Chapter 5

Type Assignment

The lambda calculus as treated so far is usually referred to as a *type-free* theory. This is so, because every expression (considered as a function) may be applied to every other expression (considered as an argument). For example, the identity function $\mathsf{I} \equiv \lambda x.x$ may be applied to any argument x to give as result that same x. In particular I may be applied to itself.

There are also typed versions of the lambda calculus. These are introduced essentially in Curry (1934) (for the so called Combinatory Logic, a variant of the lambda calculus) and in Church (1940). Types are usually objects of a syntactic nature and may be assigned to lambda terms. If M is such a term and a type A is assigned to M, then we say '*M has type A*' or '*M in A*'; the denotation used for this is $M : A$. For example in some typed systems one has $\mathsf{I} : (A{\rightarrow}A)$, that is, the identity I may get as type $A{\rightarrow}A$. This means that if x being an argument of I is of type A, then also the value $\mathsf{I}x$ is of type A. In general, $A{\rightarrow}B$ is the type of functions from A to B.

Although the analogy is not perfect, the type assigned to a term may be compared to the dimension of a physical entity. These dimensions prevent us from wrong operations like adding 3 volt to 2 ampère. In a similar way types assigned to lambda terms provide a partial specification of the algorithms that are represented and are useful for showing partial correctness.

Types may also be used to improve the efficiency of compilation of terms representing functional algorithms. If for example it is known (by looking at types) that a subexpression of a term (representing a funtional program) is purely arithmetical, then fast evaluation is possible. This is because the expression then can be executed by the ALU of the machine and not in the slower way in which symbolic expressions are evaluated in general.

The two original papers of Curry and Church introducing typed versions of the lambda calculus give rise to two different families of systems. In the typed lambda calculi *à la* Curry terms are those of the type-free theory. Each term has a set of possible types. This set may be empty, be a singleton or consist of several (possibly infinitely many) elements. In the systems *à la* Church the terms are annotated versions of the type-free terms. Each term has (up to an equivalence relation) a unique type that is usually derivable from the way the term is annotated.

The Curry and Church approaches to typed lambda calculus correspond to two paradigms in programming. In the first of these a program may be written without typing at all. Then a compiler should check whether a type can be assigned to the program. This will be the case if the program is correct. A well-known example of such a language is *ML*, see Milner (1984). The style of typing is called *implicit typing*. The other paradigm in programming is called *explicit typing* and corresponds to the Church version of typed lambda calculi. Here a program should be written together with its type. For these languages type-checking is usually easier, since no types have to be constructed. Examples of such languages are *Algol 68* and *Pascal*. Some authors designate the Curry systems as 'lambda calculi *with type assignment*' and the Church systems as 'systems of *typed* lambda calculus'.

Within each of the two paradigms there are several versions of typed lambda calculus. In many important systems, especially those *à la* Church, it is the case that terms that do have a type always possess a normal form. By the unsolvability of the halting problem this implies that not all computable functions can be represented by a typed term, see Barendregt (1990), Theorem 4.2.15. This is not so bad as it sounds, because in order to find such computable functions that cannot be represented, one has to stand on one's head. For example in $\lambda 2$, the second order typed lambda calculus, only those partial recursive functions cannot be represented that happen to be total, but not provably so in mathematical analysis (second order arithmetic).

Considering terms and types as programs and their specifications is not the only possibility. A type A can also be viewed as a proposition and a term M in A as a proof of this proposition. This so called propositions-as-types interpretation is independently due to de Bruijn (1970) and Howard (1980) (both papers were conceived in 1968). Hints in this direction were given in Curry and Feys (1958) and in Läuchli (1970). Several systems of proof checking are based on this interpretation of propositions-as-types and of proofs-as-terms. See e.g. de Bruijn (1980) for a survey of the so called AUTOMATH proof checking system. Normalization of terms corresponds in the formulas-as-types interpretation to normalisation of proofs in the sense of Prawitz (1965). Normal proofs often give useful proof theoretic information, see e.g. Schwichtenberg (1977).

In this section a typed lambda calculus will be introduced in the style of Curry. For more information, see Barendregt (1992).

