cursor node $\llcorner$ corresponds to a variable and does not correspond to constants or any other constructs. Each of the prefix terms defined above ends with the cursor node $\llcorner$. The cursor node $\llcorner$ has as its attribute the (partial) spelling of a variable now being input. The spelling may be an empty spelling, which we write ϵ. We may write the spelling as the subscript of the cursor node like $\llcorner_{\mathtt{f}}$ and $\llcorner_{\epsilon}$ when necessary.

Here we formally specify the prefix relation between P and M, with the variable name being input at the cursor position, by the function *pre* in Figure 2. The function *pre* takes a term M and returns all the prefix terms of M, each of which is paired with a variable name being focused on. Note that *pre* returns the empty set for the case of constant c since we do not complete constants. Note also that we do not complete the identifiers bound at function abstractions or declarations, which is reflected in the definition

of *pre*. We show an example for *pre*. By applying *pre* to a term $(\lambda\texttt{abc.\ abc})\ \texttt{1}$, we obtain the following set.

$$\{(\texttt{(}\lambda\texttt{abc.\ }\textvisiblespace_{\epsilon}\texttt{, abc)}, \texttt{(}(\lambda\texttt{abc.\ }\textvisiblespace_{\texttt{a}}\texttt{, abc)},\\ \texttt{(}(\lambda\texttt{abc.\ }\textvisiblespace_{\texttt{ab}}\texttt{, abc)}, \texttt{(}(\lambda\texttt{abc.\ }\textvisiblespace_{\texttt{abc}}\texttt{, abc)}\}$$

The constant `1` is not included in this result since *pre* returns an empty set in the case of constant.

Now we are ready to specify the problem.

PROBLEM 1 (Variable completion). *Given a prefix term P and a type environment Γ, find a set V of variables such that $\forall v \in V,\ \exists M,\ \exists \tau, \Gamma \triangleright M : \tau,\ (P,\ v) \in$ pre M.*

The set V of variables are the candidates to be popped up. Most desirable answer to the problem is the *largest* set V of variables. Actually our algorithm for solving the problem presented in this paper gives the largest one. Note that in practical situations Γ includes types of variables declared in libraries.

Let us see a simple example, where the following prefix term (partial program) P is given.

$$\textbf{let}\ \texttt{xx} = 1\ \textbf{in let}\ \texttt{xy} = \lambda\texttt{x.}\lambda\texttt{y.x y}\ \textbf{in let}\ \texttt{xz} = \lambda\texttt{x.x}\ \textbf{in}\ \texttt{xy}\ \textvisiblespace_{\texttt{x}}$$

The above term corresponds to the situation where we are typing a character `x` at the cursor position. Suppose the type environment Γ is $\emptyset$. Then candidates to be popped up have to be within their scope, so they must be some of the variables `xx`, `xy`, and `xz`. Among these variables, `xy` and `xz` satisfy the condition while `xx` does not. So the largest set is $\{\texttt{xy}, \texttt{xz}\}$. To be concrete, the types of these three variables are int, $\forall\alpha\beta.\ (\alpha \to \beta) \to \alpha \to \beta$, and $\forall\alpha.\ \alpha \to \alpha$, respectively. The cursor node $\textvisiblespace_{\texttt{x}}$ is immediately after the variable `xy`, so every candidate becomes the argument of the function `xy`. By typing constraint the variable `xx` is excluded.

Here let us see another example, where the following prefix term P is given.

$$\textbf{fix}\ \texttt{xf.}\lambda\texttt{xx.+ (xf (- xx 1)) (xf}\ \textvisiblespace_{x}$$

This term P corresponds to the situation where we are in the middle of writing the definition of a recursive function. We suppose again the type environment Γ is $\emptyset$. There are two variables, `xf` and `xx`, which are in the scope at the cursor position. The types of `xf` and `xx` are $int \to int$ and int respectively. In this case the type of the candidates must be int since the function `xf` takes as its argument an expression `(- xx 1)` of type int. So the largest set is $\{\texttt{xx}\}$.

3. Basic ideas

Here we show basic ideas for solving the variable completion problem. Following the problem specification, if we could generate all the terms of M having P as their prefix, then by type inference we could solve the problem. But obviously there are infinitely many terms M. In order to reduce the number of terms generated, we introduce the notion of *dummy node*. We then give a naive preliminary solution based on this notion, which may take much time and does not necessarily produce all the candidates. By introducing another notion of *marked node*, we reached our current solution, which produces all the candidates in time short enough to be used practically (see Section 9 for results of measuring time for completion).

3.1 A preliminary solution

Here we introduce the notion of *dummy node*. In other words, we generate terms of M that additionally have as its constructs the dummy node and cursor node. A dummy node is a placeholder for an arbitrary term. If we could generate all the terms using dummy nodes, we could solve the problem. But even by using dummy nodes there are still infinitely many terms to generate. So we had to use some threshold with respect to some criteria, e.g. the depth of the term. Then we do type inference for each of the terms. In type inference we assign a fresh type variable for the cursor node and fresh type variables for dummy nodes. For each term for which type inference succeeds, we enumerate the variables that are within their scope and then unify each of them with the type of the cursor node. We filter all variables for which the unification succeeds, and obtain those variables whose prefixes match the characters currently being typed.

Here we illustrate the naive solution by using the following simple term P.

$$\textbf{let}\ \texttt{ff} = \lambda\texttt{x.+ x}\ 1\ \textbf{in}\ \texttt{ff (}\textvisiblespace_{\texttt{f}}$$

Firstly we generate the terms each of which has the above term as its prefix, using dummy nodes. The above term with completing a closing parenthesis is one of the generated terms. In type checking we assign some fresh type variable α for the cursor node $\textvisiblespace_{\texttt{f}}$. As a result we obtain $\alpha = int$. In this case `ff` is the only variable that is within its scope. Since `ff` has type of $int \to int$, there is no candidate in this case.

There is another possible term, which is the above term with `ff (`$\textvisiblespace_{\texttt{f}}$ replaced by `ff (`$\textvisiblespace_{\texttt{f}}$ `[])`. Here $[\,]$ indicates a dummy node, which represents an arbitrary term that can be an argument of the function application. In type checking we assign fresh type variables α and β for $\textvisiblespace_{\texttt{f}}$ and $[\,]$ respectively, and obtain $\alpha = \beta \to int$. In this case the type of `ff` matches against the type of $\textvisiblespace_{\texttt{f}}$, so `ff` remains to be a candidate, which finally becomes a candidate to be popped up by checking `ff` has `f` as its prefix.

There are many other possible terms by adding function applications. One of such examples is the above term with `ff (`$\textvisiblespace_{\texttt{f}}$ replaced by `ff (`$\textvisiblespace_{\texttt{f}}$ `[] [])`. In this case the type of the cursor node becomes $\alpha \to \beta \to int$, which does not match with the type of `ff`.

There are actually infinitely many possible terms even by using dummy nodes since the programmer can write infinitely many expressions as arguments of function applications after the cursor position. So we used some threshold with respect to the depth of the terms and thus we may not necessarily obtain all the candidates.

