LAMBDA CALCULI WITH TYPES

Henk Barendregt
Catholic University Nijmegen

To appear in

Handbook of Logic in Computer Science, Volume II,
Edited by
S. Abramsky, D.M. Gabbay and T.S.E. Maibaum
Oxford University Press

Comments are welcome. Author's address:

Faculty of Mathematics and Computer Science
Toernooiveld 1
6525 ED Nijmegen
The Netherlands
E-mail: henk@cs.kun.nl

Lambda Calculi with Types

H.P. Barendregt

Contents

Dedication

This work is dedicated to

Nol and *Riet Prager*

the musician/philosopher and the poet.

Acknowledgements

Two persons have been quite influential on the form and contents of this chapter. First of all, Mariangiola Dezani-Ciancaglini clarified to me the essential differences between the Curry and the Church typing systems. She provided a wealth of information on these systems (not all of which has been incorporated in this chapter; see the forthcoming Barendregt and Dekkers (to appear) for more on the subject). Secondly, Bert van Benthem Jutting introduced me to the notion of type dependency as presented in the systems AUTOMATH and related calculi like the calculus of constructions. His intimate knowledge of these calculi—obtained after extensive mathematical texts in them—has been rather useful. In fact it helped me to introduce a fine structure of the calculus of constructions, the so called λ-cube. Contributions of other individuals—often important ones—will be clear form the contents of this chapter.

The following persons gave interesting input or feedback for the contents of this chapter: Steffen van Bakel, Erik Barendsen, Stefano Berardi, Val Breazu-Tannen, Dick (N.G.) de Bruijn, Adriana Compagnoni, Mario Coppo, Thierry Coquand, Wil Dekkers, Ken-etsu Fujita, Herman Geuvers, Jean-Yves Girard, Susumu Hayashi, Leen Helmink, Kees Hemerik, Roger Hindley, Furio Honsell, Martin Hyland, Johan de Iongh, Bart Jacobs, Hidetaka Kondoh, Giuseppe Longo, Sophie Malecki, Gregory Mints, Albert Meyer, Reinhard Muskens, Mark-Jan Nederhof, Rob Nederpelt, Andy Pitts, Randy Pollack, Andre Scedrov, Richard Statman, Marco Swaen, Jan Terlouw, Hans Tonino, Yoshihito Toyama, Anne Troelstra and Roel de Vrijer.

Financial support came from several sources. First of all *Oxford University Press* gave the necessary momentum to write this chapter. The *Research Institute for Declarative Systems* at the *Department of Computer Science* of the *Catholic University Nijmegen* provided the daily environment where I had many discussions with my colleagues and Ph.D. students. The EC Stimulation Project ST2J-0374-C (EDB) *lambda calcul typé* helped me to meet many of the persons mentioned above, notably Berardi and Dezani-Ciancaglini. *Nippon Telephon and Telegraph (NTT)* made it possible to meet Fujita and Toyama. The most essential support came from *Philips Research Laboratories Eindhoven* where I had extensive discussions with van Benthem Jutting leading to the definition of the λ-cube.

Finally I would thank Mariëlle van der Zandt and Jane Spurr for typing and editing the never ending manuscript and Wil Dekkers for proofreading and suggesting improvements. Erik Barendsen was my guru for TEX. Use has been made of the macros of Paul Taylor for commutative diagrams and prooftrees. Erik Barendsen, Wil Dekkers and Herman Geuvers helped me with the production of the final manuscript.

Nijmegen, December 20, 1991 Henk Barendregt

1 Introduction

The lambda calculus was originally conceived by Church (1932;1933) 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, as shown in Kleene and Rosser(1936), the subsystem dealing with functions only became a successful model for the computable functions. This system is called now the *lambda calculus*. Books on this subject e.g. are Church(1941), Curry and Feys (1988), Curry *et al.*(1958; 1972), Barendregt [1984), Hindley and Seldin(1986) and Krivine(1990).

In Kleene and Rosser (1936) it is proved that all recursive functions can be represented in the lambda calculus. On the other hand, in Turing(1937) it is shown that exactly the functions computable by a Turing machine can be represented in the lambda calculus. Representing computable functions as λ-terms, i.e. as expressions in the lambda calculus, gives rise to so-called *functional programming*. See Barendregt(1990) for an introduction and references.

The lambda calculus, as treated in the references cited above, 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 $\mathsf{I} = \lambda x.x$ 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 notation used for this is $M : A$. For example in most systems with types 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 volts to 2 ampères. 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 functional 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 a type that is usually unique (up to an equivalence relation) and that is 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 chapter several typed lambda calculi will be introduced, both *à la* Curry and *à la* Church. Since in the last two decades several dozens of systems have appeared, we will make a selection guided by the following methodology.

> *Only the simplest versions of a system will be considered. That is, only with β-reduction, but not with e.g. η-reduction. The Church systems will have types built up using only $\rightarrow$ and Π, not using e.g. $\times$ or Σ. The Curry systems will have types built up using only $\rightarrow$, $\cap$ and μ.*

(For this reason we will not consider systems of constructive type theory as developed e.g. in Martin-Löf (1984), since in these theories Σ plays an essential role.) It will be seen that there are already many interesting systems in this simple form. Understanding these will be helpful for the understanding of more complicated systems. No semantics of the typed lambda calculi will be given in this chapter. The reason is that, especially for the Church systems, the notion of model is still subject to intensive investigation. Lambek and Scott (1986) and Mitchell (1990), a chapter on typed lambda calculus in another handbook, do treat semantics but only for one of the systems given in the present chapter. For the Church systems several proposals for notions of semantics have been proposed. These have been neatly unified using fibred categories in Jacobs (1991). See also Pavlović (1990). For the semantics of the Curry systems see Hindley (1982), (1983) and Coppo (1985). A later volume of this handbook will contain a chapter on the semantics of typed lambda calculi.

Barendregt and Hemerik (1990) and Barendregt (1991) are introductory versions of this chapter. Books including material on typed lambda calculus are Girard *et al.* (1989) (treats among other things semantics of the Church version of $\lambda 2$), Hindley and Seldin (1986) (Curry and Church versions of $\lambda\!\rightarrow$), Krivine (1990) (Curry versions of $\lambda 2$ and $\lambda\cap$), Lambek and Scott (1986) (categorical semantics of $\lambda\!\rightarrow$) and the forthcoming Barendregt and Dekkers (199-) and Nerode and Odifreddi (199-).

Section 2 of this chapter is an introduction to type-free lambda-calculus and may be skipped if the reader is familiar with this subject. Section 3 explains in more detail the Curry and Church approach to lambda calculi with types. Section 4 is about the Curry systems and Section 5 is about the Church systems. These two sections can be read independently of each other.

2 Type-free lambda calculus

The introduction of the type-free lambda calculus is necessary in order to define the system of Curry type assignment on top of it. Moreover, although the Church style typed lambda calculi can be introduced directly, it is nevertheless useful to have some knowledge of the type-free lambda calculus. Therefore this section is devoted to this theory. For more information see Hindley and Seldin [1986] or Barendregt [1984].

2.1 The system

In this chapter the type-free lambda calculus will be called 'λ-calculus' or simply λ. We start with an informal description.

Application and abstraction

The λ-calculus has two basic operations. The first one is application. The expression

$$F.A$$

(usually written as FA) denotes the data F considered as algorithm applied to A considered as input. The theory λ is *type-free*: it is allowed to consider expressions like FF, that is, F applied to itself. This will be useful to simulate recursion.

The other basic operation is *abstraction*. If $M \equiv M[x]$ is an expression containing ('depending on') x, then $\lambda x.M[x]$ denotes the intuitive map

$$x \mapsto M[x],$$

i.e. to x one assigns $M[x]$. The variable x does not need to occur actually in M. In that case $\lambda x.M[x]$ is a constant function with value M.

Application and abstraction work together in the following intuitive formula:

$$(\lambda x.x^2 + 1)3 = 3^2 + 1 \ \ (= 10).$$

That is, $(\lambda x.x^2 + 1)3$ denotes the function $x \mapsto x^2 + 1$ applied to the argument 3 giving $3^2 + 1$ (which is 10). In general we have

$$(\lambda x.M[x])N = M[N].$$

This last equation is preferably written as

$$(\lambda x.M)N = M[x := N], \tag{β}$$

where $[x := N]$ denotes substitution of N for x. This equation is called β-conversion. It is remarkable that although it is the only essential axiom of the λ-calculus, the resulting theory is rather involved.

Free and bound variables

Abstraction is said to *bind* the *free* variable x in M. For example, we say that $\lambda x.yx$ has x as bound and y as free variable. Substitution $[x := N]$ is only performed in the free occurrences of x:

$$yx(\lambda x.x)[x := N] = yN(\lambda x.x).$$

In integral calculus there is a similar variable binding. In $\int_a^b f(x, y)dx$ the variable x is bound and y is free. It does not make sense to substitute 7 for x, obtaining $\int_b^a f(7, y)d7$; but substitution for y does make sense, obtaining $\int_b^a f(x, 7)dx$.

For reasons of hygiene it will always be assumed that the bound variables that occur in a certain expression are different from the free ones. This can be fulfilled by renaming bound variables. For example, $\lambda x.x$ becomes $\lambda y.y$. Indeed, these expressions act the same way:

$$(\lambda x.x)a = a = (\lambda y.y)a$$

and in fact they denote the same intended algorithm. Therefore expressions that differ only in the names of bound variables are identified. Equations like $\lambda x.x \equiv \lambda y.y$ are usually called α-conversion.

Functions of several arguments

Functions of several arguments can be obtained by iteration of application. The idea is due to Schönfinkel (1924) but is often called 'currying', after H.B. Curry who introduced it independently. Intuitively, if $f(x, y)$ depends on two arguments, one can define

$$\begin{aligned} F_x &= \lambda y.f(x, y) \\ F &= \lambda x.F_x. \end{aligned}$$

Then

$$(Fx)y = F_x y = f(x, y). \tag{1}$$

This last equation shows that it is convenient to use *association to the left* for iterated application:

$$FM_1 \ldots M_n \text{ denotes } (..((FM_1)M_2) \ldots M_n).$$

The equation (1) then becomes

$$Fxy = f(x, y).$$

Dually, iterated abstraction uses *association to the right:*

$$\lambda x_1 \cdots x_n.f(x_1, \ldots, x_n) \text{ denotes } \lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.f(x_1, \ldots, x_n))..)).$$

Then we have for F defined above

$$F = \lambda xy.f(x, y)$$

and (1) becomes

$$(\lambda xy.f(x, y))xy = f(x, y).$$

For n arguments we have

$$(\lambda x_1 \ldots x_n.f(x_1, \ldots, x_n))x_1 \ldots x_n = f(x_1, \ldots, x_n),$$

by using (β) n times. This last equation becomes in convenient vector notation

$$(\lambda \vec{x}.f(\vec{x}))\vec{x} = f(\vec{x});$$

more generally one has

$$(\lambda \vec{x}.f(\vec{x}))\vec{N} = f(\vec{N}).$$

Now we give the formal description of the λ-calculus.

Definition 2.1.1. The set of λ-*terms*, notation Λ, is built up from an infinite set of variables $V = \{v, v', v'', \ldots\}$ using application and (function) abstraction:

$$\begin{array}{lcl} x \in V & \Rightarrow & x \in \Lambda, \\ M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda, \\ M \in \Lambda, x \in V & \Rightarrow & (\lambda x M) \in \Lambda. \end{array}$$

Using abstract syntax one may write the following.

$$\begin{array}{l} V ::= v \mid V' \\ \Lambda ::= V \mid (\Lambda\Lambda) \mid (\lambda V \Lambda) \end{array}$$

Example 2.1.2. The following are λ-terms:

$$\begin{array}{l} v; \\ (vv''); \\ (\lambda v(vv'')); \\ ((\lambda v(vv''))v'); \\ ((\lambda v'((\lambda v(vv''))v'))v'''). \end{array}$$

Convention 2.1.3.

1. $x, y, z, \ldots$ denote arbitrary variables;
   $M, N, L, \ldots$ denote arbitrary λ-terms.
2. As already mentioned informally, the following abbreviations are used:

   $$FM_1 \ldots M_n \text{ stands for } (..((FM_1)M_2) \ldots M_n)$$

   and
   $$\lambda x_1 \cdots x_n.M \text{ stands for } (\lambda x_1(\lambda x_2(\ldots(\lambda x_n(M))..))).$$
3. Outermost parentheses are not written.

Using this convention, the examples in 2.1.2 now may be written as follows:

$$\begin{array}{l} x; xz; \lambda x.xz; \\ (\lambda x.xz)y; \\ (\lambda y.(\lambda x.xz)y)w. \end{array}$$

Note that $\lambda x.yx$ is $(\lambda x(yx))$ and not $((\lambda xy)x)$.

Notation 2.1.4. $M \equiv N$ denotes that M and N are the same term or can be obtained from each other by renaming bound variables. For example,

$$\begin{array}{rcl} (\lambda x.x)z & \equiv & (\lambda x.x)z; \\ (\lambda x.x)z & \equiv & (\lambda y.y)z; \\ (\lambda x.x)z & \not\equiv & (\lambda x.y)z. \end{array}$$

Definition 2.1.5.

1. The set of *free variables* of M, (notation $FV(M)$), is defined inductively as follows:

$$\begin{array}{rcl} FV(x) & = & \{x\}; \\ FV(MN) & = & FV(M) \cup FV(N); \\ FV(\lambda x.M) & = & FV(M) - \{x\}. \end{array}$$

2. M is a *closed λ-term* (or *combinator*) if $FV(M) = \emptyset$. The set of closed λ-terms is denoted by Λ^0.

3. The result of *substitution* of N for (the free occurrences of) x in M, notation $M[x := N]$, is defined as follows: Below $x \not\equiv y$.

$$\begin{array}{rcl} x[x := N] & \equiv & N; \\ y[x := N] & \equiv & y; \\ (PQ)[x := N] & \equiv & (P[x := N])(Q[x := N]); \\ (\lambda y.P)[x := N] & \equiv & \lambda y.(P[x := N]), \text{ provided } y \not\equiv x; \\ (\lambda x.P)[x := N] & \equiv & (\lambda x.P). \end{array}$$

In the λ-term

$$y(\lambda xy.xyz)$$

y and z occur as free variables; x and y occur as bound variables. The term $\lambda xy.xxy$ is closed.

Names of bound variables will be always chosen such that they differ from the free ones in a term. So one writes $y(\lambda xy'.xy'z)$ for $y(\lambda xy.xyz)$. This so-called 'variable convention' makes it possible to use substitution for the λ-calculus without a proviso on free and bound variables.

Proposition 2.1.6 (Substitution lemma). *Let $M, N, L \in \Lambda$. Suppose $x \not\equiv y$ and $x \notin FV(L)$. Then*

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof. By induction on the structure of M. ■ ∎

Now we introduce the λ-calculus as a formal theory of equations between λ-terms.

Definition 2.1.7.

1. The principal axiom scheme of the λ-calculus is

$$(\lambda x.M)N = M[x := N] \tag{β}$$

   for all $M, N \in \Lambda$. This is called *β-conversion.*

2. There are also the 'logical' axioms and rules:

$$\begin{array}{rcl} & & M = M; \\ M = N & \Rightarrow & N = M; \\ M = N, N = L & \Rightarrow & M = L; \\ M = M' & \Rightarrow & MZ = M'Z; \\ M = M' & \Rightarrow & ZM = ZM'; \\ M = M' & \Rightarrow & \lambda x.M = \lambda x.M'. \end{array} \tag{ξ}$$

3. If $M = N$ is provable in the λ-calculus, then we write $\lambda \vdash M = N$ or sometimes just $M = N$.

Remarks 2.1.8.

1. We have identified terms that differ only in the names of bound variables. An alternative is to add to the λ-calculus the following axiom scheme of *α-conversion.*

$$\lambda x.M = \lambda y.M[x := y], \tag{α}$$

   provided that y does not occur in M. The axiom (β) above was originally the second axiom; hence its name. We prefer our version of the theory in which the identifications are made on a syntactic level. These identifications are done in our mind and not on paper.

2. Even if initially terms are written according to the variable convention, α-conversion (or its alternative) is necessary when rewriting terms. Consider e.g. $\omega \equiv \lambda x.xx$ and $1 \equiv \lambda yz.yz$. Then

$$\begin{array}{rcl} \omega 1 & \equiv & (\lambda x.xx)(\lambda yz.yz) \\ & = & (\lambda yz.yz)(\lambda yz.yz) \\ & = & \lambda z.(\lambda yz.yz)z \\ & \equiv & \lambda z.(\lambda yz'.yz')z \end{array}$$

$$\begin{aligned} &= \lambda zz'.zz' \\ &\equiv \lambda yz.yz \\ &\equiv 1. \end{aligned}$$

3. For implementations of the λ-calculus the machine has to deal with this so called α-conversion. A good way of doing this is provided by the 'name-free notation' of N.G. de Bruijn, see Barendregt (1984), Appendix C. In this notation $\lambda x(\lambda y.xy)$ is denoted by $\lambda(\lambda 21)$, the 2 denoting a variable bound 'two lambdas above'.

The following result provides one way to represent recursion in the λ-calculus.

Theorem 2.1.9 (Fixed point theorem).

1. $\forall F \exists X\, FX = X$.
   (This means that for all $F \in \Lambda$ *there is an* $X \in \Lambda$ *such that* $\lambda \vdash FX = X$.*)*
2. *There is a fixed point combinator*

$$\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

$$\forall F\ F(\mathsf{Y}F) = \mathsf{Y}F.$$

Proof. 1. Define $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then $X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$.

2. By the proof of (1). Note that $\mathsf{Y}F = (\lambda x.F(xx))(\lambda x.F(xx)) \equiv X$. ■

■

Corollary 2.1.10. *Given a term* $C \equiv C[f,x]$ *possibly containing the displayed free variables, then*

$$\exists F \forall X\ FX = C[F,X].$$

Here $C[F,X]$ *is of course the substitution result* $C[f := F][x := X]$.

Proof. Indeed, we can construct F by supposing it has the required property and calculating back:

$$\begin{array}{rrcl} & \forall X\ FX & = & C[F,X] \\ \Leftarrow & Fx & = & C[F,x] \\ \Leftarrow & F & = & \lambda x.C[F,x] \\ \Leftarrow & F & = & (\lambda fx.C[f,x])F \\ \Leftarrow & F & \equiv & \mathsf{Y}(\lambda fx.C[f,x]).\ \blacksquare \end{array}$$

■

This also holds for more arguments: $\exists F \forall \vec{x}\ F\vec{x} = C[F,\vec{x}]$.

As an application, terms F and G can be constructed such that for all terms X and Y

$$\begin{aligned} FX &= XF, \\ GXY &= YG(YXG). \end{aligned}$$

2.2 Lambda definability

In the lambda calculus one can define numerals and represent numeric functions on them.

Definition 2.2.1.

1. $F^n(M)$ with $n \in \mathbb{N}$ (the set of natural numbers) and $F, M \in \Lambda$, is defined inductively as follows:

$$\begin{aligned} F^0(M) &\equiv M; \\ F^{n+1}(M) &\equiv F(F^n(M)). \end{aligned}$$

2. The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n \equiv \lambda fx.f^n(x).$$

Proposition 2.2.2 (J. B. Rosser). *Define*

$$\begin{aligned} \mathsf{A}_+ &\equiv \lambda xypq.xp(ypq); \\ \mathsf{A}_* &\equiv \lambda xyz.x(yz); \\ \mathsf{A}_{exp} &\equiv \lambda xy.yx. \end{aligned}$$

Then one has for all $n, m \in \mathbb{N}$

1. $\mathsf{A}_+ \mathbf{c}_n \mathbf{c}_m = \mathbf{c}_{n+m}$.
2. $\mathsf{A}_* \mathbf{c}_n \mathbf{c}_m = \mathbf{c}_{n.m}$.
3. $\mathsf{A}_{exp} \mathbf{c}_n \mathbf{c}_m = \mathbf{c}_{(n^m)}$, *except for $m = 0$ (Rosser starts at 1).*

Proof. We need the following lemma.

Lemma 2.2.3.

1. $(\mathbf{c}_n x)^m(y) = x^{n*m}(y)$;
2. $(\mathbf{c}_n)^m(x) = \mathbf{c}_{(n^m)}(x)$, *for* $m > 0$.

Proof. 1. By induction on m. If $m = 0$, then LHS $= y =$ RHS. Assume (1) is correct for m (Induction Hypothesis: IH). Then

$$\begin{array}{rcl} (\mathbf{c}_n x)^{m+1}(y) & = & \mathbf{c}_n x((\mathbf{c}_n x)^m(y)) \\ & =_{IH} & \mathbf{c}_n x(x^{n*m}(y)) \\ & = & x^n(x^{n*m}(y)) \\ & \equiv & x^{n+n*m}(y) \\ & \equiv & x^{n*(m+1)}(y). \end{array}$$

2. By induction on $m > 0$. If $m = 1$, then LHS $\equiv \mathbf{c}_n x \equiv$ RHS. If (2) is correct for m, then

$$\begin{array}{rcl} \mathbf{c}_n^{m+1}(x) & = & \mathbf{c}_n(\mathbf{c}_n^m(x)) \\ & =_{IH} & \mathbf{c}_n(\mathbf{c}_{(n^m)}(x)) \\ & = & \lambda y.(\mathbf{c}_{(n^m)}(x))^n(y) \\ & =_{(1)} & \lambda y.x^{n^m * n}(y) \\ & = & \mathbf{c}_{(n^{m+1})}x. \blacksquare \end{array}$$

$\blacksquare$

Now the proof of the proposition.

1. By induction on m.
2. Use the lemma (1).
3. By the lemma (2) we have for $m > 0$

$$\mathbf{A}_{exp}\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_m\mathbf{c}_n = \lambda x.\mathbf{c}_n{}^m(x) = \lambda x.\mathbf{c}_{(n^m)}x = \mathbf{c}_{(n^m)},$$

since $\lambda x.Mx = M$ if $M = \lambda y.M'[y]$ and $x \notin FV(M)$. Indeed,

$$\begin{array}{rcl} \lambda x.Mx & = & \lambda x.(\lambda y.M'[y])x \\ & = & \lambda x.M'[x] \\ & \equiv & \lambda y.M'[y] \\ & = & M. \blacksquare \end{array}$$

$\blacksquare$

We have seen that the functions plus, times and exponentiation on $\mathbb{N}$ can be represented in the λ-calculus using Church's numerals. We will show that all computable (recursive) functions can be represented.

Boolean truth values and a conditional can be represented in the λ-calculus.

Definition 2.2.4 (Booleans, conditional).

1. $\mathbf{true} \equiv \lambda xy.x$, $\mathbf{false} \equiv \lambda xy.y$.

2. If B is a Boolean, i.e. a term that is either **true**, or **false**, then

$$\mathbf{if}\ B\ \mathbf{then}\ P\ \mathbf{else}\ Q$$

   can be represented by BPQ. Indeed, $\mathbf{true}PQ = P$ and $\mathbf{false}PQ = Q$.

Definition 2.2.5 (Pairing). For $M, N \in \Lambda$ write

$$[M, N] \equiv \lambda z.zMN.$$

Then

$$[M, N]\,\mathbf{true} = M$$
$$[M, N]\,\mathbf{false} = N$$

and hence $[M, N]$ can serve as an ordered pair.

Definition 2.2.6.

1. A *numeric function* is a map $f : \mathbb{N}^p \to \mathbb{N}$ for some p.

2. A numeric function f with p arguments is called λ-*definable* if one has for some combinator F

$$F\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_p} = \mathbf{c}_{f(n_1,\ldots,n_p)} \tag{1}$$

   for all $n_1, \ldots, n_p \in \mathbb{N}$. If (1) holds, then f is said to be λ-*defined* by F.

Definition 2.2.7.

1. The *initial functions* are the numeric functions U_r^i, S^+, Z defined by:

$$\begin{array}{rcl} U_r^i(x_1, \ldots, x_r) & = & x_i, \quad 1 \le i \le r; \\ S^+(n) & = & n + 1; \\ Z(n) & = & 0. \end{array}$$

2. Let $P(n)$ be a numeric relation. As usual

$$\mu m.P(m)$$

   denotes the least number m such that $P(m)$ holds if there is such a number; otherwise it is undefined.

As we know from Chapter 2 in this handbook, the class $\mathcal{R}$ of recursive functions is the smallest class of numeric functions that contains all initial functions and is closed under composition, primitive recursion and minimalization. So $\mathcal{R}$ is an inductively defined class. The proof that all recursive functions are λ-definable is by a corresponding induction argument. The result is originally due to Kleene (1936).

Lemma 2.2.8. *The initial functions are λ-definable.*

Proof. Take as defining terms

$$\begin{array}{lcl} \mathsf{U}_p^i & \equiv & \lambda x_1 \cdots x_p . x_i; \\ \mathsf{S}^+ & \equiv & \lambda xyz.y(xyz) \quad (= \mathsf{A}_+ \mathbf{c}_1); \\ \mathsf{Z} & \equiv & \lambda x.\mathbf{c}_0. \blacksquare \end{array}$$

∎

Lemma 2.2.9. *The λ-definable functions are closed under composition.*

Proof. Let $g, h_1, \ldots, h_m$ be λ-defined by $G, H_1, \ldots, H_m$ respectively. Then

$$f(\vec{n}) = g(h_1(\vec{n}), \ldots, h_m(\vec{n}))$$

is λ-defined by

$$F \equiv \lambda \vec{x}.G(H_1\vec{x}) \ldots (H_m\vec{x}). \blacksquare$$

∎

Lemma 2.2.10. *The λ-definable functions are closed under primitive recursion.*

Proof. Let f be defined by

$$\begin{array}{l} f(0, \vec{n}) = g(\vec{n}) \\ f(k+1, \vec{n}) = h(f(k, \vec{n}), k, \vec{n}) \end{array}$$

where g, h are λ-defined by G, H respectively. We have to show that f is λ-definable. For notational simplicity we assume that there are no parameters $\vec{n}$ (hence $G = \mathbf{c}_{f(0)}$.) The proof for general $\vec{n}$ is similar.

If k is not an argument of h, then we have the scheme of iteration. Iteration can be represented easily in the λ-calculus, because the Church numerals are iterators. The construction of the representation of f is done in two steps. First primitive recursion is reduced to iteration using ordered pairs; then iteration is represented. Here are the details. Consider

$$T \equiv \lambda p.[\mathsf{S}^+(p\,\mathbf{true}), H(p\,\mathbf{false})(p\,\mathbf{true})].$$

Then for all k one has

$$\begin{array}{rcl} T([\mathbf{c}_k, \mathbf{c}_{f(k)}]) & = & [\mathbf{f}\mathsf{S}^+\mathbf{c}_k, H\mathbf{c}_{f(k)}\mathbf{c}_k] \\ & = & [\mathbf{c}_{k+1}, \mathbf{c}_{f(k+1)}]. \end{array}$$

By induction on k it follows that

$$[\mathbf{c}_k, \mathbf{c}_{f(k)}] = T^k[\mathbf{c}_0, \mathbf{c}_{f(0)}].$$

Therefore

$$\mathbf{c}_{f(k)} = \mathbf{c}_k T[\mathbf{c}_0, \mathbf{c}_{f(0)}]\ \mathbf{false},$$

and f can be λ-defined by

$$F \equiv \lambda k.kT[\mathbf{c}_0, G]\ \mathbf{false}. \blacksquare$$

$\blacksquare$

Lemma 2.2.11. *The λ-definable functions are closed under minimalization.*

Proof. Let f be defined by $f(\vec{n}) = \mu m[g(\vec{n}, m) = 0]$, where $\vec{n} = n_1, \ldots, n_k$ and g is λ-defined by G. We have to show that f is λ-definable. Define

$$\mathbf{zero} \equiv \lambda n.n(\mathbf{true}\ \mathbf{false})\mathbf{true}.$$

Then

$$\begin{array}{rcl} \mathbf{zero}\ \mathbf{c}_0 & = & \mathbf{true}, \\ \mathbf{zero}\ \mathbf{c}_{n+1} & = & \mathbf{false}. \end{array}$$

By Corollary 2.1.10 there is a term H such that

$$H\vec{n}y = \ \mathbf{if}\ (\mathbf{zero}(G\vec{n}y))\ \mathbf{then}\ y\ \mathbf{else}\ H\vec{n}(\mathsf{S}^+y).$$

Set $F = \lambda\vec{n}.H\vec{x}\mathbf{c}0$. Then F λ-defines f:

$$\begin{array}{rcll} F\mathbf{c}_{\vec{x}} & = & H\mathbf{c}_{\vec{n}}\mathbf{c}_0 & \\ & = & \mathbf{c}_0, & \text{if } G\mathbf{c}_{\vec{n}}\mathbf{c}_0 = \mathbf{c}_0, \\ & = & H\mathbf{c}_{\vec{n}}\mathbf{c}_1 & \text{else;} \\ & = & \mathbf{c}_1, & \text{if } G\mathbf{c}_{\vec{n}}\mathbf{c}_1 = \mathbf{c}_0, \\ & = & H\mathbf{c}_{\vec{n}}\mathbf{c}_2 & \text{else;} \\ & = & \mathbf{c}_2, & \text{if } \ldots \\ & = & \ldots & \end{array}$$

Here $\mathbf{c}_{\vec{n}}$ stands for $\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k}$. $\blacksquare$ $\blacksquare$

Theorem 2.2.12. *All recursive functions are λ-definable.*

Proof. By 2.2.8-2.2.11. ■ ■

The converse also holds. The idea is that if a function is λ-definable, then its graph is recursively enumerable because equations derivable in the λ-calculus can be enumerated. It then follows that the function is recursive. So for numeric functions we have f is recursive iff f is λ-definable. Moreover also for partial functions a notion of λ-definability exists and one has ψ is partial recursive iff ψ is λ-definable. The notions λ-definable and recursive both are intended to be formalizations of the intuitive concept of computability. Another formalization was proposed by Turing in the form of Turing computable. The equivalence of the notions recursive, λ-definable and Turing computable (for the latter see besides the original Turing, 1937, e.g., Davis 1958) Davis provides some evidence for the Church–Turing thesis that states that 'recursive' is the proper formalization of the intuitive notion 'computable'.

We end this subsection with some undecidability results. First we need the coding of λ-terms. Remember that the collection of variables is $\{v, v', v'', \ldots\}$.

Definition 2.2.13.

1. Notation. $v^{(0)} = v$; $v^{(n+1)} = v^{(n)\prime}$.

2. Let $\langle\ ,\ \rangle$ be a recursive coding of pairs of natural numbers as a natural number. Define

$$\begin{array}{rcl} \sharp(v^{(n)}) & = & \langle 0, n\rangle; \\ \sharp(MN) & = & \langle 2, \langle \sharp(M), \sharp(N)\rangle\rangle; \\ \sharp(\lambda x.M) & = & \langle 3, \langle \sharp(x), \sharp(M)\rangle\rangle. \end{array}$$

3. Notation

$$\ulcorner M \urcorner = \mathbf{c}_{\sharp M}.$$

Definition 2.2.14. Let $\mathcal{A} \subseteq \Lambda$.

1. *A is closed under* $=$ if

$$M \in \mathcal{A},\ \lambda \vdash M = N \quad \Rightarrow \quad N \in \mathcal{A}.$$

2. *A is non-trivial* if $\mathcal{A} \neq \emptyset$ and $\mathcal{A} \neq \Lambda$.

3. *A is recursive* if $\sharp\mathcal{A} = \{\sharp M \mid M \in \mathcal{A}\}$ is recursive.

The following result due to Scott is quite useful for proving undecidability results.

Theorem 2.2.15. *Let $\mathcal{A} \subseteq \Lambda$ be non-trivial and closed under $=$. Then $\mathcal{A}$ is not recursive.*

Proof. (J. Terlouw) Define

$$\mathcal{B} = \{M \mid M\ulcorner M \urcorner \in \mathcal{A}\}.$$

Suppose $\mathcal{A}$ is recursive; then by the effectiveness of the coding also $\mathcal{B}$ is recursive (indeed, $n \in \sharp\mathcal{B} \Leftrightarrow \langle 2, \langle n, \sharp \mathbf{c}_n \rangle\rangle \in \sharp\mathcal{A}$). It follows that there is an $F \in \Lambda^0$ with

$$\begin{array}{lcl} M \in \mathcal{B} & \Leftrightarrow & F\ulcorner M \urcorner = \mathbf{c}_0; \\ M \notin \mathcal{B} & \Leftrightarrow & F\ulcorner M \urcorner = \mathbf{c}_1. \end{array}$$

Let $M_0 \in \mathcal{A}, M_1 \notin \mathcal{A}$. We can find a $G \in \Lambda$ such that

$$\begin{array}{lcl} M \in \mathcal{B} & \Leftrightarrow & G\ulcorner M \urcorner = M_1 \notin \mathcal{A}, \\ M \notin \mathcal{B} & \Leftrightarrow & G\ulcorner M \urcorner = M_0 \in \mathcal{A}. \end{array}$$

[Take $Gx =$ **if zero**(Fx) **then** M_1 **else** M_0, with **zero** defined in the proof of 2.2.11.] In particular

$$\begin{array}{lclcl} G \in \mathcal{B} & \Leftrightarrow & G\ulcorner G \urcorner \notin \mathcal{A} & \Leftrightarrow_{\mathrm{Def}} & G \notin \mathcal{B}, \\ G \notin \mathcal{B} & \Leftrightarrow & G\ulcorner G \urcorner \in \mathcal{A} & \Leftrightarrow_{\mathrm{Def}} & G \in \mathcal{B}, \end{array}$$

a contradiction. ■ ■

The following application shows that the lambda calculus is not a decidable theory.

Corollary 2.2.16 (Church). *The set*

$$\{M \mid M = \mathbf{true}\}$$

is not recursive.

Proof. Note that the set is closed under $=$ and is nontrivial. ■ ■

2.3 Reduction

There is a certain asymmetry in the basic scheme (β). The statement

$$(\lambda x.x^2 + 1)3 = 10$$

can be interpreted as '10 is the result of computing $(\lambda x.x^2 + 1)3$', but not vice versa. This computational aspect will be expressed by writing

$$(\lambda x.x^2 + 1)3 \twoheadrightarrow 10$$

which reads '$(\lambda x.x^2 + 1)3$ *reduces to* 10'.

Apart from this conceptual aspect, reduction is also useful for an analysis of convertibility. The Church–Rosser theorem says that if two terms are convertible, then there is a term to which they both reduce. In many cases the inconvertibility of two terms can be proved by showing that they do not reduce to a common term.

Definition 2.3.1.

1. A binary relation R on Λ is called *compatible* (w.r.t. operations) if

$$\begin{aligned} M\,R\,N \quad \Rightarrow \quad & (ZM)\,R\,(ZN), \\ & (MZ)\,R\,(NZ), \text{ and} \\ & (\lambda x.M)\,R\,(\lambda x.N). \end{aligned}$$

2. A *congruence* relation on Λ is a compatible equivalence relation.

3. A *reduction* relation on Λ is a compatible, reflexive and transitive relation.

Definition 2.3.2. The binary relations $\rightarrow_\beta, \twoheadrightarrow_\beta$ and $=_\beta$ on Λ are defined inductively as follows:

1. (a) $(\lambda x.M)N \rightarrow_\beta M[x := N]$;
   (b) $M \rightarrow_\beta N \;\Rightarrow\; ZM \rightarrow_\beta ZN,\ MZ \rightarrow_\beta NZ$ and $\lambda x.M \rightarrow_\beta \lambda x.N$.
2. (a) $M \twoheadrightarrow_\beta M$;
   (b) $M \rightarrow_\beta N \quad\Rightarrow\quad M \twoheadrightarrow_\beta N$;
   (c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \quad\Rightarrow\quad M \twoheadrightarrow_\beta L$.
3. (a) $M \twoheadrightarrow_\beta N \quad\Rightarrow\quad M =_\beta N$;
   (b) $M =_\beta N \quad\Rightarrow\quad N =_\beta M$;
   (c) $M =_\beta N, N =_\beta L \quad\Rightarrow\quad M =_\beta L$.

These relations are pronounced as follows:

$$\begin{aligned} M \twoheadrightarrow_\beta N &\ :\ M\ \beta\text{-}\textit{reduces to } N; \\ M \rightarrow_\beta N &\ :\ M\ \beta\text{-}\textit{reduces to } N \textit{ in one step}; \\ M =_\beta N &\ :\ M \text{ is } \beta\text{-}\textit{convertible to } N. \end{aligned}$$

By definition $\rightarrow_\beta$ is compatible. The relation $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$ and therefore a reduction relation. The relation $=_\beta$ is a congruence relation.

Proposition 2.3.3. $M =_\beta N \quad \Leftrightarrow \quad \lambda \vdash M = N$.

Proof. ($\Leftarrow$) By induction on the generation of $\vdash$. ($\Rightarrow$) By induction one shows

$$\begin{array}{lcl} M \rightarrow_\beta N & \Rightarrow & \lambda \vdash M = N; \\ M \twoheadrightarrow_\beta N & \Rightarrow & \lambda \vdash M = N; \\ M =_\beta N & \Rightarrow & \lambda \vdash M = N. \blacksquare \end{array}$$

∎

Definition 2.3.4.

1. A *β-redex* is a term of the form $(\lambda x.M)N$. In this case $M[x := N]$ is its *contractum*.

2. A λ-term M *is* a *β-normal form* (β-nf) if it does not have a β-redex as subexpression.

3. A term M *has* a β-normal form if $M =_\beta N$ and N is a β-nf, for some N.

Example 2.3.5. $(\lambda x.xx)y$ is not a β-nf, but has as β-nf the term yy.

An immediate property of nf's is the following.

Lemma 2.3.6. *Let $M, M', N, L \in \Lambda$.*

1. *Suppose M is a β-nf. Then*

$$M \twoheadrightarrow_\beta N \quad \Rightarrow \quad N \equiv M.$$

2. *If $M \rightarrow_\beta M'$, then $M[x := N] \rightarrow_\beta M'[x := N]$.*

Proof. 1. If M is a β-nf, then M does not contain a redex. Hence never $M \rightarrow_\beta N$. Therefore if $M \twoheadrightarrow_\beta N$, then this must be because $M \equiv N$.

2. By induction on the generation of $\rightarrow_\beta$. ■

$\blacksquare$

Theorem 2.3.7 (Church–Rosser theorem). *If $M \twoheadrightarrow_\beta N_1, M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$; in diagram*

$$\begin{tikzcd}
 & M \arrow[dl, two heads] \arrow[dr, two heads] & \\
N_1 \arrow[dr, two heads, dotted] & & N_2 \arrow[dl, two heads, dotted] \\
 & N_3 &
\end{tikzcd}$$

The proof is postponed until 2.3.17.

Corollary 2.3.8. *If $M =_\beta N$, then there is an L such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$.*

Proof. Induction on the generation of $=_\beta$.

Case 1. $M =_\beta N$ because $M \twoheadrightarrow_\beta N$. Take $L \equiv N$.

Case 2. $M =_\beta N$ because $N =_\beta M$. By the IH there is a common β-reduct L_1 of N, M. Take $L \equiv L_1$.

Case 3. $M =_\beta N$ because $M =_\beta N', N' =_\beta N$. Then

$$\begin{tikzcd}
M \arrow[dr, two heads] & \text{(IH)} & N' \arrow[dl, two heads] \arrow[dr, two heads] & \text{(IH)} & N \arrow[dl, two heads] \\
 & L_1 \arrow[dr, two heads, dotted] & \text{CR} & L_2 \arrow[dl, two heads, dotted] & \\
 & & L & &
\end{tikzcd}$$

$\blacksquare$

$\blacksquare$

Corollary 2.3.9.

1. *If M has N as β-nf, then $M \twoheadrightarrow_\beta N$.*
2. *A λ-term has at most one β-nf.*

Proof. 1. Suppose $M =_\beta N$ with N in β-nf. By corollary 2.3.8 one has $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$ for some L. But then $N \equiv L$, by Lemma 2.3.6, so $M \twoheadrightarrow_\beta N$.

2. Suppose M has β-nf's N_1, N_2. Then $N_1 =_\beta N_2$ $(=_\beta M)$. By Corollary 2.3.8 one has $N_1 \twoheadrightarrow_\beta L, N_2 \twoheadrightarrow_\beta L$ for some L. But then $N_1 \equiv L \equiv N_2$ by Lemma 2.3.6(1). ■

■

Some consequences.

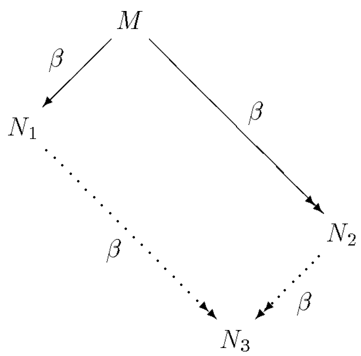
1. The λ-calculus is consistent, i.e. $\lambda \nvdash$ **true** $=$ **false**. Otherwise **true** $=_\beta$ **false** by Proposition 2.3.3, which is impossible by Corollary 2.3.8 since **true** and **false** are distinct β-nf's. This is a syntactical consistency proof.

2. $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ has no β-nf. Otherwise $\Omega \twoheadrightarrow_\beta N$ with N in β-nf. But Ω only reduces to itself and is not in β-nf.

3. In order to find the β-nf of a term, the various subexpressions of it may be reduced in different orders. If a β-nf is found, then by Corollary 2.3.9 (2) it is unique. Moreover, one cannot go wrong: every reduction of a term can be continued to the β-nf of that term (if it exists). See also Theorem 2.3.20.

Proof of the Church–Rosser theorem

This occupies 2.3.10 - 2.3.17. The idea of the proof is as follows. In order to prove the theorem, it is sufficient to show the following strip lemma:



In order to prove this lemma, let $M \rightarrow_\beta N_1$ be a one step reduction resulting from changing a redex R in M in its contractum R' in N_1. If one makes a bookkeeping of what happens with R during the reduction $M \twoheadrightarrow N_2$, then by reducing all 'residuals' of R in N_2 the term N_3 can be found. In order to do the necessary bookkeeping an extended set $\underline{\Lambda} \supseteq \Lambda$

and reduction $\underline{\beta}$ is introduced. The underlining is used in a way similar to 'radioactive tracing isotopes' in experimental biology.

Definition 2.3.10 (Underlining).

1. $\underline{\Lambda}$ is the set of terms defined inductively as follows:

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \underline{\Lambda}; \\ M, N \in \underline{\Lambda} & \Rightarrow & (MN) \in \underline{\Lambda}; \\ M \in \underline{\Lambda}, x \in V & \Rightarrow & (\lambda x.M) \in \underline{\Lambda}; \\ M, N \in \underline{\Lambda}, x \in V & \Rightarrow & ((\underline{\lambda} x.M)N) \in \underline{\Lambda}. \end{array}$$

2. Underlined (one step) reduction ($\rightarrow_{\underline{\beta}}$ and) $\twoheadrightarrow_{\underline{\beta}}$ are defined starting with the contraction rules

$$(\lambda x.M)N \rightarrow M[x := N],$$

$$(\underline{\lambda} x.M)N \rightarrow M[x := N].$$

   Then $\rightarrow$ is extended to the compatible relation $\rightarrow_{\underline{\beta}}$ (also w.r.t. $\underline{\lambda}$-abstraction) and $\twoheadrightarrow_{\underline{\beta}}$ is the transitive reflexive closure of $\rightarrow_{\underline{\beta}}$.

3. If $M \in \underline{\Lambda}$, then $|M| \in \Lambda$ is obtained from M by leaving out all underlinings. For example, $|(\lambda x.x)((\underline{\lambda} x.x)(\lambda x.x))| \equiv \mathsf{I}(\mathsf{II})$.

4. Substitution for $\underline{\Lambda}$ is defined by adding to the schemes in definition 2.1.5(3) the following:

$$((\underline{\lambda} x.M)N)[y := L] \equiv (\underline{\lambda} x.M[y := L])(N[y := L]).$$

Definition 2.3.11. A map $\varphi : \underline{\Lambda} \rightarrow \Lambda$ is defined inductively as follows:

$$\begin{array}{rcl} \varphi(x) & \equiv & x; \\ \varphi(MN) & \equiv & \varphi(M)\varphi(N), \text{ if } M, N \in \underline{\Lambda}; \\ \varphi(\lambda x.M) & \equiv & \lambda x.\varphi(M); \\ \varphi((\underline{\lambda} x.M)N) & \equiv & \varphi(M)[x := \varphi(N)]. \end{array}$$

In other words, the map φ contracts all redexes that are underlined, from the inside to the outside.

Notation 2.3.12. If $|M| \equiv N$ or $\varphi(M) \equiv N$, then this will be denoted by respectively

$$M \xrightarrow[|\,|]{} N \text{ or } M \xrightarrow[\varphi]{} N.$$

Lemma 2.3.13.

$$\begin{tikzcd} M' \arrow[r, dotted, two heads, "\underline{\beta}"] \arrow[d, "|\,|"'] & N' \arrow[d, "|\,|"] \\ M \arrow[r, two heads, "\beta"'] & N \end{tikzcd} \qquad \begin{array}{l} M', N' \in \underline{\Lambda}, \\ M, N \in \Lambda. \end{array}$$

Proof. First suppose $M \to_{\beta} N$. Then N is obtained by contracting a redex in M and N' can be obtained by contracting the corresponding redex in M'. The general statement follows by transitivity. ■ ■

Lemma 2.3.14. *Let $M, M', N, L \in \underline{\Lambda}$. Then*

1. *Suppose $x \not\equiv y$ and $x \notin FV(L)$. Then*

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

2.

$$\varphi(M[x := N]) \equiv \varphi(M)[x := \varphi(N)].$$

3.

$$\begin{tikzcd} M \arrow[r, two heads, "\underline{\beta}"] \arrow[d, "\varphi"'] & N \arrow[d, "\varphi"] \\ \varphi(M) \arrow[r, dotted, two heads, "\beta"'] & \varphi(N) \end{tikzcd} \qquad M, N \in \underline{\Lambda}.$$

Proof. 1. By induction on the structure of M.

2. By induction on the structure of M, using (1) in case $M \equiv (\underline{\lambda} y.P)Q$. The condition of (1) may be assumed to hold by our convention about free variables.

3. By induction on the generation of $\twoheadrightarrow_{\underline{\beta}}$, using (2). ■

■

Lemma 2.3.15.

$$\begin{tikzcd}
& M \arrow[dl, "|\ |"'] \arrow[dr, "\varphi"] & \\
N \arrow[rr, dotted, two heads, "\beta"'] & & L
\end{tikzcd}
\qquad
\begin{array}{l} M \in \underline{\Lambda}, \\ N, L \in \Lambda. \end{array}$$

Proof. By induction on the structure of M. ■

■

Lemma 2.3.16 (Strip lemma).

$$\begin{tikzcd}
& M \arrow[dl, "\beta"'] \arrow[ddr, two heads, "\beta"] & \\
N_1 \arrow[ddr, dotted, two heads, "\beta"'] & & \\
& & N_2 \arrow[dl, dotted, two heads, "\beta"] \\
& N_3 &
\end{tikzcd}
\qquad
M, N_1, N_2, N_3 \in \Lambda.$$

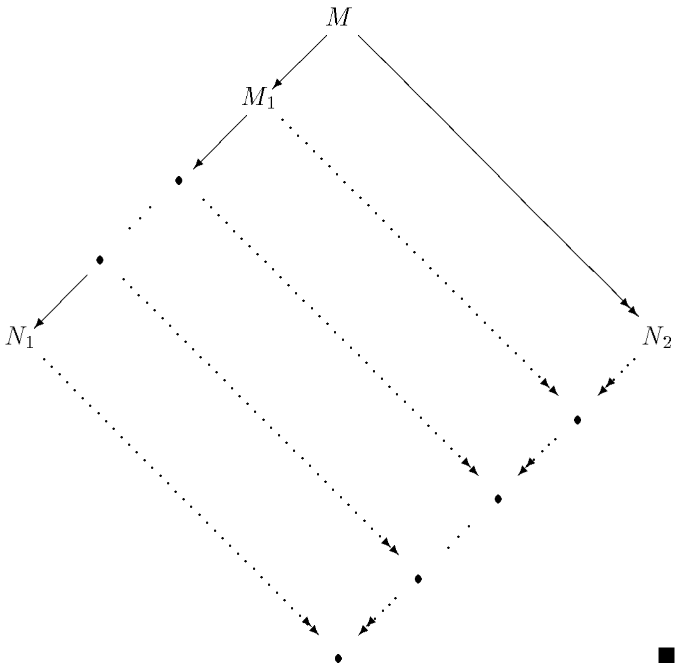
Proof. Let N_1 be the result of contracting the redex occurrence $R \equiv (\lambda x.P)Q$ in M. Let $M' \in \underline{\Lambda}$ be obtained from M by replacing R by $R' \equiv (\underline{\lambda} x.P)Q$. Then $|M'| \equiv M$ and $\varphi(M') \equiv N_1$. By Lemmas 2.3.12, 2.3.13 and 2.3.14 we can construct the following diagram which proves the strip lemma.

$$\begin{tikzcd}
& M \arrow[dl, "\beta"'] \arrow[ddrrr, two heads, "\beta"] & & & & \\
N_1 \arrow[ddrrr, dotted, two heads, "\beta"'] & & M' \arrow[ul, "|\ |"'] \arrow[ll, "\varphi"] \arrow[ddrrr, dotted, two heads, "\beta"] & & & \\
& & & & N_2 \arrow[dl, dotted, two heads, "\beta"'] & \\
& & & N_3 & & N_2' \arrow[ul, "|\ |"'] \arrow[ll, "\varphi"]
\end{tikzcd}$$ ■

$\blacksquare$

Theorem 2.3.17 (Church-Rosser theorem). *If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$.*

Proof. If $M \twoheadrightarrow_\beta N_1$, then $M \equiv M_0 \to_\beta M_1 \to_\beta \ldots M_n \equiv N_1$. Hence the CR property follows from the strip lemma and a simple diagram chase:



$\blacksquare$

$\blacksquare$

Normalization

Definition 2.3.18. For $M \in \Lambda$ the *reduction graph* of M, notation $G_\beta(M)$, is the directed multigraph with vertices $\{N \mid M \twoheadrightarrow_\beta N\}$ and directed by $\to_\beta$. We have a multigraph because contractions of different redexes are considered as different edges.

Example 2.3.19. $G_\beta(\mathsf{I}(\mathsf{I}a))$ is

$$\begin{tikzcd} \mathsf{I}(\mathsf{I}a) \arrow[d, shift left] \arrow[d, shift right] & & \bullet \arrow[d, shift left] \arrow[d, shift right] \\ \mathsf{I}a \arrow[d] & \text{or simply} & \bullet \arrow[d] \\ a & & \bullet \end{tikzcd}$$

A lambda term M is called *strongly normalizing* iff all reduction sequences starting with M terminate (or equivalently iff $G_\beta(M)$ is finite). There are terms that do have an nf, but are not strongly normalizing because they have an infinite reduction graph. Indeed, let $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. Then

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \ldots .$$

Now $\mathsf{KI}\Omega =_\beta \mathsf{I}$, but the left hand side also has an infinite reduction graph. Therefore a so-called *strategy* is necessary in order to find normal forms.

We state the following theorem due to Curry; for a proof see Barendregt (1984), theorem 13.2.2.

Theorem 2.3.20 (Normalization theorem). *If M has a normal form, then iterated contraction of the leftmost redex (i.e. with its main lambda leftmost) leads to that normal form.*

In other words: the leftmost reduction strategy is *normalizing*.

The functional language (pure) LISP uses an *eager* or *applicative* evaluation strategy, i.e. whenever an expression of the form FA has to be evaluated, A is reduced to normal form first, before 'calling' F. In the λ-calculus this strategy is not normalizing as is shown by the two reduction paths for $\mathsf{KI}\Omega$ above. There is, however, a variant of the lambda calculus, called the $\lambda\mathsf{I}$-calculus, in which the eager evaluation strategy is normalizing. See Barendregt [1984], Ch 9, and §11.3. In this $\lambda\mathsf{I}$-calculus terms like K, 'throwing away' Ω in the reduction $\mathsf{KI}\Omega \twoheadrightarrow \mathsf{I}$, do not exist. The 'ordinary' λ-calculus is sometimes referred to as $\lambda\mathsf{K}$-calculus.

In several lambda calculi with types one has that typable terms are strongly normalizing, see subsections 4.3 and 5.3.

Böhm trees and approximation

We end this subsection on reduction by introducing Böhm trees, a kind of 'infinite normal form'.

Lemma 2.3.21. *Each $M \in \Lambda$ is either of the following two forms.*

1. $M \equiv \lambda x_1 \ldots x_n.yN_1 \ldots N_m$, *with* $n, m \geq 0$, *and* y *a variable.*
2. $M \equiv \lambda x_1 \ldots x_n.(\lambda y.N_0)N_1 \ldots N_m$, *with* $n \geq 0, m \geq 1$.

