The Journal of Logic and
Algebraic Programming 54 (2003) 65–107

THE JOURNAL OF
LOGIC AND
ALGEBRAIC
PROGRAMMING

www.elsevier.com/locate/jlap

Revisiting the notion of function

Fairouz Kamareddine[a,*], Twan Laan[b], Rob Nederpelt[c]

[a] *Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, UK*
[b] *Weerdstede 45, 3431 LS Nieuwegein, The Netherlands*
[c] *Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

Received 18 April 2001; accepted 14 December 2001

Abstract

Functions play a central role in type theory, logic and computation. We describe how the notions of *functionalisation* (the way in which functions can be constructed) and *instantiation* (the process of applying a function to an argument) have been developed in the last century. We explain how both processes were implemented in Frege's *Begriffschrift*, Russell's *Ramified Type Theory*, and the λ-calculus (originally introduced by Church) showing that the λ-calculus misses a crucial aspect of functionalisation. We then pay attention to some special forms of function abstraction that do not exist in the λ-calculus and we show that various logical constructs (e.g., *let expressions* and *definitions* and the use of parameters in mathematics), can be seen as forms of the missing part of functionalisation. Our study of the function concept leads to: (a) an extension of the Barendregt cube [4] with all of definitions, Π-reduction and explicit substitutions giving all their advantages in one system; and (b) a natural refinement of the cube with parameters. We show that in the refined Barendregt cube, systems like AUTOMATH, LF, and ML, can be described more naturally and accurately than in the original cube.

Keywords: Function abstraction and application; Π-reduction; Explicit substitution; Automath

1. Summary

Throughout the last 100 years, many different type systems have been presented. Such type systems played an important role in the foundation and implementation of mathematics and computation. Throughout the history, type theory made a heavy use of the notion of *function*. In the abstract theory of functions, two constructions (which are inverses of each other) can be identified:

[*] Corresponding author. Tel.: +44-131-451-3868; fax: +44-131-451-3327.
E-mail addresses: fairouz@cee.hw.ac.uk (F. Kamareddine), twan.laan@wxs.nl (T. Laan), r.p.nederpelt@tue.nl (R. Nederpelt).

- *Functionalisation* which is the process in which a function is constructed out of an expression. For example: the construction of the function $x \mapsto x + 3$ from the expression $2 + 3$. This function is denoted in the λ-calculus by $\lambda x.(x + 3)$.
- *Instantiation* which is the process in which a function value is calculated when a suitable argument is assigned to a function. For example: the construction of the term $2 + 3$ by applying the function $x \mapsto x + 3$ to the argument 2.

Both functionalisation and instantiation are present in many type theories, logical systems and programming languages. But different theories/systems focus on different forms of these processes. For example, Frege [17] uses a different form of functionalisation than that used by Church in his λ-calculus (see Section 2). In this paper we argue that various extensions of λ-calculi have been proposed in order to accommodate the missing form of functionalisation used by Frege.

In Sections 2 and 3 we give a description of functionalisation and instantiation, and explain to which extent they are present in well-known logical systems and in the λ-calculus. In doing so, we show that functionalisation and instantiation form the heart of function theory, and hence also the heart of type theory. In Section 4 we present the formal machinery. In Sections 5 and 6 we show that the use of different forms of function abstraction, application, explicit substitutions, let expressions, definitions and parameters in type theory is based on the more general functionalisation rather than just the λ-abstraction found in the λ-calculus. In order to overcome the restricted form of functionalisation permitted in the λ-calculus (via function (or λ-) abstraction), we give two extensions of the Barendregt cube of [4]. In Section 5 we give an extension of the cube with explicit substitutions, Π-reduction and definitions which brings all the advantages of these functionalisation features in one system while still guaranteeing all the desirable properties of the cube. In Section 6 we give a concrete refinement of the cube in which we can go half way as well as all the way in the functionalisation process. This refinement shows that indeed, we can place systems such as AUTOMATH and LF on the refined cube, whereas they could not be accurately placed on the usual cube. Again, we establish all the desirable properties. We conclude in Section 7.

2. Functionalisation

Consider the mathematical expression $2 + 3$. We can replace the object 2 by a symbol that denotes "any natural number": a variable (say: x). This results in the expression $x + 3$. This expression does not denote one specific natural number unless we replace x by a natural number. This replacement-activity gives rise to an algorithm: We feed a natural number to the algorithm, the algorithm adds 3 to that natural number, and returns the result to us. Notice that the algorithm that returns $x + 3$ whenever we assign a natural number to x is different from the expression $x + 3$. Moreover, the expression $y + 3$ is an expression that is different from $x + 3$. But if we construct functions from $x + 3$ and $y + 3$ by the method described above, we obtain the same algorithm.

There are various ways to denote the algorithm in the example above:

- Frege [17] simply wrote $x + 3$, and did not distinguish the algorithm from the expression $x + 3$;[1]
- Whitehead and Russell [42] wrote $\hat{x} + 3$ to distinguish the algorithm from the expression $x + 3$;

[1] Frege used $\grave{x}(x + 3)$ to denote the *course-of-value* of this algorithm but not the algorithm itself.

- Church [12] used $\lambda x.(x+3)$ where Russell wrote $\hat{x}+3$;
- In the proof checker AUTOMATH ([34], 1968), the notation $[x{:}\mathbb{N}](x+3)$ is used;
- In explicitly typed λ-calculi (also known as λ-calculi "in Church-style") one writes $\lambda x{:}\mathbb{N}.(x+3)$. The $x{:}\mathbb{N}$ denotes that the algorithm requires "special food" (natural numbers);
- In many mathematical texts, we find the notation $x \mapsto x+3$.

Thus, the process of constructing a function (*Functionalisation*) can be split up into two parts:

Definition 1 (*Splitting functionalisation*).

- *Abstraction from a subexpression.* First, we replace a subexpression k in an expression f by a variable x, at one or more places where k occurs in f. Thus we obtain a new expression, f'. For instance, in the expression $2+3$ we can replace the subexpression 2 by a variable x, thus obtaining $x+3$.
- *Function construction.* Then we construct the function $\lambda x.f'$ that assigns the expression $f'[x{:=}a]$ to a value a. Here, $f'[x{:=}a]$ denotes f', in which a has been substituted for x.

The first part (abstraction from a subexpression), is present in Frege's *Begriffsschrift*:

"If in an expression, [...] a simple or a compound sign has one or more occurrences and if we regard that sign as replaceable in all or some of these occurrences by something else (but everywhere by the same thing), then we call the part that remains invariant in the expression a function, and the replaceable part the argument of the function".

(*Begriffsschrift*, Section 9)

Frege, did not employ the second part of functionalisation, the function *construction*.

However, both parts of the functionalisation process are present in *Principia Mathematica* by Whitehead and Russell [42]. The first part is represented by the "vice versa" part of $*9{\cdot}14$ below, and the combination of the first and the second part is represented by $*9{\cdot}15$ below:

"$*\mathbf{9}\cdot\mathbf{14}$. If '$\varphi x$' is significant, then if x is of the same type as a, 'φa' is significant, and vice versa.

$*\mathbf{9}\cdot\mathbf{15}$. If, for some a, there is a proposition φa, then there is a function $\varphi\hat{x}$, and vice versa"

(*Principia Mathematica*, p. 133)

Here, φx denotes an expression in which a variable x occurs. Similarly, φa denotes the expression in which a sub-expression a occurs on the place(s) where x occurs in φx, and $\varphi\hat{x}$ denotes the function (algorithm) that assigns the value φa to an argument a. Note that the function ϕ is not used as a separate entity but always has an argument, be it x, a or $\hat{x}$. Indeed, in Laan's formalisation of propositional functions of *Principia Mathematica* (cf. [30, Definition 2.3]), we see that e.g. $R(x)$ and $S(x, y)$ are propositional functions but R and S alone are not. However, the propositional function $R(x)$ translates into λ-calculus as $\lambda x.Rx$, since λ-abstraction is the only abstraction mechanism available, and $\lambda x.Rx$ can only be typed in λ-calculus if R can be typed on its own.

Remark 2. Both *Begriffsschrift* and *Principia Mathematica* exclude *constant* functions. A function like $\lambda x.3$ cannot be constructed in these theories, because the expression 3 does not contain x. This class of functions can be obtained by weakening the procedure of abstraction of subexpressions of the functionalisation procedure: We can replace an object in an expression f by a variable x at *zero* places where this object occurs in f (the object does not have to *occur* in f). If we then apply the second part of the functionalisation procedure, we can obtain a constant function like $\lambda x.3$.