We actually have implemented a system based on this naive solution, but it took much time (e.g., 100 seconds) in some cases and does not even necessarily generate all the candidates. So we considered how to overcome these two points. We show the outline of the solution we have reached in Section 3.2.

3.2 Our solution

By introducing another notion of *marked node*, which represents conceptually zero or more function applications with dummy nodes given as its arguments, we developed a solution that generates all the candidates without generating infinitely many terms. In the following, we illustrate our solution by the example above.

Since any expression can take an argument syntactically, the preliminary solution complemented a dummy node for each subterm. As a result there appeared a sequence of function applications with dummy nodes. Our idea is that this sequence can be abstracted by marking such nodes. For the above example our new algorithm constructs the following term by marking the nodes that can take an argument syntactically.

$$(\textbf{let}\ \texttt{ff} = \lambda\texttt{x.+ x}\ 1\ \textbf{in}\ (\texttt{ff}\ (\textvisiblespace_{\texttt{f}}{}^{*}))^{*}\ \textbf{end})^{*}$$

Nodes with asterisk $*$ are marked nodes. For example, $\textvisiblespace_{\texttt{f}}{}^{*}$ represents $\textvisiblespace_{\texttt{f}}$, $\textvisiblespace_{\texttt{f}}$ $[\,]$, $\textvisiblespace_{\texttt{f}}$ $[\,]$ $[\,]$, $\textvisiblespace_{\texttt{f}}$ $[\,]$ $[\,]$ $[\,]$, and so on, and $(\texttt{ff}\ (\textvisiblespace_{\texttt{f}}{}^{*}))^{*}$ represents `ff` $\textvisiblespace_{\texttt{f}}$, `ff (`$\textvisiblespace_{\texttt{f}}$ `[])`, `ff` $\textvisiblespace_{\texttt{f}}$ $[\,]$, `ff (`$\textvisiblespace_{\texttt{f}}$ `[])` $[\,]$, and so on. The type of the variable `ff` is $int \to int$, which is not influenced by whatever the nodes with the asterisk $*$ represent. Since the variable `ff` is the only variable that is within the scope at the cursor position, the possible types of $\textvisiblespace_{\texttt{f}}{}^{*}$ should be the possible types of $\texttt{ff}^{*}$. Since the type of `ff` is $int \to int$, possible terms represented

$$
\begin{array}{lcl}
cmp\ :\ P\ \to D & & \\
cmp\ \llcorner & = & \llcorner^* \\
cmp\ (\lambda x.P) & = & \lambda x.(cmp\ P)^* \\
cmp\ (M\ P) & = & (M\ (cmp_2\ P))^* \\
cmp\ (\mathbf{let}\ x = M\ \mathbf{in}\ P) & = & (\mathbf{let}\ x = M\ \mathbf{in}\ cmp\ P\ \mathbf{end})^* \\
cmp\ (\mathbf{let}\ x = P) & = & (\mathbf{let}\ x = cmp\ P\ \mathbf{in}\ [\,]\ \mathbf{end})^* \\
cmp\ (\mathbf{fix}\ x.P) & = & \mathbf{fix}\ x.(cmp\ P)^* \\
cmp\ (\texttt{(}P\ ) & = & (cmp\ P)^* \\
 & & \\
cmp_2\ :\ P\ \to D & & \\
cmp_2\ \llcorner & = & \llcorner \\
cmp_2\ (\mathbf{let}\ x = M\ \mathbf{in}\ P) & = & (\mathbf{let}\ x = M\ \mathbf{in}\ cmp\ P\ \mathbf{end}) \\
cmp_2\ (\mathbf{let}\ x = P) & = & (\mathbf{let}\ x = cmp\ P\ \mathbf{in}\ [\,]\ \mathbf{end}) \\
cmp_2\ (\texttt{(}P\ ) & = & cmp\ P
\end{array}
$$

Figure 3. Term completion function cmp

by $\texttt{ff}^*$ are either `ff` or `ff` $[\,]$ in order for the term to be well typed. The type of ${\llcorner_{\texttt{f}}}^*$ must be int since ${\llcorner_{\texttt{f}}}^*$ is in the argument position of the variable `ff`. So ${\llcorner_{\texttt{f}}}^*$ can only take the form of `ff` $[\,]$. The type of $(\texttt{ff}\ ({\llcorner_{\texttt{f}}}^*))^*$ must be int since the type of $\texttt{ff}\ ({\llcorner_{\texttt{f}}}^*)$ is int. As a result, `ff` becomes the candidate in this example.

Based on this idea we develop an algorithm for the variable completion in the following sections. The algorithm generates all the candidates and runs substantially faster than the preliminary naive solution.

4. Term completion with dummy nodes and marked nodes

As the first phase of the algorithm, we generate terms that have as their prefix the given partial term P by using dummy nodes and marked nodes. We call this generation phase as *term completion*. We define terms that include dummy nodes and marked nodes as follows.

$$
\begin{array}{rcl}
D & ::= & \llcorner \mid D^* \mid \lambda x.D \mid M\ D \mid \mathbf{let}\ x = D\ \mathbf{in}\ [\,]\ \mathbf{end} \\
 & & \mid \mathbf{let}\ x = M\ \mathbf{in}\ D\ \mathbf{end} \mid \mathbf{fix}\ x.D
\end{array}
$$

We omit the parentheses here since terms of D are already completed by dummy nodes $[\,]$.

We define a term completion function cmp from P to D in Figure 3. Although we are manipulating abstract syntax we would like to treat it in a way consistent with the concrete syntax. So we introduce the function cmp_2 to exclude the outer most mark in the argument part of the function applications, since otherwise function applications would become right associative, which violates the convention of lambda notations. Note that cmp_2 does not take $\lambda x.P$ or $M\ P$ since such cases do not occur according to the convention in describing lambda terms. Also note that the result of $cmp\ (\lambda x.P)$ is not $(\lambda x.(cmp\ P))^*$ since an expression input after P becomes part of the body of the lambda abstraction according to the convention in describing lambda terms.

We illustrate the function cmp by using an example. Suppose the following term P is given.

$$\mathbf{let}\ \texttt{xa} = \lambda\texttt{x.x}\ 2\ \mathbf{in}\ \mathbf{let}\ \texttt{yy} = \lambda\texttt{x.x}\ \mathbf{in}\ \mathbf{let}\ \texttt{xc} = 3\ \mathbf{in}\ \texttt{xa}\ (\llcorner_{\texttt{x}}$$

By applying cmp to this, we obtain the following term D.

$$
\begin{array}{l}
(\mathbf{let}\ \texttt{xa} = \lambda\texttt{x.x}\ 2\ \mathbf{in}\ (\mathbf{let}\ \texttt{yy} = \lambda\texttt{x.x}\ \mathbf{in} \\
\quad (\mathbf{let}\ \texttt{xc} = 3\ \mathbf{in}\ (\texttt{xa}\ ({\llcorner_{\texttt{x}}}^*))^*\ \mathbf{end})^*\ \mathbf{end})^*\ \mathbf{end})^*
\end{array}
$$

5. Type inference

In this phase we do type inference to obtain types of the variables and the cursor node. Our type inference algorithm $\mathcal{V}$, defined in Figure 4, is based on Milner's type inference algorithm $\mathcal{W}$ [18]. The algorithm $\mathcal{V}$ takes a pair of a term D and a type environment Γ and returns a set of tuples of a substitution, a type, and a pair of type environment and type of the cursor position. A substitution is a function from type variables to types. A substitution may be applied to types or type environments by natural extension. We use C as a meta variable for representing a pair of type environment and type of the cursor position and S as a meta variable for representing a substitution.

