Journal of Computer Science

Original Research Paper

ML-Style Multi-Abstraction Calculus with Type Inference Algorithm

Azza A. Taha

Department of Mathematics, Ain Shams University, Cairo, Egypt

Article history
Received: 01-03-2019
Revised: 12-04-2019
Accepted: 30-05-2019

Email: azzataha@sci.asu.edu.eg

Abstract: ML-style multi-abstraction calculus, λ_{letmx}, 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 α 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 λ_{letmx} 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; $\mu\left(x_1,\ldots,x_n\right).M$ abstracts a sequence of variables $x_1,\ldots,x_n$ in M. The simultaneous application; $M\bullet(|N_1,\ldots,N_n|)$ applies all the arguments $N_1,\ldots,N_n$ to M simultaneously. The explicit simultaneous substitution, $M<x_1,\ldots,x_n:=N_1,\ldots,N_n>$ has a number of substitutions rather than one substitution. It has the advantage of performing all of the substitutions, $<x_i := N_i>: 1 \le i \le n$, in parallel which reduces reduction steps and consequently decreases programs execution time. The multi-abstraction of λ_{letmx} calculus resembles in some sense the simultaneous abstraction defined by Ruhrberg (1996) but it has the advantage of saving the usual α-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 λ-calculus, as a consequence of the un-ordered variables, the system partially lost the α-equivalence.

The λ_{letmx} calculus can be used to represent contexts; terms with some holes. For example, writing [.] for a hole, the term (λy.[.]) 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 (λy.[.]) is filled with the term $x+y$, the term ($\lambda y.x+y$) is obtained in which the variable y becomes bound by λy. 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 α-conversion rule when it is necessary. The main problem in computing with contexts, in the framework of the lambda calculus, is when the β- 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 β-reducing this term, we get the term

$\lambda x.x+(2+z)$. On the other hand, if the β-reduction is first performed in the term $\lambda x.(\lambda y.[.])(2+z)$, we get the term $\lambda x.[.]$. Then, by filling the hole of this term by the term $x+y$ we get different result $\lambda x.x+y$. That is, the β-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 β-reduction is performed before filling the hole.

This problem can be solved by using λ_{letmx} terms to represent the context $\lambda x.(\lambda y.[.])(2+z)$ as:

$$\lambda x.\left(\lambda y.\ X \bullet \left(|x, y|\right)\right)(2+z)$$

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 $\left(|\tau_1, \tau_2|\right) \Rightarrow \upsilon$, where the type variables τ_1, τ_2 are the types of the variables x and y respectively (the variables intended to get captured after filling the hole). To fill the hole of the term $\lambda x.(\lambda y.\ X \bullet (|\ x, y\ |))(2+z)$ with the term $x+y$, the new multi-abstraction term $\mu(x,y).x+y$ is substituted for X in this term, to get:

$$\lambda x.(\lambda y.\underline{(\mu(x, y).x + y) \bullet (|\ x, y\ |)})\ (2+z)$$

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

On the other hand, if the β-reduction is first performed in the term $\lambda x.(\lambda y.\ X\bullet(|x, y|))(2+z)$, 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 β-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:

$$\lambda x.\underline{(\mu(x, y).x + y) \bullet (|\ x, (2\ +\ z)\ |)}$$

The underline subterm can be reduced by the reduction rules of λ_{letmx} to the term $x+(2+z)$. Therefore, the above term is reduced to the term $\lambda x.x+(2+z)$ as required. In this representation, hole filling and β-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.,:

$\lambda z.\lambda x.\lambda y.\left(\mu(x, y, z).M\right) \bullet \left(|\ x, y, z\ |\right)$ and

$\lambda x_2.\lambda x_1.\left(\mu(x_1, x_2).M\right) \bullet \left(|\ x_1, x_2\ |\right)$

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 α-equivalent to $\lambda y.y$. It is desired to find a representation of a context which is α-equivalent to $\lambda x.[.]$ and which is at the same time when filled with x becomes $\lambda y.y$. This is achieved by using λ_{letmx} terms to represent the context $\lambda x.[.]$ as $\lambda x.(X\bullet(|x|))$ which is α-equivalent to $\lambda y.(X\bullet(|y|))$. 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 $\lambda y.(\mu(x).x)\bullet(|y|)$ respectively. These two terms are respectively reduced by the reduction rules of λ_{letmx} to the two α-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 β-reduction rule can be combined under a restriction that a term containing a hole cannot be β-reduced.

Using λ_{letmx} 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 β-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 λ_{letmx} 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 Bloo 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 λ_{letmx} 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 α-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 α-conversion between terms.

Materials and Methods

The Calculus

Types

The *types* of the λ_{letmx} calculus are defined by the following grammar.

τ,υ :: $=\alpha$	; type variable
$\mid b$	; basic type (e.g., int, ···)
$\mid \tau \Rightarrow \upsilon$	; function type
$\mid (\mid \overline{\tau} \mid)_n \Rightarrow \upsilon$	; multi-function type

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

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

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, $\tau,\upsilon,\varsigma,\tau_1,\upsilon_1,\varsigma_1,\ldots$ over types and $\sigma,\sigma_1,\ldots$ over type schemes. The type scheme $\sigma = \forall\alpha_1\forall\alpha_2\ldots\forall\alpha_n.\tau$ is abbreviated as $\forall\alpha_1\alpha_2\ldots\alpha_n.\tau$, or $\forall\overline{\alpha}.\tau$. In this type, the type variables $\alpha_1,\ldots,\alpha_n$ are said to be bound in σ. Type variables that occur in τ and are not bound are said to be free in σ. We write FTV(σ) for the set of free type variables of σ. If FTV(τ) = $\emptyset$, then τ is said to be a monotype. A type scheme is closed if it has no free type variables. We assume α-equivalence on $\forall$ abstraction. For instance, $\forall\alpha.\alpha$ is α-equivalent to $\forall\beta.\beta$.

The set FTV, is defined inductively over λ_{letmx} types as:

$$\begin{aligned}
\mathrm{FTV}(\alpha) &= \{\alpha\}\\
\mathrm{FTV}(b) &= \emptyset\\
\mathrm{FTV}(\tau \Rightarrow \upsilon) &= \mathrm{FTV}(\tau) \cup \mathrm{FTV}(\upsilon)\\
\mathrm{FTV}\left((\mid \overline{\tau} \mid)_n \Rightarrow \upsilon\right) &= \mathrm{FTV}(\overline{\tau}) \cup \mathrm{FTV}(\upsilon)\\
\mathrm{FTV}(\forall\alpha,\sigma) &= \mathrm{FTV}(\sigma) - \{\alpha\}
\end{aligned}$$

where, $\mathrm{FTV}(\overline{\tau}) = \mathrm{FTV}(\tau_1) \cup \ldots \cup \mathrm{FTV}(\tau_n)$.