Proof. By definition a λ-term is either a variable, or of the form PQ (an application) or $\lambda x.P$ (an abstraction).

If M is a variable, then M is of the form (1) with $n = m = 0$.

If M is an application, then $M \equiv P_0P_1 \ldots P_m$ with P_0 not an application. Then M is of the form (1) or (2) with $n = 0$, depending on whether P_0 is a variable (giving (1)) or an abstraction (giving (2)).

If M is an abstraction, then a similar argument shows that M is of the right form. ■ ■

Definition 2.3.22.

1. A λ-term M is a *head normal form* (hnf) if M is of the form (1) in Lemma 2.3.21. In that case y is called the *head variable* of M.

2. M has an hnf if $M =_\beta N$ for some N that is an hnf.

3. If M is of the form (2) in 2.3.21, then $(\lambda y.N_0)N_1$ is called the *head redex* of M.

Lemma 2.3.23. *If $M =_\beta M'$ and*

$$M \text{ has hnf } M_1 \equiv \lambda x_1 \cdots x_n.yN_1 \ldots N_m,$$

$$M' \text{ has hnf } M_1' \equiv \lambda x_1 \cdots x_{n'}.y' N_1' \ldots N_{m'}',$$

then $n = n', y \equiv y', m = m'$ *and* $N_1 =_\beta N_1', \ldots, N_m =_\beta N_{m'}'$.

Proof. By the corollary to the Church–Rosser theorem 2.3.8 M_1 and M_1' have a common reduct L. But then the only possibility is that

$$L \equiv \lambda x_1 \cdots x_{n''}.y'' N_1'' \ldots N_{m''}''$$

with

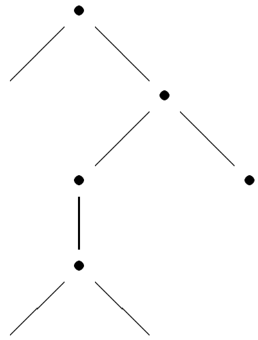
$$n = n' = n', \; y = y'' = y', \; m = m'' = m' \text{ and } N_1 =_\beta N_1'' =_\beta N_1', \ldots . \blacksquare$$

■

The following definitions give the flavour of the notion of Böhm tree. The definitions are not completely correct, because there should be an ordering in the direct successors of a node. However, this ordering is displayed in the drawings of the trees. For a precise definition, covering this order, see Barendregt (1984), Ch.10.

Definition 2.3.24.

1. A *tree* has the form depicted in the following figure.



That is, a tree is a partially ordered set such that

(a) there is a root;

(b) each node (point) has finitely many direct successors;

(c) the set of predecessors of a node is finite and is linearly ordered.

2. A *labeled* tree is a tree with symbols at some of its nodes.

Definition 2.3.25. Let $M \in \Lambda$. The *Böhm tree* of M, notation $BT(M)$, is the labeled tree defined as follows:

$$\begin{array}{rcll} BT(M) & = & \lambda x_1 \cdots x_n.\ \begin{array}{ccc} & y & \\ / & & \backslash \\ BT(N_1) & \ldots & BT(N_m) \end{array}, & \text{if } M \text{ has as hnf } \lambda x_1 \cdots x_n . y N_1 \ldots N_m; \\ & = & \perp, & \text{if } M \text{ has no hnf.} \end{array}$$

Example 2.3.26.

1.

$$BT(\lambda abc.ac(bc)) = \lambda abc.\ \begin{array}{ccc} & a & \\ / & & \backslash \\ c & & b \\ & & | \\ & & c \end{array}.$$

2.

$$BT((\lambda x.xx)(\lambda x.xx)) = \perp.$$

3.
$$BT(\mathsf{Y}) = \lambda f. \begin{array}{c} f \\ | \\ f \\ | \\ \vdots \end{array} .$$

This is because $\mathsf{Y} = \lambda f.\omega_f\omega_f$ with $\omega_f \equiv \lambda x.f(xx)$.
Therefore $\mathsf{Y} = \lambda f.f(\omega_f\omega_f)$ and

$$BT(\mathsf{Y}) = \lambda f. \begin{array}{c} f \\ | \\ BT(\omega_f\omega_f) \end{array} ;$$

now $\omega_f\omega_f = f(\omega_f\omega_f)$ so

$$BT(\omega_f\omega_f) = \begin{array}{c} f \\ | \\ BT(\omega_f\omega_f) \end{array} = \begin{array}{c} f \\ | \\ f \\ | \\ \vdots \end{array} .$$

Remark 2.3.27. Note that Definition 2.3.25 is not an inductive definition of $BT(M)$. The $N_1, \ldots, N_m$ in the tail of an hnf of a term may be more complicated than the term itself. See again Barendregt (1984), Ch.10.

Proposition 2.3.28. *$BT(M)$ is well defined and*

$$M =_\beta N \;\Rightarrow\; BT(M) = BT(N).$$

Proof. What is meant is that $BT(M)$ is independent of the choice of the hnf's. This and the second property follow from Lemma 2.3.23. ■ █

Definition 2.3.29.

1. $\lambda\bot$ is the extension of the lambda calculus defined as follows. One of the variables is selected for use as a constant and is given the name $\bot$. Two contraction rules are added:

$$\lambda x.\bot \to \bot;$$

$$\bot M \to \bot.$$

The resulting reduction relation is called $\beta\bot$- reduction and is denoted by $\twoheadrightarrow_{\beta\bot}$.

2. A $\beta\bot$- *normal form* is such that it cannot be $\beta\bot$-reduced
3. Böhm trees for $\lambda\bot$ are defined by requiring that a $\lambda\bot$-term

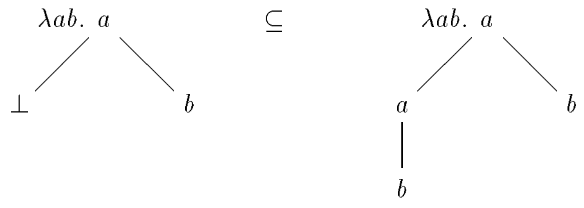
$$\lambda x_1 \cdots x_n.yN_1 \ldots N_m$$

is only in $\beta\bot$-hnf if $y \not\equiv \bot$ or if $n = m = 0$.

Note that if M has a β-nf or β-hnf, then M also has a $\beta\bot$-hnf. This is because an hnf $\lambda x_1 \ldots x_n.yN_1 \ldots N_m$ is also a $\beta\bot$-hnf unless $y = \bot$. But in that case $\lambda x_1 \ldots x_n.yN_1 \ldots N_m \twoheadrightarrow_{\beta\bot} \bot$ and hence M has a β-hnf.

Definition 2.3.30.

1. Let A and B be Böhm trees of some $\lambda\bot$-terms. Then A is *included* in B, notation $A \subseteq B$, if A results from B by cutting off some subtrees, leaving an empty node. For example,



2. Let P, Q be $\lambda\bot$- terms. Then P *approximates* Q, notation $P \subseteq Q$, if $BT(P) \subseteq BT(Q)$.
3. Let P be a $\lambda\bot$-term. The set of *approximate normal forms* (anf's) of P, is defined as

$$\mathcal{A}(P) \quad = \quad \{Q \subseteq P \mid Q \text{ is a } \beta\bot\text{-nf}\}.$$

Example 2.3.31. The set of anf's of the fixedpoint operator Y is

$$\mathcal{A}(\mathsf{Y}) = \{\bot, \lambda f.f\bot, \lambda f.f^2\bot, \ldots\}.$$

Without a proof we mention the following 'continuity theorem', due to Wadsworth (1971).

Proposition 2.3.32. *Let $F, M \in \Lambda$ be given. Then*

$$\forall P \in \mathcal{A}(FM)\, \exists Q \in \mathcal{A}(M) \quad P \in \mathcal{A}(FQ).$$

See Barendregt (1984), proposition 14.3.19, for the proof and a topological explanation of the result.

3 Curry versus Church typing

In this section the system $\lambda{\rightarrow}$ of simply typed lambda calculus will be introduced. Attention is focused on the difference between typing *à la* Curry and *à la* Church by introducing $\lambda{\rightarrow}$ in both ways. Several other systems of typed lambda calculus exist both in a Curry and a Church version. However, this is not so for all systems. For example, for the Curry system $\lambda\cap$ (the system of intersection types, introduced in 4.1) it is not clear how to define a Church version. And for the Church system λC (calculus of constructions) it is not clear how to define a Curry version. For the systems that exist in both styles there is a clear relation between the two versions, as will be explained for $\lambda{\rightarrow}$.

3.1 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), 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 $\vdash M : \sigma$, often 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.

Definition 3.1.1. The set of *types* of $\lambda{\rightarrow}$, notation Type($\lambda{\rightarrow}$), is inductively defined as follows. We write $\mathbb{T}$= Type($\lambda{\rightarrow}$).

$$\begin{array}{ll} \alpha, \alpha', \ldots \in \mathbb{T} & \text{(type variables);} \\ \sigma, \tau \in \mathbb{T} \Rightarrow (\sigma{\rightarrow}\tau) \in \mathbb{T} & \text{(function space types).} \end{array}$$

Such definitions will occur more often and it is convenient to use the following abstract syntax to form $\mathbb{T}$:

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

with $\mathbb{V}$ defined by

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

Notation 3.1.2.

1. If $\sigma_1, \ldots, \sigma_n \in \mathbb{T}$ then

   $$\sigma_1 {\rightarrow} \sigma_2 {\rightarrow} \cdots {\rightarrow} \sigma_n$$

   stands for

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

   that is, we use association to the right.

2. $\alpha, \beta, \gamma, \ldots$ denote arbitrary type variables.

Definition 3.1.3 ($\lambda{\rightarrow}$-Curry).

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

2. A *declaration* is a statement with as subject a (term) variable.

3. A *basis* is a set of declarations with distinct variables as subjects.

Definition 3.1.4. A statement $M : \sigma$ is *derivable from* a basis Γ, notation

$$\Gamma \vdash_{\lambda\rightarrow\text{-Curry}} M : \sigma$$

(or

$$\Gamma \vdash_{\lambda\rightarrow} M : \sigma$$

or

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

if there is no danger for confusion) if $\Gamma \vdash M : \sigma$ can be produced by the following rules.

λ→-Curry (version 0)

$$
\begin{array}{rcl}
(x{:}\sigma) \in \Gamma & \Rightarrow & \Gamma \vdash x : \sigma; \\[6pt]
\Gamma \vdash M : (\sigma {\to} \tau),\ \ \Gamma \vdash N : \sigma & \Rightarrow & \Gamma \vdash (MN) : \tau; \\[6pt]
\Gamma,\ x{:}\sigma \vdash M : \tau & \Rightarrow & \Gamma \vdash (\lambda x.M) : (\sigma {\to} \tau).
\end{array}
$$

Here $\Gamma, x{:}\sigma$ stands for $\Gamma \cup \{x{:}\sigma\}$. If $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ (or $\Gamma = \emptyset$) then instead of $\Gamma \vdash M : \sigma$ one writes $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash M : \sigma$ (or $\vdash M : \sigma$). Pronounce $\vdash$ as 'yields'.

The rules given in Definition 3.1.3 are usually notated as follows:

λ→-Curry (version 1)

$$
\begin{array}{lll}
\text{(axiom)} & \Gamma \vdash x : \sigma, & \text{if } (x{:}\sigma) \in \Gamma; \\[10pt]
(\to\text{-elimination}) & \dfrac{\Gamma \vdash M : (\sigma {\to} \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\,; & \\[14pt]
(\to\text{-introduction}) & \dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma {\to} \tau)}\,. &
\end{array}
$$

Another notation for these rules is the natural deduction formulation.

λ→-Curry (version 2)

Elimination rule	Introduction rule
$\dfrac{M : (\sigma {\to} \tau) \quad N : \sigma}{MN : \tau}$	$\dfrac{\begin{array}{c}\cancel{x : \sigma} \\ \vdots \\ M : \tau\end{array}}{(\lambda x.M) : (\sigma {\to} \tau)}$

In this version the axiom of version 0 or 1 is considered as implicit and is not notated. The notation

$$
\begin{array}{c}
x : \sigma \\
\vdots \\
M : \tau
\end{array}
$$

means that from the assumption $x{:}\sigma$ (together with a set Γ of other statements) one can derive $M : \tau$. The introduction rule in the table states that from this one may infer that $(\lambda x.M) : (\sigma{\to}\tau)$ is derivable even without the assumption $x{:}\sigma$ (but still using Γ). This process is called *cancellation* of an assumption and is indicated by striking through the statement $\cancel{(x{:}\sigma)}$.

Examples 3.1.5.

1. Using version 1 of the system, the *derivation*

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

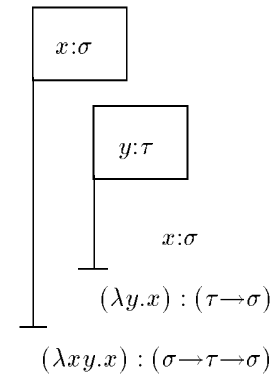
shows that $\vdash (\lambda xy.x) : (\sigma{\to}\tau{\to}\sigma)$ for all $\sigma, \tau \in \mathbb{T}$.

A *natural deduction* derivation (for version 2 of the system) of the same type assignment is

$$\dfrac{\dfrac{\dfrac{\cancel{x{:}\sigma}\ 2 \qquad \cancel{y{:}\tau}\ 1}{x{:}\sigma}}{(\lambda y.x)\ :\ (\tau{\to}\sigma)}\,1}{(\lambda xy.x)\ :\ (\sigma{\to}\tau{\to}\sigma)}\,2$$

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

A more explicit way of dealing with cancellations of statements is the 'flag-notation' used by Fitch (1952) and in the languages AUTOMATH of de Bruijn (1980). In this notation the above derivation becomes as follows.



As one sees, the bookkeeping of cancellations is very explicit; on the other hand it is less obvious how a statement is derived from previous statements.

2. Similarly one can show for all $\sigma \in \mathbb{T}$

$$\vdash (\lambda x.x) : (\sigma \to \sigma).$$

3. An example with a non-empty basis is the following

$$y{:}\sigma \vdash (\lambda x.x)y : \sigma.$$

In the rest of this chapter we usually will introduce systems of typed lambda calculi in the style of version 1 of $\lambda\to$-Curry.

Pragmatics of constants

In applications of typed lambda calculi often one needs constants. For example in programming one may want a type constant **nat** and term constants **0** and **suc** representing the set of natural numbers, zero and the successor function. The way to do this is to take a type variable and two term variables and give these the names **nat**, **0** and **suc**. Then one forms as basis

$$\Gamma_0 = \{\mathbf{0}{:}\mathbf{nat}, \mathbf{suc}{:}(\mathbf{nat}\to\mathbf{nat})\}.$$

This Γ_0 will be treated as a so called 'initial basis'. That is, only bases Γ will be considered that are extensions of Γ_0. Moreover one promises not to bind the variables in Γ_0 by changing e.g.

$$\mathbf{0}{:}\mathbf{nat}, \mathbf{suc}{:}(\mathbf{nat}\to\mathbf{nat}) \vdash M : \sigma$$

into

$$\vdash (\lambda\mathbf{0}\lambda\mathbf{suc}.M) : (\mathbf{nat}\to(\mathbf{nat}\to\mathbf{nat})\to\sigma).$$

(If one does not keep the promise no harm is done, since then **0** and **suc** become ordinary bound variables.)

The programming language ML, see Milner [1984], is essentially $\lambda\to$-Curry extended with a constant Y and type assignment $\mathsf{Y} : ((\sigma\to\sigma)\to\sigma)$ for all σ.

Properties of $\lambda\to$-Curry

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.

Properties of $\lambda{\rightarrow}$-Curry

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

Definition 3.1.6. Let $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ be a basis.

1. Write $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$; $\sigma_i = \Gamma(x_i)$. That is, Γ is considered as a partial function.
2. Let V_0 be a set of variables. Then $\Gamma \restriction V_0 = \{x{:}\sigma \mid x \in V_0 \ \& \ \sigma = \Gamma(x)\}$.
3. For $\sigma, \tau \in \mathbb{T}$ substitution of τ for α in σ is denoted by $\sigma[\alpha := \tau]$.

Proposition 3.1.7 (Basis lemma for $\lambda{\rightarrow}$-Curry).
Let Γ be a basis.

1. *If $\Gamma' \supseteq \Gamma$ is another basis, then* $\Gamma \vdash M : \sigma \Rightarrow \Gamma' \vdash M : \sigma$.
2. $\Gamma \vdash M : \sigma \Rightarrow FV(M) \subseteq$ *dom* Γ.
3. $\Gamma \vdash M : \sigma \Rightarrow \Gamma \restriction FV(M) \vdash M : \sigma$.

Proof. 1. 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 briefer 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 $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau{\rightarrow}\sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma' \vdash M_1 : (\tau{\rightarrow}\sigma)$ and $\Gamma' \vdash M_2 : \tau$. Hence $\Gamma' \vdash (M_1 M_2) : \sigma$.

Case 3. $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1{\rightarrow}\sigma_2)$ 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 dom Γ'. 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 $\Gamma' \vdash (\lambda x.M_1) : (\sigma_1{\rightarrow}\sigma_2)$.

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

3. By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau{\rightarrow}\sigma)$ and

$M_2 : \tau$ for some τ. By the IH one has $\Gamma \restriction FV(M_1) \vdash M_1 : (\tau \to \sigma)$ and $\Gamma \restriction FV(M_2) \vdash M_2 : \tau$. By (1) it follows that $\Gamma \restriction FV(M_1 M_2) \vdash M_1 : (\tau \to \sigma)$and $\Gamma \restriction FV(M_1 M_2) \vdash M_2 : \tau$ and hence $\Gamma \restriction FV(M_1 M_2) \vdash (M_1 M_2) : \sigma$. ■

■

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.

Proposition 3.1.8 (Generation lemma for $\lambda\!\to$-Curry).

1. $\Gamma \vdash x : \sigma \Rightarrow (x{:}\sigma) \in \Gamma$.
2. $\Gamma \vdash MN : \tau \Rightarrow \exists \sigma\ [\Gamma \vdash M : (\sigma \to \tau)\ \&\ \Gamma \vdash N : \sigma]$.
3. $\Gamma \vdash \lambda x.M : \rho \Rightarrow \exists \sigma, \tau\ [\Gamma, x{:}\sigma \vdash M : \tau\ \&\ \rho \equiv (\sigma \to \tau)]$.

Proof. By induction on the length of derivation. ■ ■

Proposition 3.1.9 (Typability of subterms in $\lambda\!\to$-Curry). *Let M' be a subterm of M. Then $\Gamma \vdash M : \sigma \Rightarrow \Gamma' \vdash M' : \sigma'$ for some Γ' and σ'. 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. ■ ■

Proposition 3.1.10 (Substitution lemma for $\lambda\!\to$-Curry).

1. $\Gamma \vdash M : \sigma \Rightarrow \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.
2. *Suppose* $\Gamma, x{:}\sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$. *Then* $\Gamma \vdash M[x := N] : \tau$.

Proof. 1. By induction on the derivation of $M : \sigma$.

2. By induction on the generation of $\Gamma, x{:}\sigma \vdash M : \tau$. ■

■

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

Proposition 3.1.11 (Subject reduction theorem for $\lambda{\to}$-Curry). *Suppose $M \twoheadrightarrow_\beta M'$. Then*

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

Proof. Induction on the generation of $\twoheadrightarrow_\beta$ using Propositions 3.1.8 and 3.1.10. 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 3.1.8 that for some τ one has

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

Hence once more by Proposition 3.1.8 that

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

and therefore by the substitution lemma 3.1.10

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

∎

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

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

See Exercise 3.1.13. One even has the following stronger failure of subject expansion, as is observed in van Bakel (1991).

Observation 3.1.12. There are $M, M' \in \Lambda$ and $\sigma, \sigma' \in \mathbb{T}$ such that $M' \twoheadrightarrow_\beta M$ and

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

but

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

Proof. Take $M \equiv \lambda xy.y$, $M' \equiv \mathsf{SK}$, $\sigma \equiv \alpha{\to}(\beta{\to}\beta)$ and $\sigma' \equiv (\beta{\to}\alpha){\to}(\beta{\to}\beta)$; do Exercise 3.1.13. $\blacksquare$ ∎

Exercises 3.1.13.

- Let $\mathsf{I} = \lambda x.x, \mathsf{K} = \lambda xy.x$ and $\mathsf{S} = \lambda xyz.xz(yx)$.
  - ∗ Show that for all $\sigma, \tau, \rho \in \mathbb{T}$ one has
    $\vdash \mathsf{S} : (\sigma{\to}\tau{\to}\rho){\to}(\sigma{\to}\tau){\to}(\sigma{\to}\rho)$
    $\vdash \mathsf{SK} : (\sigma{\to}\tau){\to}\sigma{\to}\sigma$;
    $\vdash \mathsf{KI} : (\tau{\to}\sigma{\to}\sigma)$
  - ∗ Show that $\nvdash \mathsf{SK} : (\tau{\to}\sigma{\to}\sigma)$.
  - ∗ Show that $\lambda x.xx$ and $\mathsf{KI}(\lambda x.xx)$ have no type in $\lambda{\to}$.

3.2 The system $\lambda\!\rightarrow$-Church

Before we give the formal definition, let us explain right away what is the difference between the Church and Curry versions of the system $\lambda\!\rightarrow$. One has

$$\vdash_{\text{Curry}} (\lambda x.x) : (\sigma\!\rightarrow\!\sigma),$$

but on the other hand

$$\vdash_{\text{Church}} (\lambda x{:}\sigma.x) : (\sigma\!\rightarrow\!\sigma).$$

That is, the term $\lambda x.x$ is annotated in the Church system by '$:\sigma$'. The intuitive meaning is that $\lambda x{:}\sigma.x$ takes the argument x from the type (set) σ. This explicit mention of types in a term makes it possible to decide whether a term has a certain type. For some Curry systems this question is undecidable.

Definition 3.2.1. Let $\mathbb{T}$ be some set of types. The set of $\mathbb{T}$-*annotated* λ-*terms* (also called *pseudoterms*), notation $\Lambda_{\mathbb{T}}$, is defined as follows:

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

Here V denotes the set of term variables.

The same syntactic conventions for $\Lambda_{\mathbb{T}}$ are used as for Λ. For example

$$\lambda x_1{:}\sigma_1 \cdots x_n{:}\sigma_n.M \equiv (\lambda x_1{:}\sigma_1(\lambda x_2{:}\sigma_2 \ldots (\lambda x_n{:}\sigma_n(M)))).$$

This term may also be abbreviated as

$$\lambda\vec{x}{:}\vec{\sigma}.M.$$

Several systems of typed lambda calculi *à la* Church consist of a choice of the set of types $\mathbb{T}$ and of an assignment of types $\sigma \in \mathbb{T}$ to terms $M \in \Lambda_{\mathbb{T}}$. However, as will be seen in Section 5, this is not the case in all systems *à la* Church. In systems with so-called (term) dependent types the sets of terms and types are defined simultaneously. Anyway, for $\lambda\!\rightarrow$-Church the separate definition of the types and terms is possible and one has as choice of types the same set $\mathbb{T} = \text{Type}\,(\lambda\!\rightarrow)$ as for $\lambda\!\rightarrow$-Curry.

Definition 3.2.2. The typed lambda calculus $\lambda\!\rightarrow$-Church is defined as follows:

1. The set of types $\mathbb{T} = \text{Type}\,(\lambda\!\rightarrow)$ is defined by

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

2. A *statement* is of the form $M : \sigma$ with $M \in \Lambda_{\mathbb{T}}$ and $\sigma \in \mathbb{T}$.

3. A *basis* is again a set of statements with only distinct variables as subjects.

Definition 3.2.3. A statement $M : \sigma$ is *derivable* from the basis Γ, notation $\Gamma \vdash M : \sigma$, if $M : \sigma$ can be produced using the following rules.

$\lambda{\rightarrow}$-Church

(axiom)	$\Gamma \vdash x : \sigma,$	if $(x{:}\sigma) \in \Gamma$;
($\rightarrow$-elimination)	$\dfrac{\Gamma \vdash M : (\sigma{\rightarrow}\tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$;	
($\rightarrow$-introduction)	$\dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma.M) : (\sigma{\rightarrow}\tau)}$.	

As before, derivations can be given in several styles. We will not need to be explicit about this.

Definition 3.2.4. The set of *(legal)* $\lambda{\rightarrow}$*-terms*, notation $\Lambda(\lambda{\rightarrow})$, is defined by

$$\Lambda(\lambda{\rightarrow}) = \{M \in \Lambda_{\mathbb{T}} \mid \exists \Gamma, \sigma \ \ \Gamma \vdash M : \sigma\}.$$

In order to refer specifically to $\lambda{\rightarrow}$-Church, one uses the notation

$$\Gamma \vdash_{\lambda\rightarrow\mathrm{Church}} M : \sigma.$$

If there is little danger of ambiguity one uses also $\vdash_{\lambda\rightarrow}, \vdash_{\mathrm{Church}}$ or just $\vdash$.

Examples 3.2.5. In $\lambda{\rightarrow}$-Church one has

1. $\vdash (\lambda x{:}\sigma.x) : (\sigma{\rightarrow}\sigma)$;

2. $\vdash (\lambda x{:}\sigma \lambda y{:}\tau.x) : (\sigma{\rightarrow}\tau{\rightarrow}\sigma)$;

3. $x{:}\sigma \vdash (\lambda y{:}\tau.x) : (\tau{\rightarrow}\sigma)$.

As for the type-free theory one can define reduction and conversion on the set of pseudoterms $\Lambda_{\mathbb{T}}$.

Definition 3.2.6. On $\Lambda_{\mathbb{T}}$ the binary relations *one-step β-reduction*, *many-step β-reduction* and *β-convertibility*, notations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ respectively, are generated by the contraction rule

$$(\lambda x{:}\sigma.M)N \;\rightarrow\; M[x := N] \tag{β}$$

For example one has

$$(\lambda x{:}\sigma.x)(\lambda y{:}\tau.yy) \rightarrow_\beta \lambda y{:}\tau.yy.$$

Without a proof we mention that the Church–Rosser theorem 2.3.7 for $\twoheadrightarrow_\beta$ also holds on $\Lambda_{\mathbb{T}}$. The proof is similar to that for Λ; see Barendregt and Dekkers (to appear) for the details. The following results for $\lambda\!\rightarrow$-Church are essentially the same as Propositions 3.1.7 - 3.1.11 for $\lambda\!\rightarrow$-Curry. Therefore proofs are omitted.

Proposition 3.2.7 (Basis lemma for $\lambda\!\rightarrow$-Church). *Let Γ be a basis.*

1. *If $\Gamma' \supseteq \Gamma$ is another basis, then $\Gamma \vdash M : \sigma \Rightarrow \Gamma' \vdash M : \sigma$.*
2. *$\Gamma \vdash M : \sigma \Rightarrow FV(M) \subseteq$ dom (Γ).*
3. *$\Gamma \vdash M : \sigma \Rightarrow \Gamma \restriction FV(M) \vdash M : \sigma$.*

Proposition 3.2.8 (Generation lemma for $\lambda\!\rightarrow$-Church).

1. *$\Gamma \vdash x : \sigma \;\;\Rightarrow\;\; (x{:}\sigma) \in \Gamma$.*
2. *$\Gamma \vdash MN : \tau \;\;\Rightarrow\;\; \exists\sigma\, [\Gamma \vdash M : (\sigma\rightarrow\tau)$ and $\Gamma \vdash N : \sigma]$*
3. *$\Gamma \vdash (\lambda x{:}\sigma.M) : \rho \Rightarrow \exists\tau\, [\rho = (\sigma\rightarrow\tau)$ and $\Gamma, x{:}\sigma \vdash M : \tau]$.*

Proposition 3.2.9 (Typability of subterms in $\lambda\!\rightarrow$-Church). *If M has a type, then every subterm of M has a type as well.*

Proposition 3.2.10 (Substitution lemma for $\lambda\!\rightarrow$-Church).

1. *$\Gamma \vdash M : \sigma \Rightarrow \Gamma[\alpha := \tau] \vdash M[\alpha := \tau] : \sigma[\alpha := \tau]$.*
2. *Suppose $\Gamma, x{:}\sigma \vdash M : \tau$ and $\Gamma \vdash N : \sigma$. Then $\Gamma \vdash M[x := N] : \tau$.*

Proposition 3.2.11 (Subject reduction theorem for $\lambda\!\rightarrow$-Church). *Let $M \twoheadrightarrow_\beta M'$. Then*

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

This proposition implies that the set of legal expressions is closed under reduction. It is not closed under expansion or conversion. Take for example

$\mathsf{I} =_\beta \mathsf{K I \Omega}$ annotated with the appropriate types; it follows from proposition 3.2.9 that $\mathsf{K I \Omega}$ has no type. On the other hand convertible *legal* terms have the same type with respect to a given basis.

Proposition 3.2.12 (Uniqueness of types lemma for $\lambda{\to}$-Church).

1. *Suppose* $\Gamma \vdash M : \sigma$ *and* $\Gamma \vdash M : \sigma'$. *Then* $\sigma \equiv \sigma'$.
2. *Suppose* $\Gamma \vdash M : \sigma$, $\Gamma \vdash M' : \sigma'$ *and* $M =_\beta M'$. *Then* $\sigma \equiv \sigma'$.

Proof. 1. Induction on the structure of M.

2. By the Church–Rosser theorem for $\Lambda_{\mathbb{T}}$, the subject reduction theorem 3.2.11 and (1). ■

■

As observed in 3.1.12 this proposition does not hold for $\lambda{\to}$-Curry.

Original version of $\lambda{\to}$

Church defined his $\lambda{\to}$ in a slightly different, but essentially equivalent, way. He defined the set of (legal) terms directly and not as a subset of the pseudoterms $\Lambda_{\mathbb{T}}$. Each variable carries its own type. The set of terms of type σ, notation $\Lambda_\sigma(\lambda{\to})$ or simply Λ_σ, is defined inductively as follows. Let V be the set of variables.

$$\begin{array}{lcl} \sigma \in \mathbb{T},\ x \in V & \Rightarrow & x^\sigma \in \Lambda_\sigma; \\ M \in \Lambda_{\sigma\to\tau}, N \in \Lambda_\sigma & \Rightarrow & (MN) \in \Lambda_\tau; \\ M \in \Lambda_\tau & \Rightarrow & (\lambda x^\sigma.M) \in \Lambda_{\sigma\to\tau}. \end{array}$$

Then Church's definition of legal terms was

$$\Lambda(\lambda{\to}) = \cup_{\sigma\in\mathbb{T}} \Lambda_\sigma(\lambda{\to}).$$

The following example shows that our version is equivalent to the original one.

Example 3.2.13. The statement in $\lambda{\to}$-Church

$$x{:}\sigma \vdash (\lambda y{:}\tau.x) : (\tau{\to}\sigma)$$

becomes in the original system of Church

$$(\lambda y^\tau.x^\sigma) \in \Lambda_{\tau\to\sigma}.$$

It turns out that this original notation is not convenient for more complicated typed lambda calculi. The problem arises if types themselves become subject to reduction. Then one would expect that

$$\begin{array}{rcl} \sigma \twoheadrightarrow_\beta \tau & \Rightarrow & x^\sigma \twoheadrightarrow_\beta x^\tau \\ & \Rightarrow & \lambda x^\sigma.x^\sigma \twoheadrightarrow_\beta \lambda x^\sigma.x^\tau. \end{array}$$

However, in the last term it is not clear how to interpret the binding effect of λx^σ (is x^τ bound by it?). Therefore we will use the notation of definition 3.2.1.

Relating the Curry and Church systems

For typed lambda calculi that can be described both *à la* Curry and *à la* Church, there is often a simple relation between the two versions. This will be explained for $\lambda\!\rightarrow$.

Definition 3.2.14. There is a 'forgetful' map $|\cdot| : \Lambda_{\mathbb{T}} \rightarrow \Lambda$ defined as follows:

$$\begin{array}{rcl} |x| & \equiv & x; \\ |MN| & \equiv & |M||N|; \\ |\lambda x{:}\sigma.M| & \equiv & \lambda x.|M|. \end{array}$$

The map $|\cdot|$ just erases all type ornamentations of a term in $\Lambda_{\mathbb{T}}$. The following result states that ornamented legal terms in the Church version 'project' to legal terms in the Curry version of $\lambda\!\rightarrow$; conversely, legal terms in $\lambda\!\rightarrow$-Curry can be 'lifted' to legal terms in $\lambda\!\rightarrow$-Church.

Proposition 3.2.15.

1. *Let* $M \in \Lambda_{\mathbb{T}}$. *Then*

$$\Gamma \vdash_{\text{Church}} M : \sigma \;\Rightarrow\; \Gamma \vdash_{\text{Curry}} |M| : \sigma.$$

2. *Let* $M \in \Lambda$. *Then*

$$\Gamma \vdash_{\text{Curry}} M : \sigma \;\Rightarrow\; \exists M' \in \Lambda_{\mathbb{T}}\, [\Gamma \vdash_{\text{Church}} M' : \sigma \;\&\; |M'| \equiv M].$$

Proof. (1), (2). By induction on the given derivation. ∎ ∎

Corollary 3.2.16. *In particular, for a type* $\sigma \in \mathbb{T}$ *one has*

$$\sigma \textit{ is inhabited in } \lambda\!\rightarrow\!\textit{-Curry} \;\Leftrightarrow\; \sigma \textit{ inhabited in } \lambda\!\rightarrow\!\textit{-Church}.$$

Proof. Immediate. ∎ ∎

4 Typing *à la* Curry

4.1 The systems

In this subsection the main systems for assigning types to type-free lambda terms will be introduced. The systems to be discussed are $\lambda\!\!\rightarrow$, $\lambda 2$, $\lambda\mu$ and $\lambda\cap$. Moreover, there are also two extra derivation rules EQ and $\mathcal{A}$ that can be added to each of these systems. In Figure 1 the systems are represented in a diagram.

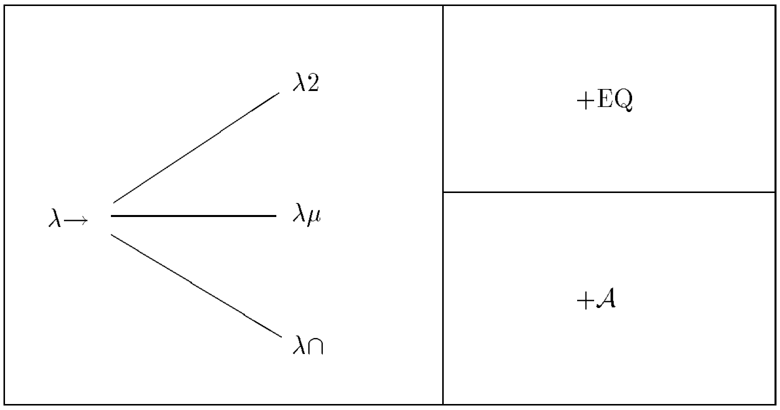


Fig. 1. The systems *à la* Curry

The systems $\lambda 2$, $\lambda\mu$ and $\lambda\cap$ are all extensions of $\lambda\!\!\rightarrow$-Curry. Several stronger systems can be defined by forming combinations like $\lambda 2\mu$ or $\lambda\mu\cap$. However, such systems will not be studied in this chapter.

Now we will first describe the rules EQ and $\mathcal{A}$ and then the systems $\lambda 2$, $\lambda\mu$ and $\lambda\cap$.

Definition 4.1.1.

1. The *equality rule*, notation EQ is the rule

$$\frac{M : \sigma \qquad M =_\beta N}{N : \sigma}$$

2. The *approximation rule*, notation $\mathcal{A}$, consists of the following two rules. These rules are defined for $\lambda\bot$ introduced in Definition 2.3.29. The constant $\bot$ plays a special role in the rule $\mathcal{A}$.

$$\text{Rule } \mathcal{A} \quad \boxed{\begin{array}{c} \dfrac{\Gamma \vdash P : \sigma \text{ for all } P \in \mathcal{A}(M)}{\Gamma \vdash M : \sigma}\ ; \\[2ex] \dfrac{}{\Gamma \vdash \bot : \sigma}\ . \end{array}}$$

See 2.3.30 for the definition of $\mathcal{A}(M)$. Note that in these rules the requirements $M =_\beta N$ and $P \in \mathcal{A}(M)$ are not statements, but are, so to speak, side conditions. The last rule states that $\bot$ has any type.

Notation 4.1.2.

1. $\lambda\text{--}^+$ is $\lambda\text{--}$ extended by rule EQ.
2. $\lambda\text{--}\mathcal{A}$ is $\lambda\text{--}$ extended by rule $\mathcal{A}$.

So for example $\lambda 2^+ = \lambda 2 + \text{EQ}$ and $\lambda\mu\mathcal{A} = \lambda\mu + \mathcal{A}$.

Examples 4.1.3.

1. One has
$$\vdash_{\lambda\to^+} (\lambda pq.(\lambda r.p)(qp)) : (\sigma\to\tau\to\sigma)$$
since $\lambda pq.(\lambda r.p)(qp) = \lambda pq.p$. Note, however, that this statement is in general not provable in $\lambda\!\to$ itself. The term has in $\lambda\!\to$ only types of the form $\sigma\to(\sigma\to\tau)\to\sigma$, as follows form the generation lemma.
2. Let Y be the fixed point operator $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Then
$$\vdash_{\lambda\to\mathcal{A}} \mathsf{Y} : ((\sigma\to\sigma)\to\sigma),$$
Indeed, the approximants of Y are
$$\{\bot, \lambda f.f\bot, \ldots, \lambda f.f^n\bot, \ldots\}$$
and these all have type $((\sigma\to\sigma)\to\sigma)$. Again, this statement is not derivable in $\lambda\!\to$ itself. (In $\lambda\!\to$ all typable terms have a normal form as will be proved in Section 4.2)

Now it will be shown that the rule EQ follows from the rule $\mathcal{A}$. So in general one has $\lambda\text{--}\mathcal{A}^+ = \lambda\text{--}\mathcal{A}$.

Proposition 4.1.4. *In all systems of type assignment λ–$\mathcal{A}$ one has the following.*

1. $\Gamma \vdash M : \sigma$ *and* $P \in \mathcal{A}(M) \;\Rightarrow\; \Gamma \vdash P : \sigma$.
2. *Let* $BT(M) = BT(M')$. *Then*
$$\Gamma \vdash M : \sigma \;\;\Rightarrow\;\; \Gamma \vdash M' : \sigma$$
3. *Let* $M =_\beta M'$. *Then*
$$\Gamma \vdash M : \sigma \;\;\Rightarrow\;\; \Gamma \vdash M' : \sigma.$$

Proof. 1. If P is an approximation of M, then P results from $BT(M)$ by replacing some subtrees by $\perp$ and writing the result as a λ-term. Now $\perp$ may assume arbitrary types, by one of the rules $\mathcal{A}$. Therefore P has the same type as M. [Example. Let $M \equiv \mathsf{Y}$, the fixedpoint combinator and let $P \equiv \lambda f.f(f\perp)$ be an approximant. We have $\vdash \mathsf{Y} : (\sigma\to\sigma)\to\sigma$. By choosing σ as type for $\perp$, one obtains $\vdash P : (\sigma\to\sigma)\to\sigma$.]

2. Suppose $BT(M) = BT(M')$; then $\mathcal{A}(M) = \mathcal{A}(M')$. Hence
$$\begin{aligned}\Gamma \vdash M : \sigma \;\;&\Rightarrow\;\; \forall P \in \mathcal{A}(M) = \mathcal{A}(M')\ \Gamma \vdash P : \sigma, \text{ by (1)},\\ &\Rightarrow\;\; \Gamma \vdash M' : \sigma, \text{ by rule } \mathcal{A}\end{aligned}$$

3. If $M =_\beta M'$, then $BT(M) = BT(M')$, by proposition 2.3.28. Hence the result follows from (2). ■

■

The system $\lambda 2$

The system $\lambda 2$ was introduced independently in Girard (1972) and Reynolds (1974). In these papers the system was introduced in the Church paradigm. Girard's motivation to introduce $\lambda 2$ was based on proof theory. He extended the dialectica translation of Gödel, see Troelstra (1973), to analysis, thereby relating provability in second-order arithmetic to expressibility in $\lambda 2$. Reynolds' motivation to introduce $\lambda 2$ came from programming. He wanted to capture the notion of explicit polymophism.

Other names for $\lambda 2$ are

- polymorphic typed lambda calculus
- second-order typed lambda calculus
- second-order polymorphic typed lambda calculus
- system F.

Usually these names refer to $\lambda 2$-Church. In this section we will introduce the Curry version of $\lambda 2$, leaving the Church version to Section 5.1.

The idea of polymorphism is that in $\lambda{\rightarrow}$

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

for arbitrary α. So one stipulates in $\lambda 2$

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

to indicate that $\lambda x.x$ has all types $\sigma{\rightarrow}\sigma$.

As will be seen later, the mechanism is rather powerful.

Definition 4.1.5. The set of types of $\lambda 2$, notation $\mathbb{T} = \mathrm{Type}(\lambda 2)$, is defined by the following abstract grammar:

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T}{\rightarrow}\mathbb{T} \mid \forall\mathbb{V}\mathbb{T}$$

Notation 4.1.6.

1. $\forall\alpha_1 \cdots \alpha_n.\sigma$ stands for $(\forall\alpha_1(\forall\alpha_2 \ldots (\forall\alpha_n(\sigma))\ldots))$.
2. $\forall$ binds more strongly than $\rightarrow$.

So $\forall\alpha\sigma{\rightarrow}\tau \equiv (\forall\alpha\sigma){\rightarrow}\tau$; but $\forall\alpha.\sigma{\rightarrow}\tau \equiv \forall\alpha(\sigma{\rightarrow}\tau)$.

Definition 4.1.7. Type assignment in $\lambda 2$-Curry is defined by the following natural deduction system:

$\lambda 2$

(start rule)	$\dfrac{(x{:}\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$;
($\rightarrow$-elimination)	$\dfrac{\Gamma \vdash M : (\sigma{\rightarrow}\tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$;
($\rightarrow$-introduction)	$\dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma{\rightarrow}\tau)}$;
($\forall$-elimination)	$\dfrac{\Gamma \vdash M : (\forall\alpha.\sigma)}{\Gamma \vdash M : (\sigma[\alpha := \tau])}$;
($\forall$-introduction)	$\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : (\forall\alpha.\sigma)}$, $\alpha \notin FV(\Gamma)$.

Examples 4.1.8. In $\lambda 2$-Curry one has the following.

1. $\vdash (\lambda x.x) \quad : \quad (\forall \alpha.\alpha \to \alpha)$;
2. $\vdash (\lambda xy.y) \quad : \quad (\forall \alpha\beta.\alpha \to \beta \to \beta)$;
3. $\vdash (\lambda fx.f^n x) \quad : \quad (\forall \alpha.(\alpha \to \alpha) \to \alpha \to \alpha)$;
4. $\vdash (\lambda x.xx) \quad : \quad (\forall \beta.\forall \alpha\alpha \to \beta)$;
5. $\vdash (\lambda x.xx) \quad : \quad (\forall \beta.\forall \alpha\alpha \to (\beta \to \beta))$;
6. $\vdash (\lambda x.xx) \quad : \quad (\forall \alpha\alpha) \to (\forall \alpha\alpha)$.

Example (3) shows that the Church numerals $\mathbf{c}_n \equiv \lambda fx.f^n x$ have type $\forall \alpha.(\alpha \to \alpha) \to \alpha \to \alpha$. This type is sometimes called 'polynat'. One reason for the strength of $\lambda 2$ is that the Church numerals may not only be used as iterators for functions of a fixed type $\alpha \to \alpha$, but also for iteration on $\sigma \to \sigma$ for arbitrary σ. This makes it possible to represent in $\lambda 2$ the term R for primitive recursion of Gödel's T and many other computable functions, see subsection 5.4.

In subsection 4.3 it will be shown that only strongly normalizing terms have a type in $\lambda 2$.

The system $\lambda\mu$

The system $\lambda\mu$ is that of *recursive types.* These come together with an equivalence relation $\approx$ on them. The type assignment rules are such that if $M : \sigma$ and $\sigma \approx \sigma'$, then $M : \sigma'$. A typical example of a recursive type is a σ_0 such that

$$\sigma_0 \approx \sigma_0 \to \sigma_0. \tag{1}$$

This σ_0 can be used to type arbitrary elements $M \in \Lambda$. For example

$$\begin{array}{l} x{:}\sigma_0 \vdash x : \sigma_0 \to \sigma_0 \\ x{:}\sigma_0 \vdash xx : \sigma_0 \\ \vdash \lambda x.xx : \sigma_0 \to \sigma_0 \\ \vdash \lambda x.xx : \sigma_0 \\ \vdash (\lambda x.xx)(\lambda x.xx) : \sigma_0 \end{array}$$

A proof in natural deduction notation of the last statement is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cancel{x : \sigma_0}^{1}}{x : \sigma_0 \to \sigma_0 \qquad\qquad x : \sigma_0}}{(xx) : \sigma_0}}{(\lambda x.xx) : \sigma_0 \to \sigma_0}\,1}{(\lambda x.xx) : \sigma_0 \to \sigma_0 \qquad (\lambda x.xx) : \sigma_0}}{(\lambda x.xx)(\lambda x.xx) : \sigma_0}$$

In fact, equation (1) is like a recursive domain equation $D \cong [D \to D]$ that enables us to interpret elements of Λ. In order to construct a type σ_0

satisfying (1), there is an operator μ such that putting $\sigma_0 \equiv \mu\alpha.\alpha{\to}\alpha$ implies (1).

Definition 4.1.9.

1. The set of types of $\lambda\mu$, notation $\mathbb{T} = \mathrm{Type}(\lambda\mu)$, is defined by the following abstract grammar.

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

2. Let $\sigma \in T$. The *tree of* σ, notation $T(\sigma)$, is defined as follows:

$$T(\alpha) = \alpha, \qquad \text{if } \alpha \text{ a is type variable;}$$

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

$$\begin{array}{lcll} T(\mu\alpha.\sigma) & = & \bot, & \text{if } \sigma \equiv \mu\beta_1 \ldots \mu\beta_n.\alpha \\ & & & \text{for some } n \geq 0; \\ & = & T(\sigma[\alpha := \mu\alpha.\sigma]), & \text{else.} \end{array}$$

3. For $\sigma, \tau \in \mathbb{T}$ one defines

$$\sigma{\approx}\tau \quad \Leftrightarrow \quad T(\sigma) = T(\tau).$$

Examples 4.1.10.

1. If $\tau \equiv \mu\alpha.\alpha{\to}\gamma$, then

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

2. If $\sigma \equiv (\mu\alpha.\alpha{\to}\gamma){\to}\mu\delta\mu\beta.\beta$, then

$$T(\tau) = \begin{tikzcd}[column sep=small] & & & \to \arrow[dl, no head] \arrow[dr, no head] & \\ & & \to \arrow[dl, no head] \arrow[dr, no head] & & \bot \\ & \to \arrow[dl, no head] \arrow[dr, no head] & & \gamma & \\ \dots & & \gamma & & \end{tikzcd} .$$

3. $(\mu\alpha.\alpha\to\gamma) \approx (\mu\alpha(\alpha\to\gamma)\to\gamma)$.
4. $\mu\alpha.\sigma \approx \sigma[\alpha := \mu\alpha.\sigma]$ for all σ, even if $\sigma \equiv \mu\vec{\beta}.\alpha$.

Definition 4.1.11. The type assignment system $\lambda\mu$ is defined by the natural deduction system shown in the following figure.

$\lambda\mu$

(start rule)	$\dfrac{(x{:}\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\,;$
($\to$-elimination)	$\dfrac{\Gamma \vdash M : (\sigma\to\tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\,;$
($\to$-introduction)	$\dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma\to\tau)}\,;$
($\approx$-rule)	$\dfrac{\Gamma \vdash M : \sigma \quad \sigma \approx \tau}{\Gamma \vdash M : \tau}\,.$

The following result is taken from Coppo(1985).

Proposition 4.1.12.
Let σ be an arbitrary type of $\lambda\mu$. Then one can derive in $\lambda\mu$

1. $\vdash \mathsf{Y} : (\sigma\to\sigma)\to\sigma$;

2. $\vdash \Omega : \sigma$.

Proof. 1. Let $\tau \equiv \mu\alpha.\alpha\to\sigma$. Then $\tau \approx \tau\to\sigma$.
Then the following is a derivation for

$$\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\sigma{\to}\sigma){\to}\sigma.$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cancel{f:\sigma\to\sigma}^{2} \qquad \dfrac{\dfrac{\cancel{x:\tau}^{1}}{x:\tau\to\sigma \qquad x:\tau}}{xx:\sigma}}{f(xx):\sigma}}{\lambda x.f(xx):\tau\to\sigma}\,1}{\lambda x.f(xx):\tau\to\sigma \quad \lambda x.f(xx):\tau}}{(\lambda x.f(xx))(\lambda x.f(xx)):\sigma}}{\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\sigma\to\sigma)\to\sigma}\,2
$$

2. Note that $\mathsf{YI} \twoheadrightarrow_\beta \Omega$ and prove and use the subject reduction theorem for $\lambda\mu$; or show $\vdash \Omega : \sigma$ directly. ■

■

The System $\lambda\cap$

The system $\lambda\cap$ of *intersection types* is sometimes called the *Torino system*, since the initial work on this system was done in that city, for example by Coppo, Dezani and Venneri [1981], Barendregt, Coppo and Dezani [1983], Coppo, Dezani, Honsell and Longo [1984], Dezani and Margaria [1987] and Coppo, Dezani and Zacchi [1987]. See also Hindley [1982].

The system makes it possible to state that a variable x has two types σ and τ at the same time. This kind of polymorphism is to be contrasted to that which is present in $\lambda 2$. In that system the polymorphism is parametrized. For example the type assignment

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

states that $\lambda x.x$ has type $\alpha{\to}\alpha$ *uniformly* in α. The assignment $x : \sigma \cap \tau$ states only that x has both type σ and type τ.

Definition 4.1.13.

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

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

2. One of the type variables will be selected as a constant and is notated as ω.

In order to define the rules of type assignment, it is necessary to introduce a preorder on $\mathbb{T}$.

Definition 4.1.14.

1. The relation $\leq$ is defined on $\mathbb{T}$ by the following axioms and rules:
$\sigma \leq \sigma$;
$\sigma \leq \tau, \tau \leq \rho \;\;\Rightarrow\;\; \sigma \leq \rho$
$\sigma \leq \omega$;
$\omega \leq \omega \rightarrow \omega$;
$(\sigma \rightarrow \rho) \cap (\sigma \rightarrow \tau) \leq (\sigma \rightarrow (\rho \cap \tau))$;
$\sigma \cap \tau \leq \sigma, \sigma \cap \tau \leq \tau$;
$\sigma \leq \tau, \sigma \leq \rho \;\;\Rightarrow\;\; \sigma \leq \tau \cap \rho$;
$\sigma \leq \sigma', \tau \leq \tau' \;\;\Rightarrow\;\; \sigma' \rightarrow \tau \leq \sigma \rightarrow \tau'$.

2. $\sigma \sim \tau \;\;\Leftrightarrow\;\; \sigma \leq \tau \;\&\; \tau \leq \sigma$.

For example one has

$$\omega \sim (\omega \rightarrow \omega);$$

$$((\sigma \rightarrow \tau) \cap (\sigma' \rightarrow \tau)) \leq ((\sigma \cap \sigma') \rightarrow \tau).$$

Definition 4.1.15. The system of type assignment $\lambda\cap$ is defined by the following axioms and rules:

$\lambda\cap$

(start rule)	$\dfrac{(x{:}\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$;
($\rightarrow$-elimination)	$\dfrac{\Gamma \vdash M : (\sigma \rightarrow \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$;
($\rightarrow$-introduction)	$\dfrac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \rightarrow \tau)}$;
($\cap$-elimination)	$\dfrac{\Gamma \vdash M : (\sigma \cap \tau)}{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}$;
($\cap$-introduction)	$\dfrac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : (\sigma \cap \tau)}$;
(ω-introduction)	$\dfrac{}{\Gamma \vdash M : \omega}$;
($\leq$-rule)	$\dfrac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau}$.

Examples 4.1.16. In $\lambda\cap$ one has

1. $\vdash \lambda x.xx : ((\sigma\to\tau)\cap\sigma)\to\tau$
2. $\vdash \Omega : \omega$
3. $\vdash (\lambda pq.(\lambda r.p)(qp)) : (\sigma\to(\tau\to\sigma))$.

Proof. 1. The following derivation proves the statement:

$$\dfrac{\dfrac{\dfrac{\cancel{x:(\sigma\to\tau)\cap\sigma}^{1}}{x:\sigma\to\tau \quad x:\sigma}}{(xx):\tau}}{(\lambda x.xx):((\sigma\to\tau)\cap\sigma)\to\tau}\,1$$

2. Obvious. In fact it can be shown that M has no head normal form iff only ω is a type for M, see Barendregt, *et al.* (1983).

