TYPES IN LAMBDA CALCULI AND PROGRAMMING LANGUAGES

Henk Barendregt
Faculty of Mathematics and Computer Science, University Nijmegen, Toernooiveld 1,
6525 ED Nijmegen, The Netherlands

Kees Hemerik
Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2,
5600 MB Eindhoven, The Netherlands

1. INTRODUCTION

The lambda calculus was originally conceived by Church as part of a general theory of functions and logic, intended as a foundation for mathematics. Although the full system turned out to be inconsistent, the subsystem dealing with functions only turned out to be a successful model of the computable functions. For an introduction to the subject and its relation to functional programming, see Barendregt [1990]. Some books on the lambda calculus are Barendregt [1984] and Hindley and Seldin [1986]. A major problem with the lambda calculus as functional programming language is the great amount of freedom in combining terms. A way to restrict this combinatorial freedom is the use of *types,* which are characterisations of classes of terms. The first type systems for lambda calculus were introduced in Curry [1934] and Church [1941]. Since then many typed lambda calculi have been proposed for different purposes.

The evolution of programming languages shows a similar pattern. The oldest varieties, like assembler and LISP, were essentially typefree and allowed too much combinatorial freedom. In languages like ALGOL 68 and Fortran simple but rather rigid type systems were introduced. For various applications these type systems were too restrictive and they have been extended in many ways. Type theory is currently an active research area. Publications, like the IEEE-LICS and ACM-POPL proceedings and papers like Cardelli and Wagner [1985] and Reynolds [1985], show that the developments in lambda cacluli and programming languages are converging. It also displays a bewildering variety of systems and confusion of notations and nomenclature. What is clearly needed is a classification or taxonomy of these systems on the basis of a common frame of reference. This paper presents a contribution towards such a classification. The exposition is based on the extensive treatment in Barendregt [199-], but because of space limitations proofs have been omitted. Although the paper concentrates on typed lambda calculi, at many places the relations with programming languages are illustrated.

The main criterion for the classification is based on the original systems of Curry and Church, which might be called *implicit* typing and *explicit* typing, respectively. This division is the subject of section 2. Section 3 systematically discusses the main ingredients of implicit type systems, and section 4 does the same for explicit type systems. Section 4 concludes with a taxonomy of explicit systems by means of generalised type systems, including the 'λ-cube' introduced in Barendregt [1989]. Since all systems discussed in this paper are ultimately based on the type-free lambda calculus, we conclude this introduction with a short review of that system for reference purposes.

1.1 TYPE-FREE LAMBDA CALCULUS

The aspects of lambda calculus important for functional programming consist of the syntax of terms and the reduction (rewrite) relation on these.

1.1.1. Definition. The sets of (term) variables V, of constants C and of lambda terms Λ are defined by the following abstract syntax.

$$V = x \mid V'$$
$$C = c \mid C'$$
$$\Lambda = V \mid C \mid \Lambda\,\Lambda \mid \lambda V.\Lambda$$

So $V = \{x, x', x'', \ldots\}$ and $C = \{c, c', c'', \ldots\}$.

1.1.2. Conventions.

(i) x, y, z,... range over V.

(ii) **a**, **b**, **c**,... range over C.

(iii) M, N, L,... range over Λ.

(iv) $\equiv$ stands for syntactic equality.

(v) $MN_1 \ldots N_k \equiv (..((MN_1)\,N_2)\ldots N_k)$, association to the left.

(vi) $\lambda x_1 \ldots x_k.M \equiv \lambda x_1.\,(\lambda x_2.\,\ldots(\lambda x_k.\,(M))..)$, association to the right.

1.1.3. Examples. The following are lambda terms.

(i) $\lambda x.x$.

(ii) $y(\lambda x.x)$.

(iii) $(\lambda x.xx)(\lambda x.xx)$

1.1.4. Definition

(i) FV(M) is the set of free variables of M.

(ii) $M\,[x := N]$ is the result of substituting N for the free occurrences of x in M.

1.1.5. Definition (Reduction).

(i) A binary relation $\rightarrow_\beta$ (one step reduction) is defined on Λ as follows.

$(\lambda x.M)N \rightarrow_\beta M\,[x := N]$;

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}; \qquad \frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'}; \qquad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}.$$

(ii) Many step reduction $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$.

(iii) Conversion $=_\beta$ is the equivalence relation generated by $\lambda\text{-}\twoheadrightarrow_\beta$.

Instead of $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ we often write $\rightarrow$, $\twoheadrightarrow$ and $=$, respectively.

1.1.6. Examples.

(i) $(\lambda x.xx)(\lambda y.y)z \rightarrow (\lambda y.y)(\lambda y.y)z$
$\twoheadrightarrow z$.

(ii) $(\lambda xy.x)\ z\ w \twoheadrightarrow z$.

(iii) $(\lambda x.xx)(\lambda y.y)z = (\lambda xy.x)zw$.

1.1.7. Definition.

(i) $M \in \Lambda$ is called a *normal form* (nf) if for no N one has $M \rightarrow N$.

(ii) M *has* a nf N if $M \twoheadrightarrow N$ and N is a nf.

(iii) M is *strongly normalising*, notation SN(M), if for no infinite sequence $M_1, M_2, \ldots$ one has $M \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots$.

1.1.8. Examples.

(i) $(\lambda x.xx)y$ has yy as nf.

(ii) $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ has no nf.

(iii) $(\lambda xy.x)\ \mathbf{a}\ \Omega$ has **a** as nf, but is not strongly normalising since $\Omega \rightarrow \Omega \rightarrow \ldots$.

1.1.9. Theorem (Church-Rosser property).

(i) If $M \twoheadrightarrow M_1$, and $M \twoheadrightarrow M_2$ then for some $M_3 \in \Lambda$ one has $M_1 \twoheadrightarrow M_3$ and $M_2 \twoheadrightarrow M_3$.

(ii) If $M_1 = M_2$, then for some $M_3 \in \Lambda$ one has $M_1 \twoheadrightarrow M_3$ and $M_2 \twoheadrightarrow M_3$.

1.1.10. Corollary (Unicity of normal forms). A lambda term has at most one nf.

There are certain terms in normal form $\ulcorner 0 \urcorner$, $\ulcorner 1 \urcorner$, ... (*numerals*) that represent the elements of $\mathbb{N}$, the set of natural numbers.

1.1.11. Theorem (Lambda definability of computable functions).

(i) Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a computable function. Then for some $F \in \Lambda$ one has for all $n \in \mathbb{N}$

$$F \ulcorner n \urcorner \twoheadrightarrow \ulcorner f(n) \urcorner.$$

(ii) More generally, if $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is computable, then for some $F \in \Lambda$ one has for all $n_1,\ldots,n_k \in \mathbb{N}$

$$F \ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \twoheadrightarrow \ulcorner f(n_1,\ldots,n_k) \urcorner.$$

This is the basis of evaluation of functional programs. The input and program form together one expression and by the Church-Rosser theorem it does not matter how reductions are done, as long as one cares to find the normal form (if it exists). For this there are several possible normalising strategies, for example leftmost reduction.

One way (but not the only) of obtaining representations of recursive functions is to use the so called fixed point operator Y.

1.1.12. Theorem. Let $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Then Y produces fixed points, i.e. for every F one has $F(YF) = YF$. Turing's fixed point operator $\Theta \equiv (\lambda ab.b(aab))(\lambda ab.b(aab))$ even has the property $\Theta F \twoheadrightarrow F(\Theta F)$.

2. IMPLICIT VERSUS EXPLICIT TYPING

There are two ways in which expressions denoting algorithms can be typed: the implicit way, originating with Curry [1934], and the explicit way, originating with Church [1941].

In the systems of implicit typing the expressions are the type-free lambda terms. To each such term a set of possible types is assigned. The number of elements in this set may be zero, one or more. If type σ is assigned to term M, then one writes $M : \sigma$ (pronounce 'M in σ'). An example (to be treated in detail later) is

$(\lambda x.x) : (\alpha \rightarrow \alpha)$,

that is, the identity is of type $\alpha \rightarrow \alpha$. This means that if $y : \alpha$, then $((\lambda x.x)y) : \alpha$. Also one has $(\lambda x.x) : ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta))$. Indeed, if $f : \alpha \rightarrow \beta$, then $((\lambda x.x)f) : (\alpha \rightarrow \beta)$. Therefore one, says that the identity is *polymorphic.*

In the systems of explicit typing the expressions are annotated versions of lambda terms. Such an expression has a type that usually is uniquely determined by the annotations. An example is

$(\lambda x{:}\alpha.x) : (\alpha \rightarrow \alpha)$.

One may write $I_\sigma \equiv (\lambda x{:}\sigma.x) : (\sigma \rightarrow \sigma)$ for the identity on σ. In particular $I_{\alpha \rightarrow \beta} : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$.

The components of an algorithm as given by a term usually have a fixed intended meaning. Therefore the explicitly typed terms are rather natural. However, it is often space and time consuming to annotate programs with types. Moreover, the annotation often can be constructed from the type-free expression by automatic means. Therefore the implicit typing paradigm is rather convenient.

Now we will introduce the most basic typed system, the *simply typed lambda calculus,* both in the style of Curry and that of Church. These systems will be denoted by $\lambda\rightarrow$-Curry and $\lambda\rightarrow$-Church.

2.1 THE IMPLICIT VERSION OF $\lambda\rightarrow$

Types and terms of $\lambda\rightarrow$-Curry.

There are many systems of types. The set of types for $\lambda\rightarrow$, notation Type $(\lambda\rightarrow)$, is defined by the following abstract grammar. We write $\mathbb{T}$ = Type $(\lambda\rightarrow)$.

2.1.1. Definition

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

$\mathbb{C} = \gamma \mid \mathbb{C}'$ (type constants)

$\mathbb{T} = \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \rightarrow \mathbb{T}$ (types)

Notice that $\mathbb{V} = \{\alpha, \alpha', \alpha'', ...\}$. We will use α, β, γ, ... to denote arbitrary type variables. Similarly $\mathbb{C} = \{\boldsymbol{\gamma}, \boldsymbol{\gamma}', \boldsymbol{\gamma}'', ...\}$. We will use $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$, ... to denote arbitrary type constants. Elements of $\mathbb{T}$ are α, $\alpha \to \alpha$, $\boldsymbol{\beta} \to \alpha \to \alpha$. Here and elsewhere we use association to the right for $\to$; that is, the last type is really $\boldsymbol{\beta} \to (\alpha \to \alpha)$. The letters σ, τ, ρ,... denote arbitrary elements of $\mathbb{T}$.

2.1.2. Definition. (i) A *statement* is of the from $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$. In this case M is called the *subject* and σ the *predicate*.