A *type substitution*, S, of types for type variables is a function that maps from type variables to types $\left[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\right]$, where each τ_i/α_i is called a component of S and all α_i, $0 \le i \le n$ are distinct. This function will be shortened to $\left[\overline{\tau/\alpha}\right]$. 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 $dom(S) \cup FTV(\overline{\tau})$. The empty substitution is denoted by [ ]. The *composition* of two substitutions S and R, denoted by ($S \circ R$) or SR. If σ is a type scheme, then $(S \circ R)\sigma$, $SR\sigma$ or $S(R\sigma)$, is the substitution which has the same effect as applying R then S to σ. The substitution SR 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 $S = \left[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\right]$ and $R = \left[\upsilon_1/\beta_1,\ldots,\upsilon_n/\beta_n\right]$, then SR is a new substitution:

$$\left[S\upsilon_1/\beta_1,\ldots,S\upsilon_n/\beta_n,\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\right]$$

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

If σ is a type scheme and S is type substitution $\left[\overline{\tau/\alpha}\right]$, then $S\sigma$ is a type scheme obtained by replacing each free occurrence of α_i in σ by τ_i. $S\sigma$ is defined inductively on λ_{letmx} types as:

$$\begin{aligned}
\left[\overline{\tau/\alpha}\right]\alpha_i &= \tau_i\\
\left[\overline{\tau/\alpha}\right]\beta &= \beta,\ if\ \beta \notin \{\overline{\alpha}\}\\
\left[\overline{\tau/\alpha}\right]b &= b\\
\left[\overline{\tau/\alpha}\right](\upsilon_1 \Rightarrow \upsilon_2) &= \left[\overline{\tau/\alpha}\right]\upsilon_1 \Rightarrow \left[\overline{\tau/\alpha}\right]\upsilon_2\\
\left[\overline{\tau/\alpha}\right]\left((\mid \overline{\upsilon_1} \mid)_n \Rightarrow \upsilon_2\right) &= \left(\mid [\overline{\tau/\alpha}]\overline{\upsilon_1} \mid\right)_n \Rightarrow \left[\overline{\tau/\alpha}\right]\upsilon_2\\
\left[\overline{\tau/\alpha}\right](\forall\beta.\sigma) &= \begin{cases}
\forall\beta.\left[\overline{\tau/\alpha}\right]\sigma, if\ \beta \notin \{\overline{\alpha}\}\\
\quad and\ \beta \notin FTV(\overline{\tau})\\
\forall\gamma.\left[\overline{\tau/\alpha}\right]\left([\gamma/\beta]\sigma\right),\\
otherwise.
\end{cases}
\end{aligned}$$

A type scheme σ_1 is an *instance* of σ_2 if $\sigma_1 = S\sigma_2$ for some substitution *S*. A type scheme $\sigma_1 = \forall\beta_1...\beta_m.\tau_1$ is a*generic instance* of $\sigma_2 = \forall\alpha_1\ldots\alpha_n.\tau_2$, written as $\sigma_2 \succ \sigma_1$, if there is a substitution *S* with domain $\{\alpha_1,\ldots,\alpha_n\}$ s.t. $\tau_1 = S\tau_2$ and $\beta_i, 1 \le i \le n$, are not free in σ_2. For example, *int* $\Rightarrow$ *int* is a generic instance of $\forall\alpha.\alpha \Rightarrow \alpha$, written as $\forall\alpha.\alpha \Rightarrow \alpha \succ \text{int} \Rightarrow \text{int}$, where $S = [int/\alpha]$. The relation $\succ$ is reflexive and transitive.

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

Lemma 1 (*Damas, 1985)*

If $\sigma_1 \succ \sigma_2$ then for any substitution *S*, $S\sigma_1 \succ S\sigma_2$.

Terms

Assume *V* is a countable infinite set of term variables $x,y,x_1,y_1,X,Y,X_1,Y_1,\ldots$. We use capital letters $X,Y,\ldots$ for hole variables; variables of type $(|\bar{\tau}|) \Rightarrow \upsilon$.The *terms* of λ_{letmx} are defined by the following grammar:

$$
\begin{array}{lll}
M,N :: = x & & ;\textit{variable}\\
\quad | \lambda x.M & & ;\textit{abstraction}\\
\quad | \text{MN} & & ;\textit{application}\\
\quad | \text{ let } x := M \textit{ in } N & & ;\textit{let polymorphism}\\
\quad | M < \overline{x := N} > & & ;\textit{explicit simultaneous substitution}\\
\quad | \mu_n(\bar{x}).M & & ;\textit{multi-abstraction}\\
\quad | M \bullet (|\bar{N}|)_n & & ;\textit{simultaneous application}
\end{array}
$$

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

The set of free variables in a term *M*, *FV*(*M*), is defined inductively on λ_{letmx} term *M* as:

$$
\begin{array}{lll}
FV(x) & = & \{x\}\\
FV(\lambda x.M) & = & FV(M)-\{x\}\\
FV(MN) & = & FV(M)\cup FV(N)\\
FV(\text{let } x := M \text{ in } N) & = & (FV(N)-\{x\})\cup FV(M)\\
FV(M < \overline{x := N} >) & = & (FV(M)-\{\bar{x}\})\cup FV(\bar{N})\\
FV(\mu_n(\bar{x}).M) & = & FV(M)-\{\bar{x}\}\\
FV(M\bullet(|\bar{N}|)_n) & = & FV(M)\cup FV(\bar{N})
\end{array}
$$

where, $FV(\bar{N}) = FV(N_1)\cup\ldots\cup FV(N_n)$.