3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cancel{q{:}\tau}^{2} \quad \cancel{p{:}\sigma}^{3} \quad \cancel{r{:}\omega}^{1}}{(\lambda r.p):(\omega\to\sigma)}\,1 \qquad \dfrac{}{(qp):\omega}}{(\lambda r.p)(qp):\sigma}}{(\lambda q.(\lambda r.p)(qp)):(\tau\to\sigma)}\,2}{(\lambda pq.(\lambda r.p)(qp)):(\sigma\to(\tau\to\sigma))}\,3 \quad \blacksquare$$

$\blacksquare$

In van Bakel (1990) it is observed that assignment (3) in Example 4.1.16 is not possible in $\lambda\!\to$.

Also for $\lambda\cap$ there are some variants for the system. For example one can delete the rule (axiom) that assigns ω to any term. In van Bakel (1990) several of these variants are studied; see theorem 4.3.12.

Combining the systems à la Curry

The system $\lambda 2, \lambda\mu$ and $\lambda\cap$ are all extensions of $\lambda\!\to$. An extension $\lambda 2\mu\cap$ including all these systems and moreover cartesian products and direct sums is studied in MacQueen *et al.* (1984).

Basic Properties

The Curry systems $\lambda{\rightarrow}$, $\lambda 2$, $\lambda\mu$ and $\lambda\cap$ enjoy several properties. The most immediate ones, valid for all four systems, will be presented now. In subsection 4.2 it will be shown that subject reduction holds for all systems. Some other properties like strong normalization are valid for only some of these systems and will be presented in subsections 4.2, 4.3 and 4.4.

In the following $\vdash$ refers to one of the Curry systems $\lambda{\rightarrow}$, $\lambda 2$, $\lambda\mu$ and $\lambda\cap$. The following three properties are proved in the same way as is done in section 3.1 for $\lambda{\rightarrow}$.

Proposition 4.1.17 (Basis lemma for the Curry systems). *Let Γ be a basis.*

1. *If $\Gamma' \supseteq \Gamma$ is another basis, then $\Gamma \vdash M : \sigma \;\;\Rightarrow\;\; \Gamma' \vdash M : \sigma$.*
2. $\Gamma \vdash M : \sigma \;\;\Rightarrow\;\; FV(M) \subseteq \mathrm{dom}(\Gamma)$.
3. $\Gamma \vdash M : \sigma \;\;\Rightarrow\;\; \Gamma \restriction FV(M) \vdash M : \sigma$.

Proposition 4.1.18 (Subterm lemma for the Curry systems). *Let M' be a subterm of M. Then*

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

The moral is: If M has a type, then every subterm has a type as well.

Proposition 4.1.19 (Substitution lemma for the Curry systems).

1. $\Gamma \vdash M : \sigma \Rightarrow \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.
2. *Suppose $\Gamma, x{:}\sigma \vdash M : \tau$ and $\Gamma \vdash N : \sigma$. Then*

$$\Gamma \vdash M[x := N] : \tau.$$

Exercise 4.1.20. Show that for each of the systems $\lambda{\rightarrow}$, $\lambda 2$, $\lambda\mu$ and $\lambda\cap$ one has $\not\vdash \mathsf{K} : (\alpha{\rightarrow}\alpha)$ in that system.

4.2 Subject reduction and conversion

In this subsection it will be shown that for the main systems of type assignment *à la* Curry, viz. $\lambda{\rightarrow}$, $\lambda 2$, $\lambda\mu$ and $\lambda\cap$ with or without the extra rules $\mathcal{A}$ and EQ, the subject reduction theorem holds. That is,

$$\Gamma \vdash M : \sigma \text{ and } M \twoheadrightarrow_\beta M' \quad\Rightarrow\quad \Gamma \vdash M' : \sigma.$$

Subject conversion or closure under the rule EQ is stronger and states that

$$\Gamma \vdash M : \sigma \text{ and } M =_\beta M' \quad \Rightarrow \quad \Gamma \vdash M' : \sigma.$$

This property holds only for the systems including $\lambda\cap$ or rule $\mathcal{A}$ (or trivially if rule EQ is included).

Subject reduction

We start with proving the subject reduction theorem for all the systems. For $\lambda{\rightarrow}$ this was already done in 3.1.11. In order to prove the result for $\lambda 2$ some definitions and lemmas are needed. This is because for example Proposition 3.1.8 is not valid for $\lambda 2$. So for the time being we focus on $\lambda 2$ and $\mathbb{T} = \mathrm{Type}(\lambda 2)$.

Definition 4.2.1.

1. Write $\sigma > \tau$ if either

$$\tau \equiv \forall\alpha.\sigma, \text{ for some } \alpha,$$

   or

$$\sigma \equiv \forall\alpha.\sigma_1 \text{ and } \tau \equiv \sigma_1[\alpha := \pi] \text{ for some } \pi \in \mathbb{T}.$$

2. $\geq$ is the reflexive and transitive closure of $>$.

3. A map $o : \mathbb{T}{\rightarrow}\mathbb{T}$ is defined by

$$\begin{array}{lcll} \alpha^o & = & \alpha, & \text{if } \alpha \text{ is a type variable;} \\ (\sigma{\rightarrow}\tau)^o & = & \sigma{\rightarrow}\tau; & \\ (\forall\alpha.\sigma)^o & = & \sigma^o. & \end{array}$$

Note that there are exactly two deduction rules for $\lambda 2$ in which the subject does not change: the $\forall$ introduction and elimination rules. Several of these rules may be applied consecutively, obtaining

$$\cfrac{M : \sigma}{\cfrac{\vdots}{\cfrac{\vdots}{M : \tau}}}$$

The definition of $\geq$ is such that in this case $\sigma \geq \tau$. Also one has the following.

Lemma 4.2.2. *Let $\sigma \geq \tau$ and suppose no free type variable in σ occurs in Γ. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \vdash M : \tau$$

Proof. Suppose $\Gamma \vdash M : \sigma$ and $\sigma \geq \tau$. Then $\sigma \equiv \sigma_1 > \cdots > \sigma_n \equiv \tau$ for some $\sigma_1, \ldots, \sigma_n$. By possibly renaming some variables it may be assumed that for $1 \leq i < n$ one has

$$\sigma_{i+1} \equiv \forall\alpha.\sigma_i \;\Rightarrow\; \alpha \notin FV(\Gamma)$$

By definition of the relation $>$ and the rules of $\lambda 2$ it follows that for all $i < n$ one has $\Gamma \vdash M : \sigma_i \;\Rightarrow\; \Gamma \vdash M : \sigma_{i+1}$. Therefore $\Gamma \vdash M : \sigma_n \equiv \tau$. ■
■

Lemma 4.2.3 (Generation lemma for $\lambda 2$-Curry).

1. $\Gamma \vdash x : \sigma \;\Rightarrow\; \exists\sigma' \geq \sigma\ (x{:}\sigma') \in \Gamma.$
2. $\Gamma \vdash (MN) : \tau \;\Rightarrow\; \exists\sigma\exists\tau' \geq \tau\ [\Gamma \vdash M : \sigma\to\tau'$ *and* $\Gamma \vdash N : \sigma]$.
3. $\Gamma \vdash (\lambda x.M) : \rho \;\Rightarrow\; \exists\sigma, \tau\ [\Gamma, x{:}\sigma \vdash M : \tau$ *and* $\sigma\to\tau \geq \rho]$.

Proof. By induction on derivations. ■ ■

Lemma 4.2.4.

1. *Given σ, τ there exists a τ' such that $(\sigma[\alpha := \tau])^o \equiv \sigma^o[\alpha := \tau']$.*
2. $\sigma_1 \geq \sigma_2 \;\Rightarrow\; \exists\vec{\alpha}\ \exists\vec{\tau}\ \ \sigma_2^o \equiv \sigma_1^o[\vec{\alpha} := \vec{\tau}]$.
3. $(\sigma\to\rho) \geq (\sigma'\to\rho') \;\Rightarrow\; \exists\vec{\alpha}\ \exists\vec{\tau}\ \ \sigma'\to\rho' \equiv (\sigma\to\rho)[\vec{\alpha} := \vec{\tau}]$.

Proof. 1. Induction on the structure of σ.

2. It suffices to show this for $\sigma_1 > \sigma_2$.
   Case 1. $\sigma_2 \equiv \forall\alpha.\sigma_1$. Then $\sigma_2^o \equiv \sigma_1^o$.
   Case 2. $\sigma_1 \equiv \forall\alpha.\rho$ and $\sigma_2 \equiv \rho[\alpha := \tau]$.
   Then by (1) one has $\sigma_2^o \equiv \rho^o[\alpha := \tau'] \equiv \sigma_1^o[\alpha := \tau']$.
3. By (2) we have

$$(\sigma'\to\rho') \equiv (\sigma'\to\rho')^o \equiv (\sigma\to\rho)^o[\vec{\alpha} := \vec{\tau}] \equiv (\sigma\to\rho)[\vec{\alpha} := \vec{\tau}]. \blacksquare$$

■

Theorem 4.2.5 (Subject reduction theorem for $\lambda 2$-Curry). *Let $M \twoheadrightarrow_\beta M'$. Then for $\lambda 2$-Curry one has $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \vdash M' : \sigma$.*

Proof. Induction on the derivation of $M \twoheadrightarrow_\beta M'$. We will treat only the case that $M \equiv (\lambda x.P)Q$ and $M' \equiv P[x := Q]$. Now

$$\begin{array}{ll} & \Gamma \vdash ((\lambda x.P)Q) : \sigma \\ \Rightarrow & \exists\rho\exists\sigma' \geq \sigma\,[\Gamma \vdash (\lambda x.P) : (\rho{\to}\sigma')\&\ \Gamma \vdash Q : \rho] \\ \Rightarrow & \exists\rho'\exists\sigma'' \geq \sigma\,[\Gamma, x{:}\rho' \vdash P : \sigma''\&\ \rho'{\to}\sigma'' \geq \rho{\to}\sigma'\&\ \Gamma \vdash Q : \rho] \end{array}$$

by Lemma 4.2.4 (3) it follows that

$$(\rho{\to}\sigma') \equiv (\rho'{\to}\sigma'')[\vec{\alpha} := \vec{\tau}]$$

and hence by Lemma 4.1.19 (1)

$$\begin{array}{ll} \Rightarrow & \Gamma, x{:}\rho \vdash P : \sigma',\ \Gamma \vdash Q : \rho \text{ and } \sigma' \geq \sigma, \\ \Rightarrow & \Gamma \vdash P[x := Q] : \sigma' \text{ and } \sigma' \geq \sigma, \text{ by Lemma 4.1.19 (2)} \\ \Rightarrow & \Gamma \vdash P[x := Q] : \sigma, \text{ by Lemma 4.2.2.} \blacksquare \end{array}$$

∎

In Mitchell (1988) a semantic proof of the subject reduction theorem for $\lambda 2$ is given.

The proof of the subject reduction theorem for $\lambda\mu$ is somewhat easier than for $\lambda 2$.

Theorem 4.2.6 (Subject reduction theorem for $\lambda\mu$).
Let $M \twoheadrightarrow_\beta M'$. Then for $\lambda\mu$ one has

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

Proof. As for $\lambda 2$, but using the relation $\approx$ instead of $\geq$. ∎ ∎

The subject reduction theorem holds also for $\lambda\cap$. This system is even closed under the rule EQ as we will see soon.

Subject conversion

For the systems $\lambda\cap$ and λ–$\mathcal{A}$ we will see that the subject conversion theorem holds. It is interesting to understand the reason why $\lambda\cap$ is closed under β-expansion. This is not so for $\lambda{\to}$, $\lambda 2$ and $\lambda\mu$. Let $M \equiv (\lambda x.P)Q$ and $M' \equiv P[x := Q]$. Suppose $\Gamma \vdash_{\lambda\cap} M' : \sigma$ in order to show that $\Gamma \vdash_{\lambda\cap} M : \sigma$. Now Q occurs $n \geq 0$ times in M' , each occurrence having type τ_i, say, for $1 \leq i \leq n$. Define $\tau \equiv \tau_1 \cap \cdots \cap \tau_n$ if $n > 0$ and $\tau \equiv \omega$ if $n = 0$. Then $\Gamma \vdash Q : \tau$ and $\Gamma, x : \tau \vdash P : \sigma$. Hence $\Gamma \vdash (\lambda x.P) : (\tau{\to}\sigma)$ and $\Gamma \vdash (\lambda x.P)Q : \sigma$.

In $\lambda\!\rightarrow$, $\lambda 2$ and $\lambda\mu$ it may not be possible to find a common type for the different occurrences of Q. Note also that the type ω is essential in case $x \notin FV(P)$.

Theorem 4.2.7 (Subject conversion theorem for $\lambda\cap$). *Let $M =_\beta M'$. Then for $\lambda\cap$ one has*

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

Proof. See Barendregt *et al.* (1983), corollary 3.8. ■ ■

Exercise 4.2.8. Let $M \equiv \lambda pq.(\lambda r.p)(qp)$.

- Show that although $M =_\beta \lambda pq.p : (\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha)$ in $\lambda\!\rightarrow$, the term M does not have $\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$ as type in $\lambda\!\rightarrow$, $\lambda 2$ or $\lambda\mu$.
- Give a derivation in $\lambda\cap$ of $\vdash M : (\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha)$.

4.3 Strong normalization

Remember that a lambda term M is called *strongly normalizing* iff all reduction sequences starting with M terminate. For example KIK is strongly normalizing, while $\mathsf{KI\Omega}$ not. In this subsection it will be examined in which systems of type assignment *à la* Curry one has that the terms that do have a type are strongly normalizing. This will be the case for $\lambda\!\rightarrow$ and $\lambda 2$ but of course not for $\lambda\mu$ and $\lambda\cap$ (since in the latter systems all terms are typable). However, there is a variant $\lambda\cap^-$ of $\lambda\cap$ such that one even has

$$M \text{ is strongly normalizing} \quad \Leftrightarrow \quad M \text{ is typable in } \lambda\cap^-.$$

Turing proved that all terms typable in $\lambda\!\rightarrow$ are normalizing; this proof was only first published in Gandy (1980). As was discussed in Section 2, normalization of terms does not imply in general strong normalization. However, for $\lambda\!\rightarrow$ and several other systems one does have strong normalization of typable terms. Methods of proving strong normalization from (weak) normalization due to Nederpelt (1973) and Gandy (1980) are described in Klop (1980).

Also in Tait (1967) it is proved that all terms typable in $\lambda\!\rightarrow$ are normalizing. This proof uses the so called method of 'computable terms' and was already presented in the unpublished 'Stanford Report' by Howard et al. [1963]. In fact, using Tait's method one can also prove strong normalization and applies to other systems as well, in particular to Gödel's T; see Troelstra [1973].

Girard (1972) gave an 'impredicative twist' to Tait's method in order to show normalization for terms typable in (the Church version of) $\lambda 2$ and in the system $\lambda\omega$ to be discussed in Section 5. Girard's proof was reformulated in Tait (1975) and we follow the general flavour of that paper.

We start with the proof of SN for $\lambda\rightarrow$.

Definition 4.3.1.

1. $\mathrm{SN} = \{M \in \Lambda \mid M \text{ is strongly normalizing}\}$.
2. Let $A, B \subseteq \Lambda$. Define $A \rightarrow B$ a subset of Λ by
$$A \rightarrow B = \{F \in \Lambda \mid \forall a \in A \; Fa \in B\}.$$
3. For every $\sigma \in \mathrm{Type}(\lambda\rightarrow)$ a set $[\![\sigma]\!] \subseteq \Lambda$ is defined as follows:
$$\begin{array}{rcl} [\![\alpha]\!] & = & \mathrm{SN}, \text{ where } \alpha \text{ is a type variable;} \\ [\![\sigma\rightarrow\tau]\!] & = & [\![\sigma]\!]\rightarrow[\![\tau]\!]. \end{array}$$

Definition 4.3.2.

1. A subset $X \subseteq \mathrm{SN}$ is called *saturated* if
   (a) $\forall n \geq 0 \, \forall R_1, \ldots, R_n \in \mathrm{SN} \;\; x\vec{R} \in X$,
   where x is any term variable;
   (b) $\forall n \geq 0 \, \forall R_1, \ldots, R_n \in \mathrm{SN} \forall Q \in \mathrm{SN}$
$$P[x := Q]\vec{R} \in X \quad \Rightarrow \quad (\lambda x.P)Q\vec{R} \in X.$$
2. $\mathrm{SAT} = \{X \subseteq \Lambda \mid X \text{ is saturated}\}$.

Lemma 4.3.3.

1. $\mathrm{SN} \in \mathrm{SAT}$.

2. $A, B \in \mathrm{SAT} \;\; \Rightarrow \;\; A \rightarrow B \in \mathrm{SAT}$.

3. Let $\{A_i\}_{i \in I}$ *be a collection of members of* SAT, *then* $\bigcap_{i \in I} A_i \in SAT$.

4. For all $\sigma \in$ *Type*$(\lambda\rightarrow)$ *one has* $[\![\sigma]\!] \in \mathrm{SAT}$.

Proof. 1. One has $\mathrm{SN} \subseteq \mathrm{SN}$ and satisfies condition (a) in the definition of saturation. As to condition (b), suppose
$$P[x := Q]\vec{R} \in \mathrm{SN} \text{ and } Q, \vec{R} \in \mathrm{SN} \tag{1}$$
We claim that also
$$(\lambda x.P)Q\vec{R} \in \mathrm{SN} \tag{2}$$
Indeed, reductions inside P, Q or the $\vec{R}$ must terminate since these terms are SN by assumption ($P[x := Q]$ is a subterm of a term in

SN, by (1), hence itself SN; but then P is SN); so after finitely many steps reducing the term in (2) we obtain $(\lambda x.P')Q'\vec{R}'$ with $P \twoheadrightarrow_\beta P'$ etcetera. Then the contraction of $(\lambda x.P')Q'\vec{R}'$ gives

$$P'[x := Q']\vec{R}'. \tag{3}$$

This is a reduct of $P[x := Q]\vec{R}$ and since this term is SN also (3) and the term $(\lambda x.P)Q$ are SN.

2. Suppose $A, B \in \mathrm{SAT}$. Then by definition $x \in A$ for all variables x. Therefore

$$\begin{array}{rcl} F \in A \rightarrow B & \Rightarrow & Fx \in B \\ & \Rightarrow & Fx \in \mathrm{SN} \\ & \Rightarrow & F \in \mathrm{SN}. \end{array}$$

So indeed $A \rightarrow B \subseteq \mathrm{SN}$. As to condition 1 of saturation, let $\vec{R} \in \mathrm{SN}$. We must show for a variable x that $x\vec{R} \in A \rightarrow B$. This means

$$\forall Q \in A \;\; x\vec{R}Q \in B,$$

which is true since $A \subseteq \mathrm{SN}$ and B is saturated.

3. Similarly.

4. By induction on the generation of σ, using (1) and (2). ■

■

Definition 4.3.4.

1. A *valuation* in Λ is a map $\rho{:}V \rightarrow \Lambda$, where V is the set of term variables.

2. Let ρ be a valuation in Λ. Then

$$[\![M]\!]_\rho = M[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)],$$

where $\vec{x} = x_1, \ldots, x_n$ is the set of free variables in M.

3. Let ρ be a valuation in Λ. Then ρ *satisfies* $M : \sigma$, notation $\rho \vDash M : \sigma$, if $[\![M]\!]_\rho \in [\![\sigma]\!]$.

If Γ is a basis, then ρ *satisfies* Γ, notation $\rho \models \Gamma$, if $\rho \models x : \sigma$ for all $(x{:}\sigma) \in \Gamma$.

4. A basis Γ *satisfies* $M : \sigma$, notation $\Gamma \models M : \sigma$, if

$$\forall \rho\, [\rho \models \Gamma \Rightarrow \rho \models M : \sigma].$$

Proposition 4.3.5 (Soundness).

$$\Gamma \vdash_{\lambda\to} M : \sigma \;\Rightarrow\; \Gamma \models M : \sigma.$$

Proof. By induction on the derivation of $M : \sigma$.

Case 1. $\Gamma \vdash M : \sigma$ with $M \equiv x$ follows from $(x{:}\sigma) \in \Gamma$.
Then trivially $\Gamma \models x : \sigma$.

Case 2. $\Gamma \vdash M : \sigma$ with $M \equiv M_1 M_2$ is a direct consequence of $\Gamma \vdash M_1 : \tau\to\sigma$ and $\Gamma \vdash M_2 : \tau$.
Suppose $\rho \models \Gamma$ in order to show $\rho \models M_1 M_2 : \sigma$. Then $\rho \models M_1 : \tau\to\sigma$ and $\rho \models M_2 : \tau$, i.e. $[\![M_1]\!]_\rho \in [\![\tau\to\sigma]\!] = [\![\tau]\!]\to[\![\sigma]\!]$ and $[\![M_2]\!]_\rho \in [\![\tau]\!]$. But then $[\![M_1 M_2]\!]_\rho = [\![M_1]\!]_\rho [\![M_2]\!]_\rho \in [\![\sigma]\!]$, i.e. $\rho \models M_1 M_2 : \sigma$.

Case 3. $\Gamma \vdash M : \sigma$ with $M \equiv \lambda x.M'$ and $\sigma \equiv \sigma_1\to\sigma_2$ is a direct consequence of $\Gamma, x{:}\sigma_1 \vdash M' : \sigma_2$.
By the IH one has

$$\Gamma, x : \sigma_1 \models M' : \sigma_2 \tag{1}$$

Suppose $\rho \models \Gamma$ in order to show $\rho \models \lambda x.M' : \sigma_1\to\sigma_2$. That is, we must show

$$[\![\lambda x.M']\!]_\rho N \in [\![\sigma_2]\!] \text{ for all } N \in [\![\sigma_1]\!].$$

So suppose that $N \in [\![\sigma_1]\!]$. Then $\rho(x := N) \models \Gamma, x : \sigma_1$, and hence

$$[\![M']\!]_{\rho(x:=N)} \in [\![\sigma_2]\!],$$

by (1). Since

$$\begin{array}{rcl} [\![\lambda x.M']\!]_\rho N & \equiv & (\lambda x.M')[\vec{y} := \rho(\vec{y})]N \\ & \to_\beta & M'[\vec{y} := \rho(\vec{y}), x := N] \\ & \equiv & [\![M']\!]_{\rho(x:=N)}, \end{array}$$

it follows from the saturation of $[\![\sigma_2]\!]$ that $[\![\lambda x.M']\!]_\rho N \in [\![\sigma_2]\!]$. ∎

∎

Theorem 4.3.6 (Strong normalization for $\lambda\to$-Curry). *Suppose $\Gamma \vdash_{\lambda\to} M : \sigma$. Then M is strongly normalizing.*

Proof. Suppose $\Gamma \vdash M : \sigma$. Then $\Gamma \models M : \sigma$. Define $\rho_o(x) = x$ for all x. Then $\rho_o \models \Gamma$ (since $x \in [\![\tau]\!]$ holds because $[\![\tau]\!]$ is saturated). Therefore $\rho_o \models M : \sigma$, hence $M \equiv [\![M]\!]_{\rho_o} \in [\![\sigma]\!] \subseteq \text{SN}$. ■ ■

The proof of SN for $\lambda\!\rightarrow$ has been given in such a way that a simple generalization of the method proves the result for $\lambda 2$. This generalization will be given now.

Definition 4.3.7.

1. A *valuation* in SAT is a map

   $\xi : \mathbb{V} \rightarrow \text{SAT}$

   where $\mathbb{V}$ is the set of type variables.

2. Given a valuation ξ in SAT one defines for every $\sigma \in \text{Type}(\lambda 2)$ a set $[\![\sigma]\!]_\xi \subseteq \Lambda$ as follows:

$$\begin{aligned} [\![\alpha]\!]_\xi &= \xi(\alpha), \text{ where } \alpha \in \mathbb{V}; \\ [\![\sigma\!\rightarrow\!\tau]\!]_\xi &= [\![\sigma]\!]_\xi \rightarrow [\![\tau]\!]_\xi; \\ [\![\forall\alpha.\sigma]\!]_\xi &= \textstyle\bigcap_{X \in \text{SAT}} [\![\sigma]\!]_{\xi(\alpha:=X)} \end{aligned}$$

Lemma 4.3.8. *Given a valuation* ξ *in* SAT *and a* σ *in Type*$(\lambda 2)$, *then* $[\![\sigma]\!]_\xi \in \text{SAT}$.

Proof. As for Lemma 4.3.3(4) using also that SAT is closed under arbitrary intersections. ■ ■

Definition 4.3.9.

1. Let ρ be a valuation in Λ and ξ be a valuation in SAT. Then

$$\rho, \xi \models M : \sigma \quad \Leftrightarrow \quad [\![M]\!]_\rho \in [\![\sigma]\!]_\xi.$$

2. For such ρ, ξ one writes

$$\rho, \xi \models \Gamma \quad \Leftrightarrow \quad \rho, \xi \models x : \sigma \text{ for all } x{:}\sigma \text{ in } \Gamma.$$

3. $\Gamma \models M : \sigma \;\Leftrightarrow\; \forall \rho, \xi\; [\rho, \xi \models \Gamma \;\;\Rightarrow\;\; \rho, \xi \models M : \sigma]$.

Proposition 4.3.10.

$$\Gamma \vdash_{\lambda 2} M : \sigma \Rightarrow \Gamma \models M : \sigma.$$

Proof. As for Proposition 4.3.5 by induction on the derivation of $\Gamma \vdash M : \sigma$. There are two new cases corresponding to the $\forall$-rules.

Case 4. $\Gamma \vdash M : \sigma$ with $\sigma \equiv \sigma_0[\alpha := \tau]$ is a direct consequence of $\Gamma \vdash M : \forall\alpha.\sigma_0$. By the IH one has

$$\Gamma \vDash M : \forall\alpha.\sigma_0. \tag{1}$$

Now suppose $\rho, \xi \vDash \Gamma$ in order to show that $\rho, \xi \vDash M : \sigma_0[\alpha := \tau]$. By (1) one has

$$[\![M]\!]_\rho \in [\![\forall\alpha.\sigma_0]\!]_\xi = \bigcap_{X \in \mathrm{SAT}} [\![\sigma_0]\!]_{(\alpha:=X)}.$$

Hence

$$[\![M]\!]_\rho \in [\![\sigma_0]\!]_{\xi(\alpha:=[\![\tau]\!]_\xi)}.$$

We are done since

$$[\![\sigma_0]\!]_{\xi(\alpha:=[\![\tau]\!]_\xi)} = [\![\sigma_0[\alpha := \tau]]\!]_\rho$$

as can be proved by induction on $\sigma_0 \in$ Type($\lambda 2$) (some care is needed in case $\sigma_0 \equiv \forall\beta.\tau_0$).

Case 5. $\Gamma \vdash M : \sigma$ with $\sigma \equiv \forall\alpha.\sigma_0$ and $\alpha \notin FV(\Gamma)$ is a direct consequence of $\Gamma \vdash M : \sigma_0$. By the IH one has

$$\Gamma \vDash M : \sigma_0. \tag{2}$$

Suppose $\rho, \xi \vDash \Gamma$ in order to show $\rho, \xi \vDash \forall\alpha.\sigma_0$. Since $\alpha \notin FV(\Gamma)$ one has for all $X \in$ SAT that also $\rho, \xi(\alpha := X) \vDash \Gamma$. Therefore

$$[\![M]\!]_\rho \in [\![\sigma_0]\!]_{\xi(\alpha:=X)} \text{ for all } X \in \mathrm{SAT},$$

by (2), hence

$$[\![M]\!]_\rho \in [\![\forall\alpha.\sigma_0]\!]_\xi,$$

i.e. $\rho, \xi \vDash M : \forall\alpha.\sigma_0$. ∎

∎

Theorem 4.3.11 (Strong normalization for $\lambda 2$-Curry).

$$\Gamma \vdash_{\lambda 2} M : \sigma \Rightarrow M \textit{ is strongly normalizing.}$$

Proof. Similar to the proof of Theorem 4.3.6 ∎ ∎

Although the proof of SN for $\lambda 2$ follows the same pattern as for $\lambda\!\rightarrow$, there is an essential difference. The proof of SN($\lambda\!\rightarrow$) can be formalized in

Peano arithmetic. However, as was shown in Girard (1972), the proof of SN($\lambda 2$) cannot even be formalized in the rather strong system A_2 of 'mathematical analysis' (second order arithmetic); see also Girard *et al.* (1989). The reason is that SN($\lambda 2$) implies (within Peano arithmetic) the consistency of A_2 and hence Gödel's second incompleteness theorem applies. An attempt to formalize the given proof of SN($\lambda 2$) breaks down at the point trying to formalize the predicate '$M \in [\![\sigma]\!]_\xi$'. The problem is that SAT is a third-order predicate.

The property SN does not hold for the systems $\lambda\mu$ and $\lambda\cap$. This is obvious, since all lambda terms can be typed in these two systems. However, there is a restriction of $\lambda\cap$ that does satisfy SN.

Let $\lambda\cap^-$ be the system $\lambda\cap$ without the type constant ω. The following result is an interesting characterization of strongly normalizing terms.

Theorem 4.3.12 (van Bakel; Krivine).

$$M \textit{ can be typed in } \lambda\cap^- \quad\Leftrightarrow\quad M \textit{ is strongly normalizing.}$$

Proof. See van Bakel (1990), theorem 3.4.11 or Krivine (1990), p. 65. ■

4.4 Decidability of type assignment

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

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

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

1. Given M and σ, does one have $\vdash M : \sigma$?

2. Given M, does there exists a σ such that $\vdash M : \sigma$?

3. Given σ, does there exists 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 $? : \sigma$.

In this subsection the decidability of these three problems will be examined for the various systems. The results can be summarized as follows:

Decidability of type checking, typability and inhabitation

	$\boldsymbol{M : \sigma}$?	$\boldsymbol{M}$: ?	? : $\boldsymbol{\sigma}$
$\boldsymbol{\lambda\to}$	yes	yes	yes
$\boldsymbol{\lambda 2}$	??	??	no
$\boldsymbol{\lambda\mu}$	yes	yes, always	yes, always
$\boldsymbol{\lambda\cap}$	no	yes, always	??
$\boldsymbol{\lambda\to^+}$	no	no	yes
$\boldsymbol{\lambda 2^+}$	no	no	no
$\boldsymbol{\lambda\mu^+}$	no	yes, always	yes, always
$\boldsymbol{\lambda\to\mathcal{A}}$	no	no	yes, always
$\boldsymbol{\lambda 2\mathcal{A}}$	no	no	yes, always
$\boldsymbol{\lambda\mu\mathcal{A}}$	no	yes, always	yes, always
$\boldsymbol{\lambda\cap\mathcal{A}}$	no	yes, always	yes, always

Remarks 4.4.1. The system $\lambda\cap^+$ is the same as $\lambda\cap$ and therefore it is not mentioned. The two question marks for $\lambda 2$ indicate—to quote Robin Milner—'embarrassing open problems'. For partial results concerning $\lambda 2$ and related systems see Pfenning (1988), Giannini and Ronchi (1988), Henglein (1990), and Kfoury et al. (1990). In 4.4.10 it will be shown that for $\lambda 2$ the decidability of type checking implies that of typability. It is generally believed that both problems are undecidable for $\lambda 2$.

Sometimes a question is trivially decidable, simply because the property always holds. Then we write 'yes, always'. For example in $\lambda\cap$ every term M has ω as type. For this reason it is more interesting to ask whether terms M are typable in a weaker system $\lambda\cap^-$. However, by theorem 4.3.12 this question is equivalent to the strong normalization of M and hence undecidable.

We first will show the decidability of the three questions for $\lambda\to$. This occupies 4.4.2 - 4.4.13 and in these items $\mathbb{T}$ stands for Type($\lambda\to$) and $\vdash$ for $\vdash_{\lambda\to\text{-Curry}}$.

Definition 4.4.2.

1. A substitutor is an operation
$$* : \mathbb{T}\to\mathbb{T}$$
such that
$$*(\sigma\to\tau) \equiv *(\sigma)\to*(\tau).$$
2. We write σ^* for $*(\sigma)$.
3. Usually a substitution $*$ has a finite support, that is, for all but finitely many type variables α one has $\alpha^* \equiv \alpha$ (the support of $*$ being $\sup(*) = \{\alpha \mid \alpha^* \not\equiv \alpha\}$).

In that case we write

$$*(\sigma) = \sigma[\alpha_1 := \alpha_1^*, \ldots, \alpha_n := \alpha_n^*],$$

where $\{\alpha_1, \ldots, \alpha_n\}$ is the support of $*$. We also write

$$* = [\alpha_1 := \alpha_1^*, \ldots, \alpha_n := \alpha_n^*].$$

Definition 4.4.3.

1. Let $\sigma, \tau \in T$. A *unifier* for σ and τ is a substitutor $*$ such that $\sigma^* \equiv \tau^*$.
2. The substitutor $*$ is a most general unifier for σ and τ if
   (a) $\sigma^* \equiv \tau^*$
   (b) $\sigma^{*_1} \equiv \tau^{*_1} \;\Rightarrow\; \exists *_2 \quad *_1 \equiv *_2 \circ *$.
3. Let $E = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$ be a finite set of equations between types. The equations do not need to be valid. A *unifier* for E is a substitutor $*$ such that $\sigma_1^* \equiv \tau_1^*$ & $\cdots$ & $\sigma_n^* \equiv \tau_n^*$. In that case one writes $* \models E$. Similarly one defines the notion of a most general unifier for E.

Examples 4.4.4. The types $\beta{\rightarrow}(\alpha{\rightarrow}\beta)$ and $(\gamma{\rightarrow}\gamma){\rightarrow}\delta$ have a unifier. For example $* = [\beta := \gamma{\rightarrow}\gamma,\ \delta := \alpha{\rightarrow}(\gamma{\rightarrow}\gamma)]$ or $*_1 = [\beta := \gamma{\rightarrow}\gamma,\ \alpha := \varepsilon{\rightarrow}\varepsilon,\ \delta := \varepsilon{\rightarrow}\varepsilon{\rightarrow}(\gamma{\rightarrow}\gamma)]$. The unifier $*$ is most general, $*_1$ is not.

Definition 4.4.5. σ is a *variant* of τ if for some $*_1$ and $*_2$ one has

$$\sigma = \tau^{*_1} \text{ and } \tau = \sigma^{*_2}.$$

Example 4.4.6. $\alpha{\rightarrow}\beta{\rightarrow}\beta$ is a variant of $\gamma{\rightarrow}\delta{\rightarrow}\delta$ but not of $\alpha{\rightarrow}\beta{\rightarrow}\alpha$.

Note that if $*_1$ and $*_2$ are both most general unifiers of say σ and τ, then σ^{*_1} and σ^{*_2} are variants of each other and similarly for τ.

The following result due to Robinson (1965) states that unifiers can be constructed effectively.

Theorem 4.4.7 (Unification theorem).

1. *There is a recursive function U having (after coding) as input a pair of types and as output either a substitutor or* fail *such that*

$$\sigma \text{ and } \tau \text{ have a unifier} \quad \Rightarrow \quad U(\sigma, \tau) \text{ is a most general unifier}$$

$$\begin{array}{rcl} & & \textit{for } \sigma \textit{ and } \tau; \\ \sigma \textit{ and } \tau \textit{ have no unifier} & \Rightarrow & U(\sigma,\tau) = \mathsf{fail}. \end{array}$$

2. *There is (after coding) a recursive function U having as input finite sets of equations between types and as output either a substitutor or* fail *such that*

$$\begin{array}{rcl} E \textit{ has a unifier} & \Rightarrow & U(E) \textit{ is a most general unifier for } E; \\ E \textit{ has no unifier} & \Rightarrow & U(E) = \mathsf{fail}. \end{array}$$

Proof. Note that $\sigma_1{\rightarrow}\sigma_2 \equiv \tau_1{\rightarrow}\tau_2$ holds iff $\sigma_1 \equiv \tau_1$ and $\sigma_2 \equiv \tau_2$ hold.

1. Define $U(\sigma,\tau)$ by the following recursive loop, using case distinction.

$$\begin{array}{rcll} U(\alpha,\tau) & = & [\alpha := \tau], & \text{if } \alpha \notin \mathrm{FV}(\tau), \\ & = & Id, \text{ the identity}, & \text{if } \tau = \alpha, \\ & = & \mathsf{fail}, & \text{else}; \end{array}$$

$$\begin{array}{rcl} U(\sigma_1{\rightarrow}\sigma_2,\alpha) & = & U(\alpha,\sigma_1{\rightarrow}\sigma_2); \\ U(\sigma_1{\rightarrow}\sigma_2,\tau_1{\rightarrow}\tau_2) & = & U(\sigma_1^{U(\sigma_2,\tau_2)},\tau_1^{U(\sigma_2,\tau_2)}) \circ U(\sigma_2,\tau_2), \end{array}$$

   where this last expression is considered to be fail if one of its parts is. Let $\#_{var}(\sigma,\tau)$ ='the number of variables in $\sigma{\rightarrow}\tau$' and $\#_{\rightarrow}(\sigma,\tau)$='the number of arrows in $\sigma{\rightarrow}\tau$'. By induction on $(\#_{var}(\sigma,\tau),\#_{\rightarrow}(\sigma,\tau))$ ordered lexicographically one can show that $U(\sigma,\tau)$ is always defined. Moreover U satisfies the specification.

2. If $E = \{\sigma_1 = \tau_1,\ldots,\sigma_n = \tau_n\}$, then define $U(E) = U(\sigma,\tau)$, where $\sigma = \sigma_1{\rightarrow}\cdots{\rightarrow}\sigma_n$ and $\tau = \tau_1{\rightarrow}\cdots{\rightarrow}\tau_n$. ■

■

See Section 7 in Klop's chapter in this handbook for more on unification. The following theorem is essentially due to Wand (1987) and simplifies the proof of the decidability of type checking and typability for $\lambda{\rightarrow}$.

Proposition 4.4.8. *For every basis Γ, term $M \in \Lambda$ and $\sigma \in \mathbb{T}$ such that $\mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$ there is a finite set of equations $E = E(\Gamma,M,\sigma)$ such that for all substitutors $*$ one has*

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

$$\Gamma^* \vdash M : \sigma^* \quad \Rightarrow \quad *_1 \models E(\Gamma, M, \sigma), \tag{2}$$

for some $*_1$ such that $*$ and $*_1$ have the same effect on the type variables in Γ and σ.

Proof. Define $E(\Gamma, M, \sigma)$ by induction on the structure of M:

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

By induction on M one can show (using the generation lemma (3.1.8)) that (1) and (2) hold. ■ ■

Definition 4.4.9.

1. Let $M \in \Lambda$. Then (Γ, σ) is a *principal pair* (pp) for M if

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

   (2) $\Gamma' \vdash M : \sigma' \ \Rightarrow \ \exists * \, [\Gamma^* \subseteq \Gamma' \ \ \& \ \ \sigma^* \equiv \sigma']$.

   Here $\{x_1{:}\sigma_1, \ldots\}^* = \{x_1{:}\sigma_1^*, \ldots\}$.

2. Let $M \in \Lambda$ be closed. Then σ is a *principal type* (pt) for M if

   (1) $\vdash M : \sigma$

   (2) $\vdash M : \sigma' \ \Rightarrow \ \exists * \ [\sigma^* \equiv \sigma']$.

Note that if (Γ, σ) is a *pp* for M, then every variant (Γ', σ') of (Γ, σ), in the obvious sense, is also a *pp* for M. Conversely if (Γ, σ) and (Γ', σ') are *pp*'s for M, then (Γ', σ') is a variant of (Γ, σ). Similarly for closed terms and *pt*'s. Moreover, if (Γ, σ) is a *pp* for M, then $\mathrm{FV}(M) = \mathrm{dom}(\Gamma)$.

The following result is independently due to Curry (1969), Hindley (1969) and Milner (1978). It shows that for $\lambda{\to}$ the problems of type checking and typability are decidable.

Theorem 4.4.10 (Principal type theorem for $\lambda{\to}$-Curry).

1. *There exists (after coding) a recursive function pp such that one has*

$$M \text{ has a type} \quad \Rightarrow \quad pp(M) = (\Gamma, \sigma), \text{ where } (\Gamma, \sigma) \text{ is a } pp \text{ for } M;$$

$$M \text{ has no type} \quad \Rightarrow \quad pp(M) = \mathsf{fail}.$$

2. *There exists (after coding) a recursive function pt such that for closed terms M one has*

$$\begin{array}{rcl} M \text{ has a type} & \Rightarrow & pt(M) = \sigma, \text{ where } \sigma \text{ is a pt for } M; \\ M \text{ has no type} & \Rightarrow & pt(M) = \mathsf{fail}. \end{array}$$

Proof. 1. Let $\mathrm{FV}(M) = \{x_1, \ldots, x_n\}$ and set $\Gamma_0 = \{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\}$ and $\sigma_0 = \beta$. Note that

$$\begin{array}{rcl} M \text{ has a type} & \Leftrightarrow & \exists \Gamma\ \exists \sigma\ \ \Gamma \vdash M : \sigma \\ & \Leftrightarrow & \exists *\ \ \Gamma_0^* \vdash M : \sigma_0^* \\ & \Leftrightarrow & \exists *\ \ * \models E(\Gamma_0, M, \sigma_0). \end{array}$$

Define

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

Then $pp(M)$ satisfies the requirements. Indeed, if M has a type, then $U(E(\Gamma_0, M, \sigma_0)) = *$ is defined and $\Gamma_0^* \vdash M : \sigma_0^*$ by (1) in proposition 4.4.8. To show that (Γ_0^*, σ_0^*) is a pp, suppose that also $\Gamma' \vdash M : \sigma'$. Let $\widetilde{\Gamma} = \Gamma' \upharpoonright \mathrm{FV}(M)$; write $\widetilde{\Gamma} = \Gamma_0^{*_0}$ and $\sigma' = \sigma_0^{*_0}$. Then also $\Gamma_0^{*_0} \vdash M : \sigma_0^{*_0}$. Hence by (2) in proposition 4.4.8 for some $*_1$ (acting the same as $*_0$ on Γ_0, σ_0) one has $*_1 \models E(\Gamma_0, M, \sigma_0)$. Since $*$ is a most general unifier (proposition 4.4.7) one has $*_1 = *_2 \circ *$ for some $*_2$. Now indeed

$$(\Gamma_0^*)^{*_2} = \Gamma_0^{*_1} = \Gamma_0^{*_0} = \widetilde{\Gamma} \subseteq \Gamma'$$

and

$$(\sigma_0^*)^{*_2} = \sigma_0^{*_1} = \sigma_0^{*_0} = \sigma'.$$

If M has no type, then $\neg\exists *\ \ * \models E(\Gamma_0, M, \sigma_0)$ hence

$$U(\Gamma_0, M, \sigma_0) = \mathsf{fail} = pp(M).$$

2. Let M be closed and $pp(M) = (\Gamma, \sigma)$. Then $\Gamma = \emptyset$ and we can put $pt(M) = \sigma$. ■

∎

Corollary 4.4.11. *Type checking and typability for* $\lambda\!\rightarrow$ *are decidable.*

Proof. As to type checking, let M and σ be given. Then

$$\vdash M : \sigma \iff \exists * \, [\sigma = pt(M)^*].$$

This is decidable (as can be seen using an algorithm—*pattern matching*—similar to the one in Theorem 4.4.7).

As to the question of typability, let M be given. Then M has a type iff $pt(M) \neq \mathsf{fail}$. ∎ ∎

Theorem 4.4.12. *The inhabitation problem for* $\lambda\!\rightarrow$, *i.e.*

$$\exists M \in \Lambda \;\vdash_{\lambda\rightarrow} M : \sigma$$

is a decidable property of σ.

Proof. One has by Corollary 3.2.16 that

$$\begin{aligned} \sigma \text{ inhabited in } \lambda\!\rightarrow\!\text{-Curry} &\iff \sigma \text{ inhabited in } \lambda\!\rightarrow\!\text{-Church} \\ &\iff \sigma \text{ provable in PROP,} \end{aligned}$$

where PROP is the minimal intuitionistic proposition calculus with only $\rightarrow$ as connective and σ is considered as an element of PROP, see Section 5.4. Using finite Kripke models it can be shown that the last statement is decidable. Therefore the first statement is decidable too. ∎ ∎

Without a proof we mention the following result of Hindley (1969).

Theorem 4.4.13 (Second principal type theorem for $\lambda\!\rightarrow$-Curry). *Every type* $\sigma \in \mathbb{T}$ *there exists a basis* Γ *and term* $M \in \Lambda$ *such that* (Γ, σ) *is a pp for* M.

Now we consider $\lambda 2$. The situation is as follows. The question whether type checking and typability are decidable is open. However, one has the following result by Malecki (1989).

Proposition 4.4.14. *In* $\lambda 2$ *the problem of typability can be reduced to that of type checking. In particular*

$$\{(M : \sigma) \mid \vdash_{\lambda 2} M : \sigma\} \text{ is decidable } \Rightarrow \{M \mid \exists \sigma \;\vdash_{\lambda 2} M : \sigma\} \text{ is decidable.}$$

Proof. One has

$$\exists \sigma \vdash M : \sigma \;\Leftrightarrow\; \vdash (\lambda xy.y)M : (\alpha\!\rightarrow\!\alpha).$$

The implication $\Rightarrow$ is obvious, since $\vdash (\lambda xy.y) : (\sigma\!\rightarrow\!\alpha\!\rightarrow\!\alpha)$ for all σ. The implication $\Leftarrow$ follows from Proposition 4.1.18. ∎

Theorem 4.4.15. *The inhabitation problem for* $\lambda 2$ *is undecidable.*

Proof. As for $\lambda\!\rightarrow$ one can show that

$$\begin{aligned} \sigma \text{ inhabited in } \lambda 2\text{-Curry} &\iff \sigma \text{ inhabited in } \lambda 2\text{-Church} \\ &\iff \sigma \text{ provable in PROP2}, \end{aligned}$$

where PROP2 is the constructive second-order proposition calculus. In Löb (1976) it is proved that this last property is undecidable. ∎ ∎

Proposition 4.4.16. *For $\lambda\mu$ one has the following:*

1. *Type checking is decidable.*
2. *Typability is trivially decidable: every λ-term has a type.*
3. *The inhabitation problem for $\lambda\mu$ is trivially decidable: all types are inhabited.*

Proof. 1. See Coppo and Cardone (to appear) who use the same method as for $\lambda\!\rightarrow$ and the fact that $T(\sigma) = T(\tau)$ is decidable.

2. Let $\sigma_0 = \mu\alpha.\alpha\!\rightarrow\!\alpha$. Then every $M \in \Lambda$ has type σ_0, see the example before 4.1.
3. All types are inhabited by Ω, see 4.1.12 (2). ∎

∎

Lemma 4.4.17. *Let $\lambda-$ be a system of type assignment that satisfies subject conversion, i.e.*

$$\Gamma \vdash_{\lambda-} M : \sigma \;\&\; M =_\beta N \;\;\Rightarrow\;\; \Gamma \vdash_{\lambda-} N : \sigma.$$

1. *Suppose some closed terms have type $\alpha\!\rightarrow\!\alpha$, others not.*
   Then the problem of type checking is undecidable.
2. *Suppose some terms have a type, other terms not.*
   Then the problem of typability is undecidable.

Proof. 1. If the set $\{(M, \sigma) \mid \vdash M : \sigma\}$ is decidable, then so is $\{M \mid \vdash M : \alpha\!\rightarrow\!\alpha\}$. But this set is by assumption closed under $=$ and non-trivial, contradicting Scott's theorem 2.2.15.

2. Similarly. ∎

■

Proposition 4.4.18. *For $\lambda\cap$ one has the following:*

1. *Type checking problem is undecidable.*
2. *Typability is trivially decidable: all terms have a type.*

Proof. 1. Lemma 4.4.17(1) applies by 4.2.7, the fact that $\vdash \mathsf{I} : \alpha{\to}\alpha$ and Exercise 4.1.20.

2. For all M one has $M : \omega$. ■

■

It is not known whether inhabitation in $\lambda\cap$ is decidable.

Lemma 4.4.19. *Let $\lambda-$ be one of the systems à la Curry. Then*

1. $\Gamma \vdash_{\lambda-^+} M : \sigma \;\;\Leftrightarrow\;\; \exists M' \, [M \twoheadrightarrow_\beta M' \;\&\; \Gamma \vdash_{\lambda-} M' : \sigma]$.
2. *σ is inhabited in $\lambda-^+$ $\;\Leftrightarrow\;$ σ is inhabited in $\lambda-$.*

Proof. 1. ($\Leftarrow$) Trivial, since $M \twoheadrightarrow_\beta M'$ implies $M =_\beta M'$. ($\Rightarrow$) By induction on the derivation of $M : \sigma$. The only interesting case is when the last applied rule is an application of rule EQ. So let it be

$$\frac{M_1 : \sigma \quad M_1 = M}{M : \sigma}.$$

The induction hypothesis says that for some M_1' with $M_1 \twoheadrightarrow_\beta M_1'$ one has $\Gamma \vdash_{\lambda-} M_1' : \sigma$. By the Church–Rosser theorem 2.3.7 M_1' and M have a common reduct, say M'. But then by the subject reduction theorem one has $\Gamma \vdash_{\lambda-} M' : \sigma$ and we are done.

2. By (1). ■

■

Proposition 4.4.20. *For the systems $\lambda-^+$ one has the following:*

1. *Type checking is undecidable.*
2. *Typability is undecidable for $\lambda{\to}^+$ and $\lambda 2^+$, but trivially decidable for $\lambda\mu^+$ and $\lambda\cap^+$.*
3. *The status of the inhabitation problem for $\lambda-^+$ is the same as for $\lambda-$.*

Proof. 1. By definition subject conversion holds for the systems $\lambda-^+$. In all systems $\mathsf{I} : \alpha{\to}\alpha$. From Lemma 4.4.19(1) and Exercise 4.1.20 it follows that Lemma 4.4.17(1) applies.

2. By Theorems 4.3.6 and 4.3.11 terms without an nf have no type in $\lambda{\to}$ or $\lambda 2$. Hence by Lemma 4.4.19(1) these terms have no type in

$\lambda\!\rightarrow^+$ or $\lambda 2^+$. Since for these systems there are terms having a type lemma 4.4.17(2) applies.

In $\lambda\mu^+$ and $\lambda\cap^+$ all terms have a type.

3. By Lemma 4.4.19(2). ■

■

Lemma 4.4.21. *Let M be a term in nf. Then*

$$\vdash_{\lambda - \mathcal{A}} M : \sigma \quad \Rightarrow \vdash_{\lambda -} M : \sigma.$$

Proof. By induction on the given derivation, using that $M \in \mathcal{A}(M)$. ■ ■

Proposition 4.4.22. *For the systems $\lambda - \mathcal{A}$ the situation is as follows:*

1. *The problem of type checking is undecidable for the systems $\lambda\!\rightarrow\!\mathcal{A}$, $\lambda 2\mathcal{A}$, $\lambda\mu\mathcal{A}$ and $\lambda\cap\mathcal{A}$.*
2. *The problem of typability is undecidable for the system $\lambda\!\rightarrow\!\mathcal{A}$ and $\lambda 2\mathcal{A}$ but trivially decidable for the systems $\lambda\mu\mathcal{A}$ and $\lambda\cap\mathcal{A}$ (all terms are typable).*
3. *The problem of inhabitation is trivially decidable for all four systems including rule $\mathcal{A}$ (all types are inhabited).*

Proof. 1. By Lemma 4.4.21 and Exercise 4.1.20 one has $\not\vdash \mathsf{K} : \alpha\!\rightarrow\!\alpha$. Hence 4.4.17(1) applies.

2. Similarly.

3. The inhabitation problem becomes trivial: in all four systems one has

$$\vdash \Omega : \sigma$$

for all types σ. This follows from Example 4.1.3(2) and the facts that $\mathsf{YI} =_\beta \Omega$ and $\lambda - \mathcal{A}$ is closed under the rule EQ.

■

The results concerning decidability of type checking, typability and inhabitation are summarised in the table at the beginning of this subsection.

5 Typing *à la* Church

In this section several systems of typed lambda calculus will be described in a uniform way. Church versions will be given for the systems $\lambda\!\rightarrow$ and $\lambda 2$, already encountered in the Curry style. Then a collection of eight lambda-calculi *à la* Church is given, the so called *λ-cube*. Two of the cornerstones of this cube are essentially $\lambda\!\rightarrow$ and $\lambda 2$ and another system is among the family of AUTOMATH languages of de Bruijn (1980). The λ-cube forms a natural fine structure of the *calculus of constructions* of Coquand and Huet (1988) and is organized according to the possible 'dependencies' between terms and types. This will be done in 5.1.

The description method of the systems in the λ-cube is generalized in subsection 5.2, obtaining the so called 'pure type systems' (PTSs). In preliminary versions of this chapter PTSs were called 'generalized type systems' (GTSs). Several elementary properties of PTS's are derived.

