Notes on Foundations of Programming Languages Types

Sandra Alves

October 16, 2023

Abstract

So far we have discussed the type-free λ-calculus. In this chapter we will discuss type systems for the λ-calculus. The initial motivation to define typed versions of the λ-calculus was to avoid paradoxical uses of the untyped calculus [Church, 1940].

1 Simple Types

The Curry Type System was first studied in [Curry, 1934] for the theory of combinators. In [Curry and Feys, 1958] this system was modified for the λ-calculus. The definitions and proofs of results in this section can be found in [Barendregt, 1992].

We start by defining the set of types for this system.

Definition 1.1 *Let $\mathbb{V}$ be an infinite set of type variables. The* set of simple types, *$\mathbb{T}_C$ is inductively defined from $\mathbb{V}$ in the following way:*

$$\begin{array}{rcl} \alpha \in \mathbb{V} & \Rightarrow & \alpha \in \mathbb{T}_C \\ \tau, \tau' \in \mathbb{T}_C & \Rightarrow & (\tau \to \tau') \in \mathbb{T}_C \end{array}$$

Notation If $\tau_1, \ldots, \tau_n \in \mathbb{T}_C$, then

$$\tau_1 \to \tau_2 \to \cdots \to \tau_n$$

represents

$$(\tau_1 \to (\tau_2 \to \cdots \to (\tau_{n-1} \to \tau_n) \ldots))$$

that is, the type constructor $\to$ is right associative.

Definition 1.2 *If x is a term variable in $\mathcal{V}$ and τ is a type in $\mathbb{T}_C$ then:*

- *A* statement *is of the form $M : \tau$, where the type τ is called the* predicate, *and the term M is called the* subject *of the statement.*
- *A* declaration *is a statement where the subject is a term variable.*
- *A* basis Γ *is a set of declarations where all the subjects are distinct.*

A basis where the subjects are pairwise distinct, is called *monovalent* (or consistent).

Definition 1.3 *If $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a basis, then:*

- *Γ is a partial function, with* domain, *denoted* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$, *and* $\Gamma(x_i) = \tau_i$.
- *Let $\mathcal{V}_0$ be a set of variables. Then $\Gamma \restriction \mathcal{V}_0 = \{x : \Gamma(x) \mid x \in \mathcal{V}_0\}$.*
- *We define Γ_x as $\Gamma \setminus \{x : \tau\}$ (another possible notation is $\Gamma, x : \tau$).*

Definition 1.4 *In the Curry type system, we say that M has type τ given the basis Γ, and write*

$$\Gamma \vdash_C M : \tau,$$

if $\Gamma \vdash_C M : \tau$ can be obtained from the following derivation rules:

$$\Gamma \cup \{x : \tau\} \vdash_C x : \tau \qquad (\mathsf{Axiom})$$

$$\frac{\Gamma_x \cup \{x : \tau_1\} \vdash_C M : \tau_2}{\Gamma \vdash_C \lambda x.M : \tau_1 \to \tau_2} \qquad (\to \mathsf{Intro})$$

$$\frac{\Gamma \vdash_C M : \tau_1 \to \tau_2 \qquad \Gamma \vdash_C N : \tau_1}{\Gamma \vdash_C MN : \tau_2} \qquad (\to \mathsf{Elim})$$

Example 1.5 *For the* λ*-term* $(\lambda xy.x)(\lambda x.x)$ *the following derivation is obtained in the Curry Simple Type System:*

$$\dfrac{\dfrac{\dfrac{\{x:\alpha\to\alpha, y:\beta\}\vdash_{\mathcal{C}} x:\alpha\to\alpha}{\{x:\alpha\to\alpha\}\vdash_{\mathcal{C}} \lambda y.x:\beta\to\alpha\to\alpha}}{\vdash_{\mathcal{C}} \lambda xy.x:(\alpha\to\alpha)\to\beta\to\alpha\to\alpha} \qquad \dfrac{\{x:\alpha\}\vdash_{\mathcal{C}} x:\alpha}{\vdash_{\mathcal{C}} \lambda x.x:\alpha\to\alpha}}{\vdash_{\mathcal{C}} (\lambda xy.x)(\lambda x.x):\beta\to\alpha\to\alpha}$$

Proposition 1.6 (Basic lemmas) *Let* Γ *be a basis:*

- *Let* Γ' *be a basis such that* $\Gamma\subseteq\Gamma'$*, then*

$$\Gamma\vdash_{\mathcal{C}} M:\tau \Rightarrow \Gamma'\vdash_{\mathcal{C}} M:\tau.$$

- *If* $\Gamma\vdash_{\mathcal{C}} M:\tau$*, then* $\mathsf{fv}(M)\subseteq\mathsf{dom}(\Gamma)$.
- *If* $\Gamma\vdash_{\mathcal{C}} M:\tau$*, then* $\Gamma\restriction\mathsf{fv}(M)\vdash_{\mathcal{C}} M:\tau$.