Two terms *M* and *N* are α-equivalent, written as $M\equiv N$, if they are identical except for the renaming of bound variables bound by λ, by x_i in $M < \overline{x := N} >$, by *x* in let $x := M$ in *N* or by x_i in $\mu_n(\bar{x}).M$. This equivalence is defined inductively over λ_{letmx} term *M* as:

1. $x \equiv x$
2. $\lambda x.M \equiv \lambda y.N$ if $(M[z/x]) \equiv (N[z/y])$, for some $z \notin FV(MN)$
3. $MN \equiv PQ$ if $M \equiv P$ and $N \equiv Q$
4. let $x := M$ in $N \equiv$ let $y := M'$ in N' if $M \equiv M'$ and $N[z/x] \equiv N'[z/y]$, for $z \notin FV(NN')$
5. $M < \overline{x := N} > \equiv P < \overline{y := Q} >$, if there are distinct $i_1,\ldots,i_n$ s.t. $N_1 \equiv Q_{i_1},\ldots,N_n \equiv Q_{i_n}$ and $M[z/x] \equiv P\left[z_1/y_{i_1},\ldots,z_n/y_{i_n}\right]$, for some $\{\bar{z}\} \cap FV(MP) = \emptyset$, where, $z_1,\ldots,z_n$ are mutually distinct.
6. $\mu_n(\bar{x}).M \equiv \mu_n(\bar{y}).N$, if $M[\overline{z/y}] \equiv N[\overline{z/y}]$, for some $\{\bar{z}\} \cap FV(MN) = \emptyset$, where, $z_1,\ldots,z_n$ are mutually distinct.
7. $M \bullet (|\bar{N}|)_n \equiv P \bullet (|\bar{Q}|)_n$ if $M \equiv P$ and $N_i \equiv Q_i$ for $i = 1,\ldots,n$

Note that:

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

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

1. $x_i[\overline{N/x}] = N_i$
2. $y[\overline{N/x}] = y$, if $y \notin \{\bar{x}\}$
3. $(\lambda y.M)[\overline{N/x}] = \lambda y.M[\overline{N/x}]$, if $y \notin \{\bar{x}\}$ and $y \notin FV(\bar{N})$
4. $(MP)[\overline{N/x}] = (M[\overline{N/x}])(P[\overline{N/x}])$
5. (let $y := M$ in P)$[\overline{N/x}]$ = let $y := M[\overline{N/x}]$ in $P[\overline{N/x}]$, if $y \notin \{\bar{x}\}$ and $y \notin FV(\bar{N})$
6. $(M < \overline{y := P} >)[\overline{N/x}] = M[\overline{N/x}] < \overline{y := P[\overline{N/x}]}$, if $\{\bar{x}\} \cap \{\bar{y}\} = \emptyset$ and $\{\bar{y}\} \cap FV(\bar{N}) = \emptyset$

7. $(\mu(\bar{y}).M)\left[\overline{N/x}\right] = \mu(\bar{y}).\left(M\left[\overline{N/x}\right]\right)$, if $\{\bar{x}\}\cap\{\bar{y}\}=\emptyset$ and $\{\bar{y}\}\cap FV(\bar{N})=\emptyset$
8. $\left(M\bullet(|\bar{P}|)\right)\left[\overline{N/x}\right] = M\left[\overline{N/x}\right]\bullet\left(\left|\overline{P\left[\overline{N/x}\right]}\right|\right)$

Typing Rules

A *type assignment*, Γ, is a set of assumptions of the form x: σ. If Γ = {x_1: σ_1,…,x_n : σ_n}, then dom (Γ) = {x_1,…,x_n} and Γ(x_i) = σ_i. If V is a set of variables, then the restriction of Γ to the set V, Γ | V, is {x : σ| $x \in V$ and σ= Γ(x)}. A *typing judgment* is an expression of the form $\Gamma \vdash M : \sigma$, where Γ is a type assignment, M is a term and σ is a type scheme. The set of free type variables of a type assignment, *FTV*(Γ), is defined as:

$$\begin{aligned}\mathrm{FTV}(\emptyset) &= \emptyset\\ \mathrm{FTV}(\Gamma, x:\sigma) &= \mathrm{FTV}(\Gamma)\cup\mathrm{FTV}(\sigma)\end{aligned}$$

The following abbreviations are often be used:

$$\begin{aligned}&x:\sigma \vdash M:\sigma_1 \quad \text{for} \quad \{x:\sigma\}\vdash M:\sigma_1\\ &\Gamma, x:\sigma \vdash M:\sigma_1 \quad \text{for} \quad \Gamma\cup\{x:\sigma\}\vdash M:\sigma_1\\ &\vdash M:\sigma \quad \text{for} \quad \emptyset\vdash M:\sigma\end{aligned}$$

Substitution over type assignments is defined as:

$$\begin{aligned}S(\emptyset) &= \emptyset\\ S(\Gamma, x:\sigma) &= S(\Gamma), x:S(\sigma)\end{aligned}$$

The type assignment Γ has an *instance* Γ' if and only if there exists a substitution S such that $S\,\Gamma = \Gamma'$. A typing judgment $\Gamma' \vdash M:\sigma'$ is an *instance* of $\Gamma \vdash M:\sigma$ if there exists a substitution S with $S\,\Gamma \subseteq \Gamma'$ and σ'= Sσ.

If Γ is a type assignment, *the closure* of a type scheme σ with respect to Γ is $Clos(\Gamma,\sigma) = \forall\bar{\alpha}.\sigma$, where $\bar{\alpha} = \mathrm{FTV}(\sigma) - \mathrm{FTV}(\Gamma)$. For example, if Γ is $\{x:\forall\alpha.\alpha\to\gamma\}$ and σ is $\beta\to\gamma$, then Clos $(\Gamma,\sigma) = \forall\beta.\beta\to\gamma$.

Assume that Γ contains at most one assumption for each variable x. The type assignment Γ_x stands for removing any assumption x from Γ. The notation, $\overline{x:\sigma}$ is used as an abbreviation for $x_1:\sigma_1,\ldots,x_n:\sigma_n$. The typing rules of λ_{letmx} 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 Γ and τ such that we can derive $\Gamma \vdash M:\tau$ by using the λ_{letmx} typing rules.

Lemma 2 (Milner, 1978)

Let S be a substitution, Γ be a type assignment and τ be a type, then: $SClos(\Gamma,\tau) = Clos(S\Gamma, S\tau)$, where $S' = S\left[\overline{\beta/\alpha}\right]$, $\bar{\alpha} = FTV(\tau) - FTV(\Gamma)$ and $\bar{\beta}$ are fresh type variables.

