暗に型付けられた関数型言語に対する変数名補完方式の提案

後藤 拓実 [1], 篠埜 功 [2]

[1] 芝浦工業大学大学院 工学研究科 電気電子情報工学専攻
m110057@shibaura-it.ac.jp

[2] 芝浦工業大学 工学部 情報工学科
sasano@sic.shibaura-it.ac.jp

概 要 変数名補完は Eclipse や Visual Studio 等の開発環境で広く利用されている機能である。補完の候補はカーソル位置において有効な変数であり、型情報により候補の絞込みを行うと、コンパイル時の型エラーを減らす効果がある。ML 等の暗に型付けられた言語においては変数の型を得るために型推論を行う必要があるが、作成途中のプログラムを用いるため不完全な情報を元に型推論を行わなければならない。本論文ではプログラムは最初から順に記述されていくという前提で、暗に型付けられた関数型言語を対象とした変数名補完問題を定義し、それを解くアルゴリズムを提案する。提案するアルゴリズムは、候補となるべき変数のみが全て列挙されるという性質を持つ。提案手法に基づき Standard ML のサブセットについて変数名補完を行う Emacs モードを実装した。

1 はじめに

統合開発環境 (以下では IDE と略記) は大規模なソフトウェア開発において重要な役割を担っている。IDE は自動字下げやキーワードへの色付け、変数名補完などの機能を提供する。変数名補完はそれらの中でも最も基本的で便利な機能のひとつである。変数名補完とは変数名を入力する時に入力中の文字列を接頭辞に持つ変数名をポップアップウィンドウ等に表示し、その中からプログラマが選んだ変数名を自動で入力する機能である。大規模なプログラムの開発においてはプログラムを読みやすくするために長い変数名を用いることがよくあり、そのような場合に変数名補完により変数名を思い出す時間や入力の時間、綴りミスを削減することが出来る。広く使われている C や C++、Java などの言語の IDE においては様々な機能が提供されているが、関数型言語については十分な機能が提供されていない状況にある。関数型言語の利用者は近年徐々に増えつつあり、十分な機能を備えた関数型言語用の IDE の開発が望まれている。

IDE はコンパイラ同様にその設計、実装について信頼性が求められる。特に静的型付言語用の IDE は静的に型付けられているという特徴を活かすことが強く望まれる。変数名補完時に型情報を用いることで補完候補を絞り込みつつ型エラーを減らすことができる。実際に Eclipse を含む Java の IDE はドットを入力した時点でドットの左側の式の型情報を用いてメンバー名、メソッド名を補完する機能を備えている。Java のような陽に型付けられた言語では型情報はプログラムテキストに含まれている。それに対して、型注釈を省略可能な暗に型付けられた言語においては変数の型情報を得るために型推論を行う必要があり、型情報を用いた変数名補完システムをどのように設計するかはあまり自明ではない。

小さなプログラムの開発においては型の制約を考慮しない変数名補完でも十分であるが、ライブラリやモジュールの中で定義された変数名を補完する場合は、型の制約を用いた補完が有効である。例えば、Standard ML で以下のようなプログラム断片を入力しているとする[1]。

```
print (Int.
```

[1] この例は本論文で扱う変数名補完システムではまだ扱っていないが、型による絞込みが有効な例として挙げている。

ここではプログラマーは整数を文字列へ変換して画面へ出力するプログラムを書こうとしているとする。この時プログラマーは使いたい関数がストラクチャ`Int` の中に宣言されていることは覚えているが、関数名を思い出せないとする。ストラクチャ`Int` の中には 29 個の宣言があるが型情報を用いると、関数 `print` の引数の型が文字列型であることと、ストラクチャ`Int` には文字列型を返す関数が `fmt` と `toString` の 2 つしかないことから、候補の数を 2 つに絞り込むことが出来る。

我々の先行研究 [19] において、暗に型付けされた ML 系の核言語を対象とした変数名補完の基本的な枠組みおよび素朴な補完アルゴリズムを提案した。変数の型情報を得るためには型推論を行う必要があり、不完全なプログラムテキストに対して型推論を行うには様々な方法が考えられるが、我々はプログラムの先頭からカーソル位置までが完全に与えられているという仮定のもとで考えた。つまり、変数名補完の候補の計算にカーソル以前のテキストのみを利用し、カーソルより後ろのテキストは利用しない。ML 系の言語では（（相互）再帰関数の宣言を除いて）全ての変数は使用される位置よりも前に宣言される為、この仮定の下でも補完候補となる変数の情報は得ることが出来る。本論文においても同じ仮定を用いる。

候補となるべき変数が候補に入っていないことがあると、表示される候補の他に候補が存在する可能性を考えなければならず、そのような変数名補完システムは不便である。その為、変数名補完システムは候補となるべき変数は全て表示されるという性質（以降では完全性と呼ぶ）を持っていることが強く望まれる。また、選択された候補によって補完した場合に型エラーが起きるような候補は含まないという性質（以降では健全性と呼ぶ）も望まれる。我々の先行研究 [19] で提案した素朴なアルゴリズムは健全性は備えていたが、完全性は備えていなかった。本論文では、連続する関数適用を抽象化するマーク式を導入することにより健全性と完全性を備えたアルゴリズムを構築する。さらに、我々のアルゴリズムはカーソル位置で有効な変数の型を適切に抽象化することで冗長な計算を減らしている。

提案手法に基づいて ML 系の核言語を対象とした変数名補完を Emacs モードとして実装した。実験により核言語については補完候補の計算が十分実用になる時間内で行われることが示された。

本論文の構成は次の通りである。2 章で解くべき変数名補完問題を定義する。3 章で変数名補完の基本方針を示す。4 章でダミー式とマーク式を導入し、式の補完について述べる。5 章で型推論アルゴリズムについて述べる。6 章で型の制約と入力中の綴りによって候補を絞り込む方法について述べる。7 章でアルゴリズムのまとめとその性質について述べる。8 章で Standard ML の小さなサブセットを対象とした変数名補完の実装について述べる。9 章で実験とその結果について述べる。10 章で関連研究について述べる。11 章で将来の課題と本論文のまとめについて述べる。

2　変数名補完問題の定義

この章では本論文で扱う変数名補完問題を定義する。変数名補完の対象言語は以下の式 M とする。

$$M ::= x \mid c \mid \lambda x.M \mid M\ M \mid \textbf{let}\ x = M\ \textbf{in}\ M\ \textbf{end} \mid (M)$$

x は変数、c は定数、$\lambda x.M$ はラムダ抽象、$M\ M$ は関数適用、**let** $x = M$ **in** M **end** は let 式を表す。ラムダ式の慣習に従い、関数適用は左結合とし、ラムダ抽象の本体は出来るだけ大きくとることとする。括弧が閉じられていない場合を表現する為に、式 M の定義に明示的に括弧を含めている。また、議論を簡単にするために式 M には型注釈は入れていない。

式 M の型を次のように定義する。

$$\begin{aligned} \sigma &::= \forall\alpha_1 \ldots \alpha_n.\tau \\ \tau &::= int \mid \alpha \mid \tau \to \tau \end{aligned}$$