Definition 1.7 (Substitution) *We call* type-substitution *to*

$$\mathbb{S}=[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n]$$

where $\alpha_1,\ldots,\alpha_n$ *are distinct type variables and* $\tau_1,\ldots,\tau_n$ *are types in* $\mathbb{T}_C$*. If* τ *is a type in* $\mathbb{T}_C$*, then* $\mathbb{S}(\tau)$ *is the type obtained by simultaneously substituting* α_i *by* τ_i*, with* $1\le i\le n$*, in* τ*.*

The type $\mathbb{S}(\tau)$ *is called an* instance *of the type* τ*. The notion of substitution can be extended to basis in the following way:*

$$\mathbb{S}(\Gamma) \;=\; \{x_1:\mathbb{S}(\tau_1),\ldots,x_n:\mathbb{S}(\tau_n)\} \qquad \text{if } \Gamma=\{x_1:\tau_1,\ldots,x_n:\tau_n\}$$

The basis $\mathbb{S}(\Gamma)$ *is called an* instance *of the basis* Γ*.*

Next we will present some standard properties of the Simple Type System. Details and proofs can be found in [Hindley, 1997].

Lemma 1.8 (Substitution lemmas)

1. *If* $\Gamma\vdash_{\mathcal{C}} M:\tau$*, then* $\mathbb{S}(\Gamma)\vdash_{\mathcal{C}} M:\mathbb{S}(\tau)$.
2. *If* $\Gamma\cup\{x:\tau_1\}\vdash_{\mathcal{C}} M:\tau$ *and* $\Gamma\vdash_{\mathcal{C}} N:\tau_1$*, then* $\Gamma\vdash_{\mathcal{C}} M[N/x]:\tau$.

Theorem 1.9 (Subject reduction) *Let* M *be a* λ*-term, and* $M\twoheadrightarrow_\beta M'$*, then*

$$\Gamma\vdash_{\mathcal{C}} M:\tau \Rightarrow \Gamma\vdash_{\mathcal{C}} M':\tau.$$

The implication in the other direction is called *subject expansion*, and does not hold for this system. For example $(\lambda xy.y)(\lambda z.zz)\twoheadrightarrow_\beta(\lambda y.y)$, where $(\lambda y.y)$ is typable in this system, and $(\lambda xy.y)(\lambda z.zz)$ is not.

Subject reduction also holds for the three sub-calculi defined before. The property of subject expansion is verified in the $\lambda_{\mathcal{I}}$ calculus and in the linear and affine λ-calculus.

Theorem 1.10 (Strong normalization) *Let* M *be a* λ*-term.*

$$\Gamma\vdash_{\mathcal{C}} M:\tau \Rightarrow M \textit{ is strongly normalisable.}$$

Notice that the implication in the other direction does not hold. There are many strongly normalisable λ-terms that are not typable in this system. For example, the term $\lambda x.xx$ is in normal form therefore is strongly normalisable, but it is not typable in the Curry Simple Type System (notice that, to type the subterm xx, the variable x has to be of both type α and $\alpha\to\beta$).

1.1 Type-checking and Typability

In the Curry Type System, as well in other type systems the following questions arise:

1. Given a term M in Λ, a type τ and a basis Γ, do we have $\Gamma \vdash_{\mathcal{C}} M : \tau$?
2. Given a term M in Λ, is there a type τ and a basis Γ, such that $\Gamma \vdash_{\mathcal{C}} M : \tau$?

The first question concerns *type checking* and the second *typability*, and will be discuss in this section.

Type checking and typability in the Curry Type System are both decidable problems. Moreover, for the typability problem there exists a function that, for any typable term M, returns the most general type for M in this system. Such type is called the *principal type* of the term.

We first introduce the notions of principal pair and principal type.

Definition 1.11 (Principal pair) *Let M be a term in Λ. Then (Γ, τ) is called a* principal pair *for M if:*

1. $\Gamma \vdash_{\mathcal{C}} M : \tau$;
2. *If* $\Gamma' \vdash_{\mathcal{C}} M : \tau'$, *then* $\exists\, \mathbb{S}.\ (\mathbb{S}(\Gamma) \subseteq \Gamma'$ *and* $\mathbb{S}(\tau) \equiv \tau')$.