Lemma 3 (Damas and Milner, 1982)

If S is a substitution and Γ⊢M : σ holds, then: SΓ⊢M : Sσ also holds

Lemma 4 (Damas and Milner, 1982)

If $\sigma_1 \succ \sigma_2$ and $\Gamma_x, x:\sigma_2 \vdash M:\sigma_0$ then:

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

$$\frac{}{\Gamma, x:\sigma \vdash x:\sigma}(var)$$

$$\frac{\Gamma_x, x:\tau \vdash M:\upsilon}{\Gamma \vdash \lambda x.M:\tau\Rightarrow\upsilon}(abs) \qquad \frac{\Gamma\vdash M:\tau\Rightarrow\upsilon \;\; \Gamma\vdash N:\tau}{\Gamma\vdash MN:\upsilon}(app)$$

$$\frac{\Gamma\vdash M:\sigma,\ \alpha\notin FTV(\Gamma)}{\Gamma\vdash M:\forall\alpha.\sigma}(typeGen) \qquad \frac{\Gamma\vdash M:\sigma_1 \;\; (\sigma_1\succ\sigma_2)}{\Gamma\vdash M:\sigma_2}(inst)$$

$$\frac{\Gamma\vdash M:\sigma \qquad \Gamma_x, x:\sigma\vdash N:\tau}{\Gamma\vdash \ let\ x=M\ in\ N:\tau}(let)$$

$$\frac{\Gamma_{\bar{x}}, x_1:\sigma_1,\ldots,x_n:\sigma_n\vdash M:\tau \qquad \Gamma\vdash N_1:\sigma_1\ldots\Gamma\vdash N_n:\sigma_n}{\Gamma\vdash M<\overline{x:=N}>:\tau}(expSub)$$

$$\frac{\Gamma x, \overline{x:\tau}\vdash M:\upsilon}{\Gamma\vdash\mu_n(\bar{x}).M:(|\bar{\tau}|)_n\Rightarrow\upsilon}(mAbs) \qquad \frac{\Gamma\vdash M:(|\bar{\tau}|)_n\Rightarrow\upsilon \quad \Gamma\vdash N_1:\tau_1\ldots\Gamma\vdash N_n:\tau j}{\Gamma\vdash M\bullet(|\bar{N}|)\ :\upsilon}(sApp)$$

Fig. 1: The typing rules for λ_{letmx}

Reductions

The reduction rules, $\rightarrow_{\lambda_{letmx}}$, of λ_{letmx} are the union of the following reduction rules, $\rightarrow_{\lambda}$, $\rightarrow_{let}$, $\rightarrow_{m}$ and $\rightarrow_{x}$:The reduction rule $\rightarrow_{\lambda}$ is defined by the rule:

$$(\lambda)\ \ (\lambda x.M)N \rightarrow_{\lambda} M<x:=N>$$

The reduction rule $\rightarrow_{let}$ is defined by the rule:

$$(let)\ \ let\, x=N\, in\, M \rightarrow_{let} M<x:=N>$$

The reduction rule $\rightarrow_{m}$ is defined by the rule:

$$(m)\ \left(\mu_n(\bar{x}).M\right)\bullet(|\bar{N}|)_n \rightarrow_m M<\overline{x:=N}>$$

Note that, The order of the variables $x_i\ (i=1,\ldots,n)$ in $\mu_n(\bar{x})$ is important and there is no reduction for the term $\left(\mu_n(\bar{x}).M\right)\bullet\left(\left|\bar{N}\right|\right)_m$ when $n\neq m$.

The reduction rule $\rightarrow_{x}$ is defined by the following 6 rules:

(xVar) $\quad x_i<\overline{x:=N}>\rightarrow_x N_i$.

(gc) $\quad M<\overline{x:=N}>\rightarrow_x M$, if $\{\bar{x}\}\cap FV(M)=\emptyset$.

(xAbs) $\quad (\lambda y.M)<\overline{x:=N}>\rightarrow_x \lambda y.M<\overline{x:=N}>$,
if $y\notin\{\bar{x}\}$, and $y\notin FV(\bar{N})$.

(xApp) $\quad (M_1M_2)<\overline{x:=N}>\rightarrow_x$
$(M_1<\overline{x:=N}>)(M_2<\overline{x:=N}>)$.

(xmAbs) $\left(\mu_n(\bar{y}).M\right)<\overline{x:=N}>\rightarrow_x \mu_n(\bar{y}).M<\overline{x:=N}>$
if $\{\bar{x}\}\cap\{\bar{y}\}=\emptyset$ and $\{\bar{y}\}\cap \mathrm{FV}(\bar{N})=\emptyset$.

(xsApp) $\quad \left(M\bullet(|\bar{N}|)_n\right)<\overline{x:=P}>\rightarrow_x$
$\left(M<\overline{x:=P}>\right)\bullet\left(\left|\overline{N<\overline{x:=P}>}\right|\right)_n$.

Let R = {λ, let, m, x}, we write $M\rightarrow_r N$, if N is obtained from M by replacing a subterm M_1 in M by N_1 such that $M_1\rightarrow_r N_1$, where $r\in R$. The reduction $\rightarrow_{q,r,s}$ is the union of the three reductions $\rightarrow_q$, $\rightarrow_r$ and $\rightarrow_s$, where $q,r,s\in R$. The reflexive and transitive closure of the reduction $\rightarrow$ is denoted by $\rightarrow^*$.

Example 1

The term:

$$\left(\lambda X.\left(\lambda x.\lambda y.X\bullet(|x,y|)\right)zx\right)\left(\mu(x,y).xy\right)$$

is reduced by λ_{letmx} reduction rules as:

$$\begin{aligned}
&\left(\lambda X.\left(\lambda x.\lambda y.X\bullet(|x,y|)\right)zx\right)\left(\mu(x,y).xy\right)\\
\rightarrow_{\lambda}\ &\left(\lambda X.\left(\lambda y.X\bullet(|x,y|)\right)<x:=z>x\right)\left(\mu(x,y).xy\right)\\
\rightarrow_{x}^{*}\ &\left(\lambda X.\left(\lambda y.X\bullet(|z,y|)\right)x\right)\left(\mu(x,y).xy\right)\\
\rightarrow_{\lambda}\ &\left(\lambda X.X\bullet(|z,y|)\right)<y:=x>\left(\mu(x,y).xy\right)\\
\rightarrow_{x}^{*}\ &\left(\lambda X.X\bullet(|z,x|)\right)\left(\mu(x,y).xy\right)\\
\rightarrow_{\lambda}\ &\left(X\bullet(|z,x|)\right)<X:=\left(\mu(x,y).xy\right)>\\
\rightarrow_{x}^{*}\ &\left(\mu(x,y).xy\right)\bullet(|z,x|)\\
\rightarrow_{m}\ &(xy)<x:=z,y:=x>\\
\rightarrow_{x}^{*}\ &zx
\end{aligned}$$

