ML-Style Multi-Abstraction Calculus with Type Inference Algorithm

Azza A. Taha

Department of Mathematics, Ain Shams University, Cairo, Egypt

Abstract: ML-style multi-abstraction calculus, $\lambda_{\text{limo}}$, is introduced as an extension of the Hindley-Milner type system. The calculus has a new multi-abstraction, a simultaneous application and a simultaneous explicit substitution. In comparison with some other multi-abstraction formalization, the calculus has the advantage of saving the usual $\alpha$ equivalence between all terms. The calculus can be used to represent contexts, where contexts in lambda calculus are lambda terms with holes. The calculus has a subject reduction property, is confluent and has a sound and complete type inference algorithm.

Keywords: Lambda Calculus, Hindley-Milner Type System, Contexts, Explicit Substitution, Unification, Type-Inference Algorithm

Introduction

The simply typed lambda calculus enriched with the let-expression is the core of most functional programming languages like ML (Harper, 2011) and Haskell (Thompson, 2011). One of the key properties of this type system is that it supports parametric polymorphism that allows a part of a program to be used with different types in different situations. It was first studied by Hindley (1969) in the field of combinatory logic and later independently by Milner (1978). This type system is often referred to as ML-style or Hindley-Milner type system. One of the key properties of this type system is that every well-typed term has a most general type. The computation of the most general type is called principal type. Algorithm W is a well-known type inference algorithm in the literature for the Hindley-Milner type system (Milner, 1978; Damas, 1985; Damas and Milner, 1982; Urban and Nipkow, 2009). The algorithm is based on Robinson’s unification Algorithm (Robinson, 1965). In this study, the calculus $\lambda_{\text{limo}}$ is introduced as an extension of the Hindley-Milner type system to include a multi-abstraction, a simultaneous application and an explicit simultaneous substitution. The multi-abstraction, $\mu(x_1,...,x_n).M$ abstracts a sequence of variables $x_1,...,x_n$ in $M$. The simultaneous application, $M_{\text{sim}}([N_1,...,N_n])$ applies all the arguments $N_1,...,N_n$ to $M$ simultaneously. The explicit simultaneous substitution, $M{\Rightarrow}_{x_1,...,x_n} = N_1,...,N_n$ has a number of substitutions rather than one substitution. It has the advantage of performing all of the substitutions, $<x_i,N_i>_{1 \leq i \leq n}$, in parallel which reduces reduction steps and consequently decreases programs execution time. The multi-abstraction of $\lambda_{\text{limo}}$ calculus resembles in some sense the simultaneous abstraction defined by Ruhrberg (1996) but it has the advantage of saving the usual $\alpha$-equivalence between all terms. The author introduced a simple simultaneous calculus where the usual lambda-abstraction over a single variable is replaced by abstraction over a set of variables, terms are applied to records assigning terms to variables. This system overcomes the strict ordering requirements of the standard $\lambda$-calculus, as a consequence of the un-ordered variables, the system partially lost the $\alpha$-equivalence.

The $\lambda_{\text{limo}}$ calculus can be used to represent contexts; terms with some holes. For example, writing [.] for a hole, the term $(\lambda y.[.])$ is a context. The distinctive feature of contexts is filling its hole with a term, in which some free variables may get captured and become bound. For example, if the hole of the context $(\lambda y.[.])$ is filled with the term $x+y$, the term $(\lambda y.x+y)$ is obtained in which the variable $y$ becomes bound by $\lambda y$. This feature is called variable capturing or capturing substitution. Capturing-substitution is different from the usual capture-avoiding substitution of the lambda calculus which avoids variable capturing by applying the $\alpha$-conversion rule when it is necessary. The main problem in computing with contexts, in the framework of the lambda calculus, is when the $\beta$-reduction is directly performed on contexts, some terms may be lost. To see this problem, consider the context $\lambda x.(\lambda y.[.])(2+z)$, if the hole of this context is filled with the term $x+y$, the term $\lambda x.(\lambda y.x+y)(2+z)$ is obtained. By $\beta$-reducing this term, we get the term

Keywords: Lambda Calculus, Hindley-Milner Type System, Contexts, Explicit Substitution, Unification, Type-Inference Algorithm

© 2019 Azza A. Taha. This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license.
\[ \lambda x.x + (2 + z) \]. On the other hand, if the \( \beta \)-reduction is first performed in the term \( \lambda x.(\lambda y.[.])(x + y) \), we get the term \( \lambda x.y.[.] \). Then, by filling the hole of this term by the term \( x + y \) we get different result \( \lambda x.x + y \). That is, the \( \beta \)-reduction and filling holes are not consistent and consequently the confluence property is lost. It is remarkable that, the correct result is \( \lambda x.x + (2 + z) \). The term \( 2 + z \) is lost when the \( \beta \)-reduction is performed before filling the hole.

This problem can be solved by using \( \lambda_{\text{lcmt}} \) terms to represent the context \( \lambda x.(\lambda y.[.])(x + y) \) as:

\[ \lambda x.(\lambda y.x \bullet ([x, y]))(x + y) \]

where, the hole is represented by the variable \( X \) applied to the sequence of variables \( x, y \) using the new simultaneous application \( \bullet \). The type of the variable \( X \) is \((\{\tau_1, \tau_2\}) \Rightarrow u \), where the type variables \( \tau_1, \tau_2 \) are the types of the variables \( x, y \) respectively (the variables intended to get captured after filling the hole). To fill the hole of the term \( \lambda x.(\lambda y.(x \bullet ([x, y])))(x + y) \) with the term \( x + y \), the new multi-abstraction term \( \mu(x, y, x + y) \) is substituted for \( X \) in this term, to get:

\[ \lambda x.(\lambda y.(\mu(x, y, x + y))(x + y)) \bullet ([x, y]) \]

The underline subterm can be reduced by the reduction rules of \( \lambda_{\text{lcmt}} \) to the term \( x + y \). Therefore, the above term is reduced to \( \lambda x.(\lambda y.x + y)(x + y) \), which can farther be reduced to \( \lambda x.x + (2 + z) \) as required.

On the other hand, if the \( \beta \)-reduction is first performed in the term \( \lambda x.(\lambda y. X \bullet ([x, y]))(x + y) \), we get the term:

\[ \lambda x.X \bullet ([x, (2 + z)]) \]

The term \( 2 + z \) is stored in the variable \( y \), of \((x, y)\), in case if the \( \beta \)-reduction is performed before filling the hole.

Filling the hole of this term with the term \( x + y \) is then achieved by substituting the new abstraction \( \mu(x, y, x + y) \) for \( X \) to get:

\[ \lambda x.\mu(x, y, x + y) \bullet ([x, (2 + z)]) \]

The underline subterm can be reduced by the reduction rules of \( \lambda_{\text{lcmt}} \) to the term \( x + (2 + z) \). Therefore, the above term is reduced to \( \lambda x.x + (2 + z) \) as required. In this representation, hole filling and \( \beta \)-reduction commute, that is the formalization is confluent.