(ii) A *basis* is a set of statements with as subjects distinct (term) variables. The letters Γ, Δ,... range over bases.

Some of the type constants are given special names, like **int**, **bool**, **char**. These are used for formulating extensions of $\lambda\!\to$ in which certain elements are highlighted. For example term constants **0**, s may be selected and $\lambda\!\to$ can be extended with the axioms

$$\mathbf{o} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}$$

Of course everything could be done with free variables. Indeed term variables x_0 and x_s and a type variable α_{int} can be selected and one can consider as a basis

$$x_0 : \alpha_{int}, x_s : \alpha_{int} \to \alpha_{int}$$

In fact a constant is nothing but a variable that is not and will not be bound.

Assignment rules of $\lambda\!\to$-Curry

2.1.3 Definition. A statement $M : \sigma$ is derivable in $\lambda\!\to$ form a basis Γ, notation $\Gamma \vdash_{\lambda\to} M : \sigma$ or simply $\Gamma \vdash M : \sigma$ if there is no danger of confusion, if $\Gamma \vdash M : \sigma$ can be produced by the following assignment rules.

(Start) $\quad \Gamma \vdash x : \sigma$, if $x{:}\sigma$ is in Γ;

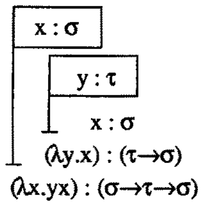
$$(\to E) \qquad \frac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau};$$

$$(\to I) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \to \tau)}.$$

Here Γ, $x{:}\sigma$ stands for $\Gamma \cup \{x{:}\sigma\}$. If $\Gamma = \emptyset$, then $\Gamma \vdash M : \sigma$ is written as $\vdash M : \sigma$. Pronounce $\vdash$ as yields. $\Gamma \nvdash M : \sigma$ means that $\Gamma \vdash M : \sigma$ does not hold. The rule $\to$E stands for $\to$-elimination; $\to$I for $\to$-introduction.

2.1.4. Examples.

(i) The following derivation shows that $\vdash_{\lambda\to} (\lambda xy.x) : (\sigma \to \tau \to \sigma)$. The derivation is written in a version of natural deduction, due to N.G. de Bruijn, in which the scope of assumptions is made explicit.



(ii) Similarly one has $\vdash_{\lambda\rightarrow} (\lambda xy.y) : (\sigma\rightarrow\tau\rightarrow\tau)$.

(iii) $\vdash_{\lambda\rightarrow} (\lambda x.x) : (\sigma\rightarrow\sigma)$.

(iv) $y{:}\sigma \vdash_{\lambda\rightarrow} ((\lambda x.x)y) : \sigma$.

Properties of λ→-Curry.

There are several valid properties of λ→ in which reduction and type assignment play a role. Since several (but not all) of these are valid also for other systems, we will indicate them by an acronym for quick reference. Not all versions of some property are exactly the same though and the precise formulation is given in case the statement differs from the standard meaning.

2.1.5. Definition. The following acronyms are used for properties of type systems.

(i) CR (Church-Rosser property).

This refers to the set of terms and reduction and conversion. The formulation is in 1.1.9.

(ii) SR (Subject reduction property). This states the following. Let $M \twoheadrightarrow M'$. Then

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

(iii) SN (Strong normalisation property). This is

$$\Gamma\vdash M : \sigma \Rightarrow \mathrm{SN}\ (M).$$

(iv) UT (Unicity of types). This is

$$\Gamma\vdash M : \sigma \ \&\ \Gamma\vdash M : \sigma' \Rightarrow \sigma \equiv \sigma'.$$

(v) $\Gamma\vdash M{:}\sigma$? is decidable. This states that given, Γ, M, σ it is decidable whether $\Gamma\vdash M{:}\sigma$.

(vi) $\Gamma\vdash M{:}?$ is computable. This states that given Γ, M it can be decided whether there is a σ such that $\Gamma\vdash M : \sigma$ and moreover one such σ can be computed from Γ, M.

Property 2.1.5 (iii) implies that not every computable function is lambda definable, see Barendregt [1990], theorem 4.2.15.

2.1.6. Theorem. For λ→-Curry the following properties hold.

(i) CR, for Λ and β-reduction/ -conversion.

(ii) SR.

(iii) SN.

(iv) $\Gamma\vdash M : \sigma?$ is decidable.

(v) $\Gamma\vdash M : ?$ is computable.

2.1.7. Remarks. (i) The converse of SR does not hold for $\lambda\!\to$. Let $S \equiv \lambda xyz.xz(yz)$ and $K \equiv \lambda pq.p$. Then $SK \twoheadrightarrow \lambda xy.y$ and $\vdash (\lambda xy.y) : (\sigma\to\tau\to\tau)$ but $\nvdash SK : (\sigma\to\tau\to\tau)$.

(ii) M is called typable if $\Gamma \vdash M : \sigma$ for some Γ and σ. SN states that if M is typable, then M is strongly normalising. The converse is not true. E.g. $\lambda x.xx$ is not typable.

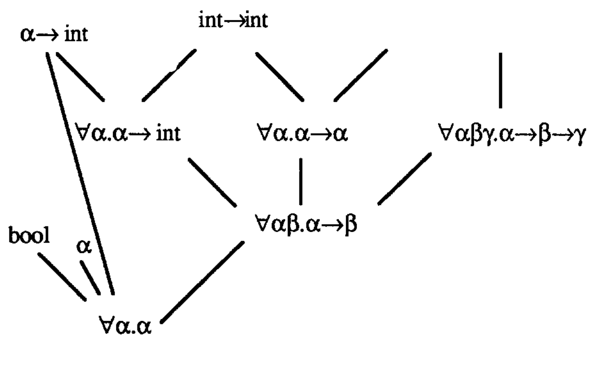
(iii) Computability of $\Gamma \vdash M : ?$ has been established independently by Hindley [1969] for a combinator version of $\lambda\!\to$ and by Milner [1978] for the closely related programming language ML. The ML-algorithm analyses the term M and generates a set E of equations between types in such a way that M is typable iff E has a solution. Moreover, the most general type of M corresponds to the most general solution of E. The latter solution can be found by applying a unification algorithm to E. A clear description of such an algorithm has been given by Wand [1987].

Pragmatics of $\lambda\!\to$ - Curry

There exist various programming languages with type systems closely resembling $\lambda\!\to$-Curry, such as ML (Harper [1986]), Hope (Burstall [1980]) and Miranda (Turner [1985]). The ML type system, invented by Milner [1978] independently from Hindley's work, has been discussed extensively in the literature (Damas [1982], Mycroft [1984], Kfoury [1988, 1989]). The system is based on an extension of $\lambda\!\to$-Curry with a declaration mechanism of the form **let** x = M **in** N, which allows a form of polymorphism in the sense that different occurrences of x in N may have different types which are instances of a type scheme for x and M. In this section we consider the basic system introduced by Milner and some of its extensions. All systems are based on the following sets of terms, types and type schemes.

$$\begin{aligned}
\Lambda \quad &= V \mid C \mid \lambda\, V.\Lambda \mid \textbf{let}\ V = \Lambda\ \textbf{in}\ \Lambda \mid \textbf{fix}\ V\,.\,\Lambda\\
\mathbb{T} \quad &= \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \to \mathbb{T}\\
\Sigma \quad &= \mathbb{T} \mid \forall \mathbb{V}.\Sigma
\end{aligned}$$

It follows that universal quantifiers may only occur at the top level of type schemes. In the remainder of this section we shall use metavariables τ and σ to range over types and type schemes, respectively. It will be necessary to consider 'generic instances' of type schemes. Let $\sigma = \forall\alpha_1...\alpha_m.\ \tau$ and $\sigma' = \forall\beta_1...\beta_n.\tau'$. Then σ' is a generic instance of σ, written $\sigma\le\sigma'$, iff there is a substitutor S acting only on $\{\alpha_1...\alpha_m\}$ sucht that $\tau' = S(\tau)$ and no β_i is free in σ. If we consider $\sigma{=}\sigma'$ iff $\sigma\le\sigma'\le\sigma$, then $(\Sigma, \le)$ is a partial order with least element $\forall\alpha.\alpha$. The following graph, taken from Mycroft [1984], illustrates the order $\le$.



The first type assignment system we consider for this language is the one of Milner [1978] in the form of Damas [1982]. Once more we note that τ ranges over types and σ over type schemes.

(var) $\Gamma \vdash x : \sigma$, if $(x : \sigma) \in \Gamma$;

(inst) $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma'} \qquad \text{if } \sigma \leq \sigma';$$

(gen) $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma'} \qquad \text{if } \alpha \notin FV(\Gamma);$$

(m-app) $$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2};$$

(m-abs) $$\frac{\Gamma, x:\tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2};$$

(m-fix) $$\frac{\Gamma, x:\tau \vdash M : \tau}{\Gamma \vdash \mathbf{fix}\ x.M : \tau};$$

(p-let) $$\frac{\Gamma, x:\sigma \vdash N : \tau \quad \Gamma \vdash M : \sigma}{\Gamma \vdash \mathbf{let}\ x{=}M\ \mathbf{in}\ N : \tau}.$$

The intended meaning of **fix** x.M is in terms of the type-free lambda calculus $\Theta(\lambda x.M)$. The meaning of **let** x=M **in** N is N[x:=M]. For practical purposes this basic language can be extended with other language constructs defined in terms of the given ones. For example, the construct

letrec x = M **in** N

can be defined as

let x = **fix** x.M **in** N.

Consider as an example the following declaration (taken from Milner [1978])

```
letrec map =
        λf, m.if        (null m)
                        nil
                        (cons (f (hd m)) (map f (tl m)) )
```

which is equivalent to

```
let map =
        fix map'.λf,m. if    (null m)
                             nil
                             (cons (f (hd m)) (map' f (tl m)) )
```

Assuming the following basis

null	:	$\forall\alpha.(\text{list } \alpha \to \text{bool})$
nil	:	$\forall\alpha.(\text{list } \alpha)$
cons	:	$\forall\alpha.(\alpha \to \text{list } \alpha \to \text{list } \alpha)$
hd	:	$\forall\alpha.(\text{list } \alpha \to \alpha)$
tl	:	$\forall\alpha.(\text{list}\alpha \to \text{list } \alpha)$
if	:	$\forall\alpha.(\text{bool} \to \alpha \to \alpha \to \alpha)$

one can deduce

map : $\forall\alpha\forall\beta.((\alpha\to\beta)\to(\text{list } \alpha\to\text{list } \beta))$.