Functions of more variables can be constructed by repeatedly applying functionalisation. This repetition process is often called "currying"after H.B. Curry, and is usually accredited to Schönfinkel ([39], 1924) but some of the ideas behind it are already present in Frege's *Begriffsschrift* (1879):

> "If, given a function, we think of a sign[2] that was hitherto regarded as not replaceable as being replaceable at some or all of its occurrences, then by adopting this conception we obtain a function that has a new argument in addition to those it had before."
>
> (*Begriffsschrift*, Section 9)

The λ-calculus does not have special notation for abstraction from a subexpression. It only has function construction in the form of $\lambda x.a$.

But the first step of functionalisation can also be made clear:

Example 3. The term $2 + 3$ is β-equivalent to the λ-term $(\lambda x.(x + 3))2$, which can be regarded as the term $\lambda x.(x + 3)$ (a *function*) applied to an *argument*, viz. 2. More precisely, $(\lambda x.(x + 3))2$ is a β-expansion of $2 + 3$. In this β-expansion, both the newly introduced variable, x, and the object that is replaced by x, namely 2, are present. They are linked via the λ-abstractor. By removing the argument 2 and the 'head' λx from $(\lambda x.(x + 3))2$, we obtain the term $x + 3$.

The notion of β-expansion makes it possible to construct more complicated examples.

Example 4. $(\lambda z.z23)(\lambda xy.(x + y))$ is a β-expansion of 2+3. Removing argument $\lambda xy.(x + y)$ and head λz results in $z23$.

More generally, we can construct a function from a λ-term f by first taking a β-expansion $(\lambda x.f')a$ of f, and then removing the argument a and the part λx. This leads to the following definition:

Definition 5 (*Translating functionalisation into λ-calculus*).

- **Abstraction from a subexpression.** Let M, N be λ-terms. We say that *Ncan be obtained from $M$by abstraction from a subexpression* if there are x and a, such that $(\lambda x.N)a \to_\beta M$ via head reduction (so $M \equiv N[x := a]$).
- **Function construction.** Let f be a λ-term. We say that *the term $\lambda x.f$ can be obtained from the term f by function construction*.
- **Functionalisation.** Let M, P be λ-terms. We say that *P can be obtained from M by functionalisation* if there is N such that N can be obtained from M by abstraction from a subexpression, and P can be obtained from N by function construction.

[2] We can now regard a sign that previously was considered replaceable as replaceable also in those places in which up to this point it was considered fixed (footnote by Frege).

(We allow both 'abstraction from a subexpression' and 'function construction' to be repeatedly applied.) This is a more precise description of functionalisation than that of Frege.

3. Instantiation

Instantiation is the inverse of functionalisation (cf. Fig. 1). It consists of applying a function to an argument and calculating the result of this application. As the function is an algorithm, it is prescribed how this calculation should be made.

Definition 6 (*Splitting instantiation*).
1. *Application construction.* Juxtaposing the function f to an argument a; the result is usually written as $f(a)$ or fa and denotes an *intended* function application. For instance: applying the function $\lambda x.(x+3)$ to the argument 2 leads to the intended function application $(\lambda x.(x+3))2$.
2. *Concretisation to a subexpression.* The result of this intended application is calculated via *β-reduction*: if f is a function of the free variable x, then this calculation consists of the substitution of a for x. In our example: 2 is substituted for x, for every occurrence of x in $x+3$. The result is: $2+3$.

It is not always that simple. If f is not of the form $\lambda x.f'$, then application construction cannot be followed by the concretisation to a subexpression. For example, f may be the single variable z. In the λ-calculus, z may be applied to 2, resulting in the expression $z2$, but no further calculation is possible. As with functionalisation, instantiation can be precisely defined in terms of the λ-calculus.

Definition 7 (*Translating instantiation into λ-calculus*).
- **Application construction.** We say that a term $(\lambda x.f)a$ can be obtained from $\lambda x.f$ and a by *application construction.*
- **Concretisation to a subexpression.** We say that a term M can be obtained from a term $(\lambda x.f)a$ if $M \equiv f[x := a]$.
- **Instantiation.** We say that a term M can be obtained from $\lambda x.f$ and a by instantiation if $M \equiv f[x := a]$.

In the works of Frege and Russell, we do not find such a precise description of instantiation. The application construction is well-described (for instance in the "vice-versa"

$$\begin{tikzcd}[column sep=huge]
2+3 \arrow[r, shift left, "\text{abstraction from a subexpression}"] \arrow[rr, bend left, "\text{functionalisation}"] & \begin{array}{c} x+3 \\ (\lambda x.(x+3))2 \end{array} \arrow[l, shift left, "\text{concretisation to a subexpression}"] \arrow[r, shift left, "\text{function construction}"] & \lambda x.(x+3) \arrow[l, shift left, "\text{application construction}"] \arrow[ll, bend left, "\text{instantiation}"]
\end{tikzcd}$$

Fig. 1. Functionalisation and instantiation are each others inverse.

part of ∗9·15 in *Principia Mathematica*—see the quotation above), but there is no precise definition of the concretisation to subexpressions by means of substitution. This is not so very important as long as it is straightforward how the substitution should be carried out. However, we see in [42] that substitution is not a straightforward procedure in *Principia Mathematica*.

The precise definition of substitution in the λ-calculus is due to Curry and Feys [16]. However, we must remark that "precise" is a relative notion here. The presentations of functionalisation and instantiation that were given by Frege and Russell were very precise for those days. And the definition of substitution by Curry and Feys in 1958 is not the last word to be said on the notion of substitution. Currently, there is quite some research on so-called *explicit* substitutions (cf. [29] for a summary), which are refinements of the notion of substitution of Curry and Feys.

4. The formal machinery and the Barendregt cube

Functionalisation and instantiation as used in λ-calculus are quite powerful, but have some disadvantages. The functionalisation part 'abstraction from a subexpression' is missing. In this paper, we extend λ-calculus and type theory with various forms of this missing feature. We devote this section to introducing the Barendregt cube which will be used as basis of our extensions.

In the Barendregt cube of [4], eight well-known type systems are presented in a uniform way. The weakest systems of the cube is Church's simply typed λ-calculus λ→ [14], and the strongest system is the Calculus of Constructions λC [15]. The second order λ-calculus [19,38] discovered independently by Girard and Reynolds figures on the cube between λ→ and λC. Moreover, via the Propositions-as-Types principle (see [23]), many logical systems can be described in the systems of the cube, see [18].

In the cube, we have in addition to the usual λ-abstraction, a type forming operator Π. Briefly, if A is a type, and B is a type possibly containing the variable x, then $\Pi x{:}A.B$ is the type of functions that, given a term $a : A$, output a value of type $B[x := a]$. Here $a : A$ expresses that a is of type A, and $B[x := a]$ means the result of the substitution of a for x in B. If x does not occur in B, then $\Pi x{:}A.B$ is the type of functions from A to B, written $A \to B$. To the Π-abstraction at the level of types corresponds λ-abstraction at the level of objects. Roughly speaking, if M is a term of type B (M and B possibly containing x), then $\lambda x{:}A.M$ is a term of type $\Pi x{:}A.B$. The cube has two sorts $*$ (the set of types) and $\Box$ (the set of kinds) with $* : \Box$. If $A : *$ (resp. $A : \Box$) we say A is a type (resp. a kind). All systems of the cube have the same typing rules but are distinguished from one another by the set $\boldsymbol{R}$ of pairs of sorts (s_1, s_2) allowed in the so-called *type-formation* or *Π-formation* rule (Π). Each system of the cube has its own set $\boldsymbol{R}$ (which must contain $(*, *)$). A Π-type can only be formed in a specific system of the cube if rule (Π) is satisfied for some (s_1, s_2) in the set $\boldsymbol{R}$ of that system. The rule (Π) is as follows:

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_2} \quad (s_1, s_2) \in \boldsymbol{R}.$$

Note that as there are only two sorts, $*$ and $\Box$, and as each set $\boldsymbol{R}$ must contain $(*, *)$, there are only eight possible different systems of the cube (see Fig. 2). The dependencies

$\lambda\!\rightarrow$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
$\lambda\mathrm{P}$	$(*,*)$		$(*,\Box)$	
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
$\lambda\mathrm{P}2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda\mathrm{P}\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda\mathrm{C}$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

Fig. 2. Different type formation conditions.