Note that, the order of the variables in the simultaneous application as well as in the multi-abstraction are given according to their lexicographic order, e.g.,:

\[ \lambda x.\lambda y.(\mu(x, y, z)).M \bullet ([x, y, z]) \]

and

\[ \lambda x_2.\lambda x_1.(\mu(x_1, x_2)).M \bullet ([x_1, x_2]) \]

There is also another problem in computing with context. Consider the context \( \lambda x.[.] \). If the hole of this context is filled with \( x \) we get the term \( \lambda x.x \). This term is \( \alpha \)-equivalent to \( \lambda y.y \). It is desired to find a representation of a context which is \( \alpha \)-equivalent to \( \lambda x.[] \) and which is at the same time when filled with \( x \) becomes \( \lambda y.y \). This is achieved by using \( \lambda_{\text{lcmt}} \) terms to represent the context \( \lambda x.[] \) as \( \lambda x.(\lambda y.\mu([x])) \) which is \( \alpha \)-equivalent to \( \mu([y]) \). Filling the hole in each of these two contexts with the variable \( x \) is achieved by substituting the new abstraction \( \mu(x, x) \) for the hole variable \( X \) in these two contexts to get the two terms \( \lambda x.\mu(x, x) \bullet ([x]) \) and \( \lambda y.\mu(x, y) \bullet ([y]) \) respectively. These two terms are respectively reduced by the reduction rules of \( \lambda_{\text{lcmt}} \) to the two \( \alpha \)-equivalent terms \( \lambda x.x \) and \( \lambda y.y \) respectively as required.

There has been several contributions to the field of formalization and computation with contexts, e.g., Hashimoto (1998); Hashimoto and Ohori (2001); Bognar (2002); Sands (1998); Sato et al. (2002); Taha et al. (2002) and Tobisawa (2015); among these only the system given by Hashimoto (1998) is an ML-style polymorphic type system with a sound and complete type inference algorithm. This system is based on the simply typed system Hashimoto and Ohori (2001), in which hole filling and \( \beta \)-reduction rule can be combined under a restriction that a term containing a hole cannot be \( \beta \)-reduced.

Using \( \lambda_{\text{lcmt}} \) to represent contexts, holes of contexts are represented by the normal variables, the type of these variables includes the type of the variables that intended to get captured after filling the hole. The usual lambda abstraction is used to abstract hole variables. Filling holes is represented by the usual application and the new multi-abstraction which represents the variables intended to get captured after filling the hole. The \( \beta \)-reduction and filling holes can be combined without restrictions. A context is a first-class object; it can be passed as argument and returned from functions. Filling holes is an explicit operation; it can be computed within the system. In this representation, there is no need to use any specific operations for context; the ordinary lambda abstraction and application can be used together with the new multi-abstraction and simultaneous application as indicated above. The \( \lambda_{\text{lcmt}} \) calculus has a subject reduction property, is confluent and has a sound and complete type inference algorithm. The calculus also has an explicit substitution in the sense of Bloor and Rose (1995). The explicit substitution describes the details of the computation process; it distributes the substitution through terms to be finally evaluated at variables. The explicit substitution is important to make the process of executing the reduction explicit as a part of the calculus rather than implicit at the meta-level. Moreover, the
explicit substitution of the $\lambda_{lmem}$ has a number of substitutions rather than one substitution. It has the advantage of performing all the substitutions simultaneously which reduces some of the reduction steps and consequently reduces programs execution time. Last, in comparison with some other formalization the ordinary $\alpha$-equivalence is defined for all terms.

Since holes of contexts are place holders for some unknown terms, filling holes with terms has the effect of dynamic binding which enables a given program to interact with other programs dynamically. This mechanism can be useful in systems like distributed programming and mobile computing. We refer to Hashimoto and Ohori (2001) for a detailed explanation for contexts applications. The calculus can also be used in other applications that needs to abstract a sequence of variables at a time, needs to apply all the arguments simultaneously and at the same time needs to keep the $\alpha$-conversion between terms.

**Materials and Methods**

**The Calculus**

**Types**

The types of the $\lambda_{lmem}$ calculus are defined by the following grammar.

$$\tau, \upsilon ::= \alpha \quad ; \text{type variable}$$

$$b \quad ; \text{basic type (e.g., int, \ldots)}$$

$$| \tau \Rightarrow \upsilon \quad ; \text{function type}$$

$$| (| \tau |_n \Rightarrow \upsilon |) \quad ; \text{multi-function type}$$

The multi-function type $(| |\tau |_n \Rightarrow \upsilon |)$ is an abbreviation of $(| \tau_1, \ldots, \tau_n |_n \Rightarrow \upsilon |)$ and $n \in N$, where $(| \tau |)$ is an $n$-tuple. (The braces $( |, | )$ are used instead of the usual $(, )$ which are used to group terms). The type schemes are defined as:

$$\sigma ::= \tau \mid \forall \alpha \sigma$$

A type scheme is a type that may contain quantification of type variables at the outermost position only. Let $\alpha, \beta, \gamma, \alpha_1, \beta_1, \gamma_1, \ldots$ range over type variables, $\tau, \upsilon, \zeta, \tau_1, \upsilon_1, \zeta_1, \ldots$ over types and $\sigma_1, \sigma_2, \ldots$ over type schemes. The type scheme $\sigma = \forall \alpha_1 \forall \alpha_2 \ldots \forall \alpha_n \tau$ is abbreviated as $\forall \alpha_1 \alpha_2 \ldots \alpha_n \cdot \tau$. In this type, the type variables $\alpha_1, \ldots, \alpha_n$ are said to be bound in $\sigma$. Type variables that occur in $\tau$ and are not bound are said to be free in $\sigma$. We write $FTV(\sigma)$ for the set of free type variables of $\sigma$. If $FTV(\tau) = \emptyset$, then $\tau$ is said to be a monotype. A type scheme is closed if it has no free type variables. We assume $\alpha$-equivalence on $\forall$ abstraction. For instance, $\forall \alpha. \alpha$ is $\alpha$-equivalent to $\forall \beta. \beta$.

The set FTV, is defined inductively over $\lambda_{lmem}$ types as:

$$FTV(\alpha) = \{ \alpha \}$$

$$FTV(b) = \emptyset$$

$$FTV(\tau \Rightarrow \upsilon) = FTV(\tau) \cup FTV(\upsilon)$$

$$FTV(| \tau |_n \Rightarrow \upsilon |) = FTV(\tau) \cup FTV(\upsilon)$$

$$FTV(\forall \alpha. \sigma) = FTV(\sigma) \setminus \{ \alpha \}$$

where, $FTV(\tau) = FTV(\tau_1) \cup \ldots \cup FTV(\tau_n)$.

A type substitution, $S$, of types for type variables is a function that maps from type variables to types $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$, where each $\tau_i/\alpha_i$ is called a component of $S$ and all $\alpha_i, 0 \leq i \leq n$ are distinct. This function will be shortened to $[\tau/\alpha]$. The domain of $S$, $dom(S)$, is $\{ \alpha_1, \ldots, \alpha_n \}$. The codomain of $S$, cod(S), is $[\tau_1, \ldots, \tau_n]$.