Our algorithm uses Milner's $\mathcal{W}$ as a subroutine in the cases of function application $M\ D$ and let expression $\mathbf{let}\ x = M\ \mathbf{in}\ D\ \mathbf{end}$. The algorithm $\mathcal{W}$ takes a type environment Γ and a term M as its arguments and returns a substitution S and a type τ. When $\mathcal{W}$ succeeds the judgment $S(\Gamma) \rhd M : \tau$ holds. We omit the definition of $\mathcal{W}$. The algorithm $\mathcal{V}$ uses the unification algorithm $\mathcal{U}$ [22] in the case of function application and fix expression, as is also the case for $\mathcal{W}$. The algorithm $\mathcal{U}$ takes a set of pairs of type expressions and returns one of the most general unifiers when they are unifiable and fails when they are not.

$\mathcal{V}$ is different from $\mathcal{W}$ in the following points.

- The terms given as the argument include the mark, dummy, and cursor node.
- The algorithm $\mathcal{W}$ returns a pair of a substitution and a type while the algorithm $\mathcal{V}$ returns a set of tuples of a substitution, a type, and a pair of the type environment at the cursor position and a type of the cursor position.
- We have not written explicitly in the algorithm $\mathcal{V}$ the cases where some unification fails, as is usual in the description of the algorithm $\mathcal{W}$. Unlike $\mathcal{W}$, when some unification fails, the entire algorithm $\mathcal{V}$ does not fail but just eliminates the case when taking union of the results.

A term D that includes marked nodes, dummy nodes, or a cursor node conceptually represents multiple terms of M. The type inference algorithm $\mathcal{V}$ instantiates each occurrence of the three constructs in an appropriate way to cover all the cases, so that $\mathcal{V}$ returns multiple results. The cursor node $\llcorner$ conceptually represents all the variables that are within their scope with the types constrained by the contexts. So for the case of the cursor node $\llcorner$, $\mathcal{V}$ enumerates all the variables in the domain of Γ. As for *arity* and g, we explain later. A marked node D^* conceptually represents zero or more function applications with dummy nodes given as its arguments. So for the case of D^*, $\mathcal{V}$ computes all the types taken from the *right spine* of the types of D, that is, $at(\tau)$ where τ is a type of D. The dummy node $[\,]$ conceptually represents all the terms and can thus have any type depending on the context, so fresh type variables are generated for the dummy node in the case of $\mathbf{let}\ x = D\ \mathbf{in}\ [\,]\ \mathbf{end}$.

In order to reduce the number of elements in the result of type inference, the algorithm $\mathcal{V}$ abstracts their types $\tau_1 \to \cdots \to \tau_k$, where τ_k is int or a type variable, into $\alpha_1 \to \cdots \to \alpha_k$ where $\alpha_1, \ldots, \alpha_k$ are fresh type variables. This is done by the functions *arity* and g. The function *arity* takes a type τ and counts the number of arguments, or *arity*, in τ. For example, *arity* $(int \to int \to int)$ results in 2. The function g takes n as its argument and generates a type that consists of $n + 1$ fresh type variables connected by the arrows. For example, $g(2)$ results in a type $\alpha_0 \to \alpha_1 \to \alpha_2$, where α_0, α_1, and α_2 are distinct fresh type variables. Using the functions *arity* and g the algorithm $\mathcal{V}$ groups types of variables that are within their scope at the cursor position into equivalence classes with respect to the arity. This makes computations about functions that have the same arity be performed at once. Since in typical cases functions take a few arguments, this abstraction effectively works for reducing redundant computations. The precise computation is delayed until the filtering phase in Section 6.

$$
\begin{array}{l}
\mathcal{V}(\Gamma,\ D) = \text{case } D \text{ of} \\
\quad {\scriptstyle\sqcup} \quad \Rightarrow \text{let } T = \{g(\mathit{arity}(\Gamma(x))) \mid x \in \mathit{dom}(\Gamma)\} \text{ in } \{(\emptyset,\ \tau,\ (\Gamma,\ \tau)) \mid \tau \in T\} \\
\quad \mid D^* \Rightarrow \{(S,\ \tau',C) \mid (S,\ \tau,\ C) \in \mathcal{V}(\Gamma,\ D),\ \tau' \in at(\tau)\} \\
\quad \mid \lambda x.D \Rightarrow \text{let } \{(S_0,\ \tau_0,\ C_0),\ldots,(S_i,\ \tau_i,\ C_i)\} = \mathcal{V}(\Gamma\{x:\alpha\},\ D) \quad (\alpha \text{ fresh}) \\
\qquad\qquad \text{in } \{(S_0,\ S_0(\alpha) \to \tau_0,\ C_0),\ldots,(S_i,\ S_i(\alpha) \to \tau_i,\ C_i)\} \\
\quad \mid M\ D \Rightarrow \text{let } (S_1,\ \tau_1) = \mathcal{W}(\Gamma,\ M) \\
\qquad\qquad \{(S_{2,0},\ \tau_{2,0},\ C_{2,0}),\ldots,(S_{2,i},\ \tau_{2,i},\ C_{2,i})\} = \mathcal{V}(S_1(\Gamma),\ D) \\
\qquad\qquad S_{3,j} = \mathcal{U}\{(S_{2,j}(\tau_1),\ \tau_{2,j} \to \alpha_j)\}\ (\alpha_j \text{ fresh})\ (j \in \{0,\ldots,i\}) \\
\qquad\quad \text{in } \{(S_{3,j} \circ S_{2,j} \circ S_1,\ S_{3,j}(\alpha_j),\ C_{2,j}) \mid j \in \{0,\ldots,i\}\} \\
\quad \mid \textbf{let } x = D \textbf{ in } [\,] \textbf{ end} \Rightarrow \text{let } \{(S_0,\ \tau_0,\ C_0),\ldots,(S_i,\ \tau_i,\ C_i)\} = \mathcal{V}(\Gamma,\ D) \\
\qquad\qquad\qquad \text{in } \{(S_0,\ \alpha_0,\ C_0),\ldots,(S_i,\ \alpha_i,\ C_i)\} \quad (\alpha_0,\ldots,\alpha_i \text{ fresh}) \\
\quad \mid \textbf{let } x = M \textbf{ in } D \textbf{ end} \Rightarrow \text{let } (S_1,\ \tau_1) = \mathcal{W}(\Gamma,\ M) \\
\qquad\qquad\qquad\qquad \{(S_{2,0},\ \tau_{2,0},\ C_{2,0}),\ldots,(S_{2,i},\ \tau_{2,i},\ C_{2,i})\} = \mathcal{V}(S_1(\Gamma)\{x:\mathit{Cls}(S_1(\Gamma),\ \tau_1)\},\ D) \\
\qquad\qquad\qquad\quad \text{in } \{(S_{2,j} \circ S_1,\ \tau_{2,j},\ C_{2,j}) \mid j \in \{0,\ldots,i\}\} \\
\quad \mid \textbf{fix } x.D \ \Rightarrow \text{let } \{(S_{1,0},\ \tau_0,\ C_0),\ldots,(S_{1,i},\ \tau_i,\ C_i)\} = \mathcal{V}(\Gamma\{x:\alpha\},\ D)\ (\alpha \text{ fresh}) \\
\qquad\qquad \{S_{2,0},\ldots,S_{2,i}\} = \mathcal{U}(\tau_j,\ S_{1,j}(\alpha))\ (j \in \{0,\ldots,i\}) \\
\qquad\quad \text{in } \{(S_{2,j} \circ S_{1,j},\ \tau_j, C_j) \mid j \in \{0,\ldots,i\}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{arity}(\forall\alpha_1\ldots\alpha_n.\tau) & = & \mathit{arity}(\tau) \\
\mathit{arity}(\tau_1 \to \tau_2) & = & \mathit{arity}(\tau_2) + 1 \\
\mathit{arity}(\alpha) & = & 0 \\
\mathit{arity}(int) & = & 0
\end{array}
\qquad
\begin{array}{lcl}
at(\tau_1 \to \tau_2) & = & \{\tau_1 \to \tau_2\} \cup at(\tau_2) \\
at(\alpha) & = & \{\alpha\} \\
at(int) & = & \{int\} \\
g(n+1) & = & \alpha \to g(n) \quad (\alpha \text{ fresh}) \\
g(0) & = & \alpha \quad (\alpha \text{ fresh})
\end{array}
$$