To every occurrence of the variable map within the scope of the declaration must be assigned a type which is a generic instance of this type scheme. E.g., if we assume a type xxx and variables

x	:	list xxx
f	:	xxx $\to$ int
odd	:	int $\to$ bool

then in the expression

map odd (map f x)

the inner and outer occurrence of map will have types

$(\text{xxx}\to\text{int}) \quad \to(\text{list xxx} \to \text{list int})$

$(\text{int} \to \text{bool}) \ \to(\text{list int} \to \text{list bool})$

respectively.

The first extension to the system above has been suggested by Mycroft [1984] in order to remedy some unexpected consequences of the basic system. E.g. if the definition of map as given above is followed by a definition

let oddlist = λl.map odd l

then the ML system will derive the types

map	:	$\forall\alpha\forall\beta.\ ((\alpha\to\beta) \to (\text{list}.\alpha) \to (\text{list } \beta))$
oddlist	:	$(\text{list int}) \to (\text{list bool})$

whereas, if they are defined simultaneously by

letrec map = λf. m ...

and oddlist = λl. map odd l

the system will derive

$$\begin{array}{lll} \text{map} & : & (\text{int} \to \text{bool}) \to (\text{list int}) \to (\text{list bool}) \\ \text{oddlist} & : & (\text{list int}) \to (\text{list bool}). \end{array}$$

This phenomenon is due to the fact that a simultaneous recursive definition

$$\begin{array}{ll} \textbf{letrec } f_1 & = \lambda x_1.M_1 \\ \textbf{and } f_2 & = \lambda x_2.M_2 \end{array}$$

is equivalent to

$$\textbf{let} < f_1, f_2> = \textbf{fix} \ <f_1, f_2>.<M_1,M_2>$$

and according to the type assignment rule (m - fix) bound variables of a fix-expression should have types rather than type schemes. In Mycroft [1984] and Kfoury [1988] it has been suggested to replace rule (m - fix) by

$$(\text{p - fix}) \ \frac{\Gamma, x{:}\sigma \vdash M : \sigma}{\Gamma \vdash \textbf{fix}\, x\, .\, M : \sigma}$$

The resulting system is sometimes referred to as the Milner-Mycroft system. Mycroft could not prove the decidability of type derivation for this system. In Kfoury [1988] a proof is given, but it does not result in a practical algorithm.

A second extension to the basic system has been inspired by another problem, mentioned in Milner [1978] and Kfoury [1989]. Consider a declaration of the form

$$\textbf{let}\ m \quad = \lambda f.\lambda x.\lambda y.\ <f\ x,\ f\ y>.$$

One would expect that application of m to arguments

$$\begin{array}{ll} (\lambda x.x) & : \forall \alpha.\alpha \to \alpha \\ M_1 & : \tau_1 \\ M_2 & : \tau_2 \end{array}$$

results in

$$<M_1, M_2> \ : \tau_1 \times \tau_2$$

but in ML the term

$$\textbf{let}\ m = \lambda f.\lambda x.\lambda y.\ <fx, fy>\ \textbf{in}\ m\ (\lambda x.x)\ M_1 M_2$$

cannot be typed, because according to rule (m-abs) all occurrences of f should have the same type. In Kfoury [1989] it is proposed to replace (m-abs) and (m-app) by

$$(\text{p-abs}) \ \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

$$(\text{p-app}) \ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

thus allowing functions to accept polymorphic arguments. For the resulting system the computability of type derivation is open. In fact that system comes close to the system λ2-Curry, which is the subject of section 3.1.

2.2. THE EXPLICIT VERSION OF $\lambda\!\rightarrow$

Unless stated otherwise, in this subsection $\lambda\!\rightarrow$ stands for the explicit (Church) version of the simply typed lambda calculus.

Types and terms of $\lambda\!\rightarrow$ - Church

The set of types for this version of $\lambda\!\rightarrow$, notation $\mathbb{T}$ = Type $(\lambda\!\rightarrow)$, is the same as for the implicit version of $\lambda\!\rightarrow$, viz.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \rightarrow \mathbb{T}.$$

2.2.1 Definition. The set of pseudoterms of $\lambda\!\rightarrow$, notation $\Lambda_{\mathbb{T}}$, is defined as follows.

$$\Lambda_{\mathbb{T}} = V \mid C \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \lambda V{:}\mathbb{T}.\Lambda_{\mathbb{T}}.$$

For example $\lambda x{:}\alpha.x$ and $y\ (\lambda x{:}\alpha\rightarrow\alpha\cdot xy)$ are in $\Lambda_{\mathbb{T}}$. The first one will turn out to have a type, the second one not.

Assignment rules of $\lambda\!\rightarrow$ - Church.

2.2.2 Definition. $\Gamma \vdash_{\lambda\rightarrow} M : \sigma$ is defined by the following axiom and rules.

(start) $\quad \Gamma \vdash x : \sigma$, if $(x{:}\sigma) \in \Gamma$;

$(\rightarrow E)$ $$\frac{\Gamma \vdash M : (\sigma\rightarrow\tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau};$$

$(\rightarrow I)$ $$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma.M) : \sigma\rightarrow\tau}.$$

2.2.3 Examples.

(i) $\vdash_{\lambda\rightarrow} (\lambda x{:}\sigma\, \lambda y{:}\tau.x) : (\sigma\rightarrow\tau\rightarrow\sigma)$.

(ii) $\vdash_{\lambda\rightarrow} (\lambda x{:}\sigma\lambda y{:}\tau.y) : (\sigma\rightarrow\tau\rightarrow\tau)$.

(iii) $\vdash_{\lambda\rightarrow} (\lambda x{:}\sigma.x) : (\sigma\rightarrow\sigma)$.

(iv) $y{:}\sigma \vdash_{\lambda\rightarrow} ((\lambda x{:}\sigma.x)y) : \sigma$.

2.2.4 Definition. A pseudoterm M is called *legal* if for some Γ and σ one has $\Gamma \vdash_{\lambda\rightarrow} M : \sigma$.

For example $(\lambda x{:}\sigma.x)$ and $(\lambda x{:}\sigma.x)y$ are legal, but $(\lambda x{:}\alpha.xx)$ is not.

The notations of β-reduction and β-conversion are extended to pseudoterms. First of all the notion of substitution is extended to $\Lambda_{\mathbb{T}}$ in the obvious way. Then the contraction rule on pseudoterms

$$(\lambda x{:}\sigma.M)N \rightarrow M\,[x{:=}\,N] \qquad (\beta)$$

generates β-reduction, denoted again by $\twoheadrightarrow$, and β-conversion, denoted by =. Note that in (β) the term N does not need to match the type σ.

Properties of $\lambda\!\rightarrow$ - Church

2.2.5 Theorem. The following properties hold for $\lambda\!\rightarrow$.

(i) CR,for the extended notion of reduction and conversion on $\Lambda_{\mathbb{T}}$.

(ii) SR.

(iii) SN.

(iv) $\Gamma \vdash_{\lambda\rightarrow} M:\sigma$? is decidable.

(v) $\Gamma \vdash_{\lambda\rightarrow} M:?$ is computable.

Relation between $\lambda\!\rightarrow$-Curry and $\lambda\!\rightarrow$-Church

There is a relation between the implicit and explicit version of $\lambda\!\rightarrow$. In order to express this we need a map from pseudoterms to type-free terms.

2.2.6. Definition. A map $|\text{-}| : \Lambda_{\mathbb{T}} \rightarrow \Lambda$ is defined by erasing all type annotations.

$|x| = x$, $|c| = c$,

$|MN| = |M||N|$,

$|\lambda x{:}\sigma.M| = \lambda x.|M|$.

2.2.7. Proposition.

(i) $\Gamma \vdash_{\lambda\rightarrow\text{-Church}} M:\sigma \Rightarrow \Gamma \vdash_{\lambda\rightarrow\text{-Curry}} |M| : \sigma$.

(ii) $\Gamma \vdash_{\lambda\rightarrow\text{-Currry}} M:\sigma \Rightarrow \exists M' \, [\, |M'| \equiv M \;\&\; \Gamma \vdash_{\lambda\rightarrow\text{-Church}} M':\sigma]$.

Pragmatics of $\lambda\!\rightarrow$ - Church

Edinburgh LCF, a Logic of Computable Functions, see Gordon et al.[1979], is essentially $\lambda\!\rightarrow$-Church, extended with a fixed point combinator.

3. IMPLICIT TYPING

In this section three systems of typed lambda calculus à la Curry will be introduced. These systems are all extensions of the implicit version of $\lambda\!\rightarrow$. The three systems are the polymorphic lambda calculus λ2, the system with recursive types λμ and the system with intersection types $\lambda\cap$. The system λ2 will also be encountered in section 4.1 in an explicit version. A system à la Curry that includes all of λ2, λμ and $\lambda\cap$ as subsystems has been described in MacQueen, Plotkin and Sethi [1986].

3.1. POLYMORPHISM

In $\lambda\!\rightarrow$-Curry one has that $\vdash_{\lambda\rightarrow} (\lambda x.x) : (\sigma\rightarrow\sigma)$ for every type σ. The fact that λx.x has several types is called polymorphism. This can be made explicit by allowing as type $\forall\alpha.\alpha\rightarrow\alpha$ and as statement $(\lambda x.x)$:

$\forall\alpha.\alpha\to\alpha$. The system $\lambda 2$, introduced by Girard [1972] and Reynolds [1974] in the Church version, will be able to express this. In this subsection we will introduce $\lambda 2$-Curry.

Types and terms of $\lambda 2$- Curry

The terms of $\lambda 2$ are those of $\lambda\to$, see definition 1.1.1.

3.1.1 Definition. The types of $\lambda 2$, notation $\mathbb{T}$ = Type ($\lambda 2$), are defined by the following abstract grammar.

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

Notation. (i) $\forall\alpha_1\ldots\alpha_n.\sigma \equiv \forall\alpha_1.(\forall\alpha_2.\ \ldots(\forall\alpha_n.(\sigma))\ldots)$.
(ii) $\bot = \forall\alpha.\alpha$.

Examples of types
$\forall\alpha.\alpha\to\alpha$;
$\forall\alpha\beta.\alpha\to\beta$;
$\forall\alpha.\bot\to\beta$.

