5. Type inference

In this section we prove the decidability of typing in the simply typed lambda calculus and show how to adapt the typing algorithm for $\lambda\!\rightarrow$ in order to compute types in 'polymorphic' functional programming languages.

Typing in $\boldsymbol{\lambda\!\rightarrow}$

We will show that the system $\lambda\!\rightarrow$ is decidable, in the sense that it can be verified whether a certain term is typable. If so, a so-called principal type can be computed. As a consequence, type checking (verification of an assignment $M : \sigma$) is also decidable.

The presentation below roughly follows Barendregt (1992); see also Barendsen and Smetsers (1996). The idea of strictly splitting type reconstruction into generation of equations and unification is due to Wand (1987).

5.1. Definition. (i) A *type substitution* is a function $* : \mathbb{V} \rightarrow \mathbb{T}$. The result of *applying* $*$ to a type σ is denoted by σ^*.

(ii) A *unifier* of two types σ, τ is a substitution $*$ such that $\sigma^* \equiv \tau^*$. This $*$ is a *most general unifier* if $*$ is a unifier and for all substitutions $*'$

$$*' \text{ is a unifier of } \sigma, \tau \;\Rightarrow\; \exists *_1 \; [*' = *_1 \circ *].$$

(iii) A *system of type equations* is a finite set

$$\mathcal{E} = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$$

of equations between types. A *solution* for $\mathcal{E}$ is a substitution $*$ such that $*$ is a unifier of σ_i, τ_i for all i. In this case we write $* \models \mathcal{E}$. This naturally gives rise to a notion of *most general solution* for $\mathcal{E}$.

Note that a most general solution is unique up to renaming of type variables.

Robinson (1965) showed that a most general solution (if it exists) can be computed effectively.

5.2. Unification Theorem. (i) *There exists a recursive function $\mathcal{U}$ having as input a pair of types and as output either a substitution or* fail *such that*

$$\begin{aligned} (\sigma,\tau) \textit{ has a unifier} \;&\Rightarrow\; \mathcal{U}(\sigma,\tau) \textit{ is a most general unifier of } \sigma,\tau, \\ (\sigma,\tau) \textit{ has no unifier} \;&\Rightarrow\; \mathcal{U}(\sigma,\tau) = \mathsf{fail}. \end{aligned}$$

(ii) *There exists a recursive function $\mathcal{U}$ having as input systems of type equations ans as output either a substitution or* fail *such that*

$$\begin{aligned} \mathcal{E} \textit{ has a solution} \;&\Rightarrow\; \mathcal{U}(\mathcal{E}) \textit{ is a most general solution of } \mathcal{E}, \\ \mathcal{E} \textit{ has no unifier} \;&\Rightarrow\; \mathcal{U}(\mathcal{E}) = \mathsf{fail}. \end{aligned}$$

PROOF. (i) Define $\mathcal{U}(\sigma,\tau)$ by recursion, as follows.

$$\begin{array}{rcll} \mathcal{U}(\alpha,\tau) & = & [\alpha := \tau] & \text{if } \alpha \notin \mathrm{TV}(\tau), \\ & = & \mathit{Id} \text{ (the identity)} & \text{if } \tau \equiv \alpha, \\ & = & \mathsf{fail} & \text{otherwise;} \\ \mathcal{U}(\sigma_1{\to}\sigma_2,\alpha) & = & \mathcal{U}(\alpha,\sigma_1{\to}\sigma_2); & \\ \mathcal{U}(\sigma_1{\to}\sigma_2,\tau_1{\to}\tau_2) & = & \mathcal{U}(\sigma_1^{\mathcal{U}(\sigma_2,\tau_2)},\tau_1^{\mathcal{U}(\sigma_2,\tau_2)}) \circ \mathcal{U}(\sigma_2,\tau_2). & \end{array}$$

Define the measure $\sharp(\sigma,\tau)$ by

$$\sharp(\sigma,\tau) = (\sharp_{\mathrm{var}}(\sigma,\tau), \sharp_{\to}(\sigma,\tau)),$$

where $\sharp_{\mathrm{var}}(\sigma,\tau)$ denotes the number of variables in σ,τ and $\sharp_{\to}$ the number of arrows. One can show that this measure of $\mathcal{U}$-arguments decreases (w.r.t. the lexicographical ordering) with every recursive call. Hence $\mathcal{U}$ is a total function. Moreover $\mathcal{U}$ satisfies the requirements.