Figure 4. Type inference algorithm $\mathcal{V}$

The second element of the result for the case of D^* is the set of all types that are taken from the right spine of the type of D. This reflects the situation where the production $M := M_1\ M_2$ could be applied arbitrarily many times in the term completion phase without using the notion of marking.

In the algorithm $\mathcal{V}$ we do not apply substitution to the environments and types of the cursor position while traversing the term, but apply it to the result of the algorithm $\mathcal{V}$. The environments and types at the cursor position are obtained as follows: in each tuple $(S, \tau, (\Gamma_{\sqcup}, \tau_{\sqcup}))$ in the results of $\mathcal{V}(\Gamma,\ D)$, apply the substitution S to $\Gamma_{\sqcup}$ and $\tau_{\sqcup}$, respectively.

Let us see an example. Let D be the result of term completion in Section 4. In the following we show the process of computing $\mathcal{V}(\emptyset,\ D)$. The type environment given to V can be empty since the term D does not have free variables. When $\mathcal{V}$ encounters the cursor expression ${\scriptstyle\sqcup}\texttt{f}$, the type environment $\Gamma_{\sqcup}$ for the cursor position has been obtained as follows.

$$\Gamma_{\sqcup} \quad = \quad \{\texttt{xa} : \forall\alpha.(int \to \alpha) \to \alpha,\ \texttt{yy} : \forall\alpha.\alpha \to \alpha,\ \texttt{xc} : int\}$$

The algorithm $\mathcal{V}$ returns the following set of two tuples when it encounters the cursor expression ${\scriptstyle\sqcup}\texttt{x}$.

$$\{(\emptyset,\ \beta_1 \to \beta_2, (\Gamma_{\sqcup}, \beta_1 \to \beta_2)),\ (\emptyset,\ \beta_3, (\Gamma_{\sqcup}, \beta_3))\}$$

By taking all the types from the right spine of each of the above two types $\beta_1 \to \beta_2$ and β_3, $\mathcal{V}$ returns the following set of three tuples for ${\scriptstyle\sqcup}\texttt{x}^*$.

$$\begin{array}{l}\{(\emptyset,\ \beta_1 \to \beta_2, (\Gamma_{\sqcup}, \beta_1 \to \beta_2)),\ (\emptyset,\ \beta_2, (\Gamma_{\sqcup}, \beta_1 \to \beta_2)), \\ \ (\emptyset,\ \beta_3, (\Gamma_{\sqcup}, \beta_3))\}\end{array}$$

Next, when $\mathcal{V}$ processes $\texttt{xa}\ ({\scriptstyle\sqcup}\texttt{x}^*)$ the following unifications are tried:

$$\begin{array}{l} S_1 = \mathcal{U}\{(int \to \alpha_0) \to \alpha_0,\ (\beta_1 \to \beta_2) \to \gamma_1)\} \\ S_2 = \mathcal{U}\{(int \to \alpha_1) \to \alpha_1,\ \beta_2 \to \gamma_2)\} \\ S_3 = \mathcal{U}\{(int \to \alpha_2) \to \alpha_2,\ \beta_3 \to \gamma_3)\} \end{array}$$

where the types $(int \to \alpha_i) \to \alpha_i\ (i = 0, 1, 2)$ are instantiated types of the function `xa`. These instantiations are done in the function $\mathcal{W}$. All the unifications succeed and $\mathcal{V}$ returns the following set of three tuples for $\texttt{xa}\ ({\scriptstyle\sqcup}\texttt{x}^*)$.

$$\begin{array}{l}\{(S_1, \alpha_0, (\Gamma_{\sqcup}, \beta_1 \to \beta_2)),\ (S_2, \alpha_1, (\Gamma_{\sqcup}, \beta_1 \to \beta_2)), \\ \ (S_3, \alpha_2, (\Gamma_{\sqcup}, \beta_3))\}\end{array}$$

Eventually these three tuples become the results of $\mathcal{V}(\emptyset,\ D)$. As a final step, we apply S_1, S_2 and S_3 to $\beta_1 \to \beta_2$, $\beta_1 \to \beta_2$ and β_3 and obtain types $int \to \alpha_0$, $\beta_1 \to int \to \alpha_1$, and $int \to \alpha_2$ respectively, for the cursor. Similarly by applying S_1, S_2 and S_3 to $\Gamma_{\sqcup}$ we obtain $\Gamma_{\sqcup}$ as the type environment for the cursor position. Note that all the first elements of the third elements in the tuples of the result of the algorithm $\mathcal{V}$ are the same. We show filtering for this example in the next section.

6. Filtering candidates

In this phase we make the candidates to be shown in pop-up window. The initial candidates are taken from the variables in the domain of the type environment $\Gamma_{\sqcup}$ obtained in the type inference phase. They are filtered by unifying their types with the types of the cursor. Variables that failed to unify with any of the types of the cursor are deleted from candidates. When a candidate has a type of τ, we unify τ with each of the types of the cursor. When a candidate has a polymorphic type of $\forall\alpha_1\ldots\alpha_k.\tau$, we unify the type of $[\beta_1/\alpha_1]\ldots[\beta_k/\alpha_k]\tau\ (\beta_1,\ldots,\beta_k \text{ fresh})$ with each of the types of the cursor. Note that all of the types of the cursor are monomorphic. We finally filter the candidates by matching the spellings of the candidates with the (partial) spelling of the variable name being input.