σ は多相型を表すメタ変数、τ は単相型を表すメタ変数、int は整数型、α は型変数を表すメタ変数、$\tau_1 \to \tau_2$ は型 τ_1 から型 τ_2 への関数型である。$\sigma = \forall\alpha_1 \ldots \alpha_n.\tau_0$ で、かつ $\tau = [\tau_1/\alpha_1\ \ldots\ \tau_n/\alpha_n]\tau_0$ となるような $\tau_1, \ldots, \tau_n$ が存在するとき、τ を多相型 σ の例と言い、$\tau < \sigma$ のように書く。

$$\text{(const)}\quad \Gamma \triangleright c : \tau \ \ (c : \tau \in Const) \qquad \text{(var)}\quad \Gamma\{x : \sigma\} \triangleright x : \tau \ \text{ if } \tau < \sigma$$

$$\text{(app)}\quad \frac{\Gamma \triangleright M_1 : \tau_1 \to \tau_2 \ \ \Gamma \triangleright M_2 : \tau_1}{\Gamma \triangleright M_1 \ M_2 : \tau_2} \qquad \text{(abs)}\quad \frac{\Gamma\{x : \tau_1\} \triangleright M : \tau_2}{\Gamma \triangleright \lambda x.M : \tau_1 \to \tau_2}$$

$$\text{(let)}\quad \frac{\Gamma \triangleright M_1 : \tau_1 \ \ \Gamma\{x_1 : Cls(\Gamma, \tau_1)\} \triangleright M_2 : \tau_2}{\Gamma \triangleright \textbf{let } x_1 = M_1 \textbf{in } M_2 \textbf{ end} : \tau_2}$$

図 1. 式 M の型システム

$$\begin{aligned}
&pre\ c \ = \ \{\} \\
&pre\ x \ = \ \{(\llcorner_s,\ x) \mid s \text{ is a prefix of } x\} \\
&pre\ (M_1\ M_2) \ = \ \{(M_1\ P_2,\ x) \mid (P_2,\ x) \in pre\ M_2\} \cup \{(P_1,\ x) \mid (P_1,\ x) \in pre\ M_1\} \\
&pre\ (\lambda x.M) \ = \ \{(\lambda x.P,\ x) \mid (P,\ x) \in pre\ M\} \\
&pre\ (\textbf{let } x = M_1 \textbf{ in } M_2 \textbf{ end}) \ = \ \{(\textbf{let } x = M_1 \textbf{ in } P_2,\ x) \mid (P_2,\ x) \in pre\ M_2\} \cup \\
&\qquad\qquad\qquad\qquad \{(\textbf{let } x = P_1,\ x) \mid (P_1,\ x) \in pre\ M_1\} \\
&pre((M)) \ = \ \{((P,\ x) \mid (P,\ x) \in pre\ M\}
\end{aligned}$$

図 2. 接頭辞関係を表す関数 pre

式 M の型システムは let 多相の ML 系の型システム [17] とし、図 1 のように定義する。図 1 では (M) の場合は省略している。型環境 Γ において式 M が型 τ を持つという型判定を $\Gamma \triangleright M : \tau$ と書く。型環境は変数から型への対応関係である。型環境 Γ に変数 x から型 τ への対応関係を追加した型環境を $\Gamma\{x : \tau\}$ と書く。型環境 Γ における変数 x の型を $\Gamma(x)$ と書く。関数 Cls は引数に型環境 Γ と型 τ を取り、$FTV(\tau) \setminus FTV(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$ の時 $\forall\alpha_1 \cdots \alpha_n.\ \tau$ を返す。関数 FTV は型や型環境に含まれる自由変数の集合を返す関数である。$Const$ は定数の型の集合である。

1 章で述べたように、補完候補の計算にはカーソル位置より前までのプログラムテキストのみを用いる。カーソル位置までのプログラム断片を表現するために、以下の**接頭辞式** P を導入する。

$$P ::= \llcorner \mid \lambda x.P \mid M\ P \mid \textbf{let } x = M \textbf{ in } P \mid \textbf{let } x = P \mid (P$$

ここでプログラム中のカーソル位置に対応するカーソル式 $\llcorner$ を導入した。議論を簡単にするために補完対象は変数名のみとし、定数やキーワードの補完は行わないものとする。つまり、カーソル式は定数やその他の構成子には対応しない。また、接頭辞式は常にカーソル式 $\llcorner$ で終わる。カーソル式 $\llcorner$ は入力中の変数名の部分的な綴りをその属性として持ち、論文中では必要に応じて部分的な綴り `f` をカーソル式 $\llcorner$ の添字として $\llcorner_{\texttt{f}}$ と書く。

次に入力中の（不完全な）変数名、M、および P の間の関係（以降では接頭辞関係と呼ぶ）を図 2 の関数 pre として定義する。関数 pre は引数に式 M を取り、M の接頭辞と接頭辞中のカーソル式に対応する変数の組の集合を返す。定数 c の場合は補完対象ではないので pre は空集合を返す。また、ラムダ抽象や let 式で変数名を定義する箇所では変数名補完は行わないが、そのことは pre の定義に反映されている。関数 pre の具体例として、M の式 `(λabc. abc) 1` に関数 pre を適用すると $\{((\lambda\texttt{abc.}\ \llcorner_{\texttt{a}}, \texttt{abc}), (\lambda\texttt{abc.}\ \llcorner_{\texttt{ab}}, \texttt{abc}), ((\lambda\texttt{abc.}\ \llcorner_{\texttt{abc}}, \texttt{abc})\}$ を得る。関数 pre は定数の場合に空集合を返すため、`1` はこの結果に含まれない。

これまでに導入した定義を用いて変数名補完問題を以下のように定義する。

問題 1 (変数名補完) 接頭辞式 P と型環境 Γ が与えられたとき、以下の条件を満たす V を求める。

$$\forall v \in V,\ \exists M,\ \exists \tau, \Gamma \triangleright M : \tau,\ (P,\ v) \in pre\ M$$

得られる集合 V が補完候補となる変数の集合である。この問題の最も望ましい解は最大の変数集合 V を求めることであり、我々の提案するアルゴリズムは最大の変数集合を返す。

変数名補完問題の例として以下のような接頭辞式 P が与えられた場合を考える。

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

この式はカーソル位置において x を入力している状況に対応している。型環境 Γ が $\emptyset$ の時、補完の候補は xx や xy、xz などのスコープの中にある変数でなければならない。これらの変数の中で型の制約を満たすのは xy と xz であり、xx は条件を満たさない。よって、候補となる変数の最大の集合は $\{\texttt{xy}, \texttt{xz}\}$ となる。具体的には、それら 3 つの変数はそれぞれ int、 $\forall\alpha\beta.\ (\alpha \to \beta) \to \alpha \to \beta$、$\forall\alpha.\ \alpha \to \alpha$ という型を持つ。カーソル式 ␣x は関数 xy の引数であり、補完候補は関数 xy の引数となるので、型の制約により変数 xx は除外される。

3　変数名補完の基本方針

この章では変数名補完問題を解くための基本的な方針について述べる。P を接頭辞に持つ式 M を全て生成できれば、それら M の中で型が付く式から変数を取り出すことにより補完候補を全て得られるが、P を接頭辞に持つ式 M は無限に存在する。そこで、生成する式の数を減らすために任意の式を表すダミー式を導入する。我々の既存研究 [19] においてダミー式を使った素朴な解法を提案したが、計算時間が長く、また候補となるべき変数が必ずしも候補に含まれないという問題があった。本論文では、連続する関数適用を抽象化するマーク式を新たに導入する。これにより、十分に短い時間で全ての候補を得ることが出来る。まず 3.1 節において我々の先行研究 [19] で提案した素朴な手法について概観したのち、3.2 節において本論文で提案する手法の中核となる考え方について述べる。