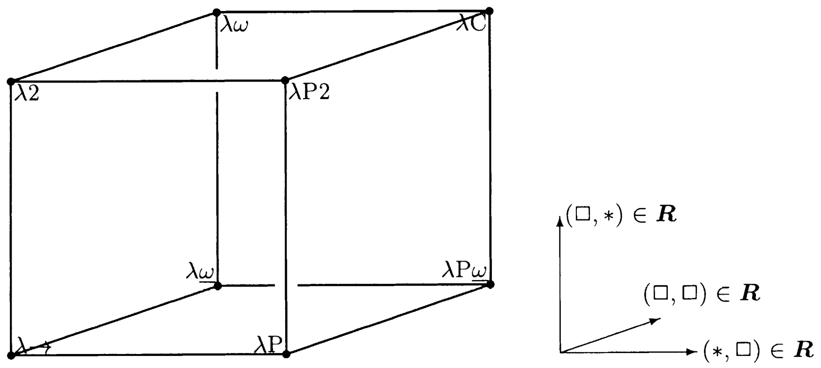
$\lambda\omega$
$\lambda\mathrm{C}$
$\lambda 2$
$\lambda\mathrm{P}2$
$\lambda\underline{\omega}$
$\lambda\mathrm{P}\underline{\omega}$
$\lambda\!\rightarrow$
$\lambda\mathrm{P}$
$(\Box,*) \in \boldsymbol{R}$
$(\Box,\Box) \in \boldsymbol{R}$
$(*,\Box) \in \boldsymbol{R}$

Fig. 3. The Barendregt cube.

System	Related system	Names, references
$\lambda\!\rightarrow$	λ^{τ}	simply typed λ-calculus; (Appendix A), (Chapter 14)
$\lambda 2$	F	second order typed λ-calculus
$\lambda\mathrm{P}$	AUT-QE	
	LF	
$\lambda\mathrm{P}2$		
$\lambda\underline{\omega}$	POLYREC	
$\lambda\omega$	Fω	
$\lambda\mathrm{C}$	CC	Calculus of Constructions

Fig. 4. Systems of the Barendregt cube [3,10,14,15,19,20,22,32,37,38].

between these systems is depicted in the Barendregt cube (Fig. 3). Furthermore, the systems in the cube are related to other type systems as is shown in the overview of Fig. 4 (see [4]). With the rule (*Π*), an important aspect of the cube is that it provides a factorisation of the expressive power of the Calculus of Constructions into three features: *polymorphism*, *type constructors*, and *dependent types*:

- $(*,*)$ is the basic rule that forms types. All type systems of the cube have this rule.
- $(\Box,*)$ is the rule that takes care of polymorphism. Girard's System (also known as $\lambda 2$) is the weakest system on the cube that features this rule.
- $(\Box,\Box)$ takes care of type constructors. The system $\lambda\underline{\omega}$ is the weakest system on the cube that features this rule.
- $(*,\Box)$ takes care of term dependent types. The system $\lambda\mathrm{P}$ is the weakest system on the cube that features this rule.

Definition 8 (*Terms of the ordinary cube*). The set $\mathscr{T}$ of terms of the ordinary cube is defined by the following abstract syntax: $\mathscr{T} ::= * \mid \square \mid \mathscr{V} \mid \pi\mathscr{V}{:}\mathscr{T}.\mathscr{T} \mid \mathscr{T}\mathscr{T}$, where $\pi \in \{\lambda, \Pi\}$.

Notations and Conventions 9. Throughout the paper, we take $\mathscr{V}$ to be a set of variables over which, x, y, z, x_1, etc. range. We take capital letters A, B, etc. sometimes also indexed by Arabic numerals such as A_1, A_2, to range over terms. We use $\text{FV}(A)$ resp. $\text{BV}(A)$ to denote the free resp. bound variables of A, and $A[x := B]$ to denote the (implicit) substitution of all the free occurrences of x in A by B. We assume familiarity with the notion of *compatibility*. As usual, we take terms to be equivalent up to variable renaming and let $\equiv$ denote syntactic equality. We also assume the Barendregt convention (BC) where names of bound variables are always chosen so that they differ from free ones in a term and where different abstraction operators bind different variables. Hence, we will write $(\pi_{y:A}.y)x$ instead of $(\pi_{x:A}.x)x$. This convention (BC) will also be assumed for contexts and typings (for each of the calculi presented) so that for example for the ordinary cube, if $\Gamma \vdash \lambda x : A.B : C$, then x will not occur in Γ.

Definition 10 (*Declarations*, *contexts*, $\subseteq'$ *in the ordinary cube*).

1. A *declaration* d is of the form $x : A$. We define $\mathtt{var}(d) \equiv x$, $\mathtt{type}(d) \equiv A$ and $\text{FV}(d) \equiv \text{FV}(A)$.
2. We let d, d', d_1, etc. range over declarations.
3. A *context* Γ is a (possibly empty) concatenation of declarations $d_1, d_2, \ldots, d_n$ such that if $i \neq j$, then $\mathtt{var}(d_i) \not\equiv \mathtt{var}(d_j)$. We define $\text{DOM}(\Gamma) = \{\mathtt{var}(d) \mid d \in \Gamma\}$. The *empty context* is denoted throughout by $\langle\rangle$ or simply by $\emptyset$. We use Γ, Γ_1, Δ, etc. as meta-variables for contexts.
4. We define substitutions on contexts by: $\emptyset[x := A] \equiv \emptyset$, and $(\Gamma, y : B)[x := A] \equiv \Gamma[x := A], y : B[x := A]$.
5. Define $\subseteq'$ between contexts as the least reflexive transitive relation satisfying $\Gamma, \Delta \subseteq' \Gamma, d, \Delta$ for d a declaration.

Definition 11 (*β-reduction in the ordinary cube*). The cube uses β-reduction where $\to_\beta$ is defined as the compatible closure of $(\lambda x : A.B)C \to_\beta B[x := C]$. As usual, $\twoheadrightarrow_\beta$ is defined as the reflexive transitive closure of $\to_\beta$ and $=_\beta$ is defined as the equivalence closure of $\to_\beta$. We write $\twoheadrightarrow_\beta^+$ to denote β-reduction in one or more steps.

Definition 12 (*Systems of the Barendregt cube*). Let $\boldsymbol{R}$ be a subset of $\{(*, *), (*, \square), (\square, *), (\square, \square)\}$ such that $(*, *) \in \boldsymbol{R}$. The type system $\lambda\boldsymbol{R}$ describes in which ways judgements $\Gamma \vdash_{\boldsymbol{R}} A : B$ (or $\Gamma \vdash A : B$, if it is clear which $\boldsymbol{R}$ is used) can be derived. $\Gamma \vdash A : B$ states that A has type B in context Γ. The typing rules are inductively defined as follows ($s, s_1, s_2 \in \{*, \square\}$):

(axiom) $$\langle\rangle \vdash * : \square$$

(start) $$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad x \notin \text{DOM}(\Gamma)$$

(weak) $$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B} \qquad x \notin \text{DOM}(\Gamma)$$

(*Π*) $$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_2} \qquad (s_1, s_2) \in \boldsymbol{R}$$

(λ) $$\frac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)}$$

(appl) $$\frac{\Gamma \vdash F : (\Pi x{:}A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$$

(conv) $$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$$

Definition 13 (*Statement*, *judgement*, *legal context and term*). Let Γ be a context, $A, B, C \in \mathscr{T}$.

1. $A : B$ is called a *statement*. A and B are its *subject* and *predicate* respectively.
2. $\Gamma \vdash A : B$ is called a *judgement*, and $\Gamma \vdash A : B : C$ denotes $\Gamma \vdash A : B \wedge \Gamma \vdash B : C$.
3. Γ is called *legal* if $\exists A_1, B_1 \in \mathscr{T}$ such that $\Gamma \vdash A_1 : B_1$.
4. A is called a *Γ-term* if $\exists B_1 \in \mathscr{T}[\Gamma \vdash A : B_1 \vee \Gamma \vdash B_1 : A]$. A is called *legal* if $\exists \Gamma_1[A$ is a Γ_1-term$]$.
5. If d is a declaration, then $\Gamma \vdash d$ iff $\Gamma \vdash \mathtt{var}(d) : \mathtt{type}(d)$.

5. Extending the cube with definitions, substitutions and *Π*-reduction

In this section we discuss three extensions of the cube which are all related to the functionalisation/instantiation process explained in Sections 2 and 3. These extensions are: definitions, substitution and *Π*-reduction. We also introduce a system which incorporates these three notions and establish that it has all the desirable meta-theoretical properties. We show the usefulness of each of the three concepts of functionalisation and claim that a system with the combination of the three can be advantageous since it combines the benefits.