The system $\lambda\!\rightarrow$-Curry

Originally the implicit typing paradigm was introduced in Curry (1934) for the theory of combinators. In Curry and Feys (1958) and Curry et al. (1972) the theory was modified in a natural way to the lambda calculus assigning elements of a given set $\mathbb{T}$ of types to type free lambda terms. For this reason these calculi *à la* Curry are sometimes called *systems of type assignment*. If the type $\sigma \in \mathbb{T}$ is assigned to the term $M \in \Lambda$ one writes $\vdash M : \sigma$, sometimes with a subscript under $\vdash$ to denote the particular system. Usually a set of assumptions Γ is needed to derive a type assignment and one writes $\Gamma \vdash M : \sigma$ (pronounce this as 'Γ yields M in σ'). A particular Curry type assignment system depends on two parameters, the set $\mathbb{T}$ and the rules of type assignment. As an example we now introduce the system $\lambda\!\rightarrow$-Curry.

5.1. Definition. The set of *types* of $\lambda\!\rightarrow$, notation $\mathrm{Type}(\lambda\!\rightarrow)$, is inductively defined as follows. We write $\mathbb{T} = \mathrm{Type}(\lambda\!\rightarrow)$. Let $\mathbb{V} = \{\alpha, \alpha', \ldots\}$ be a set of *type variables*. It will be convenient to allow *type constants* for basic types such as Nat, Bool. Let $\mathbb{B}$ be such a collection. Then

$$\begin{array}{rcl} \alpha \in \mathbb{V} & \Rightarrow & \alpha \in \mathbb{T}, \\ \mathsf{B} \in \mathbb{B} & \Rightarrow & \mathsf{B} \in \mathbb{T}, \\ \sigma, \tau \in \mathbb{T} & \Rightarrow & (\sigma\!\rightarrow\!\tau) \in \mathbb{T} \quad \text{(function space types).} \end{array}$$

For such definitions it is convenient to use the following abstract syntax to form $\mathbb{T}$.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\!\rightarrow\!\mathbb{T}$$

with

$$\mathbb{V} = \alpha \mid \mathbb{V}' \qquad \text{(type variables).}$$

Notation. (i) If $\sigma_1, \ldots, \sigma_n \in \mathbb{T}$ then

$$\sigma_1\!\rightarrow\!\sigma_2\!\rightarrow\!\cdots\!\rightarrow\!\sigma_n$$

stands for

$$(\sigma_1\!\rightarrow\!(\sigma_2\!\rightarrow\!\cdots\!\rightarrow\!(\sigma_{n-1}\!\rightarrow\!\sigma_n)\cdot\cdot));$$

that is, we use association to the right.

(ii) $\alpha, \beta, \gamma, \ldots$ denote arbitrary type variables.

5.2. Definition. (i) A *statement* is of the form $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$. This statement is pronounced as 'M in σ'. The type σ is the *predicate* and the term M is the *subject* of the statement.

(ii) A *basis* is a set of statements with only distinct (term) variables as subjects.

5.3. Definition. Type *derivations* in the system $\lambda\!\rightarrow$ are built up from assumptions $x{:}\sigma$, using the following inference rules.

$$\frac{M : \sigma\!\rightarrow\!\tau \qquad N : \sigma}{MN : \tau} \qquad\qquad \frac{\begin{array}{c}\cancel{x : \sigma} \\ \vdots \\ M : \tau\end{array}}{\lambda x.M : \sigma\!\rightarrow\!\tau}$$

5.4. Definition. (i) A statement $M : \sigma$ is *derivable from* a basis Γ, notation

$$\Gamma \vdash M : \sigma$$

(or $\Gamma \vdash_{\lambda\rightarrow} M : \sigma$ if we wish to stress the typing system) if there is a derivation of $M : \sigma$ in which all non-cancelled assumptions are in Γ.

(ii) We use $\vdash M : \sigma$ as shorthand for $\emptyset \vdash M : \sigma$.