We show an example for filtering candidates by using the result of type inference in Section 5. Candidates must have a type that is unifiable with $int \to \alpha_0$, $\beta_1 \to int \to \alpha_1$, or $int \to \alpha_2$. There are thus the following nine combinations.

$$\begin{array}{ll}
\mathcal{U}\{(int \to \alpha_0,\ (int \to \alpha_5) \to \alpha_5)\}, & (\text{for } \texttt{xa}) \\
\mathcal{U}\{(int \to \alpha_0,\ \alpha_6 \to \alpha_6)\}, & (\text{for } \texttt{yy}) \\
\mathcal{U}\{(int \to \alpha_0,\ int)\}, & (\text{for } \texttt{xc}) \\
\mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ (int \to \alpha_5) \to \alpha_5)\}, & (\text{for } \texttt{xa}) \\
\mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ \alpha_6 \to \alpha_6)\}, & (\text{for } \texttt{yy}) \\
\mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ int)\}, & (\text{for } \texttt{xc})
\end{array}$$

$\mathcal{U}\{(int \to \alpha_2,\ (int \to \alpha_5) \to \alpha_5)\}$, (for `xa`)
$\mathcal{U}\{(int \to \alpha_2,\ \alpha_6 \to \alpha_6)\}$, (for `yy`)
$\mathcal{U}\{(int \to \alpha_2,\ int)\}$ (for `xc`)

The second, the fourth, the fifth and the eighth unifications succeed. As a result we get two candidates `xa` and `yy`. Among these candidates the variable `xa` matches with the (partial) spelling `x` of the variable name currently being input while `yy` does not. So we obtain the singleton set $\{$`xa`$\}$ as the final candidate set.

There is another alternative about when filtering the candidates with the spelling of the variable name being input: to filter them in the first step. In the future we may incrementalize the variable completion system by caching the intermediate results of the computation. When we accomplish this, matching in the last step may effectively work in such cases where the programmer erases the last character of a variable name being input.

7. Properties of the algorithm

Here we summarize the algorithm and give some statements about the properties of the algorithm. The algorithm for solving Problem 1 can be summarized as follows.

ALGORITHM 1. *Let* $\{(S_0,\ \tau'_0,\ (\Gamma_\sqcup,\ \tau_0)), \ldots, (S_i,\ \tau'_i,\ (\Gamma_\sqcup,\ \tau_i))\}$ $= \mathcal{V}(\Gamma,\ cmp\ P)$ *and* s *be the spelling of the cursor node* $\sqcup_s$ *in* P. *Then compute the set* V *of variables as follows.*

$$V = \bigcup_{j \in \{0,\ldots,i\}} \left\{ \{x \mid \begin{array}{l} x \in dom(\Gamma_\sqcup), \\ s \text{ is a prefix of the spelling of } x, \\ \mathcal{U}\{((S_j(\Gamma_\sqcup))(x),\ S_j(\tau_j))\} \text{ succeeds} \end{array} \} \right\}$$

We expect Algorithm 1 satisfies the following two properties.

PROPERTY 1 (Soundness). *The set* V *of variables obtained by the algorithm satisfies the condition in Problem 1. That is,* $\forall v \in V,\ \exists M,\ \exists \tau, \Gamma \rhd M : \tau,\ (P,\ v) \in pre\ M$.

PROPERTY 2 (Completeness). *Any variable that satisfies the condition in Problem 1 is included in the set* V *of variables obtained by the algorithm. That is, if* $\exists M,\ \exists \tau, \Gamma \rhd M : \tau,\ (P,\ v) \in pre\ M$, *then* $v \in V$.

Properties 1 and 2 are our conjectures. As far as we have tried we have found no counterexample to the above two properties. The soundness property assures that all the unnecessary candidates are eliminated with respect to the typing constraint. By having this property the length of the list of candidates substantially decreases when there are many candidates without considering typing constraint. The completeness property is considered to be rather important in variable completion since programmers strongly expect that the variable they are trying to recall should be shown in the candidates. Having both properties makes the variable completion system really suitable for practical use.

8. Implementation

Based on our approach we have developed an Emacs mode called lambda-mode that provides variable name completion for a small subset of the Standard ML on Emacs. Lambda-mode is available on our web site `http://www.cs.ise.shibaura-it.ac.jp/lambda-mode/`. In this section we describe our implementation of lambda-mode. The variable name completion consists of the phases of computing the candidates and interacting with users, for which we give details in the followings. As for the filtering phase, it is implemented by following exactly what we have presented in Section 6.

start	:=	*exp*	(1)
exp	:=	*appexp*	(2)
	|	`fn` *id* `=>` *exp*	(3)
appexp	:=	*atexp*	(4)
	|	*appexp atexp*	(5)
atexp	:=	*id*	(6)
	|	*const*	(7)
	|	`(`*exp*`)`	(8)
	|	`let` *decseq* `in` *exp* `end`	(9)
dec	:=	`val` *valbind*	(10)
decseq	:=	*dec decseq*	(11)
	|	ϵ	(12)
valbind	:=	*id* `=` *exp*	(13)
	|	`rec` *id* `=` *exp*	(14)

Figure 5. Concrete syntax for the core language

startP	:=	*expP*
expP	:=	*appexpP*
	|	`fn` *id* `=>` *expP*
appexpP	:=	*atexpP*
	|	*appexp atexpP*
atexp	:=	$\sqcup$
	|	`(`*expP*
	|	`let` *decseq* `in` *expP*
	|	`let` *decseqP*
decP	:=	`val` *valbindP*
decseqP	:=	*dec decseqP*
	|	*decP*
	|	ϵ
valbindP	:=	*id* `=` *expP*
	|	`rec` *id* `=` *expP*

Figure 6. Prefix syntax for the core language

8.1 Core language

We have implemented the lambda-mode for the core language given in Figure 5, which is a subset of Standard ML. An expression `fn` x `=>` exp corresponds to a function abstraction $\lambda x.M$ for some M. A recursive function declaration `val rec` f `=` exp corresponds to a declaration $f = \mathbf{fix}\ f.M$ for some M.

The tokens for the core language are as follows.

`let`, id, `val`, `rec`, `in`, `end`, `=`, `=>`,
`fn`, `(`, `)`, $const$, *ws*, EOF

The tokens in type writer font correspond to the string of their own spellings. The token id is for variable names with the spelling as its attribute, $const$ is for constants with the value as its attribute, *ws* is for whitespace, and EOF is for representing the end of the sequence of tokens input to the parsing phase. More concretely, the token id is for sequences of capital and small alphabets, $const$ is for sequences of numbers from 0 to 9 that start with numbers from 1 to 9 and for the operators `+` and `-`, and *ws* is for tabs, spaces, and new line characters. The sequence of numbers have type of int, the operators have type of $int \to int \to int$.