In subsection 5.3 it is shown that all terms in the systems of the λ-cube are strongly normalizing. However in 5.5 it turns out that this is not generally true in PTS's.

In subsection 5.4 a cube of eight logical systems will be described. Each logical system L_i corresponds to one of the systems λ_i on the λ-cube. One has for sentences A

$$\vdash_{L_i} A \quad \Rightarrow \quad \exists M \; \Gamma \vdash_{\lambda_i} M : [\![A]\!]$$

where Γ depends on the similarity type of the language of L_i and $[\![A]\!]$ is a canonical interpretation of A in λ_i. Moreover, the term M can be found uniformly from the proof of A in L_i. The map $[\![-]\!]$ is called the *propositions-as-types* interpretation. It turns out also that the logical systems can be described as PTSs and that in this way the propositions-as-type interpretation becomes a very simple forgetful map from the logical cube into the λ-cube.

As an application of the propositions-as-types interpretation one can represent in a natural way data types in $\lambda 2$. Data types correspond to inductively defined sets and these can be naturally represented in second-order predicate logic, one of the systems on the logical cube. Then, by means of a map from predicate to proposition logic and by the propositions-as-types interpretation one obtains an interpretation of data types in $\lambda 2$.

5.1 The cube of typed lambda calculi

In this subsection we introduce in a uniform way the eight typed lambda calculi $\lambda\!\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$, $\lambda\omega$, λP, λP2, λP$\underline{\omega}$, and λPω. (The system λPω is often called λC.) The eight systems form a cube as follows:

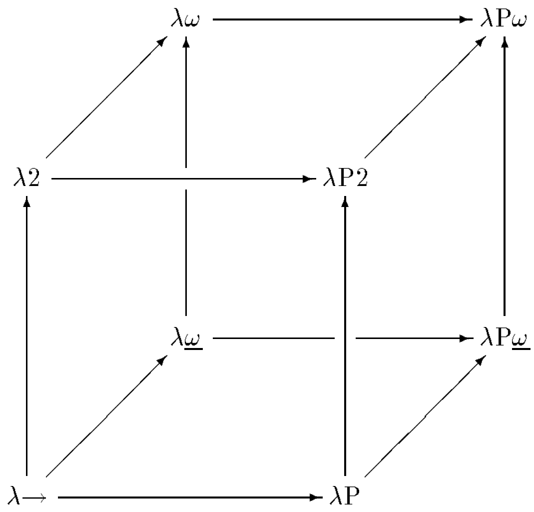


Fig. 2. The λ-cube.

where each edge $\rightarrow$ represents the inclusion relation $\subseteq$. This cube will be referred to as the λ-*cube*.

The system $\lambda\!\rightarrow$ is the simply typed lambda calculus, already encountered in section 3.2. The system $\lambda 2$ is the *polymorphic* or *second order* typed lambda calculus and is essentially the system F of Girard (1972); the system has been introduced independently in Reynolds (1974). The Curry version of $\lambda 2$ was already introduced in Section 4.1. The system $\lambda\omega$ is essentially the system $F\omega$ of Girard (1972). The system λP reasonably corresponds to one of the systems in the family of AUTOMATH languages, see de Bruijn (1980). (A more precise formulation of several AUTOMATH systems can be given as PTSs, see subsection 5.2.) This system λP appears also under the name *LF* in Harper *et al.* (1987). The system λP2 is studied in Longo and Moggi (1988) under the same name. The system $\lambda\mathrm{C} = \lambda\mathrm{P}\omega$ is one of the versions of the calculus of constructions introduced by Coquand and Huet (1988). The system $\lambda\underline{\omega}$ is related to a system studied by Renardel de Lavalette (1991). The system $\lambda\mathrm{P}\underline{\omega}$ seems not to have been studied before. (For $\lambda\underline{\omega}$ and $\lambda\mathrm{P}\underline{\omega}$ read: 'weak $\lambda\omega$' and 'weak $\lambda\mathrm{P}\omega$' respectively.)

As we have seen in Section 4, the system $\lambda\!\rightarrow$ and $\lambda 2$ can be given also *à la* Curry. A Curry version of $\lambda\omega$ appears in Giannini and Ronchi (1988) and something similar can probably be done for $\lambda\underline{\omega}$. On the other hand, no natural Curry versions of the systems $\lambda\mathrm{P}, \lambda\mathrm{P}2, \lambda\mathrm{P}\underline{\omega}$ and $\lambda\mathrm{C}$ seem possible.

Now first the systems $\lambda\!\rightarrow$ and $\lambda 2$ *à la* Church will be introduced in the usual way. Also $\lambda\underline{\omega}$ and λP will be defined. Then the λ-cube will be defined in a uniform way and two of the systems on it turn out to be equivalent to $\lambda\!\rightarrow$ and $\lambda 2$.

$\lambda\!\rightarrow$-Church

Although this system has been introduced already in subsection 3.2, we will repeat its definition in a stylistic way, setting the example for the definition of the other systems.

Definition 5.1.1. The system $\lambda\!\rightarrow$-Church consists of a set of types $\mathbb{T} = \text{type}(\lambda\!\rightarrow)$, a set of pseudoterms $\Lambda_{\mathbb{T}}$, a set of bases, a conversion (and reduction) relation on $\Lambda_{\mathbb{T}}$ and a type assignment relation $\vdash$.

The sets $\mathbb{T}$ and $\Lambda_{\mathbb{T}}$ are defined by an abstract syntax, bases are defined explicitly, the conversion relation is defined by a contraction rule and $\vdash$ is defined by a deduction system as follows:

1. Types	$\mathbb{T} = \mathbb{V} \mid \mathbb{T}\!\rightarrow\!\mathbb{T}$;
2. Pseudoterms	$\Lambda_{\mathbb{T}} = V \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \lambda V{:}\mathbb{T}.\Lambda$;
3. Bases	$\Gamma = \{x_1{:}A_1, \ldots, x_n{:}A_n\}$, with all x_i distinct and all $A_i \in \mathbb{T}$;
4. Contraction rule	$(\lambda x{:}A.M)N \rightarrow_\beta M[x := N]$;
5. Type assignment	$\Gamma \vdash M : A$ is defined as follows.

$\lambda\!\rightarrow$

(start-rule)	$\dfrac{(x{:}A) \in \Gamma}{\Gamma \vdash x{:}A}$;
($\rightarrow$-elimination)	$\dfrac{\Gamma \vdash M : (A\!\rightarrow\!B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$;
($\rightarrow$-introduction)	$\dfrac{\Gamma,\ x{:}A \vdash M{:}B}{\Gamma \vdash (\lambda x{:}A.M) : (A\!\rightarrow\!B)}$.

Remarks 5.1.2.

1. In 1 the character $\mathbb{V}$ denotes the syntactic category of type variables. Similarly in 2 the character V denotes the category of term variables. In 4 the letter x denotes an arbitrary term variable. In 3 the $x_1, \ldots, x_n$ are distinct term variables. In 4 and 5 the letters A, B denote arbitrary types and M, N arbitrary pseudoterms. The basis $\Gamma, x{:}A$ stands for $\Gamma \cup \{x{:}A\}$, where it is necessary that x is a variable that does not occur in Γ.

2. A pseudoterm M is called *legal* if for some Γ and A one has $\Gamma \vdash M{:}A$.

Typical examples of type assignments in $\lambda\!\rightarrow$ are the following. Let $A, B \in \mathbb{T}$.

$$\begin{array}{rcl} & \vdash & (\lambda a{:}A.a) : (A \rightarrow A); \\ b{:}B & \vdash & (\lambda a{:}A.b) : (A \rightarrow B); \\ b{:}A & \vdash & ((\lambda a{:}A.a)b) : A; \\ c{:}A, b{:}B & \vdash & (\lambda a{:}A.b)c : B; \\ & \vdash & (\lambda a{:}A.\lambda b{:}B.a) : (A \rightarrow B \rightarrow A). \end{array}$$

The system λT

Type and term constants are not officially introduced in this chapter. However, these are useful to make axiomatic extensions of $\lambda\!\rightarrow$ in which certain terms and types play a special role. We will simulate constants via variables. For example one may select a type variable $\mathbf{0}$ and term variables $0, S$ and R_σ for each σ in $\mathbb{T}$ as constants: one postulates in an initial context the following.

$$\begin{array}{rcl} 0 & : & \mathbf{0}; \\ S & : & \mathbf{0} \rightarrow \mathbf{0}; \\ R_\sigma & : & (\sigma \rightarrow (\sigma \rightarrow \mathbf{0} \rightarrow \sigma) \rightarrow \mathbf{0} \rightarrow \sigma). \end{array}$$

Further one extends the definitional equality by adding to the β-contraction rule the following contraction rule for R_σ.

$$\begin{array}{rcl} R_\sigma M N 0 & \rightarrow & M; \\ R_\sigma M N (Sx) & \rightarrow & N(R_\sigma M N x)x. \end{array}$$

This extension of $\lambda\!\rightarrow$ is called λT or *Gödel's theory T of primitive recursive functionals* ('Gödel's T'). The type $\mathbf{0}$ stands for the natural numbers with element 0 and successor function S; the R_σ stand for the recursion operator creating recursive functionals of type $\mathbf{0} \rightarrow \sigma$. In spite of the name, more than just the primitive recursive functions are representable. This is because recursion is allowed on higher functionals; see e.g. Barendregt (1984), appendix A.2.1. and Terlouw (1982) for an analysis.

λ2-*Church*

Definition 5.1.3. The system λ2-Church is defined as follows:

1. Types $\mathbb{T} = \mathbb{V} \mid \mathbb{T}{\rightarrow}\mathbb{T} \mid \forall\mathbb{V}\mathbb{T}$;
2. Pseudoterms $\Lambda_{\mathbb{T}} = V \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \Lambda_{\mathbb{T}}\mathbb{T} \mid \lambda V{:}\mathbb{T}\Lambda_{\mathbb{T}} \mid \Lambda\mathbb{V}\Lambda_{\mathbb{T}}$;
3. Bases $\Gamma = \{x_1{:}A_1, \ldots, x_n{:}A_n\}$, with $\vec{x}$ distinct and $\vec{A} \in \mathbb{T}$;
4. Contraction rules $(\lambda a{:}A.M)N \rightarrow_\beta M[a := N]$
   $(\Lambda\alpha.M)A \rightarrow_\beta M[\alpha := A]$
5. Type assignment $\Gamma \vdash M : A$ is defined as follows.

$\lambda 2$

(start-rule) $$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash x : A},$$

($\rightarrow$-elimination) $$\frac{\Gamma \vdash M : (A{\rightarrow}B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B};$$

($\rightarrow$-introduction) $$\frac{\Gamma, a{:}A \vdash M : B}{\Gamma \vdash (\lambda a{:}A.M) : (A{\rightarrow}B)};$$

($\forall$-elimination) $$\frac{\Gamma \vdash M : (\forall\alpha.A)}{\Gamma \vdash MB : A[\alpha{:}= B]}, \quad B \in \mathbb{T};$$

($\forall$-introduction) $$\frac{\Gamma \vdash M : A}{\Gamma \vdash (\Lambda\alpha.M) : (\forall\alpha.A)}, \quad \alpha \notin \mathrm{FV}(\Gamma).$$

Typical assignments in $\lambda 2$ are the following:

$$\begin{array}{rl} & \vdash (\lambda a{:}\alpha.a) : (\alpha{\rightarrow}\alpha); \\ & \vdash (\Lambda\alpha\lambda a{:}\alpha.a) : (\forall\alpha.\alpha{\rightarrow}\alpha); \\ & \vdash (\Lambda\alpha\lambda a{:}\alpha.a)A : (A{\rightarrow}A); \\ b{:}A & \vdash (\Lambda\alpha\lambda a{:}\alpha.a)Ab : A; \end{array}$$

{of course the following reduction holds:

$$(\Lambda\alpha\lambda a{:}\alpha.a)Ab \rightarrow (\lambda a{:}A.a)b \rightarrow b;\}$$
$$\vdash (\Lambda\beta\lambda a{:}(\forall\alpha.\alpha).a((\forall\alpha.\alpha){\rightarrow}\beta)a) : (\forall\beta.(\forall\alpha.\alpha){\rightarrow}\beta);$$

{for this last example one has to think twice to see that it is correct; a simpler term of the same type is the following}

$$\vdash (\Lambda\beta\lambda a{:}(\forall\alpha{:}\alpha).a\beta) : (\forall\beta.(\forall\alpha.\alpha){\rightarrow}\beta).$$

Without a proof we mention that the Church–Rosser property holds for reduction on pseudoterms in $\lambda 2$.

Dependency

Types and terms are mutually dependent; there are

terms depending on terms;
terms depending on types;
types depending on terms;
types depending on types.

The first two sorts of dependency we have seen already. Indeed, in $\lambda\!\!\rightarrow$ we have

$$F : A \rightarrow B \quad M : A \quad \Rightarrow \quad FM : B.$$

Here FM is a term depending on a term (e.g. on M). For $\lambda 2$ we saw

$$G : \forall \alpha . \alpha \rightarrow \alpha \quad A \text{ a type} \quad \Rightarrow \quad GA : A \rightarrow A.$$

Hence for $G = \Lambda\alpha\lambda a{:}\alpha.a$ one has that GA is a term depending on the type A.

In $\lambda\!\!\rightarrow$ and $\lambda 2$ one has also function abstraction for the two dependencies. For the two examples above

$$\lambda m{:}A.Fm : A \rightarrow B,$$

$$\Lambda\alpha.G\alpha : \forall\alpha.\alpha \rightarrow \alpha.$$

Now we shall define two other systems $\lambda\underline{\omega}$ and λP with *types* FA (FM resp) depending on types (respectively terms). We will also have function abstraction for these dependencies in $\lambda\underline{\omega}$ and λP.

Types depending on types; the system $\lambda\underline{\omega}$

A natural example of a type depending on another type is $\alpha \rightarrow \alpha$ that depends on α. In fact it is natural to define $f = \lambda\alpha \in \mathbb{T}.\alpha \rightarrow \alpha$ such that $f(\alpha) = \alpha \rightarrow \alpha$. 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 $\sigma : *$ corresponds to $\sigma \in \mathbb{T}$. The informal statement

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

now becomes the formal

$$\alpha{:}*, \beta{:}* \vdash (\alpha \rightarrow \beta) : *.$$

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

$$\mathbb{K} = * \mid \mathbb{K} \rightarrow \mathbb{K}.$$

That is $\mathbb{K} = \{*, * \rightarrow *, * \rightarrow * \rightarrow *, \ldots\}$. 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 $\vdash (\lambda\alpha{:}*.\alpha{\rightarrow}\alpha) : (*{\rightarrow}*)$, i.e. our f is a constructor of kind $*{\rightarrow}*$. Each element of $\mathbb{T}$ will be a constructor of kind $*$.

Although types and terms of $\lambda\underline{\omega}$ can be kept separate, we will consider them as subsets of one general set $\mathcal{T}$ of pseudo expressions. This is a preparation to 5.1.8, 5.1.9 and 5.1.10 in which it is essential that types and terms are being mixed.

Definition 5.1.4 (Types and terms of $\lambda\underline{\omega}$).

1. A set of pseudo-expressions $\mathcal{T}$ is defined as follows

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

   where V is an infinite collection of variables and C of constants.

2. Among the constants C two elements are selected and given the names $*$ and $\Box$. These so called *sorts* $*$ and $\Box$ are the main reason to introduce constants.

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

$$\begin{array}{rcl} \alpha{:}*, x{:}\alpha & \vdash & x : \alpha; \\ \alpha{:}* & \vdash & (\lambda x{:}\alpha.x) : (\alpha{\rightarrow}\alpha) \end{array}$$

but not

$$\begin{array}{rcl} x{:}\alpha, \alpha{:}* & \vdash & x : \alpha; \\ x{:}\alpha & \vdash & (\lambda\alpha{:}*.x) : (*{\rightarrow}\alpha) \end{array}$$

in which α occurs both free and bound.

Definition 5.1.5 (Contexts for $\lambda\underline{\omega}$).

1. A *statement* of $\lambda\underline{\omega}$ is of the form $M : A$ with $M, A \in \mathcal{T}$.

2. A *context* is a finite linearly ordered set of statements with distinct variables as subjects. $\Gamma, \Delta, \ldots$ range over contexts.

3. $<>$ denotes the empty context. If $\Gamma = <x_1{:}A_1, \ldots, x_n{:}A_n>$ then $\Gamma, y{:}B = <x_1{:}A_1, \ldots, x_n{:}A_n, y{:}B>$.

Definition 5.1.6 (Typing rules for $\lambda\underline{\omega}$). The notion $\Gamma \vdash_{\lambda\underline{\omega}} M : A$ is defined by the following axiom and rules. The letter s ranges over $\{*, \Box\}$.

$\lambda\underline{\omega}$

(axiom)	$<> \vdash * : \square;$
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$, $x \notin \Gamma;$
(weakening rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$, $x \notin \Gamma;$
(type/kind formation)	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash (A{\rightarrow}B) : s}.$
(application rule)	$\dfrac{\Gamma \vdash F : (A{\rightarrow}B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B};$
(abstraction rule)	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (A{\rightarrow}B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (A{\rightarrow}B)};$
(conversion rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}.$

Example 5.1.7.

$$\begin{array}{rll} \alpha{:}*, \beta{:}* & \vdash_{\lambda\underline{\omega}} & (\alpha{\rightarrow}\beta) : *; \\ \alpha{:}*, \beta{:}*, x{:}(\alpha{\rightarrow}\beta) & \vdash_{\lambda\underline{\omega}} & x : (\alpha{\rightarrow}\beta); \\ \alpha{:}*, \beta{:}* & \vdash_{\lambda\underline{\omega}} & (\lambda x{:}(\alpha{\rightarrow}\beta).x) : ((\alpha{\rightarrow}\beta){\rightarrow}(\alpha{\rightarrow}\beta)). \end{array}$$

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

$$\begin{array}{rll} & \vdash_{\lambda\underline{\omega}} & D : (*{\rightarrow}*). \\ \alpha{:}* & \vdash_{\lambda\underline{\omega}} & (\lambda x{:}D\alpha.x) : D(D\alpha). \end{array}$$

Types depending on terms; the system λP

An intuitive example of a type depending on a term is $A^n{\rightarrow}B$ with n a natural number. In order to formalize 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 $A{\rightarrow}k$ is a kind. In particular $A{\rightarrow}*$ is a kind. Then if $f : A{\rightarrow}*$ and $a : A$, one has $fa : *$. This fa is a term dependent type. Moreover one has function abstraction for this dependency.

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 want to form the function

$$\lambda a{:}A.b_a$$

that should have as type the cartesian product

$$\Pi a{:}A.B_a$$

of the B_a's. Once these product types are allowed, the function space type of A and B can be written as

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

where a is a variable not occurring in B. This is analogous to the fact that a product of equal numbers is a power:

$$\prod_{i=1}^{n} b_i = b^n$$

provided that $b_i = b$ for $1 \leq i \leq n$. So by using products, the type constructor $\rightarrow$ can be eliminated.

Definition 5.1.8 (Types and terms of λP).

1. The set of pseudo-expressions of λP, notation, $\mathcal{T}$ is defined as follows

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

   where V is the collection of variables and C that of constants. No distinction between type- and term-variables is made.

2. Among the constants C two elements are called $*$ and $\Box$.

Definition 5.1.9 (Assignment 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 statements).

The notion $\vdash$ is defined by the following axiom and rules. Again the letter s ranges over $\{*, \Box\}$.

λP

(axiom)	$<> \vdash * : \Box;$
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}, \ x \notin \Gamma;$
(weakening rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}, \ x \notin \Gamma;$
(type/kind formation)	$\dfrac{\Gamma \vdash A : * \quad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash (\Pi x{:}A.B) : s}.$
(application rule)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]};$
(abstraction rule)	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)};$
(conversion rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}.$

Typical assignments in λP are the following:

$$\begin{array}{rcl}
A{:}* & \vdash & (A{\rightarrow}*) : \Box; \\
A{:}*, P{:}A{\rightarrow}*, a{:}A & \vdash & Pa : *; \\
A{:}*, P{:}A{\rightarrow}*, a{:}A & \vdash & Pa{\rightarrow}* : \Box; \\
A{:}*, P{:}A{\rightarrow}* & \vdash & (\Pi a{:}A.Pa{\rightarrow}*) : \Box; \\
A{:}*, P{:}A{\rightarrow}* & \vdash & (\lambda a{:}A\lambda x{:}Pa.x) : (\Pi a{:}A.(Pa{\rightarrow}Pa))
\end{array}$$

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

$$\text{prop}{:}*, \text{Imp}{:}\text{prop}{\rightarrow}\text{prop}{\rightarrow}\text{prop}.$$

Write $\varphi \supset \psi$ for $\text{Imp}\,\varphi\psi$. In order to express that a proposition is valid a variable $\text{T} : \text{prop}{\rightarrow}*$ is declared and $\varphi : \text{prop}$ is defined to be valid if

$\mathrm{T}\varphi$ is inhabited, i.e. $M : \mathrm{T}\varphi$ for some M. Now in order to express that implication has the right properties, one assumes $\supset_e$ and $\supset_i$ such that

$$\supset_e \varphi\psi : \mathrm{T}(\varphi \supset \psi) \rightarrow \mathrm{T}\varphi \rightarrow \mathrm{T}\psi.$$

$$\supset_i \varphi\psi : (\mathrm{T}\varphi \rightarrow \mathrm{T}\psi) \rightarrow \mathrm{T}(\varphi \supset \psi).$$

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

$$\begin{array}{rcl} \mathrm{T} & : & \mathrm{prop} \rightarrow * \\ \supset_e & : & \Pi\varphi{:}\mathrm{prop}\Pi\psi{:}\mathrm{prop}.\mathrm{T}(\varphi \supset \psi) \rightarrow \mathrm{T}\varphi \rightarrow \mathrm{T}\psi \\ \supset_i & : & \Pi\varphi{:}\mathrm{prop}\Pi\psi{:}\mathrm{prop}.(\mathrm{T}\varphi \rightarrow \mathrm{T}\psi) \rightarrow \mathrm{T}(\varphi \supset \psi). \end{array}$$

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

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

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

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

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

and for $\supset_i \varphi\psi$

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

In this way the $\{\rightarrow, \forall\}$ fragment of (manysorted constructive) predicate logic can be interpreted too. A predicate P on a set (type) A can be represented as a $P{:}(A \rightarrow *)$ 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 like

$$[\forall x \in A \forall y \in A.Pxy] \rightarrow [\forall x \in A.Pxx]$$

can be seen to be valid because its translation is inhabited

$$\begin{array}{rcl} A{:}*, P{:}A \rightarrow A \rightarrow * & \vdash & (\lambda z{:}(\Pi x{:}A\Pi y{:}A.Pxy)\lambda x{:}A.zxx) : \\ & & ([\Pi x{:}A\Pi y{:}A.Pxy] \rightarrow [\Pi x{:}A.Pxx]). \end{array}$$

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 coworkers 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 and Huet (1988) (calculus of constructions). The project LF uses the interpretation of formulas using T:(prop$\rightarrow$$*$) like the original use in AUTOMATH. In Martin-Löf (1984) the proposition-as-types paradigm is used for formulating results in the foundation of mathematics.

The λ-cube

We will now introduce a cube of eight systems of typed lambda calculi. This so called 'λ-cube' forms a natural framework in which several known systems *à la* Church, including $\lambda\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$ and λP are given in a uniform way. It provides a finestructure of the calculus of constructions, which is the strongest system in the cube. The differentiation between the systems is obtained by controlling the way in which abstractions are allowed.

The systems $\lambda\rightarrow$ and $\lambda 2$ in the λ-cube are not given in their original version, but in a equivalent variant. Also for some of the other known systems the versions on the cube are only in essence equivalent to the original ones. The point is that there are some choices for the precise formulation of the systems and in the cube these choices are made uniformly.

Definition 5.1.10 (Systems of the λ-cube).

1. The systems of the λ-cube are based on a set of pseudo-expressions $\mathcal{T}$ defined by the following abstract syntax.

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

   where V and C are infinite collections of variables and constants respectively. No distinction between type- and term-variables is made.

2. On $\mathcal{T}$ the notions of β-conversion and β-reduction are defined by the following contraction rule:

$$(\lambda x{:}A.B)C \rightarrow B[x := C].$$

3. A *statement* is of the form $A : B$ with $A, B \in \mathcal{T}$. A is the *subject* and B is the *predicate* of $A : B$. A *declaration* is of the form $x{:}A$ with $A{\in}\mathcal{T}$ and x a variable. A *pseudo-context* is a finite ordered sequence of declarations, all with distinct subjects. The empty context is denoted by $<>$. If $\Gamma =< x_1{:}A_1, \ldots, x_n{:}A_n >$, then

$$\Gamma, x{:}B =< x_1{:}A_1, \ldots, x_n{:}A_n, x{:}B > .$$

   Usually we do not write the $<>$.

4. The rules of type assignment will axiomatize the notion

$$\Gamma \vdash A : B$$

stating that $A : B$ can be derived from the pseudo-context Γ; in that case A and B are called (legal) expressions and Γ is a (legal) context.

The rules are given in two groups:

(a) the general axiom and rules, valid for all systems of the λ-cube;

(b) the specific rules, differentiating between the eight systems; these are parametrized Π-introduction rules.

Two constants are selected and are given the names $*$ and $\square$. These two constants are called *sorts*. Let $\mathcal{S} = \{*, \square\}$ and s, s_1, s_2 range over $\mathcal{S}$.

Systems in the λ-cube

1. General axiom and rules.

(axiom)	$<> \vdash * : \square;$
(start rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}, \; x \notin \Gamma;$
(weakening rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}, \; x \notin \Gamma;$
(application rule)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]};$
(abstraction rule)	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)};$
(conversion rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}.$

2. The specific rules

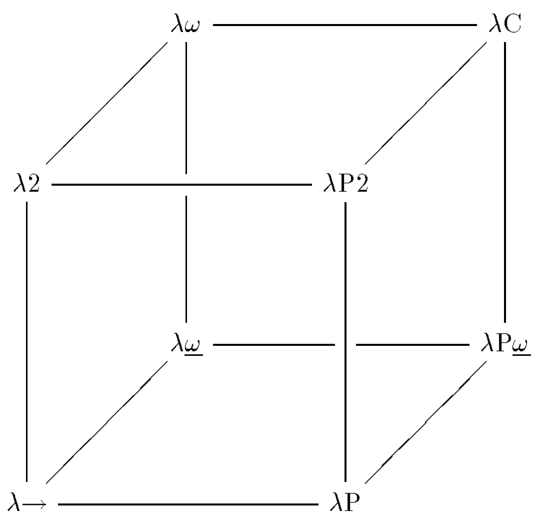
(s_1, s_2) rule	$\dfrac{\Gamma \vdash A : s_1, \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_2}.$

We use $A, B, C, a, b, \ldots$ for abitrary pseudo-terms and $x, y, z, \ldots$ for arbitrary variables.

5. *The eight systems of the* λ-cube are defined by taking the general rules plus a specific subset of the set of rules $\{(*,*), (*,\Box), (\Box,*), (\Box,\Box)\}$.

System	Set of specific rules			
$\lambda\to$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
λP	$(*,*)$		$(*,\Box)$	
λP2	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
λP$\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
λPω=λC	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

The λ-cube will usually be drawn in the *standard orientation* displayed as follows; the inclusion relations are often left implicit.



Remark 5.1.11. Most of the systems in the λ-cube appear elsewhere in the literature, often in some variant form.

System	*related system(s)*	*names and references*
$\lambda\!\rightarrow$	λ^{τ}	simply typed lambda calculus; Church (1940), Barendregt (1984), Appendix A, Hindley and Seldin (1986), Ch 14.
$\lambda 2$	F	scond order (typed) lambda calculus; Girard (1972), Reynolds (1974).
λP	AUT-QE; LF	de Bruijn (1970); Harper *et al.* (1987).
λP2		Longo and Moggi (1988).
$\lambda\underline{\omega}$	POLYREC	Renardel de Lavalette (1991).
$\lambda\omega$	Fω	Girard (1972).
λPω = λC	CC	calculus of constructions; Coquand and Huet (1988).

Remarks 5.1.12.

1. The expression $(\Pi\alpha{:}{*}.(\alpha{\rightarrow}\alpha))$ in $\lambda 2$ being a cartesian product of types will also be a type, so $(\Pi\alpha{:}{*}.(\alpha{\rightarrow}\alpha)) : *$. But since it is a product over all possible types α, including the one *in statu nascendi* (i.e. $(\Pi\alpha{:}{*}.(\alpha{\rightarrow}\alpha))$ itself is among the types in $*$), there is an essential impredicativity here.

2. Note that in $\lambda\!\rightarrow$ one has also in some sense terms depending on types and types depending on types:

   $\lambda x{:}A.x$ is a term depending on the type A,
   $A{\rightarrow}A$ is a type depending on the type A.

   But in $\lambda\!\rightarrow$ one has no function abstraction for these dependencies. Note also that in $\lambda\!\rightarrow$ (and even in $\lambda 2$ and $\lambda\underline{\omega}$) one has no types depending on terms. The types are given beforehand. The right-hand side of the cube is essentially more difficult then the left-hand side because of the mixture of types and terms.

The two versions of $\lambda\!\rightarrow$ and $\lambda 2$

Now we have given the definition of the λ-cube, we want to explain why $\lambda\!\rightarrow$ and $\lambda 2$ in the cube are essentially the same as the systems with the same name defined in 5.1.1 and 5.1.3 respectively.

Definition 5.1.13. In the systems of the λ-cube we use the following notation:

$$A{\to}B \equiv \Pi x{:}A.B, \text{ where } x \text{ is fresh (not in } A, B).$$

Lemma 5.1.14. *Consider $\lambda{\to}$ in the λ-cube. If $\Gamma \vdash A : *$ in this system, then A is built up from the set $\{B \mid (B : *) \in \Gamma\}$ using only $\to$ (as defined in 5.1.13).*

Proof. By induction on the generation of $\vdash$. ■ ■

Notice that the application rule implies the $\to$-elimination rule:

$$\frac{\Gamma \vdash F : (A{\to}B)(\equiv \Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash (Fa) : B[x := a] \equiv B},$$

since x does not occur in B. It follows that if e.g. in $\lambda{\to}$ in the λ-cube one derives

$$A{:}*, B{:}*, a{:}A, b{:}B \vdash M : C : *$$

then

$$a{:}A, b{:}B \vdash M : C$$

is derivable in the system $\lambda{\to}$ as defined in 5.1.1.

Similarly one shows that both variants of $\lambda 2$ are the same by first defining in the λ-cube

$$\forall\alpha.A \equiv \Pi\alpha{:}*.A,$$

$$\Lambda\alpha.M \equiv \lambda\alpha{:}*.M.$$

Of course the use of the greek letter α is only suggestive; after all, it is a bound variable and its name is irrelevant.

Some derivable type assignments in the λ-cube

We end this subsection by giving some examples of type assignment for the systems in the λ-cube. The examples for $\lambda{\to}$ and $\lambda 2$ given before are essentially repeated in the new style of the systems.

The reader is invited to carefully study these examples in order to gain some intuition in the systems of the λ-cube. Some of the examples are followed by a comment {in curly brackets}. In order to understand the intended meaning for the systems on the right plane in the λ-cube (i.e. the rule pair $(*, \Box)$ is present), some of the elements of $*$ have to be considered as sets and some as propositions. The examples show that the systems in the λ-cube are related to logical systems and form a preview of the propositions-as-type interpretation described in subsection 5.4. Names of variables are chosen freely as either Roman or Greek letters, in order to follow the intended interpretation. The notation $\Gamma \vdash A : B : C$ stands for the conjunction of $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$.

Examples 5.1.15.

1. In $\lambda\!\to$ the following can be derived:

$$\begin{array}{rcl}
A{:}* & \vdash & (\Pi x{:}A.A) : *; \\
A{:}* & \vdash & (\lambda a{:}A.a) : (\Pi x{:}A.A); \\
A{:}*, B{:}*, b{:}B & \vdash & (\lambda a{:}A.b) : (A{\to}B), \\
 & & \text{where } (A{\to}B) \equiv (\Pi x{:}A.B); \\
A{:}*, b{:}A & \vdash & ((\lambda a{:}A.a)b) : A; \\
A{:}*, B{:}*, c{:}A, b{:}B & \vdash & ((\lambda a{:}A.b)c) : B; \\
A{:}*, B{:}* & \vdash & (\lambda a{:}A\lambda b{:}B.a) : (A{\to}(B{\to}A)) : *.
\end{array}$$

2. In $\lambda 2$ the following can be derived:

$$\begin{array}{rcl}
\alpha{:}* & \vdash & (\lambda a{:}\alpha.a) : (\alpha{\to}\alpha); \\
 & \vdash & (\lambda\alpha{:}*\lambda a{:}\alpha.a) : (\Pi\alpha{:}*.(\alpha{\to}\alpha)) : *; \\
A{:}* & \vdash & (\lambda\alpha{:}*\lambda a{:}\alpha.a)A : (A{\to}A); \\
A{:}*, b{:}A & \vdash & (\lambda\alpha{:}*\lambda a{:}\alpha.a)Ab : A;
\end{array}$$

   of course the following reduction holds:

$$\begin{array}{rcl}
(\lambda\alpha{:}*\lambda a{:}\alpha.a)Ab & \to & (\lambda a{:}A.a)b \\
 & \to & b.
\end{array}$$

   The following two examples show a connection with second-order proposition logic.

$$\vdash (\lambda\beta{:}*\lambda a{:}(\Pi\alpha{:}*.\alpha).a((\Pi\alpha{:}*.\alpha){\to}\beta)a) : (\Pi\beta{:}*.(\Pi\alpha{:}*.\alpha){\to}\beta).$$

   {For this last example one has to think twice to see that it is correct; a simpler term of the same type is the following; write $\bot \equiv (\Pi\alpha{:}*.\alpha)$, which is the second-order definition of falsum.}

$$\vdash (\lambda\beta{:}*\lambda a{:}\bot.a\beta) : (\Pi\beta{:}*.\bot{\to}\beta).$$

   {The type considered as proposition says: *ex falso sequitur quodlibet*, i.e. anyting follows from a false statement; the term in this type is its proof.}

3. In $\lambda\underline{\omega}$ the following can be derived

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

$\{(\lambda\alpha{:}{*}.\alpha{\to}\alpha)$ is a constructor mapping types into types$\}$;

$$\beta{:}{*} \vdash (\lambda\alpha{:}{*}.\alpha{\to}\alpha)\beta : {*};$$
$$\beta{:}{*}, x{:}\beta \vdash (\lambda y{:}\beta.x) : (\lambda\alpha{:}{*}.\alpha{\to}\alpha)\beta$$

$\{$note that $(\lambda y{:}\beta.x)$ has type $\beta{\to}\beta$ in the given context$\}$;

$$\alpha{:}{*}, f{:}{*}{\to}{*} \vdash f(f\alpha) : {*};$$
$$\alpha{:}{*} \vdash (\lambda f{:}{*}{\to}{*}.f(f\alpha)) : ({*}{\to}{*}){\to}{*}$$

$\{$in this way higher-order constructors are formed$\}$.

4. In λP the following can be derived:

$$A{:}{*} \vdash (A{\to}{*}) : \Box$$

$\{$if A is a type considered as set, then $A{\to}{*}$ is the kind of predicates on $A\}$;

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

$\{$if A is a set, $a \in A$ and P is a predicate on A, then Pa is a type considered as proposition (true if inhabited; false otherwise)$\}$;

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

$\{$if P is a binary predicate on the set A, then $\forall a \in A\ Paa$ is a proposition$\}$;

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

$\{$this proposition states that the predicate P considered as a set is included in the predicate $Q\}$;

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

$\{$this proposition states the reflexivity of inclusion$\}$;

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

$\{$the subject in this assignment provides the 'proof' of reflexivity of inclusion$\}$;

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

$$A{:}{*}, P{:}A{\to}{*}, Q{:}{*}, a_0{:}A \vdash (\lambda x{:}(\Pi a{:}A.Pa{\to}Q)\lambda y{:}(\Pi a{:}A.Pa).xa_o(ya_o)) :$$

$$(\Pi x{:}(\Pi a{:}A.Pa\to Q)\Pi y{:}(\Pi a{:}A.Pa).Q) \equiv$$
$$(\Pi a{:}A.Pa\to Q)\to(\Pi a{:}A.Pa)\to Q$$

{this proposition states that the proposition

$$(\forall a\in A.Pa\to Q)\to(\forall a\in A.Pa)\to Q$$

is true in non-empty structures A; notice that the lay out explains the functioning of the λ–rule; in this type assignment the subject is the 'proof' of the previous true proposition; note that in the context the assumption $a_0{:}A$ is needed in this proof.}

5. In $\lambda\omega$ the following can be derived.
Let $\alpha\&\beta \equiv \Pi\gamma{:}{*}.(\alpha\to\beta\to\gamma)\to\gamma$, then

$$\alpha{:}{*}, \beta{:}{*} \vdash \alpha\&\beta : *$$

{this is the 'second-order definition of &' and is definable already in $\lambda 2$}.

Let AND $\equiv \lambda\alpha{:}{*}\lambda\beta{:}{*}.\alpha\&\beta$ and $K \equiv \lambda\alpha{:}{*}\lambda\beta{:}{*}\lambda x{:}\alpha\lambda y{:}\beta.x$, then

$$\vdash \text{ AND} : (*\to*\to*),$$
$$\vdash K : (\Pi\alpha{:}{*}\Pi\beta{:}{*}.\alpha\to\beta\to\alpha).$$

{Note that $\alpha\&\beta$ and K can be derived already in $\lambda 2$, but the term AND cannot}.

$$\alpha{:}{*}, \beta{:}{*} \vdash (\lambda x{:}\text{AND}\alpha\beta.x\alpha(K\alpha\beta)) : (\text{ AND}\alpha\beta\to\alpha) : *$$

{the subject is a proof that AND$\alpha\beta\to\alpha$ is a tautology}.

6. In λP2 {corresponding to second-order predicate logic} the following can be derived.

$$\begin{array}{rcl} A{:}{*}, P{:}A\to* & \vdash & (\lambda a{:}A.Pa\to\bot) : (A\to*) \\ A{:}{*}, P{:}A\to A\to* & \vdash & [(\Pi a{:}A\Pi b{:}A.Pab\to Pba\to\bot) \\ & & \to(\Pi a{:}A.Paa\to\bot)] : * \end{array}$$

{the proposition states that a binary relation that is asymmetric is irreflexive}

7. In $\lambda\mathrm{P}\underline{\omega}$ the following can be derived.

$$A{:}{*} \vdash (\lambda P{:}A\to A\to*\lambda a{:}A.Paa) : ((A\to A\to*)\to(A\to*)) : \square$$

{this constructor assigns to a binary predicate P on A its 'diagonalization'};

$$\vdash (\lambda A{:}{*}\lambda P{:}A{\to}A{\to}{*}\lambda a{:}A.Paa) : (\Pi A{:}{*}\Pi P{:}A{\to}A{\to}{*}\Pi a{:}A.{*}) : \Box$$

{the same is done uniformly in A}.

8. In λPω = λC the following can be derived.

$$\vdash (\lambda A{:}{*}\lambda P{:}A{\to}{*}\lambda a{:}A.Pa{\to}\bot) : (\Pi A{:}{*}.(A{\to}{*}){\to}(A{\to}{*})) : \Box$$

{this constructor assigns to a type A and to a predicate P on A the negation of P}.

Let ALL $\equiv (\lambda A{:}{*}\lambda P{:}A{\to}{*}.\Pi a{:}A.Pa)$; then

$$A{:}{*}, P{:}A{\to}{*} \vdash \ \mathrm{ALL}\,AP : {*} \text{ and } (\mathrm{ALL}\,AP) =_\beta (\Pi a{:}A.Pa)$$

{universal quantification done uniformly}.

Exercise 5.1.16.

1. Define $\neg \equiv \lambda\alpha{:}{*}.\alpha{\to}\bot$. Construct a term M such that in $\lambda\omega$

$$\alpha : {*}, \beta : {*} \vdash M : ((\alpha{\to}\beta){\to}(\neg\beta{\to}\neg\alpha)).$$

2. Find an expression M such that in λP2

$$A{:}{*},\ P{:}(A{\to}A{\to}{*}) \vdash$$

$$M : [(\Pi a{:}A\Pi b{:}A.Pab{\to}Pba{\to}\bot){\to}(\Pi a{:}A.Paa{\to}\bot)] : {*}.$$

3. Find a term M such that in λC

$$A{:}{*},\ P{:}A{\to}{*}, a{:}A \vdash M : (\mathrm{ALL}\,AP{\to}Pa).$$

5.2 Pure type systems

The method of generating the systems in the λ-cube has been generalized independently by Berardi (1989) and Terlouw (1989). This resulted in the notion of *pure type system* (PTS). Many systems of typed lambda calculus *à la* Church can be seen as PTSs. Subtle differences between systems can be described neatly using the notation for PTSs.

One of the successes of the notion of PTS's is concerned with logic. In subsection 5.4 a cube of eight logical systems will be introduced that is in a close correspondence with the systems on the λ-cube. This result is the so called 'propositions-as-types' interpretation. It was observed by Berardi (1989) that the eight logical systems can each be described as a PTS in such a way that the propositions-as-types interpretation obtains a canonical simple form.

Another reason for introducing PTSs is that several propositions about the systems in the λ-cube are needed. The general setting of the PTSs makes it nicer to give the required proofs. Most results in this subsection are taken form Geuvers and Nederhof (1991) and also serve as a preparation for the strong normalization proof in Section 5.3.

The pure type systems are based on the set of pseudo-terms $\mathcal{T}$ for the λ-cube. We repeat the abstract syntax for $\mathcal{T}$.

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

Definition 5.2.1. The *specification of a* PTS consists of a triple S = (S, A, R) where

1. S is a subset of C, called the *sorts*;
2. A is a set of *axioms* of the form

$$c : s$$

   with $c \in \mathcal{C}$ and $s \in \mathcal{S}$;
3. $\mathcal{R}$ is a set of rules of the form

$$(s_1, s_2, s_3)$$

   with $s_1, s_2, s_3 \in \mathcal{S}$.

It is useful to divide the set V of variables into disjoint infinite subsets V_s for each sort $s \in \mathcal{S}$. So $V = \cup\{V_s \mid s \in \mathcal{S}\}$. The members of V_s are denoted by ${}^s x, {}^s y, {}^s z, \ldots$. Arbitrary variables are often still denoted by $x, y, z, \ldots$; however if necessary one writes $x \equiv {}^s x$ to indicate that $x \in V_s$. The first version of $\lambda 2$ introduced in 5.1.3 can be understood as $x, y, z, \ldots$ ranging over V_* and $\alpha, \beta, \gamma, \ldots$ over $V_\Box$. For reasons of hygiene it will be useful to assume that if ${}^{s_1}x_1$ and ${}^{s_2}x_2$ occur both in a pseudo-term M, then

$$s_1 \not\equiv s_2 \;\Rightarrow\; x_1 \equiv x_2.$$

If this is not the case, then a simple renaming can establish this.

Definition 5.2.2. The PTS determined by the specification S = $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, notation λS=$\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, is defined as follows. Statements and contexts are defined as for the λ-cube. The notion of type derivation $\Gamma \vdash_{\lambda S} A : B$ (we just write $\Gamma \vdash A : B$) is defined by the following axioms and rules:

$$\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$$

(axioms)	$<>\vdash c : s,$	if $(c : s) \in \mathcal{A}$;
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A},$	if $x \equiv {}^s x \notin \Gamma$;
(weakening)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B},$	if $x \equiv {}^s x \notin \Gamma$;
(product)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3},$	if $(s_1, s_2, s_3) \in \mathcal{R}$;
(application)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]};$	
(abstraction)	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)};$	
(conversion)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}.$	

In the above we use the following conventions.
s ranges over $\mathcal{S}$, the set of sorts;
x ranges over variables.
The proviso in the conversion rule ($B =_\beta B'$) is a priori not decidable. However it can be replaced by the decidable condition

$$B' \twoheadrightarrow_\beta B \text{ or } B \twoheadrightarrow_\beta B'$$

without changing the set of derivable statements.

Definition 5.2.3.

1. The rule (s_1, s_2) is an abbreviation for (s_1, s_2, s_2). In the λ-cube only systems with rules of this simpler form are used.
2. The PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is called *full* if

$$\mathcal{R} = \{(s_1, s_2) \mid s_1, s_2 \in \mathcal{S}\}.$$

Examples 5.2.4.

1. $\lambda 2$ is the PTS determined by:

$$\begin{array}{rcl} \mathcal{S} & = & \{*, \Box\} \\ \mathcal{A} & = & \{* : \Box\} \\ \mathcal{R} & = & \{(*, *), (\Box, *)\}. \end{array}$$

   Specifications like this will be given more stylistically as follows.

$$\lambda 2 \; \begin{array}{|ll|} \hline \mathcal{S} & *, \Box \\ \mathcal{A} & * : \Box \\ \mathcal{R} & (*, *), (\Box, *) \\ \hline \end{array}$$

2. λC is the full PTS with

$$\lambda\mathrm{C} \; \begin{array}{|ll|} \hline \mathcal{S} & *, \Box \\ \mathcal{A} & * : \Box \\ \mathcal{R} & (*, *), (\Box, *), (*, \Box), (\Box, \Box) \\ \hline \end{array}$$

3. A variant λC$'$ of λC is the full PTS with

$$\lambda\mathrm{C}' \; \begin{array}{|ll|} \hline \mathcal{S} & *^t, *^p, \Box \\ \mathcal{A} & *^t : \Box, *^p : \Box \\ \mathcal{R} & \mathcal{S}^2, \text{ i.e. all pairs} \\ \hline \end{array}$$

4. $\lambda\!\rightarrow$ is the PTS determined by

$$\lambda\!\rightarrow \; \begin{array}{|ll|} \hline \mathcal{S} & *, \Box \\ \mathcal{A} & * : \Box \\ \mathcal{R} & (*, *) \\ \hline \end{array}$$

5. A variant of $\lambda\!\rightarrow$, called λ^τ in Barendregt (1984) Appendix A, is the PTS determined by

$$\lambda^\tau \; \begin{array}{|ll|} \hline \mathcal{S} & * \\ \mathcal{A} & 0 : * \\ \mathcal{R} & (*, *) \\ \hline \end{array}$$

The difference with $\lambda\!\rightarrow$ is that in λ^τ no type variables are possible but only has constant types like $0, 0{\rightarrow}0, 0{\rightarrow}0{\rightarrow}0, \ldots$.

6. The system $\lambda*$ in which $*$ is the sort of all types, including itself, is specified by

$\lambda*$		
	$\mathcal{S}$	$*$
	$\mathcal{A}$	$* : *$
	$\mathcal{R}$	$(*, *)$

In subsection 5.5 it will be shown that the system $\lambda*$ is 'inconsistent', in the sense that all types are inhabited. This result is known as Girard's paradox. One may think that the result is caused by the circularity in $* : *$, however Girard (1972) showed that also the following system is inconsistent in the same sense, see Section 5.5.

λU		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*, *), (\Box, *), (\Box, \Box), (\Delta, \Box), (\Delta, *)$

7. (Geuvers (1990)). The system of higher-order logic in Church (1940) can be described by the following PTS; see Ssection 5.4 for its use.

λHOL		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*, *), (\Box, *), (\Box, \Box)$

8. (van Benthem Jutting (1990)). So far none of the rules has been of the form (s_1, s_2, s_3). Several members of the AUTOMATH family, see van Daalen (1980) and de Bruijn (1980), can be described as PTSs with such rules. The sort Δ serves as a 'parking place' for certain terms.

λAUT-68		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box$
	$\mathcal{R}$	$(*, *), (*, \Box, \Delta), (\Box, *, \Delta)$
		$(\Box, \Box, \Delta), (*, \Delta, \Delta), (\Box, \Delta, \Delta)$

This system is a strengthening of $\lambda\!\rightarrow$ in which there are more powerful contexts.

λAUT-QE		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box$
	$\mathcal{R}$	$(*, *), (*, \Box), (\Box, *, \Delta)$
		$(\Box, \Box, \Delta), (*, \Delta, \Delta), (\Box, \Delta, \Delta)$

This system corresponds to λP.

λPAL	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box$
	$\mathcal{R}$	$(*, *, \Delta), (*, \Box, \Delta), (\Box, *, \Delta)$
		$(\Box, \Box, \Delta), (*, \Delta, \Delta), (\Box, \Delta, \Delta)$

This system is a subsystem of $\lambda\!\rightarrow$. An interesting conjecture of de Bruijn states that mathematics from before the year 1800 can all be formalized in λPAL.

In subsection 5.4 we will encounter rules of the form (s_1, s_2, s_3) in order to represent first-order but not higher-order functions.

Properties of arbitrary PTS*s*

Now we will state and prove some elementary properties of PTSs. In 5.2.5 - 5.2.17 the notions of context, derivability etc. refer to $\lambda S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, an arbitrary PTS. The results are taken from Geuvers and Nederhof (1991).

Notation 5.2.5.

1. $\Gamma \vdash A : B : C$ means $\Gamma \vdash A : B \& \Gamma \vdash B : C$.
2. Let $\Delta \equiv u_1{:}B_1, \ldots, u_n{:}B_n$ with $n \geq 0$ be a pseudocontext. Then $\Gamma \vdash \Delta$ means $\Gamma \vdash u_1{:}B_1$ & ... & $\Gamma \vdash u_n{:}B_n$.

Definition 5.2.6. Let Γ be a pseudocontext and A be a pseudoterm.

1. Γ is called *legal* if $\exists P, Q \in \mathcal{T}\, \Gamma \vdash P : Q$.
2. A is called a Γ-*term* if $\exists B \in \mathcal{T}[\Gamma \vdash A : B$ or $\Gamma \vdash B : A]$.
3. A is called a Γ-*type* if $\exists s \in \mathcal{S}[\Gamma \vdash A : s]$.
4. If $\Gamma \vdash A : s$, then A is called a Γ-*type of sort* s.
5. A is called a Γ-*element* if $\exists B \in \mathcal{T}\, \exists s \in \mathcal{S}[\Gamma \vdash A : B : s]$.
6. If $\Gamma \vdash A : B : s$ then A is called a Γ-*element of type* B *and of sort* s.
7. $A \in \mathcal{T}$ is called *legal* if $\exists \Gamma, B\ [\Gamma \vdash A : B$ or $\Gamma \vdash B : A]$.

Definition 5.2.7. Let $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ and $\Delta \equiv y_1{:}B_1, \ldots, y_m{:}B_m$ be pseudo-contexts.

1. A statement $x{:}A$ *is in* Γ, notation $(x{:}A) \in \Gamma$, if $x \equiv x_i$ and $A \equiv A_i$ for some i.
2. Γ is *part* of Δ, notation $\Gamma \subseteq \Delta$, if every $x{:}A$ in Γ is also in Δ.
3. Let $1 \leq i \leq n+1$. Then the *restriction* of Γ to i, notation $\Gamma \upharpoonright i$, is $x_1{:}A_1, \ldots, x_{i-1}{:}A_{i-1}$.
4. Γ is an *initial segment* of Δ, notation $\Gamma \leq \Delta$, if for some $j \leq m+1$ one has $\Gamma \equiv \Delta \upharpoonright j$.

Lemma 5.2.8 (Free variable lemma for PTS's).
Let $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ be a legal context, say $\Gamma \vdash B : C$. Then the following hold.

1. *The $x_1, \ldots, x_n$ are all distinct.*
2. *$FV(B), FV(C) \subseteq \{x_1, \ldots, x_n\}$.*
3. *$FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $1 \leq i \leq n$.*

Proof. (1), (2), (3). By induction on the derivation of $\Gamma \vdash B : C$. ■ ■

The following lemmas show that legal contexts behave as expected.

Lemma 5.2.9 (Start lemma for PTS's). *Let Γ be a legal context. Then*

1. *$(c : s)$ is an axiom $\Rightarrow \quad \Gamma \vdash c : s$;*
2. *$(x{:}A) \in \Gamma \quad \Rightarrow \quad \Gamma \vdash x : A$.*

Proof. (1), (2). By assumption $\Gamma \vdash B : C$ for some B and C. The result follows by induction on the derivation of $\Gamma \vdash B : C$. ■ ■

Lemma 5.2.10 (Transitivity lemma for PTS's). *Let Γ and Δ be contexts of which Γ is legal. Then*

$$[\Gamma \vdash \Delta \;\&\; \Delta \vdash A : B] \Rightarrow \Gamma \vdash A : B.$$

Proof. By induction on the derivation of $\Delta \vdash A : B$.
We treat two cases:

Case 1. $\Delta \vdash A : B$ is $<> \vdash c : s$ with $c : s$ an axiom. Then by the start lemma 5.2.9 (1) we have $\Gamma \vdash c : s$, since Γ is legal. (Note that trivially $\Gamma \vdash <>$, so one needs to postulate that Γ is legal.)