5.5. Example. (i) Let $\sigma \in \mathbb{T}$. Then $\vdash \lambda fx.f(fx) : (\sigma{\to}\sigma){\to}\sigma{\to}\sigma$, which is shown by the following derivation.

$$\dfrac{\dfrac{\dfrac{\cancel{f:\sigma{\to}\sigma}^{(2)} \qquad \dfrac{\cancel{f:\sigma{\to}\sigma}^{(2)} \qquad \cancel{x:\sigma}^{(1)}}{fx:\sigma}}{f(fx):\sigma}}{\lambda x.f(fx):\sigma{\to}\sigma}\,(1)}{\lambda fx.f(fx):(\sigma{\to}\sigma){\to}\sigma{\to}\sigma}\,(2)$$

The indices (1) and (2) are bookkeeping devices that indicate at which application of a rule a particular assumption is being cancelled.

(ii) One has $\vdash \mathsf{K} : \sigma{\to}\tau{\to}\sigma$ for any $\sigma,\tau \in \mathbb{T}$, which is demonstrated as follows.

$$\dfrac{\dfrac{\cancel{x:\sigma}^{(1)}}{\lambda y.x:\tau{\to}\sigma}}{\lambda xy.x:\sigma{\to}\tau{\to}\sigma}\,(1)$$

(iii) Similarly one can show for all $\sigma \in \mathbb{T}$

$$\vdash \mathsf{I} : \sigma{\to}\sigma.$$

(iv) An example with a non-empty basis is the statement

$$y{:}\sigma \vdash \mathsf{I}y : \sigma.$$

Properties of $\lambda{\to}$

Several properties of type assignment in $\lambda{\to}$ are valid. The first one analyses how much of a basis is necessary in order to derive a type assignment.

5.6. Definition. Let $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ be a basis.

(i) Write $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\sigma_i = \Gamma(x_i)$. That is, Γ is considered as a partial function.

(ii) Let V_0 be a set of variables. Then $\Gamma \restriction V_0 = \{x{:}\sigma \mid x \in V_0 \,\&\, \sigma = \Gamma(x)\}$.

(iii) For $\sigma,\tau \in \mathbb{T}$ substitution of τ for α in σ is denoted by $\sigma[\alpha := \tau]$.

5.7. Basis Lemma. *Let Γ be a basis.*

(i) *If $\Gamma' \supseteq \Gamma$ is another basis, then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M : \sigma.$$

(ii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$.

(iii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \restriction \mathrm{FV}(M) \vdash M : \sigma$.

Proof. (i) By induction on the derivation of $M : \sigma$. Since such proofs will occur frequently we will spell it out in this simple situation in order to be shorter later on.

Case 1. $M : \sigma$ is $x{:}\sigma$ and is element of Γ. Then also $x{:}\sigma \in \Gamma'$ and hence $\Gamma' \vdash M : \sigma$.

Case 2. $M : \sigma$ is $(M_1M_2) : \sigma$ and follows directly from $M_1 : (\tau{\to}\sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma' \vdash M_1 : (\tau{\to}\sigma)$ and $\Gamma' \vdash M_2 : \tau$. Hence $\Gamma' \vdash (M_1M_2) : \sigma$.

Case 3. $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$ and follows directly from $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$. By the variable convention it may be assumed that the bound variable x does not occur in $\mathrm{dom}(\Gamma')$. Then $\Gamma', x{:}\sigma_1$ is also a basis which extends $\Gamma, x{:}\sigma_1$. Therefore by the IH one has $\Gamma', x{:}\sigma_1 \vdash M_1 : \sigma_2$ and so $\Gamma' \vdash (\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$.

(ii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$ and follows directly from $\Gamma, x{:}\sigma_1 \vdash M_1 : \sigma_2$. Let $y \in \mathrm{FV}(\lambda x.M_1)$, then $y \in \mathrm{FV}(M_1)$ and $y \not\equiv x$. By the IH one has $y \in \mathrm{dom}(\Gamma, x{:}\sigma_1)$ and therefore $y \in \mathrm{dom}(\Gamma)$.

(iii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(M_1M_2) : \sigma$ and follows directly from $M_1 : (\tau{\to}\sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma \restriction \mathrm{FV}(M_1) \vdash M_1 : (\tau{\to}\sigma)$ and $\Gamma \restriction \mathrm{FV}(M_2) \vdash M_2 : \tau$. By (i) it follows that $\Gamma \restriction \mathrm{FV}(M_1M_2) \vdash M_1 : (\tau{\to}\sigma)$and $\Gamma \restriction \mathrm{FV}(M_1M_2) \vdash M_2 : \tau$ and hence $\Gamma \restriction \mathrm{FV}(M_1M_2) \vdash (M_1M_2) : \sigma$. $\square$

The second property analyses how terms of a certain form get typed. It is useful among other things to show that certain terms have no types.