Assignment rules of $\lambda 2$-Curry

3.1.2. Definition.

(Start) $\Gamma\vdash x{:}\sigma$, for $(x{:}\sigma)\in\Gamma$;

$$(\to E)\quad \frac{\Gamma\vdash M:\sigma\to\tau,\ \Gamma\vdash N:\sigma}{\Gamma\vdash MN{:}\tau}; \qquad (\to I)\quad \frac{\Gamma, x{:}\sigma\vdash M:\tau}{\Gamma\vdash(\lambda x.M):\sigma\to\tau};$$

$$(\forall E)\quad \frac{\Gamma\vdash M:\forall\alpha.\sigma}{\Gamma\vdash M:\sigma[\alpha:=\tau]}; \qquad (\forall I)\quad \frac{\Gamma\vdash M:\sigma}{\Gamma\vdash M:\forall\alpha.\sigma},\ \text{if } \alpha\notin FV(\Gamma).$$

3.1.3 Examples

$$\vdash_{\lambda 2}(\lambda x.x) \ :(\forall\alpha.\alpha\to\alpha)$$
$$\vdash_{\lambda 2}(\lambda xy.x):(\forall\beta.\alpha\to\beta\to\alpha)$$
$$\vdash_{\lambda 2}(\lambda x.xx):(\forall\beta.\bot\to\beta)$$

This last fact has the following derivation

$$\begin{array}{l} \boxed{x : \bot} \\ \quad x : \bot \\ \quad x : \bot \to \beta \\ \quad xx : \beta \\ (\lambda x.xx) : \bot \to \beta \\ (\lambda x.xx) : \forall \beta.\ \bot \to \beta \end{array}$$

Properties of λ2-Curry

3.1.4. Theorem. The following properties hold for λ2.

(i) CR.

(ii) SR.

(iii) SN.

It is not known whether the properties

$\Gamma \vdash M : \sigma$? is decidable

$\Gamma \vdash M :$? is computable

are valid. (R. Milner calls them 'embarrassing open problems'.)

3.2. RECURSIVE TYPES

If we had a type σ such that $\sigma=\sigma\to\sigma$, and moreover an $x:\sigma$, then x can be applied to itself to obtain $xx:\sigma$. Therefore for such a σ one has $(\lambda x.xx) : (\sigma\to\sigma)$ and hence $(\lambda x.xx) : \sigma$. Solutions of the equation $\sigma=\sigma\to\sigma$ are obtained axiomatically by writing $\sigma\equiv\mu\alpha.\alpha\to\alpha$ which has to be interpreted like 'some solution of $\alpha=\alpha\to\alpha$'.

Types and terms of λμ-Curry

The terms of λμ are those of λ→, see definition 1.1.1.

3.2.1. Definition. The set of types of λμ, notation $\mathbb{T}$ = Type (λμ), is defined as follows.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \to \mathbb{T} \mid \mu\mathbb{V}.\,\mathbb{T}$$

Notation. $\mu\alpha_1 ... \alpha_n.\sigma \equiv \mu\alpha_1.(\mu\alpha_2. ... (\mu\alpha_n.(\sigma))..).$

Assignment rules of λμ-Curry

In order to state that the μ types satisfy what they are intended to do, some congruence relation $\approx$ on types is defined such that for example $\sigma \approx (\sigma\to\sigma)$ for $\sigma\equiv\mu\alpha.\alpha\to\alpha$. This relation $\approx$ is defined using trees corresponding to types. The definition of the trees is somewhat informal, but clear enough when seeing the examples.

3.2.2. Definition. Let $\sigma \in \text{Type}(\lambda\mu)$. The *tree* corresponding to σ, notation $T(\sigma)$, is defined as follows.

$T(\delta) = \delta,$ if δ is a type variable or constant;

$$T(\sigma\to\tau) = \begin{tikzcd} & \to \arrow[dl, dash] \arrow[dr, dash] & \\ T(\sigma) & & T(\tau) \end{tikzcd} ;$$

$T(\mu\alpha.\sigma) = T(\sigma[\alpha:=\mu\alpha.\sigma])$ if defined; else $\bullet$.

Examples.

$$T(\alpha\to\beta\to\alpha) = \begin{tikzcd} & \to \arrow[dl, dash] \arrow[dr, dash] & & \\ \alpha & & \to \arrow[dl, dash] \arrow[dr, dash] & \\ & \beta & & \alpha \end{tikzcd} ;$$

$$T(\mu\alpha.\alpha\to\alpha) = \begin{tikzcd} & \to \arrow[dl, dash] \arrow[dr, dash] & \\ T(\mu\alpha.\alpha\to\alpha) & & T(\mu\alpha.\alpha\to\alpha) \end{tikzcd} = \begin{tikzcd} & & & \to \arrow[dll, dash] \arrow[drr, dash] & & & \\ & \to \arrow[dl, dash] \arrow[dr, dash] & & & & \to \arrow[dl, dash] \arrow[dr, dash] & \\ \dots & & \dots & & \dots & & \dots \end{tikzcd} ;$$

$T(\mu\alpha.\alpha) = T(\alpha[\alpha:=\mu\alpha.\alpha]) = T(\mu\alpha.\alpha) = \bullet$;

$$T(\gamma\to\mu\alpha\beta.\beta) = \begin{tikzcd} & \to \arrow[dl, dash] \arrow[dr, dash] & \\ \gamma & & \bullet \end{tikzcd}$$

3.2.3. Definition. Let $\sigma,\tau \in \text{Type}(\lambda\mu)$. Then σ and τ are *equivalent*, notation $\sigma\approx\tau$, if $T(\sigma) = T(\tau)$.
Examples. If $\sigma \equiv \mu\alpha.\alpha\to\alpha$, then $\sigma\approx\sigma\to\sigma$. This could have been derived from the axiom

$$\mu\alpha.\sigma \approx \sigma[\alpha:=\mu\alpha.\sigma].$$

However, the following equation cannot be obtained from this axiom

$$\mu\alpha.\beta\to\alpha \approx \mu\alpha.\,\beta\to\beta\to\alpha.$$

Therefore the definition of $\approx$ is given using trees.

3.2.4. Definition.

(Start) $\Gamma \vdash x{:}\sigma$ for $(x{:}\sigma) \in \Gamma$

(→E) $\dfrac{\Gamma \vdash M : \sigma\to\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$; (→I) $\dfrac{\Gamma, x{:}\sigma \quad \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma\to\tau}$;

(≈) $\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \tau}$, if $\sigma \approx \tau$.

3.2.5. Examples.

(i) Morris [1968]. Let $Y \equiv \lambda f.\ (\lambda x.f(xx))(\lambda x.f(xx))$. Then $\vdash_{\lambda\mu}\ Y : (\sigma\to\sigma)\to\sigma$ for all σ. Indeed, define $\tau = \mu\alpha.\alpha\to\sigma$, then $\tau \approx \tau\to\sigma$ and we can construct the following derivation.

$\boxed{f : \sigma\to\sigma}$

  $\boxed{x : \tau}$

  $x : \tau$

  $x : \tau\to\sigma$

  $xx : \sigma$

  $f : \sigma\to\sigma$

  $f\,(xx) : \sigma :$

 $(\lambda x.f(xx)) : \tau\to\sigma$

 $(\lambda x.f(xx)) : \tau$

 $(\lambda x.\ f(xx))\ (\lambda x.\ f(xx)) : \sigma$

$(\lambda f(\lambda x.f\,(xx))\ (\lambda x.\ f(xx)) : (\sigma\to\sigma)\to\sigma$

$Y : (\sigma\to\sigma)\to\sigma$

(ii) Similarly, one shows $\vdash_{\lambda\mu} (\lambda x.xx)\ (\lambda x.xx) : \sigma$.

Properties of λμ-Curry

3.2.6 Theorem. The following properties hold for λμ.

(i) CR.

(ii) SR.

(iii) SN fails, (Y can be typed).

(iv) $\Gamma \vdash M : \sigma$? is decidable (Cardone and Coppo [1990]).

(v) $\Gamma \vdash M :$? is computable (trivial, using $\mu\alpha.\alpha\to\alpha$).

Mendler [1987] has shown that there exists a restricted version of λμ-Church in which all typable terms are strongly normalizing, the restriction being that in $\mu\alpha.\sigma$ the variable α may not occur at negative positions in σ. This result probably holds for λμ-Curry as well.

Pragmatics of λμ-Curry

Recursive types occur in many programming languages, but usually in combination with simple type constructors like + (disjoint sum) and × (cartesian product) only. For example, given a unit type **1** and a type τ, one can form the type of lists over τ as

$$\mu\alpha.\mathbf{1} + \tau \times \alpha$$

and the type of binary trees over τ as

$$\mu\alpha.\mathbf{1} + \tau \times \tau \times \alpha.$$

Few programming languages allow combinations of μ in full generality. One of the first languages to do so was ALGOL 68 (Van Wijngaarden [1969]), where e.g. the 'mode declaration'

mode m = proc (m) m

corresponds to $\mu\alpha.\alpha\to\alpha$. An illuminating discussion on ALGOL 68 mode declarations and their relation to domain equations is presented in Lehmann [1977].

3.3. INTERSECTION TYPES

The system of lambda calculus with intersection types, notation $\lambda\cap$, is taken from Barendregt, Coppo and Dezani [1983], which paper is based on earlier work of Coppo, Dezani, Sallé and Venneri. If in $\lambda\cap$ one has $M : \sigma$ and $M : \tau$, then one has $M : (\sigma\cap\tau)$. Moreover there is a universal type ω such that $M : \omega$ for all terms M. The operation $\cap$ induces a pre-ordering on types in which ω is the largest element.

Types and terms of λ∩

The terms of $\lambda\cap$ are those of $\lambda\to$, see definition 1.1.1.

3.3.1. Definition. The set of types of $\lambda\cap$, notation $\mathbb{T}$ = Type ($\lambda\cap$), is defined as follows

$$\mathbb{T} = \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \to \mathbb{T} \mid \mathbb{T} \cap \mathbb{T}.$$

One of the constants of $\mathbb{C}$ is selected and is called $\underline{\omega}$.

Assignment rules of λ∩- Curry

3.3.2. Definition (i) On $\mathbb{T}$ a relation $\leq$ is defined as follows

$$\sigma \leq \sigma;$$

$$\sigma \leq \omega; \qquad \omega \leq \omega \to \omega;$$

$$\sigma \cap \tau \leq \sigma; \qquad \sigma \cap \tau \leq \tau;$$

$$\frac{\sigma \leq \tau,\ \tau \leq \rho}{\sigma \leq \rho}; \qquad \frac{\sigma \leq \tau,\ \sigma \leq \rho}{\sigma \leq \tau \cap \rho};$$

$$(\sigma\to\rho) \cap (\sigma\to\tau) \leq \sigma \to (\rho \cap \tau);$$

$$\frac{\sigma \leq \sigma', \sigma \leq \tau'}{(\sigma' \to \tau) \leq (\sigma \to \tau')}.$$

(ii) $\sigma \sim \tau \Leftrightarrow \sigma \leq \tau$ and $\tau \leq \sigma$.