8.2 Lexical analysis

The lexer reads all the program text until the character being just typed. We have implemented it by moving the cursor to the position immediately before the token being input and restoring the cursor position after the parsing. We do not move the cursor in the case the programmer input tabs, spaces, and new line characters. When the lexer reaches the cursor position, the lexer returns an id token

with the information being in the cursor position and memorizes the next token to return is EOF. We memorize the spelling of the token in the middle of typing and use it in the final filtering phase. In the following we write the token as $id_{\lrcorner}$.

8.3 Parsing

After lexical analysis, we do parsing to make a partial term P. In order to parse programs up to the cursor position, we made a concrete syntax for the prefix programs of the core language. We give the syntax for the prefix programs in Figure 6. Currently, we manually make the syntax for the prefix programs from the concrete syntax for the core language in Figure 5. The parser produces a prefix term P as a result of parsing. After obtaining a prefix term P, we proceed with exactly following the method we described in the previous sections.

We wrote the parser for the prefix programs fully in Emacs Lisp with the state transition table produced by applying a parser generator kmyacc [7], which is compatible with yacc, to a description of the syntax in Figure 6 with actions to construct a prefix term P.

We illustrate the parsing phase by using the following example.

```
let val ff = fn x => + x 1 in ff (f_
```

By parsing the above prefix program, we obtain the following prefix term P.

$$(\textbf{let}\ \texttt{ff} = \lambda\texttt{x}.\texttt{+}\ \texttt{x}\ \texttt{1}\ \textbf{in}\ (\texttt{ff}\ (\lrcorner_{\texttt{f}}^{*}))^{*})^{*}$$

There is one thing to note about the syntax of Standard ML. In Standard ML the right hand side of **val rec** declaration must be a function abstraction by the definition of the language [19]. When parsing **val rec** declaration, we check whether or not the right hand side of **val rec** declaration is a function abstraction. By the definition, it is allowed to (meaninglessly) parenthesize the right hand side of the **val rec** declaration like the following example.

```
let val rec f = (((fn x => f x))) in f 2 end
```

Although the parentheses are ignored in parse trees in general, we have to take into account the parentheses not yet closed. For example, let us consider the situation where we are inputting the argument `x` in the above example.

```
let val rec f = (((fn x => f _
```

In order to allow this kind of cases, the parser checks the constructed prefix term for the right hand side of the `val rec` declaration to allow arbitrarily many open parentheses before the function abstraction.

8.4 Type inference

The concrete syntax of the subset is almost same as the core language M, except for that let expressions may have one or more declarations $decseq$. There are two alternative approaches to handle let expressions having $decseq$. One is to convert them into nested let expressions and use the type inference the algorithm $\mathcal{V}$. The other is to replace the cases for the let expressions in the algorithm $\mathcal{V}$. We take the latter approach.

In the former approach, a let expression

```
let val ff = fn x => + x 1
    val yy = 3 in f y_
```

is transformed into a nested let expression. By parsing and term completion on the obtained prefix term P, we obtain the following term D.

$$\begin{array}{l}(\textbf{let}\ \textbf{val}\ \texttt{ff}\ = \lambda\texttt{x}.+\ \texttt{x}\ \texttt{1}\ \textbf{in}\\ \quad(\textbf{let}\ \textbf{val}\ \texttt{yy} = \texttt{3}\ \textbf{in}\\ \quad\quad(\texttt{f}\ (\lrcorner_{\texttt{y}}^{*}))^{*}\ \textbf{end})^{*}\ \textbf{end})^{*}\end{array}$$

As seen in the above, the term completion introduces a mark node for each partial let expression in the input partial expression P. Generally the complexity of the type inference algorithm $\mathcal{V}$ is exponential with respect to the number of mark nodes. But actually the mark nodes introduced by the nested let expressions obtained by the transformation do not increase the complexity exponentially because the computation corresponding to each of the mark nodes produces the same tuples as the input ones. Here the computation for the case of D^* once divides the types into their right spines and then makes the set union operations on them, producing the same set as the input one.

The former approach has the same approximate complexity as the latter approach, but the former approach unnecessarily divides the types and makes the set union operations. In order to avoid the redundant computation, we take the latter approach. We replace the cases for let expressions in the algorithm $\mathcal{V}$ in Figure 7. We modified $\mathcal{V}$ by replacing the cases for let expressions with the cases $\textbf{let}\ x_1 = M_1, \ldots,\ x_n = M_n,\ x_{n+1} = D\ \textbf{in}\ [\,]\ \textbf{end}$ and $\textbf{let}\ x_1 = M_1, \ldots,\ x_n = M_n\ \textbf{in}\ D\ \textbf{end}$. We convert the completed concrete syntax trees to a term D with the above modification before entering the type inference phase.

There is one thing to note about the type inference. Standard ML has value restriction [24], which we need to take into account in the future. The core language does not have side effect, so we do not check value restriction in the current implementation.

8.5 User interaction

We utilize an Emacs mode called auto-complete mode [5] for interacting with users. The auto-complete mode calls some function for computing candidates to pop up each time the programmer types a character. If there are some candidates then the mode displays them in a pop-up window. In this case the programmer either select a candidate or proceed on inputting next characters without selecting any candidate. If there are no candidates then the mode does nothing. Our implementation of the lambda-mode utilizes the auto-complete mode with providing a function that is called every time a character is typed to return the candidates. We show a screen shot of the lambda-mode in Figure 8, which is taken when the programmer has just typed the character `x`.

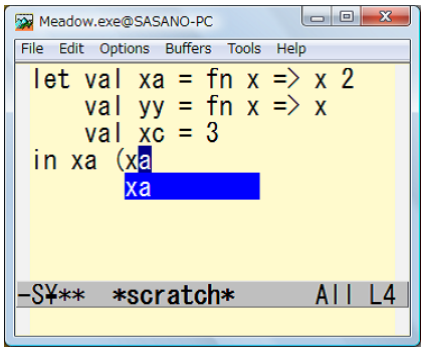

Figure 8. A snap shot of the lambda-mode

9. Experimental results

In usual cases the lambda-mode shows the candidates immediately after the programmer types a character. Since we do not have real large programs of the core language, we measured the time for extreme programs automatically generated.

The time is measured from when the programmer typed a key until when candidates are displayed. The environment in measuring the time is as follows: CPU is Intel Core i7 920, the size of memory is 6GB, OS is Windows 7 Home Premium 64bit, and emacs is