The set of variables in a substitution $S$, Var(S), is $dom(S) \cup FTV(\tau)$. The empty substitution is denoted by $[\ ]$. The composition of two substitutions $S$ and $R$, denoted by $(S \circ R)$ or $SR$. If $S$ is a type scheme, then $(S \circ R)\alpha, S\alpha R\alpha, SR\alpha$, is the substitution which has the same effect as applying $R$ then $S$ to $\alpha$. The substitution $SR\alpha$ is a new substitution constructed from $S$ and $R$ by first modifying $R$ by applying $S$ to its components and then adding the components of $S$ not found in $R$. Therefore, if $S = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ and $R = [\upsilon_1/\beta_1, \ldots, \upsilon_n/\beta_n]$, then $SR\alpha$ is a new substitution:

$$[S\alpha_1/\beta_1, \ldots, S\alpha_n/\beta_n, \tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

if this substitution should be adjusted by deleting any binding $S\alpha_i/\beta_i$ for which $S\alpha_i=\beta_i$ and any binding $\tau_i/\alpha_i$ for which $\alpha_i \in \{ \beta_1, \ldots, \beta_n \}$. Finally, the substitution $S$ is said to be idempotent if and only if $SS = S$.

If $\sigma$ is a type scheme and $S$ is type substitution $[\tau/\alpha]$, then $S\sigma$ is a type scheme obtained by replacing each free occurrence of $\alpha$ in $\sigma$ by $\tau$. $S\sigma$ is defined inductively on $\lambda_{lmem}$ types as:

$$[\tau/\alpha] \alpha_i = \tau_i$$

$$[\tau/\alpha] \beta = \beta, \text{ if } \beta \notin [\tau]$$

$$[\tau/\alpha] b = b$$

$$[\tau/\alpha] (\upsilon_1 \Rightarrow \upsilon_2) = [\tau/\alpha] \upsilon_1 \Rightarrow [\tau/\alpha] \upsilon_2$$

$$[\tau/\alpha] \left( (| \tau |_n \Rightarrow \upsilon |) \right)_n = [\tau/\alpha] \upsilon$$

$$[\tau/\alpha] (\forall \beta \sigma) = \begin{cases} [\forall \beta \tau/\alpha] \sigma, & \text{if } \beta \notin [\alpha] \\ [\forall \gamma \tau/\alpha] [\gamma/\beta] \sigma, & \text{otherwise.} \end{cases}$$

DOI: 10.3844/jcssp.2019.745.757

A type scheme $\sigma_i$ is an instance of $\sigma_j$ if $\sigma_i = \sigma_j\tau$ for some substitution $\tau$. A type scheme $\sigma_i = \forall \lambda i_1, \lambda i_2 \tau_i$, written as $\sigma_i \models \sigma_j$, if there is a substitution $\tau$ with domain $[\lambda i_1, \ldots, \lambda i_n]$ s.t. $\tau_i = \delta\tau$ and $\beta_\tau, 1 \leq i \leq n$, are not free in $\sigma_i$. For example, $\int \Rightarrow \text{int}$ is a generic instance of $[\forall \alpha. \alpha \Rightarrow \text{int}]$, written as $\forall \alpha. \alpha \Rightarrow \text{int}$, where $\alpha = \int$. The relation $\models$ is reflexive and transitive.

The following Lemma shows that $\models$ is preserved by substitutions.

**Lemma 1 (Damas, 1985)**

If $\sigma_i \models \sigma_j$ then for any substitution $S, S\sigma_i \models S\sigma_j$.

**Terms**

Assume $V$ is a countable infinite set of term variables $x, y, x_1, y_1, x_2, y_2, \ldots$ We use capital letters $X, Y, \ldots$ for hole variables; variables of type $(\overline{\tau}) \Rightarrow \Rightarrow$ The terms of $\lambda\text{enc}$ are defined by the following grammar:

$$
M, N ::= x \quad \text{variable} \\
| \lambda x. M \quad \text{abstraction} \\
| MN \quad \text{application} \\
| \text{let } x := M \text{ in } N \quad \text{let polymorphism} \\
| M < x := N \quad \text{explicit simultaneous substitution} \\
| \mu_n(\overline{\tau}). M \quad \text{multi-abstraction} \\
| M \bullet \overline{N} \quad \text{simultaneous application}
$$

where, the variables $x, x_1, x_2, \ldots, x_r$ range over $V$. The term $\mu_n(\overline{\tau}). M$ is an abbreviation of $\mu_n(x_1, \ldots, x_n). M$, the term $M \bullet \overline{N}$ is an abbreviation of $M((N_1, \ldots, N_n))$, and the term $M < x := N \Rightarrow \overline{N}$ is an abbreviation of $M < x := N_1, \ldots, x_r := N_r \Rightarrow$ where $n \in N$. It is clear that the $x_i$'s in $\mu_n(\overline{\tau}). M$ and in $M < x := N \Rightarrow \overline{N}$ should be distinct. When it is clear, the index $n$ in $\mu_n(\overline{\tau}). M$ and $M \bullet \overline{N}$ will sometimes be omitted.

The set of free variables in a term $M$, $FV(M)$, is defined inductively on $\lambda\text{enc}$ term $M$ as:

$$
FV(x) = \{x\} \\
FV(\lambda x. M) = FV(M) - \{x\} \\
FV(MN) = FV(M) \cup FV(N) \\
FV(\text{let } x := M \text{ in } N) = FV(N) - \{x\} \cup FV(M) \\
FV(M < x := \overline{N} \Rightarrow \overline{N}) = FV(M) - \{\overline{X}\} \\
FV(\mu_n(\overline{\tau}).M) = FV(M) - \{\overline{X}\} \\
FV(M \bullet \overline{N}) = FV(N_1) \cup \ldots \cup FV(N_r)
$$

where, $FV(\overline{N}) = FV(N_1) \cup \ldots \cup FV(N_r)$.

Two terms $M$ and $N$ are $\alpha$-equivalent, written as $M \equiv N$, if they are identical except for the renaming of bound variables bound by $\lambda$, by $x$, in $M < x := N \Rightarrow \overline{N}$, by $x$ in $x := M \text{ in } N$ or by $x$ in $\mu_n(\overline{\tau}). M$. This equivalence is defined inductively over $\lambda\text{enc}$ term $M$ as:

1. $x = x$ 
2. $\lambda x. M = \lambda y. N$ if $FV(M) \equiv FV(N)$, for some $\alpha \equiv \overline{\alpha}$
3. $M N P Q$ if $M = P$ and $N = Q$ 
4. let $x := M$ in $N = let y := M'$ in $N'$ if $M = M'$ and $N[x/y] = N'[x/y]$, for some $\alpha \equiv \overline{\alpha}$
5. $\lambda x. M \equiv \lambda x. N \Rightarrow \lambda \tau. \alpha \equiv \overline{\alpha}$ \text{where, } \alpha = \overline{\alpha} 
6. $\mu_n(\overline{\tau}). M \equiv \mu_n(\overline{\tau}). N$ \text{if, } $M[x_1 = z_1, \ldots, x_r = z_r] = N[x_1 = z_1, \ldots, x_r = z_r]$ 
7. $M \bullet \overline{N}_1 \equiv P \bullet \overline{N}_2$ \text{if, } $M = P$ and $N_i = Q_i$ for $i = 1, \ldots, n$

Note that:

- In 5 above, the number of $\overline{\tau}$ and $\overline{\alpha}$ variables in $M < x := N \Rightarrow P < y := Q \Rightarrow$ should be equal
- If $M = N$, then $FV(M) = FV(N)$