For example, one has $\omega \sim (\omega \to \omega)$ and $((\sigma \cap \sigma' \to \tau) \sim ((\sigma \to \tau) \cap (\sigma' \to \tau))$. We let $\cap$ bind stronger than $\to$, so one can write $\sigma \cap \sigma' \to \tau$ for $(\sigma \cap \sigma') \to \tau$.

3.3.3. Definition.

(Start) $\Gamma \vdash x : \sigma$, for $(x{:}\sigma) \in \Gamma$;

(top) $\Gamma \vdash M : \omega$;

$(\to E)$ $$\frac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau};$$ $(\to I)$ $$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau};$$

$(\cap E)$ $$\frac{\Gamma \vdash M : (\sigma \cap \tau)}{\Gamma \vdash M : \sigma , \Gamma \vdash M : \tau};$$ $(\cap I)$ $$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau};$$

(sub) $$\frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau}.$$

3.3.4. Examples.

(i) $\vdash_{\lambda\cap} (\lambda x.xx) : (\sigma \cap (\sigma \to \tau) \to \tau)$;

(ii) $\vdash_{\lambda\cap} (\lambda x.xx)(\lambda x.xx) : \omega$;

(iii) $\vdash_{\lambda\cap} (\lambda xyz.xz\,(yz))(\lambda xy.x) : (\alpha \to \beta \to \beta)$.

In Van Bakel [1990] it is remarked that although the term in (iii) has a type in $\lambda\to$, it does not have the type $\alpha \to \beta \to \beta$ in that system.

Properties of $\lambda\cap$- Curry

The properties valid for $\lambda\cap$ are somewhat different from those for the other systems of implicitly typed lambda calculus.

3.3.5. Theorem. For $\lambda\cap$ the following properties hold.

(i) CR.

(ii) SR.

The stronger notion of Subject Conversion also holds, i.e. if $M =_\beta M'$ then

$$\Gamma \vdash_{\lambda\cap} M : \sigma \;\Leftrightarrow\; \Gamma \vdash_{\lambda\cap} M' : \sigma$$

(iii) SN fails, because every term has type ω.
(iv) $\Gamma \vdash M: \sigma$? is undecidable.
(v) $\Gamma \vdash M:$? holds trivially, because every term has type ω.

3.3.6. Remark (i) For the system $\lambda\cap^{-}$, obtained form $\lambda\cap$ by omitting rule (top), one has the following result of Van Bakel [1990]

$$\mathrm{SN}\,(M) \Leftrightarrow \exists\,\Gamma\,\exists\,\sigma\,[\Gamma \vdash_{\lambda\cap^{-}} M : \sigma]$$

(ii) M has normal form $\Leftrightarrow \exists\,\Gamma\,\exists\,\sigma\,[\Gamma \vdash M : \sigma$ and ω does not occur in $\sigma]$,
see Barendregt et al. [1983].

Pragmatics of $\lambda\cap$

In $\lambda\cap$ intersection types are used for objects that have various types which are not structurally related. The same phenomenon occurs in many programming languages, where it is called 'overloading'. A well-known example is the use of the symbol + to denote both integer addition, real addition and string concatenation. In the intersection type discipline such a symbol can be given the type (**int**→**int**) $\cap$ (**real**→**real**) $\cap$ (**string**→**string**).

Recently new uses of intersection types have been suggested by Reynolds [1989] and Tennent [1989] in connection with ALGOL-like languages. The imperative features of such languages are accomodated by the introduction of so-called phrase types, such as

exp $[\tau]$	for constructs producing a value of type τ
acc $[\tau]$	for constructs accepting a value of type τ, such as the updating component of variables or result parameters of procedures. In denotational semantics these constructors are sometimes called 'l-values'.
var $[\tau]$	for variables of type τ.

For example, in an assignment x : = 3, the left-hand side and right-hand side have phrase types **acc** [**int**] and **exp** [**int**], respectively.

Because in ALGOL-like languages a variables of type τ can both accept and produce a value of type τ, we need the coercions **var** $[\tau] \leq$ **acc** $[\tau]$ and **var** $[\tau] \leq$ **exp** $[\tau]$. Using intersection types this can be achieved by defining

$$\mathbf{var}\,[\tau] = \mathbf{acc}\,[\tau] \cap \mathbf{exp}\,[\tau].$$

Another phenomenon occuring in ALGOL-like languages is interference, e.g. when one command assigns to a variable that is evaluated or assigned to by an other command. In some cases this interference is undesirable, but in other cases it is very useful or even essential. What is needed are some syntactic constraints to control interference. It is possible to define these constraints using a variant of intersection types. For more details we refer to Reynolds [1989].

4. EXPLICIT TYPING

In this section several extensions of $\lambda{\to}$-Church will be considered. Three important extensions are $\lambda 2$-Church (second order lambda calculus), the sytstem $\lambda\underline{\omega}$ (weakly higher order lambda calculus) and the system λP (lambda calculus with dependent types). The intuition behind these extensions is the following. Types represent sets (spaces) of functions. Terms denote algorithms representing elements of these function spaces. Now there are four basic dependencies.

- elements depend on elements;
- elements depend on types;
- types depend on types;
- types depend on elements.

These four dependencies will be explained as the main features of respectively the systems $\lambda{\to}$, $\lambda 2$, $\lambda\underline{\omega}$ and λP.

Elements depending on elements are ubiquitous. A function f of two arguments on a set A may be used to form fxx for x in A. This fxx is an element depending on the element x. The mechanism of abstraction makes this dependency functional. One can form the function $g = \lambda x(:A).fxx$. This formation of functions is the main feature of all versions of the lambda calculus, in particular of $\lambda{\to}$. The other kind of dependencies will be described in subsections 4.1, 4.2 and 4.3 respectively. Subsection 4.4 gives a uniform treatment of the explicity typed systems.

4.1 ELEMENTS DEPENDING ON TYPES

In $\lambda{\to}$-Church one has

$$\vdash_{\lambda\to} (\lambda x{:}\alpha.x) : (\alpha\to\alpha). \qquad (1)$$

If one writes $I_\sigma \equiv (\lambda x{:}\sigma.x)$, then I_σ is an element that depends on the type σ. The system $\lambda 2$ - Chuch makes this depending functional by deriving from (1)

$$\vdash_{\lambda 2} (\Lambda\alpha\lambda x{:}\alpha.x) : (\forall\alpha.(\alpha\to\alpha)).$$

Write $I_{poly} \equiv \Lambda\alpha.\ I_\alpha$. In order to obtain I_σ uniformly in σ from I_{poly}, a new reduction rule is introduced

$$I_{poly}\ \sigma \equiv (\Lambda\alpha.\ I_\alpha)\sigma \to I_\sigma. \qquad (\beta_2)$$

In particular this implies that terms may be applied to types. In subsection 4.4 it will be seen that (β_2) can be considered as a particular case of ordinary β-reduction.

Types and terms of $\lambda 2$-Church.

4.1.1. Definition. (i) The set of types for $\lambda 2$-Church, notation $\mathbb{T}$ = Type $(\lambda 2)$, is the same as for the Curry version

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

(ii) The set of pseudoterms for $\lambda 2$-Church, notation $\Lambda_{\mathbb{T}}$ = Term$(\lambda 2)$ is defined as follows.

$$\Lambda_{\mathbb{T}} = V \mid C \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \Lambda_{\mathbb{T}}\,\mathbb{T} \mid \lambda V{:}\mathbb{T}.\Lambda_{\mathbb{T}} \mid \Lambda\mathbb{V}.\Lambda_{\mathbb{T}}.$$

(iii) On these pseudoterms $\Lambda_{\mathbb{T}}$ two notions of reduction are defined by

$$(\lambda x{:}\sigma.M)N \;\to\; M[x := N], \qquad (\beta)$$
$$(\Lambda\alpha.M)\sigma \;\to\; M[\alpha := \sigma]. \qquad (\beta_2)$$

4.1.2 Convention. $\Lambda\alpha\lambda x{:}\sigma\Lambda\beta.M$ stands for $\Lambda\alpha.(\lambda x{:}\sigma.(\Lambda\beta.(M)))$.

For example $(\Lambda\alpha\Lambda\beta\lambda x{:}\alpha\,\lambda y{:}\beta.x)(\gamma\to\gamma)(\forall\delta.\delta)ab$ is a pseudoterm that reduces in four $\beta\beta_2$-steps to a.

Assigment rules of λ2- Church.

4.1.3. Definition. $\Gamma\vdash_{\lambda 2} M{:}\sigma$ is defined by the following axiom and rules

(Start) $\Gamma\vdash x{:}\sigma$, if $(x{:}\sigma)\in\Gamma$;

$$(\to E)\ \frac{\Gamma\vdash M:(\sigma\to\tau)\quad \Gamma\vdash N:\sigma}{\Gamma\vdash MN:\tau}; \qquad (\to I)\ \frac{\Gamma, x{:}\sigma\vdash M:\tau}{\Gamma\vdash(\lambda x.M):(\sigma\to\tau)};$$

$$(\forall E)\ \frac{\Gamma\vdash M:\forall\alpha.\sigma}{\Gamma\vdash M\tau:\sigma[\alpha:=\tau]}; \qquad (\forall I)\ \frac{\Gamma\vdash M:\sigma}{\Gamma\vdash(\Lambda\alpha.M):(\forall\alpha.\sigma)}, \text{ if } \alpha\notin FV(\Gamma).$$

4.1.4. Examples. (i) $\vdash_{\lambda 2} (\Lambda\alpha\lambda x{:}\alpha.x) : (\forall\alpha.\alpha\to\alpha)$.
(ii) $\vdash_{\lambda 2} (\Lambda\alpha\,\Lambda\beta\,\lambda x{:}\alpha\,\lambda y{:}\beta.x) : (\forall\alpha\forall\beta.\alpha\to\beta\to\alpha)$.
(iii) $\vdash_{\lambda 2} (\Lambda\beta\,\lambda x{:}\bot.\,x(\bot\to\beta)x) : (\forall\beta.\,\bot\to\beta)$, where $\bot \equiv \forall\alpha.\alpha$.

Properties of λ2- Church

4.1.5 Theorem. The following properties are valid for λ2 (see 2.1.5 for the meaning of the acronyms).
(i) CR, for $\beta\beta_2$ - reduction on Λ_T.
(ii) SR.
(iii) SN.
(iv) UT.
(v) $\Gamma\vdash M:\sigma?$ is decidable.
(vi) $\Gamma\vdash M:?$ is computable.

Pragmatics of λ2 - Church.

In Reynolds [1974] the system λ2 was used to formalise various type-related concepts in programming languages, such as type definitions, abstract data types, and polymorphism. For example, an expression containing a type definition, like

type t = τ **in** e

can be represented by the λ2- term $(\Lambda t.e)\,\tau$.