3.1　素朴な手法

素朴な手法では、与えられた P を接頭辞に持つ式は無限に存在するので、生成する式を減らすためにダミー式を導入した。ダミー式は直感的には任意の式を表す。ダミー式を用いて生成される式は M の構成要素にダミー式とカーソル式を追加したもので定義される。ダミー式の導入により生成する式は減少するが、まだ無限に存在する。素朴な手法では無限個の式の生成を避けるために式の深さについて閾値を設け、閾値を超えた場合に生成を中断することとした。また、型推論は生成された全ての式に対して行う。

型推論ではカーソル式とダミー式の型はどちらも新しい型変数とする。型推論が成功した式それぞれについて、カーソル位置で有効な変数を列挙する。それら列挙された変数の型とカーソル式の型が単一化 [18] 可能で、かつその中で入力中の文字を接頭辞に持つものを補完候補とする。

以下の接頭辞式 P が与えられた場合に対して、素朴な手法による補完候補の計算手順の例を示す。

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

ここで、+は $int \to int \to int$ 型の定数である。まずダミー式を用いて上記の式 P を接頭辞に持つ式を生成する。式 P に閉じ括弧とキーワード **end** を補って得られる式は生成される式の 1 つである。この式の型推論において、カーソル式 ␣f の型を新しい型変数 α とおき、$\alpha = int$ を得る。よってカーソル位置に補完可能な変数の型は int 型である。この式では変数 ff がスコープ内にある唯一の変数であり、変数 ff の型は $int \to int$ であるので、この場合は候補となる変数はない。

次に、上記の式 P の ff (␣f を ff (␣f []) で置き換えた後、キーワード **end** を補って得られる式について考える。[] はダミー式であり、任意の式を表す。この式について型検査をすると、␣f、[] の型をそれぞれ α、β とし、$\alpha = \beta \to int$ を得る。変数 ff の型 $int \to int$ は ␣f の型 $\beta \to int$ と単一化可能であり、さらに変数 ff は接頭辞に f を持つ。よって変数 ff は補完候補となる。

同様に、ダミー式を引数に取る関数適用を追加することで他にも多くの式を生成することが出来る。例えば、式 P の `ff (␣f` を `ff (␣f [] [])` で置き換えた後、キーワード **end** を補って得られる式の場合、カーソル式の型は $\alpha \to \beta \to int$ となり、変数 `ff` の型とは単一化できない。

カーソル位置より後ろには関数適用の引数を構文上は何個でも書くことが出来るので、ダミー式を用いて生成される式は実質的に無限にある。素朴な手法では、無限個の式を生成するのを避ける為に式の深さについて閾値を用いて生成を打ち切ることとした。我々は実際にこの手法に基づいて変数名補完システムを実装したが、対象とするプログラムの大きさや閾値によっては候補の計算に 100 秒以上かかったり、候補となるべき変数が候補にならない場合があるという問題があった。今回提案する手法はこれらの問題を解決しており、その概要を以下で述べる。

3.2 提案手法

素朴な手法では閾値以下の式を全部生成するため計算量が大きくなっていた。また、閾値があるためにすべての可能性が考慮されず、候補となるべき変数が候補にならない場合があった。これらを解決するため、ダミー式を引数に取る 0 回以上の関数適用を表すマーク式を導入する。これにより、無限個の式を生成せずに全ての候補を得ることができる。

任意の式は（構文上は）引数を取ることが出来る為、素朴な手法ではカーソル式を含んでいる全ての部分式にダミー式を引数として補っていた。結果としてダミー式を引数にとる関数適用が連続する式が生成されることになるが、このような関数適用の連続をマーク式によって抽象化できるということが本論文で提案する手法の考え方である。例えば、3.1 節の接頭辞式 P に対して提案手法を適用すると、構文上引数を取ることが可能な部分式にマークを付けることにより以下の式を得る。

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

アスタリスク $*$ の付いた式がマーク式である。例えば、$(\texttt{␣f}^*)$ は `␣f`, `␣f []`, `␣f [] []`, `␣f [] [] []` 等を表し、$(\texttt{ff } (\texttt{␣f}^*))^*$ は `ff ␣f`, `ff (␣f [])`, `ff ␣f []`, `ff (␣f []) []` 等を表す。変数 `ff` の型は $int \to int$ であり、これはアスタリスク付きの式の影響を受けない。カーソル位置で有効な変数は変数 `ff` のみであり、変数 `ff` が候補になるためには、$\texttt{␣f}^*$ の型は、$\texttt{ff}^*$ の取り得る型（変数 `ff` に任意個のダミー式を引数として与えた場合に取り得る型）となるべきである。変数 `ff` の型は $int \to int$ であるので $\texttt{ff}^*$ によって表される式で型が付くものは `ff` と `ff []` である。また、$\texttt{␣f}^*$ は変数 `ff` の引数なので $\texttt{␣f}^*$ の型は int でなければならない。したがって $\texttt{␣f}^*$ は `ff []` の形だけを取ることができる。`ff` $(\texttt{␣f}^*)$ の型が int であり、引数を取ることは出来ないので、$(\texttt{ff } (\texttt{␣f}^*))^*$ の型は int である。つまり、カーソル位置に変数 `ff` を補完した場合には型が付くので、`f` が変数 `ff` の接頭辞であることにより、`ff` は候補となる。

次の章からはこの考えに基づいたアルゴリズムの詳細について述べる。提案するアルゴリズムは全ての候補を実用に耐える時間で計算する。

4 ダミー式とマーク式による式の補完

この章以降において、変数名補完アルゴリズムを提示する。この章では、補完アルゴリズムの最初のステップとして、接頭辞式 P に対してダミー式とマーク式を用いることにより、P を接頭辞に持つ式を生成する。この式を生成する処理を**式の補完**と呼び、式の補完により生成されるダミー式とマーク式を含む式を以下のように定義する。

$$D \ ::= \ \texttt{␣} \mid D^* \mid \lambda x.D \mid M \ D \mid \textbf{let } x = D \textbf{ in } [] \textbf{ end} \mid \textbf{let } x = M \textbf{ in } D \textbf{ end}$$

接頭辞式 P の定義では閉じられていない括弧を表現する為に括弧の場合を明示的に入れていたが、D は補完後の式であり、閉じられていない括弧はないので、括弧の場合については入れていない。

式の補完関数 cmp を図 3 のように定義する。関数適用の引数部分にマークが付くと関数適用が右

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

図 3. 式の補完関数 cmp

結合になりラムダ式の慣習から外れるので、関数適用の引数部分の最も外側にはマークが付かないようにするために cmp_2 を用いている。cmp_2 は関数適用の引数の位置の式のみを引数に取っており、ラムダ式の慣習により $\lambda x.P$ や $M\ P$ は関数適用の引数になるには括弧が必要であるため、cmp_2 はこれらを引数にとらない。そのためこれらの場合は cmp_2 の定義に入れていない。また、$cmp\ (\lambda x.P)$ は $(\lambda x.(cmp\ P))^*$ ではない。なぜなら、P の後に入力される式はラムダ式の慣習によりラムダ抽象の本体に含まれるからである。式の補完関数 cmp の適用例を以下の P 式を用いて示す。