5.8. Generation Lemma. (i) $\Gamma \vdash x : \sigma \;\Rightarrow\; (x{:}\sigma) \in \Gamma$.
(ii) $\Gamma \vdash MN : \tau \;\Rightarrow\; \exists\sigma\,[\Gamma \vdash M : (\sigma{\to}\tau)\;\&\;\Gamma \vdash N : \sigma]$.
(iii) $\Gamma \vdash \lambda x.M : \rho \;\Rightarrow\; \exists\sigma,\tau\,[\Gamma, x{:}\sigma \vdash M : \tau\;\&\;\rho \equiv (\sigma{\to}\tau)]$.

Proof. By induction on the structure of derivations. $\square$

5.9. Proposition (Typability of subterms). *Let M' be a subterm of M. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M' : \sigma' \quad \textit{for some } \Gamma' \textit{ and } \sigma'.$$

The moral is: if M has a type, i.e. $\Gamma \vdash M : \sigma$ for some Γ and σ, then every subterm has a type as well.

Proof. By induction on the generation of M. $\square$

5.10. Substitution Lemma.
(i) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.
(ii) *Suppose* $\Gamma, x{:}\sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$. *Then* $\Gamma \vdash M[x := N] : \tau$.

Proof. (i) By induction on the derivation of $M : \sigma$.
(ii) By induction on the derivation showing $\Gamma, x{:}\sigma \vdash M : \tau$. $\square$

The following result states that the set of $M \in \Lambda$ having a certain type in $\lambda{\to}$ is closed under reduction.

5.11. Subject Reduction Theorem. *Suppose $M \twoheadrightarrow_\beta M'$. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \vdash M' : \sigma.$$

PROOF. Induction on the generation of $\twoheadrightarrow_\beta$ using the Generation Lemma 5.8 and the Substitution Lemma 5.10. We treat the prime case, namely that $M \equiv (\lambda x.P)Q$ and $M' \equiv P[x := Q]$. Well, if

$$\Gamma \vdash (\lambda x.P)Q : \sigma$$

then it follows by the Generation Lemma that for some τ one has

$$\Gamma \vdash (\lambda x.P) : (\tau{\to}\sigma) \text{ and } \Gamma \vdash Q : \tau.$$

Hence once more by the Generation Lemma

$$\Gamma, x{:}\tau \vdash P : \sigma \text{ and } \Gamma \vdash Q : \tau$$

and therefore by the Substitution Lemma

$$\Gamma \vdash P[x := Q] : \sigma. \;\Box$$

Terms having a type are not closed under expansion. For example,

$$\vdash \mathbf{I} : (\sigma{\to}\sigma), \text{ but } \nvdash \mathbf{KI}\,(\lambda x.xx) : (\sigma{\to}\sigma).$$

See Exercise 5.1. One even has the following stronger failure of subject expansion, as is observed in van Bakel (1992).

5.12. OBSERVATION. There are $M, M' \in \Lambda$ and $\sigma, \sigma' \in \mathbb{T}$ such that $M' \twoheadrightarrow_\beta M$ and

$$\vdash M : \sigma, \qquad \vdash M' : \sigma',$$

but

$$\nvdash M' : \sigma.$$

PROOF. Take $M \equiv \lambda xy.y$, $M' \equiv \mathbf{SK}$, $\sigma \equiv \alpha{\to}(\beta{\to}\beta)$ and $\sigma' \equiv (\beta{\to}\alpha){\to}(\beta{\to}\beta)$; do Exercise 5.1. $\Box$

All typable terms have a normal form. In fact, the so-called *strong normalization* property holds: if M is a typable term, then all reductions starting from M are finite.