(ii) Let $\mathcal{E}$ be a system of type equations; say $\mathcal{E} = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$. Then define

$$\mathcal{U}(\mathcal{E}) = \mathcal{U}(\sigma_1{\to}\cdots{\to}\sigma_n, \tau_1{\to}\cdots{\to}\tau_n). \ \square$$

The next step is to associate with each term M a set of type equations in such a way that typability of M can be formulated in terms of solvability of those equations.

5.3. DEFINITION. Let $*, *'$ be substitutions.

(i) Let σ be a type. Then $*, *'$ are *equivalent* with respect to σ (notation $* \sim_\sigma *'$) if $*(\alpha) = *'(\alpha)$ for all $\alpha \in \mathrm{TV}(\sigma)$.

(ii) This notion is extended sequences of bases and/or types: we write $* \sim_{\Gamma,\sigma} *'$ if $* \sim_\tau *'$ for all types τ appearing in Γ, σ.

5.4. DEFINITION. Let $M \in \Lambda$, $\sigma \in \mathbb{T}$ and let Γ be a basis. A set of type equations $\mathcal{E}$ is *exact* for Γ, M, σ if for each substitution $*$ one has

(1) $* \models \mathcal{E} \quad \Rightarrow \quad \Gamma^* \vdash M : \sigma^*$

(2) $\Gamma^* \vdash M : \sigma^* \quad \Rightarrow \quad *' \models \mathcal{E}$ for some $*'$ with $*' \sim_{\Gamma,\sigma} *$.

5.5. PROPOSITION. *For each Γ, M, σ there exists a system of type equations $\mathcal{E} = \mathcal{E}(\Gamma, M, \sigma)$, computable from Γ, M, σ, such that $\mathcal{E}$ is exact for Γ, M, σ.*

PROOF. Define $\mathcal{E}(\Gamma, M, \sigma)$ by induction on M:

$$\begin{array}{rcl} \mathcal{E}(\Gamma, x, \sigma) & = & \{\Gamma(x) = \sigma\} \\ & & \text{(regard } \Gamma \text{ as a partial function)}, \\ \mathcal{E}(\Gamma, MN, \sigma) & = & \mathcal{E}(\Gamma, M, \alpha{\to}\sigma) \cup \mathcal{E}(\Gamma, N, \alpha) \\ & & (\alpha \text{ fresh}), \\ \mathcal{E}(\Gamma, \lambda x.M, \sigma) & = & \mathcal{E}(\Gamma \cup \{x{:}\alpha\}, M, \beta) \cup \{\sigma = \alpha{\to}\beta\} \\ & & (\alpha, \beta \text{ fresh}). \end{array}$$

We can verify that $\mathcal{E}(\Gamma, M, \sigma)$ is indeed exact by induction on the structure of M. $\square$

5.6. Definition. Let $M \in \Lambda$. The pair (Γ, σ) is a *principal typing* for M if

(1) $\Gamma \vdash M : \sigma$,

(2) $\Gamma' \vdash M : \sigma' \;\;\Rightarrow\;\; \Gamma' \supseteq \Gamma^*,\; \sigma' \equiv \sigma^*$ for some substitution $*$.