$$
\begin{array}{l}
\mathcal{V}(\Gamma,\ D) = \text{case } D \text{ of} \\
\quad \ldots \\
\quad | \ \textbf{let}\ x_1 = M_1,\ \ldots,\ x_n = M_n,\ x_{n+1} = D\ \textbf{in}\ [\,]\ \textbf{end}\ \Rightarrow \\
\qquad \text{let}\ (S_1,\ \tau_1) = \mathcal{W}(\Gamma, M_1) \\
\qquad\quad (S_2,\ \tau_2) = \mathcal{W}(S_1(\Gamma)\{x_1 : Cls(S_1(\Gamma),\ \tau_1)\}, M_1) \\
\qquad\qquad \ldots \\
\qquad\quad (S_n,\ \tau_n) = \mathcal{W}(S_{n-1} \circ \cdots \circ S_1(\Gamma)\{x_1 : Cls(S_1(\Gamma), \tau_1), \ldots, \\
\qquad\qquad\qquad\qquad\qquad\qquad x_{n-1} : Cls(S_{n-1} \circ \cdots \circ S_1(\Gamma), \tau_{n-1})\},\ M_n) \\
\qquad\quad \{(S_{n+1,0},\ \tau_{n+1,0},\ C_0), \ldots, (S_{n+1,i},\ \tau_{n+1,i},\ C_i)\} = \mathcal{V}(S_n \circ \cdots \circ S_1(\Gamma),\ D) \\
\qquad \text{in}\ \{(S_{n+1,0},\ \alpha_0,\ C_0), \ldots, (S_{n+1,i},\ \alpha_i,\ C_i)\}\ \ (\alpha_0, \ldots, \alpha_i \text{ fresh}) \\
\quad | \ \textbf{let}\ x_1 = M_1,\ \ldots,\ x_n = M_n\ \textbf{in}\ D\ \textbf{end} \Rightarrow \\
\qquad \text{let}\ (S_1,\ \tau_1) = \mathcal{W}(\Gamma, M_1) \\
\qquad\quad (S_2,\ \tau_2) = \mathcal{W}(S_1(\Gamma)\{x_1 : Cls(S_1(\Gamma),\ \tau_1)\}, M_1) \\
\qquad\qquad \ldots \\
\qquad\quad (S_n,\ \tau_n) = \mathcal{W}(S_{n-1} \circ \cdots \circ S_1(\Gamma)\{x_1 : Cls(S_1(\Gamma), \tau_1), \ldots, \\
\qquad\qquad\qquad\qquad\qquad\qquad x_{n-1} : Cls(S_{n-1} \circ \cdots \circ S_1(\Gamma), \tau_{n-1})\},\ M_n) \\
\qquad \text{in}\ \mathcal{V}(S_n \circ \cdots \circ S_1(\Gamma)\{x_1 : Cls(S_1(\Gamma), \tau_1), \ldots, \\
\qquad\qquad\qquad\qquad\qquad\qquad x_n : Cls(S_n \circ \cdots \circ S_1(\Gamma),\ \tau_n)\},\ D)
\end{array}
$$

Figure 7. The cases for the let expressions having the sequence of the declarations in the modified type inference algorithm $\mathcal{V}$

Meadow (GNU Emacs 22.2.1), although this kind of rather high spec machine is not necessarily required.

We tested for the extreme cases where let expressions have various numbers of declarations and some constructs such as let expressions, function abstractions, and parentheses have various depth of nesting. We show the case for let expressions with various numbers of declarations in Table 1, which is typical in functional languages. We typically write a sequence of many declarations, say 30 declarations, in one let expression.

The time complexity of our algorithm increases exponentially with respect to the depth of the nesting of let expressions, function abstractions, and function applications including the cursor node. The depth of the nesting of function abstractions, function applications, and let expressions is usually small and it is practically regarded as constant. Let expressions typically have a sequence of declarations, in which case the time complexity does not increase exponentially as mentioned in Section 8. In Table 1 we show the time for computing candidates for let expressions having 10, 30, and 50 declarations. In Table 1, the column "Constant" shows the cases where all declarations have different names in the left hand side and constant 1 in the right hand side. "Various" shows the cases where all declarations have different names in the left hand side and various expressions of various types in the right hand side. We have included the programs used in the experiment in the archive file of lambda-mode.

One thing to note is that there are many declarations in external libraries but ordinary type inference can be applied to them in advance. In the future we are planning to extend our variable name completion system to deal with external libraries.

Table 1. Comparison of completion time for let expressions with various numbers of declarations

		Time(millisec)	
		Constant	Various
The number of declarations	10	12	28
	30	30	93
	50	54	218

10. Related work and discussions

Variable name completion, more generally identifier completion, has naturally arised from practically used IDEs such as Eclipse. Currently many IDEs support identifier completion. We begin by summarising the current status of major IDEs supporting identifier completion in Section 10.1. We then discuss two aspects of our work, type inference and term generation, in Section 10.2 and 10.3 respectively, in comparison with several other studies. There is few academic research directly concerning identifier completion and actually as far as we know there is just one such work [21], about which we discuss in Section 10.4. Finally we make a remark about identifier completion for object oriented languages in Section 10.5.

10.1 Identifier completion in major IDEs

Here we summarize the current status of major IDEs that provide the functionality of identifier completion. They use various engines for identifier completion. Some of them simply compute candidates based on words that have been input in the current file. Some of them compute candidates based on identifiers that have been defined in some other modules or classes which were already compiled. Some of them are more intelligent: *Intellisense* in Visual Studio, *omni completion* in vim, and *content assist* [2] in Eclipse plugins for various languages. They try to give the candidates that fit in the context of editing program. For example intellisense and content assist take into account the scope of the variables. Although they are becoming more convenient, as far as we know, none of them filter the candidates by their types in languages with type inference.

In the following we describe individual IDEs with respect to the *intelligence* of engines for completion. First of all, there are many IDEs with no support for identifier completion. Such IDEs for functional languages include OCaml Development Tools [9], which is an Eclipse plugin for OCaml, and Caml mode [1] and tuareg mode [11], which are Emacs modes for editing Caml and OCaml code.

Some IDEs use engines that compute the candidates only from packages. Such IDEs include Java Development Environment for Emacs (JDEE) [6] which is an Emacs mode for Java, Eclipse FP [4] which is an Eclipse plugin for Haskell, and Leksah [8] which is an IDE for Haskell. JDEE complete the member names only in compiled classes, and Leksah and Eclipse FP complete the function and type names in packages. These IDEs do not complete names

declared in the current buffer not yet compiled, and hense cannot complete local variable names without compilation. JDEE partially takes into account the syntactic context of the cursor position such as the cases where the cursor is following dot, while Leksah does not.

More intelligent identifier completion engines are used in Visual F#, Visual C++, Visual C#, and Eclipse plugins for C++ and Java. They take into account the syntactic context of the cursor position to some extent. They show the candidates only when an expression can appear in the cursor position. They also consider the scope of variables.

The IDEs listed above which have variable completion functionality allow completion of identifiers that do not satisfy the typing constraint. In contrast, our completion fully utilizes the types when filtering candidates and excludes the candidates that do not satisfy the typing constraint.

As for typing information, Caml mode can show the type of the variable pointed by the cursor and show the location of its declaration, provided that OCamlSpotter [10], a tool for generating types and locations of the variables, processes the file including the declaration of the variable in advance.

10.2 Type inference

There have been some studies concerning type inference for incomplete programs, some of which we discuss in the followings.

Haack et al. [14] presented an approach to identifying the set of program points (a *slice*) that causes a type error in implicitly typed languages like Standard ML. They identified the criteria of *completeness* and *minimality* for type error slices and presented algorithms for finding complete and minimal type error slices. In this paper we assume that the input programs before the cursor position do not have any type error. When in the future we allow programs to have some type errors we might utilize their work to eliminate variables related to the type error from the candidates.