Note that, if (Γ, τ) is a principal pair for a term M, then $\mathsf{fv}(M) = \mathsf{dom}(\Gamma)$.

Definition 1.12 (Principal type) *Let M be a closed term in Λ. Then τ is called a* principal type *for M if:*

1. $\vdash_{\mathcal{C}} M : \tau$;
2. *If* $\vdash_{\mathcal{C}} M : \tau'$, *then* $\exists\, \mathbb{S}.\ (\mathbb{S}(\tau) \equiv \tau')$.

The principal type of a term M is a characterisation of the set of types that can be assigned to the term M. Note that every type that can be assigned to M, can be obtained from the principal type of M by applying a type-substitution.

The following result is independently due to Curry [Curry, 1969], Hindley [Hindley, 1969], and Milner [Milner, 1978] (see [Barendregt, 1992] for more details).

Theorem 1.13 (Principal type theorem)

1. *Let M be a term in Λ. There exists a total function* pp, *such that:*

$$\begin{array}{lcl} M \textit{ has a type} & \Rightarrow & \mathsf{pp}(M) = (\Gamma, \tau),\ \textit{and } (\Gamma, \tau) \textit{ is a principal pair for } M \\ M \textit{ has no type} & \Rightarrow & \mathsf{pp}(M) = \mathsf{fail}. \end{array}$$

2. *Let M be a closed term in Λ. There exists a total function* pt, *such that:*

$$\begin{array}{lcl} M \textit{ has a type} & \Rightarrow & \mathsf{pt}(M) = \tau,\ \textit{and } \tau \textit{ is a principal type of } M \\ M \textit{ has no type} & \Rightarrow & \mathsf{pt}(M) = \mathsf{fail}. \end{array}$$

Based on the functions define above, decidability for type-checking and typability in the Curry Type System, can be stated. Notice that that for $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ we have

$$\Gamma \vdash_{\mathcal{C}} M : \tau \text{ if and only if } \vdash_{\mathcal{C}} \lambda x_1 \ldots x_n.M : \tau_1 \to \cdots \to \tau_n \to \tau.$$

Therefore one can state the questions of type-checking and type-assignment, taking Γ to be an empty set.

Corollary 1.14 *Type checking and typability are decidable problems in the Curry Type System.*

Type checking decidability is proved by noticing that, given M and τ:

$$\vdash_{\mathcal{C}} M : \tau \iff \exists\, \mathbb{S}.\ (\mathbb{S}(\tau) = \mathsf{pt}(M)).$$

The type-substitution $\mathbb{S}$ is found using Robinson's of first-order unification [Robinson, 1965], which is decidable.

As for typability, notice that

$$M \text{ is typable} \iff \mathsf{pt}(M) \neq \mathsf{fail}.$$

2 Type-inference

We now present two principal type algorithms for the Curry Type System [Wand, 1987, Hindley, 1997], based on Robinson's unification algorithm [Robinson, 1965], that given a term M, return its principal typing.

To type-check a given type for a given term, one can use the type inference algorithms to calculate the principal type of the term and then verify if the given type is an instance of the principal type.

2.1 Unification

Robinson's unification algorithm [Robinson, 1965] plays a key role in type inference. We will present a definition of the unification algorithm for the particular case where the terms we want to unify are simple types.

Definition 2.1 *A unification problem is a finite set of equations $S = \{s_1 = t_1, \ldots, s_n = t_n\}$. A unifier (or solution) of S is a substitution $\mathbb{S}$, such that $\mathbb{S}s_i = \mathbb{S}t_i$, for $i = 1, \ldots, n$. We call $\mathbb{S}\,s_i$ (or $\mathbb{S}\,t_i$), a common instance of s_i and t_i. S is unifiable if it has at least one unifier. $\mathcal{U}(S)$ is the set of unifiers of S.*

Example 2.2 *The types $(\alpha \to \beta \to \alpha)$ and $((\gamma \to \gamma) \to \delta)$ are unifiable. For the substitution $\mathbb{S} = [(\gamma \to \gamma)/\alpha, \beta \to (\gamma \to \gamma)/\delta]$, the common instance is $((\gamma \to \gamma) \to \beta \to (\gamma \to \gamma))$.*

Definition 2.3 *A substitution $\mathbb{S}$ is a most general unifier (mgu) of S if $\mathbb{S}$ is a least element of $\mathcal{U}(S)$. That is,*

$$\mathbb{S} \in \mathcal{U}(S) \quad \textit{and} \quad \forall \mathbb{S}_1 \in \mathcal{U}(S).\exists \mathbb{S}_2.\mathbb{S}_1 = \mathbb{S}_2 \circ \mathbb{S}$$

A substitution is idempotent if $\mathbb{S} = \mathbb{S} \circ \mathbb{S}$.