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

これに関数 cmp を適用すると次の式 D を得る。

$$(\mathbf{let}\ \mathtt{xa} = \lambda \mathtt{x}.\mathtt{x}\ 2\ \mathbf{in}\ (\mathbf{let}\ \mathtt{yy} = \lambda \mathtt{x}.\mathtt{x}\ \mathbf{in}\ (\mathbf{let}\ \mathtt{xc} = 3\ \mathbf{in}\ (\mathtt{xa}\ (\llcorner_{\mathtt{x}})^*)^*\ \mathbf{end})^*\ \mathbf{end})^*\ \mathbf{end})^*$$

5 型推論

この章では、補完アルゴリズムの 2 番目のステップとして、式の補完の結果得られる式 D の中の変数およびカーソル式の型を推論する。この推論を行うアルゴリズム $\mathcal{V}$ を、Milner の型推論アルゴリズム $\mathcal{W}$ [17] を基に図 4 のように定義する。アルゴリズム $\mathcal{V}$ は式 D と型環境 Γ の組を受け取り、置換、型、カーソル位置の型環境とカーソル式の型の組の 3 つ組の集合を返す。置換は型変数から型への関数であり、置換は型や型環境に対しても適用できるように拡張できる。

アルゴリズム $\mathcal{V}$ は Milner の型推論アルゴリズム $\mathcal{W}$ に基づいており、また、$\mathcal{V}$ の中で $\mathcal{W}$ を利用している。$\mathcal{W}$ は引数に型環境 Γ と式 M を取り、結果として置換 S と型 τ を返す。$\mathcal{W}$ が成功した場合、型判定 $S(\Gamma) \triangleright M : \tau$ が成り立つ。$\mathcal{W}$ の定義は省略する。アルゴリズム $\mathcal{V}$ では $\mathcal{W}$ と同様、関数適用の型を推論するときに単一化アルゴリズム $\mathcal{U}$[18] を使う。$\mathcal{U}$ は型式の対の集合を受け取り、単一化が可能なとき最も一般的な単一化代入 (の 1 つ) を返し、単一化が可能でないときは失敗するアルゴリズムである。$\mathcal{V}$ は次の点で $\mathcal{W}$ と異なる。

- 引数として与えられる式はマークや、ダミー、カーソル式を含んでいる。
- アルゴリズム $\mathcal{W}$ は置換と型のペアを返すが、アルゴリズム $\mathcal{V}$ は置換と型に加え、カーソル位置における型環境とカーソル式の型の組の 3 つ組の集合を返す。
- $\mathcal{V}$ では $\mathcal{W}$ と同様、単一化が失敗した場合について明記していないが、$\mathcal{V}$ は $\mathcal{W}$ とは違い単一化が失敗してもアルゴリズム $\mathcal{V}$ 全体は失敗とはならず、単一化が失敗した場合の結果の組を取り除くだけである。

$$
\begin{array}{lcl}
\mathcal{V}(\Gamma,\ D) = \text{case } D \text{ of} & & \\
\quad \llcorner & \Rightarrow & \text{let } N = \{count(\Gamma(x)) \mid x\ \in\ dom(\Gamma)\} \\
& & \quad\ T = \{g(x) \mid x\ \in\ N\} \\
& & \text{in } \{(\emptyset,\ \tau,\ (\Gamma,\ \tau)) \mid \tau \in T\} \\
\quad \mid D* & \Rightarrow & \text{let } \{(S_0,\ \tau_0\ , C_0), \ldots, (S_i,\ \tau_i,\ C_i)\} = \mathcal{V}(\Gamma,\ D) \\
& & \quad\ \{\tau_{0,0}, \ldots, \tau_{0,k}\} = at(\tau_0) \\
& & \qquad\qquad \vdots \\
& & \quad\ \{\tau_{i,0}, \ldots, \tau_{i,m}\} = at(\tau_i) \\
& & \text{in } \{(S_0,\ \tau_{0,0},\ C_0), \ldots, (S_0,\ \tau_{0,k},\ C_0), \\
& & \qquad\qquad \vdots \\
& & \quad\ (S_i,\ \tau_{i,0}, C_i), \ldots, (S_i,\ \tau_{i,m},\ C_i)\} \\
\quad \mid \lambda x.D & \Rightarrow & \text{let } \{(S_0,\ \tau_0,\ C_0), \ldots, (S_i,\ \tau_i,\ C_i)\} \\
& & \qquad\qquad = \mathcal{V}(\Gamma\{x : \alpha\},\ D) \quad (\alpha \text{ fresh}) \\
& & \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) \\
& & \quad\ \{(S_{2,0},\ \tau_{2,0},\ C_{2,0}), \ldots, (S_{2,i},\ \tau_{2,i},\ C_{2,i})\} \\
& & \qquad\qquad = \mathcal{V}(S_1(\Gamma),\ D) \\
& & \quad\ S_{3,j} = \mathcal{U}\{(S_{2,j}(\tau_1),\ \tau_{2,j} \to \alpha_j)\} \quad (\alpha_j \text{ fresh}) \\
& & \qquad\qquad (j \in \{0, \ldots, i\}) \\
& & \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) \\
& & \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) \\
& & \quad\ \{(S_{2,0},\ \tau_{2,0},\ C_{2,0}), \ldots, (S_{2,i},\ \tau_{2,i},\ C_{2,i})\} \\
& & \qquad\qquad = \mathcal{V}(S_1(\Gamma)\{x : Cls(S_1(\Gamma),\ \tau_1)\},\ D) \\
& & \text{in } \{(S_{2,j} \circ S_1,\ \tau_{2,j},\ C_{2,j}) \mid j \in \{0, \ldots, i\}\}
\end{array}
$$

$$
\begin{array}{lcl}
count(\tau_1 \to \tau_2) & = & count(\tau_2) + 1 \\
count(\alpha) & = & 0 \\
count(int) & = & 0
\end{array}
\qquad
\begin{array}{lcl}
at(\tau_1 \to \tau_2) & = & \{\tau_1 \to \tau_2\} \cup at(\tau_2) \\
at(\alpha) & = & \{\alpha\} \\
at(int) & = & \{int\}
\end{array}
$$

$$
\begin{array}{lcl}
g(n+1) & = & \alpha \to g(n) \quad (\alpha \text{ fresh}) \\
g(0) & = & \alpha \quad (\alpha \text{ fresh})
\end{array}
$$

図 4. 型推論アルゴリズム $\mathcal{V}$

マーク式、ダミー式、およびカーソル式を含む式 D は直感的には複数の式 M を表す。型推論アルゴリズム $\mathcal{V}$ はそれら 3 つの構成要素を適切な方法によって具体化し、複数の推論結果を返す。任意の式を表すダミー式 [] は任意の型を持つので、ダミー式 [] の型として $\mathcal{V}$ は新しい型変数を返す。マーク式 D^* は D に引数としてダミー式を 0 回以上与えた場合の連続した関数適用を表し、$\mathcal{V}$ は D の型それぞれに対して、その型自身とその型の結果部分の型を再帰的にすべて取り出す計算をする。この計算は、図 4 の関数 at によって行われる。