Properties of the Calculus

In this section, we show that λ_{letmx} 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 FV $(M)\subseteq$ dom (Γ)
2. If $\Gamma\vdash M:\sigma$ then $\Gamma\,|\,FV(M)\vdash M:\sigma$

Theorem 1 (Subject Reduction)

If $\Gamma\vdash M:\sigma$ and $M\rightarrow^*_{\lambda letmx} N$, 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 $\rightarrow_{\lambda letmx}$ is confluent, we first show that the reduction $\rightarrow_x$ is confluent.

Lemma 6

The reduction rule $\rightarrow_x$ on λ_{letmx} terms is neotherian and confluent.

Proof

To show that $\rightarrow_x$ is neotherian, 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. $|M < \bar{x} := \bar{N} >| = (|\bar{N}| + 1)|M|$, where $|\bar{N}| = |N_1| + \ldots + |N_n|$
6. $|\mu_n(\bar{x}).M| = |M| + n$
7. $|M \bullet (|\bar{N}|)| = |M| + |\bar{N}| + 1$

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

A λ_{letmx} term M is said to be x-normal if $M \rightarrow_x N$ holds for no N. From Lemma 6, it is clear that for any λ_{letmx} term M there uniquely exist x-normal term N such that $M \rightarrow_x^* N$. We will write x(M) to denote N.

Next, the parallel reduction (Takahashi, 1989), $\Rightarrow$ on x-normal λ_{letmx} 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 $(\lambda x.M_1)N_1 \Rightarrow x(M_2 < x := N_2 >)$
4. If $M_1 \Rightarrow M_2$ and $N_1 \Rightarrow N_2$, then $M_1 N_1 \Rightarrow M_2 N_2$
5. If $M \Rightarrow N$, then $\mu(\bar{x}).M \Rightarrow \mu(\bar{x}).N$
6. If $M \Rightarrow N$ and $P_i \Rightarrow Q_i$, $1 \le i \le n$, then $(\mu(\bar{x}).M) \bullet (|\bar{P}|) \Rightarrow x(N < \overline{x := Q} >)$
7. If $M \Rightarrow N$ and $P_i \Rightarrow Q_i, 1 \le i \le n$, then $M \bullet (|\bar{P}|) \Rightarrow N \bullet (|\bar{Q}|)$
8. If $M_1 \Rightarrow M_2$ and $N_1 \Rightarrow N_2$, then $let\ x := M_1\ in\ N_1 \Rightarrow x(N_2 < x := M_2 >)$

Then, with each x-normal term M, we associate an x-normal term M^* as:

1. $x^* := x$
2. $(\lambda x.M)^* := \lambda x.M^*$
3. $((\lambda x.M)N)^* := x(M^* < x := N^* >)$
4. $(MN)^* := M^* N^*$, if M is not a λ abstraction
5. $(\mu(\bar{x}).M)^* := \mu(\bar{x}).M^*$
6. $((\mu(\bar{x})).M \bullet (|\bar{P}|))^* := x(M^* < \overline{x := P^*} >)$
7. $(M \bullet (|\bar{P}|))^* := M^* \bullet (|\bar{P}^*|)$ if M is not a multi-abstraction.
8. $(let\ x :=\ = M\ in\ N)^* := x(N^* < x := M^* >)$

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 $M_1 \rightarrow_{\lambda letmx}^* M_2$.

Proof

By induction on the construction of M_1.

Lemma 8 (Substitution Lemma)

If M is an x-normal term, then:

$x(M < \overline{x := N} >< \overline{y := Q} >) \equiv x(M < \overline{y := Q} >< \overline{x := N < \overline{y := Q} >>})$,

where $\overline{N}$ and $\overline{Q}$ are λ_{letmx} terms, $\{\bar{x}\} \cap \{\bar{y}\} = \emptyset$ and $\{\bar{x}\} \cap FV\{\bar{y}\} = \emptyset$.

Proof

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

Lemma 9

If $x(M_1) \Rightarrow x(M_2)$ *and* $x(N_i) \Rightarrow x(P_i), 1 \le i \le n,$, then $x(M_1 < \overline{x := N} >) \Rightarrow x(M_2 < \overline{x := P} >)$.

Proof

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

Lemma 10

If $M_1 \rightarrow_{\lambda,let,m} M_2$, then $x(M_1) \Rightarrow x(M_2)$

Proof

By induction on the construction of M_1.

Lemma 11

If $M_1 \rightarrow_x M_2$, then $x(M_1) \Rightarrow x(M_2)$.

Proof

Immediate from the reflexivity of $\Rightarrow$.

Remark 1

From Lemma 10 and Lemma 11, we have:

If $M_1 \rightarrow_{\lambda letmx} M_2$, 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 $\rightarrow_{\lambda letmx}$ on λ_{letmx} terms is confluent, that is, if $M \rightarrow^*_{\lambda letmx} N$ and $M \rightarrow^*_{\lambda letmx} P$, then there is a term Q such that $N \rightarrow^*_{\lambda letmx} Q$ and $P \rightarrow^*_{\lambda letmx} Q$.

Proof

Suppose that M $\rightarrow^*_{\lambda letmx}$ N and M $\rightarrow^*_{\lambda letmx}$ 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. From Lemma 7, x(N) $\rightarrow^*_{\lambda letmx}$ Q and x(P) $\rightarrow^*_{\lambda letmx}$ Q. Since N $\rightarrow^*_{\lambda letmx}$ x(N) and $P \rightarrow^*_{\lambda letmx}$ x(P), we have N $\rightarrow^*_{\lambda letmx}$ Q and P $\rightarrow^*_{\lambda letmx}$ 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 Γ 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 Γ and σ 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 Γ, algorithm *W* finds a substitution *S* and a type τ such that's $\Gamma \vdash M : \sigma$.