Many programming languages contain some form of abstract data type definition like

abstype t **with** x : σ (t) **is** τ **with** e_1 **in** e_2

which introduces an "abstract" type t with operations x : σ(t), together with a "representation" consisting of a "concrete" type τ and "concrete" operations e_1 of type σ (t). In $\lambda 2$ such a construct can be modelled by the term

$(\Lambda t\ (\lambda x{:}\sigma(t).\ e_2))\ \tau\ e_1$

In extensions of $\lambda 2$ with so-called existential or Σ types it is even possible to consider the pair (τ, e_1) as a term of type $\Sigma t.\sigma(t)$, the type of representations of the abstract type. Thus representations become first-class citizens. For more information on this view of abstract data types we refer to Mitchell [1985], Cardelli [1985].

For an experimental language based on the Church version of $\lambda 2$ and $\lambda\mu$, see Barendregt and van Leeuwen [1985]. That language (TALE) includes arrays and disjoint sums and its syntax and operational semantics are given in full detail.

The polymorphism provided by $\lambda 2$-Church differs from that of $\lambda 2$-Curry in the presence of explicit abstraction over type variables and application to type expressions. To illustrate the differences we rephrase the map example of section 2 in the system à la Church. The basis remains

null	:	$\forall\alpha.$ list $\alpha\to$bool
nil	:	$\forall\alpha.$ list α
cons	:	$\forall\alpha.\ \alpha\to$list $\alpha\to$list α
hd	:	$\forall\alpha.$ list $\alpha\to\alpha$
tl	:	$\forall\alpha.$ list $\alpha\to$list α
if	:	$\forall\alpha.$ bool $\to\alpha\to\alpha\to\alpha$

but the definition of map becomes

```
letrec map = (ΛαΛβ
                  . (λf : (α→β) λm : list α
                  . if list β
                        (null α m)
                        (nil β)
                        (cons β (f(hd α m)) (map α β f (tl α m)))
                        )
                    ).
```

The differences with the Curry style will be clear. On the positive side, type inference is trivial. On the negative side, terms tend to become cluttered up with type information. For practical purposes one would like to have the best of both worlds, i.e. the conciseness provided by the Curry style and the simplicity of type inference provided by the Church style. Some partial solutions have been presented in McCracken

[1984], Boehm [1989], O'Toole 89 [1989], Pfenning [1988]. Although TALE is explicitly typed, type information may be left out in clearly defined cases when it can be reconstructed.

4.2. TYPE DEPENDENT TYPES

A natural example of a type depending on another type is $\alpha\to\alpha$ depending on α. In fact it is natural to define $f = \lambda\alpha\in\mathbb{T}.\alpha\to\alpha$ such that $f(\alpha) = \alpha\to\alpha$. This will be possible in the system $\lambda\underline{\omega}$.

Another feature of $\lambda\underline{\omega}$ is that types are generated by the system itself and not in the informal metalanguage. There is a constant $*$ such that $\sigma : *$ correspondends to $\sigma\in\mathbb{T}$. The informal statement

$$\alpha, \beta \in \mathbb{T} \Rightarrow (\alpha\to\beta) \in \mathbb{T}$$

now becomes the formal

$$\alpha : *, \beta : * \vdash (\alpha\to\beta) : *.$$

For the f above we then write $f \equiv \lambda\alpha : *.\ \alpha\to\alpha$. The question arises where this f lives. Neither on the level of the elements (of types), nor among the types. Therefore a new category K (of kinds) is introduced

$$K = * \mid K \to K.$$

That is $\mathbb{K} = \{*, *\to*, *\to*\to*, \ldots\}$. A constant $\Box$ will be introduced such that $k : \Box$ corresponds to $k \in \mathbb{K}$. If $\vdash k : \Box$ and $\vdash F : k$, then F is called a *constructor* of kind k. We will see that $\vdash (\lambda\alpha{:}*.\alpha\to\alpha) : (*\to*)$, i.e. our f is a constructor of kind $*\to*$. Each element of $\mathbb{T}$ will be a constructor of kind $*$.

Types and terms of $\lambda\underline{\omega}$

Although types and terms of $\lambda\underline{\omega}$ can be kept separate, we will consider them as subsets of one general set $\mathbf{T}$ of pseudo expressions. This is a preparation to 4.3 and 4.4 in which it is essential that types and terms are being mixed.

4.2.1. Definition. (i) A set of pseudo expressions $\mathbf{T}$ is defined as follows.

$$\mathbf{T} = V \mid C \mid \mathbf{T}\,\mathbf{T} \mid \lambda V{:}\mathbf{T}.\mathbf{T} \mid \mathbf{T} \to \mathbf{T}$$

(ii) Among the constants C two elements are selected and given the names $*$ and $\Box$.

4.2.2. Notation. (i) x, y, z, ..., α, β, γ... range over V.

(ii) **a**, **b**, **c**,..., $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ ...range over C.

Assigment rule of $\lambda\underline{\omega}$.

Because types and terms come from the same set $\mathbf{T}$, the definition of statement is modified accordingly. Bases have to become linearly ordered. The reason is that in $\lambda\underline{\omega}$ one wants to derive

$$\alpha{:}*, x{:}\alpha \vdash x{:}\alpha$$

$$\alpha{:}* \vdash (\lambda x{:}\alpha.x) : (\alpha\to\alpha)$$

but not

$$x{:}\alpha,\ \alpha{:}* \vdash x{:}\alpha$$

$$x{:}\alpha \vdash (\lambda\alpha{:}*.x) : (*\to\alpha)$$

in which α occurs both free and bound.

4.2.3. Definition. (i) A *statement* of $\lambda\underline{\omega}$ is if the form $M : A$ with $M, A \in \mathbf{T}$.

(ii) A *context* is a finite linearly ordered set of statements with distinct variables as subjects. Γ, Δ,... range over contexts.

(iii) $< >$ denotes the emtpy context. If $\Gamma = < x_1{:}A_1,\dots, x_n{:}A_n>$ then $\Gamma, y{:}B = < x_1{:}A_1,\dots, x_{n}{:}A_n, y{:}B>$.

4.2.4. Definition. The notion $\Gamma \vdash_{\lambda\underline{\omega}} M : A$ is defined by the following axiom and rules. The letter s ranges over $\{*, \square\}$.

(Axiom) $\quad < > \vdash * : \square;$

(Start) $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}, \text{ where x is fresh, i.e. not in } \Gamma \text{ or } A;$$

(Weakening) $$\frac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}, \text{ where x is fresh;}$$

(Type/ kind formation) $$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash (A \to B) : s};$$

($\to$E) $$\frac{\Gamma \vdash F : (A \to B) \;\; \Gamma \vdash a : A}{\Gamma \vdash Fa : B};$$

($\to$I) $$\frac{\Gamma \vdash A : s, \;\; \Gamma, x{:}A \vdash B : s \quad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash (\lambda x{:}A.b) : (A \to B)};$$

(Conversion rule) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}, \text{ if } B =_\beta B'.$$

4.2.6. Examples. (i) $\alpha{:}*, \beta{:}* \vdash_{\lambda\underline{\omega}} (\alpha \to \beta) : *$.

(ii) $\alpha{:}*, \beta{:}*, x{:}(\alpha \to \beta) \vdash_{\lambda\underline{\omega}} x : (\alpha \to \beta)$.

(iii) $\alpha{:}*, \beta{:}* \vdash_{\lambda\underline{\omega}} (\lambda x{:}(\alpha \to \beta) .x) : ((\alpha \to \beta) \to (\alpha \to \beta)$.

Write $D \equiv \lambda\beta{:}*.\ \beta \to \beta$. Then the following hold.

(iv) $\vdash_{\lambda\underline{\omega}} D : (* \to *)$.

(v) $\alpha{:}* \vdash_{\lambda\underline{\omega}} (\lambda x{:}D\alpha .x) : D\,(D\alpha)$.

Properties of $\lambda\underline{\omega}$

4.2.7. Theorem. The following results hold for $\lambda\underline{\omega}$.

(i) CR, for reduction on $\mathbf{T}$.
(ii) SR.
(iii) SN, i.e. $\Gamma \vdash M : A \Rightarrow SN(M) \,\&\, SN(A)$.
(iv) UT, i.e. $\Gamma \vdash M : A \,\&\, \Gamma \vdash M : A' \Rightarrow A =_{\beta} A'$.
(v) $\Gamma \vdash M : A?$ is decidable.
(vi) $\Gamma \vdash M : ?$ is computable.

Pragmatics of $\lambda\underline{\omega}$

The relevance of $\lambda\underline{\omega}$ for programming languages is that the system permits computations with types and definitions of type constructors. Consider the following example.

4.2.8. Example.
If we assume constructor declarations

$1 : *$	$\text{int} \ : *$
$\times : * \to * \to *$	$\text{bool} : *$
$+ : * \to * \to *$	

and recursive types of the form $\mu\alpha{:}*.\sigma$ similarly to section 3.2. we can from the type of integer lists as

$$\mu\beta{:}*.\ 1 + \text{int} \times \beta.$$

The 1 stands for the singleton type {nil}. Similarly for boolean lists we take

$$\mu\beta{:}*.\ 1 + \text{bool} \times \beta.$$

These two types can be obtained by applying a certain constructor of kind $* \to *$ to int and bool respectively, viz. the constructor

$$(\lambda\alpha{:}*\ \mu\beta{:}*\ .\ 1 + \alpha \times \beta) : * \to *.$$

Using a **let** definition mechanism for constructors, we can form the type of functions form integer lists to boolean lists as

$$\textbf{let}\ \text{list} = (\lambda\alpha{:}*\ \mu\beta{:}*\ .\ 1 + \alpha \times \beta)$$
$$\textbf{in}\ \text{list int} \to \text{list bool}$$

which is an abbreviation for

$$(\lambda\text{list}{:}* \to *.\ \text{list int} \to \text{list bool})\ (\lambda\alpha{:}\text{type}\ \mu\beta{:}*.\ 1 + \alpha\ \times \beta)$$

where

$$(\lambda\text{list}{:}* \to *.\ \text{list int} \to \text{list bool}) : (* \to *) \to *$$

and which β-reduces to

$$(\lambda\alpha{:}*\mu\beta{:}*\ 1 + \alpha \times \beta)\ \text{int} \to (\lambda\alpha{:}*\mu\beta{:}*.\ 1 + \alpha \times \beta)\ \text{bool}$$

and subsequently to

$$(\mu\beta{:}\ *.1 + \text{int} \times \beta) \to (\mu\beta{:}\ *.1 + \text{bool} \times \beta).$$

The <u>let</u> -construction in the example above is of the form **let** $\alpha = \rho$ **in** σ which is an abbrevation for

$$(\lambda\alpha{:}k.\sigma)\rho$$

where σ and ρ are both constructors. In order to have the full benefit of the definition facility for constructors we also need

$$\textbf{let } \alpha = \rho \textbf{ in } M$$

which is an abbrevation for

$$(\lambda\alpha{:}k.M)\,\rho$$

where ρ is a constructor and M is a term. This construction is not well-formed in $\lambda\underline{\omega}$. It can be formed in the system $\lambda 2$ in the special case that $k \equiv *$. The wish to form terms $(\lambda\alpha{:}k\,.M)$ where k is an arbitrary kind leads to the system $\lambda\omega$, which is the "union" of $\lambda\underline{\omega}$ and $\lambda 2$. The system $\lambda\omega$ will be defined in section 4.4.