アルゴリズム $\mathcal{V}$ では推論結果の数を減らすために、スコープに含まれる有効な変数の型 $\tau_1 \to \cdots \to \tau_k$($\tau_k$ は int 型か型変数) の $\tau_1, \ldots, \tau_k$ を新しい型変数 $\alpha_1, \ldots, \alpha_k$ で置き換え、$\alpha_1 \to \cdots \to \alpha_k$ に抽象化する。この処理はカーソル式の場合において関数 $count$ と関数 g によって行われる。関数 $count$ は引数に型 τ を取り、型 τ の引数部分の数を数える。例えば、 $count\ (int \to int \to int)$ の結果は 2 である。関数 g は引数に n をとり、$n+1$ 個の新しい型変数を矢印で繋いだ型を返す。例えば、$g(2)$ の結果は型 $\alpha_0 \to \alpha_1 \to \alpha_2$ であり、その中の型変数 $\alpha_0, \alpha_1, \alpha_2$ はそれぞれ異なる新しい型変数である。関数 $count$ と関数 g によりカーソル位置において有効な変数の型は引数の個数によってグループ化される。これにより同じ個数の引数を取る関数についての計算が 1 回にまとめて行われる。典型的な関数は数個の引数を取るので、この抽象化によって計算量は効果的に減少する。抽象化された部分の具体的な計算は 6 章の絞り込みの段階まで遅延される。

D^* の返す結果のそれぞれの 2 番目の要素は、D に対する推論結果それぞれの 2 番目の要素 $\tau_0, \ldots, \tau_i$ について、引数部分を 0 回以上取り除いて得られる型である。これは D に引数としてダミー式を 0 回以上与えた状況を反映している。$\mathcal{V}$ の結果の内、それぞれ 3 番目の要素は直感的に言えば、カーソル位置における型環境とカーソル式の型からなる組である。

アルゴリズム $\mathcal{V}$ ではカーソル位置における型環境とカーソル式の型に対して推論途中では置換を適用せず、最後に $\mathcal{V}$ の結果に対して置換を適用する。これは計算量を減らすことを目的としている。最終的なカーソル位置の型環境と型は、$\mathcal{V}(\Gamma,\ D)$ の結果の中の組 $(S, \tau, (\Gamma_{\lrcorner}, \tau_{\lrcorner}))$ それぞれについて、置換 S を $\Gamma_{\lrcorner}$ と $\tau_{\lrcorner}$ に対してそれぞれ適用することで得られる。

アルゴリズム $\mathcal{V}$ による型推論の例として、4 章の最後に述べた例における補完結果の式 D に対し、型環境 $\emptyset$ の下で型推論アルゴリズム $\mathcal{V}$ を適用する場合の計算の過程を示す。つまり $\mathcal{V}(\emptyset,\ D)$ の計算過程を示す。式 D は自由変数を持っていないので型環境は空としてよい。$\mathcal{V}$ がカーソル式 $\lrcorner_{\mathtt{f}}$ に辿り着いたとき、カーソル位置の型環境 $\Gamma_{\lrcorner}$ が以下のように得られる。

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

アルゴリズム $\mathcal{V}$ はカーソル式 $\lrcorner_{\mathtt{x}}$ の型を推論したときに次の 2 つの 3 つ組からなる集合を返す。

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

1 つ目の 3 つ組の 2 番目の要素の型 $\beta_1 \to \beta_2$ 及び 2 つ目の 3 つ組の 2 番目の要素の型 β_3 にそれぞれ関数 at を適用することにより、$\mathcal{V}$ は $(\lrcorner_{\mathtt{x}})^*$ について以下の 3 つの 3 つ組からなる集合を返す。

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

次に、$\mathtt{xa}\ (\lrcorner_{\mathtt{x}})^*$ に対して $\mathcal{V}$ が適用され、以下の 3 つの単一化が行われる。

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

ここで $(int \to \alpha_i) \to \alpha_i\ (i = 0, 1, 2)$ は関数 $\mathtt{xa}$ の型 $\forall\alpha.(int \to \alpha) \to \alpha$ の束縛されている型変数 α を新しい型変数 $\alpha_0, \alpha_1, \alpha_2$ でそれぞれ具体化した型であり、この具体化は関数 $\mathcal{W}$ の中で行われる。また、$\gamma_1, \gamma_2, \gamma_3$ はアルゴリズム $\mathcal{V}$ の関数適用の場合の定義に従って生成される新しい型変数である。

単一化はすべて成功し、 $\mathcal{V}$ は $\mathtt{xa}\ (\lrcorner_{\mathtt{f}})^*$ について以下の 3 つの 3 つ組からなる集合を返す。

$$\{(S_1, \alpha_0, (\Gamma_{\lrcorner}, \beta_1 \to \beta_2)),\ (S_2, \alpha_1, (\Gamma_{\lrcorner}, \beta_1 \to \beta_2)),\ (S_3, \alpha_2, (\Gamma_{\lrcorner}, \beta_3))\}$$

最終的にこれら3つの3つ組が$\mathcal{V}(\emptyset,\ D)$の結果となる。最後に、置換$S_1$, S_2, S_3を$\beta_1 \to \beta_2$, $\beta_1 \to \beta_2$, β_3へと適用し、カーソルの型として3つの型$int \to \alpha_0$, $\beta_1 \to int \to \alpha_1$, $int \to \alpha_2$を得る。同様に置換S_1, S_2, S_3を$\Gamma_\lrcorner$に適用することでカーソル位置の型環境として$\Gamma_\lrcorner$を得る。また、アルゴリズム$\mathcal{V}$の結果の3つ組の3番目の要素の中の最初の要素はすべて同じである。次の章ではこの例を用いて候補の絞り込みの例を示す。

6　候補の絞り込み

この章では最終的にプログラマーに提示する補完候補を作成する。まず、絞り込む前の候補はカーソル位置の型環境$\Gamma_\lrcorner$の定義域にある変数全てとする。それら候補をカーソル式の型と単一化を行うことにより絞り込む。どのカーソル式の型とも単一化が成功しなかった変数は候補から取り除かれる。候補の変数の型がτである時は、それぞれのカーソル式の型とτの単一化を行い、変数の型が$\forall\alpha_1 \ldots \alpha_k.\tau$という多相型である場合には、具体化した型$[\beta_1/\alpha_1] \ldots [\beta_k/\alpha_k]\tau$ $(\beta_1, \ldots, \beta_k$ fresh$)$とそれぞれのカーソル式の型で単一化を行う。また、カーソル式の型はすべて単相型である。最後に、現在入力中の綴りと候補の綴りが前方一致するものが最終的な候補となる。

5章で述べた型推論の例の結果を用いて絞り込みの例を示す。候補は$int \to \alpha_0$と$\beta_1 \to int \to \alpha_1$と$int \to \alpha_2$のうちのどれかと単一化可能な型を持っていなければならない。カーソル位置における型環境は$\Gamma_\lrcorner = \{\texttt{xa} : \forall\alpha.(int \to \alpha) \to \alpha,\ \texttt{yy} : \forall\alpha.\alpha \to \alpha,\ \texttt{xc} : int\}$となっており、有効な変数は xa, yy, xc の3つある。よって、単一化には$3 \times 3 = 9$通りの組み合わせがある。

$$
\begin{array}{ll}
\mathcal{U}\{(int \to \alpha_0,\ (int \to \alpha_5) \to \alpha_5)\}, & \mathcal{U}\{(int \to \alpha_0,\ \alpha_6 \to \alpha_6)\}, \\
\mathcal{U}\{(int \to \alpha_0,\ int)\}, & \mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ (int \to \alpha_5) \to \alpha_5)\}, \\
\mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ \alpha_6 \to \alpha_6)\}, & \mathcal{U}\{(\beta_1 \to (int \to \alpha_1),\ int)\}, \\
\mathcal{U}\{(int \to \alpha_2,\ (int \to \alpha_5) \to \alpha_5)\}, & \mathcal{U}\{(int \to \alpha_2,\ \alpha_6 \to \alpha_6)\}, \\
\mathcal{U}\{(int \to \alpha_2,\ int)\} &
\end{array}
$$