5.1. Definitions

In the λ-calculus, the first step in the functionalisation process (the abstraction from a subexpression) is not carried out. In particular, unlike the systems of Frege and Russell, the functionalisation process in the λ-calculus does not show from which term (object), the function has been abstracted.

There are many modern functionalisation processes in which it is essential to remember the original term from which the function has been abstracted. A good example is the use of *definitions* or *let expression* in functional languages like Haskell and ML and theorem provers like AUTOMATH and Coq. If k occurs in a text f (such a text can be a single expression or a list of expressions, for example a book), it is sometimes practical to introduce an abbreviation for k, for several reasons:

- The syntactical representation of the object k may be long. This makes manipulations with f a time- and memory-consuming task, in particular when k occurs several times in f. Abbreviating k can make manipulations with f easier. This has been exploited in the theorem prover AUTOMATH where definitions are heavily present. Similarly, programming languages make heavy use of such concept (referred to as *let expressions*) where examples like

$$\texttt{let}\ \texttt{add}_2 = (\lambda x.x + 2)\ \texttt{in}\ \texttt{add}_2(y) + \texttt{add}_2(z)$$

often occur. An important part of such a let expression is the abstraction from a subexpression where in $(\lambda x.x + 2)y + (\lambda x.x + 2)z$, we replace $(\lambda x.x + 2)$ by $\texttt{add}_2$ obtaining $\texttt{add}_2(y) + \texttt{add}_2(z)$.

- The object k may represent a structure that is particularly interesting. Abbreviating k opens the possibility to introduce a significant name for k. This makes the expression easier to understand for human beings.

Often, it is a combination of reasons which makes it useful to introduce an abbreviation (i.e. a definition). Abbreviating k can be seen as a functionalisation process: we replace all the occurrences of k by its definiendum (its name), and then have an *unfolding algorithm* that can be used to replace the definiendum by its definiens k when the internal structure of k is needed in the term f.

The situation can be described precisely in the functionalisation/instantiation format. Consider the example

$$\texttt{let}\ y{=}\texttt{S(S0)}\ \texttt{in}\ y + y.$$

We write here variable y instead of (the expected) construct 2, since we want to emphasize the similarity of this let-construct to the term $(\lambda y.(y + y))(S(S0))$. Now, how can the let-construct be composed from the original term $S(S0) + S(S0)$? First, 'abstraction from a subexpression' gives $y + y$, then 'function abstraction' delivers $\lambda y.(y + y)$ and finally, 'application construction' gives $(\lambda y.(y + y))(S(S0))$, which is similar to $\texttt{let}\ y{=}\texttt{S(S0)}\ \texttt{in}\ y + y$.

Storing the definition in a context is usually done for definitions that are used at several places, in several expressions; storing the definition in front of a term usually takes place when the definition is important for that term only.

Definitions are not only useful for the above mentioned reasons. They have been essential to restore the well-behavedness to various extensions of type theory. For example, [9] resp. [24] resp. [8] used definitions to restore subject reduction to the cube extended with generalised reduction, resp. Π-reduction resp. explicit substitutions. In this section, we present for the first time, a single extension of the cube with all of definitions, explicit substitutions and Π-reduction. We ignore generalised reduction as it needs the item notation of [27] in order to be described clearly.

5.2. Explicit substitutions

In the λ-calculus, although we could artificially view $x + 3$ as an abstraction from a subexpression (say $2 + 3$), this is not useful because we cannot apply $x + 3$ to an argument (say 2 to get $(x + 3)2 = 2 + 3$). Hence, $x + 3$ cannot be treated as a function. The explosion of works in explicit substitutions in the λ-calculus [1,5,7,29] in the past two decades could be viewed as an attempt to treat expressions like $x + 3$ as functions and apply them to arguments. Hence, in these accounts, the following holds in the λ-calculus extended with explicit substitutions: $(x + 3)[x \leftarrow 2] \rightarrow_{\sigma} 2 + 3$, where $(x + 3)[x \leftarrow 2]$ is internal to the λ-calculus with explicit substitutions rather than being an external (meta-)operation as in the usual λ-calculus.

In any real implementation of the λ-calculus (without explicit substitution), the substitution required by β-reduction (and similar higher-order operations) must be implemented via smaller operations. Thus, there is a conceptual gap between the theory of the λ-calculus and its implementation in programming languages and proof assistants.

By representing substitutions in the structure of terms and by providing (step-wise) reductions to propagate the substitutions, explicit substitution provides a number of benefits:

- Explicit substitution allows more flexibility in ordering work. Propagating substitutions through a particular subterm can wait until the subterm is the focus of computation. This allows all of these substitutions to be done at once. Obtaining more control over the ordering of work has become an important issue in functional programming language implementation (cf. [35]).
- The flexibility provided by explicit substitution also allows postponing unneeded work indefinitely (i.e., avoiding it completely). This can yield profits, since implicit substitution can be an inefficient, maybe even exploding, process by the many repetitions it causes.
- Explicit substitution allows formal modeling of the techniques used in real implementations, e.g., environments. Because explicit substitution is closer to real implementations, it has the potential to provide a more accurate cost model.

Again, there is a functionalisation/instantiation process behind explicit substitution. Consider the example $(x + 3)[x \leftarrow 2]$. This can be constructed as follows from the term $2 + 3$: 'abstraction from a subexpression' gives $x + 3$, 'function construction' then yields $\lambda x.(x + 3)$ and 'application construction' finally gives the term $(\lambda x.(x + 3))2$, which is comparable to the term $(x + 3)[x \leftarrow 2]$. Note, however, that this similarity does not go as far as that we allow β-reduction. In fact, $(x + 3)[x \leftarrow 2]$ reduces in *two* steps to $2 + 3$:

$$(x + 3)[x \leftarrow 2] \rightarrow x[x \leftarrow 2] + 3[x \leftarrow 2] \rightarrow 2 + 3,$$

and not in one step, as is suggested by the 'alternative' notation $(\lambda x.(x + 3))2$.

5.3. Π-reductions

Type theory has almost always been studied without Π-conversion (which is the analogue of β-conversion on product type level). That is: $(\lambda x : A.b)C \rightarrow_{\beta} b[x := C]$ is always assumed but not: $(\Pi x : A.B)C \rightarrow_{\Pi} B[x := C]$. The exceptions to this are: some AUTOMATH-languages (see [34]), the λ-cube extended with Π-reduction in [24,26] and the intermediate language in compilers for source languages as in [36]. Kamareddine et al. [24] claims that $\rightarrow_{\Pi}$ is desirable for the following reasons:

1. ***Π* is, in a sense, a kind of λ.** In higher order type theory, arrow-types of the form $A \rightarrow B$ are replaced by dependent products $\Pi_{x:A}.B$, where x may be free in B, and thus B *depends on* x. This means that abstraction can be over types: $\Pi_{x:A}.B$ as well as over terms: $\lambda_{x:A}.b$. But, once we allow abstraction over types, it would be nice to discuss the reduction rules which govern these types. In fact, Π is indeed a kind of λ as regards the abstraction over a variable and hence is eligible for an application.
2. **Compatibility.** Look at the (λ) and (appl) rules of Definition 12. The (λ) rule may be regarded as the compatibility property for typing with respect to abstraction. That

is: $b : B$ implies $\lambda x : A.b : \Pi x : A.B$. The compatibility property for the typing with respect to application is lost however. In fact, from the (appl) rule, $F : \Pi x : A.B$ implies $Fa : B[x := a]$ instead of $Fa : (\Pi x : A.B)a$. To get compatibility for typing with respect to application, one needs to add $\to_{\Pi}$-reduction and to change the (appl) rule to:

$$\text{(new appl)} \quad \frac{\Gamma \vdash F : \Pi x : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : (\Pi x : A.B)a}.$$

Of course, one needs now to incorporate, apart from β-reduction, also Π-reduction (see above).

3. **The AUTOMATH experience.** One might argue that *implicit* Π-reduction (as is the case of the ordinary cube with the (appl) rule) is closer to intuition in the most usual applications. However, experiences with the AUTOMATH-languages [34], containing *explicit* Π-reduction, demonstrated that there exists no formal or informal objection against the use of this explicit Π-reduction in natural applications of type systems.
4. **Programming languages.** In programming language studies, a thriving area is that of the use of richly-typed intermediate languages in sophisticated compilers for higher-order, typed source languages. The recently developed language [36] aims at reducing the number of data types and the volume of code required in the compiler, by avoiding duplications. To do this, [36] uses the whole λ-cube extended with Π-reduction and gives the following motivation:
   - With the old application rule, matters get very complicated when one adds further expressions (such as let and case).
   - In a compiler, Π-reduction allows to separate the type finder from the evaluator since $\vdash$ no longer mentions substitution. One first extracts the type and only then evaluates it.