Case 2. $\Delta \vdash A : B$ is $\Delta \vdash (\Pi x{:}A_1.A_2) : s_3$ and is a direct consequence of $\Delta \vdash A_1 : s_1$ and $\Delta, x{:}A_1 \vdash A_2 : s_2$ for some $(s_1, s_2, s_3) \in \mathcal{R}$. It may be assumed that x does not occur in Γ. Write $\Gamma^+ \equiv \Gamma, x{:}A_1$. Then by the induction hypothesis $\Gamma \vdash A_1 : s_1$, so $\Gamma^+ \vdash \Delta, x{:}A_1$. Hence

$$\Gamma, x{:}A_1 \vdash A_2 : s_2$$

and hence by the product rule

$$\Gamma \vdash (\Pi x_1{:}A_1.A_2){:}s_3$$

i.e. $\Gamma \vdash A : B$. ■

■

Lemma 5.2.11 (Substitution lemma for PTS's). *Assume*

$$\Gamma, x{:}A, \Delta \vdash B : C \tag{1}$$

and

$$\Gamma \vdash D : A. \tag{2}$$

Then

$$\Gamma, \Delta[x := D] \vdash B[x := D] : C[x := D].$$

Proof. By induction on the derivation of (1). We treat two cases. Write M^* for $M[x := D]$.

Case 1. The last rule used to obtain (1) is the start rule.

Subcase 1.1. $\Delta = <>$. Then the last step in the derivation of (1) is

$$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A},$$

so in this subcase $(B : C) \equiv (x : A)$. We have to show

$$\Gamma \vdash (x : A)^* \equiv (D : A)$$

which holds by assumption (2).

Subcase 1.2. $\Delta = \Delta_1, y{:}E$ and the last step in the derivation of (1) is

$$\frac{\Gamma, x{:}A, \Delta_1 \vdash E : s}{\Gamma, x{:}A, \Delta_1, y{:}E \vdash y : E}.$$

We have to show

$$\Gamma, \Delta_1^*, y{:}E^* \vdash y : E^*,$$

but this follows directly from the induction hypothesis $\Gamma, \Delta_1^* \vdash E^* : s$.

Case 2. The last applied rule to obtain (1) is the application rule, i.e.

$$\frac{\Gamma, x{:}A, \Delta, \vdash B_1 : (\Pi y{:}C_1.C_2) \quad \Gamma, x{:}A, \Delta \vdash B_2 : C_1}{\Gamma, x{:}A, \Delta \vdash (B_1 B_2) : C_2[y := B_2]}.$$

By the induction hypothesis one has

$$\Gamma, \Delta^* \vdash B_1^* : (\Pi y{:}C_1^*.C_2^*) \text{ and } \Gamma, \Delta^* \vdash B_2^* : C_1^*$$

and hence

$$\Gamma, \Delta^* \vdash (B_1^* B_2^*) : (C_2^*[y := B_2^*])$$

so by the substitution lemma for terms, 2.1.6, one has

$$\Gamma, \Delta^* \vdash (B_1 B_2)^* : (C_2[y := B_2])^*. \blacksquare$$

∎

Lemma 5.2.12 (Thinning lemma for PTS's). *Let Γ and Δ be legal contexts such that $\Gamma \subseteq \Delta$. Then*

$$\Gamma \vdash A : B \Rightarrow \Delta \vdash A : B.$$

Proof. By induction on the length of derivation of $\Gamma \vdash A : B$. We treat two cases.
Case 1. $\Gamma \vdash A : B$ is the axiom $<> \vdash c : s$. Then by the start lemma 5.2.9 one has $\Delta \vdash c : s$.
Case 2. $\Gamma \vdash A : B$ is an $\Gamma \vdash (\Pi x{:}A_1.A_2) : s_3$ and follows from $\Gamma \vdash A_1 : s_1$ and $\Gamma, x{:}A_1 \vdash A_2 : s_2$. By the IH one has $\Delta \vdash A_1 : s_1$ and since it may be assumed that x does not occur in Δ it follows that $\Delta, x{:}A_1 \vdash x : A_1$,i.e. $\Delta, x{:}A_1$ is legal. But then again by the IH $\Delta, x{:}A_1 \vdash A_2 : s_2$ and hence $\Delta \vdash (\Pi x{:}A_1.A_2) : s_3$. $\blacksquare$

∎

The following result analyses how a type assignment $\Gamma \vdash A : B$ can be obtained, according to whether A is a variable, a constant, an application, a λ-abstraction or a Π-abstraction.

Lemma 5.2.13 (Generation lemma for PTS's).

1.	$\Gamma \vdash c : C$	$\Rightarrow$	$\exists s \in \mathcal{S}\ [C =_\beta s\ \&\ (c : s)$ *is an axiom*$]$.
2.	$\Gamma \vdash x : C$	$\Rightarrow$	$\exists s \in \mathcal{S} \exists B =_\beta C\ [\Gamma \vdash B : s\ \&\ (x{:}B) \in \Gamma$ $\&\ x \equiv {}^s x]$.
3.	$\Gamma \vdash (\Pi x{:}A.B) : C$	$\Rightarrow$	$\exists (s_1, s_2, s_3) \in \mathcal{R}\ [\Gamma \vdash A : s_1\ \&$ $\Gamma, x{:}A \vdash B : s_2\ \&\ C =_\beta s_3]$.
4.	$\Gamma \vdash (\lambda x{:}A.b) : C$	$\Rightarrow$	$\exists s \in \mathcal{S} \exists B\ [\Gamma \vdash (\Pi x{:}A.B){:}s\ \&$ $\Gamma, x{:}A \vdash b : B\ \&\ C =_\beta (\Pi x{:}A.B)]$.
5.	$\Gamma \vdash (Fa) : C$	$\Rightarrow$	$\exists A, B\ [\Gamma \vdash F : (\Pi x{:}A.B)\ \&$ $\Gamma \vdash a : A\ \&\ C =_\beta B[x := a]]$.

Proof. Consider a derivation of $\Gamma \vdash A : C$ in one of the cases. The rules weakening and conversion do not change the term A. We can follow the branch of the derivation until the term A is introduced the first time. This can be done by

- an axiom for 1;
- the start rule for 2
- the product-rule for 3;
- the application rule for 4;
- the abstraction-rule for 5.

In each case the conclusion of the axiom or rule is $\Gamma' \vdash A : B'$ with $\Gamma' \subseteq \Gamma$ and $B' =_\beta B$. The statement of the lemma follows by inspection of the used axiom or rule and the thinning lemma 5.2.12 . ■ ∎

The following corollary states that every Γ-term is a sort, a Γ-type or a Γ-element. Note however that the classes of sorts, Γ-types and Γ-elements overlap. For example, in $\lambda\!\to$ with context $\Gamma \equiv \alpha : *$ one has that $\alpha\!\to\!\alpha$ is both a Γ-type and a Γ-element; indeed,

$$\Gamma \vdash (\lambda x{:}\alpha.x) : (\alpha\!\to\!\alpha) : * \text{ and } \Gamma \vdash (\alpha\!\to\!\alpha) : * : \Box.$$

Also it follows that subexpressions of legal terms are again legal. Subexpressions are defined as usual. (M sub A iff $M \in \mathrm{Sub}(A)$, where $\mathrm{Sub}(A)$, the set of subexpressions of A, is defined as follows.

$\mathrm{Sub}(A)$	$=$	$\{A\}$, if A is one of the constants (including the sorts) or variables;
	$=$	$\{A\} \cup \mathrm{Sub}(P) \cup \mathrm{Sub}(Q)$, if A is of the form $\Pi x{:}P.Q$, $\lambda x{:}P.Q$ or PQ.)

Corollary 5.2.14. *In every* PTS*one has the following.*

1. $\Gamma \vdash A : B \Rightarrow \exists s[B \equiv s$ *or* $\Gamma \vdash B : s]$
2. $\Gamma \vdash A : (\Pi x{:}B_1.B_2) \Rightarrow \exists s_1, s_2[\Gamma \vdash B_1 : s_1 \;\&\; \Gamma_1, x : B_1 \vdash B_2 : s_2]$.
3. *If* A *is a* Γ*-term, then* A *is a sort, a* Γ*-type or a* Γ*-element.*
4. *If* A *is legal and* B *sub* A*, then* B *is legal.*

Proof. 1. By induction on the derivation of $\Gamma \vdash A : B$.

2. By (1) and (4) of the generation lemma (notice that $(\Pi x{:}B_1.B_2) \not\equiv s$).
3. By (1), distinguishing the cases $\Gamma \vdash A : C$ and $\Gamma \vdash C : A$.
4. Let A be legal. By definition either $\Gamma \vdash A : C$ or $\Gamma \vdash C : A$, for some Γ and C. If the first case does not hold, then by (1) it follows that $A \equiv s$, hence $B \equiv A$ is legal. So suppose $\Gamma \vdash A : B$. It follows by induction on the structure of A, using the generation lemma, that any subterm of A is also legal. ■

■

Theorem 5.2.15 (Subject reduction theorem for PTS's).

$$\Gamma \vdash A : B \;\&\; A \twoheadrightarrow_\beta A' \quad \Rightarrow \quad \Gamma \vdash A' : B.$$

Proof. Write $\Gamma \rightarrow_\beta \Gamma'$ iff $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n, \Gamma' = x_1{:}A'_1, \ldots, x_n{:}A'_n$ and for some i one has $A_i \rightarrow A'_i$ and $A_j \equiv A'_j$ for $j \neq i$. Consider the statements

$$\Gamma \vdash A : B \;\&\; A \rightarrow_\beta A' \quad \Rightarrow \quad \Gamma \vdash A' : B; \qquad (i)$$

$$\Gamma \vdash A : B \;\&\; \Gamma \rightarrow_\beta \Gamma' \quad \Rightarrow \quad \Gamma' \vdash A : B. \qquad (ii)$$

These will be proved simultaneously by induction on the generation of $\Gamma \vdash A : B$. We treat two cases.

Case 1. The last applied rule is the product rule. Then $\Gamma \vdash A : B$ is $\Gamma \vdash (\Pi x{:}A_1.A_2) : s_3$ and is a direct consequence of $\Gamma \vdash A_1 : s_1$ and $\Gamma, x{:}A_1 \vdash A_2 : s_2$ for some rule (s_1, s_2, s_3). Then (i) and (ii) follow from the IH (for (i) and (ii), and (ii), respectively).

Case 2. The last applied rule is the application rule. Then $\Gamma \vdash A : B$ is $\Gamma \vdash A_1A_2 : B_2[x := A_2]$ and is a direct consequence of $\Gamma \vdash A_1 : (\Pi x{:}B_1.B_2)$ and $\Gamma \vdash A_2 : B_1$. The correctness of (ii) follows directly from the IH. As to (i), by Corollary 5.2.14 (1) it follows that for some sort s

$$\Gamma \vdash (\Pi x{:}B_1.B_2) : s,$$

hence by the generation lemma

$$\Gamma \vdash B_1 : s_1,$$

$$\Gamma, x{:}B_1 \vdash B_2 : s_2.$$

¿From this it follows with the substitution lemma that

$$\Gamma \vdash B_2[x := A_2] : s_2 \tag{1}$$

Subcase 2.1. $A' \equiv A_1'A_2'$ and $A_1 \rightarrow A_1'$ or $A_2 \rightarrow A_2'$. The IH and the application rule give

$$\Gamma \vdash A_1'A_2' : B_2[x := A_2']$$

Therefore by (1) and the conversion rule

$$\Gamma \vdash A_1'A_2' : B_2[x := A_2]$$

which is $\Gamma \vdash A' : B$.

Subcase 2.2. $A_1 \equiv \lambda x{:}A_{11}.A_{12}$ and $A' \equiv A_{12}[x := A_2]$. Then we have

$$\Gamma \vdash (\lambda x{:}A_{11}.A_{12}) : (\Pi x{:}B_1.B_2) \tag{2}$$

$$\Gamma \vdash A_2 : B_1. \tag{3}$$

By the generation lemma applied to (2) we get

$$\Gamma \vdash A_{11} : s_2 \tag{4}$$

$$\Gamma, x{:}A_{11} \vdash A_{12} : B_2' \tag{5}$$

$$\Gamma, x{:}A_{11} \vdash B_2' : s_2$$

$$\Pi x{:}B_1.B_2 = \Pi x{:}A_{11}.B_2' \tag{6}$$

for some B_2' and rule (s_1, s_2, s_3). From (6) and the Church–Rosser property, we obtain

$$B_1 = A_{11} \text{ and } B_2 = B_2' \tag{7}$$

By (3), (4) and (7) it follows from the conversion rule

$$\Gamma \vdash A_2 : A_{11},$$

hence by (5) and the substitution lemma

$$\Gamma \vdash (A_{12}[x := A_2]) : (B_2'[x := A_2]).$$

From this (1) and the conversion rule we finally obtain

$$\Gamma \vdash (A_{12}[x := A_2]) : (B_2[x := A_2])$$

which is $\Gamma \vdash A' : B$. ∎

∎

Corollary 5.2.16. *In every* PTS*one has the following.*

1. $[\Gamma \vdash A : B \;\&\; B \twoheadrightarrow_\beta B'] \;\;\Rightarrow\;\; \Gamma \vdash A : B'$.
2. *If A is a Γ-term and $A \twoheadrightarrow_\beta A'$, then A' is a Γ-term.*

Proof. 1. If $\Gamma \vdash A : B$, then by Corollary 5.2.14 (1) $B \equiv s$ or $\Gamma \vdash B : s$, for some sort s. In the first case also $B' \equiv s$ and we are done. In the second case one has, by the subject reduction theorem, 5.2.15, $\Gamma \vdash B' : s$ and hence by the conversion rule $\Gamma \vdash A : B'$.

2. By 5.2.15 and (1). ∎

∎

The following result is proved in van Benthem Jutting (1990) extending in a nontrivial way a result of Luo (1990) for a particular type system. The proof for arbitrary PTSs is somewhat involved and will not be given here.

Lemma 5.2.17 (Condensing lemma for PTS's). *In every* PTS *one has the following:*

$$\Gamma, x{:}A, \Delta \vdash B : C \;\&\; x \notin \Delta, B, C \;\;\Rightarrow\;\; \Gamma, \Delta \vdash B : C.$$

Here $x \notin \Delta, \ldots$ means that x is not free in Δ etc.

Corollary 5.2.18 (Decidability of type checking and typability for normalizing PTS's). *Let* $\lambda\mathrm{S} = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, *with $\mathcal{S}$ finite, be a* PTS *that* *is (weakly or stongly) normalizing. Then the questions of type checking and typability (in the sense of subsection 4.4) are decidable.*

Proof. This is proved in van Benthem Jutting (1990) as a corollary to the *method* of lemma 5.2.17, not to the result itself. ■ ■

On the other hand Meyer (1988) shows that for $\lambda *$ these questions are not decidable.

In 5.2.19 - 5.2.22 we will consider results that hold only in special PTS's.

Definition 5.2.19. Let $\lambda S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a given PTS.

λS is called *singly sorted* if

1. $(c : s_1), (c : s_2) \in \mathcal{A} \;\Rightarrow\; s_1 \equiv s_2$;
2. $(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R} \;\Rightarrow\; s_3 \equiv s_3'$.

Examples 5.2.20.

1. All systems in the λ-cube and $\lambda *$ and λU as well are singly sorted.
2. The PTS specified by

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

is not singly sorted.

Lemma 5.2.21 (Uniqueness of types lemma for singly sorted PTS's). *Let λS be a* PTS*that is singly sorted. Then*

$$\Gamma \vdash A : B_1 \;\&\; \Gamma \vdash A : B_2 \;\Rightarrow\; B_1 =_\beta B_2.$$

Proof. By induction on the structure of A. We treat two cases. Assume $\Gamma \vdash A : B_i$ for $i = 1, 2$.

Case 1. $A \equiv c$, a constant. By the generation lemma it follows that

$$\exists s_i = B_i \; (c : s_i) \text{ is an axiom}$$

for $i = 1, 2$. By the assumption that λS is singly sorted we can conclude that $s_1 \equiv s_2$, hence $B_1 = B_2$.

Case 2. $A \equiv \Pi x{:}A_1.A_2$. By the generation lemma it follows that

$$\Gamma \vdash A_1 : s_1 \ \& \ \Gamma, x : A_1 \vdash A_2 : s_2 \ \& \ B_1 = s_3$$

$$\Gamma \vdash A_1 : s_1' \ \& \ \Gamma, x{:}A_1 \vdash A_2 : s_2' \ \& \ B_2 = s_3'$$

for some rules (s_1, s_2, s_3) and (s_1', s_2', s_2'). By the induction hypothesis it follows that $s_1' = s_1$ and $s_2' = s_2$ hence $s_1' \equiv s_1$ and $s_2' \equiv s_2$. Hence by the fact that λS is singly sorted we can conclude that $s_3' \equiv s_3$. Therefore $B' = B$. ∎ ∎

Corollary 5.2.22. *Let λS be a singly sorted* PTS.

1. *Suppose $\Gamma \vdash A : B$ and $\Gamma \vdash A' : B'$. Then*

$$A =_\beta A' \quad \Rightarrow \quad B =_\beta B'.$$

2. *Suppose $\Gamma \vdash B : s, B =_\beta B'$ and $\Gamma \vdash A' : B'$. Then $\Gamma \vdash B' : s$.*

Proof. 1. If $A =_\beta A'$, then by the Church–Rosser theorem $A \twoheadrightarrow_\beta A''$ and $A' \twoheadrightarrow_\beta A''$ for some A''. Hence by the subject reduction theorem 5.2.15

$$\Gamma \vdash A'' : B \text{ and } \Gamma \vdash A'' : B'.$$

But then by uniqueness of types $B =_\beta B'$.

2. By the assumption and Corollary 5.2.14 it follows that $\Gamma \vdash B' : s'$ or $B' \equiv s'$ for some sort s'.

Case 1. $\Gamma \vdash B' : s'$. Since B and B' have a common reduct B'', it follows by the subject reduction theorem that $\Gamma \vdash B'' : s$ and $\Gamma \vdash B'' : s'$. By uniqueness of types one has $s \equiv s'$ and hence $\Gamma \vdash B' : s$.
Case 2. $B' \equiv s'$. Then $B \twoheadrightarrow_\beta s'$, hence by subject reduction $\Gamma \vdash s' : s$, i.e. $\Gamma \vdash B' : s$. ∎ ∎

Now we introduce a classification of pseudoterms that is useful for the analysis of legal terms in systems of the λ-cube.

Definition 5.2.23. A map $\sharp : \mathcal{T} \to \{0, 1, 2, 3\}$ is defined as follows:

$$\sharp(\Box) = 3; \sharp(*) = 2; \sharp({}^{\Box}x) = 1; \sharp({}^{*}x) = 0;$$

$$\sharp(s) = \sharp({}^{s}x) = \text{ arbitary, say 0, if } s \not\equiv \Box, *;$$

$$\sharp(\lambda x{:}A.B) = \sharp(\Pi x{:}A.B) = \sharp(BA) = \sharp(B).$$

For $A \in \mathcal{T}$ the value $\sharp(A)$ is called the *degree* of A.

It will be shown for all systems in the λ-cube that if $\Gamma \vdash A : B$, then $\sharp(A) + 1 = \sharp(B)$. This is a folklore result for AUTOMATH-like systems and the proof below is due to van Benthem Jutting. First some lemmas.

Lemma 5.2.24. *In* λC *and hence in all systems of the* λ*-cube one has the following:*

1. $\Gamma \not\vdash \Box : A$.
2. $\Gamma \not\vdash (AB) : \Box$.
3. $\Gamma \not\vdash (\lambda x{:}A.b) : \Box$.

Proof. 1. By induction on derivations one shows

$$\Gamma \vdash B : A \Rightarrow B \not\equiv \Box$$

2. Similarly one shows $\Gamma \vdash (AB) : C \Rightarrow C \not\equiv \Box$.
   We treat the case that the application rule is used last.

$$\frac{\Gamma \vdash A : (\Pi x{:}P.Q) \quad \Gamma \vdash B : P}{\Gamma \vdash (AB) : Q[x := B](\equiv C)}$$

   By 5.2.14 (1) one has $\Gamma \vdash (\Pi x{:}P.Q) : s$. hence by the generation lemma $\Gamma, x{:}P \vdash Q : s$. Therefore by $\Gamma \vdash B : P$ and the substitution lemma

$$\Gamma \vdash C \equiv Q[x := B] : s$$

   By (1) it follows that $C \not\equiv \Box$.

3. If $\Gamma \vdash (\lambda x{:}A.b) : \Box$, then by the generation lemma for some B one has $(\Pi x{:}A.B) =_\beta \Box$, contradicting the Church–Rosser theorem. ■
■

Lemma 5.2.25.

1. $\Gamma \vdash_{\lambda C} A : \Box \Rightarrow \sharp(A) = 2$.
2. $\Gamma \vdash_{\lambda C} A : B \;\&\; \sharp(A) \in \{2, 3\} \quad \Rightarrow \quad B \equiv \Box$.

Proof. 1. By induction on derivations.

2. Similarly. We treat two cases (that turn out to be impossible).

Case 1. The abstraction rule is used last:

$$\frac{\Gamma, x{:}A_1 \vdash b : B_1 \quad \Gamma \vdash (\Pi x{:}A_1.B_1) : s}{\Gamma \vdash (\lambda x{:}A_1.b) : (\Pi x{:}A_1.B_1)}.$$

Since $\sharp(b) = \sharp(\lambda x{:}A_1.b) \in \{2,3\}$ one has by the IH that $B_1 \equiv \Box$. By the generation lemma it follows that $\Gamma, x{:}A_1 \vdash B_1 : s'$, which is impossible by 5.2.24 (1).

Case 2. The conversion rule is used last:

$$\frac{\Gamma \vdash A : b' \quad \Gamma \vdash B' : s \quad B' =_\beta B}{\Gamma \vdash A : B}.$$

By the IH one has $B' \equiv \Box$. But then $B \twoheadrightarrow_\beta \Box$ so by subject reduction $\Gamma \vdash \Box : s$. Again this contradicts 5.2.24 (i). ■

■

Lemma 5.2.26. *If $\sharp(x) = \sharp(Q)$. Then $\sharp(P[x := Q]) = \sharp(P)$.*

Proof. Induction in the structure of P. ■ ■

Definition 5.2.27.

1. A statement $A : B$ is *ok* if $\sharp(A) + 1 = \sharp(B)$.
2. A statement $A : B$ is *hereditarily ok*, notation hok, if it is ok and moreover all substatements $y : P$ (occurring just after a symbol 'λ' or 'Π') in A and B are ok.

Proposition 5.2.28. *Let $\Gamma \vdash_{\lambda C} A : B$. Then $A : B$ and all statements in Γ are hok.*

Proof. By induction on the derivation of $\Gamma \vdash A : B$. We treat four cases.

Case 1. (axiom). The statement in $<> \vdash * : \Box$ is hok.

Case 2. (start rule). Suppose all statements in $\Gamma \vdash A : s$ are *hok*. Then also in $\Gamma, {}^s x{:}A \vdash {}^s x{:}A$, since $\sharp({}^s x) = \sharp(s) - 2$ and $\sharp(A) = \sharp(s) - 1$.

Case 3. (application rule). Suppose that the statements in $\Gamma \vdash F : (\Pi x{:}A.B)$ and $\Gamma \vdash a : A$ are *hok*. We have to show that $(Fa) : (B[x := a])$ is *hok*. This statement is ok since

$$\sharp(Fa) + 1 = \sharp(F) + 1 = \sharp(\Pi x{:}A.B) = \sharp(B) = \sharp(B[x := a])$$

by Lemma 5.2.26 and the fact that $x : A$ and $a : A$ are ok (so that $\sharp(x) = \sharp(a)$). The statement is also *hok* since all parts $y : P$ occur already in $\Gamma, F, (\Pi x{:}A.B)$ or a.

Case 4. (conversion rule). Suppose that all statements in $\Gamma \vdash A : B, \Gamma \vdash B' : s$ are *hok* and that $B =_\beta B'$. If we can show that

$\sharp(B) = \sharp(B')$ it follows that also $A : B'$ is *hok* and we are done. By Lemma 5.2.22 (2) one has $\Gamma \vdash B : s$.

Subcase 4.1. $s \equiv \Box$. Then $\sharp(B) = 2 = \sharp(B')$ by Lemma 5.2.25(1)

Subcase 4.2. $s \equiv *$. Then $\Gamma \vdash B : *$ and hence by Lemma 5.2.25(2) one has $\sharp(B) \notin \{2, 3\}$. Since $A : B$ is ok, we must have $\sharp(B) = 1$. Moreover $B' : s \equiv *$ is ok, hence also $\sharp(B') = 1$. ∎

∎

Corollary 5.2.29. $\Gamma \vdash_{\lambda C} A : B \Rightarrow \sharp(A) + 1 = \sharp(B)$.

Proposition 5.2.30.

1. *Let $(\lambda x{:}A.b)a$ be legal in* λC. *Then $\sharp(x) = \sharp(a)$.*
2. *Let A be legal in* λC. *Then*

$$A \twoheadrightarrow_\beta B \quad \Rightarrow \quad \sharp(A) = \sharp(B).$$

Proof. 1. By Corollary 5.2.14(1) one has $\Gamma \vdash (\lambda x{:}A.b)a : B$ for some Γ and B. Using the generation lemma once it follows that

$$\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A'.B') \text{ and } \Gamma \vdash a : A',$$

and using it once more that $\Gamma \vdash A : s$ and $(\Pi x{:}A.B'') =_\beta (\Pi x{:}A'.B')$, for some s and B''. Then $A =_\beta A'$, by the Church-Rosser theorem. Hence by the conversion rule $\Gamma \vdash a : A$. Therefore $a : A$ is ok. But also $x : A$ is ok. Thus it follows that $\sharp(x) = \sharp(a)$.

2. By induction on the generation of $A \twoheadrightarrow_\beta B$, using (1) and lemma 5.2.26. ∎

∎

Finally we show that PTS's extending λ2 the type $\perp \equiv (\Pi\alpha{:}*.\alpha)$ can be inhabited only by non normalizing terms. Hence, if one knows that the system is normalizing—as is the case for e.g. λ2 and λC—then this implies that $\perp$ is not inhabited. On the other hand if in a PTS the type $\perp$ is inhabited—as is the case for e.g. $\lambda*$—then not all typable terms are normalizing.

Proposition 5.2.31. *Let λS be a* PTS *extending* λ2. *Suppose $\vdash_{\lambda S} M : \perp$. Then M has no normal form.*

Proof. Suppose towards a contradiction that M has a nf N. Then by the subject reduction theorem 5.2.15 one has $\vdash_{\lambda S} N : \bot$. By the generation lemma N cannot be constant or a term starting with Π, since both kinds of terms should belong to a sort, but $\bot$ is not a sort. Moreover N is not a variable since the context is empty. Suppose N is an application; write $N \equiv N_1 N_2 \ldots N_k$, where N_1 is not an application anymore. By a reasoning as before N_1 cannot be a variable or a term starting with Π. But then $N_1 \equiv (\lambda x{:}A.P)$; hence N contains the redex $(\lambda x{:}A.P)N_2$, contradicting the fact that N is a nf. Therefore N neither can be an application. The only remaining possibility is that N starts with a λ. Then $N \equiv \lambda a{:}*.B$ and since $\vdash N : \bot$ one has $a{:}* \vdash B : a$. Again by the generation lemma B cannot be a constant nor a term starting with Π or λ. The only remaining possibility is that $B \equiv x C_1 \ldots C_k$. But then $x \equiv a$ and $k = 0$. Hence $a{:}* \vdash a : a$ which implies $a = *$, a contradiction. (The sets V and C are disjoint.) ■ ■

5.3 Strong normalization for the λ-cube

Recall that a pseudo-term M is called strongly normalizing, notation SN(M), if there is no infinite reduction starting from M.

Definition 5.3.1. Let λS be a PTS. Then λS is *strongly normalizing*, notation
$\lambda S \models$ SN, if all legal terms of λS are SN, i.e.

$$\Gamma \vdash A : B \Rightarrow \mathrm{SN}(A) \;\&\; \mathrm{SN}(B).$$

In this subsection it will be proved that all systems in the λ-cube satisfy SN. For this it is sufficient to show $\lambda C \models$ SN. This was first proved by Coquand (1985). We follow a proof due to Geuvers and Nederhof (1991) which is modular: first it is proved that

$$\lambda\omega \models \mathrm{SN} \quad\Rightarrow\quad \lambda\mathrm{C} \models \mathrm{SN} \tag{1}$$

and then

$$\lambda\omega \models \mathrm{SN} \tag{2}$$

The proof of (2) is due to Girard (1972) and is a direct generalization of his proof of $\lambda 2 \models$ SN as presented in subsection 4.3. Although the proof is relatively simple, it is ingenious and cannot be carried out in higher-order arithmetic. On the other hand the proof of (1) can be carried out in Peano arithmetic. This has as consequence that $\lambda\omega \models$ SN and $\lambda C \models$ SN are provably equivalent in Peano arithmetic, a fact that was first shown by Berardi (1989) using proof theoretic methods. The proof of Geuvers and

Nederhof uses a translation between λC and $\lambda\omega$ preserving reduction. This translation is inspired by the proof of Harper *et al.* (1987) showing that

$$\lambda{\to} \models \mathrm{SN} \quad \Rightarrow \quad \lambda\mathrm{P} \models \mathrm{SN}$$

using a similar translation. Now (1) and (2) will be proved. The proof is rather technical and the readers may skip it when first reading this chapter.

Proof of $\lambda\omega \models SN \;\Rightarrow\; \lambda\mathrm{C} \models SN$

This proof occupies 5.3.2 – 5.3.14. Two partial maps $\tau{:}\mathcal{T}{\to}\mathcal{T}$ and $[\![\ ]\!]{:}\mathcal{T}{\to}\mathcal{T}$ will be defined. Then τ will be extended to contexts and it will be proved that

$$\Gamma \vdash_{\lambda C} A : B \quad \Rightarrow \quad \tau(\Gamma) \vdash_{\lambda\omega} [\![A]\!] : \tau(B)$$

and

$$A {\to}_\beta A' \quad \Rightarrow \quad [\![A]\!] \twoheadrightarrow_{\neq 0} [\![A']\!].$$

($M \twoheadrightarrow_{\neq 0} N$ means that $M \twoheadrightarrow_\beta N$ in at least one reduction step. Then assuming that $\lambda\omega \models \mathrm{SN}$ one has

$$\begin{aligned} \Gamma \vdash_{\lambda C} A : B \quad &\Rightarrow \quad \mathrm{SN}([\![A]\!]) \\ &\Rightarrow \quad \mathrm{SN}(A). \end{aligned}$$

as is not difficult to show. This implies that we are done since by Corollary 5.2.14 it follows that also

$$\Gamma \vdash_{\lambda C} A : B \quad \Rightarrow \quad \mathrm{SN}(B).$$

In order to fulfill this program, next to τ and $[\![\ ]\!]$ another partial map ρ is needed.

Definition 5.3.2.

1. Write $\mathcal{T}_i = \{M \in \mathcal{T} \mid \sharp(M) = i\}$ and $\mathcal{T}_{i,j} = \mathcal{T}_i \cup \mathcal{T}_j$; similarly $\mathcal{T}_{i,j,k}$ is defined.

2. Let $A \in \mathcal{T}$. In λC one uses the following terminology.

$$\begin{array}{lcl} A \text{ is a } \textit{kind} & \Leftrightarrow & \exists\Gamma[\Gamma \vdash A : \Box]; \\ A \text{ is a } \textit{constructor} & \Leftrightarrow & \exists\Gamma, B[\Gamma \vdash A : B : \Box]; \\ A \text{ is a } \textit{type} & \Leftrightarrow & \exists\Gamma[\Gamma \vdash A : *]; \\ A \text{ is an } \textit{object} & \Leftrightarrow & \exists\Gamma, B[\Gamma \vdash A : B : *]. \end{array}$$

Note that types are constructors and that for A legal in λC one has

$$\begin{array}{lcl} A \text{ is kind} & \Leftrightarrow & \sharp(A) = 2; \\ A \text{ is constructor or type} & \Leftrightarrow & \sharp(A) = 1; \\ A \text{ is object} & \Leftrightarrow & \sharp(A) = 0. \end{array}$$

Moreover for legal A one has $\sharp(A) = 3$ iff $A \equiv \Box$.

Definition 5.3.3. A map $\rho:\mathcal{T}_{2,3}\to\mathcal{T}$ is defined as follows:

$$
\begin{array}{rcll}
\rho(\Box) & = & *; & \\
\rho(*) & = & *; & \\
\rho(\Pi x{:}A.B) & = & \rho(A)\to\rho(B), & \text{if } \sharp(A)=2; \\
 & = & \rho(B), & \text{if } \sharp(A)\neq 2; \\
\rho(\lambda x{:}A.B) & = & \rho(B); & \\
\rho(BA) & = & \rho(B). &
\end{array}
$$

It is clear that if $\sharp(A)\in\{2,3\}$, then $\rho(A)$ is defined and moreover $FV(\rho(A)) = \emptyset$.

Lemma 5.3.4.

1. $\Gamma\vdash_{\lambda C} A:\Box \quad\Rightarrow\quad \vdash_{\lambda\omega}\rho(A):\Box$.
2. *Let* $A\in\mathcal{T}_{2,3}$ *and* $\sharp(a)=\sharp(x)$. *Then* $\rho(A[x:=a])\equiv\rho(A)$.
3. *Let* $A\in\mathcal{T}_{2,3}$ *be legal and* $A\twoheadrightarrow_\beta B$. *Then* $\rho(A)\equiv\rho(B)$.
4. *Let* $\Gamma\vdash_{\lambda C} A_i:\Box, i=1,2$. *Then*

$$A_1 =_\beta A_2 \quad\Rightarrow\quad \rho(A_1)\equiv\rho(A_2).$$

Proof. 1. By induction on the generation of $A:\Box$. We treat two cases.
Case 1. $\Gamma\vdash_{\lambda C} A:\Box$ is $\Gamma', x{:}C\vdash_{\lambda C} A:\Box$ and follows directly from $\Gamma'\vdash_{\lambda C} A:\Box$ and $\Gamma'\vdash_{\lambda C} C:s$. By the induction hypothesis one has $\vdash_{\lambda\omega}\rho(A):\Box$.
Case 2. $\Gamma\vdash_{\lambda C} A:\Box$ is $\Gamma\vdash_{\lambda C}(A_1A_2):B[x:=A_2]$ and follows directly from $\Gamma\vdash_{\lambda C} A_1:(\Pi x{:}C.B)$ and $\Gamma\vdash_{\lambda C} A_2:C$. Then either $B\equiv\Box$, which is impossible by Lemma 5.2.24(2), or $B\equiv x$ and $A_2\equiv\Box$. But also $\Gamma\vdash_{\lambda C}\Box:C$ is impossible.

2. By induction on the structure of A.

3. By induction on the relation $\twoheadrightarrow$, using (2) and Proposition 5.2.30 for the case $A\equiv(\lambda x{:}D.P)Q$ and $B\equiv P[x:=Q]$.

4. By (3). ■

∎

A special variable 0 with $0:*$ will be used in the definition of τ. Moreover, in order to define the required map from λC to $\lambda\omega$ 'canonical' constants in types are needed. For this reason a fixed context Γ_0 will be introduced from which it follows that every type has an inhabitant.

Definition 5.3.5.

1. Γ_0 is the $\lambda\omega$ context
$$0{:}*, c{:}\bot,$$
where $\bot \equiv \Pi x{:}*.x$.

2. If $\Gamma \vdash_{\lambda\omega} B : *$, then c^B is defined as cB.

3. If $\Gamma \vdash_{\lambda\omega} B : \Box$, then c^B is defined inductively as follows; note that if $B \not\equiv *$, then it follows from the generation Lemma 5.2.13 that $B \equiv B_1 {\rightarrow} B_2$. Therefore we can define
$$\begin{array}{rcl} c^* & \equiv & 0; \\ c^{B_1\rightarrow B_2} & \equiv & \lambda x{:}B_1.c^{B_2}. \end{array}$$

Lemma 5.3.6. *If* $\Gamma \vdash_{\lambda\omega} B : s$, *then* $\Gamma_0, \Gamma \vdash_{\lambda\omega} c^B : B$.

Proof. If $s \equiv *$, then $c^B \equiv cB$ and the conclusion clearly holds. If $s \equiv \Box$, then the result follows by induction on B. ■ ■

Definition 5.3.7.

1. A map $\tau{:}\mathcal{T}_{1,2,3} {\rightarrow} \mathcal{T}$ is defined as follows.
$$\begin{array}{lcll} \tau(\Box) & = & 0; & \\ \tau(*) & = & 0; & \\ \tau({}^{\Box}x) & = & {}^{\Box}x; & \\ \tau(\Pi x{:}A.B) & = & \Pi x{:}\rho(A).\tau(A){\rightarrow}\tau(B), & \text{if } \sharp(A) = 2; \\ & = & \Pi x{:}\tau(A).\tau(B), & \text{if } \sharp(A) = 1; \\ & = & \tau(B), & \text{else}; \\ \tau(\lambda x{:}A.B) & = & \lambda x{:}\rho(A).\tau(B), & \text{if } \sharp(A) = 2; \\ & = & \tau(B), & \text{else}; \\ \tau(BA) & = & \tau(B), & \text{if } \sharp(A) = 0; \\ & = & \tau(B)\tau(A), & \text{else}. \end{array}$$

2. The map τ is extended to pseudo-contexts as follows.
$$\tau({}^{*}x{:}A) = {}^{*}x{:}\tau(A); \tau({}^{\Box}x{:}A) = {}^{\Box}x{:}\rho(A), {}^{*}x{:}\tau(A).$$
Let $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ be a pseudo-context. Then
$$\tau(\Gamma) = \Gamma_0, \tau(x_1{:}A_1), \ldots, \tau(x_n{:}A_n).$$

By induction on the structure of A it follows that if $A \in \mathcal{T}_{1,2,3}$, then $\tau(A)$ is defined and moreover ${}^{*}x \notin FV(\tau(A))$.

Lemma 5.3.8.

1. *Let $B \in \mathcal{T}_{1,2,3}$ and $\sharp(a) = \sharp(x)$. Then*

$$\begin{array}{rcll} \tau(B[x := a]) & = & \tau(B)[x := \tau(a)], & \textit{if } x \equiv {}^{\square}x; \\ & = & \tau(B), & \textit{if } x \equiv {}^{*}x. \end{array}$$

2. *If $A \in \mathcal{T}_{1,2,3}$ is legal and $A \twoheadrightarrow B$, then $\tau(A) \twoheadrightarrow \tau(B)$.*

Proof. 1. By induction on the structure of B, using Lemma 5.3.4(3).

2. By induction on the generation of $A \twoheadrightarrow B$. We only treat the case $A \equiv (\lambda x{:}D.b)a$ and $B \equiv b[x := a]$. By the generation lemma it follows that $\Gamma \vdash D : s$ with $s \equiv *$ or $s \equiv \square$. In the first case one has $x \equiv {}^{*}x$ and by (1)

$$\tau((\lambda x{:}D.b)a) \equiv \tau(b) \equiv \tau(b[x := a]) \equiv \tau(B).$$

In the second case one has $x \equiv {}^{\square}x$ and by (1)

$$\begin{array}{rcl} \tau(A) & \equiv & (\lambda x{:}\rho(D).\tau(b))\tau(a) \\ & \rightarrow & \tau(b)[x := \tau(a)] \\ & \equiv & \tau(B). \blacksquare \end{array}$$

$\blacksquare$

Lemma 5.3.9. *Let $\Gamma \vdash_{\lambda C} B : \square$ or $B \equiv \square$. Then*

$$\Gamma \vdash_{\lambda C} A : B \Rightarrow \tau(\Gamma) \vdash_{\lambda\omega} \tau(A) : \rho(B).$$

Proof. By induction on the proof of $\Gamma \vdash_{\lambda C} A : B$. We treat three cases.

Case 1. $\Gamma \vdash_{\lambda C} A : B$ is $\Gamma', x{:}C \vdash_{\lambda C} A : B$ and follows from $\Gamma' \vdash_{\lambda C} A : B$ and $\Gamma' \vdash_{\lambda C} C : s$ by the weakening rule. By the IH one has

$$\tau(\Gamma') \vdash_{\lambda\omega} \tau(A) : \rho(B) \ \&\ \tau(\Gamma') \vdash_{\lambda\omega} \tau(C) : *.$$

We must show

$$\tau(\Gamma'), \tau(x{:}C) \vdash_{\lambda\omega} \tau(A) : \rho(B). \tag{1}$$

If $x \equiv {}^{*}x$, then $\tau(x{:}C) \equiv x{:}\tau(C)$ and (1) follows from the IH by weakening. If $x \equiv {}^{\square}x$, then $\tau(x{:}C) \equiv {}^{\square}x{:}\rho(C), {}^{*}x{:}\tau(C)$ and (1) follows from the IH by weakening twice. (Note that in this case $\Gamma' \vdash_{\lambda C} C : \Box$, so by Lemma 5.3.4 (1) one has $\vdash_{\lambda\omega} \rho(C) : \Box$.)

Case 2. $\Gamma \vdash_{\lambda C} A : B$ is $\Gamma \vdash_{\lambda C} (\lambda x{:}D.b) : (\Pi x{:}D.B)$ and follows from $\Gamma \vdash_{\lambda C} (\Pi x{:}D.B) : s$ and $\Gamma, x{:}D \vdash_{\lambda C} b : B$. By the assumption of the theorem one has $s \equiv \Box$.

Subcase 2.1. $\sharp(D) = 2$. By the IH it follows among other things that

$$\tau(\Gamma) \vdash_{\lambda\omega} [\Pi x{:}\rho(D).\tau(D) {\rightarrow} \tau(B)] : *$$

$$\tau(\Gamma), {}^{\Box}x{:}\rho(D), {}^{*}x{:}\tau(D) \vdash_{\lambda\omega} \tau(b) : \rho(B). \tag{2}$$

We must show

$$\tau(\Gamma) \vdash_{\lambda\omega} (\lambda x{:}\rho(D).\tau(D)) : (\rho(D) {\rightarrow} \rho(B)).$$

Now ${}^{*}x$ does not occur in $\rho(B)$ since it is closed, nor in $\tau(b)$. Therefore, by (2) and the substitution lemma, using $c^{\tau(D)}$ in context $\Gamma_0 \subseteq \tau(\Gamma)$, one has

$$\tau(\Gamma), {}^{\Box}x{:}\rho(D) \vdash_{\lambda\omega} \tau(b) : \rho(B)$$

and hence

$$\begin{aligned} \tau(\Gamma) \vdash_{\lambda\omega} (\lambda x{:}\rho(D).\tau(b)) : (\Pi x{:}\rho(D).\rho(B)) \quad &\equiv \quad \rho(D) {\rightarrow} \rho(B) \\ &\equiv \quad \rho(\Pi x{:}D.B), \end{aligned}$$

since $\rho(B)$ is closed.

Subcase 2.2. $\sharp(D) = 1$. Similarly.

Case 3. $\Gamma \vdash_{\lambda C} A : B$ is $\Gamma \vdash_{\lambda C} (\Pi x{:}D.E) : s_2$
and follows directly from $\Gamma \vdash_{\lambda C} D : s_1$ and $\Gamma, x{:}D \vdash_{\lambda C} E : s_2$.

Subcase 3.1. $s_1 \equiv *$. The IH states

$$\begin{aligned} &\tau(\Gamma) \vdash_{\lambda\omega} \tau(D) : *; \\ &\tau(\Gamma), x{:}\tau(D) \vdash_{\lambda\omega} \tau(E) : *. \end{aligned}$$

We have to show

$$\tau(\Gamma) \vdash_{\lambda\omega} (\Pi x{:}\tau(D).\tau(E)) : *;$$

but this follows immediately from the IH.

Subcase 3.2. $s_1 \equiv \Box$. The IH states now

$$\tau(\Gamma) \vdash_{\lambda\omega} \tau(D) : *,$$
$$\tau(\Gamma), {}^{\Box}x{:}\rho(D), {}^{*}x{:}\tau(D) \vdash_{\lambda\omega} \tau(E) : *.$$

We have to show

$$\tau(\Gamma) \vdash_{\lambda\omega} (\Pi x{:}\rho(D).\tau(D) \to \tau(E)) : *;$$

this follows from the IH and the fact that the fresh variable ${}^{*}x$ does not occur in $\tau(E)$. ■

■

Now the third partial map on pseudo-terms will be defined.

Definition 5.3.10. The map $[\![-]\!]{:}\mathcal{T}_{0,1,2} \to \mathcal{T}$ is defined as follows. Remember that in the context $\Gamma_0 \equiv 0{:}*, c{:}\bot$ we defined expressions c^A such that $\Gamma \vdash A : s \Rightarrow \Gamma_0, \Gamma \vdash c^A : A$.

$$\begin{array}{rcll}
[\![*]\!] & = & c^0 & \\
[\![{}^{*}x]\!] & = & {}^{*}x & \\
[\![{}^{\Box}x]\!] & = & {}^{*}x; & \\
[\![\Pi x{:}A.B]\!] & = & c^{0\to 0\to 0}[\![A]\!]([\![B]\!][{}^{\Box}x := c^{\rho(A)}][{}^{*}x := c^{\tau(A)}]), & \text{if } \sharp(A) = 2; \\
 & = & c^{0\to 0\to 0}[\![A]\!]([\![B]\!][{}^{*}x := c^{\tau(A)}]), & \text{if } \sharp(A) \neq 2; \\
[\![\lambda x{:}A.B]\!] & = & (\lambda z{:}0\lambda^{\Box}x{:}\rho(A)\lambda^{*}x{:}\tau(A).[\![B]\!])[\![A]\!], & \text{if } \sharp(A) = 2; \\
 & = & (\lambda z{:}0\lambda^{*}x{:}\tau(A).[\![B]\!])[\![A]\!], & \text{if } \sharp(A) \neq 2; \\
[\![BA]\!] & = & [\![B]\!]\tau(A)[\![A]\!], & \text{if } \sharp(A) = 2. \\
 & = & [\![B]\!][\![A]\!], & \text{if } \sharp(A) \neq 2.
\end{array}$$

In the above $z \equiv {}^{*}z$ is fresh.

Proposition 5.3.11.

$$\Gamma \vdash_{\lambda C} A : B \quad \Rightarrow \quad \tau(\Gamma) \vdash_{\lambda\omega} [\![A]\!] : \tau(B).$$

Proof. By induction on the derivation of $A : B$. We treat two cases.

Case 1. $\Gamma \vdash_{\lambda C} A : B$ is $\Gamma \vdash_{\lambda C} (\Pi x{:}D.E) : s_2$ and follows from $\Gamma \vdash_{\lambda C} D : s_1$ and $\Gamma, x{:}D \vdash E : s_2$. By the IH one has $\tau(\Gamma) \vdash_{\lambda\omega} [\![D]\!] : 0$ and $\tau(\Gamma, x{:}D) \vdash_{\lambda\omega} [\![E]\!] : 0$. By Lemma 5.3.9 one has $\tau(\Gamma) \vdash_{\lambda\omega} \tau(D) : *$, hence $\tau(\Gamma) \vdash_{\lambda\omega} c^{\tau(D)} : \tau(D)$.

If $s_1 \equiv *$, then $x \equiv {}^*x$ and $\tau(\Gamma, x{:}D) \equiv \tau(\Gamma), x{:}\tau(D)$. Therefore by the substitution lemma

$$\tau(\Gamma) \vdash_{\lambda\omega} [\![E]\!][x := c^{\tau(D)}] : 0.$$

Hence by the application rule twice

$$\tau(\Gamma) \vdash_{\lambda\omega} c^{0\to 0\to 0}[\![D]\!]([\![E]\!][x := c^{\tau(D)}]) : 0.$$

If $s_1 \equiv \Box$, then $x \equiv {}^\Box x$ and $\tau(\Gamma, x{:}D) \equiv \tau(\Gamma), {}^\Box x{:}\rho(D), {}^*x{:}\tau(D)$. Therefore by the substitution lemma

$$\tau(\Gamma) \vdash_{\lambda\omega} [\![E]\!][{}^\Box x := c^{\rho(D)}][{}^*x := c^{\tau(D)}] : 0.$$

Hence by the application rule twice

$$\tau(\Gamma) \vdash_{\lambda\omega} c^{0\to 0\to 0}[\![D]\!]([\![E]\!][{}^\Box x := c^{\rho(D)}][{}^*x := c^{\tau(D)}]) : 0.$$

In both cases one has

$$\tau(\Gamma) \vdash_{\lambda\omega} [\![\Pi x{:}D.E]\!] : 0$$

Case 2. $\Gamma \vdash_{\lambda C} A : B$ is $\Gamma \vdash_{\lambda C} (\lambda x{:}D.b) : (\Pi x{:}D.B)$ and follows from

$$\Gamma, x{:}D \vdash_{\lambda C} b : B$$

and

$$\Gamma \vdash_{\lambda C} (\Pi x{:}D.B) : s.$$

By the generation lemma (and the Church-Rosser theorem) one has for some sort s_1

$$\Gamma \vdash_{\lambda C} D : s_1 \ \&\ \Gamma, x : D \vdash_{\lambda C} B : s.$$

By the IH one has

$$\tau(\Gamma, x{:}D) \vdash_{\lambda\omega} [\![b]\!] : \tau(B)$$

and

$$\tau(\Gamma) \vdash_{\lambda\omega} [\![D]\!] : 0.$$

By Lemma 5.3.9 one has

$$\tau(\Gamma) \vdash_{\lambda\omega} \tau(D) : *$$

and

$$\tau(\Gamma, x{:}D) \vdash_{\lambda\omega} \tau(B) : *.$$

If $s_1 \equiv *$, then $x \equiv {}^*x$ and $\tau(\Gamma, x{:}D) \equiv \tau(\Gamma), x{:}\tau(D)$.

Therefore by two applications of the abstraction rule and one application of the product rule one obtains

$$\tau(\Gamma) \vdash_{\lambda\omega} ((\lambda z{:}0\lambda x{:}\tau(D).[\![b]\!])[\![D]\!]) : (\tau(D) \rightarrow \tau(B)).$$

If $s_1 \equiv \Box$, then a similar argument shows

$$\tau(\Gamma) \vdash_{\lambda\omega} (\lambda z{:}0\lambda^{\Box}x{:}\rho(D)\lambda^{*}x{:}\tau(D).[\![b]\!])[\![D]\!] : (\Pi x{:}\rho(D).\tau(D) \rightarrow \tau(B)).$$

In both cases one has

$$\tau(\Gamma) \vdash_{\lambda\omega} [\![\lambda x{:}D.b]\!] : \tau(\Pi x{:}D.B). \blacksquare$$

$\blacksquare$

Lemma 5.3.12. *Let $A.B \in T$. Then*

1. $x \equiv {}^{*}x \Rightarrow [\![A[{}^{*}x := B]]\!] \equiv [\![A]\!][{}^{*}x := [\![B]\!]]$
2. $x \equiv {}^{\Box}x \Rightarrow [\![A[{}^{\Box}x := B]]\!] \equiv [\![A]\!][{}^{\Box}x := \tau(B), {}^{*}x := [\![B]\!]]$.

Proof. 1. By induction on the structure of A. We treat one case: $A \equiv \Pi y{:}D.E$. Write $P^{+} \equiv P[x := B]$. Now

$$\begin{array}{rcl} [\![A^{+}]\!] & \equiv & [\![\Pi y{:}D^{+}.E^{+}]\!] \\ & \equiv & c^{0\rightarrow 0\rightarrow 0}[\![D^{+}]\!][\![E^{+}]\!][y := c^{\tau(D^{+})}] \\ & \equiv & (c^{0\rightarrow 0\rightarrow 0}[\![D]\!][\![E]\!][y := c^{\tau(D)}])[x := [\![B]\!]] \\ & \equiv & [\![\Pi y{:}D.E]\!][x := [\![B]\!]], \end{array}$$

by the induction hypothesis, the substitution lemma and the fact that $\tau(D[{}^{*}x := B]) \equiv \tau(D)$.

2. Similarly, using the convention about hygiene made in definition 5.2.1. $\blacksquare$

$\blacksquare$

Lemma 5.3.13. *Let $A, B \in \mathcal{T}_{0,1,2}$. Then*

$$A \rightarrow B \quad \Rightarrow \quad [\![A]\!] \twoheadrightarrow_{\neq 0} [\![B]\!].$$

where $\twoheadrightarrow_{\neq 0}$ denotes that the reduction takes at least one step.