2番目、4番目、5番目、8番目の単一化が成功し、結果として2つの候補 xa と yy が得られる。候補 xa は入力中の綴り x と前方一致するが、yy は一致しないので最終的な候補集合は {xa} となる。

7　アルゴリズムの性質

この章ではアルゴリズムのまとめとその性質について述べる。変数名補完問題1を解くアルゴリズムは次のようにまとめることができる。

アルゴリズム 1 接頭辞式Pおよび型環境Γに対し5章で述べた型推論アルゴリズム$\mathcal{V}$を適用した結果を以下のように置く。

$$\{(S_0,\ \tau_0',\ (\Gamma_\lrcorner,\ \tau_0)), \ldots, (S_i,\ \tau_i',\ (\Gamma_\lrcorner,\ \tau_i))\} = \mathcal{V}(\Gamma,\ cmp\ P)$$

この時、補完候補の集合Vを以下のように計算する。

$$
\begin{aligned}
V = \bigcup_{j \in \{0,\ldots,i\}} \Big\{ \{ x \mid x \in dom(S_j(\Gamma_\lrcorner)),\ & \\
& \text{単一化 } \mathcal{U}\{(S_j(\Gamma_\lrcorner)(x),\ S_j(\tau_j))\} \text{ が成功}, \\
& P \text{ の中のカーソル式 } \lrcorner_s \text{ が属性として持つ } s \text{ が } x \text{ の接頭辞である} \} \Big\}
\end{aligned}
$$

上記アルゴリズム1は以下の2つの性質を満たすと予想される。

予想 1 (健全性) アルゴリズム 1 によって得られた変数集合 V は問題 1 の条件 $\forall v \in V,\ \exists M,\ \exists \tau, \Gamma \triangleright M : \tau,\ (P,\ v) \in pre\ M$ を満たす。

予想 2 (完全性) 問題 1 の条件を満たす変数は全てアルゴリズム 1 によって得られる変数集合 V に含まれる。つまり、もし $\exists M,\ \exists \tau, \Gamma \triangleright M : \tau,\ (P,\ v) \in pre\ M$ ならば $v \in V$ である。

我々の試した限りにおいて上記 2 つの予想に反する例は見つかっていない。健全性は、型の制約に反する変数は全て候補から取り除かれることを保証する。この性質により、型の制約を考えない場合に比べて大幅に候補の数を減らすことができる。候補に漏れがあると不便であるため、候補となるべき変数はすべて候補になるということを保証する完全性は変数名補完においてとても重要な性質である。これらの 2 つの性質により変数名補完システムは実用に適したものとなる。

アルゴリズム $\mathcal{V}$ の計算量については、D^* の型を推論するときに関数型のネストの分だけ計算量が増えるが、通常は関数型のネストはあまり深くならない。D^* のネストの深さに関して指数的に計算量が増えるが、実際のプログラムではネストが深くなることはあまり無く、ネストの深さは実質的に定数と見なせる。さらにカーソル式の型を推論する場合において型を抽象化することで計算量を抑えている。

8 実装

提案手法に基づいて我々は Standard ML の小さなサブセットを対象として Emacs 上で変数名補完を行う lambda-mode という Emacs モードを開発した。Lambda-mode は web ページ `http://www.cs.ise.shibaura-it.ac.jp/complement/` から入手出来る。この章では、lambda-mode の実装について述べる。lambda-mode は変数名補完機能だけでなく簡易な字下げ機能も備えている。

変数名補完の処理は大きく次の 3 つに分けられる。最初にプログラマが文字を入力した時に候補の計算を行う。次にプログラマへ候補を提示する。最後にプログラマが選択した候補で補完を行う。

2 番目と 3 番目の処理については auto-complete モード [5] を利用する。auto-complete は文字を入力するごとに補完候補を計算する関数を呼び出し、候補の表示から補完までを行う。候補が存在するならば、ポップアップウィンドウに候補が表示され、プログラマは候補を選択するか、選択をせずに入力を続けることができる。もし候補がない場合は何もしない。図 5 はプログラマが文字 `x` を入力した時の lambda-mode の動作の様子である。

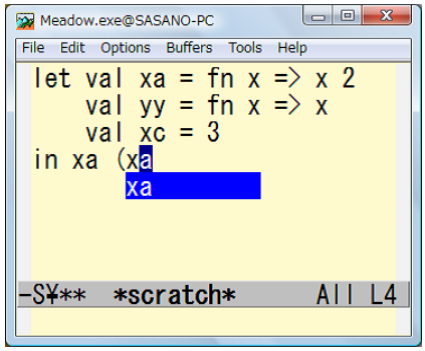

図 5. lambda-mode のスナップショット

lambda-mode は Standard ML のサブセットである図 6 の核言語を対象とする。核言語の字句は次のものとする。

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

$$
\begin{array}{lcll}
 & & & \\
start & := & exp & (1) \\
exp & := & appexp & (2) \\
 & | & \texttt{fn}\ id\ \texttt{=>}\ exp & (3) \\
appexp & := & atexp & (4) \\
 & | & appexp\ atexp & (5) \\
 & & & \\
 & & & \\
atexp & := & id & (6) \\
 & | & const & (7) \\
 & | & (exp) & (8) \\
 & | & \texttt{let}\ decseq\ \texttt{in}\ exp\ \texttt{end} & (9) \\
dec & := & \texttt{val}\ id\ \texttt{=}\ exp & (10) \\
decseq & := & dec\ decseq & (11) \\
 & | & \epsilon & (12)
\end{array}
$$

図 6. 核言語の構文規則

タイプライターフォントで書かれた字句はその綴り通りの文字列に対応する。id は変数名を表しその綴りを属性として持つ。$const$ は定数を表しその値を属性として持つ。ws は空白を表す。EOF は字句列の終端を表す。字句 id は英字の大文字と小文字の列であり、$const$ は 0 以外の数字で始まる 0 から 9 の数字の列か、0 である。$const$ を扱う演算子に+や-がある。ws はタブや空白、改行のことである。数字の列は int 型であり、演算子の+と-は $int \to int \to int$ 型の関数である。

字句解析器はプログラマーが入力している最中の字句の直前までを字句解析の対象とする。実装ではカーソル位置を現在入力している字句の直前まで移動し、字句解析が終了した後カーソル位置を戻している。字句解析器がカーソル位置までたどり着いた時は、カーソル位置を表す字句 $id_{\lrcorner}$ を返す。一度 $id_{\lrcorner}$ を返した後はそのことを記憶しておき、次に字句解析器が呼び出されたときからは EOF を返す。カーソルの移動によって字句解析の対象から外された入力途中の字句の綴りは候補絞り込みで使用するので記憶しておく。