Π-reduction as described above, also fits in the functionalisation/instantiation format. From term B we obtain by 'function construction' $\Pi x : A.B$, and then 'application construction' gives $(\Pi x : A.B)a$.

5.4. The extension Π_{DEF}-cube

Terms of the cube extended with definitions, substitutions and Π-reduction ($\Pi\sigma$DEF-cube) differ from those of the ordinary cube given in Definition 8 by the presence of explicit substitution terms.

Definition 14 (*Terms of the cube extended with definitions, substitutions and Π-reduction*). The set $\mathscr{T}_e$ of terms of the $\Pi\sigma$DEF-cube is defined by: $\mathscr{T}_e ::= * \mid \Box \mid \mathscr{V} \mid \pi\mathscr{V}{:}\mathscr{T}_e.\mathscr{T}_e \mid \mathscr{T}_e\mathscr{T}_e \mid \mathscr{T}_e[\mathscr{V} \leftarrow \mathscr{T}_e]$, where $\pi \in \{\Pi, \lambda\}$.

In Notations and Conventions 9, the notions of $\mathrm{FV}(A)$, $\mathrm{BV}(A)$, implicit substitution $A[x := B]$ and compatibility are extended to take into account the new explicit substitution terms of the form $A[x \leftarrow B]$. In particular, $\mathrm{FV}(A[x \leftarrow B]) = \mathrm{FV}(A) \setminus \{x\}) \cup \mathrm{FV}(B)$ and $(A[x \leftarrow B])[y := C] \equiv (A[y := C])[x \leftarrow B[y := C]]$. In addition, Barendregt's Convention BC is extended so that a term $(\pi y{:}A.B)[y \leftarrow C]$ will be automatically renamed to $(\pi x : A.B[y := x])[y \leftarrow C]$ where x is fresh.

Contexts Γ (see Definition 10) are now a list of declarations of the form $x : A$ or of *definitions* of the form $x = B : A$. These latter definitions define x to be B and to have the type A.

Definition 15 (*Declarations*, *definitions*, *contexts*, $\subseteq'$ *in* $\Pi\sigma$DEF-*cube*).
1. A *declaration* d is as in Definition 10.
2. A *definition* d is of the form $x = B : A$ and defines x of type A to be B. We define $\mathtt{var}(d)$, $\mathtt{type}(d)$ and $\mathtt{ab}(d)$ to be x, A, and B respectively. We define $\mathrm{FV}\,(d) \equiv \mathrm{FV}\,(A) \cup \mathrm{FV}\,(B)$.
3. We use $d, d', d_1, \ldots$ to range over declarations and definitions.
4. A *context* Γ is a (possibly empty) concatenation of declarations and definitions $d_1, d_2, \ldots, d_n$ such that if $i \neq j$, then $\mathtt{var}(d_i) \not\equiv \mathtt{var}(d_j)$. We define $\mathrm{DOM}\,(\Gamma) = \{\mathtt{var}(d) \mid d \in \Gamma\}$, $\Gamma\text{-}\mathtt{decl} = \{d \in \Gamma \mid d \text{ is a declaration}\}$ and $\Gamma\text{-}\mathtt{abb} = \{d \in \Gamma \mid d \text{ is a definition}\}$. Note that $\mathrm{DOM}\,(\Gamma) = \{\mathtt{var}(d) \mid d \in \Gamma\text{-}\mathtt{decl} \cup \Gamma\text{-}\mathtt{abb}\}$. We use $\Gamma, \Delta, \Gamma', \Gamma_1, \Gamma_2, \ldots$ to range over contexts.
5. We define substitutions on contexts by:
$$\emptyset[x := A] \equiv \emptyset,\ (\Gamma, y : B)[x := A] \equiv \Gamma[x := A], y : B[x := A], \quad \text{and}$$
$$(\Gamma, y = B : C)[x := A] \equiv \Gamma[x := A], y = B[x := A] : C[x := A].$$
6. Define $\subseteq'$ between contexts as the least reflexive transitive relation satisfying:
   - $\Gamma, \Delta \subseteq' \Gamma, d, \Delta$ for d a declaration or a definition.
   - $\Gamma, x : A, \Delta \subseteq' \Gamma, x = B : A, \Delta$.

Recall that without explicit substitution, Π-reduction is defined as the compatible closure of $(\Pi x : A.B)C \to_\Pi B[x := C]$. Similarly to β-reduction, one can define $\twoheadrightarrow_\Pi$, $\twoheadrightarrow^+_\Pi$ and $=_\Pi$. We define $\beta\Pi$-reduction as the union of $\to_\beta$ and $\to_\Pi$ and as usual write $\twoheadrightarrow_{\beta\Pi}$ and $=_{\beta\Pi}$ to be the reflexive transitive respectively equivalence closures of $\to_{\beta\Pi}$. We write $\twoheadrightarrow^+_{\beta\Pi}$ to denote $\beta\Pi$-reduction in one or more steps. With explicit substitution, β-reduction and Π-reduction will both create substitution terms. The first two rules in the definition below take this into account. The presence of explicit substitution terms calls for σ-reduction.

Definition 16 (*e-Reduction*). e-Reduction $\to_e$ is defined as the union of $\to_{\beta'}$, $\to_{\Pi'}$ and $\to_\sigma$ which are defined as the compatible closures of, respectively:

$$\begin{array}{lll}
(\lambda x : A.B)C & \to_{\beta'} & B[x \leftarrow C] \\
(\Pi x : A.B)C & \to_{\Pi'} & B[x \leftarrow C] \\
(\pi y : A.B)[x \leftarrow C] & \to_\sigma & \pi y : A[x \leftarrow C].B[x \leftarrow C] \quad \text{for } \pi \in \{\lambda, \Pi\} \\
(AB)[x \leftarrow C] & \to_\sigma & A[x \leftarrow C].B[x \leftarrow C] \\
x[x \leftarrow C] & \to_\sigma & C \\
A[x \leftarrow C] & \to_\sigma & A \quad \text{if } x \notin \mathrm{FV}\,(A)\,.
\end{array}$$

As usual, we define $\twoheadrightarrow_e$ resp. $=_e$ to be the reflexive, transitive resp. the equivalence closure of $\to_e$. In the usual way we define also $\twoheadrightarrow_r$, $\twoheadrightarrow^+_r$ and $=_r$ for $r \in \{\sigma, \beta', \beta'\Pi', \Pi'\}$.

The λe-calculus represents the calculus with terms $\mathscr{T}_e$ and e-reduction.

Definition 17. The new typing relation $\vdash_e$ is obtained by adding four new rules to the typing rules of Definition 12: (start-def), (weak-def), (subst) and (def) below, and by replacing the (conv) and (appl) rules by (new-conv) and (new-appl) as follows:

(start-def) $$\frac{\Gamma \vdash_e A : s \qquad \Gamma \vdash_e B : A}{\Gamma, x = B{:}A \vdash_e x : A} \qquad x \notin \mathrm{DOM}(\Gamma)$$

(weak-def) $$\frac{\Gamma \vdash_e A : B \qquad \Gamma \vdash_e C : s \qquad \Gamma \vdash_e D : C}{\Gamma, x = D{:}C \vdash_e A : B} \qquad x \notin \mathrm{DOM}(\Gamma)$$

(subst) $$\frac{\Gamma, x = B{:}A \vdash_e C : D}{\Gamma \vdash_e C[x \leftarrow B] : D[x := B]}$$

(def) $$\frac{\Gamma, x = B{:}A \vdash_e C : D}{\Gamma \vdash_e (\pi x : A.C)B : D[x := B]} \qquad \text{for } \pi \in \{\lambda, \Pi\}$$

(new-conv) $$\frac{\Gamma \vdash_e A : B \qquad \Gamma \vdash_e B' : s \qquad \Gamma \vdash_e B =_{\mathrm{def}} B'}{\Gamma \vdash_e A : B'}$$

(new-appl) $$\frac{\Gamma \vdash_e F : (\Pi x{:}A.B) \qquad \Gamma \vdash_e a : A}{\Gamma \vdash_e Fa : (\Pi x{:}A.B)a}$$

In (new-conv), $\Gamma \vdash_e B =_{\mathrm{def}} B'$ is defined on $\mathscr{T}_e$ as the smallest equivalence relation closed under:

- If $B =_e B'$, then $\Gamma \vdash_e B =_{\mathrm{def}} B'$
- If $x = D : C \in \Gamma$ and B' arises from B by substituting one particular free occurrence of x in B by D, then $\Gamma \vdash_e B =_{\mathrm{def}} B'$.