Example 2.4 *Consider the types $\tau_1 = (\alpha \to \alpha)$ e $\tau_2 = (\beta \to \gamma)$. The substitution $\mathbb{S}' = [(\alpha_1 \to \alpha_2)/\alpha, (\alpha_1 \to \alpha_2)/\beta, (\alpha_1 \to \alpha_2)/\gamma]$ is a unifier of τ_1 and τ_2, but it is not the mgu.*

The mgu of τ_1 and τ_2 is $\mathbb{S} = [\alpha/\beta, \alpha/\gamma]$. The common instance of τ_1 and τ_2 by $\mathbb{S}'$, $(\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)$ is an instance of $(\alpha \to \alpha)$.

Theorem 2.5 *If a unification problem has a solution, then it has an idempotent mgu.*

Definition 2.6 *A unification problem $S = \{x_1 = t_1, \ldots, x_n = t_n\}$ is in solved form if x_i are all pairwised distinct variables that do not occur in any of the t_i. In this case we define $\mathbb{S}_S = [t_1/x_1, \ldots, t_n/x_n]$.*

Definition 2.7 *We define the following relation $\Rightarrow$ on unification problems.*

$$
\begin{array}{lllll}
\text{(DELETE)} & \{t = t\} \cup S & \Rightarrow & S & \\
\text{(DECOMPOSE)} & \{f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)\} \cup S & \Rightarrow & \{t_1 = u_1, \ldots, t_n = u_n\} \cup S & \\
\text{(ORIENT)} & \{t = x\} \cup S & \Rightarrow & \{x = t\} \cup S & \\
\text{(ELIMINATE)} & \{x = t\} \cup S & \Rightarrow & \{x = t\} \cup [t/x]S & \textit{if } x \in \mathsf{fv}(S) \setminus \mathsf{fv}(t) \\
\text{(CLASH)} & \{f(t_1, \ldots, t_n) = g(u_1, \ldots, u_m)\} \cup S & \Rightarrow & \mathsf{FAIL} & \\
\text{(OCCUR-CHECK)} & \{x = t\} \cup S & \Rightarrow & \mathsf{FAIL} & \textit{if } x \in \mathsf{fv}(t) \textit{ and } x \neq t
\end{array}
$$

Example 2.8 *Some examples:*

$$
\begin{array}{l}
\{x = f(a), g(x, x) = g(x, y)\} \Rightarrow \\
\{x = f(a), g(f(a), f(a)) = g(f(a), y)\} \Rightarrow \\
\{x = f(a), f(a) = f(a), f(a) = y\} \Rightarrow \\
\{x = f(a), f(a) = y\} \Rightarrow \\
\{x = f(a), y = f(a)\}
\end{array}
$$

$$
\begin{array}{l}
\{f(x, x) = f(y, g(y))\} \Rightarrow \\
\{x = y, x = g(y)\} \Rightarrow \\
\{x = y, y = g(y)\} \Rightarrow \mathsf{FAIL}
\end{array}
$$

Definition 2.9 (Unification algorithm) *Let S be a unification problem. The unification function* $\mathsf{UNIFY}(S)$ *is defined as:*

$\mathsf{UNIFY}(S) =$ *while* $(S \Rightarrow T)$ *do* {
$S := T$;
if (S *is in solved form*) *return* $\mathbb{S}_S$ *else* FAIL;
}

Lemma 2.10 UNIFY *terminates on all inputs.*

Lemma 2.11 *If $S \Rightarrow T$ then $\mathcal{U}(S) = \mathcal{U}(T)$.*

Lemma 2.12 (Soundness) *If* $\mathsf{UNIFY}(S)$ *returns a substitution $\mathbb{S}$, then $\mathbb{S}$ is an idempotent mgu of S.*

Lemma 2.13 (Completeness) *If S is solvable,* $\mathsf{UNIFY}(S)$ *does not fail.*