構文解析ではプログラムテキストから部分的な構文木を作成する。構文解析は字句解析器を呼び出しながらカーソル位置まで行う。構文解析器は一般的な LR 構文解析器 [16] にいくつか変更を加えたもので、yacc と互換性のある kmyacc[8] の出力した構文解析表を元に作成した構文解析器を使用している。字句解析器が生成した字句列を EOF に辿りつくまで構文解析を行う。構文解析器が EOF を先読みしたとき、構文解析器は直ちにその時点でのスタックに積まれた終端記号・非終端記号と、その時までに構築された部分的な構文木を返して終了する。構文解析器はスタックに遷移状態に関する情報も持っているが、その後の処理で状態の情報は使わないのでそれら情報は返さない。

我々の実装では EOF を先読みした時点で直ちに終了するが、一般的な構文解析器は EOF を先読みした時点ではすぐには終了しない。直ちに終了する理由は、そうしなければ字句 $id_{\lrcorner}$ の後に字句がないものとして構文解析器は還元を行なってしまうからである。

2 章で述べたように変数を使用する箇所のみを補完の対象とする。その為に構文木の中の $id_{\lrcorner}$ の親ノードが $atexp$ であるかを調べることで、字句 $id_{\lrcorner}$ が変数の定義ではなく使用であることを確認する。もし、変数名の使用でない場合は変数名補完の処理を中断する。

具象構文木の補完を行うアルゴリズムは 4 章の式の補間関数 cmp に基づいて作成した。関数 cmp は接頭辞式 P を受け取り、補完された式 D を構築していたが、実装では明示的に接頭辞式 P に対応するものは作らず、代わりに一般的な構文解析器を用いて構文解析を行った。構文解析器がカーソル位置までたどり着いた時に構文解析を中断し、中断時までに作成された部分的な構文木と構文解析器のスタックを得る。それらを用いてマーク式とダミー式を含む完全な構文木を構築する。

構文解析の結果はスタック上に積まれていたシンボルの列と、そのシンボル列中の非終端記号を根に持つ部分的な構文木である。スタック上のシンボル列と構文規則を比較することで部分的な構文木を補完していき、完全な構文木を構築する。また、対象としている核言語では、スタック上の 1 番目と 2 番目のシンボルを比較すればどの構文規則を適用するかを決定できる [19]。

核言語ではスタックの 1 番目が $appexp$ であるときに適用できる規則は規則 2 と規則 5 の 2 つで

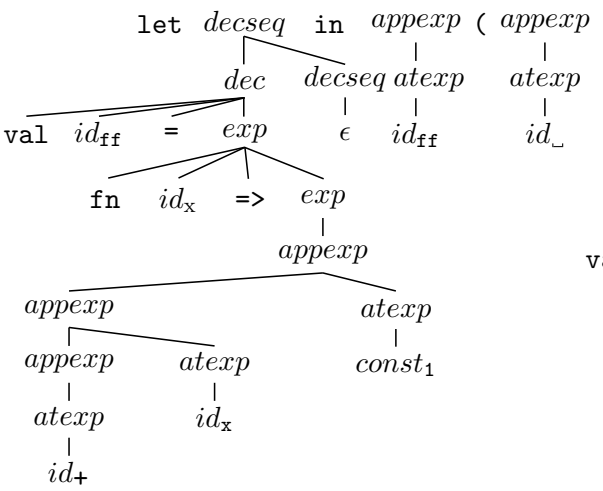

図 7. 構文解析の結果

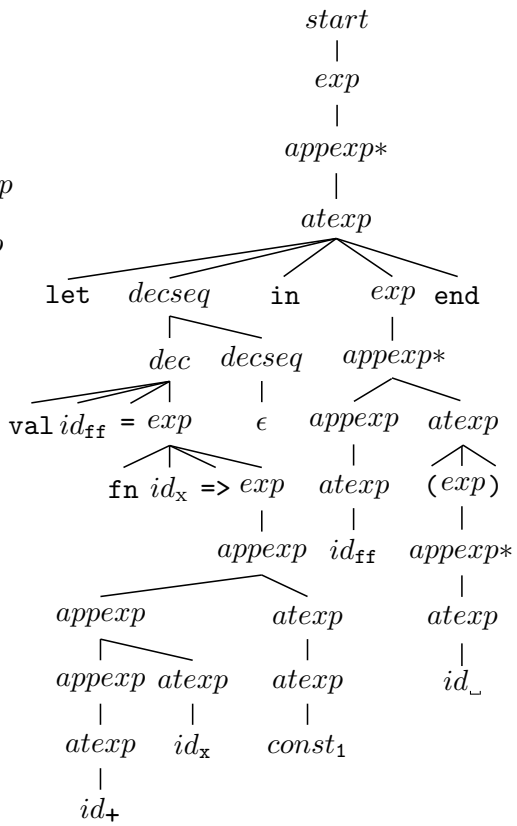

図 8. 補完の結果

ある。規則 5 は直接左再帰している規則である。左再帰がある構文規則で構文木補完を行った場合、補完により無限個の構文木を得ることになる。そこでマークノードを導入する。マークノードは直接左再帰を避けるために使う。もし、$appexp ::= appexp\ atexp$ のような直接左再帰の構文規則があった場合、まず規則 $appexp ::= appexp\ atexp$ を取り除き、全ての構文規則に含まれる $appexp$ を $mark$ に置き換える。そして、新しく規則 $mark ::= appexp$ を追加することで直接左再帰のない構文規則を得る。その構文規則は省略する。得られた構文規則を用いて構文木補完を行う場合、スタックの 1 番目が $appexp$ の時、それは $mark$ へと還元され、$mark$ は exp へと還元される。こうして、直接左再帰による無限ループを避けることができる。

構文木補完の例として次のプログラムを編集しているときに、文字 f を入力した場合を考える。

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

ここで、_はカーソル位置を表しており、+は $int \to int \to int$ 型の定数である。

文字 f を入力した直後に候補の計算が開始され、字句解析の結果として次の字句列を得る。

$$\texttt{let},\ \texttt{val},\ \texttt{ff},\ \texttt{=},\ \texttt{fn},\ id_x,\ \texttt{=>},\ id_+,\ id_x,\ const_1,\ \texttt{in},\ \texttt{ff},\ (,\ id_{\lrcorner},\ \text{EOF}$$

この字句列を構文解析することによって図 7 の構文木と以下のスタック上のシンボル列を得る。

$$\texttt{let},\ decseq,\ \texttt{in},\ appexp,\ (,\ appexp$$

構文木補完により図 8 の完全な構文木を得る。図 8 では、図を見やすくするために生成規則 $mark ::= appexp$ に対応するノードにアスタリスクを使っている。

$\mathcal{V}$ で型推論を行うために、具象構文木を 4 章で定義した式 D へと変換する。変換関数は省略する。

残りの型推論と絞り込みの処理については 5 章と 6 章に従って実装した。

lambda-mode は一文字入力するごとに候補の計算を最初から行う仕様になっているが、候補は実際に使用するのに十分短い時間でポップアップウィンドウに表示される。

9 実験

この章では Lambda-mode による補完候補の計算時間がプログラムの規模にどの程度影響を受けるかを調べる為に実験を行った。核言語で書かれた大規模なプログラムはないので、自動生成したプログラムを用いて候補の計算にかかる時間を計測した。

計測する時間はプログラマがキーを押してから候補が表示されるまでの時間とした。計測時の環境は、CPU が Intel Core i7 920、メモリ 6GB、 OS は Windows 7 Home Premium 64bit、そして Emacs は Meadow (GNU Emacs 22.2.1) を使用した。