4.3 ELEMENT DEPENDENT TYPES

An intuitive example of a type depending on an element is $A^n \to B$ with $n \in \mathrm{Int}$. In order to formalise the possibility of such "dependent types" in the system λP, the notion of kind is extended such that if A is a type and k is a kind, then $A \to k$ is a kind. In particular $A \to *$ is a kind. Then if $f{:}A \to *$ and $a{:}A$, one has $fa : *$. This fa is a dependent type.

Another idea important for a system with dependent types is the formation of cartesian products. Suppose that for each a:A a type B_a is given and that there is an element $b_a : B_a$. Then we may form the function $\lambda a{:}A.b_a$ which has as type the cartesian product $\Pi a{:}A.B_a$ of the B_a's. The formation of function spaces $A \to B$ can be seen as a particular instance of cartesian products. Indeed,

$$A \to B \equiv \Pi a{:}A.B \ (\equiv B^A \text{ informally})$$

provided that a does not occur (freely) in B. This is similar to the fact that a power of numbers is a constant product

$$\prod_{i=1}^{3} b_i = b_1 b_2 b_3 = b^3$$

provided that $b_1 = b_2 = b_3 = b$. Therefore the type constructor $\to$ can be left out in the presence of Π.

Types and terms of λP.

4.3.1 Definition. (i) The set of pseudo expressions of λP, notation $\mathcal{T}$, is defined as follows.

$$\mathcal{T} = V \mid C \mid \mathcal{T}\mathcal{T} \mid \lambda V{:}\mathcal{T}.\mathcal{T} \mid \Pi V{:}\mathcal{T}.\mathcal{T}$$

(ii) Among the constants C two elements are called $*$ and $\Box$.

Assigment rules for λP

Statements and contexts are defined as for $\lambda\underline{\omega}$ (statements are of the form $M : A$ with $M, A \in \mathcal{T}$; contexts are finite linearly ordered sets of statements).

4.3.2. Definition. The notion $\Gamma \vdash_{\lambda P} M : A$ is defined by the following axiom and rules. Again the letter s ranges over $\{*, \Box\}$.

(Axiom) $<> \vdash * : \square;$

(Start) $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$, where x is fresh;

(Weakening) $\dfrac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash B : C}$, where x is fresh;

(Type/ kind formation) $\dfrac{\Gamma \vdash A : * \quad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash (\Pi x{:}A.B) : s};$

($\to$I) $\dfrac{\Gamma \vdash A : * \quad \Gamma, x{:}A \vdash B : s \quad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)};$

($\to$E) $\dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a.: A}{\Gamma \vdash Fa : B[x:= a]};$

(Conversion) $\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$, if $B =_\beta B'$.

4.3.3. Examples. In λP the following hold.

(i) $A{:}* \vdash (A \to *) : \square$.

(ii) $A{:}*, P{:}(A \to *), a{:}A \vdash Pa : *$.

(iii) $A{:}*, P{:}(A \to *), a{:}A \vdash (Pa \to *) : \square$.

(iv) $A{:}*, P{:}(A \to *) \vdash (\Pi\, a{:}A.\ Pa \to *) : \square$.

(v) $A{:}*, P{:}(A \to *) \vdash (\lambda a{:}A\ \lambda p{:}Pa.p) : (\Pi a{:}A.Pa \to Pa)$.

Properties of λP.

4.3.4. Theorem. The following results hold for λP.

(i) CR, for reduction on $\mathbf{T}$.

(ii) SR.

(iii) SN, i.e. $\Gamma \vdash M : A \Rightarrow SN(M) \,\&\, SN(A)$.

(iv) UT.

(v) $\Gamma \vdash M: A?$ is decidable.

(vi) $\Gamma \vdash M: ?$ is computable.

Pragmatics of λP

Systems like λP have been introduced by N.G. de Bruijn [1970], [1980] in order to represent mathematical theorems and their proofs. The method is as follows.

One assumes there is a set prop of propositions that is closed under implication. This is done by taking as context Γ_0 defined as

$$\text{prop} : *, \text{ imp} : \text{prop} \to \text{prop} \to \text{prop}.$$

Write $\varphi \supset \psi$ for $\underline{\text{imp}}\ \varphi\ \psi$. In order to express that a proposition is valid a $T : \underline{\text{prop}} \to *$ is assumed and φ:prop is defined to be valid if $T\varphi$ is inhabited, i.e. $M : T\varphi$ for some M. Now in order to express that implication has the right properties, one assumes $\supset_e$ and $\supset_i$ such that

$$\supset_e \varphi\ \psi : T(\varphi \supset \psi) \to T\varphi \to T\psi.$$

$$\supset_i \varphi\ \psi : T\varphi \to T\psi \to T(\varphi \supset \psi).$$

So for the representation of implicational proposition logic one wants to work in context Γ_{prop} consisting of Γ_0 followed by

$$T : \underline{\text{prop}} \to *$$

$$\supset_e : \Pi\varphi{:}\underline{\text{prop}}\ \Pi\ \psi{:}\underline{\text{prop}}.\ T(\varphi \supset \psi) \to T\ \varphi \to T\psi$$

$$\supset_i : \Pi\varphi{:}\underline{\text{prop}}\ \Pi\ \psi{:}\underline{\text{prop}}.\ (T\varphi \to T\psi) \to T(\varphi \supset \psi).$$

As an example we want to formulate that $\varphi \supset \varphi$ is valid for all propositions. The translation as type is $T(\varphi \supset \varphi)$ which indeed is inhabited

$$\Gamma_{\text{prop}} \vdash_{\lambda P} (\supset_i \varphi\varphi\ (\lambda x{:}T\varphi.x)) : T(\varphi \supset \varphi).$$

(Note that since $\vdash T\varphi : *$ one has $\vdash (\lambda x{:}T\varphi.x) : (T\varphi \to T\varphi)$.)

Having formalised many valid statements de Bruijn realised that it was rather tiresome to carry around the T. He therefore proposed to use $*$ itself for prop, the constructor $\to$ for $\supset$ and the identity for T. Then for $\supset_e \varphi\ \psi$ one can use

$$\lambda x{:}(\varphi \to \psi)\ \lambda y{:}\varphi.xy$$

and for $\supset_i \varphi\ \psi$

$$\lambda x{:}(\varphi \to \psi).x.$$

In this way the $\{\to, \forall\}$ fragment of (manysorted constructive) predicate logic can be interpreted too. A predicate P on a set (type) A can be represented as a $P{:}A \to *$ and for a:A one defines Pa to be valid if it is inhabited. Quantification $\forall x{\in}A.Px$ is translated as $\Pi x{:}A.Px$. Now a formula

$$\forall x{\in}A\ \forall y{\in}A.Pxy \to \forall x{\in}A\ Pxx$$

can be seen to be valid because its translation is inhabited

$$A{:}*, P{:}A \to * \vdash (\lambda z{:}(\Pi x{:}A\ \Pi y{:}A\ Pxy)\lambda x{:}A.zxx) : (\Pi x{:}A\Pi y{:}A.Pxy \to \Pi x{:}A.Pxx).$$

The system λP is given that name because predicate logic can be interpreted in it. The method interprets propositions (or formulas) as types and proofs as inhabiting terms and is the basis of several languages in the family AUTOMATH designed and implemented by de Bruijn and co-workers for the automatic verification of proofs. Similar projects inspired by AUTOMATH are described in Constable et al. [1986] (NUPRL), Harper et al. [1987] (LF) and Coquand et al. [1989] (calculus of constructions). The project LF uses the interpretation of formulas using T:prop $\to *$ like the original use in AUTOMATH.

4.4 GENERALIZED TYPE SYSTEMS

So far we have introduced four systems of typed lambda calculus à la Church, viz. $\lambda\!\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$ and λP. Although it may not seem so, there is a uniform way of describing these systems. We will define the notation 'generalised type system' (GTS) and show how the four systems in question are particular cases. The differentiation of systems is obtained by controlling which abstractions are allowed.

4.4.1. Definition. (i) The set of pseudo expressions of a GTS, notation $\mathcal{T}$

$$\mathcal{T} = V \mid C \mid \mathcal{T}\,\mathcal{T} \mid \lambda V{:}\mathcal{T}.\mathcal{T} \mid \Pi V{:}\mathcal{T}.\mathcal{T}$$

(ii) A *statement* of a GTS is of the form $M : A$ with $M, A \in \mathcal{T}$. M is called the *subject* and A the *predicate*.

(iii) A *context* is a finite linearly ordered sequence of statements with as subjects distinct variables.

4.4.2. Definition. A *specification* of a GTS is a triple $\mathbf{S} = (S,A,R)$ such that

- $S \subseteq C$, the elements of S are called *sorts*
- A is a set of statements of the form $c{:}s$ with $c \in C$ and $s \in S$; the elements of A are called *axioms*.
- R is a set of pairs of the form (s_1, s_2) with $s_1, s_2 \in S$; the elements of R are called *rules*.

4.4.3. Definition. Given a specification of a GTS $\mathbf{S}= (S,A,R)$, the corresponding GTS, notation $\lambda\mathbf{S}$, derives statements relative to a context Γ. The rules $(s_1, s_2) \in R$ determine which abstraction are allowed.

(Axiom) $\quad < > \vdash c : s, \qquad$ if $(c{:}s) \in A$;

(Start) $\quad \dfrac{\Gamma \vdash A : s}{\Gamma,\ x{:}A \vdash x : A}$, if $s \in S$ and x is fresh;

(Weakening) $\quad \dfrac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma,\ x{:}A \vdash B : C}$, if $s \in S$ and x is fresh;

(Π-elimination) $\quad \dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$;

(Π-formation) $\quad \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_2}$, if $(s_1, s_2) \in R$;

(Π-introduction) $\quad \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2 \quad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)}$, if $(s_1, s_2) \in R$;

(Conversion) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}, \text{ if } s \in S \text{ and } B =_{\beta} B'.$$

4.4.4. Examples.

(i) Consider the following specification

$S = \{*, \square\}$

$A = \{* : \square\}$

$R = \{(*,*), (*, \square)\}$

Such a specification is written stylistically as

S	$*, \square$
A	$* : \square$
R	$(*,*), (*, \square)$

This system λ(S,A,R) is the same as λP.