In the definition of the $\alpha$-equivalence above, the capture-avoiding meta-level simultaneous substitution $M[\overline{N}/x]$ is obtained by substituting $N_i$ for $x_i, 1 \leq i \leq n$ in $M$ simultaneously. The substitution $M[\overline{N}/x]$ is defined inductively on $\lambda\text{enc}$ term $M$ as:

1. $x[\overline{N}/x] = N_i$ 
2. $y[\overline{N}/x] = y$, if $y \not\in \{\overline{X}\}$ 
3. $\lambda x. M[\overline{N}/x] = \lambda y. M[N/y]$, if $y \not\in \{\overline{X}\}$ and $y \not\in FV(\overline{N})$ 
4. $(MP)[\overline{N}/x] = (M[N/x])(P[\overline{N}/x])$ 
5. $let y := M \text{ in } P[\overline{N}/x] = let y := M[N/x] \text{ in } P[\overline{N}/x]$, if $y \not\in \{\overline{X}\}$ and $y \not\in FV(\overline{N})$ 
6. $M < y := P \Rightarrow \overline{N} = M[N/x] < y := P[\overline{N}/x]$, if $\{\overline{X}\} \cap FV(\overline{N}) = \emptyset$ and $\{\overline{X}\} \cap FV(\overline{N}) = \emptyset$
If $\Gamma$ is a type assignment, the closure of a type scheme $\sigma$ with respect to $\Gamma$ is $\text{Clos}(\Gamma,\sigma) = \forall \alpha. \alpha$, where $\alpha = \text{FTV}(\sigma) - \text{FTV}(\Gamma)$. For example, if $\Gamma$ is $\{ x : \forall \alpha. \alpha \to \gamma \}$ and $\sigma$ is $\beta \to \gamma$, then $\text{Clos}(\Gamma,\sigma) = \forall \beta. \beta \to \gamma$.

Assume that $\Gamma$ contains at most one assumption for each variable $x$. The type assignment $\Gamma_x$ stands for removing any assumption $x$ from $\Gamma$. The notation, $\overline{x: \sigma}$, is used as an abbreviation for $x_1 : \sigma_1, \ldots, x_n : \sigma_n$. The typing rules of $\lambda_{\text{lexic}}$ that are used to derive a typing judgment of the form $\Gamma \vdash M : \sigma$, are defined in Fig. 1.

A term $M$ is called typable (or well-typed) if and only if there exist $\Gamma$ and $\tau$ such that we can derive $\Gamma \vdash M : \tau$ by using the $\lambda_{\text{lexic}}$ typing rules.

**Lemma 2 (Milner, 1978)**

Let $S$ be a substitution, $\Gamma$ be a type assignment and $\tau$ be a type, then: $S\Gamma \vdash M : \sigma$ holds, then: $S\Gamma \vdash M : S\sigma$, where $S' = S[\beta \mapsto \alpha]$. $\overline{\alpha} = \text{FTV}(\sigma) - \text{FTV}(\Gamma)$ and $\overline{\beta}$ are fresh type variables.

**Lemma 3 (Damas and Milner, 1982)**

If $S$ is a substitution and $\Gamma \vdash M : \sigma$ holds, then: $S\Gamma \vdash M : S\sigma$ holds.

**Lemma 4 (Damas and Milner, 1982)**

If $\sigma_1 \supset \sigma_2$ and $\Gamma, x : \sigma_1 \vdash M : \sigma_0$ then:

$$\Gamma, x : \sigma_1 \vdash M : \sigma_0$$

Fig. 1: The typing rules for $\lambda_{\text{lexic}}$
Reductions

The reduction rules, \( \rightarrow_{\text{term}} \), of \( \lambda_{\text{term}} \) are the union of the following reduction rules, \( \rightarrow_{\text{Var}} \), \( \rightarrow_{\text{Abs}} \), \( \rightarrow_{\text{App}} \), \( \rightarrow_{\text{let}} \), and \( \rightarrow_{\text{s}} \). The reduction rule \( \rightarrow_{\text{i}} \) is defined by the rule:

\[
(\lambda) (\lambda x.M) N \rightarrow_{\text{i}} M < x \Leftarrow N >
\]

The reduction rule \( \rightarrow_{\text{o}} \) is defined by the rule:

\[
(let) \ let x = N in M \rightarrow_{\text{o}} M < x \Leftarrow N >
\]

The reduction rule \( \rightarrow_{\text{n}} \) is defined by the rule:

\[
(m) (\mu \tau. M) \cdot (\{ N \}) \rightarrow_{\text{n}} M < x \Leftarrow N >
\]

Note that, The order of the variables \( x_i \) is important and there is no reduction for the term \( (\mu \tau. M) \cdot (\{ N \}) \), when \( n \neq m \).

The reduction rule \( \rightarrow_{\text{r}} \) is defined by the following 6 rules:

\[
\begin{align*}
(x\text{Var}) & & x_i < x \Leftarrow N > \rightarrow_{\text{r}} N_i . \\
(gc) & & M < x \Leftarrow N > \rightarrow_{\text{r}} M, \text{if } \{ \tau \} \cap \text{FV}(M) = 0. \\
(x\text{Abs}) & & (\lambda y.M) < x \Leftarrow N > \rightarrow_{\text{r}} \lambda y.M < x \Leftarrow N >, \\
& & \text{if } y \notin \{ \tau \}, \text{ and } y \notin \text{FV}(N). \\
(x\text{App}) & & (M_i, M_{i'}) < x \Leftarrow N > \rightarrow_{\text{r}} (M_i, M_{i'}) \cdot (\{ N \}) \rightarrow_{\text{r}} (M_i, M_{i'}) \cdot (\{ N \}) \rightarrow_{\text{r}} (M_i, M_{i'}) \cdot (\{ N \}) \\
(xm\text{Abs}) & & (\mu \tau. M) < x \Leftarrow N > \rightarrow_{\text{rn}} \mu \tau. M < x \Leftarrow N > \\
& & \text{if } \{ \tau \} \cap \{ \tau \} = 0 \text{ and } \{ \tau \} \cap \text{FV}(N) = 0. \\
(xs\text{App}) & & (M \cdot (\{ N \}) \cdot (\{ N \}) \rightarrow_{\text{r}} (M \cdot (\{ N \}) \cdot (\{ N \}) \rightarrow_{\text{r}} (M \cdot (\{ N \}) \cdot (\{ N \})
\end{align*}
\]

Let \( R = \{ \lambda, \text{ let, } m, x \} \), we write \( M \rightarrow_{\text{r}} N \), if \( N \) is obtained from \( M \) by replacing a subterm \( M_i \) in \( M \) by \( N_i \) such that \( M_i \rightarrow_{\text{r}} N_i \), where \( r \in R \). The reduction \( \rightarrow_{\text{q,r,s}} \) is the union of the three reductions \( \rightarrow_{\text{q}}, \rightarrow_{\text{r}}, \text{ and } \rightarrow_{\text{s}} \), where \( q,r,s \in R \). The reflexive and transitive closure of the reduction \( \rightarrow_{\text{r}} \) is denoted by \( \rightarrow_{\text{r}}^{*} \).