In Definition 17, (start-def) and (weak-def) are the start and weakening rules that deal with definitions in the context. The (subst) rule enables to type explicit substitutions whereas the (def) rule types λ- and Π-redexes using definitions in the context. Finally, (new-conv) accommodates the new reductions while (new-appl) takes advantage of the newly available Π-reduction.

At first sight, it may seem that the rule (def) is not necessary because the rule (subst) enables us to unfold definitions. However, (def) is necessary in order to guarantee correctness of types Lemma 36 where it is used in proving the (new-appl) case. If we did not extend the system with Π-reduction then we could do without (def). On the other hand, the (def) rule makes type derivation more efficient, because it permits avoiding the (Π) rule. For example without (def) we have the following type derivation:

0.		$\vdash$	$* : \Box$	axiom
1.	$\alpha : *$	$\vdash$	$\alpha : *$	0, (start)
2.	$\alpha : *, x : \alpha$	$\vdash$	$\alpha : *$	1, 1, (weak)
3.	$\alpha : *$	$\vdash$	$\Pi x : \alpha.\alpha : *$	1, 2, (Π), $(*, *)$
4.	$\alpha : *, x : \alpha$	$\vdash$	$x : \alpha$ 1, (start)	
5.	$\alpha : *$	$\vdash$	$\lambda x : \alpha.x : \Pi x : \alpha.\alpha$	4, 3, (λ)
6.	$\alpha : *, y : \alpha$	$\vdash$	$\lambda x : \alpha.x : \Pi x : \alpha.\alpha$	5, 1, (weak)
7.	$\alpha : *, y : \alpha$	$\vdash$	$y : \alpha$	1, (start)
8.	$\alpha : *, y : \alpha$	$\vdash$	$(\lambda x : \alpha.x)y : (\Pi x : \alpha.\alpha)y$	6, 7, (app)
9.	$\alpha : *, y : \alpha$	$\vdash$	$(\Pi x : \alpha.\alpha)y =_{\mathrm{def}} \alpha$	
10.	$\alpha : *, y : \alpha$	$\vdash$	$(\lambda x : \alpha.x)y : \alpha$	

Using the (def) rule this type derivation can be shortened as follows:

0.		$\vdash$	$* : \Box$	axiom
1.	$\alpha : *$	$\vdash$	$\alpha : *$	0, (start)
2.	$\alpha : *, y : \alpha$	$\vdash$	$\alpha : *$	1, 1, (weak)
3.	$\alpha : *, y : \alpha$	$\vdash$	$y : \alpha$	1, (start)
4.	$\alpha : *, y : \alpha, x = y : \alpha$	$\vdash$	$x : \alpha$	2, 3, (start-def)
5.	$\alpha : *, y : \alpha$	$\vdash$	$(\lambda x : \alpha.x)y : \alpha[x := y] \equiv \alpha$	4, (def)

Another remark concerning the (def) rule is that it may seem unsatisfactory that in this rule, definitions are being unfolded in D but not in C. However, as λs and Πs are not identified, we cannot from typings like $C{:}D$, allow typings like $(\lambda x : A.C)B : (\lambda x : A.D)B$, or like $(\Pi x{:}A.C)B : (\Pi x{:}A.D)B$. The only acceptable typings would be $(\lambda x : A.C)B : (\Pi x{:}A.D)B$. Moreover, this latter acceptable case, can be derivable from our (def) rule for non-topsorts (i.e. for $D \not\equiv \Box$) as is shown by the following lemma:

Lemma 18. *The following rule is a derived rule*:

$$(\textit{derived def rule}) \quad \frac{\Gamma, x = B{:}A \vdash_e C : D \qquad \Gamma, x = B{:}A \vdash_e D : s}{\Gamma \vdash_e (\lambda x : A.C)B : (\Pi x : A.D)B}.$$

Proof. If $\Gamma, x = B{:}A \vdash_e C : D$, then by (def), $\Gamma \vdash_e (\lambda x : A.C)B : D[x := B]$. Also, if $\Gamma, x = B{:}A \vdash_e D : s$, then by (def) $\Gamma \vdash_e (\Pi x : A.D)B : s$. Now by conversion $\Gamma \vdash_e (\lambda x : A.C)B : (\Pi x : A.D)B$ since $\Gamma \vdash_e (\Pi x : A.D)B =_{\text{def}} D[x := B]$. □

If D is a sort, then of course unfolding $x = B{:}A$ in D makes sense since the definition $x = B{:}A$ does not have an effect on D.

Finally, a remark concerning the (subst) rule might be that we did not use explicit substitution in D. Instead we used $D[x := B]$. The reason for this is that using $D[x \leftarrow B]$ will lead to problems with the Correctness of types Lemma 36 (cf. [8]).

The following definition extends Definition 13:

Definition 19 (*Statements, judgements, legal terms and contexts*). Definition 13 is extended to $\mathscr{T}_e$ and $\vdash_e$ by changing everywhere in Definition 13, $\mathscr{T}$ by $\mathscr{T}_e$, $\vdash$ by $\vdash_e$ and by changing item 5 to the following:
5. If d is a declaration, then $\Gamma \vdash_e d$ iff $\Gamma \vdash_e \mathtt{var}(d){:}\mathtt{type}(d)$. Otherwise, if d is a definition then $\Gamma \vdash_e d$ iff $\Gamma \vdash_e \mathtt{var}(d){:}\mathtt{type}(d) \wedge \Gamma \vdash_e \mathtt{ab}(d){:}\mathtt{type}(d) \wedge \Gamma \vdash_e \mathtt{var}(d) =_{\text{def}} \mathtt{ab}(d)$.

5.5. Properties of the $\Pi\sigma$DEF-cube

First, following usual techniques [7,8], one can establish that $\to_\sigma$ is confluent and strongly normalising. Hence σ-normal form (σ-nf) exist and are unique. We shall denote the σ-nf of a term A by $\sigma(A)$. The following lemma characterizes σ-normal forms.

Lemma 20. *The set of σ-normal forms is exactly $\mathscr{T}$ of Definition* 8.

Proof. Check first by induction on A that $A[x \leftarrow B]$ is not in normal form. Then check by induction on A that if A is a σ-nf, then $A \in \mathscr{T}$. Finally observe that every term in $\mathscr{T}$ is in σ-nf. □

Lemma 21. *For all $A,\ B \in \mathscr{T}_e$ we have*:
1. $\sigma(A\,B) \equiv \sigma(A)\sigma(B)$ *and* $\sigma(\pi x : A.B) \equiv \pi x : \sigma(A).\sigma(B)$ *for* $\pi \in \{\Pi, \lambda\}$.
2. $\sigma(A[x \leftarrow B]) \equiv \sigma(A)[x := \sigma(B)]$.
3. $\sigma(A[x := B]) \equiv \sigma(A)[x := \sigma(B)]$.

Proof.
1. is easy because the $\to_\sigma$-rules do not take abstraction or application terms on the left-hand side. They only take substitution terms.
2. is by induction on A using 1. and
3. is by induction on A using 1. and 2. □

The next lemma allows us to use the Interpretation Method in order to get confluence of $\to_e$:

Lemma 22. *Let $A,\ B \in \mathscr{T}_e$, if $A \to_{\beta'\Pi'} B$ then $\sigma(A) \twoheadrightarrow_{\beta\Pi} \sigma(B)$.*

Proof. Induction on A using Lemma 21. We only prove the case $A \equiv (\pi x : E.C)D$ and $B \equiv C[x \leftarrow D]$. Then $\sigma(A) \equiv (\pi x : \sigma(E).\sigma(C))\sigma(D) \to_{\beta\Pi} \sigma(C)[x := \sigma(D)] =^{\text{Lem 21}} \sigma(C[x \leftarrow D])$. □

Now, the following corollaries are immediate.

Corollary 23. *Let $A,\ B \in \mathscr{T}_e$. If $A \twoheadrightarrow_e B$ then $\sigma(A) \twoheadrightarrow_{\beta\Pi} \sigma(B)$.*

Corollary 24 (Soundness). *Let $A,\ B \in \mathscr{T}$. If $A \twoheadrightarrow_e B$ then $\sigma(A) \twoheadrightarrow_{\beta\Pi} \sigma(B)$.*

The λe-calculus is powerful enough to simulate $\beta\Pi$-reduction.