Decidability of type assignment

For the system of type assignment several questions may be asked. Note that for $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ one has

$$\Gamma \vdash M : \sigma \;\Leftrightarrow\; \vdash (\lambda x_1{:}\sigma_1 \cdots \lambda x_n{:}\sigma_n.M) : (\sigma_1{\to} \cdots {\to}\sigma_n{\to}\sigma),$$

therefore in the following one has taken $\Gamma = \emptyset$. Typical questions are

(1) Given M and σ, does one have $\vdash M : \sigma$?
(2) Given M, does there exist a σ such that $\vdash M : \sigma$?
(3) Given σ, does there exist an M such that $\vdash M : \sigma$?

These three problems are called *type checking*, *typability* and *inhabitation* respectively and are denoted by $M : \sigma$?, M : ? and ? : σ.

Type checking and typability are decidable. This can be shown using the following result, independently due to Curry (1969), Hindley (1969), and Milner (1978).

5.13. Theorem. (i) *It is decidable whether a term is typable in* $\lambda\!\to$.

(ii) *If a term* M *is typable in* $\lambda\!\to$, *then* M *has a principal type scheme, i.e. a type* σ *such that every possible type for* M *is a substitution instance of* σ. *Moreover* σ *is computable from* M.

5.14. Corollary. *Type checking for* $\lambda\!\to$ *is decidable.*

Proof. In order to check $M : \tau$ it suffices to verify that M is typable and that τ is an instance of the principal type of M. $\square$

For example, a principal type scheme of K is $\alpha\to\beta\to\alpha$.

Polymorphism

Note that in $\lambda\!\to$ one has

$$\vdash \mathsf{I} : \sigma\to\sigma \qquad \text{for all } \sigma \in \mathbb{T}.$$

In the polymorphic lambda calculus this quantification can be internalized by stating

$$\vdash \mathsf{I} : \forall\alpha.\alpha\to\alpha.$$

The resulting system is the *polymorphic* of *second-order* lambda calculus due to Girard (1972) and Reynolds (1974).

5.15. Definition. The set of *types* of $\lambda 2$ (notation $\mathbb{T} = \text{Type}(\lambda 2)$) is specified by the syntax

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\to\mathbb{T} \mid \forall\mathbb{V}.\mathbb{T}.$$

5.16. Definition. The rules of type assignment are those of $\lambda\!\to$, plus

$$\frac{M : \forall\alpha.\sigma}{M : \sigma[\alpha := \tau]} \qquad \frac{M : \sigma}{M : \forall\alpha.\sigma}$$

In the latter rule, the type variable α may not occur free in any assumption on which the premiss $M : \sigma$ depends.

5.17. Example. (i) $\vdash \mathsf{I} : \forall\alpha.\alpha\to\alpha$.

(ii) Define Nat $\equiv \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$. Then for the Church numerals $\boldsymbol{c}_n \equiv \lambda fx.f^n(x)$ we have $\vdash \boldsymbol{c}_n$: Nat.

The following is due to Girard (1972).

5.18. Theorem. (i) *The Subject Reduction property holds for* $\lambda 2$.

(ii) $\lambda 2$ *is strongly normalizing.*

Typability in $\lambda 2$ is *not* decidable; see Wells (1994).

Exercises

5.1. (i) Give a derivation of

$$\vdash \mathsf{SK} : (\alpha{\to}\beta){\to}(\alpha{\to}\alpha).$$

(ii) Give a derivation of

$$\vdash \mathsf{KI} : \beta{\to}(\alpha{\to}\alpha).$$

(iii) Show that $\nvdash \mathsf{SK} : (\alpha{\to}\beta{\to}\beta)$.

(iv) Find a common β-reduct of SK and KI. What is the most general type for this term?

5.2. Show that $\lambda x.xx$ and $\mathsf{KI}(\lambda x.xx)$ have no type in $\lambda{\to}$.

5.3. Find the most general types (if they exist) for the following terms.
(i) $\lambda xy.xyy$.
(ii) SII.
(iii) $\lambda xy.y(\lambda z.z(yx))$.

5.4. Find terms $M, N \in \Lambda$ such that the following hold in $\lambda{\to}$.
(i) $\vdash M : (\alpha{\to}\beta){\to}(\beta{\to}\gamma){\to}(\alpha{\to}\gamma)$.
(ii) $\vdash N : (((\alpha{\to}\beta){\to}\beta){\to}\beta){\to}(\alpha{\to}\beta)$.

5.5. Find types in $\lambda 2$ for the terms in the exercises 5.2 and 5.3.