Example 1

The term:

\[
(\lambda X.(\lambda x.\lambda y.X \cdot (| x,y |))z)(\mu (x,y).xy)
\]

is reduced by \( \lambda_{\text{term}} \) reduction rules as:

\[
\begin{align*}
(\lambda X.(\lambda x.\lambda y.X \cdot (| x,y |))z)(\mu (x,y).xy) \\
\rightarrow_{\text{o}} (\lambda X.(\lambda y.X \cdot (| x,y |))z)(\mu (x,y).xy) \\
\rightarrow_{\text{o}} (\lambda X.(\lambda y.X \cdot (| x,y |))z)(\mu (x,y).xy)
\end{align*}
\]

Properties of the Calculus

In this section, we show that \( \lambda_{\text{term}} \) has the subject reduction property (Theorem 1) and the confluence property (Theorem 2). The subject reduction property insures that the type is preserved by reduction and the reductions never introduce new free variables. The confluence property guarantees the uniqueness of the result if it exists.

Subject Reduction

The following proposition and Lemma are needed in the proof of the subject reduction theorem.

**Proposition 1 (Milner, 1978)**

If \( \Gamma \vdash M : \sigma \) is an instance of a provable typing judgment \( \Gamma \vdash M : \sigma \) then \( \Gamma \vdash M : \sigma' \) is also provable.

**Lemma 5 (Barendregt, 1992)**

1. If \( \Gamma \vdash M : \sigma \) then \( \text{FV}(M) \subseteq \text{dom} (\Gamma) \)
2. If \( \Gamma \vdash M : \sigma \) then \( \Gamma \vdash \text{FV}(M) \vdash M : \sigma \)

**Theorem 1 (Subject Reduction)**

If \( \Gamma \vdash M : \sigma \) and \( M \rightarrow_{\text{term}} N \), then \( \Gamma \vdash N : \sigma \).

**Proof**

By induction on the derivation of \( \Gamma \vdash M : \tau \) using Proposition 1 and Lemma 5.

Confluence

To show that the reductions \( \rightarrow_{\text{term}} \) is confluent, we first show that the reduction \( \rightarrow_{\text{s}} \) is confluent.

**Lemma 6**

The reduction rule \( \rightarrow_{\text{s}} \) on \( \lambda_{\text{term}} \) terms is neitherian and confluent.

750
Proof

To show that \(\rightarrow\), is noetherian, the length of a term \(M,|M|\), is introduced as:

1. \(|x| = 1\)
2. \(|MN| = |M|+|N|+1\)
3. \(|\lambda x.M| = |M|+1\)
4. \(|let\ x = M\ in\ N| = |M|+|N|+1\)
5. \(|M < x\mapsto N >\Rightarrow (|N|+1)|M|\), where \(|N| = |N_1|+…+|N_n|\)
6. \(|\mu x.(\overline{N})M| = |M|+n|\)
7. \(|M \bullet (\overline{N})| = |M|+|\overline{N}|+1\)

It can easily be verified that \(M \rightarrow N\) implies \(|M| > |N|\). Then, by checking the overlapping cases, it can easily be verified that \(\rightarrow\), is weakly-confluent and by Newman’s Lemma, the reduction rule \(\rightarrow\), is confluent.

A \(\lambda_{\text{closed}}\) term \(M\) is said to be x-normal if \(M\rightarrow x\text{-}N\) holds for no \(N\). From Lemma 6, it is clear that for any \(\lambda_{\text{closed}}\) term \(M\) there uniquely exist x-normal term \(N\) such that \(M \rightarrow x\text{-}N\). We will write \(x(M)\) to denote \(N\).

Next, the parallel reduction (Takahashi, 1989), \(\Rightarrow\) on x-normal \(\lambda_{\text{closed}}\) terms is defined as:

1. \(x \Rightarrow x\)
2. If \(M \Rightarrow N\), then \(\lambda x.M \Rightarrow \lambda x.N\)
3. If \(M_1 \Rightarrow M_2\) and \(N_1 \Rightarrow N_2\), then \((\lambda x.M_1)N_1 \Rightarrow x(M_1 < x := N_1 >)\)
4. If \(M_1 \Rightarrow M_2\) and \(N_1 \Rightarrow N_2\), then \(M_1N_1 \Rightarrow M_2N_2\)
5. If \(M \Rightarrow N\), then \(\mu(\tau).M \Rightarrow \mu(\tau).N\)
6. If \(M \Rightarrow N\) and \(P_i \Rightarrow Q_i, 1 \leq i \leq n\), then \((\mu(\tau).M) \bullet (\overline{P}) \Rightarrow x(N < x := Q >)\)
7. If \(M \Rightarrow N\) and \(P_i \Rightarrow Q_i, 1 \leq i \leq n\), then \(M \bullet (\overline{P}) \Rightarrow N \bullet (\overline{Q})\)
8. If \(M_1 \Rightarrow M_2\) and \(N_1 \Rightarrow N_2\), then \(let\ x := M_1\ in\ N_1 \Rightarrow x(N_2 < x := M_2 >)\)
9. Then, with each x-normal term \(M\), we associate an x-normal term \(\hat{M}\) as:
   1. \(x \Rightarrow x\)
   2. \((\lambda x.M) \Rightarrow \lambda x.M\)
   3. \(((\lambda x.M)N) \Rightarrow x(M < x := N >)\)
   4. \((MN) \Rightarrow M' \bullet N',\text{ if } M\text{ is not a }\lambda \text{ abstraction}\)
   5. \((\mu(\tau).M) \Rightarrow \mu(\tau).M\)
   6. \((\mu(\tau).M) \bullet (\overline{P}) \Rightarrow x(M' < x := P >)\)
7. \((M \bullet (\overline{P})) \Rightarrow M' \bullet (\overline{P})\) if \(M\) is not a multi-abstraction.
8. \((let\ x := M\ in\ N) \Rightarrow x(N < x := M' >)\)

It can easily be verified that:

1. The parallel reduction relation is reflexive, i.e., \(M \Rightarrow M\)
2. \(M \Rightarrow M'\) holds for any x-normal term \(M\)
3. If \(M_1 \Rightarrow M_2\), then \(x(M_1) \Rightarrow x(M_2)\)

Lemma 7

If \(M_1 \Rightarrow M_2\), then \(M_1 \Rightarrow x_{\text{closed}}\ M_1\).

Proof

By induction on the construction of \(M_1\).

Lemma 8 (Substitution Lemma)

If \(M\) is an x-normal term, then:

\[x(M < x := N > < y := Q >) = x(M < y := Q > < x := N < y := Q >)\]

where \(\overline{N}\) and \(\overline{Q}\) are \(\lambda_{\text{closed}}\) terms, \(\{\tau\} \cap \{\tau\} = \emptyset\) and \(\{\tau\} \cap \text{FV}\{\overline{P}\} = \emptyset\).

Proof

By induction on the construction of \(M\) such that \(M\) is x-normal term.

Lemma 9

If \(\lambda x.M_1\) \(\Rightarrow x(M_1)\) and \(x(N_i) \Rightarrow x(P_i),\, 1 \leq i \leq n\), then \(x(M_1 < x := N > < x := P >) \Rightarrow x(M_1 < x := P >)\).

Proof

By induction on the construction of \(x(M_1)\) such that \(x(M_1)\) is x-normal term.

Lemma 10

If \(M_1 \Rightarrow \lambda_{\text{closed}}\ M_2\), then \(x(M_1) \Rightarrow x(M_2)\)

Proof

By induction on the construction of \(M_1\).