Lerner et al. [17] presented an approach to producing better type-error messages for languages with type inference such as OCaml and C++. They generate well-typed programs from an ill-typed program by replacing a program fragment that causes the type error with a "wildcard" expression, which plays the same role as a dummy node in this paper. As well as the work by Haack et al. [14], this work might be utilized to cope with programs with type error by replacing the fragments that cause the type error with dummy nodes.

10.3 Term generation

There have been some studies concerning term generation, some of which we discuss in the followings.

Hashimoto [15] constructed an ML-style programming language with first-class contexts, *i.e.*, expressions with holes. Holes correspond to dummy nodes in this paper, but his language has the operation for filling holes as a language construct while ours does not.

Some tools have been developed for generating terms under type constraint. One is Djinn [3], which generates a term having a given type, and another is to generate a minimum term having a given type [16]. In these tools a type is given for generating terms while in our variable name completion system, given a prefix term, we generate a term having dummy and mark expressions without being given a type.

Rittri [20] developed a method to search for an identifier in a program library by using types as search keys, regarding *isomorphic* types under some equivalence relation, considering argument permutation and the currying-uncurrying, as the same one. Runciman et al. [23] independently developed a method for the same problem by regarding types that are unifiable as the same. They also tried to handle a case where there is nothing unifiable in the library but there may be some identifier that yields the desired type by taking an extra argument. Cosmo studied the problem of *valid isomorphisms* in ML-like languages with let polymorphism and provided a complete and decidable characterization for it [13]. Our problem is more general than the problems in their work in the sense that in their problem a type is given as its input to find identifiers having its equivalent types while in our problem a *context*, *i.e.*, partial program text up to the cursor position, is given as its input to find identifiers that fit in the cursor position with some (not yet input) program text after the cursor position. Under the assumption that Property 2 holds, our work covers all of these cases in this sense.

10.4 Variable name completion

Here we discuss a study directly concerning variable name completion. Robbes et al. [21] pointed out that finding the correct candidate from the pop-up window can be cumbersome or even slower than typing the full name when using completion engines like content assist in Eclipse. In order to solve this problem they limit the number of the candidates so that the programmer can select a candidate quickly. They made some assumptions that programmers are likely to use methods they have just defined or modified and that local methods are called more often than the ones in other packages. Based on these assumptions they developed an algorithm to compute the candidates by using history of program editing. They claim that the candidates computed by their algorithm include the candidate the programmer is looking for with high probability. Our algorithm computes *all* the type-correct candidates while their algorithm may not.

10.5 Remark about object oriented languages

Let us remark about completion of identifiers in object oriented languages. Let us suppose we have typed some identifier for some object in object oriented languages like C++ and Java. When the object has some member variables or member functions, the identifier can be followed by a dot and the names. This can be made into a chain so that we can obtain various types of expressions. Alternatively speaking, functional and object oriented languages apply functions in reverse order: in an object oriented language, we give a parameter like "this" firstly, and then name the function (method), where in functional languages it's the other way around. So in object oriented languages filtering by types may not be effectively applied.

11. Conclusions and future work

In this paper we have presented an approach to completing variable names for implicitly typed functional languages. We have specified the variable completion problem in the simplest form and given a solution to the problem. The key ideas of our approach were the use of cursor and dummy nodes and insertion of marks. By using these ideas we have successfully developed basic mechanism of variable name completion system, based on which we plan to develop systems for real languages like Haskell, Standard ML, OCaml, and so on. In order to extend our solution to cover these languages there are several things to overcome, which we leave as future work.

- Computation of candidates processes the program text from the beginning of the text to the cursor position whenever a character is input, so many redundant computation may be performed. In real program development, a program is divided in many files each of which is not so large. What really matters is the time to process the program in the file currently edited. In particular we can compute necessary information in advance about the identifiers declared in libraries. Moreover in the future we may

incrementalize the completion algorithm, which decreases the time to compute the candidates. There is much work about the reuse of computation, so we expect they might be used for the reuse of intermediate results of parsing and type inference. For example, as for the reuse of type inference, Aditya et. al. [12] proposed an incremental algorithm for type inference. His work is to enable type inference to be performed in units of the top-level declarations.

- When there is an error in the beginning of a program, the variable name completion does not work for the entire program since we assumed that the program text before the cursor is given completely without any syntax error or type error. This assumption is too strict, so we are planning to extend the completion algorithm to handle the programs having syntax errors in some way similar to error recovery in parsing. As for the programs having type errors, we discussed in Section 10.
- In order for our approach to scale up to the level of real languages, the completion algorithm has to cope with many constructs including mutually recursive function declarations, infix operators, pattern matching, modules, and type annotations. We expect that pattern matching, type annotations, and modules should be naturally handled. We believe our framework works well with infix operators by extending the meaning of marked nodes to include the application of infix operators as well as usual prefix function applications. Note that infix operators have to be used syntactically together with the operands, so we have only to consider the infix operators already declared before the cursor node, except for those declared after the cursor position, which may be the case in mutually-recursive declarations or in Haskell-like languages. In this case we may have to explicitly require users that the infix operators should be defined before their use. As for mutually recursive function declarations, we expect the usual type inference works with natural modifications. To be concrete, when encountering some unknown identifiers, which may appear in mutually recursive function declarations, we simply assign a fresh type variable to the unknown identifiers.
- In Haskell variables can be used before their bindings. In ML languages variables can be used before their bindings in mutually recursive function definitions. Our framework is not directly applied to these situations since currently we do not use the program text after the cursor position according to the problem specification. In the future we intend to overcome the situation by enlarging the problem specification.

Acknowledgments

We would like to thank the anonymous referees for many helpful comments and for pointing out the paper in ASE 2008.

References

[1] Caml mode. `http://www.emacswiki.org/emacs/CamlMode`.

[2] Content assist. `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm`.

[3] Djinn. `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`.

[4] Eclipse FP. `http://eclipsefp.sourceforge.net/`.

[5] EmacsWiki: Auto complete. `http://www.emacswiki.org/emacs/AutoComplete`.

[6] Java development environment for Emacs. `http://jdee.sourceforge.net/`.

[7] KMyacc. `http://www005.upp.so-net.ne.jp/kmori/kmyacc/`.

[8] Leksah. `http://leksah.org/`.

[9] OCaml Development Tools. `http://ocamldt.free.fr/`.

[10] OCamlSpotter. `http://jun.furuse.info/hacks/ocamlspotter/`.

[11] Tuareg mode. `http://www-rocq.inria.fr/~acohen/tuareg/index.html.en`.

[12] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 379–405, Cambridge, USA, 1991.

[13] Roberto Di Cosmo. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 200–210, New York, NY, USA, 1992. ACM.

[14] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50:189–224, 2004.

[15] Masatomo Hashimoto. First-class contexts in ML. In *Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 206–223. Springer, 1998.

[16] Susumu Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming*, pages 111–126, 2005.

[17] Benjamin Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, NY, USA, 2007. ACM Press.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[20] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 174–183, New York, NY, USA, 1989. ACM.

[21] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[22] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[23] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 166–173, New York, NY, USA, 1989. ACM.

[24] Andrew Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8:343–355, 1995.