Definition 2.14 (Type unification) *We define the following relation $\Rightarrow$ on type unification problems.*

$$
\begin{array}{lll}
\{\tau = \tau\} \cup S & \Rightarrow & S \\
\{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} \cup S & \Rightarrow & \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup S \\
\{\tau = \alpha\} \cup S & \Rightarrow & \{\alpha = \tau\} \cup S \\
\{\alpha = \tau\} \cup S & \Rightarrow & \{\alpha = \tau\} \cup [\tau/\alpha]S \quad \textit{if } \alpha \in \mathsf{fv}(S) \setminus \mathsf{fv}(\tau) \\
\{\alpha = \tau\} \cup S & \Rightarrow & \mathsf{FAIL} \quad \textit{if } \alpha \in \mathsf{fv}(\tau) \textit{ and } x \neq \tau
\end{array}
$$

2.1.1 A recursive algorithm for type unification

The function UNIFY_R, given two types τ_1 and τ_2, returns the mgu of τ_1 and τ_2 if it exists, and fails otherwise.

Definition 2.15 *Let τ_1 and τ_2 be two types. The unification function* $\mathsf{UNIFY}_R(\tau_1, \tau_2)$ *is inductively defined as:*

$$
\begin{array}{lcll}
\mathsf{UNIFY}_R(\alpha,\tau) & = & [\tau/\alpha] & \textit{if } \alpha \notin \mathsf{FV}(\tau) \\
 & = & \mathrm{Id} & \textit{(identity function) if } \tau = \alpha \\
 & = & \mathtt{fail} & \textit{otherwise} \\
 & & & \\
\mathsf{UNIFY}_R(\tau_1 \to \tau_2, \alpha) & = & \mathsf{UNIFY}_R(\alpha, \tau_1 \to \tau_2) & \\
 & & & \\
\mathsf{UNIFY}_R(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) & = & \textit{let} & \\
 & & \quad \mathbb{S} = \mathsf{UNIFY}_R(\sigma_2, \tau_2) & \\
 & & \textit{in} \;\; \mathsf{UNIFY}_R(\mathbb{S}\,\sigma_1, \mathbb{S}\,\tau_1) \circ \mathbb{S} &
\end{array}
$$

Note that let $\mathbb{S} = \mathsf{UNIFY}_R(\sigma_2, \tau_2)$ *in* $\mathsf{UNIFY}_R(\mathbb{S}\,\sigma_1, \mathbb{S}\,\tau_1) \circ \mathbb{S}$*, fails if one of the calls of* UNIFY_R *fails.*

2.2 Milner's Type-Inference Algorithm

The following algorithm, presented in [Milner, 1978], defines a function that, given a term M returns a basis Γ and a type τ such that $\Gamma \vdash M : \tau$, is the principal typing of M.

Definition 2.16 *Let* Γ *be a basis,* M *a term and* τ *a type. Let* UNIFY *be the unification function defined above. The function* $T(M) = (\Gamma, \tau)$ *defines a type inference algorithm for the simply typed* λ*-calculus, in the following way:*

1. *If* M *is a variable* x*, then* $\Gamma = \{x : \alpha\}$ *and* $\tau = \alpha$*, where* α *is a new variable;*
2. *If* $M \equiv M_1 M_2$*,* $T(M_1) = (\Gamma_1, \tau_1)$ *e* $T(M_2) = (\Gamma_2, \tau_2)$*, then let* $\{\nu_1, \ldots, \nu_n\} = FV(M_1) \cap FV(M_2)$*, such that* $\nu_1 : \delta_1, \ldots, \nu_n : \delta_n \in \Gamma_1$ *and* $\nu_1 : \delta'_1, \ldots, \nu_n : \delta'_n \in \Gamma_2$*. Then* $T(M) = (\mathbb{S}(\Gamma_1 \cup \Gamma_2), \mathbb{S}\alpha)$*, where* $\mathbb{S} = \mathsf{UNIFY}(\{\delta_1 = \delta'_1, \ldots, \delta_n = \delta'_n\} \cup \{\tau_1 = \tau_2 \to \alpha\})$ *(*α *is a fresh variable).*
3. *If* $M \equiv \lambda x.N$ *and* $T(N) = (\Gamma_N, \sigma)$ *then:*
   - **a)** *If* $x \notin \mathsf{dom}(\Gamma_N)$*, then* $T(M) = (\Gamma_N, \alpha \to \sigma)$*, where* α *is a new variable;*
   - **b)** *If* $\{x : \tau\} \in \Gamma_N$*,* $T(M) = (\Gamma_N \setminus \{x : \tau\}, \tau \to \sigma)$.