Lemma 11

If \(M_1 \Rightarrow \lambda_{\text{closed}}\ M_2\), then \(x(M_1) \Rightarrow x(M_2)\).

Proof

Immediate from the reflexivity of \(\Rightarrow\).

751
Remark 1

From Lemma 10 and Lemma 11, we have:

If \( M_1 \xrightarrow{\lambda_{letmx}} M_2 \), then \( x(M_1) \Rightarrow x(M_2) \).

Lemma 12

The parallel reduction \( \Rightarrow \) on x-normal form is confluent.

Proof

We can easily verify that if \( M \Rightarrow N \), then \( N \Rightarrow M^* \), the confluence of \( \Rightarrow \) follows immediately from this fact.

Theorem 2 (Confluence)

The reduction \( \Rightarrow \) on \( \lambda_{letmx} \) terms is confluent, that is, if \( M \xrightarrow{\lambda_{letmx}} N \) and \( M \xrightarrow{\lambda_{letmx}} P \), then there is a term \( Q \) such that \( N \xrightarrow{\lambda_{letmx}} Q \) and \( P \xrightarrow{\lambda_{letmx}} Q \).

Proof

Suppose that \( M \xrightarrow{\lambda_{letmx}} N \) and \( M \xrightarrow{\lambda_{letmx}} P \), then from remark 1, we have: \( x(M) \Rightarrow x(N) \) and \( x(M) \Rightarrow x(P) \), from the confluence of \( \Rightarrow \), there exists \( Q \) such that \( x(N) \Rightarrow Q \) and \( x(P) \Rightarrow Q \). Since \( N \xrightarrow{\lambda_{letmx}} x(N) \) and \( P \xrightarrow{\lambda_{letmx}} x(P) \), we have \( N \xrightarrow{\lambda_{letmx}} Q \) and \( P \xrightarrow{\lambda_{letmx}} Q \).

The Proposed Type Inference Algorithm

A Type inference algorithm decides whether a given term \( M \) has a type or not; it takes \( M \) and \( \Gamma \) as inputs and produces a principal (most general) type (Principal types is different from principal typing. An algorithm for principal typing takes just \( M \) as input and gives both \( \Gamma \) and \( \sigma \) as outputs. Many languages have principal types but not principal typing (Jim (1996)) for \( M \) if it exists.

It is known that Hindley-Milner typability is decidable, i.e., there is an algorithm which computes the principal type scheme of any term. One of the best well known type inference algorithm in the literature for Hindley-Milner type system is algorithm \( W \) (Milner, 1978; Damas, 1985; Damas and Milner, 1982; Urban and Nipkow, 2009). Given \( M \) and \( \Gamma \), algorithm \( W \) finds a substitution \( S \) and a type \( \tau \) such that’s \( \Gamma \vdash M : \sigma \).

The type scheme \( \sigma \) is called a principal type scheme, Damas and Milner (1982), of \( M \) under \( \Gamma \) if and only if:

1. \( \Gamma \vdash M : \sigma \) holds
2. If \( \Gamma \vdash M : \sigma' \) holds, then \( \sigma \succ \sigma' \)

Algorithm \( W \) depends on Robinson’s unification algorithm, (Robinson, 1965), which takes a pair of types and either returns a substitution \( S \) or fail, where \( S \) unifies the pair of types.

The Unification Algorithm, Unify

The unification algorithm, Unify, is defined for \( \lambda_{letmx} \) types as in Fig. 2. Item 5 in this algorithm catches all cases that fail if none of the previous cases apply, e.g., Unify \((\tau \Rightarrow \nu, b)\), Unify \((\tau \Rightarrow \nu_1(\nu_2, \ldots, \nu_n)) \Rightarrow \tau_j\), etc. This means that, Unify \((\tau, \nu)\) fails if \( \tau \) and \( \nu \) are two different non type variables. It is clear that:

\[\text{Unify}(\tau, \nu) \circ S \text{ fails if Unify}(\tau, \nu) \text{ fails.}\]

Fig. 2: Algorithm Unify: \( \tau \times \nu \Rightarrow S \)

752
Example 2

1. Unify \( (\langle a \rangle \Rightarrow \text{int}, \langle \text{int} \rangle \Rightarrow \beta) = [\text{int} / \beta, \text{int} / a] \)
2. Unify \( (\alpha \Rightarrow a, b \Rightarrow \alpha) \) fails.

The following two Theorems can easily be verified.

**Theorem 3 (Soundness of Unify)**

If \( \text{Unify}(\tau, \upsilon) = S \), then \( S\tau = St\).

**Theorem 4 (Completeness of Unify)**

If \( V \) unifies \( \tau \) and \( \upsilon \), then \( \text{Unify}(\tau, \upsilon) \) returns \( S \) and there is another substitution \( R \) such that \( V = RS \).

**The Type Inference Algorithm, \( \hat{W} \)**

In this study, algorithm \( W \) is extended to the terms of \( \lambda_{\text{bons}} \). The extended algorithm is called \( \hat{W} \). Algorithm \( \hat{W}(\Gamma, M) \) is defined inductively on \( \lambda_{\text{bons}} \) term \( M \) as in Fig. 3.

In algorithm \( \hat{W} \), \( x : \text{Clos}(S\Gamma, S\upsilon) \) is an abbreviation of: \( x_1 : \text{Clos}(\Gamma_1, S\upsilon_1), \ldots, x_n : \text{Clos}(\Gamma_n, S\upsilon_n) \). When any of the algorithm \( \hat{W} \) conditions are not met, the algorithm fails.

**Example 3**

\[ \hat{W} \left( (x :: \text{int} \Rightarrow \text{int}, x :: \text{int} \Rightarrow \beta) = (V, \text{int}) , \text{ where } V = \text{Unify} \left( (\langle \text{int} \rangle \Rightarrow \text{int}, \langle \text{int} \rangle \Rightarrow \beta) \right) \text{ and } \beta \text{ is a fresh type variable.} \]

**Soundness of \( \hat{W} \)**

Soundness of \( \hat{W} \) shows that the algorithm never yields any results that are not correct. In other words, any type scheme derives by \( \hat{W} \) is derivable by the \( \lambda_{\text{bons}} \) type inference rules.

**Theorem 5 (Soundness of \( \hat{W} \))**

If \( \hat{W}(\Gamma, M) \) succeeds with \( (S, \tau) \), then there is a derivation of \( S\Gamma \vdash M : \tau \).

**Proof**

By induction on the structure of \( \lambda_{\text{bons}} \) term \( M \). We consider only the following cases, other cases can be verified similarly:

1. Case \( M \) is \( x \) and \( \hat{W}(\Gamma, x :: \forall \alpha. \tau) \) succeeds with \( \left( \| \right), \tau[\beta / a] \), where \( \beta \) are fresh type variables. From the (var) typing rule, we have:

\[ \Gamma, x :: \forall \alpha. \tau \vdash x :: \forall \alpha. \tau \]

Since \( \forall \alpha. \tau \Rightarrow \tau[\beta / a] \), then from the (inst) typing rule we have the following judgment as required:

\[ \Gamma, x :: \forall \alpha. \tau \vdash x :: \tau[\beta / a] \]

2. Case \( M \) is \( P \) in \( N \) and \( \hat{W}(\Gamma, \text{let } x = P \text{ in } N) \) succeeds with \( (S_\tau, \upsilon) \), where \( (S_\tau, \upsilon) \) is \( S_\tau, \upsilon \) and \( \cdot \) differ from \( \hat{W}(\Gamma, P) \) and \( \hat{W}(\Gamma, x : \text{Clos}(S\Gamma, \tau)) \).