Let 式や関数式、括弧についてそれぞれを様々な深さで入れ子にした場合について時間を計測した。関数型言語では let 式の入れ子が使われることが多いので、ここでは let 式を入れ子にした場合の結果を表 1 に示す。1 つの let 式において多くの宣言、例えば 30 個の宣言をすることがあるが、核言語では 1 つの let 式で 1 つの宣言しか出来ないため、1 つの let 式で複数の宣言をする場合は let 式の入れ子と考えた。表 1 の 100 個の宣言等は極端な例であることを考えると、入れ子の深さが増えている割には計算にかかる時間は増加していないと言える。外部ライブラリには多くの宣言があるが、外部ライブラリの解析は通常の型推論で良く、また事前に計算しておくことが出来る。将来的には外部ライブラリを扱えるようにする予定である。表には載せてはいないが他の構成要素の場合も let 式の場合と同様の結果であった。表 1 の中の、「同じ変数名」の列は全ての変数名が同じ名前を持っている場合の結果を示しており、「異なる変数名」は全ての変数名が異なる変数名を持っている場合の結果を示している。異なる変数名の場合の方が同じ変数名の場合より候補の数が多いので異なる名前の場合のほうが時間がかかる結果となっている。

		時間 (ミリ秒)	
		同じ変数名	異なる変数名
宣言の数	10	5	5
	50	26	30
	100	77	88

表 1. 変数宣言を入れ子にした場合の計算時間の比較

10 関連研究

これまでに変数名補完を備えた IDE は多く開発されてきた。それらのうちのいくつかは編集中のファイルに入力された単語に基づいた簡易な変数名補完機能を備えている。 他のモジュールやクラス等の中で定義された識別子に基づいて変数名補完を行うものもある。Visual Studio に搭載されている *Intellisense* や、Eclipse の *content assist* [2]、vim の *omni completion* のようにさらに高度な補完を行うものもある。それらは編集中のプログラムの文脈に合うものを候補として表示する。例えば intellisense と content assist は変数のスコープを考慮している。文脈を考慮することで変数名補完はより便利になってきてはいるが、我々の知る限り型推論がある言語において候補の絞込みに型情報を用いるものはない。以下にいくつかの IDE とその特徴を列挙する。

- Visual F#、Visual C++、および Visual C#は変数名補完機能を備えている。構文上書くことができる候補のみが表示され、変数のスコープも考慮されている。候補はポップアップウィンドウに表示され、一緒に候補の型情報も表示される。Eclipse の C++や Java 用のプラグインも Visual Studio と同様の変数名補完機能を備えている。

- Eclipse FP[4] は Haskell 用の Eclipse プラグインで変数名補完機能を備えているが、ローカル変数の補完は出来ず、またカーソル位置がどこにあっても候補が表示されるなど文脈を考慮をしていない。
- Leksah[9] は Haskell 用の IDE で変数名補完機能を備えているが、補完の対象となるのはインポートしたパッケージで宣言された変数のみで、編集中のプログラム内で宣言された変数は候補にはならない。
- Caml mode [1] と tuareg mode [12] は Caml と OCaml 用の Emacs モードで、変数名補完機能は備えていないが、caml mode は OCamlSpotter[11] により指定した変数の型やその宣言場所を表示する機能がある。
- Java Development Environment for Emacs [7] は Java 用の Emacs モードで、メンバー名補完機能は備えているが、変数名補完機能やクラス名補完機能はない。
- F#用の Emacs モード [6] は変数名補完機能は備えていない。
- OCaml Development Tools [10] は Eclipse の OCaml 用のプラグインで、変数名補完機能は備えていない。

上記の中で変数名補完機能を備えた IDE は型の制約を満たさない補完も許している。それらに比べて我々の変数名補完方式では候補の絞込みに型情報を利用し、型の制約を満たさない候補を取り除いている。

不完全なプログラムを対象とした型推論や式の生成に関する研究として以下のようなものがある。

- 型エラーを含む不完全なプログラムに対して型推論を行い、型エラーの原因となる、ある意味で必要最小限の箇所を特定する研究 [13] がある。本研究ではカーソル位置までに型エラーがないという前提をおいているが、将来的に型エラーを含んだプログラムを許す場合、型エラーに関連のある変数を補完対象から外すといったことが考えられる。
- 穴のあるラムダ式を対象とした計算体系を実現する為に型システムを構築した研究 [14] がある。ただし、型推論アルゴリズムはまだ与えられていない。本研究では穴を埋める操作を扱っておらず、直接的な関係はない。
- 型の制約のもとでの式の自動生成に関し、与えられた型を持つ式を１つ生成する Djinn[3] というツールや、与えられた型を持つ式の中の最も小さい式を１つ生成するツール [15] が作成されている。これらの研究においては、生成する式の型が与えられるが、我々の変数名補完システムにおいては、接頭辞式が与えられ、補完によって（ダミー式とマーク式を含む）式を生成しており、生成される式の型は与えられていない。

11　まとめと今後の課題

本論文では暗に型付けされた関数型言語を対象とした変数名補完手法を提案した。カーソル位置以前のプログラムが完全に与えられているという仮定の下で変数名補完問題を定義し、それを解くアルゴリズムを与えた。基本的な考えはカーソル式とダミー式を使うことと構文木補完において無限ループが起きる箇所でマーク式を使うことである。それらの考えに基づき補完候補計算のアルゴリズムを構築した。アルゴリズムの性質に関する予想 1,2 の証明は今後の重要な課題である。また、我々は今後、提案手法に基づいて Haskell や Standard ML、OCaml などの言語を対象とした変数名補完システムを開発する予定である。実際の言語を対象とするにあたり、以下のような点を考慮する予定であり、これらは将来の課題とする。

- 現在の実装では文字が入力されるたびに最初から補完候補の計算を行っているので無駄が多い。構文解析や型推論の結果を保存しておいて再利用することが考えられる。計算の再利用に関する研究は多くあり、それらを利用することで計算量を減らすことが出来ると考えられる。
- カーソル位置より前には構文エラーや型エラーがないという仮定をしているため、プログラムの先頭の方に小さいエラーがあるだけで変数名補完が働かなくなる。この仮定は強すぎるので構文解析時のエラー回復などの方法によってエラーのあるプログラムでも変数名補完が働くようにできると考えられる。
- Haskell では変数を束縛する前に使用することができ、また、ML では（相互）再帰関数の定義の中では束縛の前に変数を使用できる。我々の現在の枠組みではカーソル位置より後のプログラムテキストを使っていないので、それらのケースにそのままでは適用できない。問題定義を拡張することにより、そのような状況でも補完が行えるようになると考えられる。その他、パターンマッチング、中置演算子、モジュール、型注釈などの言語要素を扱えるように拡張する。

謝辞

本論文について大変有益な意見を下さった査読者の方々に感謝します。本研究の一部は科学研究費補助金の補助を得て行われた。

参考文献

[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] Fsharp mode. `http://fsharp-mode.sourceforge.net/`.

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

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

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

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

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

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

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

[14] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, Vol. 266, No. 1-2, pp. 249–272, 2001.

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

[16] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, Vol. 8, No. 6, pp. 607–639, 1965.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, No. 3, pp. 348–375, 1978.

[18] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

[19] 後藤拓実, 篠埜功. 多相型言語の変数名補完を行う Emacs モードの開発. 第 12 回プログラミングおよびプログラミング言語ワークショップ論文集, pp. 177–190, 2010.