2.3 Wand's Algorithm

The following algorithm [Wand, 1987], given a basis, a term, and a type returns a set of equations, that when applied unification, give the principal type of the term:

Definition 2.17 *Let* Γ *be a basis,* $M \in \Lambda$ *and* $\sigma \in \mathbb{T}$*. Let* $E = (\Gamma, M, \sigma)$ *be the set of equations defined by:*

$$
\begin{array}{rcl}
E(\Gamma, x, \sigma) & = & \{\sigma = \Gamma(x)\} \\
E(\Gamma, MN, \sigma) & = & E(\Gamma, M, \alpha \to \sigma) \cup E(\Gamma, N, \alpha), \\
 & & \textit{where } \alpha \textit{ is a new variable;} \\
E(\Gamma, \lambda x.M, \sigma) & = & E(\Gamma \cup \{x : \alpha\}, M, \beta) \cup \{\alpha \to \beta = \sigma\}, \\
 & & \textit{where } \alpha, \beta \textit{ are new variables.}
\end{array}
$$

Applying unification to E *gives a substitution* $\mathbb{S}$ *such that* $\mathbb{S}(\Gamma) \vdash M : \mathbb{S}(\sigma)$.

3 The Damas-Milner Polymorphic Type System

We now present the Damas-Milner [Damas and Milner, 1982] type system for the λ-calculus with parametric polymorphism, which forms the basis of type inference algorithms for functional languages, such as ML and Haskell.

3.1 The term language

The set of terms is an extension of the λ-calculus with a a constructor *let*. Given an infinite set of term variables V, the term language is give by the following grammar:

$$M ::= x \mid MM' \mid \lambda x.M \mid \text{let } x = M \text{ in } M'$$

3.2 Type Schemes

The definition of types schemes allows us to represent the set of terms one can infer for a term M, given a basis Γ.

Definition 3.1 *We say that σ is a type scheme if σ is a simple type τ or a term of the form* $\forall\alpha_1, ..., \alpha_n.\tau$*, where* $\alpha_1, ..., \alpha_n$ *are called generic type variables.*

Definition 3.2 *If τ is a type and σ a type scheme, we say that τ is a generic instance of σ if and only if* $\sigma = \tau$ *or* $\sigma = \forall\alpha_1, ..., \alpha_n.\eta$ *and* $\exists\tau_1, ..., \tau_n$ *such that* $\tau = [\tau_i/\alpha_i]\eta$.

3.3 Types

The set of types of this system is given the the following grammar:

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau' \to \tau'' \\ \sigma & ::= & \tau \mid \forall\alpha.\sigma' \end{array}$$

where α belongs to an infinite set of type variables, τ, τ′ and τ″ are simple types, and σ and σ′ are type schemes.

3.4 The type system

Let x be a term variable, α a type variable, M and M′ terms, τ and τ′ simple types and σ and σ′ type schemes. The Damas-Milner type system is defined by the following inference rules: s

$$(Ax) \qquad \Gamma \cup \{x : \sigma\} \vdash_{ML} x : \sigma$$

$$(Gen) \qquad \frac{\Gamma \vdash_{ML} M : \sigma \qquad \alpha \notin fv(\Gamma)}{\Gamma \vdash_{ML} M : \forall\alpha.\sigma}$$

$$(Inst) \qquad \frac{\Gamma \vdash_{ML} M : \forall\alpha.\sigma}{\Gamma \vdash_{ML} M : \sigma[\tau/\alpha]}$$

$$(App) \qquad \frac{\Gamma \vdash_{ML} M : \tau' \to \tau \qquad \Gamma \vdash_{ML} M' : \tau'}{\Gamma \vdash_{ML} (MM') : \tau}$$

$$(Abs) \qquad \frac{\Gamma_x \cup \{x : \tau'\} \vdash_{ML} M : \tau}{\Gamma \vdash_{ML} \lambda x.M : (\tau' \to \tau)}$$

$$(Let) \qquad \frac{\Gamma \vdash_{ML} M : \sigma \qquad \Gamma_x \cup \{x : \sigma\} \vdash_{ML} M' : \sigma'}{\Gamma \vdash_{ML} \text{ let } x = M \text{ in } M' : \sigma'}$$

Example 3.3 *Consider the derivation tree for* $(\lambda x.x)$ *in this system:*

$$\dfrac{\dfrac{\{x:\alpha\} \vdash_{ML} x:\alpha}{\vdash_{ML} \lambda x.x : \alpha \to \alpha}\,\text{Abs}}{\vdash_{ML} \lambda x.x : \forall\alpha.\alpha \to \alpha}\,\text{Gen}$$

3.5 Type Inference

We will define the notion of closure of a type, which will be used in the definition of the algorithm.