Proof. By induction on the generation of $A \rightarrow B$. We treat only the case that $A \rightarrow B$ is

$$(\lambda x{:}D.P)Q \rightarrow P[x := Q].$$

If $x \equiv {}^*x$, then

$$\begin{array}{rcl} [\![(\lambda x{:}D.P)Q]\!] & \equiv & (\lambda z{:}0\lambda x{:}\tau(D).[\![P]\!])[\![D]\!][\![Q]\!] \\ & \twoheadrightarrow_{\neq 0} & [\![P]\!][x := [\![Q]\!]] \\ & \equiv & [\![P[x := Q]]\!]. \end{array}$$

If $x \equiv {}^{\Box}x$, then

$$\begin{array}{rcl} [\![(\lambda x{:}D.P)Q]\!] & \equiv & (\lambda z{:}0\lambda^{\Box}x{:}\rho(D)\lambda^{*}x{:}\tau(D).[\![P]\!])[\![D]\!]\tau(Q)[\![Q]\!] \\ & \twoheadrightarrow_{\neq 0} & [\![P]\!][{}^{\Box}x := \tau(Q), {}^{*}x := [\![Q]\!]] \\ & \equiv & [\![P[x := Q]]\!].\ \blacksquare \end{array}$$

∎

Theorem 5.3.14. $\lambda\omega \models SN \Rightarrow \lambda\mathrm{C} \models SN$.

Proof. Suppose $\lambda\omega \models$ SN. Let M be a legal λC term. By Corollary 5.2.14 it is sufficient to assume $\Gamma \vdash_{\lambda C} M : A$ in order to show SN(M). Consider a reduction starting with $M \equiv M_0$

$$M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \cdots$$

One has $\Gamma \vdash_{\lambda C} M_i : A$, and therefore $\Gamma \vdash_{\lambda\omega} [\![M_i]\!] : \tau(A)$ for all i, by Proposition 5.3.11. By lemma 5.3.13 one has

$$[\![M_0]\!] \twoheadrightarrow_{\neq 0} [\![M_1]\!] \twoheadrightarrow_{\neq 0} \cdots$$

But then $[\![M]\!]$ is a legal $\lambda\omega$ term and hence the sequence is finite. ■ ∎

Corollary 5.3.15 (Berardi). *In HA, the system of intuitionistic arithmetic, one can prove*

$$\lambda\omega \models SN \quad \Leftrightarrow \quad \lambda\mathrm{C} \models SN.$$

Proof. The implication $\Leftarrow$ is trivial. By inspecting the proof of 5.3.14 it can be verified that everything is formalizable in HA. ■ ∎

This corollary was first proved in Berardi (1989) by proof theoretic methods. The present proof of Geuvers and Nederhof gives a more direct argument.

The proof of $\lambda\omega \models SN$
occupies 5.3.16 -5.3.32. The result will be proved using the following steps:

1. A map $|-|:\mathcal{T}_0 \to \Lambda$ will be defined such that

$$\Gamma \vdash_{\lambda\omega} A : B : * \Rightarrow \mathrm{SN}(|A|);$$

2. $\Gamma \vdash_{\lambda\to} A : B : * \Rightarrow \mathrm{SN}(A)$;
3. $\Gamma \vdash_{\lambda\omega} A : B : \square \Rightarrow \mathrm{SN}(A)$;
4. $\Gamma \vdash_{\lambda\omega} A : B : * \Rightarrow \mathrm{SN}(A)$;
5. $\Gamma \vdash_{\lambda\omega} A : B \Rightarrow \mathrm{SN}(A)\&\mathrm{SN}(B)$.

Definition 5.3.16. A map $|-|:\mathcal{T}_0 \to \Lambda$ is defined as follows:

$$\begin{array}{lcll}
|{}^*x| & = & x; & \\
|\lambda x{:}A.B| & = & \lambda x.|B|, & \text{if } \sharp(A) = 1; \\
 & = & |B|, & \text{else}; \\
|BA| & = & |B||A|, & \text{if } \sharp(A) = 0; \\
 & = & |B|, & \text{else}; \\
|\Pi x{:}A.B| & = & |B|. &
\end{array}$$

The last clause is not used essentially, since legal terms $\Pi x{:}A.B$ never have degree 0. Typical examples of $|-|$ are the following.

$$\begin{array}{l}
|\lambda x{:}\alpha.x| = \lambda x.x; \\
|\lambda\alpha{:}*.\lambda x{:}\alpha.x| = \lambda x.x; \\
|(\lambda x{:}\alpha.x)y| = (\lambda x.x)y; \\
|(\lambda\alpha{:}*.\lambda x{:}\alpha.x)\beta| = \lambda x.x.
\end{array}$$

The following lemma shows what kinds exist in $\lambda\omega$ and what kinds and objects in $\lambda\to$.

Lemma 5.3.17. *Let K be the set of pseudo-terms defined by the abstract syntax $K = * \mid K \to K$. So $K = \{*, *\to*, *\to*\to*, \ldots\}$. Then*

1. $\Gamma \vdash_{\lambda\omega} A : \square \Rightarrow A \in K$.
2. $\Gamma \vdash_{\lambda\omega} B : A : \square \Rightarrow A, B$ *do not contain any* ${}^*x$.
3. $\Gamma \vdash_{\lambda\to} A : \square \Rightarrow A \equiv *$.
4. $\Gamma \vdash_{\lambda\to} A : * \Rightarrow A$ *is an nf.*

Proof. By induction on derivations. ■ ■

Lemma 5.3.18. *Let $A \equiv \Box$ or $\Gamma \vdash_{\lambda\omega} A : \Box$. Then for all terms B legal in $\lambda\omega$ one has*

$$A =_\beta B \Rightarrow A \equiv B.$$

Proof. First let $A \equiv \Box$. Suppose B is legal and $A =_\beta B$. By the Church–Rosser theorem one has $B \twoheadrightarrow_\beta \Box$. Then the last step in this reduction must be

$$(\lambda x{:}A_1.A_2)A_3 \rightarrow_\beta A_2[x := A_3] \equiv \Box.$$

Case 1. $A_2 \equiv x$ and $A_3 \equiv \Box$. Then by 5.2.30 one has $\sharp(\Box) = \sharp(x)$, which is impossible.
Case 2. $A_2 \equiv \Box$. Then $(\lambda x{:}A_1.\Box)$ is legal, hence $\Gamma \vdash (\lambda x{:}A_1.\Box) : C$ for some Γ, C. But then by 5.2.29 one has $\sharp(C) = \sharp(\lambda x{:}A_1.\Box) + 1 = 4$, a contradiction.

If $\Gamma \vdash_{\lambda\omega} A : \Box$, then $A \in K$ as defined in 5.3.17 and similarly a contradiction is obtained. (In case 2 one has $\Gamma \vdash (\lambda x{:}A_1.A) : (\Pi x{:}A_1.\Box)$, but then $\Gamma \vdash (\Pi x{:}A_1.\Box) : s$.) ■ ■

Now it will be proved in 5.3.19 - 5.3.24 that if $\Gamma \vdash_{\lambda\omega} A : B : *$, then $\mathrm{SN}(|A|)$. The proof is related to the one for $\lambda 2-$Curry in section 4.3. Although the proof is not very complicated, it cannot be carried out in higher-order arithmetic PA^ω (because as Girard (1972) shows $\mathrm{SN}(\lambda\omega)$ implies $\mathrm{Con}(PA^\omega)$ and Gödels second incompleteness theorem applies).

We work in ZF-set theory. Let $\mathcal{U}$ be a large enough set. (If syntax is coded via arithmetic in the set of natural numbers ω, hence the set of type-free λ-terms Λ is a subset of ω, then $\mathcal{U} = \mathcal{V}_{\omega 2}$ will do; it is closed under the operations powerset, function spaces and under syntactic operations. Here $\mathcal{V}_\alpha$ is the usual set-theoretic hierarchy defined by $\mathcal{V}_0 = \varnothing, \mathcal{V}_{\alpha+1} = P(\mathcal{V}_\alpha)$ and $\mathcal{V}_\lambda = \cup_{\alpha\in\lambda}\mathcal{V}_\alpha$; moreover $\omega 2$ is the ordinal $\omega + \omega$.)

Definition 5.3.19.

1. A valuation is a map $\rho{:}V \rightarrow \mathcal{U}$.
2. Given a valuation ρ a map $[\![-]\!]_\rho{:}\mathcal{T} \rightarrow \mathcal{U} \cup \{\mathcal{U}\}$ is defined as follows: Remember that $X \rightarrow Y = \{F \in \Lambda \mid \forall M \in X\ FM \in Y\}$and that SAT $= \{X \subseteq \Lambda \mid X \text{ is saturated}\}$.

$$\begin{array}{rcl} [\![\Box]\!]_\rho &=& \mathcal{U}; \\ [\![*]\!]_\rho &=& \mathrm{SAT}; \\ [\![x]\!]_\rho &=& \rho(x); \end{array}$$

$$
\begin{array}{rcll}
[\![\Pi x{:}A.B]\!]_\rho & = & [\![A]\!]_\rho \to [\![B]\!]_\rho & \text{if } \sharp(A) = \sharp(B) = 1, \\
& = & [\![B]\!]_\rho^{[\![A]\!]_\rho}, & \text{if } \sharp(A) = \sharp(B) = 2, \\
& = & \cap\{[\![B]\!]_{\rho[x:=f]} \mid f \in [\![A]\!]_\rho\}, & \text{if } \sharp(A) = 2, \sharp(B) = 1, \\
& = & \varnothing, & \text{else;} \\
[\![\lambda x{:}A.B]\!]_\rho & = & \lambda x.[\![B]\!]_{\rho[x:=x]}, & \text{if } \sharp(A) = 1, \sharp(B) = 0, \\
& = & \lambda f \in [\![A]\!]_\rho.[\![B]\!]_{\rho[x:=f]}, & \text{if } \sharp(A) = 2, \sharp(B) = 1, \\
& = & [\![B]\!]_\rho, & \text{if } \sharp(A) = 2, \sharp(B) = 0, \\
& = & \varnothing, & \text{else;} \\
[\![BA]\!]_\rho & = & [\![B]\!]_\rho [\![A]\!]_\rho, & \text{if } \sharp(A) = \sharp(B) = 0, \\
& = & [\![B]\!]_\rho([\![A]\!]_\rho), & \text{if } \sharp(A) = \sharp(B) = 1, \\
& = & [\![B]\!]_\rho, & \text{if } \sharp(A) = 1, \sharp(B) = 0, \\
& = & \varnothing, & \text{else.}
\end{array}
$$

Comment 5.3.20. In the first clauses of the definitions of $[\![\Pi x{:}A.B]\!]_\rho$, $[\![\lambda x{:}A.B]\!]_\rho$ and $[\![BA]\!]_\rho$ a syntactic operation (as coded in set theory) is used ($\to$ as defined in 4.3.1.(2) extended to sets, λ abstraction and application as syntactic operations extended to $\mathcal{U}$). In the second clauses some set theoretic operations are used (function spaces, lambda abstraction, function application). In the third clause in the definition of $[\![\Pi x{:}A.B]\!]_\rho$ an essential impredicativity – the 'Girard trick' – occurs: $[\![\Pi x{:}A.B]\!]_\rho$ for a fixed ρ is defined in terms of $[\![B]\!]_\rho$ for arbitrary ρ. The fourth clauses are not used essentially.

Definition 5.3.21. Let ρ be a valuation.

- $\rho \models A : B \iff [\![A]\!]_\rho \in [\![B]\!]_\rho$.
- $\rho \models \Gamma \iff \rho \models x : A$ for each $(x{:}A) \in \Gamma$.
- $\Gamma \models A : B \iff \forall \rho\, [\rho \models \Gamma \Rightarrow \rho \models A : B]$.

Lemma 5.3.22. *Let ρ be a valuation with $\rho \models \Gamma$.*

1. *Assume that A is legal in $\lambda\omega$ and $\sharp(A) = 0$. Then*

$$[\![A]\!]_\rho = |A|[\vec{x} := \rho(\vec{x})] \in \Lambda.$$

2. *Assume $\sharp(x) = \sharp(a)$. Then*

$$[\![B[x := a]]\!]_\rho = [\![B]\!]_{\rho[x:=[\![a]\!]_\rho]}.$$

3. *Let B be legal in $\lambda\omega$. Suppose either $\sharp(B) = 0$ and $\sharp(a) = \sharp(x) = 1$ or $\sharp(B) = 1$ and $\sharp(a) = \sharp(x) = 0$. Then*

$$[\![B[x := a]]\!]_\rho = [\![B]\!]_\rho$$

4. *Let A, A' be legal in $\lambda\omega$ and $\sharp(A) = \sharp(A') \neq 0$. Then for all ρ*

$$A =_\beta A' \;\;\Rightarrow\;\; [\![A]\!]_\rho = [\![A']\!]_\rho.$$

Proof. 1. By induction on the structure of A.

2. By induction on the structure of B.

3. By induction on the structure of B.

4. Show that if A legal, $\sharp(A) \neq 0$ and $A \twoheadrightarrow_\beta A'$, then $[\![A]\!]_\rho = [\![A']\!]_\rho$. ■

Proposition 5.3.23.

$$\Gamma \vdash_{\lambda\omega} A : B \;\;\Rightarrow\;\; \Gamma \models A : B.$$

Proof. By induction on the derivation of $A : B$. Since these proofs should be familiar by now, the details are left to the reader. ■

Corollary 5.3.24.

1. $\Gamma \vdash_{\lambda\omega} A : B : * \;\;\Rightarrow\;\; SN(|A|)$.
2. $\Gamma \vdash_{\lambda\to} A : B : * \;\;\Rightarrow\;\; SN(A)\;\&\;SN(B)$.

Proof. For each kind k a canonical element $f^k \in [\![k]\!]_\rho$ will be defined.

$$\begin{array}{rcl} f^* & = & \mathrm{SN} \\ f^{k_1\to k_2} & = & \lambda f \in [\![k_1]\!].f^{k_2}. \end{array}$$

Assume $\Gamma \vdash A : B : *$. Define $\rho(= \rho_\Gamma)$ by

$$\begin{array}{rcll} \rho({}^{\square}x) & = & f^A & \text{if } (x{:}A) \in \Gamma; \\ & = & f^*, & \text{if } x \notin Dom(\Gamma); \end{array}$$

$$\rho({}^*x) \quad = \quad {}^*x.$$

Then $\rho \models \Gamma$, because if ${}^*x{:}A$ is in Γ, then $\Gamma \vdash A : *$ hence $[\![A]\!]_\rho \in [\![*]\!]_\rho = \mathrm{SAT}$ and therefore $\rho(x) = x \in [\![A]\!]_\rho$ by the definition of saturation; if ${}^\Box x{:}A$ is in Γ, then $\rho \models {}^\Box x : A$ since $\rho({}^\Box x) = f^A \in [\![A]\!]_\rho$.

1. By 5.3.21 one has $[\![A]\!]_\rho \in [\![B]\!]_\rho \in \mathrm{SAT}$ and therefore

   $$|A|[\vec{x} := \rho(\vec{x})] \in [\![B]\!]_\rho \subseteq \mathrm{SN}$$

   so $|A|[\vec{x} := \rho(\vec{x})] \in \mathrm{SN}$ and hence $|A| \in \mathrm{SN}$.

2. By (1) one has $|A| \in \mathrm{SN}$. From this it follows that $A \in \mathrm{SN}$, since for legal terms of $\lambda{\to}$ one has

   $$A \to_\beta A' \quad \Rightarrow \quad |A| \to_\beta |A'|.$$

   (This is not true for $\lambda\omega$; for example

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

   but the absolute values are both $\lambda x.x$.) ■

■

¿From the previous result we will derive that constructors in $\lambda\omega$ are strongly normalizing by interpreting kinds and constructors in $\lambda\omega$ as respectively types and elements in $\lambda{\to}$. The kind $*$ will be translated as a fixed $0{:}*$. The following examples give the intuition.

valid in $\lambda\omega$	translation valid in $\lambda{\to}$
$\alpha{:}* \vdash (\lambda\beta{:}*.\alpha) : (*\to*) : \Box$	$0{:}*, a{:}0 \vdash (\lambda b{:}0.a) : (0\to0) : *;$
$\alpha{:}*, f{:}(*\to*) \vdash (f\alpha \to f\alpha) : *$	$0{:}*, a{:}0, f{:}(0\to0) \vdash c^{0\to0\to0}(fa)(fa) : 0;$
$\alpha{:}* \vdash (\Pi\beta{:}*.\beta\to\alpha) : *$	$0{:}*, a{:}0 \vdash c^{0\to0\to0}c^0 a : 0.$

Definition 5.3.25. A map $()^- {:} \mathcal{T}_{1,2,3} \to \mathcal{T}_{0,1,2}$ is defined as follows:

$$\begin{array}{rcll}
(\Box)^- & = & *; & \\
(*)^- & = & 0; & \\
({}^\Box x)^- & = & {}^*x; & \\
(BA)^- & = & B^- A^-, & \qquad \text{if } \sharp(A) \neq 0, \\
 & = & B^-, & \qquad \text{else;}
\end{array}$$

$$\begin{array}{rcll}
(\lambda x{:}A.B)^- & = & (\lambda x^-{:}A^-.B^-), & \text{if } \sharp(A) \neq 0, \sharp(x) \neq 0, \\
 & = & B^-, & \text{else;} \\
(\Pi x{:}A.B)^- & = & (\Pi x^-{:}A^-.B^-), & \text{if } \sharp(A) = \sharp(B) = 2, \\
 & = & c^{0\to 0\to 0} A^- B^-, & \text{if } \sharp(A) = \sharp(B) = 1, \\
 & = & B^-[x^- := c^{A^-}], & \text{if } \sharp(A) = 2, \sharp(B) = 1, \\
 & = & B^-, & \text{else.}
\end{array}$$

For pseudo-contexts one defines the following (remember $\Gamma_0 = \{0{:}*, c{:}\bot\}$).

$$\begin{array}{rcl}
({}^{\Box}x{:}A)^- & = & x{:}A^-; \\
({}^{*}x{:}A)^- & = & <>; \\
(x_1{:}A_1, \ldots, x_n{:}A_n)^- & = & \Gamma_0, (x_1{:}A_1)^-, \ldots, (x_n{:}A_n)^-.
\end{array}$$

Then one can prove by induction on derivations

$$\Gamma \vdash_{\lambda\omega} A : B \ \&\ \sharp(A) \neq 0 \quad \Rightarrow \quad \Gamma^- \vdash_{\lambda\to} A^- : B^-.$$

Lemma 5.3.26.

1. *For $\sharp(A) \neq 0$ and $\sharp(a) = \sharp(x) \neq 0$ one has*

$$(A[x := a])^- \equiv A^-[x^- := a^-].$$

2. *For A legal in $\lambda\omega$ with $\sharp(A) = 1$ one has*

$$A \to_\beta B \ \Rightarrow \ A^- \to_\beta B^-.$$

Proof. Both by induction on the structure of A. ■ ■

Proposition 5.3.27.

$$\Gamma \vdash_{\lambda\omega} A : B : \Box \ \Rightarrow \ SN(A).$$

Proof.

$$\begin{array}{rcl}
\Gamma \vdash_{\lambda\omega} A : B : \Box & \Rightarrow & \Gamma^- \vdash_{\lambda\to} A^- : B^- : * \\
 & \Rightarrow & \mathrm{SN}(A^-)
\end{array}$$

$$\Rightarrow \quad \mathrm{SN}(A). \blacksquare$$

$\blacksquare$

Definition 5.3.28. Let $M \equiv (\lambda x{:}A.B)C$ be a legal $\lambda\omega$-term.

1. M is a *0-redex* if $\sharp(B) = 0$ and $\sharp(A) = 1$;
2. M is a *2-redex* if $\sharp(B) = 0$ and $\sharp(A) = 2$;
3. M is an *ω-redex* if $\sharp(B) = 1$ and $\sharp(A) = 2$;
4. A 2-λ is the first lambda occurrence in a 2-redex.

The three different kinds of redexes give rise to three different notions of contraction and reduction and will be denoted by $\rightarrow_0, \rightarrow_2$ and $\rightarrow_\omega$ respectively. Note that β-reduction is $0, 2, \omega$-reduction, in the obvious sense. We will prove that β-reduction of legal $\lambda\omega$-terms is SN by first proving the same for $2, \omega$-reduction.

Lemma 5.3.29. *Let $A, B \in T_0$ be legal terms in $\lambda\omega$. Then*

1. $(A \rightarrow_2 B) \;\Rightarrow\;$ *(number of 2-λs in A)>(number of 2-λs in B).*

2. $(A \rightarrow_\omega B) \;\Rightarrow\;$ *(number of 2-λs in A) =(number of 2-λs in B).*

3. $A \rightarrow_{2,\omega} B \;\Rightarrow\; |A| \equiv |B|$.

4. $A \rightarrow_0 B \;\Rightarrow\; |A| \rightarrow_\beta |B|$.

Proof. 1. Contracting a 2-redex $(\lambda x{:}A_0.B_0)C_0$ removes one 2-λ in A, removes A_0 and moves around C_0, possibly with duplications. A 2-λ is always part of $(\lambda x{:}A_1.B_1)$ with degree 0. A kind or constructor does not contain objects, in particular no 2-redexes. Therefore removing A_0, or moving around C_0 does not change the number of 2-λ's and we have the result.

2. Similarly.

3. If $M \equiv (\lambda x{:}A_0.B_0)C_0$ in A is a 2-redex, then C_0 is a constructor and $|M| \equiv |B_0|$. Remark that a constructor in an object M can occur only as subterm of A_1 occurring in $\lambda y{:}A_1.B_1$ in M. By the definition of $|-|$ constructors are removed in $|M|$. Therefore also $|B_0[x := C_0]| \equiv |B_0|$. We can conclude $|A| \equiv |B|$.

If $M \equiv (\lambda x{:}A_0.B_0)C_0$ in A is an ω-redex, then M and its contractum M' are both constructors. Therefore $|A| \equiv |B|$, again by the fact that constructors are eliminated by $|-|$.

4. If $M \equiv (\lambda x{:}A_0.B_0)C_0$ is a 0-redex with contractum $M' \equiv B_0[x := C_0]$, then $|M| \equiv (\lambda x.|B_0|)|C_0|$ and $|M'| \equiv |B_0[x := C_0]| \equiv |B_0|[x ::= |C_0|]$ as can be proved by induction on the structure of B_0. Therefore $|M| \to_\beta |M'|$. More generally $|A| \to_\beta |B|$ if $A \to_0 B$. ■

■

Lemma 5.3.30. *Suppose M is legal in $\lambda\omega$ and $\sharp(M) = 0$. Then M is strongly normalizing for*

1. *ω-reduction;*
2. *$2, \omega$-reduction.*

Proof. 1. M is not of the form $\Pi x{:}A.B$. Therefore it follows that either

$$M \equiv \lambda x_1{:}A_1 \cdots \lambda x_n{:}A_n.yB_1 \cdots B_m, n, m \geq 0.$$

or

$$M \equiv \lambda x_1{:}A_1 \cdots \lambda x_n{:}A_n.(\lambda y{:}C_0.C_1)B_1 \cdots B_m, n \geq 0, m \geq 1.$$

In the second case $\sharp(M) = \sharp(C_1)$. Therefore $(\lambda y{:}C_o.C_1)B_1$ is not an ω-redex. So in both cases ω-reduction starting with M must take place within the constructors that are subterms of the A_i, B_i or C_i, thus leaving the overall structure of M the same. Since β-reduction on constructors is SN by 5.3.27 it follows that ω-reduction on objects is SN.

2. Suppose

$$M_0 \to_{2,\omega} M_1 \to_{2,\omega} \cdots$$

is an infinite $2, \omega-$reduction. By 5.3.29 (1), (2) it follows that after some steps we have

$$M_k \to_\omega M_{k+1} \to_\omega \cdots$$

which is impossible by (1). ■

■

Corollary 5.3.31. *Suppose $\sharp(A) = 0$ and SN($|A|$). Then SN(A).*

Proof. An infinite reduction starting with A must by 5.3.30 2 be of the form

$$A \twoheadrightarrow_{2,\omega} A_1 \rightarrow_0 A_2 \twoheadrightarrow_{2,\omega} A_3 \rightarrow_0 A_4 \twoheadrightarrow_{2,\omega} \cdots .$$

But then by 5.3.29 3,4 we have

$$|A| \equiv |A_1| \rightarrow_\beta |A_2| \equiv |A_3| \rightarrow_\beta |A_4| \equiv \cdots .$$

contradicting SN($|A|$). ∎ ∎

Proposition 5.3.32.

$$\Gamma \vdash_{\lambda\omega} A : B \;\;\Rightarrow\;\; SN(A) \,\&\, SN(B).$$

Proof. If $\Gamma \vdash_{\lambda\omega} A : B : *$, then $\sharp(A) = 0$ by 5.2.28 and SN($|A|$) by 5.3.24(1) hence SN(A) by 5.3.31; also $\Gamma \vdash_{\lambda\omega} B : * : \square$ and therefore by 5.3.27 one has SN(B). If on the other hand $\Gamma \vdash_{\lambda\omega} A : B : \square$, then SN($A$) by 5.3.27 and SN($B$) since B is in nf by 5.3.17 (1). ∎ ∎

Theorem 5.3.33 (Strong normalization for the λ-cube). *For all systems in the λ-cube one has the following:*

1. $\Gamma \vdash A : B \;\;\Rightarrow\;\; SN(A) \,\&\, SN(B)$.
2. $x_1{:}A_1, \ldots, x_n{:}A_n \vdash B : C \;\;\Rightarrow\;\; A_1, \ldots, A_n, B, C$ *are SN.*

Proof. 1. It is sufficient to prove this for the strongest system λC and hence by 5.3.15 for $\lambda\omega$. This is done in 5.3.32.

2. By induction on derivations, using (1). ∎

∎

5.4 Representing logics and data-types

In this section eight systems of intuitionistic logic will be introduced that correspond in some sense to the systems in the λ-cube. The systems are the following; there are four systems of proposition logic and four systems of many-sorted predicate logic.

PROP	proposition logic;
PROP2	second-order proposition logic;
PROP$\underline{\omega}$	weakly higher-order proposition logic;
PROPω	higher-order proposition logic;
PRED	predicate logic;
PRED2	second-order predicate logic;
PRED$\underline{\omega}$	weakly higher-order predicate logic;
PREDω	higher-order predicate logic.

All these systems are minimal logics in the sense that the only logical operators are $\rightarrow$ and $\forall$. However, for the second- and higher-order systems the operators $\neg, \&, \vee$ and $\exists$, as well as Leibniz's equality, are all definable, see 5.4.17. Weakly higher-order logics have variables for higher-order propositions or predicates but no quantification over them; a higher-order proposition has lower order propositions as arguments. Classical versions of the logics in the upper plane are obtained easily (by adding as axiom $\forall\alpha.\neg\neg\alpha{\rightarrow}\alpha$). The systems form a cube as shown in the following Figure. 3.

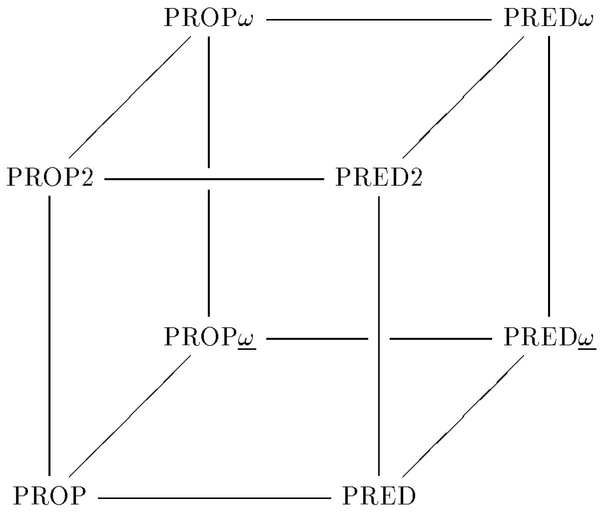


Fig. 3. The logic-cube.

This cube will be referred to as the logic-cube. The orientation of the logic-cube as drawn is called the standard orientation. Each system L_i on the logic-cube corresponds to the system λ_i on the λ-cube on the corresponding vertex (both cubes in standard orientation). The edges of the logic-cube represent inclusions of systems in the same way as on the λ-cube.

A formula A in a logic L_i on the logic-cube can be interpreted as a type $[\![A]\!]$ in the corresponding λ_i on the λ-cube. The transition $A \mapsto [\![A]\!]$ is called the *propositions-as-types* interpretation of de Bruijn (1970) and Howard (1980), first formulated for extensions of PRED and λP. The method has been extended by Martin-Löf (1984) who added to λP types $\Sigma x{:}A.B$ corresponding to (strong) constructive existence and a constructor $=_A\ : A{\rightarrow}A{\rightarrow}*$ corresponding to equality on a type A. Since Martin-Löf's principal objective is to give a constructive foundation of mathematics, he does not consider the impredicative rules $(\Box, *)$.

The propositions-as-types interpretation satisfies the following soundness result: if A is provable in PRED, then $[\![A]\!]$ is inhabited in λP. In fact, an inhabitant of $[\![A]\!]$ in λP can be found canonically from a proof of A in PRED; different proofs of A are interpreted as different terms of type $[\![A]\!]$. The interpretation has been extended to several other systems, see e.g. Stenlund (1972), Martin-Löf (1984) and Luo (1990). In Geuvers (1988) it is verified that for all systems L_i on the logic-cube soundness holds with respect to the corresponding system λ_i on the λ-cube: if A is provable in L_i, then $[\![A]\!]$ is inhabited in λ_i. Barendsen (1989) verifies that a proof D of such A can be canonically translated to $[\![D]\!]$ being an inhabitant of $[\![A]\!]$.

After seeing Geuvers (1988), it was realized by Berardi (1988a), (1990) that the systems in the logic-cube can be considered as PTSs. Doing this, the propositions-as-types interpretation obtains a simple canonical form. We will first give a description of PRED in its usual form and then in its form as a PTS.

The soundness result for the propositions-as-type interpretation raises the question whether one has also completeness in the sense that if a formula A of a logic L_i is such that $[\![A]\!]$ is inhabited in λ_i, then A is provable in L_i. For the proposition logics this is trivially true. For PRED completeness with respect to λP is proved in Martin-Löf (1971), Barendsen and Geuvers (1989) and Berardi (1990) (see also Swaen (1989)). For PREDω completeness with respect to λC fails, as is shown in Geuvers (1989) and Berardi (1989).

This subsection ends with a representation of data types in $\lambda 2$. The method is due to Leivant (1983) and coincides with an algorithm given later by Böhm and Berarducci (1985) and by Fokkinga (1987). Some results are stated about the representability of computable functions on data types represented in $\lambda 2$.

Many sorted predicate logic

Many sorted predicate logic will be introduced in its minimal form: formulas are built up from atomic ones using only $\rightarrow$ and $\forall$ as logical operators.

Definition 5.4.1.

1. The notion of a *many sorted structure* will be defined by an example. The following sequence is a typical many sorted structure

$$\mathcal{A} = \langle A, B, f, g, P, Q, c \rangle,$$

where

A, B are non-empty sets, the *sorts* of $\mathcal{A}$
$f : (A \rightarrow (A \rightarrow A))$ and $g : A \rightarrow B$ are functions;
$P \subseteq A$ and $Q \subseteq A \times B$ are relations;
$c \in A$ is a constant.

The name 'sorts' for A and B is standard terminology; in the context of PTSs it is better to call these the 'types' of $\mathcal{A}$.

2. The *signature* of $\mathcal{A}$ is $\langle 2; \langle 1,1,1\rangle, \langle 1,2\rangle; \langle 1\rangle, \langle 1,2\rangle; 1\rangle$ stating that there are two sorts; two functions, the first of which has signature $\langle 1,1,1\rangle$, i.e. having as input two elements of the first sort and as output an element of the first sort, the second of which has signature $\langle 1,2\rangle$, i.e. having an element of the first sort as input and an element of the second sort as output; etc.

Definition 5.4.2. Given the many sorted structure $\mathcal{A}$ of 5.4.1 the *language* $L_{\mathcal{A}}$ *of (minimal) many sorted predicate logic over* $\mathcal{A}$ is defined as follows. In fact this language depends only on the signature of $\mathcal{A}$.

1. $L_{\mathcal{A}}$ has the following special symbols.
   - $\mathbf{A}, \mathbf{B}$ sort symbols;
   - $\mathbf{f}, \mathbf{g}$ function symbols;
   - $\mathbf{P}, \mathbf{Q}$ relation symbols;
   - $\mathbf{c}$ constant symbol.

2. The set of variables of $L_{\mathcal{A}}$ is
$$V = \{x^{\mathbf{A}} \mid x \text{ variable}\} \cup \{x^{\mathbf{B}} \mid x \text{ variable}\}.$$

3. The set of terms of sort A and of sort B, notation $\text{Term}_{\mathbf{A}}$ and $\text{Term}_{\mathbf{B}}$ respectively, are defined inductively as follows:
   - $x^{\mathbf{A}} \in \text{Term}_{\mathbf{A}}$, $x^{\mathbf{B}} \in \text{Term}_{\mathbf{B}}$;
   - $\mathbf{c} \in \text{Term}_{\mathbf{A}}$;
   - $s \in \text{Term}_{\mathbf{A}}$ and $t \in \text{Term}_{\mathbf{A}} \ \Rightarrow\ \mathbf{f}(s,t) \in \text{Term}_{\mathbf{A}}$;
   - $s \in \text{Term}_{\mathbf{A}} \ \Rightarrow\ \mathbf{g}(s) \in \text{Term}_{\mathbf{B}}$.

4. The set of formulae of $L_{\mathcal{A}}$, notation Form, is defined inductively as follows:
   - $s \in \text{Term}_{\mathbf{A}} \ \Rightarrow\ \mathbf{P}(s) \in \text{Form}$;
   - $s \in \text{Term}_{\mathbf{A}}, t \in \text{Term}_{\mathbf{B}} \ \Rightarrow\ \mathbf{Q}(s,t) \in \text{Form}$;
   - $\varphi \in \text{Form}, \psi \in \text{Form} \ \Rightarrow\ (\varphi \to \psi) \in \text{Form}$;
   - $\varphi \in \text{Form} \ \Rightarrow\ (\forall x^{\mathbf{A}}.\varphi) \in \text{Form}$ and $(\forall x^{\mathbf{B}}.\varphi) \in \text{Form}$.

Definition 5.4.3. Let $\mathcal{A}$ be a many sorted structure. The *(minimal) many sorted predicate logic over* $\mathcal{A}$, notation $\text{PRED} = \text{PRED}_{\mathcal{A}}$, is defined as follows. If Δ is a set of formulae, then $\Delta \vdash \varphi$ denotes that φ is derivable from the assumptions Δ. This notion is defined inductively as follows (C ranges over A and B, and the corresponding $\mathbf{C}$ over $\mathbf{A}, \mathbf{B}$):

$$\begin{array}{rcl} \varphi \in \Gamma & \Rightarrow & \Gamma \vdash \varphi \\ \Gamma \vdash \varphi \to \psi, \Gamma \vdash \varphi & \Rightarrow & \Gamma \vdash \psi \\ \Gamma, \varphi \vdash \psi & \Rightarrow & \Gamma \vdash \varphi \to \psi \\ \Gamma \vdash \forall x^{\mathbf{C}}.\varphi,\ t \in \mathrm{Term}_{\mathbf{C}} & \Rightarrow & \Gamma \vdash \varphi[x := t] \\ \Gamma \vdash \varphi,\ x^{\mathbf{C}} \notin FV(\Gamma) & \Rightarrow & \Gamma \vdash \forall x^{\mathbf{C}}.\varphi, \end{array}$$

where $[x := t]$ denotes substitution of t for x and FV is the set of free variables in a term, formula or collection of formulae. For $\varnothing \vdash \varphi$ one writes simply $\vdash \varphi$ and one says that φ is a *theorem.*

These rules can be remembered best in the following natural deduction form.

$$\frac{\varphi \to \psi \quad \varphi}{\psi}\,; \qquad \frac{\begin{array}{c} \cancel{\varphi} \\ \vdots \\ \psi \end{array}}{\varphi \to \psi}\,;$$

$$\frac{\forall x^{\mathbf{C}}.\varphi}{\varphi[x := t]}\,,\ t \in \mathrm{term}_{\mathbf{C}}; \qquad \frac{\varphi}{\forall x^{\mathbf{C}}\varphi}\,,\ x \text{ not free in the assumptions.}$$

Some examples of terms, formulae and theorems are the following. The expressions $x^{\mathbf{A}}, \mathbf{c}, \mathbf{f}(x^{\mathbf{A}}, \mathbf{c})$ and $\mathbf{f}(\mathbf{c}, \mathbf{c})$ are all in $\mathrm{Term}_{\mathbf{A}}$; $\mathbf{g}(x^{\mathbf{A}})$ is in $\mathrm{Term}_{\mathbf{B}}$. Moreover

$$\forall x^{\mathbf{A}} \mathbf{P}(\mathbf{f}(x^{\mathbf{A}}, x^{\mathbf{A}})), \tag{1}$$

$$\forall x^{\mathbf{A}} [\mathbf{P}(x^{\mathbf{A}}) \to \mathbf{P}(\mathbf{f}(x^{\mathbf{A}}, \mathbf{c}))], \tag{2}$$

$$\forall x^{\mathbf{A}} [\mathbf{P}(x^{\mathbf{A}}) \to \mathbf{P}(\mathbf{f}(x^{A}, \mathbf{c}))] \to \forall x^{\mathbf{A}} \mathbf{P}(x^{\mathbf{A}}) \to \mathbf{P}(\mathbf{f}(\mathbf{c}, \mathbf{c})) \tag{3}$$

are formulae. The formula (3) is even a theorem. A derivation of (3) is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cancel{\forall x^{\mathbf{A}}[\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(f(x^{\mathbf{A}},\mathbf{c}))]}\,2}{\mathbf{P}(\mathbf{c})\to\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))} \qquad \dfrac{\cancel{\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})}\,1}{\mathbf{P}(\mathbf{c})}}{\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}}{\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}\,1}{\forall x^{\mathbf{A}}[\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(x^{\mathbf{A}},\mathbf{c}))]\to\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}\,2$$

the numbers 1, 2 indicating when a cancellation of an assumption is being made. A simpler derivation of the same formula is

$$\dfrac{\cancel{\forall x^{\mathbf{A}}[\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(x^{\mathbf{A}},\mathbf{c}))]}\,2 \qquad \dfrac{\dfrac{\cancel{\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})}\,1}{\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}}{\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}\,1}{\forall x^{\mathbf{A}}[\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(x^{\mathbf{A}},\mathbf{c}))]\to\forall x^{\mathbf{A}}\mathbf{P}(x^{\mathbf{A}})\to\mathbf{P}(\mathbf{f}(\mathbf{c},\mathbf{c}))}\,2$$

Now we will explain, first somewhat informally, the *propositions-as-types* interpretation from PRED into λP. First one needs a context corresponding to the structure $\mathcal{A}$. This is $\Gamma_{\mathcal{A}}$ defined as follows (later $\Gamma_{\mathcal{A}}$ will be defined a bit differently):

$$\begin{array}{rcl}\Gamma_{\mathcal{A}} & = & A{:}*, B{:}*,\\ & & P{:}A\to *, Q{:}A\to B\to *,\\ & & f{:}A\to A\to A, g{:}A\to B,\\ & & c{:}A.\end{array}$$

For this context one has

$$\Gamma_{\mathcal{A}} \vdash c : A \tag{$0'$}$$

$$\Gamma_{\mathcal{A}} \vdash (fcc) : A$$

$$\Gamma_{\mathcal{A}} \vdash \Pi x{:}A.P(fxx) : * \tag{$1'$}$$

$$\Gamma_{\mathcal{A}} \vdash \Pi x{:}A.(Px\to P(fxc)) : * \tag{$2'$}$$

$$\Gamma_{\mathcal{A}} \vdash (\Pi x{:}A.(Px\to P(fxc)))\to((\Pi x{:}A.Px)\to P(fcc)) : *. \tag{$3'$}$$

We see how the formulae (1)–(3) are translated as types. The inhabitants of $*$ have a somewhat 'ambivalent' behaviour: they serve both as sets (e.g. $A{:}*$) and as propositions (e.g. $Px : *$ for $x{:}A$). The fact that formulae are translated as types is called the *propositions-as-types* (or also *formulae-as-types*) interpretation. The provability of the formula (3) corresponds to the fact that the type in (3′) is inhabited. In fact

$$\Gamma_{\mathcal{A}} \vdash \lambda p{:}(\Pi x{:}A.(Px{\to}P(fxc))).\lambda q{:}(\Pi x{:}A.Px).pc(qc) :$$

$$\Pi p{:}(\Pi x{:}A.(Px{\to}P(fxc))).\Pi q{:}(\Pi x{:}A.Px).P(fcc).$$

A somewhat simpler inhabitant of the type in (3′), corresponding to the second proof of the formula (3) is

$$\lambda p{:}(\Pi x{:}A.(Px{\to}P(fxc))).\lambda q{:}(\Pi x{:}A.Px).q(fcc).$$

In fact, one has the following result that we state at this moment informally (and in fact not completely correct).

Theorem 5.4.4 (Soundness of the propositions-as-types interpretation). *Let $\mathcal{A}$ be a many sorted structure and let φ be a formula of $L_{\mathcal{A}}$. Suppose*

$$\vdash_{\text{PRED}} \varphi \textit{ with derivation } D;$$

then

$$\Gamma_{\mathcal{A}} \vdash_{\lambda\text{P}} [D] : [\varphi] : *,$$

where $[D]$ and $[\varphi]$ are canonical translations of respectively φ and D.

Now it will be shown that up to ‘isomorphism’ PRED can be viewed as a PTS. This PTS will be called λPRED. The map $\varphi \mapsto [\varphi]$ can be factorized as the composition of an isomorphism PRED $\to\lambda$PRED and a canonical forgetful homomorphism λPRED $\to\lambda$P.

Definition 5.4.5 (Berardi (1988a)). PRED considered as a PTS, notation λPRED, is determined by the following specification:

$\mathcal{S}$	$*^s, *^p, *^f, \Box^s, \Box^p$
$\mathcal{A}$	$*^s : \Box^s, *^p : \Box^p$
$\mathcal{R}$	$(*^p, *^p), (*^s, *^p), (*^s, \Box^p),$
	$(*^s, *^s, *^f), (*^s, *^f, *^f)$

Some explanations are called for. The sort $*^s$ is for sets (the ‘sorts’ of the many sorted logic). The sort $*^p$ is for propositions (the formulae of the logic will become elements of $*^p$). The sort $*^f$ is for first-order functions between the sets in $*^s$. The sort $\Box^s$ contains $*^s$ and the sort $\Box^p$ contains $*^p$. (There is no $\Box^f$, otherwise it would be allowed to have free variables for function spaces.)

The rule $(*^p, *^p)$ allows the formation of implication of two formulae:

$$\varphi{:}{*}^p, \psi{:}{*}^p \vdash (\varphi{\rightarrow}\psi) \equiv (\Pi x{:}\varphi.\psi) : {*}^p.$$

The rule $({*}^s, {*}^p)$ allows quantification over sets:

$$A{:}{*}^s, \varphi{:}{*}^p \vdash (\forall x^{\mathbf{A}}.\varphi) \equiv (\Pi x{:}A.\varphi) : {*}^p.$$

The rule $({*}^s, \Box^p)$ allows the formation of first-order predicates:

$$A{:}{*}^s \vdash (A{\rightarrow}{*}^p) \equiv (\Pi x{:}A.{*}^p) : \Box^p;$$

hence

$$A{:}{*}^s, P{:}A{\rightarrow}{*}^p, x{:}A \vdash Px : {*}^p,$$

i.e. P is a predicate over the set A.

The rule $({*}^s, {*}^s, {*}^f)$ allows the formation of a function space between the basic sets in ${*}^s$:

$$A{:}{*}^s, B{:}{*}^s \vdash (A{\rightarrow}B) : {*}^f;$$

the rule $({*}^s, {*}^f, {*}^f)$ allows the formation of curried functions of several arguments in the basic sets:

$$A{:}{*}^s \vdash (A{\rightarrow}(A{\rightarrow}A)) : {*}^f.$$

This makes it possible to have for example $g{:}A{\rightarrow}B$ and $f{:}(A{\rightarrow}(A{\rightarrow}A))$ in a context.

Now it will be shown formally that λPRED is able to simulate the logic PRED. Terms, formulae and derivations of PRED are translated into terms of λ PRED. Terms become elements, formulae become types and a derivation of a formula φ becomes an element of the type corresponding to φ.

Definition 5.4.6. Let $\mathcal{A}$ be as in 5.4.1. The *canonical context* corresponding to $\mathcal{A}$, notation $\Gamma_{\mathcal{A}}$, is defined by

$$\begin{aligned} \Gamma_{\mathcal{A}} = \quad & A{:}{*}^s, B{:}{*}^s, \\ & P{:}(A{\rightarrow}{*}^p), Q{:}(A{\rightarrow}B{\rightarrow}{*}^p), \\ & f{:}(A{\rightarrow}(A{\rightarrow}A)), g{:}(A{\rightarrow}B), \\ & c{:}A. \end{aligned}$$

Given a term $t \in L_{\mathcal{A}}$, the canonical translation of t, notation $[\![t]\!]$, and the canonical context for t, notation Γ_t, are inductively defined as follows:

t	$[\![t]\!]$	Γ_t
$x^{\mathbf{C}}$	x	$x : C$
$\mathbf{c}$	c	$\langle\rangle$
$\mathbf{f}(s, s')$	$f[\![s]\!][\![s']\!]$	$\Gamma_s \cup \Gamma_{s'}$
$\mathbf{g}(s)$	$g[\![s]\!]$	Γ_s

Given a a formula φ in $L_{\mathcal{A}}$, the canonical translation of φ, notation $[\![\varphi]\!]$, and the canonical context for φ, notation Γ_φ, are inductively defined as follows:

φ	$[\![\varphi]\!]$	Γ_φ
$\mathbf{P}(t)$	$P[\![t]\!]$	Γ_t
$\mathbf{Q}(s, t)$	$Q[\![s]\!][\![t]\!]$	$\Gamma_s \cup \Gamma_t$
$\varphi_1 \to \varphi_2$	$[\![\varphi_1]\!] \to [\![\varphi_2]\!]$	$\Gamma_{\varphi_1} \cup \Gamma_{\varphi_2}$
$\forall x^{\mathbf{C}}.\psi$	$\Pi x{:}C.[\![\psi]\!]$	$\Gamma_\psi - \{x{:}C\}$

Lemma 5.4.7.

1. $t \in \mathit{Term}_{\mathbf{C}} \;\;\Rightarrow\;\; \Gamma_{\mathcal{A}}, \Gamma_t \vdash_{\lambda\mathrm{PRED}} [\![t]\!] : C.$
2. $\varphi \in \mathit{Form} \;\;\Rightarrow\;\; \Gamma_{\mathcal{A}}, \Gamma_\varphi \vdash_{\lambda\mathrm{PRED}} [\![\varphi]\!] : *^p.$

Proof. By an easy induction. ■ ∎

In order to define the canonical translation of derivations, it is useful to introduce some notation. The following definition is a reformulation of 5.4.3, now giving formal notations for derivations.

Definition 5.4.8. In PRED the notion 'D is a *derivation showing* $\Delta \vdash \varphi$', notation $D : (\Delta \vdash \varphi)$, is defined as follows.

$$\begin{array}{rcl}
\varphi \in \Delta & \Rightarrow & P_\varphi : (\Delta \vdash \varphi); \\
D_1 : (\Delta \vdash \varphi \to \psi), D_2 : (\Delta \vdash \varphi) & \Rightarrow & (D_1 D_2) : (\Delta \vdash \psi); \\
D : (\Delta, \varphi \vdash \psi) & \Rightarrow & (I\varphi.D) : (\Delta \vdash \varphi \to \psi); \\
D : (\Delta \vdash \forall x^{\mathbf{C}}.\varphi), t \in \mathrm{Term}_{\mathbf{C}} & \Rightarrow & (Dt) : (\Delta \vdash \varphi[x := t]); \\
D : (\Delta \vdash \varphi), x^{\mathbf{C}} \notin FV(\Delta) & \Rightarrow & (Gx^{\mathbf{C}}.D) : (\Delta \vdash \forall x^{\mathbf{C}}.\varphi).
\end{array}$$

Here $\mathbf{C}$ is $\mathbf{A}$ or $\mathbf{B}$, P stands for 'projection', $I\varphi$ stands for introduction and has a binding effect on φ and $Gx^{\mathbf{C}}$ stands for 'generalization' (over C) and has a binding effect on $x^{\mathbf{C}}$.

Definition 5.4.9.

1. Let $\Delta = \{\varphi_1, \ldots, \varphi_n\} \subseteq$ Form. Then the *canonical translation* of Δ, notation Γ_Δ, is the context defined by

$$\Gamma_\Delta = \Gamma_{\varphi_1} \cup \cdots \cup \Gamma_{\varphi_n}, x_{\varphi_1}{:}[\![\varphi_1]\!], \cdots, x_{\varphi_n}{:}[\![\varphi_n]\!].$$

2. For $D : (\Delta \vdash \varphi)$ in PRED the canonical translation of D, notation $[\![D]\!]$, and the canonical context for D, notation Γ_D, are inductively defined as follows:

D	$[\![D]\!]$	Γ_D
P_φ	x_φ	$\langle\rangle$
$D_1 D_2$	$[\![D_1]\!][\![D_2]\!]$	$\Gamma_{D_1} \cup \Gamma_{D_2}$
$I\varphi.D_1$	$\lambda x_\varphi{:}[\![\varphi]\!].[\![D_1]\!]$	$\Gamma_{D_1} - \{x_\varphi{:}[\![\varphi]\!]\}$
Dt	$[\![D]\!][\![t]\!]$	$\Gamma_D \cup \Gamma_t$
$Gx^{\mathbf{C}}.D$	$\lambda x{:}C.[\![D]\!]$	$\Gamma_D - \{x{:}C\}$

The following result is valid for the structure $\mathcal{A}$ as given in 5.4.1.

Lemma 5.4.10.

$$D : (\Delta \vdash_{\mathrm{PRED}} \varphi) \;\Rightarrow\; \Gamma_{\mathcal{A}}, \Gamma_\Delta \cup \Gamma_\varphi \cup \Gamma_D \vdash_{\lambda\mathrm{PRED}} [\![D]\!] : [\![\varphi]\!].$$

Proof. By induction on the derivation in PRED. ■ ∎

Barendsen (1989) observed that in spite of Lemma 5.4.10 one has in general for e.g. a sentence φ (i.e. $FV(\varphi) = \varnothing$)

$$\vdash_{\text{PRED}} \varphi \quad \not\Rightarrow \quad \exists A\, [\Gamma_{\mathcal{A}} \vdash_{\lambda\text{PRED}} A : [\![\varphi]\!]].$$

The point is that in ordinary (minimal, intuitionistic or classical) logic it is always assumed that the universes (the sorts $A, B, \ldots$) of the structure $\mathcal{A}$ are supposed to be non-empty. For example

$$(\forall x^{\mathbf{A}}.(Px \rightarrow Q)) \rightarrow (\forall x^{\mathbf{A}}.Px) \rightarrow Q)$$

is provable in PRED, but only valid in structures with $A \neq \varnothing$. In so-called *free logic* one allows also structures with empty domains. This logic has been axiomatized by Peremans (1949) and Mostowski (1951). The system λPRED is flexible enough to cover also this free logic. The following extended context $\Gamma^{+}_{\mathcal{A}}$ explicitly states that the domains in question are not empty.

Definition 5.4.11. Given a many sorted structure $\mathcal{A}$ as in 5.4.1, the *extended context*, notation $\Gamma^{+}_{\mathcal{A}}$, is defined by $\Gamma^{+}_{\mathcal{A}} = \Gamma_{\mathcal{A}}, a{:}A, b{:}B$.

Not only there is a sound interpretation of PRED into λPRED, there is also a converse. In order to prove this completeness the following lemma, due to Fujita and Tonino, is needed.

Lemma 5.4.12. *Suppose* $\Gamma \vdash_{\lambda\text{PRED}} A : B : *^{p}$. *Then there is a many sorted structure* $\mathcal{A}$, *a set of formulae* $\Delta \subseteq L_{\mathcal{A}}$, *a formula* $\varphi \in L_{\mathcal{A}}$ *and a derivation* D *such that*

$$\begin{array}{l} \Gamma \equiv \Gamma_{\mathcal{A}}, \Gamma_{\Delta} \cup \Gamma_{\varphi} \cup \Gamma_{D}, \\ A \equiv [\![D]\!], B \equiv [\![\varphi]\!] \\ D : \Delta \vdash_{\text{PRED}} \varphi. \end{array}$$

Proof. See Fujita and Tonino (1991). ■ ∎

Corollary 5.4.13.