5.7. Principal Typing Theorem. *There exists a recursive function* pt *such that for all M*

$$\begin{array}{rcl} M \textit{ is typable} & \Rightarrow & \mathrm{pt}(M) \textit{ is a principal typing for } M; \\ M \textit{ is not typable} & \Rightarrow & \mathrm{pt}(M) = \mathsf{fail}. \end{array}$$

Proof. Let $M \in \Lambda$; say $\mathrm{FV}(M) = \{x_1, \ldots, x_n\}$. Set $\Gamma_0 = \{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\}$, and $\sigma_0 = \alpha$ (all α's fresh). Define

$$\begin{array}{rcll} \mathrm{pt}(M) & = & (\Gamma_0^*, \sigma_0^*) & \text{if } \mathcal{U}(\mathcal{E}(\Gamma_0, M, \sigma_0)) = *, \\ & = & \mathsf{fail} & \text{if } \mathcal{U}(\mathcal{E}(\Gamma_0, M, \sigma_0)) = \mathsf{fail}. \end{array}$$

The correctness of this procedure follows from Proposition 5.5 and Theorem 5.2. □

5.8. Corollary. (i) *Typability in $\lambda{\to}$ is decidable.*

(ii) *Type checking is decidable.*

Proof. (i) Immediately by Theorem 5.7.

(ii) Given Γ, M, σ, check whether $(\Gamma \restriction \mathrm{FV}(M), \sigma)$ is a substitution instance of $\mathrm{pt}(M)$. □

For the system $\lambda 2$ the situation is radically different. The following result is due to Wells (1994).

5.9. Theorem. *Type checking ans typability in $\lambda 2$-Curry are undecidable.*

Type inference in functional programming languages

In view of Theorem 5.9, there is no hope for finding a programming language with polymorphic types (in the sense of $\lambda 2$) that are automatically inferred by a compiler. Most functional languages, however, claim to have a polymorphic type system. The solution to this paradox is the observation that the polymorphism in these languages is very weak: it rather is an *instantiation mechanism* to handle the separation of function *definitions* from function *applications.*

If a functional program introduces the identity

$$\mathbf{I}\,x = x,$$

then an expression of the form

$$\cdots (\mathbf{I}\,\mathsf{3}) \cdots (\mathbf{I}\,\mathsf{true}) \cdots$$

only makes sense if $\mathbf{I}$ is regarded as a polymorphic function with type $\forall\alpha.\alpha{\to}\alpha$ (which is consistent with $\mathbf{I}$'s expansion $\lambda x.x$).

By restricting the use of the $\forall$-introduction rule to function definitions and the $\forall$-elimination rule to applications of function symbols one obtains a decidable restriction of $\lambda 2$. In the sequel we will describe a formal system based on this idea, and indicate a method for determining principal typings.

5.10. Definition. We extend Λ with explicit (non-recursive) definitions. The resulting term set (denoted by Λ^+) is defined inductively as follows.

$$\Lambda^+ ::= V \mid \Lambda^+\Lambda^+ \mid \lambda V.\Lambda^+ \mid \mathsf{let}\ V = \Lambda^+\ \mathsf{in}\ \Lambda^+.$$

Some hygiene is necessary: in $\mathsf{let}\ x = M\ \mathsf{in}\ N$, the variable x should not occur in M; moreover the bound variables in N should be chosen differently from x.

5.11. Definition. By $\mathbb{T}$ we denote the set of $\lambda\!\rightarrow$-types. The set of *type schemes* (notation $\mathbb{T}^\forall$) is introduced by setting

$$\mathbb{T}^\forall ::= \mathbb{T} \mid \forall \mathbb{V}.\mathbb{T}^\forall.$$

In the sequel, we let $\sigma, \tau, \ldots$ range over $\mathbb{T}$ and $S, T, \ldots$ over $\mathbb{T}^\forall$.