(ii) Consider

S	$*, \square$
A	$* : \square$
R	$(*,*), (\square, \square)$

Then λ(S,A,R) is $\lambda\underline{\omega}$.

4.4.5. Proposition.

(i) Consider the following specification

S	$*, \square$
A	$* : \square$
R	$(*,*)$

Then λ(S,A,R) is in fact $\lambda\to$.

(ii) Consider the following specification.

S	$*, \square$
A	$* : \square$
R	$(*,*), (\square, *)$

Then λ(S,A,R) is λ2.

Proof idea. (i) By induction on the generation of $\vdash$ it can be shown that if $\Gamma \vdash (\Pi x{:}A.B) : *$, then $x \notin FV(B)$. Therefore each $\Pi x{:}A.B$ is in fact $A \to B$.

(ii) Similarly. □

By making variations in the GTS's introduced in 4.4.4. and 4.4.5. a natural cube of eight GTS's can be defined.

4.4.6. Definition (λ-cube). A set of eight GTS's will be defined. Each systems has as sorts $S = \{*, \square\}$ and as axioms $A = \{*: \square\}$. As rules the systems have $(*,*)$ plus one of the eight subsets of $\{(*, \square), (\square, *), (\square, \square)\}$. The following list gives the correspondence between systems and subsets; also alternative names of a system are given.

System	Rules besides $(*,*)$	Alternative names
$\lambda\!\to$		simply typed lambda calculus
$\lambda 2$	$(\square,*)$	F, F_2
$\lambda\underline{\omega}$	$(\square,\square)$	
$\lambda\omega$	$(\square,*)$ $(\square,\square)$	$F\omega$
λP	$(*,\square)$	LF (logical framework)
$\lambda P2$	$(\square,*)$ $(*,\square)$	
$\lambda P\underline{\omega}$	$(\square,\square)$ $(*,\square)$	
$\lambda P\omega$	$(\square,*)$ $(\square,\square)$ $(*,\square)$	λC; calculus of constructions

In the following picture of the λ-cube the edges → represent inclusion of systems.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu, dash] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda\!\to \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

The λ-cube.

The systems $\lambda\!\to$, $\lambda 2$, $\lambda\underline{\omega}$ and λP can be seen as spanning the λ-cube. E.g. $\lambda\omega$ is the union of $\lambda 2$ and $\lambda\underline{\omega}$. The system $\lambda\omega = F\omega$ was introduced in Girard [1972]. The system $\lambda P\omega$ is the calculus of constructions of Coquand and Huet [1988]. The λ-cube is introduced in Barendregt [1989] and is a finestructure of $\lambda P\omega$. The notion of GTS is a generalisation of the λ-cube due indepently to Berardi and Terlouw (personal communication).

4.4.7. Theorem. All systems of the λ-cube enjoy the following properties.

(i) CR.

(ii) SR.

(iii) SN.

(iv) SN, i.e. $\Gamma \vdash M : A \Rightarrow SN(M)\ \&\ SN(A)$.

(v) UT, i.e. $\Gamma \vdash M: A\ \ \&\ \Gamma \vdash M : A' \Rightarrow A{=}A'$.

(vi) $\Gamma \vdash M : A$? is decidable.

(vii) $\Gamma \vdash M : ?$ is computable.

There are also other interesting GTS's. For example

S	$*$
A	$* : *$
R	$(*,*)$

specifies the system $\lambda*$ in which every type is inhabited and not all legal terms are normalising. As remarked by H. Geuvers the following GTS

S	$*, \Box, \Delta$
A	$* : \Box, \Box : \Delta$
R	$(*,*), (\Box, *)$

specifies λHOL, a GTS equivalent to higher order logic as in Church [1941]; the k in □ are used as sets. See Barendregt [1989], [1990] and Barendregt and Dekkers [199-] for more information on GTS's.

REFERENCES

Bakel, S. van.

[1990] Complete restrictions of the intersection type discipline. To appear in *Theoretical Computer Science.*

Barendregt, H.P.

[1984] *The lambda calculus, its syntax and semantics,* 2-nd revised edition, North Holland Publishing Company, Amsterdam, 1984.

[1989] Introduction to generalised type systems, *Proceedings 3rd Italian Conference on Theoretical Computer Science,* (Eds. A. Bertoni e.a.), World Scientific, Singapore, 1-37.

[1990] Functional programming and Lambda Calculus. To appear in *Handbook of Theoretical Computer Science,* (Ed. J. van Leeuwen), North Holland, Amsterdam.

[199-] Lambda calculi with types. To appear in: *Handbook of Logic in Computer Science,* (Eds. S. Abramsky, D. Gabbai and T. Maibaum), Oxford University Press, Oxford.

Barendregt, H.P., Coppo, M, Dezani-Ciancaglini, M.

[1983] A filter lambda model and the completeness of type assignment, *Journal of Symbolic Logic,* 48, 4, 931-940.

Barendregt, H.P. and M. van Leeuwen

[1985] Functional programming and the language TALE, in: Lecture Notes in Computer Science 224, Springer, Berlin, 122-208.

Barendregt, H.P., Dekkers, W.

[199-] Typed lambda calculi, syntax and semantics, to appear.

Boehm, H.J.

[1989] Type inference in the Presence of Type abstraction, in: *SIGPLAN 89 Conference on Programming Languages Design and Implementation,* Portland, Oregon 1989.

Burstall, R., MacQueen, D., Sanella, D.

[1980] *HOPE: An experimental applicative language,* report CSR-62-80, Edinburgh University.

Cardelli, L., Wegner, P.

[1985] On understanding types, data abstraction, and polymorphism, *Computing Surveys* 17, 4, 471-522.

Cardone, F., Coppo, M.

[1990] Type inference with recursive types: Syntax and semantics. To appear in *Information and Computation.*

Church, A.

[1941] A formalisation of the simple theory of types, *Journal of Symbolic Logic* 5, 56-68.

[1941a] *The calculi of lambda conversion,* Princeton University Press; Reprinted 1963 by University Microfilms Inc., Ann Arbor, Michigan, USA.

Constable, R.L. et al.

[1986] *Implementing mathematics with the nuprl proof development system,* Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Coquand, T. and G. Huet

[1988] The calculus of constructions, *Information and Computation 76*, 95-120.

Coquand, T. et al.

[1989] *The calculus of constructions, documentation and usersguide, version 4.10,* INRIA, Rocquencourt, France.

Curry, H.B.

[1934] Functionality in combinatory logic, *Proc. Nat. Acad. Science USA* 20, 584-590.

Damas, L.

[1982] Principal type schemes for functional programming, *Proceedings of the 9th ACM-POPL*, 207-212.

Girard, J.-Y.

[1972] *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre superieur.* Ph.D. Thesis, Université Paris VII.

Gordon, M.J.C., Milner, R., Wadsworth, C.

[1979] A mechanical logic of computation. Edinburgh LCF, *Lecture Notes in Computer Science 78,* Springer.

Harper, R., MacQueen, D., Milner, R.

[1986] *Standard ML,* Report ECS-LFCS-86-2, Edinburgh University.

Harper, R., F. Honsell and G. Plotkin,

[1987] A framework for defining logics, *Proceedings second Symp. Logic in Computer Science* (Ithaca, N. Y.), IEEE, Washington DC, 194 - 204.

Hindley, J.R.

[1969] The principal type-scheme of an object in combinatory logic, *Trans. Am. Math. Soc.* 146, 29-60.

Kfoury, A.J., Tiuryn, J., Uryczyn, P.

[1988] A proper extension of ML with an effective type assigment, *Proceedings of the 15th ACM, POPL.*

Kfoury, A.J., Tiuryn, J., Uryczyn, P.

[1989] Computational consequences and partial solutions of a generalized unification problem. *Proceedings of the 4th IEEE - LICS,* 98-105.

Lehmann, D.J.

[1977] Modes in ALGOL Y. *Proceedings 5th Annual I.I.I. conference,* may 1977, Guidel, France, Published by INRIA, 1977.

McCracken, N.D.

[1984] The typechecking of programs with implicit type structure, Semantics of data types, (Eds. G. Kahn e.a.), *Lecture Notes in Computer Science 173,* Springer, 301-315.

MacQueen, D., Plotkin, G., Sethi, R.

[1986] An ideal model for recursive polymorphic types, *Information and control* 71, 95-130.

Mendler, N.P.

[1987] Inductive types and type constraints in second-order lambda calculus, *Proceedings of the 2nd symposium of LICS.*

Milner, R.

[1978] A theory of type polymorphism in programming, *Journal of Comp. Syst. Sci.*, 17, 348-375.

Morris, J.H.

[1968] *Lambda calculus models of programming languages,* MAC-TR-57, Project MAC, MIT, Cambridge, Massachussets.

Mycroft, A.

[1984] Polymorphic type schemes for functional programs, *Proceedings of the 9th ACM POPL.*

O'Toole, J.W., Gifford, D.K.

[1989] Type reconstruction with first-class polymorphic values *SIGPLAN 89 Conference on Programming Languages Design and Implementation,* Portland, Oregon 1989.

Pfenning, F.

[1988] Partial polymorphic type inference and higher-order unification, *Proceedings of the ACM LISP and Functional Programming Conference.*

Reynolds, J.C.

[1974] Towards a theory of type structure, in: *Proc. of the Colloque sur la Programmation,* Paris, Lecture Notes in Computer Science 19, Springer, 408 - 425.

[1985] Three approaches to type structure, in: *Mathematical Foundations of Software Development* (Eds. Ehring e.a.), Lecture Notes in Computer Science 185, Springer, Berlin, 97-138.

[1989] Synctactic control of interference, part 2.

Tennent, R.D.

[1989] Elementary data structures in ALGOL-like languages. *Science of Computer Programming* 13 (1989/90), 73-110.

Turner, D.

[1985] Miranda, a non-strict functional language with polymorphic types, in: *Functional programming languages and computer architecture,* Nancy, Lecture Notes in Computer Science 201, Springer, Berlin, 1-16.

Wand, M.

[1987] A simple algorithm and proof for type inference, *Fundamenta Informaticae X,* 115-122.

Wijngaarden, van, A. Mailloux, B, Peck, J.E.L., Koster, C.H.A.

[1969] Report on the algorithmic language ALGOL 68, *Num. Math. 14,* 79-218.