The type scheme σ is called a *principal type scheme*, Damas and Milner (1982), of *M* under Γ 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 λ_{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 $(\tau \Rightarrow \upsilon, b)$, Unify $(\tau \Rightarrow \upsilon, (|\upsilon_1, \ldots, \upsilon_n|) \Rightarrow \tau_1)$, etc. This means that, Unify (τ,υ) fails if τ and υ are two different non type variables. It is clear that:

Unify(τ,υ) °*S* fails if Unify(τ,υ) fails.

$$
\begin{array}{lll}
1.\ Unify(\tau,\tau) & = & [\,] \\
2.\ Unify(\alpha,\tau)\ \ or\ \ Unify(\tau,\alpha) & = & \begin{cases} fails & if\ \alpha \in FTV(\tau) \\ & (Occur\,Check\,Failed) \\ [\tau/\alpha] & otherwise \end{cases} \\
3.\ Unify(\tau_1 \Rightarrow \tau_2, \upsilon_1 \Rightarrow \upsilon_2) & = & S_2, where \\
 & & S_1 = Unify(\tau_2, \upsilon_2) \\
 & & S_2 = Unify(S_1\tau_1, S_1\upsilon_1) \circ S_1 \\
4.\ Unify((|\upsilon_1,\ldots,\upsilon_n|) \Rightarrow \tau_1\ (|\varsigma_1,\ldots,\varsigma_n|) \Rightarrow \tau_2) & = & S_{n+1}, where \\
 & & S_1 = Unify(\tau_1, \tau_2) \\
 & & S_{i+1} = Unify(S_i\upsilon_i, S_i\varsigma_i) \circ S_i, i = 1,\ldots,n. \\
5.\ Unify(\tau,\upsilon) & & fails\ for\ all\ other\ cases
\end{array}
$$

Fig. 2: Algorithm Unify: τ x $\upsilon \rightarrow S$

Example 2

1. Unify $((|\alpha|)\Rightarrow int,(|int|)\Rightarrow\beta)=[int/\beta,int/\alpha]$
2. Unify $(\alpha\Rightarrow a,b\Rightarrow\alpha)$ fails.

The following two Theorems can easily be verified.

Theorem 3 (Soundness of Unify)

If Unify $(\tau,\upsilon)=S$, then $S\tau=S\upsilon$.

Theorem 4 (Completeness of Unify)

If V unifies τ and υ, then Unify (τ,υ) 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 λ_{letmx}. The extended algorithm is called $\hat{W}$. Algorithm $\hat{W}(\Gamma,M)$ is defined inductively on λ_{letmx} term M as in Fig. 3.

In algorithm $\hat{W}$, $\overline{x:Clos(S\Gamma,S\tau)}$ is an abbreviation of: x_1 : $Clos$(SΓ,Sτ$_1$),…,x_n: $Clos$(SΓ,Sτ$_n$). When any of the algorithm $\hat{W}$ conditions are not met, the algorithm fails.

Example 3

$\hat{W}\left(\left(X:(|int|)\Rightarrow int,x:int\right),X\bullet(|x|)\right)=(V,int)$, where V = Unify $((|int|)\Rightarrow int,(|int|)\Rightarrow\beta)$ and β 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 λ_{letmx} type inference rules.

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

If $\hat{W}$ (Γ, M) succeeds with (S,τ), then there is a derivation of SΓ⊢ M: τ.

Proof

By induction on the structure of λ_{letmx} term M. We consider only the following cases, other cases can be verified similarly:

1. Case M is x and $\hat{W}((\Gamma_x,x:\forall\bar{\alpha}.\tau),x)$ succeeds with ([ ], τ[$\overline{\beta/\alpha}$]), where $\bar{\beta}$ are fresh type variables. From the (var) typing rule, we have:

$$\Gamma_x,x:\forall\bar{\alpha}.\tau\vdash x:\forall\bar{\alpha}.\tau$$

Since $\forall\bar{\alpha}.\tau\succ\tau\left[\overline{\beta/\alpha}\right]$, then from the (inst) typing rule we have the following judgment as required:

$$\Gamma_x,x:\forall\bar{\alpha}.\tau\vdash x:\tau\left[\overline{\beta/\alpha}\right]$$

2. Case M is let x = P in N and $\hat{W}(\Gamma,let\,x=P\,in\,N)$ succeeds with $(S_2\circ S_1,\tau_2)$, where $(S_1\tau_1)=\hat{W}(\Gamma,P)$ and $(S_2,\tau_2)=\hat{W}\left((S_1\Gamma_x,x:Clos(S_1\Gamma,\tau_1)),N\right)$.

From the induction hypothesis, we have:

$$S_1\Gamma\vdash P:\tau_1\quad and \tag{1}$$

$$S_2S_1\Gamma_x,x:S_2\mathrm{Clos}(S_1\Gamma,\tau_1)\vdash N:\tau_2 \tag{2}$$

Let $S_2'=S_2\left[\overline{\beta/\alpha}\right]$, where $\bar{\beta}$ are fresh type variables and $\bar{\alpha}=FTV(\tau_1)-FTV(S_1\Gamma)$. Then, from Lemma 2, judgment (2) above and the fact that $S_2'S_1\Gamma=S_2S_1\Gamma$, because S_2' differ from S_2 only on bound variables of S_1Γ, we get:

$$S'_2S_1\Gamma_x,x:\mathrm{Clos}(S'_2S_1\Gamma,S'_2\tau_1)\vdash N:\tau_2 \tag{3}$$

Applying the (typeGen) typing rule to judgment (1) and since $\bar{\alpha}=FTV(\tau_1)-FTV(S_1\Gamma)$, we get:

$$S_1\Gamma\vdash P:\forall\bar{\alpha}.\tau_1 \tag{4}$$

From the definition of Clos($S_1\Gamma,\tau_1$) and judgment (4), we get:

$$S_1\Gamma\vdash P:\mathrm{Clos}\,(S_1\Gamma,\tau_1) \tag{5}$$

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

$$S'_2S_1\Gamma\vdash P:S'_2\mathrm{Clos}(S_1\Gamma,\tau_1) \tag{6}$$

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

$$S'_2S_1\Gamma\vdash P:\mathrm{Clos}(S'_2S_1\Gamma,S'_2\tau_1) \tag{7}$$

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

$$S'_2S_1\Gamma\vdash \text{let } x=P \text{ in } N:\tau_2$$

Finally, by noting that $S'_2S_1\Gamma=S_2S_1\Gamma$, We get the following required judgment:

$$S_2S_1\Gamma\vdash \text{let } x=P \text{ in } N:\tau_2$$

3. Case M is $P\bullet(|\bar{N}|)_n$ and $\hat{W}\left(\Gamma,P\bullet(|\bar{N}|)_n\right)$ succeeds with $(VoS_n o...oS_1oS',V\beta)$ where:

$$
\begin{array}{lcl}
\widehat{W}(\Gamma, x) & = & \left([\,], \left[\overline{\beta/\alpha}\right]\tau'\right) \text{ where } \Gamma(x) = \forall\bar{\alpha}.\tau' \text{ and} \\
 & & \bar{\beta} \text{ are fresh type variables.} \\
\widehat{W}(\Gamma, \lambda x.M) & = & (S, S\beta \Rightarrow \tau) \text{ where } (S,\tau) = \widehat{W}\left((\Gamma_x, x:\beta), M\right) \text{ and} \\
 & & \beta \text{ is a fresh type variable.} \\
\widehat{W}(\Gamma, MN) & = & \left(V \circ S_2 \circ S_1, V\beta\right) \text{ where } \left(S_1, \tau_1\right) = \widehat{W}(\Gamma, M), \\
 & & \left(S_2, \tau_2\right) = \widehat{W}\left(S_1\Gamma, N\right), V = Unify\ (S_2\tau_1, \tau_2 \Rightarrow \beta), \\
 & & \text{and } \beta \text{ is fresh type variable.} \\
\widehat{W}(\Gamma, let\ x = M \text{ in } N) & = & \left(S_2 \circ S_1, \tau_2\right) \text{ where} \left(S_1, \tau_1\right) = \widehat{W}(\Gamma, M) \text{ and} \\
 & & \left(S_2, \tau_2\right) = \widehat{W}\left(\left(S_1\Gamma_x, x:\ Clos\left(S_1\Gamma, \tau_1\right)\right), N\right) \\
\widehat{W}\left(\Gamma, M < \bar{x} := \bar{N} >\right) & = & \left(S' \circ S_n \circ \ldots \circ S_1, \upsilon\right) \text{ where} \\
 & & \left(S_1, \tau_1\right) = \widehat{W}\left(\Gamma, N_1\right), \ldots, \left(S_n\tau_n\right) = \widehat{W}\left(\Gamma, N_n\right) \text{ and} \\
 & & \left(S', \upsilon\right) = \widehat{W}\left(S_n \ldots S_1\Gamma_{\bar{x}}, \overline{x:\ Clos\left(S_n \ldots S_1\Gamma, S_n \ldots S_1\tau\right)}, M\right). \\
\widehat{W}\left(\Gamma, \mu(\bar{x}).M\right) & = & \left(S, (|\, S\bar{\beta}\,|) \Rightarrow \upsilon\right) \text{where } (S,\upsilon) = \widehat{W}\left(\left(\Gamma_{\bar{x}}, \overline{x:\beta}\right), M\right), \text{ and} \\
 & & \bar{\beta} \text{ are fresh type variables.} \\
\widehat{W}\left(\Gamma, M \bullet \left(|\, \bar{N}\,|\right)_n\right) & = & \left(V \circ S_n \circ \ldots \circ S_1 \circ S', V\beta\right) \text{where} \left(S', \tau\right) = \widehat{W}(\Gamma, M) \text{ and} \\
 & & \left(S_1, \upsilon_1\right) = \widehat{W}\left(S'\Gamma, N_1\right), \cdots, \left(S_n, \upsilon_n\right) = \widehat{W}\left(S'\Gamma, N_n\right), \\
 & & V = Unify\left(\left(S_n \cdots S_1\right)\tau, \left(|\, S_n, \ldots, S_1\bar{\upsilon}\,|\right) \Rightarrow \beta\right) \\
 & & \text{and } \beta \text{ is a fresh type variable.}
\end{array}
$$

Fig. 3: Algorithm $\widehat{W}: \Gamma \times M \rightarrow S \times \tau$

$(S', \tau) = \hat{W}(\Gamma, P)$ and (8)

$$\left(S_1, \upsilon_1\right) = \hat{W}\left(S'\Gamma, N_1\right), \cdots, \left(S_n, \upsilon_n\right) = \hat{W}\left(S'\Gamma, N_n\right) \quad (9)$$

and $V = Unify\left(\left(\left(S_n \cdots S_1\right)\tau\right), \left(|\, S_n, \ldots, S_1\bar{\upsilon}\,|\right) \Rightarrow \beta\right)$ and β is a fresh type variable.

From the induction hypothesis, we have the judgments:

$$S'\Gamma \vdash P : \tau \quad (10)$$

$$S_1S'\Gamma \vdash N_1 : \upsilon_1, \ldots, S_nS'\Gamma \vdash N_n : \upsilon_n \quad (11)$$

From Lemma 4 and judgment (11), since $S'\Gamma \succ S_1S'\Gamma$, we also have the following derivations:

$$S'\Gamma \vdash N_1 : \upsilon_1, \ldots, S'\Gamma \vdash N_n : \upsilon_n \quad (12)$$

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

$$\begin{array}{l}
VS_n \ldots S_1S'\Gamma \vdash P : VS_n \ldots S_1\tau \\
VS_n \ldots S_1S'\Gamma \vdash N_1 : VS_n \ldots S_1\upsilon_1, \\
\vdots \\
VS_n \ldots S_1S'\Gamma \vdash N_n : VS_n \ldots S_1\upsilon_n
\end{array} \quad (13)$$

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

$$VS_n \ldots S_1\tau = V\left((|\, S_n \ldots S_1\bar{\upsilon}\,|) \Rightarrow \beta\right)$$

We get the required judgment $VS_n \ldots S_1S'\Gamma \vdash P \bullet (|\, \bar{N}\,|) : V\beta$.

Note that, if $S\Gamma \vdash M : \tau$, then $S\Gamma \vdash M : \text{Clos}(S\Gamma, \tau)$ also holds, where 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 $\{\tau/\alpha:\ \tau/\alpha \in S \text{ and } \alpha \in A\}$. If *S* and *R* are two substitutions such that for every $\alpha \in dom(S) \cap dom(R), S\alpha = R\alpha$, the *simultaneous composition* Damas (1985), of *S* and *R*, $(S \oplus R)$, is defined by:

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

Lemma 13

If $\hat{W}$ (Γ, M)= (S, τ), then:

1. $Var(S) \subseteq FTV(\Gamma) \cup New$
2. $FTV(\tau) \subseteq FTV(\Gamma) \cup New$

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

Proof

By induction on the structure of the λ_{letmx} term *M*.