5.12. Definition. The rules of *type assignment* for Λ^+-terms are the following.

$$\Gamma, x{:}S \vdash x : S$$

$$\frac{\Gamma \vdash M : \sigma\to\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma\to\tau}$$

$$\frac{\Gamma \vdash M : S \qquad \Gamma, x{:}S \vdash N : \tau}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}$$

$$\frac{\Gamma \vdash M : \forall\alpha.S}{\Gamma \vdash M : S[\alpha := \tau]} \qquad \frac{\Gamma \vdash M : S}{\Gamma \vdash M : \forall\alpha.S}$$

An example of a correct type assignment is

$$\mathsf{let}\ \mathbf{I} = \lambda x.x\ \mathsf{in}\ \mathbf{I}(\lambda z.\mathbf{I}\,z) : \alpha\to\alpha.$$

The presentation given here contrasts Barendsen and Smetsers (1993), where types induced by function definitions and specifications of algebraic data types are collected into a separate environment.

In order to show that the system is decidable we construct a syntax-directed variant by incorporating the $\forall$-instantiation and quantification rules in the other rules.

5.13. Definition. (i) Let $\sigma \in \mathbb{T}$ and let Γ be a context. Then the Γ-*scheme* of σ (notation $\forall_\Gamma(\sigma)$) is the type $\forall\vec{\alpha}.\sigma$, where $\vec{\alpha}$ consists of the elements of $\mathrm{TV}(\sigma)$ not occurring in Γ.

(ii) The new derivation rules are the following.

$$\boxed{\begin{array}{c}
\Gamma, x{:}\forall\vec{\alpha}.\sigma \vdash x : \sigma[\vec{\alpha} := \vec{\tau}] \\[1em]
\dfrac{\Gamma \vdash M : \sigma{\to}\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad\qquad \dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma{\to}\tau} \\[1em]
\dfrac{\Gamma \vdash M : \sigma \qquad \Gamma, x{:}\forall_\Gamma(\sigma) \vdash N : \tau}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}
\end{array}}$$

We denote derivability in the syntax directed system by $\vdash^{\mathrm{sd}}$.

The following shows that the two systems are equivalent with respect to simple types.

5.14. Proposition. *For all $M \in \Lambda^+$*

$$\begin{array}{rcl}
\Gamma \vdash^{\mathrm{sd}} M : \sigma & \Rightarrow & \Gamma \vdash M : \sigma, \\
\Gamma \vdash M : \forall\vec{\alpha}.\sigma & \Rightarrow & \Gamma \vdash^{\mathrm{sd}} M : \sigma.
\end{array}$$

First observe that we cannot split type derivation into generation of equations and computation of a solution of these equations: typing a let-statement involves extending the context with a $\forall$-quantified type, which we have to infer first. The necessary unifications are therefore done *on the fly.* The unification techniques for $\lambda{\to}$ will be sufficient for computation of our types.

5.15. Proposition. *There exists a recursive function $\mathcal{T}$ having as input triples Γ, M, σ and as output either a substitution or* fail*, such that for all $*$*

$$\Gamma^* \vdash M : \sigma^* \;\Rightarrow\; \exists *' \;\; [* \sim_{\Gamma,\sigma} *' \circ \mathcal{T}(\Gamma, M, \sigma)];$$

moreover, if there does not exist $*$ with $\Gamma^* \vdash M : \sigma^*$ then $\mathcal{T}(\Gamma, M, \sigma) =$ fail.

Proof. We define $\mathcal{T}(\Gamma, M, \sigma)$ by induction on M:

$$\begin{array}{rcl}
\mathcal{T}(\Gamma, x\sigma) & = & \mathcal{U}(\widetilde{\Gamma(x)}, \sigma), \\
\mathcal{T}(\Gamma, MN, \sigma) & = & \mathcal{T}(\Gamma^*, N, \alpha^*) \circ *, \\
 & & \text{where } * = \mathcal{T}(\Gamma, M, \alpha{\to}\sigma), \quad \alpha \text{ fresh}, \\
\mathcal{T}(\Gamma, \lambda x.M, \sigma) & = & \mathcal{U}(\sigma^*, \alpha^*{\to}\beta^*) \circ *, \\
 & & \text{where } * = \mathcal{T}(\Gamma \cup \{x{:}\alpha\}, M, \beta), \quad \alpha, \beta \text{ fresh}, \\
\mathcal{T}(\Gamma, \mathsf{let}\ x = M\ \mathsf{in}\ N, \sigma) & = & \mathcal{T}(\Gamma^* \cup \{x{:}\forall_{\Gamma^*}(\alpha^*)\}, N, \sigma^*) \circ *, \\
 & & \text{where } * = \mathcal{T}(\Gamma, M, \alpha), \quad \alpha \text{ fresh}.
\end{array}$$

Here $\widetilde{\cdot}$ denotes the generation of a 'fresh instance' of a type scheme: $\widetilde{\forall\vec{\alpha}.\sigma} \equiv \sigma[\vec{\alpha} := \vec{\beta}]$, where $\vec{\beta}$ is fresh. $\Box$

5.16. Corollary. *Typability and type checking for Λ^+-terms are decidable.*

We will now discuss the extension of Λ^+ with *recursion*: function definitions in which the name of the function appears in the body. If one allows polymorphic type instantiation of function identifiers also in their own definitions, the system becomes undecidable. We therefore require that all recursive occurrences of the defined function symbol have the same type as the body of the function, cf. Milner (1978).

5.17. Definition. (i) The set Λ^+ is extended with recursive definitions:

$$\Lambda^{++} ::= V \mid \Lambda^{++}\Lambda^{++} \mid \lambda V.\Lambda^{++} \mid \mathsf{let}\ V = \Lambda^{++}\ \mathsf{in}\ \Lambda^{++} \mid \mathsf{letrec}\ V = \Lambda^{++}\ \mathsf{in}\ \Lambda^{++}.$$

In $\mathsf{letrec}\ x = M\ \mathsf{in}\ N$ we allow x to occur in M.

(ii) The (syntax directed) *typing rules* for Λ^{++} are those for Λ^+, extended with

$$\boxed{\dfrac{\Gamma, x{:}\sigma \vdash M : \sigma \qquad \Gamma, x{:}\forall_\Gamma(\sigma) \vdash N : \tau}{\Gamma \vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ N : \sigma}}$$

5.18. Proposition. *Typability and type checking for Λ^{++} are decidable.*

Proof. We can extend $\mathcal{T}$ by putting

$$\begin{aligned}\mathcal{T}(\Gamma, \mathsf{letrec}\ x = M\ \mathsf{in}\ N, \sigma) \quad &= \quad \mathcal{T}(\Gamma^* \cup \{x{:}\forall_{\Gamma^*}(\alpha^*)\}, N, \sigma^*) \circ *, \\ &\qquad \text{where } * = \mathcal{T}(\Gamma \cup \{x{:}\alpha\}, M, \alpha), \quad \alpha \text{ fresh.} \quad \square\end{aligned}$$

The extension of this system with mutually recursive function definitions is straightforward.

6. Resource conscious type systems

In these notes we will describe some type systems inspired by Linear Logic, introduced by Girard (1987). These systems do not only regulate the internal structure of objects according to their types, but also their number of occurrences in a term. Objects (especially parameters of a function) are regarded as *resources*. Some resources can be used exactly once; others can be discarded or used more than once.

Linear Logic is an example of a 'resource sensitive' logic. In these systems one keeps track of the number of times a certain assumption is used. Resourse concious logics have received considerable attention from computer scientists.

One of the potential applications is the incorporation of *assignments* in functional programming. This is paradoxical, because the absense of side effects is one of the main reasons why functional languages are often praised. As a consequence of this absense, functional languages have the fundamental property of *referential transparency*: each (sub)expression denotes a fixed value, independently of the way this value is computed.