From the induction hypothesis, we have:

\[ S_\tau \Gamma + P : \tau_1 \text{ and } (1) \]
\[ S_\tau S_\upsilon \Gamma, x : \text{Clos}(S_\Gamma, \tau_1) \vdash N : \tau_1 \text{ (2)} \]

Let \( S_\tau' = S_\tau \left[ \beta / a \right] \), where \( \beta \) are fresh type variables and \( \hat{W}(\Gamma, P) \) and \( \cdot \) differ from \( \hat{W}(\Gamma, \text{let } x = P \text{ in } N) \). Then, from Lemma 2, judgment (2) above and the fact that \( S_\tau' S_\upsilon = S_\tau' S_\upsilon \), because \( S_\tau \) differ from \( S_\tau \) only on bound variables of \( S_\tau \Gamma \), we get:

\[ S_\tau' S_\upsilon \Gamma, x : \text{Clos}(S_\Gamma', S_\tau, \tau_1) \vdash N : \tau_1 \text{ (3)} \]

Applying the (typeGen) typing rule to judgment (1) and since \( \hat{W}(\Gamma, P) \) and \( S_\tau' S_\upsilon \Gamma, x : \text{Clos}(S_\Gamma', S_\tau, \tau_1) \vdash N : \tau_1 \) are definitions of \( \text{Clos}(S\Gamma, \tau_1) \) and judgment (4), we get:

\[ S_\tau S_\upsilon \Gamma + P : \forall \alpha. \tau_1 \text{ (4)} \]

From the definition of \( \text{Clos}(S\Gamma, \tau_1) \) and judgment (4), we get:

\[ S_\tau S_\upsilon \Gamma + P : \text{Clos}(S\Gamma, \tau_1) \text{ (5)} \]

From Lemma (3) and Judgment (5), we get:

\[ S_\tau' S_\upsilon \Gamma + P : \text{Clos}(S_\Gamma', S_\upsilon, \tau_1) \text{ (6)} \]

From Lemma (2) and judgment (6), we have:

\[ S_\tau' S_\upsilon \Gamma + P : \text{Clos}(S_\Gamma', S_\upsilon, \tau_1) \text{ (7)} \]

Applying the (let) typing rule to judgments (7) and (3), we get:

\[ S_\tau S_\upsilon \Gamma + \text{let } x = P \text{ in } N : \tau_2 \text{ (8)} \]

Finally, by noting that \( S_\tau' S_\upsilon = S_\tau S_\upsilon \), We get the following required judgment:

\[ S_\tau S_\upsilon \Gamma + \text{let } x = P \text{ in } N : \tau_2 \text{ (9)} \]

3. Case \( M \) is \( P \cdot ([\tau]_n) \) and \( \hat{W}(\Gamma, P \cdot ([\tau]_n)) \) succeeds with \( \left( \forall \alpha_0 \ldots \alpha_n \cdot S_\upsilon \cdot \forall \beta \right) \) where:
\( \hat{W} (\Gamma, x) = \left( \left\lfloor \frac{x}{\beta} \right\rfloor \beta' \right) \) where \( \Gamma (x) = \forall \beta' \) and 
\( \beta \) are fresh type variables.

\( \hat{W} (\Gamma, \lambda x.M) = (S, S\beta \Rightarrow \tau) \) where \( (S, \tau) = \hat{W} (\Gamma, x : \beta, M) \) and 
\( \beta \) is a fresh type variable.

\( \hat{W} (\Gamma, MN) = \left( V \circ S_2 \circ S_1, \beta' \right) \) where \( (S_2, \tau_2) = \hat{W} (\Gamma, M) \), 
\( (S_2, \tau_2) = \hat{W} (\Gamma, N) \), \( V = \text{Unify} (S_2 \tau_2, \tau_2 \Rightarrow \beta) \), 
and \( \beta' \) is fresh type variable.

\( \hat{W} (\Gamma, let \ x = M \ in \ N) = (S_2 \circ S_1, \tau_2) \) where \( (S_2, \tau_2) = \hat{W} (\Gamma, M) \) and 
\( (S_2, \tau_2) = \hat{W} (\Gamma, N) \).

\( \hat{W} (\Gamma, M \ < \emptyset \ := \emptyset \ N) \) = \( (S' \circ S_1, \tau_1) \) where \( (S_1, \tau_1) = \hat{W} (\Gamma, N) \) and 
\( (S_1, \tau_1) = \hat{W} (\Gamma, N) \), \( S' = \hat{W} (S_1, \tau_1, \ldots, S_n, \tau_n) \), \( M \rightarrow \). 
\( \beta' \) are fresh type variables.

\( \hat{W} (\Gamma, \mu(\emptyset) \ N) \) = \( (V \circ S_1 \circ \ldots \circ S_n, \beta' \) \) where \( (S_n, \tau_n) = \hat{W} (\Gamma, M) \) and 
\( (S_n, \tau_n) = \hat{W} (\Gamma, N) \), \( V = \text{Unify} (\Gamma, n \Rightarrow \beta) \), 
and \( \beta' \) is a fresh type variable.

From the induction hypothesis, we have the 
judgments:

\( (S, \tau) = \hat{W} (\Gamma, P) \) and \( (S_1, \nu_1) = \hat{W} (\Gamma, \tau_1) \), \( (S_2, \nu_2) = \hat{W} (\Gamma, \tau_2) \), \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \). \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \). 
and \( V = \text{Unify} (\Gamma, n \Rightarrow \beta) \) and \( \beta' \) is a fresh type variable.

From Lemma 4 and judgment (11), since 
\( S \Gamma \vdash S \Gamma \vdash \beta \) we also have the following derivations:

\( S \Gamma \vdash N_1 : \nu_1, \ldots, S \Gamma \vdash N_n : \nu_n \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \).

From Lemma 3 and judgments (10) and (12), we get:

\( S \Gamma \vdash P : \tau \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \).

\( S \Gamma \vdash N_i : \nu_i, \ldots, S \Gamma \vdash N_n : \nu_n \) \( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \).

Applying the (nApp) typing rule to judgments (13) and since:

\( (S_n, \nu_n) = \hat{W} (\Gamma, \tau_n) \).

We get the required judgment \( V S_n \ldots S \Gamma \vdash P \bullet (\left\lfloor \beta \right\rfloor \} : V \mathcal{B} \).

Note that, if \( S \Gamma \vdash M : \tau \), then \( S \Gamma \vdash M : \text{Clos} (S \Gamma, \tau) \) also holds, where \( \text{Clos} (S \Gamma, \tau) \) is called the type scheme computed by \( \hat{W} \) for \( M \) (Damas and Milner, 1982).