Definition 3.4 *Let* V *be a set of type variables,* Γ *a basis and* τ *a type. The closure of* τ *with respect to* V, $\overline{V}(\tau)$ *is the type scheme* $\forall\alpha_1,\ldots,\alpha_n.\tau$, *where* $\alpha_1,\ldots,\alpha_n$ *are all the variables that occur in* τ *that do not belong to* V. $\overline{\Gamma}(\tau)$ *is the closure of* τ *with respect to the set of type variables that occur in* Γ.

We can now define the type-inference algorithm for this system.

Definition 3.5 *Let* Γ *be a basis,* M *a term,* $\mathbb{S}$ *a substitution and* τ *a type. Let* UNIFY_R *be the unification function defined above. The function* $W(\Gamma, M) = (\mathbb{S}, \sigma)$ *is defined in the following way:*

1. *If* M *is a variable* x *and* $x : \forall\alpha_1,\ldots,\alpha_n.\tau \in \Gamma$ *the* $\mathbb{S}$ *is the identity function and* $\sigma = [\beta_i/\alpha_i]\tau$, *where* β_i *is a new variable* $(1 \le i \le n)$.
2. *If* $M \equiv M_1 M_2$, *let:*
   - $W(\Gamma, M_1) = (\mathbb{S}_1, \tau_1)$;
   - $W(\mathbb{S}_1(\Gamma), M_2) = (\mathbb{S}_2, \tau_2)$;
   - $\mathbb{S}_3 = \mathcal{U}\mathsf{NIFY}(\mathbb{S}_2\tau_1, \tau_2 \to \beta)$, *where* β *is a new variable;*

   then $\mathbb{S} = \mathbb{S}_3 \circ \mathbb{S}_2 \circ \mathbb{S}_1$ *and* $\sigma = \mathbb{S}_3\beta$;
3. *If* $M \equiv \lambda x.N$, *let* β *be a new variable and* $W(\Gamma_x \cup \{x:\beta\}, N) = (\mathbb{S}_1, \tau_1)$, *then* $\mathbb{S} = \mathbb{S}_1$ *e* $\sigma = \mathbb{S}_1\beta \to \tau_1$;
4. *If* $M \equiv \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2$, *let:*
   - $W(\Gamma, M_1) = (\mathbb{S}_1, \tau_1)$ *e*
   - $W(\mathbb{S}_1(\Gamma_x) \cup \{x : \overline{\mathbb{S}_1\Gamma}(\tau_1)\}, M_2) = (\mathbb{S}_2, \tau_2)$

   then $\mathbb{S} = \mathbb{S}_2 \circ \mathbb{S}_1$ *e* $\sigma = \tau_2$.

3.6 Correctness and Completeness

Proposition 3.6 *If* $W(\Gamma, M) = (\mathbb{S}, \tau)$, *then* $\mathbb{S}(\Gamma) \vdash_{ML} M : \tau$.

Definition 3.7 *Let* Γ *be a basis and* M *a term. We say that* σ_P *is the principal type-scheme of* Γ *if and only if:*

- $\Gamma \vdash_{ML} M : \sigma_P$.
- *For any other* σ *such that* $\Gamma \vdash_{ML} M : \sigma$ *is a generic instance of* σ_P.

Theorem 3.8 *Given* Γ *and* M, *let* Γ' *be an instance of* Γ *and* σ *a type scheme such that* $\Gamma' \vdash_{ML} M : \sigma$. *Then:*

- $W(\Gamma, M)$ *succeeds*
- *If* $W(\Gamma, M) = (\mathbb{S}, \tau)$, *then there exists* $\mathbb{S}'$ *such that* $\Gamma' = \mathbb{S}'\mathbb{S}\Gamma$ *and* σ *is a generic instance of* $\mathbb{S}'\overline{\mathbb{S}\Gamma}(\tau)$.