Lemma 14 (Lee and Yi, 1998)

If $\Gamma \succ \Gamma'$, then $Clos(\Gamma, \tau) \succ Clos(\Gamma', \tau)$.

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

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

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

Proof

By induction on the structure of the λ_{letmx} 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 λ_{letmx} terms can also be represented as terms in the ordinary lambda calculus, the λ_{letmx} 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.

$(\lambda x_1 . \cdots . \lambda x_n . [.]) M_1 \cdots M_m$ which is represented in λ_{letmx} as: $(\lambda x_1 . \cdots . \lambda x_n . (X \bullet (| x_1, \cdots, x_n |))) M_1 \cdots M_m$. 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:

$$(\lambda x_1 . \cdots . \lambda x_n . (\underline{(\mu(\bar{x}).N) \bullet (| x_1, \cdots, x_n |)})) M_1 \cdots M_m \quad (14)$$

This term can be represented as the lambda term:

$$(\lambda x_1 . \cdots . \lambda x_n . (\underline{(\lambda x_1 . \cdots \lambda x_n . N) x_1 \cdots x_n})) M_1 \cdots M_m \quad (15)$$

The λ_{letmx} 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 • is used to store terms in case if the *β*-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 β*-reductions, before each of them we have to check if the *α*-conversion rule in needed or not to avoid the unintended variable capturing. An *n* meta-level substitutions is also needed after each *β*-reduction which also requires to check the necessity of the *α*-conversion. Whereas, to reduce the corresponding underline subterm of (14) only one $\rightarrow_m$ reduction step is needed. Then, only one checking for an *α*-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:

$$(\mu(x, y).xy) \bullet (| y+1, z |) \rightarrow_m (xy) < x := y+1, y := z >$$
$$\rightarrow_x x < x := y+1, y := z > y < x := y+1, y := z > \rightarrow_x^* (y+1)z$$

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

$$((\lambda x.\lambda y.xy)(y+1)z) \equiv ((\lambda x.\lambda x_1 .xx_1)(y+1)z \rightarrow_\beta$$
$$(\lambda x_1 .xx_1)[(y+1)/x]z) \equiv (\lambda x_1 .(y+1)x_1)z) \rightarrow_\beta$$
$$((y+1)x_1)[z/x_1] \equiv (y+1)z$$

Conclusion

ML-style multi-abstraction calculus, λ_{letmx}, 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 λ_{letmx} 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

Barendregt, H.P., 1992. Lambda Calculi with Types. Handbook of Logic in Computer Science, Background: Computational Structures, Oxford University Press, New York, USA, ISBN-10: 0-19-853761-1, pp: 117-309.

Bloo, R. and K.H. Rose, 1995. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. Proceeding of CSN'95 (Computer Science in Netherlands), pp: 62-72.

Bognar, M., 2002. Contexts in Lambda Calculus, Ph.D thesis, Vrije University, Amsterdam.

Damas, L. and R. Milner, 1982. Principle type-schemes for functional programs. Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan. 25-27, Albuquerque, New Mexico, ACM, pp: 207-212. DOI: 10.1145/582153.582176

Damas, L., 1985. Type Assignment in Programming Languages. Ph.D Thesis, Computer Science Department, Edinburgh University, Technical report CST-33-85.

Harper, R., 2011. Programming in Standard ML, Robert Harper. http://www.cs.cmu.edu/ rwh/introsml/

Hashimoto, M. and A. Ohori, 2001. A typed context calculus. Theoretical Comput. Sci., 266: 249-272. DOI: 10.1016/S0304-3975(00)00174-2

Hashimoto, M., 1998. First-class contexts in ML. Int. J. Foundat. Comput. Sci., 11: 65-87. DOI: 10.1142/S0129054100000053

Hindley, J.R., 1969. The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc., 146: 29-60. DOI: 10.2307/1995158

Jim, T., 1996. What are principal typings and what are they good for? Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan. 21-24, Petersburg Beach, Florida, USA, pp: 42-53. DOI: 10.1145/237721.237728

Lee, O. and K. Yi, 1998. Proofs about a folklore let-polymorphic type inference algorithm. ACM Tranact. Programm. Lang. Syst., 20: 707-723. DOI: 10.1145/291891.291892

Milner, R., 1978. A theory of type polymorphism in programming, J. Comput. Sys. Sci., 17: 348-375. DOI: 10.1016/0022-0000(78)90014-4

Robinson, J.A., 1965. A machine-oriented logic based on the resolution principle. J. ACM, 12: 23-41. DOI: 10.1145/321250.321253

Ruhrberg, P., 1996. Simultaneous abstraction and Semantic Theories, Ph.D. Thesis, University of Edinburgh. http://hdl.handle.net/1842/520

Sands, D., 1998. Computing with contexts, a simple approach. Electronic Notes Theoretical Comput. Sci., 10: 134-149. DOI: 10.1016/S1571-0661(05)80694-2

Sato, M., T. Sakurai and Y. Kameyama, 2002. A simply typed context calculus with first-class environments, J. Funct. Logic Programm, 4: 395-374. DOI: 10.1007/3-540-44716-4 23

Taha, A.A., M. Sato and Y. Kameyama, 2002. A second-order context calculus, J. Comput. Software, 19: 158-175. DOI: 10.11309/jssst.19.158

Takahashi, M., 1989. Parallel reduction in λ-calculus. J. Symbolic Computat., 7: 113-123. DOI: 10.1016/S0747-7171(89)80045-8

Thompson, S., 2011. Haskell: The Craft of Functional Programming, 3rd Edn., Pearson Education Limited, Harlow, United Kingdom: Addison Wesley, ISBN: 978-0-201-88295-7, pp: 608.

Tobisawa, K., 2015. A meta lambda calculus with cross-level computation. Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan. 15-17, Mumbai, India, pp: 383-393. DOI: 10.1145/2676726.2676976

Urban, C. and T. Nipkow, 2009. Nominal Verification of Algorithm W. In: From Semantics to Computer Science: Essays in Honour of Gilles Kahn, Huet, G., J.J. Levy, G. Plotkin (Eds.), Cambridge University Press, Cambridge, ISBN-10: 0521518253, pp: 363-382.