1. *Let* φ *be a formula and* Δ *be a set of formulae of* $L_{\mathcal{A}}$. *Then*

$$D : \Delta \vdash_{\text{PRED}} \varphi \quad \Leftrightarrow \quad \Gamma_{\mathcal{A}}, \Gamma_{\Delta} \cup \Gamma_{\varphi} \cup \Gamma_{D} \vdash_{\lambda\text{PRED}} [\![D]\!] : [\![\varphi]\!].$$

2. *Let* $\Delta \cup \{\varphi\}$ *be a set of sentences of* $L_{\mathcal{A}}$. *Then*

$$\Delta \vdash_{\mathrm{PRED}} \varphi \quad \Leftrightarrow \quad \exists M[\Gamma^+_{\mathcal{A}}, \Gamma_\Delta \vdash_{\lambda\mathrm{PRED}} M : [\![\varphi]\!]].$$

3. *Let φ be a sentence of $L_{\mathcal{A}}$. Then*

$$\vdash_{\mathrm{PRED}} \varphi \quad \Leftrightarrow \quad \exists M[\Gamma^+_{\mathcal{A}} \vdash_{\lambda\mathrm{PRED}} M : [\![\varphi]\!]].$$

Proof. 1. By 5.4.10 and 5.4.12 and the fact that $[\![-]\!]$ is injective on derivations and formulae.

2. If the members of Δ and φ are without free variables, then

$$D : (\Delta \vdash_{\mathrm{PRED}} \varphi) \quad \Leftrightarrow \quad \Gamma_{\mathcal{A}}, \Gamma_\Delta \cup \Gamma_D \vdash_{\lambda\mathrm{PRED}} [\![D]\!] : [\![\varphi]\!].$$

A statement in Γ_D is of the form $x : C$. Since $\Gamma^+_{\mathcal{A}} \vdash a : A, b : B$ one has

$$\begin{array}{rcl} \Delta \vdash_{\mathrm{PRED}} \varphi & \Leftrightarrow & \exists D[D : (\Delta \vdash_{\mathrm{PRED}} \varphi)] \\ & \Leftrightarrow & \exists D[\Gamma_{\mathcal{A}}, \Gamma_\Delta \cup \Gamma_D \vdash_{\lambda\mathrm{PRED}} [\![D]\!] : [\![\varphi]\!]] \\ & \Leftrightarrow & \exists M[\Gamma^+_{\mathcal{A}}, \Gamma_\Delta \vdash_{\lambda\mathrm{PRED}} M : [\![\varphi]\!]]. \end{array}$$

(For the last ($\Rightarrow$) take $M \equiv [\![D]\!][x, y := a, b]$; for ($\Leftarrow$) use Lemma 5.4.12.)

3. By (2), taking $\Delta = \varnothing$. ■

∎

Now that it has been established that PRED and λPRED are 'isomorphic', the propositions-as-types interpretation from PRED to λP can be factorized in two simple steps: from PRED to λPRED via the isomorphism and from λPRED to λP via a canonical forgetful map.

Definition 5.4.14 (Propositions-as-types interpretation).

1. Define the forgetful map $|-|$: term(λPRED)$\rightarrow$term(λP) by deleting all superscripts in $*$ and $\Box$, so:

$$\begin{array}{rcl} *^s & \mapsto & * \\ *^p & \mapsto & * \\ *^f & \mapsto & * \\ \Box^s & \mapsto & \Box \\ \Box^p & \mapsto & \Box. \end{array}$$

E.g. $|\lambda x{:}*^p.x| \equiv \lambda x{:}*.x$. Write $|\Gamma| \equiv \langle x_1{:}|A_1|, \ldots\rangle$ for $\Gamma \equiv \langle x_1{:}A_1, \ldots\rangle$.

2. Let $\mathcal{A}$ be a signature and let t, φ, Δ and D be respectively a term, a formula, a set of formulae and a derivation in PRED formulated in $L_{\mathcal{A}}$. Write

$$\begin{array}{rcl}[t] & = & |[\![t]\!]|;\\ [\varphi] & = & |[\![\varphi]\!]|;\\ [D] & = & |[\![D]\!]|;\\ [\Delta] & = & |\Gamma^+_{\mathcal{A}}|, |\Gamma_\Delta|.\end{array}$$

Corollary 5.4.15 (Soundness for the propositions-as-types interpretation).

1. $\Gamma \vdash_{\lambda\mathrm{PRED}} A : B \;\; \Rightarrow \;\; |\Gamma| \vdash_{\lambda\mathbf{P}} |A| : |B|,$
2. *For sentences Δ and φ in $L_{\mathbf{A}}$ one has*

$$D{:}\Delta \vdash_{\mathrm{PRED}} \varphi \;\; \Rightarrow \;\; [\Delta] \vdash_{\lambda\mathbf{P}} M : [\varphi], \text{ for some } M.$$

Proof. 1. By a trivial induction on derivations in λPRED.

2. By 5.4.13(2) and 1. ■

■

Now that we have seen the equivalence between PRED and λPRED, the other systems on the logic cube will be described directly as a PTS and not as a more traditional logical system. In this way we obtain the so called L-cube isomorphic to the logic-cube.

Definition 5.4.16.

1. The systems λPROP, λPROP2, λPROP$\underline{\omega}$ and λPROPω are the PTSs specified as follows:

λPROP

$\mathcal{S}$	$*^p, \Box^p$
$\mathcal{A}$	$*^p : \Box^p$
$\mathcal{R}$	$(*^p, *^p)$

$\lambda\mathrm{PROP2} = \lambda\mathrm{PROP} + (\Box^p, *^p)$.

λPROP2

$\mathcal{S}$	$*^p, \Box^p$
$\mathcal{A}$	$*^p : \Box^p$
$\mathcal{R}$	$(*^p, *^p), (\Box^p, *^p)$

$\lambda\mathrm{PROP}\underline{\omega} = \lambda PROP + (\Box^p, \Box^p)$.

λPROP$\underline{\omega}$	$\mathcal{S}$	$*^p, \Box^p$
	$\mathcal{A}$	$*^p : \Box^p$
	$\mathcal{R}$	$(*^p, *^p), (\Box^p, \Box^p)$

λPROP$\omega = \lambda$PROP $+ (\Box^p, *^p) + (\Box^p, \Box^p)$.

λPROPω	$\mathcal{S}$	$*^p, \Box^p$
	$\mathcal{A}$	$*^p : \Box^p$
	$\mathcal{R}$	$(*^p, *^p), (\Box^p, *^p), (\Box^p, \Box^p)$

2. The systems λPRED, λPRED2, λPRED$\underline{\omega}$ and λPREDω are the PTS's specified as follows.

λPRED	$\mathcal{S}$	$*^p, *^s, *^f, \Box^p, \Box^s$
	$\mathcal{A}$	$*^p : \Box^p, *^s : \Box^s$
	$\mathcal{R}$	$(*^p, *^p), (*^s, *^p)$
		$(*^s, *^s, *^f), (*^s, *^f, *^f), (*^s, \Box^p)$

λPRED2 $= \lambda$PRED $+ (\Box^p, *^p)$.

λPRED2	$\mathcal{S}$	$*^p, *^s, *^f, \Box^p, \Box^s$
	$\mathcal{A}$	$*^p : \Box^p, *^s : \Box^s$
	$\mathcal{R}$	$(*^p, *^p), (*^s, *^p), (\Box^p, *^p)$
		$(*^s, *^s, *^f), (*^s, *^f, *^f), (*^s, \Box^p)$

λPRED$\underline{\omega} = \lambda$PRED $+ (\Box^p, \Box^p)$.

λPRED$\underline{\omega}$	$\mathcal{S}$	$*^p, *^s, *^f, \Box^p, \Box^s$
	$\mathcal{A}$	$*^p : \Box^p, *^s : \Box^s$
	$\mathcal{R}$	$(*^p, *^p), (*^s, *^p)$
		$(*^s, *^s, *^f), (*^s, *^f, *^f), (*^s, \Box^p), (\Box^p, \Box^p)$

λPRED$\omega = \lambda$PRED $+ (\Box^p, *^p) + (\Box^p, \Box^p)$.

λPREDω	$\mathcal{S}$	$*^p, *^s, *^f, \Box^p, \Box^s$
	$\mathcal{A}$	$*^p : \Box^p, *^s : \Box^s$
	$\mathcal{R}$	$(*^p, *^p), (*^s, *^p), (\Box^p, *^p)$
		$(*^s, *^s, *^f), (*^s, *^f, *^f), (*^s, \Box^p), (\Box^p, \Box^p)$

The eight systems form a cube as shown in the following figure 4.

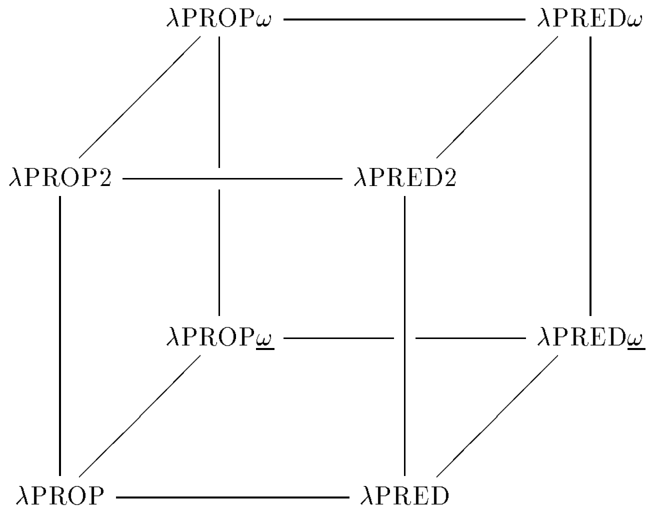


Fig. 4. The L-cube.

Since this description of the logical systems as PTSs is more uniform than the original one, we will considere only this L-cube, rather than the isomorphic one in fig. 3. In particular, fig. 4 displays the standard orientation of the L-cube and each system L_i (ranging over λPROP, λPRED etc.) corresponds to a unique system λ_i on the similar vertex in the λ-cube (in standard orientation).

Now it will be shown how in the upper plane of the L-cube the logical operators $\neg, \&, \vee$ and $\exists$ and also an equality predicate $=_L$ are definable. The relation $=_L$ is called *Leibniz' equality.*

Definition 5.4.17 (Second-order definability of the logical operations).

1. For $A, B{:}*^p$ define

$$\begin{array}{rcl} \bot & \equiv & (\Pi\beta{:}*^p.\beta); \\ \neg A & \equiv & (A{\to}\bot) \\ A\&B & \equiv & \Pi\gamma{:}*^p.(A{\to}B{\to}\gamma){\to}\gamma; \\ A \vee B & \equiv & \Pi\gamma{:}*^p.(A{\to}\gamma){\to}(B{\to}\gamma){\to}\gamma. \end{array}$$

2. For $A{:}*^p$ and $S{:}*^s$ define

$$\exists x{:}S.A \equiv \Pi\gamma{:}*^p.(\Pi x{:}S.(A{\to}\gamma)){\to}\gamma.$$

3. For $S{:}*^s$ and $x, y{:}S$ define

$$(x =_L y) \equiv \Pi P{:}S{\to}*^p.Px{\to}Py.$$

Note that the definition of & and $\vee$ make sense for systems extending λPROP2 and $\exists$ and $=_S$ for systems extending λPRED2. It is a good exercise to verify that the usual logical rules for $\&, \vee, \exists$ and $=_S$ are valid in the appropriate systems.

Example 5.4.18. We show how a part of first order Heyting Arithmetic (HA) can be done in λPRED. That is, we give a context $\Gamma_{\mathcal{A}}, \Gamma_{\Delta}$ such that $\Gamma_{\mathcal{A}}$ fixes the language of HA and Γ_{Δ} fixes a part of the axioms of HA. Take $\Gamma_{\mathcal{A}}$ to be

$$\begin{array}{rcl} N & : & *^s, \\ 0 & : & N, \\ S & : & N{\to}N, \\ + & : & N{\to}N{\to}N, \\ = & : & N{\to}N{\to}*^p\ . \end{array}$$

Take Γ_{Δ} to be

$$\begin{array}{rcl} tr & : & \Pi x, y, z : N.\ x = y \to y = z \to x = z, \\ sy & : & \Pi x, y : N.\ x = y \to y = x, \\ re & : & \Pi x : N.\ x = x, \\ a_1 & : & \Pi x, y : N.\ Sx = Sy \to x = y, \\ a_2 & : & \Pi x : N.\ x + 0 = x, \\ a_3 & : & \Pi x, y : N.\ x + Sy = S(x + y). \end{array}$$

Note that we do not have $a_4 : [\Pi x{:}N.\ Sx \neq 0]$ and $a_5 : [\Pi x{:}N.\ x \neq 0 \to \exists y{:}N.x = Sy]$, because the logic is minimal (We can't define $\neg$ and $\exists$ in first order logic.) Also we don't have an induction scheme for the natural numbers, which requires infinitely many axioms or one second order axiom $(a_6 : \Pi P{:}N{\to}*^p.P0 \to (\Pi x{:}N.Px \to P(Sx)) \to \Pi x{:}N.Px)$. One says that HA is not *finitely first order axiomatizable*. Finally, the atomic equality in λPRED is very weak, e.g. it doesn't satisfy the *substitution property*: if $\varphi(x)$ and $x = y$ hold, then $\varphi(y)$ holds. In second order predicate logic (λPRED2) HA can be axiomatized by adding a_6 and further a_4 and a_5 using the definable $\neg$ and $\exists$. Also the atomic $=$ can be replaced by the (definable) Leibniz equality on N, which does satisfy the substitution property.

Example 5.4.19. The structure of *commutative torsion groups* is not finitely nor infinitely first order axiomatizable. (This example is taken from Barwise (1977).) The manysorted structure of a commutative torsion group is $\langle A, =, \star, 0\rangle$ and it has as axioms:

$$\begin{array}{rcl} \forall x, y, z\ \ (x\star y)\star z &=& x\star(y\star z),\\ \forall x\ \ x\star 0 &=& x,\\ \forall x\ \ \exists y\ \ x\star y &=& 0,\\ \forall x, y\ \ x\star y &=& y\star x,\\ \forall x\ \ \exists n\geq 1\ \ nx &=& 0, \end{array}$$

where we write

$$nx \text{ for } \underbrace{x\star\cdots\star x}_{n}$$

If one tries to write the last formula in a first order form we get the following.

$$\forall x\ \ (x = 0 \vee 2x = 0 \vee \cdots)$$

So we obtain an 'infinitary' formula, which, can be shown to be not first order, by some use of the compactness theorem. A second order statement (as type) that expresses that the group has torsion is

$$\forall x{:}A\forall P{:}A{\to}{*}.[Px{\to}(\forall y{:}A.Py{\to}P(x\star y)){\to}P0].$$

Theorem 5.4.20 (Soundness of the propositions-as-types interpretation). *Let L_i be a system on the L-cube and let λ_i be the corresponding system on the λ-cube. The forgetful map $|\ .\ |$ that erases all superscripts in the $*$'s and $\square$'s satisfies the following*

$$\Gamma\vdash_{L_i} A : B : s\ \ \Rightarrow\ \ |\Gamma|\vdash_{\lambda_i}|A| : |B| : |s|.$$

Proof. By a trivial induction on the derivation in L_i. ■ ∎

As was remarked before, completeness for the propositions-as-types interpretation holds for PRED and λP, but not for PREDω and λC.

Theorem 5.4.21 (Berardi (1989); Geuvers (1989)). *Consider the similarity type of the structure $\mathcal{A} = \langle A\rangle$, i.e . there is one set without any relations. Then there is in the signature of $\mathcal{A}$ a sentence φ of* PREDω *such that*

$$\not\vdash_{\mathrm{PRED}\omega}\varphi$$

but for some M one has

$$\Gamma_{\mathcal{A}}\vdash_{\lambda\mathrm{C}} M : [\varphi].$$

Proof. (Berardi) Define

$$\mathrm{EXT}\ \equiv \Pi p, p'{:}{*}^p.[(p\leftrightarrow p'){\to}\Pi Q{:}{*}^p{\to}{*}^p.(Qp{\to}Qp')]$$

$$\varphi\equiv\ \mathrm{EXT}\to \text{'}A\text{ does not have exactly two elements'}$$

Obviously $\not\vdash_{\mathrm{PRED}\omega}\varphi$. Claim: interpreted in λC one has

EXT $\rightarrow$ 'if A is non-empty, then A is a type-free λ-model'.

The reason is that if $a{:}A$, then

$$\vdash (\lambda x{:}(A{\rightarrow}A).a) : ((A{\rightarrow}A){\rightarrow}A)$$

and always

$$\vdash (\lambda y{:}A.\lambda z{:}A.z) : (A{\rightarrow}(A{\rightarrow}A)),$$

therefore '$A \leftrightarrow (A{\rightarrow}A)$' and since '$A \cong A$' (i.e. there is a bijection from A to A), it follows by EXT that '$A \cong (A{\rightarrow}A)$', i.e.'$A$ is a type-free λ-model'.

By the claim A cannot have two elements, since only the trivial λ-model is finite. ■ █

Proof. (Geuvers) Consider in λPREDω the context Γ and type B defined as follows:

$$\Gamma \equiv A{:}*^s, c{:}A$$
$$B \equiv \Pi Q{:}(*^p{\rightarrow}*^p).\Pi q{:}*^p.(Q(\Pi x{:}A.q){\rightarrow}\exists q'{:}*^p.Q(q'{\rightarrow}q)).$$

Then B considered as formula is not derivable in λPREDω, but its translation $|B|$ in λC is inhabited, i.e.

1. $|\Gamma| \vdash_{\lambda C} C : |B|$, for some C.
2. $\Gamma \not\vdash_{\lambda\mathrm{PRED}\omega} C : B$, for all C.

As to 1, it is sufficient to construct a C_0 such that

$$A{:}*, c{:}A, Q{:}(*{\rightarrow}*), q{:}* \vdash C_0 : (Q(\Pi x{:}A.q){\rightarrow}\exists q'{:}*.Q(q'{\rightarrow}q)).$$

Now note that

$$Q(\Pi x{:}A.q) \equiv Q(A{\rightarrow}q)$$

and the type

$$\begin{aligned}&[Q(\Pi x{:}A.q){\rightarrow}\exists q'{:}*.Q(q'{\rightarrow}q)] \;\equiv\\ &\equiv\; Q(A{\rightarrow}q){\rightarrow}[\Pi\alpha{:}*.\Pi q'{:}*.(Q(q'{\rightarrow}q){\rightarrow}\alpha){\rightarrow}\alpha]\end{aligned}$$

is inhabited by

$$\lambda y{:}(Q(A{\rightarrow}q)).\lambda\alpha{:}*.\lambda f{:}(\Pi q'{:}*.(Q(q'{\rightarrow}q){\rightarrow}\alpha)).fAy.$$

As to 2, if $\Gamma \vdash_{\lambda\mathrm{PRED}\omega} C : B$, then also

$$\begin{aligned}&A{:}*^s, c{:}A, Q{:}(*^p{\rightarrow}*^p), q{:}*^p, r{:}(Q(\Pi x{:}A.q)), \alpha{:}*^p, t{:}(\Pi q'{:}*^p.Q(q'{\rightarrow}q){\rightarrow}\alpha)\\ &\vdash CQqr\alpha t : \alpha\end{aligned}$$

By considering the possible forms of the normal form of $CQqr\alpha t$ it can be shown that this is impossible. ■ █

The counterexample of Geuvers is shorter (and hence easier to formalize) than that of Berardi, but it is less intuitive.

As is well-known, logical deductions are subject to reduction, see e.g. Prawitz (1965) or Stenlund (1972). For example in PRED one has

and

If the deductions are represented in λPRED, then these reductions become ordinary β-reductions:

$$\begin{array}{rl} [\![(I\varphi.D_1)D_2]\!] \equiv & (\lambda x_\varphi{:}[\![\varphi]\!].[\![D_1]\!])[\![D_2]\!] \to_\beta \\ & [\![D_1]\!][x_\varphi := [\![D_2]\!]] \equiv [\![D_1[P_\varphi := D_2]]\!]; \\ [\![(Gx^{\mathbf{C}}.D)t]\!] \equiv & (\lambda x{:}C.[\![D]\!])[\![t]\!] \to_\beta \\ & [\![D]\!][x := [\![t]\!]] \equiv [\![D[x := t]]\!]. \end{array}$$

In fact the best way to define the notion of reduction for a logical system on the L-cube is to consider that system as a PTS subject to β-reductions.

Now it follows that reductions in all systems of the L-cube are strongly normalizing.

Corollary 5.4.22. *Deductions in a system on the L-cube are strongly normalizing.*

Proof. The propositions-as-types map

$$|\ \ | : L\text{-cube} \to \lambda\text{-cube}$$

preserves reduction; moreover the systems on the λ-cube are strongly normalizing. ■ ■

The following example again shows the flexibility of the notion of PTS.

Example 5.4.23 (Geuvers (1990)). The system of higher-order logic in Church (1940) can be described by the following PTS:

λHOL		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*, *), (\Box, *), (\Box, \Box)$

That is λHOL is $\lambda\omega$ plus $\Box : \Delta$. The sort $\Box$ represents the universe of domains and the sort $*$ represents the universe of formulae. The sort Δ and the rule $\Box : \Delta$ allow us to make declarations $A : \Box$ in the context. The system λHOL consists of a higher-order term language given by the sorts $* : \Box : \Delta$ and the rule $(\Box, \Box)$ (notice the similarity with $\lambda\!\to$) with a higher-order logic on top of it, given by the rules $(*, *)$ and $(\Box, *)$.

A sound interpretation of λPREDω in λHOL is determined by the map given by

$$\begin{array}{lcl} *^p & \mapsto & * \\ *^s & \mapsto & \Box \\ *^f & \mapsto & \Box \\ \Box^p & \mapsto & \Box \\ \Box^s & \mapsto & \Delta. \end{array}$$

Geuvers (1990) proves that λHOL is isomorphic with the following extended version of λPREDω,

$$\lambda\text{PRED}\omega! \quad \boxed{\begin{array}{ll} \mathcal{S} & *^p, *^s, \Box^p, \Box^s \\ \mathcal{A} & *^p : \Box^p, *^s : \Box^s \\ \mathcal{R} & (*^p, *^p), (*^s, *^p), (\Box^p, *^p) \\ & (*^s, *^s), (\Box^p, *^s), (*^s, \Box^p), (\Box^p, \Box^p) \end{array}}$$

where isomorphic means that there are mappings $F : (\lambda\text{PRED}\omega!) \rightarrow (\lambda\text{HOL})$ and $G : (\lambda\text{HOL}) \rightarrow (\lambda\text{PRED}\omega!)$ such that $G \circ F = \text{Id}$ and $F \circ G = \text{Id}$. (Here the systems (λHOL) and (λPREDω!) are identified with the set of derivable sequents in these systems.) This shows that even completeness holds for the interpretation above.

Representing data types in $\lambda 2$

In this subsection it will be shown that data types can be represented in $\lambda 2$. This result of Leivant (1983), (1989) will be presented in a modified form due to Barendsen (1989).

Definition 5.4.24.

1. A *data structure* is a many sorted structure with no given relations. A sort in a data structure is called a *data set*.

2. A *data system* is the signature of a data structure. A sort in a data system is called a *data type*.

Data systems will often be specified as shown in the following example.

- Sorts

$$\mathbf{A}, \mathbf{B}$$

- Functions

$$\mathbf{f} : \mathbf{A} \rightarrow \mathbf{B}$$

$$\mathbf{g} : \mathbf{B} \rightarrow \mathbf{A} \rightarrow \mathbf{A}$$

- Constants

$$\mathbf{c} \in \mathbf{A}.$$

In a picture:

Examples 5.4.25. Two data systems are chosen as typical examples.

1. The data system for the natural numbers **Nat** is specified as follows:

   - Sorts
   $$\mathbf{N};$$
   - Functions
   $$\mathbf{S} : \mathbf{N} \rightarrow \mathbf{N};$$
   - Constants
   $$\mathbf{0} \in \mathbf{N}.$$

2. The dat asystem of lists over a sort **A**, notation $\mathbf{List}_{\mathbf{A}}$, is specified as follows:

   - Sorts
   $$\mathbf{A}, \mathbf{L}_{\mathcal{A}};$$
   - Functions
   $$\mathbf{Cons} : \mathbf{A} \rightarrow \mathbf{L}_{\mathbf{A}} \rightarrow \mathbf{L}_{\mathbf{A}};$$
   - Constants
   $$\mathbf{nil} \in \mathbf{L}_{\mathbf{A}}.$$

Definition 5.4.26.

1. A sort in a data system is called a *parameter sort* if there is no ingoing arrow into that sort and also no constant for that sort.
2. A data system is called *parameter-free* if it does not have a parameter sort.

The data system **Nat** is parameter-free. The data system $\mathbf{List}_{\mathbf{A}}$ has the sort **A** as a parameter sort.

Definition 5.4.27. Let $\mathcal{D}$ be a data system. The language $L_\mathcal{D}$ corresponding to $\mathcal{D}$ is defined in 5.4.2

1. The *(open) termmodel* of $\mathcal{D}$, notation $\mathcal{T}(\mathcal{D})$, consists of the terms (containing free variables) of $L_\mathcal{D}$ together with the obvious maps given by the function symbols. That is, for every sort $\mathbf{C}$ of

   $\mathcal{D}$ the corresponding set C consists of the collection of the terms in $L_\mathcal{D}$ of sort $\mathbf{C}$; corresponding to a function symbol $\mathbf{f} : \mathbf{C}_1 \rightarrow \mathbf{C}_2$ a function $f : C_1 \rightarrow C_2$ is defined by

   $$f(t) = \mathbf{f}(t).$$

   A constant $\mathbf{c}$ of sort $\mathbf{C}$ is interpreted as itself; indeed one has also $\mathbf{c} \in C$.

2. Similarly one defines the *closed termmodel* of $\mathcal{D}$, notation $\mathcal{T}^o(\mathcal{D})$, as the substructure of sets of $\mathcal{T}(\mathcal{D})$ given by the closed terms.

For example the closed term model of **Nat** consists of the set

$$\mathbf{0}, \mathbf{S0}, \mathbf{SS0}, \ldots$$

with the successor function and the constant $\mathbf{0}$; this type structure is an isomorphic copy of

$$\langle \{0, 1, 2, ...\}, S, 0\rangle.$$

$\mathcal{T}(\mathbf{List}_\mathbf{A})$ consists of the finite lists of variables of type $\mathbf{A}$.

Definition 5.4.28. Given a data system $\mathcal{D}$ with

$$\begin{array}{ll}
\mathbf{A}_1, \ldots, \mathbf{A}_n & \text{parameter sorts;} \\
\mathbf{B}_1, \ldots, \mathbf{B}_m & \text{other sorts;} \\
\mathbf{f}_1 : \mathbf{A}_1 \rightarrow \mathbf{B}_1 \rightarrow \mathbf{B}_2 & \text{(say)} \\
\ldots & \\
\mathbf{c}_1 : \mathbf{B}_1 & \text{(say)} \\
\ldots &
\end{array}$$

Write

$$\begin{array}{rcl}
\Gamma_\mathcal{D} & = & A_1{:}*, \ldots, A_n{:}*, \\
 & & B_1{:}*, \ldots, B_m{:}*, \\
 & & f{:}A_1 \rightarrow B_1 \Rightarrow B_2, \\
\ldots & & \\
 & & c_1{:}B_1,
\end{array}$$

....

For every term $t \in L_{\mathcal{D}}$ define a $\lambda 2$-term $t^{\sim}$ and context Γ_t as follows.

t	$t^{\sim}$	Γ_t
$x^{\mathbf{C}}$	x	$x{:}C$
$\mathbf{f}t_1 \cdots t_n$	$ft_1^{\sim} \cdots t_n^{\sim}$	$\Gamma_{t_1} \cup \cdots \cup \Gamma_{t_n}$
$\mathbf{c}$	c	$\langle\rangle$

Lemma 5.4.29. *For a term $t \in L_{\mathcal{D}}$ of type C one has*

$$\Gamma_{\mathcal{D}}, \Gamma_t \vdash_{\lambda 2} t^{\sim} : C.$$

Proof. By induction on the structure of t. ∎ ∎

Given a data system $\mathcal{D}$, then there is a trivial way of representing $\mathcal{T}(\mathcal{D})$ into $\lambda 2$ (or even into $\lambda\!\to$) by mapping t onto $t^{\sim}$. Take for example the data system **Nat**. Then $\Gamma_{\mathbf{Nat}} = N{:}*, S{:}N\to N, 0{:}N$ and every term $\mathbf{S}^k\mathbf{0}$ can be represented as

$$\Gamma_{\mathbf{Nat}} \vdash (S^k 0) : N.$$

However, for this representation it is not possible to find, for example, a term *Plus* such that, say,

$$Plus(S0)(S0) =_{\beta} SS0.$$

The reason is that S is nothing but a variable and one cannot separate a compound $S0$ or $SS0$ into its parts to see that they represent the numbers one and two. Therefore we want a better form of representation.

Definition 5.4.30. Let $\mathcal{D}$ be a data system as in definition 5.4.28.

1. Write $\Delta_{\mathcal{D}} = \underline{A}_1{:}*, \ldots, \underline{A}_n : *$.
2. A $\lambda 2$-*representation* of $\mathcal{D}$ consists of the following.
   - Types $\underline{B}_1, \ldots, \underline{B}_m$ such that

     $$\Delta_{\mathcal{D}} \vdash \underline{B}_1 : *, \ldots, \underline{B}_m : *.$$

   - Terms $\underline{f}_1, \ldots, \underline{c}_1, \ldots$ such that

$$\Delta_{\mathcal{D}} \vdash \underline{f}_1 : \underline{A}_1 \to \underline{B}_1 \to \underline{B}_2;$$
$$\ldots$$
$$\Delta_{\mathcal{D}} \vdash \underline{c}_1 : \underline{B}_1;$$
$$\ldots$$

- Given a $\lambda 2$-representation of $\mathcal{D}$ there is for each term t of $L_{\mathcal{D}}$ a $\lambda 2$-term $\underline{t}$ and context Δ_t defined as follows.

t	$\underline{t}$	Δ_t
$x^{\mathbf{C}}$	x	$x{:}\underline{C}$
$\mathbf{f}t_1 \cdots t_n$	$\underline{f}\underline{t}_1 \cdots \underline{t}_n$	$\Delta_{t_1} \cup \cdots \cup \Delta_{t_n}$
$\mathbf{c}$	$\underline{c}$	$\langle\rangle$

- The $\lambda 2$-representation of $\mathcal{D}$ is called *free* if moreover for all terms t, s in $L_{\mathcal{D}}$ of the same type one has

$$\underline{t} =_\beta \underline{s} \quad \Leftrightarrow \quad t \equiv s.$$

Notation 5.4.31. Let $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ be a context. Then

$$\begin{array}{rcl} \lambda\Gamma.M & \equiv & \lambda x_1{:}A_1 \cdots \lambda x_n{:}A_n.M; \\ \Pi\Gamma.M & \equiv & \Pi x_1{:}A_1 \cdots \Pi x_n{:}A_n.M; \\ M\Gamma & \equiv & M x_1 \cdots x_n. \end{array}$$

Theorem 5.4.32 (Representation of data types in $\lambda 2$; Leivant (1983), Böhm-Berarducci (1985), Fokkinga (1987)). *Let D be a datasystem. Then there is a free representation of D in $\lambda 2$.*

Proof. Let D be given as in definition 5.4.28. Write

$$\begin{array}{rcl} \Theta_{\mathcal{D}} & = & B_1 : *, \ldots, B_m : *, \\ & & f_1 : A_1 \to B_1 \to B_2, \\ \ldots & & \\ & & c_1 : B_1, \\ \ldots . & & \end{array}$$

We want a representation such that for terms t in $L_{\mathcal{D}}$ of a non-parameter type

$$\underline{t}\Theta_{\mathcal{D}} =_\beta t^{\sim}[x_1 := x_1\Theta_{\mathcal{D}}] \cdots [x_n := x_n\Theta_{\mathcal{D}}], \qquad (1)$$

where $x_1, \ldots, x_n$ are free variables with non parameter types in t; for terms t of a parameter type one has

$$\underline{t} \equiv t^{\sim}. \tag{2}$$

Then for terms of the same non-parameter type one has

$$\begin{array}{rcl} \underline{t} =_\beta \underline{s} & \Rightarrow & \underline{t}\Theta_{\mathcal{D}} =_\beta \underline{s}\Theta_{\mathcal{D}} \\ & \Rightarrow & t^{\sim *} =_\beta s^{\sim *} \\ & \Rightarrow & t^{\sim} =_\beta s^{\sim} \\ & \Rightarrow & t^{\sim} \equiv s^{\sim} \\ & \Rightarrow & t \equiv s, \end{array}$$

where * denotes the substitutor $[x_1 := x_1\Theta_{\mathcal{D}}] \cdots [x_n := x_n\Theta_{\mathcal{D}}]$. For terms of the same parameter type the implication holds also. Now (2) is trivial, since a term t of a parameter type $\mathbf{A}$ is necessarily a variable and hence $t \equiv x^{\mathbf{A}}$, so $t^{\sim} \equiv x \equiv \underline{t}$. In order to fulfill (1) define

$$\begin{array}{rcl} \underline{B}_i & \equiv & \Pi\Theta_{\mathcal{D}}.B_i \\ \underline{c} & \equiv & \lambda\Theta_{\mathcal{D}}.c \\ \underline{f}_1 & \equiv & \lambda a_1{:}\underline{A}_1\lambda b_1{:}\underline{B}_1\lambda b_2{:}\underline{B}_2\lambda\Theta_{\mathcal{D}}.f a_1(b_1\Theta_{\mathcal{D}})(b_2\Theta_{\mathcal{D}}). \end{array}$$

Then by induction on the structure of t one can derive (1). Induction step:

$$\begin{array}{rcl} \underline{\mathbf{f}_1 t_1 t_2 t_3}\Theta_{\mathcal{D}} & \equiv & \underline{f}_1\underline{t}_1\underline{t}_2\underline{t}_3\Theta_{\mathcal{D}} \\ & =_\beta & f_1\underline{t}_1(\underline{t}_2\Theta_{\mathcal{D}})(\underline{t}_3\Theta_{\mathcal{D}}) \\ & =_\beta & f_1 t_1^{\sim} t_2^{\sim *} t_3^{\sim *} \\ & \equiv & (\mathbf{f}_1 t_1 t_2 t_3)^{\sim *}. \blacksquare \end{array}$$

$\blacksquare$

Now it will be shown that for a term $t \in L_{\mathcal{D}}$ the representation $\mathbf{t}$ in $\lambda 2$ given by theorem 5.4.32 can be seen as the canonical translation of a proof that t satisfies 'the second order definition of the set of elements of the free structure generated by $\mathcal{D}$'.

Definition 5.4.33.

1. The map $\sharp$: $\mathcal{T} \to \{0,1,2,3\} \times \{s,p\}$ for λC is modified as follows for pseudoterms of λPRED2. Let i range over $\{s,p\}$.

$$\begin{array}{rcll} \sharp(\Box^i) & = & 3^i, & \text{which is a notation for (3,i);} \\ \sharp(*^i) & = & 2^i; & \end{array}$$

$$\begin{array}{rcl} \sharp({}^{\square^i}x) & = & 1^i; \\ \sharp({}^{*^i}x) & = & 0^i; \end{array}$$

$$\sharp(\Pi x{:}A.B) = \sharp(\lambda x{:}A.B) = \sharp(BA) = \sharp(B).$$

2. A map $[\![\ ]\!]$: λPRED2 into λPROP2 is defined as follows.

$$\begin{array}{rcll} [\![\square^i]\!] & = & \square; & \\ [\![*^i]\!] & = & *; & \\ [\![{}^{\square^i}x]\!] & = & {}^{\square}x; & \\ [\![{}^{*^i}x]\!] & = & {}^{*}x; & \\ [\![\lambda x{:}A.B]\!] & = & [\![B]\!] & \text{if } \sharp A = 1^s, \\ & = & \lambda[\![x]\!]{:}[\![A]\!].[\![B]\!] & \text{else;} \\ [\![\Pi x{:}A.B]\!] & = & [\![B]\!] & \text{if } \sharp A = 1^s, \\ & = & \Pi[\![x]\!]{:}[\![A]\!].[\![B]\!] & \text{else;} \\ [\![BA]\!] & = & [\![B]\!] & \text{if } \sharp A = 1^s, \\ & = & [\![B]\!][\![A]\!] & \text{else.} \\ [\![x{:}A]\!] & = & \langle\rangle & \text{if } \sharp x \in \{0^s, 1^s\}, \\ & = & [\![x]\!]{:}[\![A]\!] & \text{else.} \\ [\![x_1{:}A_1, \ldots, x_n{:}A_n]\!] & = & [\![x_1{:}A_1]\!], \ldots, [\![x_n{:}A_n]\!]. & \end{array}$$

3. A map $|\ |$: λPROP2$\to\lambda$2 is defined as follows.

$$\begin{array}{rcl} |\square^i| & = & \square; \\ |*^i| & = & *; \\ |{}^{\square^i}x| & = & {}^{\square}x; \\ |{}^{*^i}x| & = & {}^{*}x; \\ |\Pi x{:}A.B| & = & \Pi|x|{:}|A|.|B|; \\ |\lambda x{:}A.B| & = & \lambda|x|{:}|A|.|B|; \\ |BA| & = & |B||A|. \end{array}$$

Finally put

$$|x{:}A| \quad = \quad |x|{:}|A|;$$

$$|x_1{:}A_1, \ldots, x_n{:}A_n| \quad = \quad |x_1{:}A_1|, \ldots, |x_n{:}A_n|.$$

4. A map $[\,]$: λPRED2$\to\lambda$2 is defined by $[A] = |[\![A]\!]|$.

Proposition 5.4.34.

1. $\Gamma \vdash_{\lambda\mathrm{PRED2}} A : B \;\;\Rightarrow\;\; [\![\Gamma]\!] \vdash_{\lambda\mathrm{PROP2}} [\![A]\!] : [\![B]\!]$.
2. $\Gamma \vdash_{\lambda\mathrm{PROP2}} A : B \;\;\Rightarrow\;\; |\Gamma| \vdash_{\lambda 2} |A| : |B|$.
3. $\Gamma \vdash_{\lambda\mathrm{PRED2}} A : B \;\;\Rightarrow\;\; [\Gamma] \vdash_{\lambda 2} [A] : [B]$.

Proof. 1. By induction on derivations using

$$[\![P[x := Q]]\!] = [\![P]\!][[\![x]\!] := [\![Q]\!]].$$

2. Similarly, using ${}^{*^s}x \notin FV([P])$.

3. By (1) and (2). ■

■

Now the alternative construction of $\underline{t}$ for $t \in L_{\mathcal{D}}$ can be given. The method is due to Leivant (1983). Let $\mathcal{D}$ be a datasystem with parameter sorts. To fix the ideas, let $\mathcal{D} = \mathbf{List}_{\mathbf{A}}$. Write $\Gamma_{\mathcal{D}} = A{:}*^s, L_A{:}*^s, nil{:}L_A$, $cons : A{\to}L_A{\to}L_A$. For the parameter type $\mathbf{A}$ a predicate $P^A{:}(A{\to}*^p)$ is declared. For $\mathbf{List}_{\mathbf{A}}$ the predicate

$$\begin{aligned} P^{L_A} \quad = \quad & \lambda z{:}(L_A).\Pi Q{:}(L_A{\to}*^p). \\ & [Q\,nil{\to}[\Pi a{:}A \Pi y{:}(L_A).P^A a{\to}Qy{\to}Q(cons\,ay)]{\to}Qz] \end{aligned}$$

says of an element $z{:}L_A$ that z belongs to the set of lists built up from elements of A satisfying the predicate P^A.

Now if $t \in L_{\mathcal{D}}$ is of type $\mathbf{List}_{\mathbf{A}}$, then intuitively $t^\sim : L_A$ satisfies P^{L_A}. Indeed, one has for such t

$$\Gamma_{\mathcal{D}}, \Gamma_t \vdash t^\sim : L_A \text{ and } \Gamma_{\mathcal{D}}, \Gamma_t \vdash d_t : (P^{L_A} t^\sim). \tag{1}$$

for some d_t constructed as follows. Let $\mathbf{C}$ range over $\mathbf{A}$ and $\mathbf{List}_{\mathbf{A}}$ with the corresponding C being A or L_A.

t	$t^\sim$	Γ_t	d_t
$x^{\mathbf{C}}$	x	$x{:}C, a_x{:}(P^C x)$	a_x
nil	*nil*	$\langle\rangle$	$\lambda Q{:}(L_A{\to}{*}^p)\lambda p{:}(Q\, nil)$ $\lambda q{:}(\Pi a{:}A\Pi y{:}L_A.$ $[P^A a{\to}Qy{\to}Q(cons\, ay)]).p$
$\mathbf{cons}\, t_1 t_2$	$cons\, t_1^\sim t_2^\sim$	$\Gamma_{t_1}, \Gamma_{t_2}$	$\lambda Q{:}(L_A{\to}{*}^p)\lambda p{:}(Q\, nil)$ $\lambda q{:}(\Pi a{:}A\Pi y{:}L_A.$ $[P^A a{\to}Qy{\to}Q(cons\, ay)]).$ $qt_1^\sim t_2^\sim d_{t_1}(d_{t_2}Qpq)$

By induction on the structure of t one verifies (1). By Proposition 5.4.34 it follows that

$$[\Gamma_{\mathcal{D}}, \Gamma_t] \vdash [d_t] : [(P^C t^\sim)]. \tag{2}$$

Write

$$\begin{array}{rcl} \underline{A} & = & [P^A], \\ \underline{L}_A & = & [P^{L_A}] = \Pi Q{:}{*}.Q{\to}(\underline{A}{\to}Q{\to}Q){\to}Q, \\ \text{nil} & = & [d_{\text{nil}}] = \lambda Q{:}{*}.\lambda p{:}Q\lambda q{:}(\underline{A}{\to}Q{\to}Q).p, \\ \text{cons} & = & \lambda a{:}\underline{A}\lambda x{:}\underline{L}_A\lambda Q{:}{*}.\lambda p{:}Q\lambda q{:}(\underline{A}{\to}Q{\to}Q).qax. \end{array}$$

Notice that this is the same $\lambda 2$-representation of $\mathbf{List}_{\mathbf{A}}$ as given in theorem 5.4.32 and that $\underline{t} =_\beta [d_t]$.

In this way many data types can be represented in $\lambda 2$.

Examples 5.4.35.

1. Lists.

   To be explicit, a *list* $\langle a_1, a_2\rangle \in L_A$ and *cons* are represented as follows.

$$\begin{array}{rcl} \underline{L}_A & = & (\Pi L{:}{*}.L{\to}(A{\to}L{\to}L){\to}L); \\ \langle \underline{a}_1, \underline{a}_2\rangle & = & (\lambda L{:}{*}\lambda nil{:}L\lambda cons{:}A{\to}L{\to}L.cons\, a_1(cons\, a_2 nil))); \\ \text{cons} & = & \lambda a{:}A\lambda x{:}(\Pi L{:}{*}.L{\to}(A{\to}L{\to}L){\to}L) \end{array}$$

$$\lambda L{:}{*}\lambda nil{:}L\lambda cons{:}A{\to}L{\to}L.cons\,(x L\, nil\, cons);$$

Moreover

$$A{:}{*}, a_1{:}A, a_2{:}A \vdash \langle \underline{a}_1, \underline{a}_2 \rangle{:}\underline{L}_A.$$

2. Booleans.
   Sorts
   Bool
   Constants
   true, false∈Bool

   are represented in $\lambda 2$ as follows.

$$\begin{array}{rcl} \underline{Bool} & = & \Pi\alpha{:}{*}.\alpha{\to}\alpha{\to}\alpha, \\ \underline{true} & = & \lambda\alpha{:}{*}\lambda x{:}\alpha\lambda y{:}\alpha.x, \\ \underline{false} & = & \lambda\alpha{:}{*}\lambda x{:}\alpha\lambda y{:}\alpha.y. \end{array}$$

3. Pairs.
   Sorts

$$\mathbf{A_1}, \mathbf{A_2}, \mathbf{B}$$

   Functions

$$\mathbf{p}{:}\mathbf{A_1}{\to}\mathbf{A_2}{\to}\mathbf{B}.$$

   Representation in $\lambda 2$

$$\begin{array}{rcl} \underline{B} & = & \Pi\alpha{:}{*}.(A_1{\to}A_2{\to}\alpha){\to}\alpha, \\ \underline{p} & = & \lambda x{:}A_1\lambda y{:}A_2\lambda\alpha{:}{*}\lambda z{:}(A_1{\to}A_2{\to}\alpha).zxy. \end{array}$$

Applying the map $|\;|: terms(\lambda 2){\to}\Lambda$ defined in 3.2.14 the usual representations of Booleans and pairing in the type-free λ-calculus is obtained. The same applies to the $\lambda 2$ representation of the data type Nat giving the type-free Church numerals.

Now that data types can be represented faithfully in $\lambda 2$, the question arises which functions on these can be represented by λ-terms. Since all terms have an nf, not all recursive functions can be represented in $\lambda 2$, see e.g. Barendregt (1990), thm. 4.2.15.

Definition 5.4.36. Let $\mathcal{D}$ be a data structure freely represented in$\lambda 2$ as usual. Consider in the closed term model $\mathcal{T}^o(\mathcal{D})$ a function $f{:}C{\to}C'$, where

C and C' are non-parameter sorts, is called *λ2-definable* if there is a term $\underline{f}$ such that

$$\Gamma_D \vdash_{\lambda 2} \underline{f} : (\underline{C} \to \underline{C'}) \ \& \ \underline{f}\underline{t} =_\beta \underline{ft} \text{ for all } t \in Term_C .$$

Definition 5.4.37. Let a data system $\mathcal{D}$ be given. A *Herbrand–Gödel system*, formulated in λPRED2, is given by

1. $\Gamma_{\mathcal{D}}$
2. $\Gamma_{f_1,\ldots,f_n}$, a finite set of function declarations of the form $f_1{:}B_1,\ldots,$ $f_n{:}B_n$ with $\Gamma_{\mathcal{D}} \vdash B_i{:}{*}^f$.
3. $\Gamma_{ax_1,\ldots,ax_m}$, a finite set of axiom declarations of the form $a_1{:}ax_1,\ldots,$ $a_m{:}ax_m$ with each ax_i of the form $f_j(s_1,\ldots,s_p) =_L r$ with the $s_1,\ldots,s_p,r$ terms in $L_{\mathcal{D}}$ of the correct type (see 5.4.17(4) for the definition of $=_L$) .

For such a Herbrand–Gödel system we write

$$HG \equiv \Gamma_{\mathcal{D}}, \Gamma_{\vec{f}}, \Gamma_{\vec{ax}}.$$

In order to emphasize the functions one may write $HG = HG(\vec{f})$. The principal function symbol is the last f_n.

Example 5.4.38. The following is a Herbrand-Gödel system (Note that the principal function symbol f_2 specifies the function $\lambda x \in Nat.x + x$).

$$\begin{aligned} HG_0 \equiv \quad & N{:}{*}^s, 0{:}N, S{:}(N \to N), \\ & f_1{:}N \to N \to N, f_2{:}N \to N_1 \\ & a_1{:}(\Pi x{:}N. f_1 x 0 =_L x), \\ & a_2{:}(f_1 x (Sy) =_L S(f_1 xy)), \\ & a_3{:}(f_2 x =_L f_1 xx). \end{aligned}$$

Definition 5.4.39. Let $\mathcal{A}$ be a data structure having no parameter sorts. Let $f : C \to C'$ be a given external function on $\mathcal{T}(\mathcal{D})$ (similar definitions can be given for functions of more arguments). Let HG be a Herbrand–Gödel system.

1. *HG computes f* ⇔ $HG = HG(f_1,\ldots,f_n)$ and for all $t \in \mathrm{Term}_C$ one has for some p

$$HG \vdash_{\lambda \mathrm{PRED2}} p : (f_n \underline{t} =_L \underline{f(t)}).$$

2. Suppose $HG(f_1, \ldots, f_n)$ computes f. Then f is called *provably type-correct* (in λPRED2) if for some B one has

$$HG \vdash_{\lambda\text{PRED2}} B : [\Pi x{:}\underline{C}.[P^C x \rightarrow P^{C'}(f_n x)]]$$

.

Note that the notion 'provably type correct' is a so-called intensional property: it depends on how the f is given to us as f_n. Now the questions about $\lambda 2$-definability can be answered in a satisfactory way. This result is due to Leivant (1983). It generalizes a result due to Girard (1972) characterizing the $\lambda 2$-definable functions on Nat as those that are provably total.

Theorem 5.4.40. *Let $\mathcal{D}$ be a parameter-free data structure.*

1. *The basic functions in $\mathcal{D}$ are $\lambda 2$-definable*
2. *A function $f{:}C \rightarrow C'$ is recursive iff f is HG computable.*
3. *A function $f{:}C \rightarrow C'$ is $\lambda 2$-definable iff f is HG-computable and provably type correct in λPRED2.*

Proof. 1. This was shown in theorem 5.4.32.

2. See Mendelson (1987).
3. See Leivant (1983), (1989). ∎

∎

5.5 Pure type systems not satisfying normalization

In this subsection some pure type systems will be considered in which there are terms of type $\bot \equiv \Pi\alpha{:}{*}.\alpha$. As a consequence there are typable terms without a normal form.

In subsection 5.2 we encountered the system $\lambda{*}$ which can be seen as a simplification of λC by identifying $*$ and $\Box$. It has as peculiarity that $* : *$ and its PTS specification is quite simple.

Definition 5.5.1. The system $\lambda{*}$ is the PTS determined as follows:

$\lambda{*}$	$\mathcal{S}$	$*$
	$\mathcal{A}$	$* : *$
	$\mathcal{R}$	$(*, *)$

Since all constructions possible in λC can be done also in $\lambda{*}$ by collapsing $\Box$ to $*$, it seems an interesting simplification. However, the system $\lambda{*}$

turns out to be 'inconsistent' in the sense that every type is inhabited, thus making the propositions-as-types interpretation meaningless. Nevertheless, the system $\lambda *$ is meaningful on the level of conversion of terms. In fact there is a nontrivial model of $\lambda *$, the so-called closure model due to Scott (1976), see also e.g. Barendregt and Rezus (1983). For a discussion on the computational relevance of $\lambda *$, see Coquand (1986) and Howe (1987).

The 'inconsistency' following from $*{:}*$ was first proved by Girard (1972). He also showed that the circularity of $*{:}*$ is not necessary to derive the paradox. For this purpose he introduced the following pure type system λU. Remember its definition.

Definition 5.5.2. The system λU is the PTS defined as follows:

λU	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*, *), (\Box, *), (\Box, \Box), (\Delta, \Box), (\Delta, *)$

So λU is an extension of $\lambda\omega$. The next theorem is the main result in this subsection. The proof occupies this whole subsection.

Theorem 5.5.3 (Girard's paradox). *In λU the type $\perp$ is inhabited, i.e. $\vdash M{:}\perp$, for some M.*

Proof. See 5.5.26. ■ ■

Corollary 5.5.4.

1. *In λU all types are inhabited.*
2. *In λU there are typable terms that have no normal form.*
3. *Results (1) and (2) also hold for $\lambda *$ in place of λU.*

Proof. 1. Let $M{:}\perp$ be provable in λU. Then

$$a{:}* \vdash Ma : a$$

and it follows that every type of sort $*$ in λU is inhabited. Types of sort $\Box$ or Δ are always inhabited; e.g. $\Pi\vec{x}{:}\vec{A}.*$ by $\lambda\vec{x}{:}\vec{A}.\perp$.

2. By proposition 5.2.31

3. By applying the contraction $f(*) = f(\Box) = f(\Delta) = *$ mapping λU onto $\lambda *$. ■

∎

The proof of Girard's paradox will be given in five steps. Use is made of ideas in Coquand (1985), Howe (1987) and Geuvers (1988).

1. Jumping out of a structure.
2. A paradox in naive set theory.
3. Formalizing.
4. An universal notation system in λU.
5. The paradox in λU.

Step 1. Jumping out of a structure
Usually the method of diagonalization provides a constructive way to 'jump out' of a structure. Hence if we make the (tacit) assumption that everything should be in our structure, then we obtain a contradiction, the paradox. Well known is the Russell paradox obtained by diagonalization. Define the naive set

$$R = \{a \mid a \notin a\}$$

Then

$$\forall a[a \in R \leftrightarrow a \notin a],$$

in particular

$$R \in R \leftrightarrow R \notin R,$$

which is a contradiction. A positive way of rephrasing this result is saying that R does not belong to the universe of sets from which we take the a; thus we are able to jump out of a system. This is the essence of diagonalization first presented in Cantor's theorem. The method of diagonalization yields also undecidable problems and sentences with respect to some given formal system (i.e. neither provable nor unprovable). (If the main thesis in Hofstadter (1979) turns out to be correct it may even be the underlying principle of self-consciousness.)

The following paradox is in its set theoretic form, due to Mirimanoff (1917). We present a game theoretic version by Zwicker (1987). Consider games for two players. Such a game is called *finite* if any run of the game cannot go on forever. For example noughts and crosses is finite. Chess is not finite (a game may go on forever, this in spite of the rule that there is a draw if the same position has appeared on the board three times; that rule is only optional). *Hypergame* is the following game: player I chooses a finite game; player II does the first move in the chosen game; player I does the second move in that game; etc. Claim: hypergame is finite. Indeed, after player I has chosen a finite game, only finitely many moves can be made within that game. Now consider the following run of hypergame.