Completeness of \( \hat{W} \)

Completeness of \( \hat{W} \) ensures that the algorithm addresses all possible inputs and does not miss any. In other words, any derivable type scheme is an instance of that computed by \( \hat{W} \).
The restriction of a substitution $S$ to a set $A$ of type variables, $S|A$, is defined by the set $\{\sigma : \exists \alpha \in S \land \alpha \in A\}$. If $S$ and $R$ are two substitutions such that for every $\alpha \in \text{dom}(S) \cap \text{dom}(R)$, $S\alpha = R\alpha$, the simultaneous composition Damas (1985), of $S$ and $R$, $(S \oplus R)$, is defined by:

$$(S \oplus R)\alpha = \begin{cases} S\alpha & \text{if } \alpha \in \text{dom}(S) \\ R\alpha & \text{otherwise} \end{cases}$$

Lemma 13

If $\tilde{W}((\Gamma, M)) = (S, \tau)$, then:

1. $\text{Var}(S) \subseteq \text{FTV}((\Gamma, \cdot)) \cup \text{New}$
2. $\text{FTV}(\tau) \subseteq \text{FTV}((\Gamma, \cdot)) \cup \text{New}$

where, New is the set of new type variables introduced by $\tilde{W}((\Gamma, M))$.

Proof

By induction on the structure of the $\lambda_{\text{conc}}$ term $M$.

Lemma 14 (Lee and Yi, 1998)

If $\Gamma \succ \Gamma'$, then $\text{Clos}(\Gamma, \cdot) \succ \text{Clos}(\Gamma', \cdot)$.

Theorem 6. (Completeness of $\tilde{W}$)

Given $\Gamma$ and $M$, let $\Gamma'$ be an instance of $\Gamma$ and $\sigma$ a type scheme such that $\Gamma' \vdash M : \sigma$ then:

1. $\tilde{W}((\Gamma, M))$ succeeds
2. If $\tilde{W}((\Gamma, M)) = (S, \tau)$, then for some substitution $R, \Gamma' = RS\Gamma$ and $R\text{Close}(S\Gamma, \cdot) \succ \sigma$

Proof

By induction on the structure of the $\lambda_{\text{conc}}$ term $M$ following the same technique of Damas (1985) using Lemma 13 and Lemma 14.

Discussion

Although the representation of contexts and hole-filling using $\lambda_{\text{conc}}$ terms can also be represented as terms in the ordinary lambda calculus, the $\lambda_{\text{conc}}$ representation has the advantage of making contexts internal to the calculus which enables to write more clear and elegant programs with fewer reduction steps and then faster reductions. For instance, consider the context:

$$\bar{\lambda}x_1, \cdots, \bar{\lambda}x_n[X \cdot (x_1, \cdots, x_n)]$$

which is represented in $\lambda_{\text{conc}}$ as:

$$\bar{\lambda}x_1, \cdots, \bar{\lambda}x_n, \{X \cdot (x_1, \cdots, x_n)\}$$

To fill the hole $X$ with the term $N$, the multi-abstraction $\mu(\bar{x}).N$ is substituted for $X$ in this term to get:

$$\bar{\lambda}x_1, \cdots, \bar{\lambda}x_n, \left[\left(\mu(\bar{x}).N\right)\mathbf{\bullet} \left(\left(x_1, \cdots, x_n\right)\right)\right]M_1 \cdots M_n$$

(14)

This term can be represented as the lambda term:

$$\bar{\lambda}x_1, \cdots, \bar{\lambda}x_n, \left(\bar{\lambda}x_1, \cdots, \bar{\lambda}x_n, N\right)x_1 \cdots x_n$$

$$M_1 \cdots M_n$$

(15)

The $\lambda_{\text{conc}}$ representation, (14), has a number of advantages over the encoding using the ordinary lambda terms, (15):

1. Using both the usual lambda calculus’ abstraction and application together with the new multi-abstraction and simultaneous application makes the representation more clear; simple inspecting of the representation in (14), it is obvious that the term with the new multi-abstraction $\mu(\bar{x}).N$ replaced the hole and the variables after the new simultaneous application $\mathbf{\bullet}$ is used to store terms in case if the $\beta$-reduction is performed before filling the hole
2. The representation in (14) has also an advantage in terms of the number of reduction steps. To reduce the underline subterm of (15), we have to use $n$ $\beta$-reductions, before each of them we have to check if the $\alpha$-conversion rule in needed or not to avoid the unintended variable capturing. An $n$ meta-level substitutions is also needed after each $\beta$-reduction which also requires to check the necessity of the $\alpha$-conversion. Whereas, to reduce the corresponding underline subterm of (14) only one $\rightarrow_m$ reduction step is needed. Then, only one checking for an $\alpha$-conversion and a sequence of the explicit simultaneous substitution. The number of reductions is clearly decreased and consequently the program execution time is reduced. To make it clear, consider the simple example:

$$\left(\mu(x, y, xy)\mathbf{\bullet} ((y + 1, z))\right)\rightarrow_m (xy)$$

$$x < x := y + 1, y := z >$$

$$\rightarrow_m x < x := y + 1, y := z > y < x := y + 1, y := z > x \rightarrow_m (xy), (y + 1, z)$$

whereas, in reducing the same term encoded as ordinary lambda term:

$$\left(\bar{\lambda}x, \bar{\lambda}y, \bar{x}y\right)(y + 1)z)\rightarrow_\beta (\bar{\lambda}x, \bar{\lambda}y, x)\rightarrow_\beta (\bar{\lambda}x, (y + 1)x)\rightarrow_\beta \bar{\lambda}x, (y + 1)x \rightarrow_\beta \bar{\lambda}x, (((y + 1)x)z \rightarrow_m \bar{\lambda}x, (((y + 1)x)z \rightarrow_\beta \bar{\lambda}x, (((y + 1)x)z \rightarrow_\beta (y + 1)z$$

Conclusion

ML-style multi-abstraction calculus, $\lambda_{\text{conc}}$, is introduced as an extension of the ML-style Hindley-
Milner type system. The calculus has a multi-abstraction, a simultaneous application and an explicit simultaneous substitution. The multi-abstraction abstracts a sequence of variables rather than just one variable at a time. The simultaneous application applies all of its arguments simultaneously and the explicit simultaneous substitution has a number of substitutions that can be performed in parallel which can decrease some reduction steps and consequently reduces programs execution time. The calculus has the advantage of saving the usual α-equivalence between all terms. The ML-style Hindley-Milner type system is chosen as a base of the calculus since its typability is decidable and since it supports parametric polymorphism that allows a part of a program to be instantiated with different types as needed in different situations. The calculus has a subject reduction property, is confluent and has a sound and complete type inference algorithm. The type inference algorithm infer the most general or principal types for terms.

The calculus can be used to represent contexts, where contexts are lambda terms with holes. The multi-abstraction and simultaneous applications of the calculus can also have several other useful applications that needs to abstract a sequence of variables at a time, needs to apply the arguments simultaneously and which needs at the same time to keep the usual α-conversion between terms. The calculus \( \lambda_{\text{time}} \) can serve as a theoretical basis for a polymorphic functional programming with multi-abstraction and simultaneous application. An implementation of the calculus and its type inference algorithm should be considered as a future work.

Acknowledgment

The author is grateful to Yukiyoshi Kameyama, the professor of Tsukuba University, for his helpful comments and suggestions to improve the early draft of this paper.

Ethics

The author confirms that this research is original and has not been published elsewhere and there is no ethical issues involved.

References