Lemma 25 (Simulation of β-reduction). *Let $A,\ B \in \mathscr{T}$, if $A \to_{\beta\Pi} B$ then $A \twoheadrightarrow_e B$.*

Proof. By induction on A. We only prove the case $A \equiv (\pi x : E.C)D$ and $B \equiv C[x := D]$. Then $A \equiv (\pi x : E.C)D \to_{\beta'\Pi'} C[x \leftarrow D] \twoheadrightarrow_\sigma \sigma(C)[x := \sigma(D)] \equiv C[x := D]$ because $C, D \in \mathscr{T}$. □

Theorem 26 (Confluence of $\to_e$). *If $A \twoheadrightarrow_e B$ and $A \twoheadrightarrow_e C$ then there exists D such that $B \twoheadrightarrow_e D$ and $C \twoheadrightarrow_e D$.*

Proof. We interpret the λe-calculus into the λ-calculus with $\beta\Pi$-reduction via σ-normalisation.

$$\begin{tikzcd}
 & B \arrow[rr, two heads, "\sigma"] & & \sigma(B) \arrow[dr, two heads, "\beta\Pi"] & \\
A \arrow[ur, two heads, "e"] \arrow[rr, two heads, "\sigma"] \arrow[dr, two heads, "e"'] & & \sigma(A) \arrow[ur, two heads, "\beta\Pi"] \arrow[dr, two heads, "\beta\Pi"'] & \text{conf. of } \to_{\beta\Pi} & D \\
 & C \arrow[rr, two heads, "\sigma"] & & \sigma(C) \arrow[ur, two heads, "\beta\Pi"'] &
\end{tikzcd}$$

The existence of the arrows $\sigma(A) \twoheadrightarrow_{\beta\Pi} \sigma(B)$ and $\sigma(A) \twoheadrightarrow_{\beta\Pi} \sigma(C)$ is guaranteed by Corollary 23. We close the diagram by the confluence of λ-calculus extended with Π-reduction as shown in [26]. Finally Lemma 25 ensures $\sigma(B) \twoheadrightarrow_e D$ and $\sigma(C) \twoheadrightarrow_e D$ proving thus CR for the λe-calculus. □

Lemma 27 (Free variable Lemma for $\vdash_e$).
1. *If d and d' are two different elements in a legal context Γ, then $\mathtt{var}(d) \not\equiv \mathtt{var}(d')$.*
2. *If $\Gamma \equiv \Gamma_1, d, \Gamma_2$ and $\Gamma \vdash B : C$, then $\mathrm{FV}(d) \subseteq \mathrm{DOM}(\Gamma_1)$ and $\mathrm{FV}(B), \mathrm{FV}(C) \subseteq \mathrm{DOM}(\Gamma)$.*

Proof. 1. If Γ is legal then for some B, C, $\Gamma \vdash B : C$. Now use induction on the derivation of $\Gamma \vdash_e B : C$. 2. is by induction on the derivation of $\Gamma \vdash_e B : C$. □

Lemma 28 (Substitution Lemma for $\vdash_e$). *Let d be $x = D : C$, Δ_d be $\Delta[x := D]$, A_d be $A[x := D]$ and B_d be $B[x := D]$. The following holds:*
1. *If $\Gamma, d, \Delta \vdash_e A =_{\mathrm{def}} B$, A and B are Γ, d, Δ-legal, then $\Gamma, \Delta_d \vdash_e A_d =_{\mathrm{def}} B_d$.*
2. *If B is a Γ, d-legal term, then $\Gamma, d \vdash_e B =_{\mathrm{def}} B_d$.*
3. *If $\Gamma, d, \Delta \vdash_e A : B$ or $(\Gamma, x : C, \Delta \vdash_e A : B$ and $\Gamma \vdash_e D : C)$, then*

$$\Gamma, \Delta_d \vdash_e A_d : B_d.$$

Proof. 1. Induction on the generation of $=_{\mathrm{def}}$. 2. Induction on the structure of B. 3. Induction on the derivation rules, using 1., 2. and Lemma 27. We only show the case (subst) where $\Gamma, d, \Delta \vdash_e A[y \leftarrow E] : B[y := E]$ comes from $\Gamma, d, \Delta, y = E : F \vdash_e A : B$.
Then by IH, $\Gamma, \Delta_d, y = E[x := D] : F[x := D] \vdash_e A_d : B_d$.
Hence by (subst) $\Gamma, \Delta_d \vdash_e A_d[y \leftarrow E[x := D]] : B_d[y := E[x := D]]$.
But $A_d[y \leftarrow E[x := D]] \equiv A[x := D][y \leftarrow E[x := D]] \equiv A[y \leftarrow E][x := D]$.
And $B_d[y := E[x := D]] \equiv B[x := D][y := E[x := D]] \equiv B[y := E][x := D]$ as $x \not\equiv y$ and $y \notin \mathrm{FV}(D)$. Hence finally, $\Gamma, \Delta_d \vdash_e (A[y \leftarrow E])_d : (B[y := E])_d$. □

Corollary 29. *If $\Gamma, d, \Delta \vdash_e A : B$ where $\mathtt{var}(d) \notin \mathrm{FV}(A) \cup \mathrm{FV}(B) \cup \mathrm{FV}(\Delta)$ then $\Gamma, \Delta \vdash_e A : B$.*

Lemma 30 (Start Lemma for $\vdash_e$). *Let Γ be a $\vdash_e$-legal context. Then*

$$\Gamma \vdash_e * : \Box \text{ and } \forall d \in \Gamma[\Gamma \vdash_e d].$$

Proof. If Γ is legal then for some terms B, C: $\Gamma \vdash_e B : C$; now use induction on the derivation of $\Gamma \vdash_e B : C$ using Corollary 29. □

Lemma 31 (Context Lemma for $\vdash_e$). *Let Γ_1, d, Γ_2 be a $\vdash_e$-legal context. Then $\Gamma_1 \vdash_e \mathtt{type}(d) : s$ for some sort s, $\Gamma_1, d \vdash_e \mathtt{var}(d) : \mathtt{type}(d)$ and if d is a definition then $\Gamma_1 \vdash_e \mathtt{ab}(d) : \mathtt{type}(d)$.*

Proof. If Γ is legal then for some terms B, C: $\Gamma \vdash_e B : C$; now use induction on the derivation of $\Gamma \vdash_e B : C$. □

Lemma 32 (Thinning Lemma for $\vdash_e$). *Let d be either a declaration or a definition and let Γ_1, d, Γ_2 be a legal context.*

1. *If $\Gamma_1, \Gamma_2 \vdash_e A =_{\text{def}} B$, then $\Gamma_1, d, \Gamma_2 \vdash_e A =_{\text{def}} B$.*
2. *If $\Gamma_1, \Gamma_2 \vdash_e A : B$, then $\Gamma_1, d, \Gamma_2 \vdash_e A : B$.*
3. *If d is $x = D : C$ and $\Gamma_1, x : C, \Gamma_2 \vdash_e A : B$, then $\Gamma_1, d, \Gamma_2 \vdash_e A : B$.*

Proof. 1. is by induction on the derivation $\Gamma_1, \Gamma_2 \vdash_e A =_{\text{def}} B$. 2. is by induction on the derivation $\Gamma_1, \Gamma_2 \vdash_e A : B$ using the start Lemma 30 for (axiom), the context Lemma 31 for (start) and (weak) and 1. for the conversion rule. 3. is by induction on the derivation $\Gamma_1, x : C, \Gamma_2 \vdash_e A : B$. We only show the (start) case where $\Gamma_1, x : C \vdash_e x : C$ comes from $\Gamma_1 \vdash_e C : s$. As $\Gamma_1, x = D : C$ is legal, by the context Lemma 31, $\Gamma_1 \vdash_e D : C$ and hence by (weak) $\Gamma_1, x = D : C \vdash_e x : C$. □

Corollary 33. *Let Γ and Δ be legal contexts such that $\Gamma \subseteq' \Delta$.*
1. *If $\Gamma \vdash_e A =_{\text{def}} B$, then $\Delta \vdash_e A =_{\text{def}} B$.*
2. *If $\Gamma \vdash_e A : B$, then $\Delta \vdash_e A : B$.*

Lemma 34 (Swap Lemma for $\vdash_e$). *Assume each of d_1 and d_2 is either a declaration or a definition such that $\texttt{var}(d_1) \notin \text{FV}(\texttt{type}(d_2))$ and if d_2 is a definition, then also $\texttt{var}(d_1) \notin \text{FV}(\texttt{ab}(d_2))$.*
1. *If $\Gamma_1, d_1, d_2, \Gamma_2 \vdash_e A =_{\text{def}} B$, then $\Gamma_1, d_2, d_1, \Gamma_2 \vdash_e A =_{\text{def}} B$.*
2. *If $\Gamma_1, d_1, d_2, \Gamma_2 \vdash_e A : B$, then $\Gamma_1, d_2, d_1, \Gamma_2 \vdash_e A : B$.*

Proof. 1. is by induction on the derivation $\Gamma_1, d_1, d_2, \Gamma_2 \vdash_e A =_{\text{def}} B$. 2. is by induction on the derivation $\Gamma_1, d_1, d_2, \Gamma_2 \vdash_e A : B$. We only show three cases: (start), (weak-def) and (def).