Player I:	hypergame
Player II:	hypergame
Player I:	hypergame
...	...

Therefore hypergame is not a finite game and we have our paradox. This paradox can be formulated also as a positive result.

Proposition 5.5.5 (Informal). *Let A be a set and let R be a binary relation on A. Define for $a \in A$*

$$SN_R a \quad \Leftrightarrow \quad \textit{there is no infinite sequence } a_0, a_1, \ldots \in A \textit{ such that } \ldots\ldots R a_1 R a_0 R a.$$

Then in A we have

$$\neg \exists b \forall a \, [SN_R a \;\leftrightarrow\; aRb].$$

Proof. Suppose towards a contradiction that for some b

$$\forall a \, [SN_R a \leftrightarrow aRb]. \tag{1}$$

Then

$$\forall a \, [aRb \rightarrow SN_R a]. \tag{2}$$

This implies

$$SN_R b,$$

because if there is an infinite sequence under b

$$\ldots R a_1 R a_0 R b$$

then there is also one under $a_0 (Rb)$, contradicting (2). But then by (1)

$$bRb$$

Hence $\ldots RbRbRb$ and this means $\neg SN_R b$. Contradiction. ∎ ∎

By taking for A the universe of all sets and for R the relation $\in$, one obtains Mirimanoff's paradox. By taking for A the collection of all ordinal numbers and for R again $\in$, one obtains the Burali–Forti paradox.

The construction in 5.5.5 is an alternative way of 'jumping out of a system'. This method and the diagonalization inherent in Cantor's theorem can be seen as limit cases of the following generalized construction. This observation is due to Quine (1963), p.36.

Proposition 5.5.6. *Let A be a set and let R be a binary relation on A. For $n = 1, 2, \ldots, \infty$ define*

$$C_n a \quad \Leftrightarrow \quad \exists a_0, \ldots, a_n \in A[a_0 = a \;\&\; \forall i < n\; a_{i+1} R a_i \;\&\; a_n = a].$$

$$B_n = \{a \in A \mid \neg C_n a\}.$$

$\{$The set B_n consists of those $a \in A$ not on an 'n-cycle'$\}$. Then in A one has

$$\neg \exists b \forall a [B_n a \leftrightarrow a R b].$$

Proof. Exercise. ■ ■

By taking $n = 1$ one obtains the usual diagonalization method of Cantor. By taking $n = \infty$ one obtains the result 5.5.5. Taking $n = 2$ gives the solution to the puzzle 'the exclusive club' of Smullyan (1985), p.21. (A person is a member of this club if and only if he does not shave anyone who shaves him. Show that there is no person that has shaved every member of the exclusive club and no one else.)

Step 2. The paradox in naive set theory

Now we will define a (naive) set T with a binary relation $<$ on it such that

$$\forall a \in T\; [SN_< a \quad \leftrightarrow \quad a < b], \tag{!}$$

for some $b \in T$. Together with Proposition 5.5.5 this gives the paradox. The particular choice of T and $<$ is such that the auxiliary lemmas needed can be formalized in λU.

Definition 5.5.7.

1. $T = \{(A, R) \mid A$ is a set and R is a binary transitive relation on $A\}$
   For $(A, R), (A', R') \in T$ and $f{:}A \rightarrow A'$ write

   $$\begin{array}{lcl} (A,R) <_f^- (A',R') & \Leftrightarrow & \forall a, b \in A\; [aRb \;\rightarrow\; f(a) R' f(b)]; \\ f \text{ is bounded} & \Leftrightarrow & \exists a' \in A' \forall a \in A. f(a) R' a'; \\ (A,R) <_f (A',R') & \Leftrightarrow & (A,R) <_f^- (A',R') \;\&\; f \text{ is bounded.} \end{array}$$

2. Define the binary relation $<$ on T by

   $$(A, R) < (A', R') \quad \Leftrightarrow \quad \exists f{:}(A \rightarrow A')[(A, R) <_f (A', R')].$$

3. Let $W = \{(A, R) \in T \mid SN_<(A, R)\}$.

We will see that $b = (W, <) \in T$ satisfies (!) above. (For notational simplicity we write for the restriction of $<$ to W also $<$.)

Definition 5.5.8. For $(A, R) \in T$ and $a \in A$ write

1. $A_a = \{b \in A \mid bRa\}$;
2. R_a is the restriction of R to A_a.

Lemma 5.5.9. *Let* $(A, R) \in T$ *and* $a, b \in A$. *Then*

1. $(A_a, R_a) < (A, R)$;
2. $aRb \rightarrow (A_a, R_a) < (A_b, R_b)$;
3. $aRb \rightarrow SN_R b \rightarrow SN_R a$;
4. $[\forall a \in A SN_R a] \rightarrow SN_<(A, R)$.

Proof. 1,2. By using the map $f = \lambda x{:}A_a.x$. For (2) the transitivity of R is needed to ensure that f has codomain A_b. In both cases f is bounded by a.

3. Suppose aRb. If there is an infinite R-chain under a, i.e. $\ldots a_1 R a_o R a$, then there is also one under b; indeed $\ldots a_1 R a_o R a R b$. Therefore $SN_R b$ imlpies $SN_R a$.

4. Suppose there is an infinite $<$-chain under (A, R):

$$\ldots (A_1, R_1) < (A_0, R_0) < (A, R).$$

¿From the figure 5 it can be seen that using the bounding elements in (A_n, R_n) for the map $f_n{:}A_{n+1} \rightarrow A_n$ (projected via the fs into A) there is an infinite R-chain, below an element of A.

This contradicts the assumption $\forall a \in A\ SN_R(a)$. ■

■

Proposition 5.5.10.

$$\forall (A, R) \in T[SN_<(A, R) \leftrightarrow (A, R) < (W, <)].$$

Proof. It suffices to show that for $(A, R) \in T$

1. $SN_<(A, R) \rightarrow (A, R) < (W, <)$;
2. $SN_<(W, <)$.

For then $(A, R) < (W, <) \rightarrow SN_<(A, R)$ by Lemma 5.5.9 (3).

As to 1, suppose $SN_<(A, R)$. Let $a{\in}A$ and define $f(a) = (A_a, R_a)$, with R_a defined in 5.5.8. By 5.5.9 (1) one has $f(a) < (A, R)$; by assumption and 5.5.9(3) applied to $(T, <)$ it follows that $SN_<(f(a))$ and hence $f(a) \in W$. Therefore $f{:}A{\rightarrow}W$. Moreover, $f{:}(A, R) < (W, <)$ by Lemma 5.5.9 (1), (2).

As to 2, note that by definition $\forall (A, R) \in W\ SN_<(A, R)$. Hence by Lemma 5.5.9 (4) one has $SN_<(W, <)$. ■ ■

Fig. 5.

Step 3. Formalizing

In this step several notions and lemmas from steps 1 and 2 will be formalized. This could be done inside the systems of the cube (in fact inside λP2). However, since we want the eventual contradiction to occur inside λU, a system that is chosen with as few axioms as seems possible, the formalization will be done in λU directly. From now on the notions of context and $\vdash$ refer to λU. Use will be made freely of logical notions (e.g. we write $\forall a{:}A$ instead of $\Pi a{:}A$).

The first task is to define the notion SN_R without referring to the concept of infinity.

Definition 5.5.11.

1. Γ_0 is the context
$$A{:}\Box,\, R{:}(A{\rightarrow}A{\rightarrow}*).$$
2. Write in context Γ_0
$\mathrm{chain}_{A,R} \equiv \lambda P{:}(A{\rightarrow}*).\forall a{:}A[Pa{\rightarrow}\exists b{:}A[Pb \;\&\; bRa]]$
$SN_{A,R} \equiv \lambda a{:}A.\forall P{:}(A{\rightarrow}*)[\mathrm{chain}_{A,R}P \rightarrow \neg Pa].$

Intuitively, $\mathrm{chain}_{A,R}P$ states that $P{:}A{\rightarrow}*$ is a predicate on (i.e. subset of) A such that for every element a in P there is an element b in P with bRa. Moreover $\mathrm{SN}_{A,R}a$ states that $a{:}A$ is not in a subset $P \subseteq A$ that is a chain.

Lemma 5.5.12. *In λU one can show*

1. $\Gamma_0 \vdash chain_{A,R} : ((A{\rightarrow}*){\rightarrow}*)$.

2. $\Gamma_0 \vdash SN_{A,R} : (A \rightarrow *)$.

Proof. Immediate. ■ ■

Proposition 5.5.13. *In context* Γ_0 *the type*

$$\neg \exists b{:}A \forall a{:}A[SN_{A,R}a \leftrightarrow aRb]$$

is in λU *inhabited.*

Proof. With a little effort the proof of Proposition 5.5.5 can be formalized in λPRED2. Then one can apply the map $f{:}\lambda\text{PRED2} \rightarrow \lambda U$ determined by

$$f(*^p) = *, f(*^s) = f(*^f) = f(\Box^p) = \Box, f(\Box^s) = \Delta.i \blacksquare$$

■

We now need a relativization of Proposition 5.5.13.

Definition 5.5.14.

1. In context Γ_0 write
   $\text{closed}_{A,R} \equiv \lambda Q{:}(A \rightarrow *).\forall a, b{:}A\ [Qa \rightarrow bRa \rightarrow Qb]$.
   {$\text{closed}_{A,R}Q$ says: 'if a is in Q and b is R-below a, then b is in Q'.}
2. In context $\Gamma_0, Q{:}A \rightarrow *$, write

$$\forall a{:}A^Q.B \equiv \forall a{:}A[Qa \rightarrow B]$$

$$\exists a{:}A^Q.B \equiv \exists a{:}A[Qa \& B].$$

   {This is relativizing to a predicate Q.}

Corollary 5.5.15. *In context* $\Gamma_0, Q{:}A \rightarrow *$ *the type*

$$\textit{closed}_{A,R}Q \rightarrow \exists b{:}A^Q \forall a{:}A^Q\ [SN_{A,R}a \leftrightarrow aRb]$$

is inhabited in λU.

Proof. The proof of Proposition 5.5.5 formalized in PRED2 can be relativized and that proof becomes, after applying the contraction f the required inhabitant. ■ ■

So far we have formalized the results in Step 1. There are several problems for the formalization of the naive paradox in Step 2 into λU. The main one is that in λU a 'subset' of a type does not form a type again. For example it is not clear how to form $A_a(\subseteq A)$ as a type. This problem is solved by considering instead of a structure (A_a, R_a) the structure (A, R^a) with

$$bR^a c \iff bRc \;\&\; bRa.$$

In order to formalize Lemma 5.5.9 the definition of $<$ has to be adjusted. Let the *domain* of R be the (naive) subset

$$\mathrm{Dom}_R = \{a{:}A \mid \exists b{:}A\ aRb\}.$$

In the new definition of $<$ it is required that the monotonic map involved is bounded, but only on the domain of R.

A second problem is that T and W are not types and that it is not clear how to realize $(W,<){\in}T$. This problem will be solved by constructing in λU a 'universal' kind **U** such that all pairs (A, R) can be 'faithfully' embedded into **U**.

Definition 5.5.16. In λU define two predicates $<^-, <$ of type

$$[\Pi\alpha{:}\Box\Pi r{:}(\alpha{\to}\alpha{\to}*)\Pi\alpha'{:}\Box\Pi r'{:}(\alpha'{\to}\alpha'{\to}*)\Pi f{:}(\alpha{\to}\alpha').*]$$

as follows. We write

$$(A,R) <_f^- (A',R') \ \text{ for } \ <^- ARA'R'f$$

and similarly for $<$.

1. $(A,R) <_f^- (A',R') \iff \forall a,b{:}A\ [aRb \to (fa)R'(fb)]$.
2. $$\begin{aligned}(A,R) <_f (A',R') \iff\ & (A,R) <_f^- (A',R')\ \& \\ & \exists a'{:}A'\ [Dom_{R'}a'\,\& \\ & \forall a{:}A\ [Dom_R a {\to} (fa)R'a']],\end{aligned}$$

   where $\mathrm{Dom}_R a$ stands for $\exists b{:}A.aRb$.
3. Write for the appropriate A, R and A', R'

   $$(A,R) <^- (A',R') \iff \exists f{:}A{\to}A'\ (A,R) <_f^- (A',R')$$

   and similarly for $<$.

The notion $SN_<$ is not a particular instance of the notion $SN_{A,R}$. This is because the 'set'

$$\{(A,R) \mid A{:}*, R{:}A{\to}A{\to}*\}$$

on which $<$ is supposed to act does not form a type. Therefore $SN_<$ has to be defined separately.

Definition 5.5.17.

1. $\text{chain}_< \equiv \lambda P{:}(\Pi\alpha{:}\Box.(\alpha\to\alpha\to *)\to *).$
$$[\forall\alpha_1{:}\Box\forall r_1{:}(\alpha_1\to\alpha_1\to *).$$
$$[P\alpha_1 r_1\to\exists\alpha_2{:}\Box\exists r_2{:}(\alpha_2\to\alpha_2\to *)$$
$$[P\alpha_2 r_2 \& (\alpha_2, r_2) < (\alpha_1, r_1)]$$
$$]$$
$$].$$
2. $SN_< \equiv \lambda\alpha{:}\Box\lambda r{:}(\alpha\to\alpha\to *).\forall P{:}[\Pi\alpha'{:}\Box.(\alpha'\to\alpha'\to *)\to *].$
$$[\text{chain}_< P\to\neg(P\alpha r)].$$
3. Trans $\text{Trans} \equiv \lambda\alpha{:}\Box\lambda r{:}(\alpha\to\alpha\to *).\forall a, b, c{:}\alpha.[arb\to brc\to arc].$
4. In context $\Gamma_0, a{:}A$ define
$$R^a \equiv \lambda b, c{:}A.[bRc \& bRa].$$

Proposition 5.5.18. *Let $A{:}\Box$, $R{:}(A\to A\to *)$, $a{:}A$, $b{:}A$ and assume*
$$\text{Trans}AR;$$
that is, work in context x :TransAR. Then the following types are inhabited.

1. $\text{Dom}_R a \to (A, R^a) < (A, R)$.
2. $aRb \to (A, R^a) < (A, R^b)$.
3. $aRb \to SN_{A,R}b \to SN_{A,R}a$.
4. $(\forall a{:}A.SN_{A,R}a) \leftrightarrow SN_< AR$.

Proof. 1. Assume $\text{Dom}_R a$. Define $f = \lambda x{:}A.x$. Then $(A, R^a) <_f^- (A, R)$. Moreover a in Dom_R bounds $fx = x$ for x in $\text{Dom}_{(R^a)}$. Indeed, $xR^a y \to xRa$. Therefore $(A, R^a) <_f (A, R)$.

2. Assume Trans AR and aRb. Again define $f = \lambda x{:}A.x$. Then
$$(A, R^a) <_f^- (A, R^b);$$
indeed, $xR^a y \to xRy \ \&\ xRaRb \to xR^b y$ by the transitivity of R. Also a is in $\text{Dom}_{(R^b)}$ and again bounds $fx = x$ for x in $\text{Dom}_{(R^a)}$.

3. Assume aRb and $SN_{A,R}b$. Let $\text{chain}_{A,R}P$ and assume towards a contradiction Pa. Define $P' = \lambda x{:}A.[Px \ \vee\ x =_L b]$. Then also $\text{chain}_{A,R}P'$ and $P'b$, contradicting $SN_{A,R}b$.

4. $(\to)$ Assume $(\forall a{:}A.\ SN_{A,R}a)$. Let $\text{chain}_< P$ and assume towards a contradiction PAR. Then for some A' and R' one has $PA'R'$ and $(A', R') < (A, R)$, and therefore for some $a{:}A$ one has

$$\begin{aligned}\mathrm{Dom}_R a \quad \& \quad & \exists f{:}(A'{\rightarrow}A)\ [(A',R') <^-_f (A,R) \\ & \& \ \forall y{:}A'[\mathrm{Dom}_{R'} y \rightarrow (fy)Ra]] \qquad (1)\end{aligned}$$

Define

$$\begin{aligned}P' \quad \equiv \quad & \lambda x{:}A.\mathrm{Dom}_R x \ \& \ [\exists \alpha{:}\Box\exists r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*). \\ & P\alpha r \ \& \ \exists f{:}(\alpha{\rightarrow}A)\ [(\alpha,r) <^-_f (A,R) \\ & \& \ \forall y{:}\alpha\ [\mathrm{Dom}_r y{\rightarrow}(fy)Rx]]].\end{aligned}$$

Then also $\mathrm{chain}_{A,R}P'$. By (1) one has $P'a$, contradicting $SN_{A,R}a$.

($\leftarrow$) Assume $SN_< AR$. Let $a{:}A$ and suppose towards a contradiction that $\mathrm{chain}_{A,R}P$ and Pa. Define

$$P' \equiv \lambda\alpha{:}\Box\lambda r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*).[\exists b{:}A.Pb \ \& \ (A,R^b) < (\alpha,r)].$$

Then $\mathrm{chain}_< P'$, by (2), and $P'AR$, by (1) and (2), contradicting $SN_< AR$. ■

■

Step 4. A universal notation system in λU

In this step the second problem mentioned in Step 3 will be solved. Terms $\mathbf{U}$ and $\mathbf{i}$ will be constructed such that $\mathbf{i}$ faithfully embeds a pair (A,R) with $A{:}n$ and $R{:}(A{\rightarrow}A{\rightarrow}*)$ into $\mathbf{U}$. Such a pair ($\mathbf{U}$, $\mathbf{i}$) is called a *universal notation system* for orderings and plays the role of the naive set $T = \{(A,R) \mid R{:}A{\rightarrow}A{\rightarrow}*\}$.

Proposition 5.5.19. *There are terms* $\mathbf{U}$ *and* $\mathbf{i}$ *such that in* λU

1. $\vdash \mathbf{U} : \Box$.
2. $\vdash \mathbf{i} : [\Pi\alpha{:}\Box.(\alpha{\rightarrow}\alpha{\rightarrow}*){\rightarrow}\mathbf{U}]$.
3. *The type* {*'faithfulness of the map* $\mathbf{i}$*'*}

$$\forall\alpha{:}\Box\forall r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*)\forall\alpha'{:}\Box\forall r'{:}$$

$$(\alpha'{\rightarrow}\alpha'{\rightarrow}*)[\mathbf{i}\alpha r =_L \mathbf{i}\alpha' r'{\rightarrow}(\alpha',r) <^- (\alpha',r')]$$

is inhabited.

Proof. Define

$$H \quad \equiv \quad \Pi\alpha{:}\Box.[(\alpha{\rightarrow}\alpha{\rightarrow}*){\rightarrow}*];$$

$$\begin{array}{rcl}\mathbf{U} & \equiv & H{\rightarrow}*;\\ \mathbf{i} & \equiv & \lambda\alpha{:}\Box\lambda r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*)\lambda h{:}H.h\alpha r.\end{array}$$

Then clearly one has in λU

$$H:\Box,\, \mathbf{U}:\Box \text{ and } \mathbf{i}:[\Pi\alpha{:}\Box.(\alpha{\rightarrow}\alpha{\rightarrow}*){\rightarrow}(H{\rightarrow}*)].$$

So we have 1 and 2. As to 3, we must show that in context

$$\alpha{:}\Box,\, r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*),\, \alpha'{:}\Box,\, r'{:}(\alpha'{\rightarrow}\alpha'{\rightarrow}*)$$

the type

$$\mathbf{i}\alpha r =_L \mathbf{i}\alpha' r'{\rightarrow}(\alpha, r) <^- (\alpha', r')$$

is inhabited. Now

$$\begin{array}{ll} & \mathbf{i}\alpha r =_L \mathbf{i}\alpha' r'\\ \rightarrow & \lambda h{:}H.h\alpha r =_L \lambda h{:}H.h\alpha' r'\\ \rightarrow & h\alpha r =_L h\alpha' r',\ \text{for all } h{:}H,\\ \rightarrow & [(\alpha, r) <^- (\alpha', r')] =_L [(\alpha', r') <^- (\alpha', r')],\end{array}$$

by taking $h \equiv \lambda\beta{:}\Box\lambda s{:}(\beta{\rightarrow}\beta{\rightarrow}*).(\beta, s) <^- (\alpha', r')$.

Since the right-hand side of the last equation is inhabited it follows that

$$(\alpha, r) <^- (\alpha', r'). \blacksquare$$

$\blacksquare$

Step 5 The paradox in λU

Using $\mathbf{U}$ in $\mathbf{i}$ of Step 4 we now can formalize the informal paradox derived in step 2.

Definition 5.5.20.

1. On $\mathbf{U}$ define the binary relation $<_{\mathbf{i}}$ as follows. For $u, u'{:}\mathbf{U}$ let

$$\begin{array}{rcl} u <_{\mathbf{i}} u' & \equiv & \exists\alpha{:}\Box\exists r{:}(\alpha{\rightarrow}\alpha{\rightarrow}*)\exists\alpha'{:}\Box\exists r'{:}(\alpha'{\rightarrow}\alpha'{\rightarrow}*).\\ & & [u =_L (\mathbf{i}\alpha r) \& u' =_L (\mathbf{i}\alpha' r') \&\\ & & \text{Trans } \alpha r \ \&\ \text{Trans } \alpha' r' \ \&\\ & & SN_<(\alpha, r)\ \&\\ & & SN_<(\alpha', r')\ \&\ (\alpha, r) < (\alpha', r')].\end{array}$$

2. On $\mathbf{U}$ define the (unary) predicate $\mathbf{I}$ as follows. For $u{:}\mathbf{U}$ let

$$\mathbf{I}u = \exists\alpha{:}\Box\exists r{:}(\alpha{\to}\alpha{\to}*).$$
$$[u =_L (\mathbf{i}\alpha r) \ \&\ \text{Trans } \alpha r \ \&\ SN_<(\alpha, r)].$$

Note that $\text{closed}_{\mathbf{U}, <_{\mathbf{i}}} \mathbf{I}$.

3. The element $\mathbf{u} : \mathbf{U}$ is defined by $\mathbf{u} \equiv \mathbf{i}\mathbf{U} <_{\mathbf{i}}$.

Lemma 5.5.21. *In context $\alpha{:}\Box, r{:}(\alpha{\to}\alpha{\to}*), \alpha'{:}\Box, r'{:}(\alpha'{\to}\alpha'{\to}*)$ the following types are inhabited.*

1. $(\mathbf{i}\alpha r) <_{\mathbf{i}} (\mathbf{i}\alpha' r') \to (\alpha, r) < (\alpha', r')$.
2. $SN_<(A, R) \to SN_{\mathbf{U}, <_{\mathbf{i}}}(\mathbf{i}AR)$.

Proof. 1. Suppose $(\mathbf{i}\alpha r) <_{\mathbf{i}} (\mathbf{i}\alpha' r')$. Then there are β, s, β', s' of appropriate type such that

$$\mathbf{i}\alpha r =_L \mathbf{i}\beta s \ \&\ \mathbf{i}\alpha' r' =_L \mathbf{i}\beta' s' \ \&\ (\beta, s) < (\beta', s').$$

By the faithfulness of $\mathbf{i}$ and the symmetry of $=_L$ it follows that

$$(\alpha, r) <^- (\beta, s) < (\beta', s') <^- (\alpha', r')$$

hence

$$(\alpha, r) < (\alpha', r').$$

2. Suppose $SN_<(A, R)$. If $\text{chain}_{\mathbf{U}, <_{\mathbf{i}}} Q$, then define

$$P\alpha r \ \equiv\ Q(\mathbf{i}\alpha r).$$

Then $\text{chain}_< P$. Since $SN_<(A, R)$ we have $\neg PAR$. But then $\neg Q(\mathbf{i}AR)$. So we proved

$$\text{chain}_{\mathbf{U}, <_{\mathbf{i}}} Q \to \neg Q(\mathbf{i}AR),$$

i.e. $SN_{\mathbf{U}, <_{\mathbf{i}}}(\mathbf{i}AR)$. ■

■

Corollary 5.5.22. *The type*

$$\forall u{:}\mathbf{U}.SN_{\mathbf{U}, <_{\mathbf{i}}} u$$

is inhabited.

Proof. Let $u{:}\mathbf{U}$ and suppose towards a contradiction

$$\mathrm{chain}_{\mathbf{U},<_{\mathbf{i}}} P \ \&\ Pu.$$

Then

$$\exists u'{:}\mathbf{U}.(u' <_{\mathbf{i}} u \ \&\ Pu').$$

Now

$$u' <_{\mathbf{i}} u \rightarrow \exists \alpha{:}\square \exists r{:}(\alpha\rightarrow\alpha\rightarrow *)[u =_L (\mathbf{i}\alpha r) \ \&\ SN_<(\alpha, r)].$$

Hence by (2) of the lemma

$$SN_{\mathbf{U},<_{\mathbf{i}}}(\mathbf{i}\alpha r) =_L SN_{\mathbf{U},<_{\mathbf{i}}} u.$$

But then, again using $\mathrm{chain}_{\mathbf{U},<_{\mathbf{i}}} P$, it follows that $\neg(Pu)$. Contradiction. ■
█

Lemma 5.5.23. *Let* $A{:}\square$, $R{:}(A\rightarrow A\rightarrow *)$ *and assume* TransAR. *Then the following type is inhabited*

$$SN_<(A, R) \rightarrow \forall a{:}A.SN_<(A, R^a).$$

Proof. Applying 5.5.18(4) one has

$$\begin{array}{rcll} SN_<(A,R) & \rightarrow & \forall b{:}A.SN_{A,R}b, & \\ & \rightarrow & \forall b{:}A.\forall a{:}A.SN_{A,(R^a)}b, & \text{see below}, \\ & \rightarrow & \forall a{:}A.SN_<(A, R^a). & \end{array}$$

The implication $SN_{A,R}b \rightarrow SN_{A,(R^a)}b$ is proved as follows. Let $SN_{A,R}b$ and assume towards a contradiction that $\mathrm{chain}_{A,(R^a)}P$ and Pb. Then also $\mathrm{chain}_{A,R}P$, contradicting $\mathrm{SN}_{A,R}b$. ■ █

Lemma 5.5.24.

1. *Let* $\alpha{:}\square$ *and* $r{:}\alpha\rightarrow\alpha\rightarrow *$ *and assume* Transαr & $SN_<(\alpha, r)$. *Then there are* $\alpha^+{:}\square$ *and* $r^+{:}\alpha^+\rightarrow\alpha^+\rightarrow *$ *such that*

$$\text{Trans}\ \alpha^+ r^+ \ \&\ SN_<(\alpha^+, r^+) \ \&\ (\alpha, r) < (\alpha^+, r^+).$$

2. $\forall v{:}\mathbf{U}^{\mathbf{I}} \exists v^+{:}\mathbf{U}^{\mathbf{I}} v <_{\mathbf{i}} v^+.$

Proof. 1. The construction is the one for representing data structures in Section 5.4. Define

$$\alpha^\circ \equiv \Pi\beta{:}\Box.\beta\to(\alpha\to\beta)\to\beta,$$

$$F \equiv \lambda x{:}\alpha\lambda\beta{:}\Box\lambda\infty{:}\beta\lambda f{:}(\alpha\to\beta).fx,$$

$$\underline{\infty} \equiv \lambda\beta{:}\Box\lambda\infty{:}\beta\lambda f{:}(\alpha\to\beta).\infty;$$

then $\underline{\infty}{:}\alpha^\circ{:}\Box$ and $F{:}(\alpha\to\alpha^\circ)$. Intuitively $\alpha^\circ = \alpha\cup\{\underline{\infty}\}$ and F is the canonical imbedding. Indeed, F is injective and $\underline{\infty}$ is not in the range of F. In fact, in the given context one has

$$(\lambda a{:}\alpha\lambda b{:}\alpha\lambda p{:}(Fa =_L Fb)\lambda Q{:}(\alpha\to *).p(\lambda x{:}\alpha^\circ.x*\bot Q)) \quad : \quad (\forall a,b{:}\alpha.(Fa =_L Fb\to a =_L b);$$

$$(\lambda a{:}\alpha\lambda p{:}(Fa =_L \underline{\infty}).p(\lambda x{:}\alpha^\circ.x*\bot(\lambda a{:}\alpha.T))(\lambda b{:}\bot.b)) \quad : \quad (\forall a{:}\alpha.Fa \neq_L \underline{\infty});$$

here $T\equiv\bot\to\bot$ stands for 'true' and has $(\lambda b{:}\bot.b)$ as inhabiting proof. Define $r^\circ{:}\alpha^\circ\to\alpha^\circ\to *$ as the canonical extension of r to α^+ making $\underline{\infty}$ larger than the elements of α:

$$r^\circ \equiv \lambda x{:}\alpha^\circ\lambda y{:}\alpha^\circ.[\exists a{:}\alpha\exists b{:}\alpha.rab \;\&\; x =_L Fa \;\&\; y =_L Fb] \vee$$

$$[\exists a{:}\alpha.x =_L Fa \;\&\; y =_L \underline{\infty}].$$

Then Trans $\alpha^\circ r^\circ \& SN_<(\alpha^\circ, r^\circ)$ and $(\alpha, r) <_F^- (\alpha^\circ, r^\circ)$ with bounding element $\underline{\infty}$. This $\underline{\infty}$ is not yet in Dom_{α^+}; but one has $(\alpha, r) <_{F\circ F} (\alpha^{\circ\circ}, r^{\circ\circ})$ with bounding element $F\underline{\infty}$ and therefore one can take $\alpha^+ = \alpha^{\circ\circ}$ and $r^+ = r^{\circ\circ}$.

2. If $v = \mathbf{i}\alpha r$, then take $v^+ = \mathbf{i}\alpha^+ r^+$. ■

■

Proposition 5.5.25. *The following type is inhabited:*

$$\exists u{:}\mathbf{U}^{\mathbf{I}}\forall v{:}\mathbf{U}^{\mathbf{I}}[SN_{\mathbf{U},<_{\mathbf{i}}} v \;\;\leftrightarrow\;\; v <_{\mathbf{i}} u].$$

Proof. For u one can take $\mathbf{u} \equiv (\mathbf{iU} <_{\mathbf{i}})$. In view of Corollary5.5.22 it is sufficient to show for $v{:}\mathbf{U}$ that {the following types are inhabited}:

1. $\mathbf{Iu}$,

2. $\mathbf{I}v \to v <_{\mathbf{i}} \mathbf{u}$.

As to 1, we know from Corollary 5.5.22

$$\begin{aligned}\forall u{:}\mathbf{U} \quad & SN_{\mathbf{U},<_{\mathbf{i}}} u\\ & \to SN_{<} \mathbf{U} <_{\mathbf{i}}), \text{ by Proposition 5.5.18(4)},\\ & \to \mathbf{I}(\mathbf{iU} <_{\mathbf{i}}), \text{ since clearly Trans } \mathbf{U} <_{\mathbf{i}},\\ & \to \mathbf{Iu}.\end{aligned}$$

As to 2, assume $\mathbf{I}v$. Then $v =_L (\mathbf{i}\alpha r)$ for some pair α, r with Transαr & $SN_<(\alpha, r)$.

Define

$$f \equiv (\lambda a{:}\alpha.(\mathbf{i}\alpha r^a)) : (\alpha \to \mathbf{U}).$$

Then for all $a{:}\alpha$ with $\mathrm{Dom}_r a$ one has

$$fa = (\mathbf{i}\alpha r^a) <_{\mathbf{i}} (\mathbf{i}\alpha r) = v,$$

{by 5.5.18(1) one has $(\alpha, r^a) < (\alpha, r)$; use Lemma 5.5.23 and the definition of $<_{\mathbf{i}}$} and similarly for all $a, b{:}\alpha$

$$\begin{aligned} arb \quad & \to (\alpha, r^a) < (\alpha, r^b),\\ & \to \mathbf{i}\alpha r^a <_{\mathbf{i}} \mathbf{i}\alpha r^b, \qquad SN_<(\alpha, r^a) \;\&\; SN_<(\alpha, r^b) \text{ since } SN_<(\alpha, r),\\ & \to fa <_{\mathbf{i}} fb.\end{aligned}$$

Therefore $(\alpha, r) <_f^- (\mathbf{U}, <_{\mathbf{i}})$; f on Dom_r is bounded by v. Since $v <_{\mathbf{i}} v^+$ one has $\mathrm{Dom}_{<_{\mathbf{i}}} v$. Therefore $(\alpha, r) <_f (\mathbf{U}, <_{\mathbf{i}})$ and hence $v =_L (\mathbf{i}\alpha r) <_{\mathbf{i}} (\mathbf{iU} <_{\mathbf{i}}) = \mathbf{u}$. ■ ■

Theorem 5.5.26 (Girard's paradox). *The type $\perp$ is inhabited in λU and hence in $\lambda *$.*

Proof. Note that Proposition 5.5.25 is in contradiction with Corollary 5.5.15, since $\mathbf{I}$ is closed in $\mathbf{U}, <_{\mathbf{i}}$. This shows that $\perp$ is inhabited in λU, so a fortiori in $\lambda *$. ■ ■

In Coquand (1989b) another term inhabiting $\perp$ is constructed. This proof can be carried out in the system λU^- which is the PTS defined as follows:

λU^-		
	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*, *), (\Box, *), (\Box, \Box), (\Delta, \Box)$

The proof is based on a category theoretic derivation of a contradiction due to Reynolds (1984). Note that $\lambda U^- = \lambda\mathrm{HOL}+(\Delta, \Box)$.

In the presence of so-called strong Σs a simpler formalization of the set theoretic paradox 5.5.10 can be formalized, see e.g. Coquand (1986) or Jacobs (1989).

Fully formalized proof of Girard's paradox

As a final souvenir we now show the reader the full term inhabiting $\perp$. The term was presented to us by Leen Helmink who constructed it on an interactive proof development system based on AUTOMATH for arbitrary PTSs. The treatment on his system found an error in an earlier version of this subsection. This kind of use has always been the aim of de Bruijn, who conceived AUTOMATH as a proof checker.

Following the series of intermediate lemmas in this subsection, it became pragmatic to deal with definitions as follows. If we need an expression like

$$C \equiv ---X - X--- \tag{1}$$

where X is defined as M of type A, then we do not fill in the (possibly large) term M for X, but write

$$(\lambda X{:}A.---X - X---)M. \tag{2}$$

This in order to keep expresions manageable. This definition mechanism is also used extensively in functional programming languages like ML. Helmink (1991) shows that if all definitions given as β-redexes are contracted, then the length of the term is multiplied by a factor 72 (so that the term will occupy 215 pages, that is more than this chapter).

Due to the presence of depending types, expressions like (2) are not always legal in a PTS, even if (1) is. {For example working in λU we often needed the expression $\alpha\rightarrow *$ for the type of predicates on α. We want to define

$$\text{Pred} =_{\text{def}} \lambda\alpha{:}\square.\alpha\rightarrow *,$$

and use it as follows:

$$[\lambda\text{Pred}{:}(\square\rightarrow\square).[\lambda R{:}(\text{Pred }\alpha)\ldots](\lambda x{:}\alpha.\perp)---](\lambda\alpha{:}\square.\alpha\rightarrow *). \tag{3}$$

This is illegal for two reasons. First of all $\square\rightarrow\square$ is not allowed in λU. Secondly, the subterm $[\lambda R{:}(\text{Pred }\alpha)\ldots](\lambda x{:}\alpha.\perp)$ is ill formed, since $(\lambda x{:}\alpha.\perp)$ is 'not yet' of type (Pred α).} These phenomena were taken into account by de Bruijn and in the AUTOMATH languages expressions like (3) are allowed. The term that follows is for these reasons only legal in a liberal version of λU.

Glancing over the next pages, the attentive reader that has worked through the proofs in this subsection may experience a free association of the whirling details.

References

VAN BAKEL, S.J.

[1991] Complete restrictions of the intersection type discipline. Theoretical Computer Science 102, 135-163.

BARENDREGT, H.P.

[1984] *The lambda calculus: its syntax and semantics*, revised edition, Studies in Logic and the Foundations of Mathematics, North-Holland.

[1990] Functional programming and lambda calculus, in: VAN LEEUWEN (1990) vol. II, 321–364.

[1991] Introduction to generalised type systems, to appear in *J. Functional Programming*.

BARENDREGT, H.P., M. COPPO and M. DEZANI-CIANCAGLINI

[1983] A filter lambda model and the completeness of type assignment, *J. Symbolic Logic* 48 (4), 931–940.

BARENDREGT, H.P. and K. HEMERIK

[1990] Types in lambda calculi and programming languages, in: *European Symposium on Programming*, ed. N. Jones, Lecture Notes in Computer Science 432, Springer, 1–36.

BARENDREGT, H.P. and W.J.M. DEKKERS

[199-] *Typed lambda calculi*, to appear.

BARENDREGT, H.P. and A. REZUS

[1983] Semantics of classical AUTOMATH and related systems, *Information and Control* 59, 127–147.

BARENDSEN, E.

[1989] Representation of logic, data types and recursive functions in typed lambda calculi, Master's Thesis, Dept. Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

BARENDSEN, E. and J.H. GEUVERS

[1989] Conservativity of λP over PRED, ms. Dept. Computer Science, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

VAN BENTHEM JUTTING, L.S.

[1989] Personal communication.

[199-] Typing in pure type systems, to appear in *Information and Computation*

BERARDI, S.

[1988] Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube, Dept. Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Università di Torino.

[1988a] Personal communication.

[1989] Personal communication.

[1990] Type dependence and constructive mathematics, Ph.D. thesis, Dipartimento Matematica, Università di Torino.

BÖHM, C. and A. BERARDUCCI

[1985] Automatic synthesis of typed λ-programs on term algebras, *Theor. Comput. Sci.* 39, 135–154.

DE BRUIJN, N.G.

[1970] The mathematical language AUTOMATH, its usage and some of its extensions, in: *Symposium on automatic demonstration* (IRIA, Versailles 1968), Lecture Notes in Mathematics 125, Springer, 29–61.

[1980] A survey of the AUTOMATH project, in: HINDLEY and SELDIN (1980), 580–606.

CARDELLI, L. and P. WEGNER

[1985] On understanding types, data abstraction and polymorphism, ACM Comp. Surveys 17-4.

CHURCH, A.

[1932/33]) A set of postulates for the foundation of logic, *Annals of Mathematics* (2) 33, 346–366 and 34, 839–864.

[1940] A formulation of the simple theory of types, *J. Symbolic Logic* 5, 56–68.

[1941] *The calculi of lambda conversion*, Princeton University Press.

COPPO, M.

[1985] A completeness theorem for recursively defined types, in: *Proceedings of the 12th Int. Coll. on Automata and Programming*, Lecture Notes in Computer Science 432, Springer, 120–129

COPPO, M. and F. CARDONE

[1991] Type inference with recursive types: syntax and semantics, *Information and Computation* 92 (1), 48–80.

COPPO, M., M. DEZANI-CIANCAGLINI, G. LONGO and F. HONSELL

[1984] Extended type structures and filter lambda models, in: *Logic Colloquium 82*, eds. G. Lolli, G. Longo and A. Marcja, Studies in Logic and the Foundations of Mathematics, North Holland, 241–262.

COPPO, M., M. DEZANI-CIANCAGLINI and B. VENNERI

[1981] Functional characters of solvable terms, *Zeitschrift f. Mathematische Logik u. Grundlagen der Mathematik* 27, 45–58.

COPPO, M., M. DEZANI-CIANCAGLINI and M. ZACCHI

[1987] Type Theories, normal forms and D_∞ lambda models, *Information and Computation* 72, 85–116.

COQUAND, TH.

[1985] *Une théorie des constructions*, Thèse de troisième cycle, Université Paris VII.

[1986] An analysis of Girard's paradox, in: *Proceedings of the First Symposium of Logic in Computer Science*, IEEE, 227–236.

[1989] Metamathematical investigation of a calculus of constructions, in: ODIFREDDI (1990), 91–122.

[1989] Reynolds paradox with the Type : Type axiom, in: The calculus of constructions, Documentation and users's guide, version 4.10, Rapports Techniques 110, INRIA, B.P. 105, 78153 Le Chesnay Cedex, France, 4 unnumbered pages at the end of the report.

COQUAND, TH. and G. HUET

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

CURRY, H.B.

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

[1969] Modified basic functionality in combinatory logic, *Dialectica* 23, 83–92.

CURRY, H.B. and R. FEYS

[1958] *Combinatory Logic*, Vol. I, Studies in Logic and the Foundations of Mathematics, North Holland.

CURRY, H.B., J.R. HINDLEY and J.P. SELDIN

[1972] *Combinatory Logic*, Vol. II, Studies in Logic and the Foundations of Mathematics, North Holland.

VAN DAALEN, D.T.

[1980] *The language theory of AUTOMATH*, Ph.D. thesis, Technical University Eindhoven, The Netherlands.

VAN DALEN, D.

[1983] *Logic and structure*, 2nd edition, Springer.

DAVIS, M.

[1958] *Computability and unsolvability*, McGraw-Hill.

DEZANI-CIANCAGLINI, M. and I. MARGARIA

[1987] Polymorphic types, fixed-point combinators and continuous lambda models, in: IFIP *Conference on Formal Description of Programming Concepts* III, Ed. M. Wirsing, North-Holland, 425–450.

FITCH, F.B.

[1952] *Symbolic logic, an introduction*, Ronald Press, New York.

[1974] *Elements of combinatory logic*, Yale University Press, New Heaven.

FOKKINGA, M.M.

[1987] Programming languages concepts - the lambda calculus approach, in: *Essays on concepts, formalism, and tools*, eds. P.R.J. Asveld and A. Nijholt, CWI tracts 42, Box 4079, 1009 AB Amsterdam, The Netherlands, 129–162.

FUJITA, K. and TONINO, H.

[1991] Logical systems are generalised type systems, ms. Technical University Delft, Faculty of Mathematics and Informatics, Julianalaan 132, 2628 BL Delft, The Netherlands.

GANDY, R.O.

[1980] Proofs of strong normalisation, in: HINDLEY and SELDIN (1980), 457-478.

GEUVERS, J.H.

[1988] The interpretation of logics in type systems, Master thesis, Dept. Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

[1989] Theory of constructions is not conservative over higher order logic, ms. Dept. Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

[1990] Type systems for higher order logic, ms. Dept. Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

GEUVERS, H. and M.J. NEDERHOF

[1991] A modular proof of strong normalisation for the calculus of constructions, *J. Functional Programming*, 1 (2), 155–189.

GIANNINI, P. and S. RONCHI DELLA ROCA

[1988] Characterisation of typings in polymorphic type discipline, in: *Proceedings of the Third Symposium of Logic in Computer Science*, IEEE, 61–70.

GIRARD, J.-Y.

[1972] *Interprétation foctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*, Ph.D. thesis, Université Paris VII.

GIRARD, J.-Y., Y. LAFONT and P. TAYLOR

[1989] *Proofs and types*, Tracts in Theoretical Computer Science 7, Cambridge University Press.

HARPER, R., F. HONSELL and G. PLOTKIN

[1987] A framework for defining logics, in: *Proceedings Second Symposium of Logic in Computer Science* (Ithaca, N.Y.), IEEE, Washington DC, 194–204.

HELMINK, L.

[1991] Girard's paradox in λU, ms. Philips Research Laboratories, Box 80.000, 5600 JA Eindhoven, The Netherlands.

HENGLEIN, F.

[1990] A lower bound for full polymorphic type inference: Girard-Reynolds typability is DEXPTIME-hard, Report RUU-CS-90-14, Dept. Computer Science, Utrecht University, The Netherlands.

HINDLEY, J.R.

[1969] The principal typescheme of an object in combinatory logic, in: *Trans. Amer. Math. Soc.* 146, 29–60.

[1983] The simple semantics for Coppo-Dezani-Sallé types, in: *International Symposium on Programming*, Eds. M. Dezani-Ciancaglini and H. Montanari, Lecture Notes in Computer Science 137, Springer, Berlin, 212–226.

HINDLEY, J.R. and SELDIN, J.P.

[1980] *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, Academic Press.

[1986] *Introduction to Combinators and λ-calculus*, London Mathematical Society Student Texts 1, Cambridge University Press.

HOFSTADTER, D.
[1979] *Gödel Escher Bach: an eternal golden braid*, Harvester Press.

HOWARD, W.A.
[1980] The formulae-as-types notion of construction, in: HINDLEY and SELDIN (1980), 479–490.

HOWARD, W. A., G. KREISEL, R. J. PARIKH and W. W. TAIT
[1963] Stanford Report, unpublished notes.

HOWE, D.
[1987] The computational behaviour of Girard's paradox, in: *Proceedings of the Second Symposium of Logic in Computer Science* (Ithaca, N.Y.), IEEE, 205–214.

JACOBS, B.P.F.
[1989] The inconsistency of higher order extensions of Martin-Löf's type theory, *J. Philosophical Logic* 18, 399–422.
[1991] *Categorical type theory*, Ph.D. thesis, Dept. Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

JACOBS, B.P.F., I. MARGERIA and M. ZACCHI
[199-] Filter models with polymorphic types, to appear in *Theoretical Computer Science*.

KFOURY, A.J., J. TIURYN and P. URZYCZYN
[1990] ML typability is DEXPTIME-complete, in: CAAP '90, ed. A. Arnold, in: *Lecture Notes in Computer Science* 431, Springer, 206–220.

KLEENE, S.C.
[1936] λ-definability and recursiveness, *Duke Math. J.* 2, 340–353.

KLEENE, S.C. and J.B. ROSSER
[1935] The inconsistency of certain formal logics, *Annals Math.* (2) 36, 630–636.

KLOP, J.-W.
[1980] *Combinatory reduction systems*, Ph.D. thesis, Utrecht University; CWI Tract, Box 4079, 1009 AB Amsterdam, The Netherlands.

KRIVINE, J. L.
[1990] *Lambda-calcul, types et modèles*, Masson, Paris.

LAMBEK, J. and P.J. SCOTT
[1986] *Introduction to higher order categorical logic*, Cambridge Studies in Advanced Mathematics, Cambridge University Press, Cambridge.

LÄUCHLI, H.
[1970] An abstract notion of realizability for which intuitionistic predicate calculus is complete, in: *Intuitionism and Proof Theory*, eds. A. Kino et al., Studies in Logic and the Foundations of Mathematics, North-Holland, 227–234.

LEEUWEN, J. VAN
[1990] *Handbook of Theoretical Computer Science*, Elsevier/MIT Press.

LEIVANT, D.

[1983] Reasoning about functional programs and complexity classes associated with type disciplines, *24th IEEE symposium on foundations of computer science*, 460–469.

[1990] Contracting proofs to programs, in: ODIFREDDI (1990), 279–327

LÖB, M.

[1976] Embedding first order predicate logic in fragments of intuitionistic logic, *J. Symbolic Logic* 41 (4), 705–718.

LONGO, G. and E. MOGGI

[1988] *Constructive natural deduction and its modest interpretation*, Report CMU-CS-88-131, Carnegie Mellon University, Pittsburgh, USA.

LUO, Z.

[1990] *An extended calculus of constructions*, Ph.D. thesis, University of Edinburgh.

MACQUEEN, D., G.D. PLOTKIN and R. SETHI

[1984] An ideal model for recursive polymorphic types, in: *11th ACM Symposium on Principles of Programming Languages*, ACM, 165–174.

MALECKI, S.

[1989] Private communication.

MARTIN-LÖF, P.

[1971] A construction of the provable wellorderings of the theory of species, ms. Mathematical Institute, University of Stockholm, Sweden, 14 pp.

[1984] *Intuitionistic type theory*, Bibliopolis, Napoli.

MENDELSON, E.

[1987] *Introduction to mathematical logic*, third edition, Wadsworth and Brooks/Cole.

MENDLER, N.P.

[1987] Inductive types and type constraints in second-order lambda calculus, in: *Proceedings of the Second Symposium of Logic in Computer Science* (Ithaca, N.Y.), IEEE, 30–36.

MEYER, A.R.

[1988] Personal communication.

MILNER, R.

[1978] A theory of type polymorphism in programming, *J. Computer and Systems Sciences* 17, 348–375.

[1984] A proposal for standard ML, in: *Proceedings of the ACM Symposium on LISP and Functional Programming* (Austin), 184–197.

MITCHELL, J.C.

[1984] Type inference and type containment, in: *Proc. Internat. Symp. on Semantics of Data Types*, ed. G. Kahn, Lecture Notes in Computer Science 173, Springer, 257–277.

[1988] Polymorphic type inference and containment, Inform. and Comput. 76 (2,3), 211–249.

[1990] Type systems for programming languages, in: VAN LEEUWEN (1990), 365–458.

Mirimanoff, D.
[1917] Les antinomies de Russell et de Burali-Forti et le problème fondamental de la théorie des ensembles, *L' Enseignement Mathématique* 19, 37–52

Mostowski, A.
[1951] On the rules of proof in the pure functional calculus of first order, *J. Symbolic Logic* 16, 107–111.

Nederpelt, R.P.
[1973] *Strong normalization in a typed lambda calculus with lambda structured types*, Ph.D. thesis, Eindhoven Technological University, The Netherlands.

Nerode, A. and P. Odifreddi
[199-] *Lambda calculi and constructive logics*, to appear.

Odifreddi, P.
[1990] *Logic in Computer Science*, Academic Press, New York.

Pavlović, D.
[1990] *Predicates and fibrations*, Ph.D. Thesis, Department of mathematics, University of Utrecht, Budapestlaan 6, 3508 TA Utrecht, The Netherlands.

Peremans, W.
[1949] Een opmerking over intuitionistische logica, Report ZW-16, CWI, Box 4079, 1009 AB Amsterdam, The Netherlands.

Pfenning, F.
[1988] Partial polymorphic type inference and higher order unification, in: *Proc. ACM Conference on LISP and Functional Programming*, 153–163.

Prawitz, D.
[1965] *Natural deduction: a proof-theoretical study*, Almqvist and Wiksell, Stockholm.

Quine, W. V. O.
[1963] *Set theory and its logics*, Cambridge, Massachussets.

Renardel de Lavalette, G.R.
[199-] Strictness analysis via abstract interpretation for recursively defined types, to appear in: *Information and Computation*.

Reynolds, J.C.
[1974] Towards a theory of type structure, in: *Mathematical Foundations of Software Development*, eds. Ehring et al., Lecture Notes in Computer Science 19, Springer, 408– 425.
[1984] Polymorphism is not settheoretic, in: *Semantics of data types*, Lecture Notes in Computer Science 173, Springer, Berlin, 145–156.
[1985] Three approaches to type theory, in: *Lecture Notes in Computer Science* 185, Springer, Berlin, 145–146.

Robinson, J.A.
[1965] A machine oriented logic based on the resolution principle, *J. ACM.* 12 (1), 23–41.

Schönfinkel, M.
[1924] Über die Bausteinen der mathematische Logik, *Math. Ann.* 92, 305–316.

Schwichtenberg, H.
[1977] Proof theory: applications of cut-elimination, in: *Handbook of Mathematical Logic*, ed. J. Barwise, North-Holland, 867–895.

Smullyan, R.
[1985] *To mock a mockingbird*, Knopf, New York.

Scott, D.S.
[1976] Data types as lattices, *SIAM J. Comput.* 5, 522–587.

Stenlund, S.
[1972] *Combinators, λ-terms and proof theory*, D. Reidel, Dordrecht.

Swaen, M.D.G.
[1989] *Weak and strong sum-elimination in intuitionistic type theory*, Ph.D. thesis, University of Amsterdam.

Tait, W.W.
[1967] Intensional interpretation of functionals of finite type I, *J. Symbolic Logic* 32, 198–212.
[1975] A realizability interpretation of the theory of species, in: *Logic Colloquium* (Boston), ed. R. Parikh, Lecture Notes in Mathematics 453, Springer, 240–251.

Terlouw, J.
[1982] On definition trees of ordinal recursive functionals: reduction of the recursion orders by means of type level raising, *J. Symbolic Logic* 47 (2), 395–402.
[1989] Een nadere bewijstheoretische analyse van GSTT's, ms. Dept. Computer Science, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

Troelstra, A.S.
[1973] *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics 344, Springer.

Turing, A.M.
[1937] Computability and λ-definability, *J. Symbolic Logic* 2, 153–163.

de Vrijer, R.
[1975] Big trees in a λ-calculus with λ-expressions as types, in: *λ-Calculus and Computer Science Theory*, ed. C. Böhm, Lecture Notes in Computer Science 37, Springer, 252–271

Wadsworth, C.P.
[1971] *Semantics and pragmatics of lambda calculus*, Ph.D. thesis, Oxford University.

Wand, M.
[1987] A simple algorithm and proof for type inference, *Fund. Informaticae* X, 115–122.

WHITEHEAD, A.N. and B. RUSSELL
[1910] *Principia mathematica*, Cambridge University Press.

ZWICKER, W.
[1987] Playing games with games: the hypergame paradox, in: *Amer. Math. Monthly*, 507–514.