Exercises

1 Use the simple types system to infer a type for:

(a) $\lambda xy.x$

(b) $\lambda x.y$ se $y : \tau$

(c) $\lambda xy.xyy$

2 Use the simple types system to verify the type of:

(a) $\vdash_C \lambda xyz.x(yz) : (\alpha \longrightarrow \beta) \longrightarrow (\gamma \longrightarrow \alpha) \longrightarrow (\gamma \longrightarrow \beta)$

(b) $\vdash_C \lambda xyz.xzy : (\alpha \longrightarrow \beta \longrightarrow \gamma) \longrightarrow \beta \longrightarrow \alpha \longrightarrow \gamma$

(c) $\vdash_C \lambda xy.xyy : (\alpha \longrightarrow \alpha \longrightarrow \beta) \longrightarrow \alpha \longrightarrow \beta$

(d) $\vdash_C \lambda xyz.xz(yz) : (\alpha \longrightarrow (\beta \longrightarrow \gamma)) \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \alpha \longrightarrow \gamma$

3 Using the type inference algoritm **T**, for the simple types system to infer a principal type for:

(a) $\lambda xyz.xzy$

(b) $\lambda xy.(\lambda z.x)(yx)$

(c) $\lambda xy.y(\lambda z.zxx)x$

(d) $\lambda xy.xy(\lambda z.yz)$

4 Implement the principal type inference algoritm **T**.

5 Prove the following basic lemmas:

(a) Let Γ be a basis. If Γ' is a basis such that $\Gamma \subseteq \Gamma'$, then $\Gamma \vdash_{\mathcal{C}} M : \tau$ implies that $\Gamma' \vdash_{\mathcal{C}} M : \tau$.

(b) Let Γ be a basis. If $\Gamma \vdash_{\mathcal{C}} M : \tau$, then $\mathsf{fv}(M) \subseteq \mathsf{dom}(\Gamma)$.

(c) Let Γ be a basis. If $\Gamma \vdash_{\mathcal{C}} M : \tau$, then $\Gamma \restriction \mathsf{fv}(M) \vdash_{\mathcal{C}} M : \tau$.

6 Show that:

(a) If $\Gamma \vdash_C M : \tau$, then $\mathbb{S}(\Gamma) \vdash_C M : \mathbb{S}(\tau)$

(b) If $\Gamma \cup \{x : \sigma\} \vdash_C M : \tau$ and $\Gamma \vdash_C N : \sigma$, then $\Gamma \vdash_C M[N/x] : \tau$

7 Using Wand's type inference for the simple types system, determine the principal type for:

(a) $\lambda xyz.xzy$

(b) $\lambda xy.(\lambda z.x)(yx)$

(c) $\lambda xy.y(\lambda z.zxx)x$

(d) $\lambda xy.xy(\lambda z.yz)$

8 Using the type inference algorithm W, for the Damas-Milner type system, infer a type for:

(a) `let` $i =\vdash_C x.xy$ `in` ii

(b) `let` $x =\vdash_C zy.zyy$ `in` xx

(c) `let` $x = \lambda zw.zw$ `in` $\lambda z.z(xx)$

9 Show that:

(a) If $\Gamma \vdash_{ML} M : \sigma$, then $\mathbb{S}(\Gamma) \vdash_{ML} M : \mathbb{S}(\sigma)$ and the size of the derivation tree for $\mathbb{S}(\Gamma) \vdash_{ML} M : \mathbb{S}(\sigma)$ is less or equal to the size of $\Gamma \vdash_{ML} M : \sigma$.

(b) If σ' is a generic instance of σ, and $\Gamma_x \cup \{x : \sigma'\} \vdash_{ML} M : \sigma''$, then $\Gamma_x \cup \{x : \sigma\} \vdash_{ML} M : \sigma''$.

References

[Barendregt, 1992] Barendregt, H. P. (1992). Lambda Calculi with Types. In Abramsky, S., Gabbay, D. M., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, Oxford.

[Church, 1940] Church, A. (1940). A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68.

[Curry, 1969] Curry, H. (1969). Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92.

[Curry and Feys, 1958] Curry, H. and Feys, R. (1958). *Combinatory Logic*, volume 1. North-Holland, Amsterdam.

[Curry, 1934] Curry, H. B. (1934). Functionality in Combinatory Logic. In *Proceedings of the National Academy of Science, U.S.A*, volume 20, pages 584–590. National Academy of Sciences.

[Damas and Milner, 1982] Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA. ACM.

[Hindley, 1969] Hindley, J. R. (1969). The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60.

[Hindley, 1997] Hindley, J. R. (1997). *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[Milner, 1978] Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery (ACM)*, 12:23–41.

[Wand, 1987] Wand, M. (1987). A Simple Algorithm and Proof for Type Inference. *Fundamenta Infomaticae*, 10:115–121.