- (start): We only consider the case where d_2 is $x : A$ and $\Gamma_1, d_1, d_2 \vdash_e x : A$ comes from $x \notin \text{DOM}(\Gamma_1, d_1)$ and $\Gamma_1, d_1 \vdash_e A : s$. As $\texttt{var}(d_1) \notin \text{FV}(A)$, then by Substitution Lemma 28, $\Gamma_1 \vdash_e A : s$ and hence by (start), $\Gamma_1, x : A \vdash_e x : A$.
  But as Γ_1, d_1 is legal, then by Context Lemma 31, $\Gamma_1 \vdash_e \texttt{type}(d_1) : s$ for some sort s, $\Gamma_1 \vdash_e \texttt{var}(d_1) : \texttt{type}(d_1)$ and if d_1 is a definition then $\Gamma_1 \vdash_e \texttt{ab}(d_1) : \texttt{type}(d_1)$.
  Hence we can now apply the corresponding (weak) or (weak-def) rule on

  $\Gamma_1, x : A \vdash_e x : A$ to get $\Gamma_1, x : A, d_1 \vdash_e x : A$.
- (weak-def): We only consider the case where d_2 is $x = B : A$ and $\Gamma_1, d_1, d_2 \vdash_e C : D$ comes from $x \notin \text{DOM}(\Gamma_1, d_1)$, $\Gamma_1, d_1 \vdash_e C : D$, $\Gamma_1, d_1 \vdash_e A : s$ and $\Gamma_1, d_1 \vdash_e B : A$. As $\texttt{var}(d_1) \notin \text{FV}(A) \cup \text{FV}(B)$, then by Substitution Lemma 28, $\Gamma_1 \vdash_e A : s$ and $\Gamma_1 \vdash_e B : A$ and hence by (start-def), $\Gamma_1, x = B : A \vdash_e x : A$ as $x \notin \text{DOM}(\Gamma)$.
  As Γ_1, d_1 is legal, then by Context Lemma 31, $\Gamma_1 \vdash_e \texttt{type}(d_1) : s$ for some sort s, $\Gamma_1 \vdash_e \texttt{var}(d_1) : \texttt{type}(d_1)$ and if d_1 is a definition then $\Gamma_1 \vdash_e \texttt{ab}(d_1) : \texttt{type}(d_1)$. Hence we can now apply the Thinning Lemma 32 (as $\Gamma_1, x = B : A$ is legal) and then the corresponding (weak) or (weak-def) rule on $\Gamma_1, x = B : A \vdash_e x : A$ to get $\Gamma_1, x = B : A, d_1 \vdash_e x : A$. Hence $\Gamma_1, x = B : A, d_1$ is legal.
  Now apply Thinning Lemma 32 to $\Gamma_1, d_1 \vdash_e C : D$ to obtain

  $\Gamma_1, x = B : A, d_1 \vdash_e C : D$.
- (def): just apply the induction hypothesis. □

Lemma 35 (Generation Lemma for $\vdash_e$).
1. *If $\Gamma \vdash_e s : C$, then $s \equiv *$ and $\Gamma \vdash_e C =_{\text{def}} \Box$, furthermore if $C \not\equiv \Box$, then $\Gamma \vdash_e C : s'$ for some sort s'.*

2. *If $\Gamma \vdash_e x : A$, then for some $d \in \Gamma$, $x \equiv \mathtt{var}(d)$, $\Gamma \vdash_e A =_{\text{def}} \mathtt{type}(d)$ and $\Gamma \vdash_e A : s$ for some sort s.*
3. *If $\Gamma \vdash_e \lambda x : A.B : C$, then for some D and sort s we have: $\Gamma, x : A \vdash_e B : D$, $\Gamma \vdash_e \Pi x : A.D : s$, $\Gamma \vdash_e \Pi x : A.D =_{\text{def}} C$ and if $\Pi x : A.D \not\equiv C$, then $\Gamma \vdash_e C : s'$ for some sort s'.*
4. *If $\Gamma \vdash_e \Pi_{x:A}.B : C$, then for some sorts s_1, s_2: $\Gamma \vdash_e A : s_1$, $\Gamma, x : A \vdash_e B : s_2$, (s_1, s_2) is a rule, $\Gamma \vdash_e C =_{\text{def}} s_2$ and if $s_2 \not\equiv C$, then $\Gamma \vdash_e C : s$ for some sort s.*
5. *If $\Gamma \vdash_e Fa : C$, $F \not\equiv \pi x : A.B$, then for some D, E: $\Gamma \vdash_e a : D$, $\Gamma \vdash_e F : \Pi x : D.E$, $\Gamma \vdash_e (\Pi x : D.E)a =_{\text{def}} C$ and if $(\Pi x : D.E)a \not\equiv C$, then $\Gamma \vdash_e C : s$ for some s.*
6. *If $\Gamma \vdash_e (\pi x : A.D)B : C$, then $\Gamma, x = B : A \vdash_e D : C$.*
7. *If $\Gamma \vdash_e A[x \leftarrow B] : C$, then for some term D we have $\Gamma, x = B : D \vdash_e A : C$.*

Proof
1. By induction on the derivation of $\Gamma \vdash_e s : C$.
2. By induction on the derivation of $\Gamma \vdash_e x : A$.
3. By induction on the derivation of $\Gamma \vdash_e \lambda x : A.B : C$.
4. By induction on the derivation of $\Gamma \vdash_e \Pi_{x:A}.B : C$.
5. By induction on the derivation of $\Gamma \vdash_e Fa : C$.
6. By induction on the derivation of $\Gamma \vdash_e (\pi x : A.D)B : C$.
7. By induction on the derivation of $\Gamma \vdash_e A[x \leftarrow B] : C$. □

Lemma 36 (Correctness of types for $\vdash_e$). *If $\Gamma \vdash_e A : B$ then ($B \equiv \Box$ or $\Gamma \vdash_e B : s$ for some sort s).*

Proof. By induction on the derivation rules. We only show (subst), (def) and (new-appl).

- (subst): If $\Gamma \vdash_e A[x \leftarrow B] : C[x := B]$, where $\Gamma, x = B : D \vdash_e A : C$, then by IH, $C \equiv \Box$ or $\exists s, \Gamma, x = B : D \vdash_e C : s$. If $C \equiv \Box$, then $C[x := B] \equiv \Box$; else, by Substitution Lemma 28 $\Gamma \vdash_e C[x := B] : s[x := B] \equiv s$.
- (def): If $\Gamma \vdash_e (\pi x : A.D)B : C[x := B]$ where $\Gamma, x = B : A \vdash_e D : C$, then by IH, $C \equiv \Box$ or $\exists s, \Gamma, x = B : A \vdash_e C : s$. If $C \equiv \Box$ then $C[x := B] \equiv \Box$; else, by Substitution Lemma 28 $\Gamma \vdash_e C[x := B] : s[x := B] \equiv s$.
- (new-appl): If $\Gamma \vdash_e Fa : (\Pi x : A.B)a$, where $\Gamma \vdash_e F : \Pi x : A.B$ and $\Gamma \vdash_e a : A$, then by IH, $\exists s, \Gamma \vdash_e \Pi x : A.B : s$. By Generation Lemma 35 $\Gamma, x : A \vdash_e B : s$. By Thinning Lemma 32 $\Gamma, x = a : A \vdash_e B : s$ and by (def) $\Gamma \vdash_e (\Pi x : A.B)a : s[x := a] \equiv s$. □

Lemma 37 (Subject reduction for contexts). *If $\Gamma \vdash_e A =_{\text{def}} B$ and Γ' results from contracting one of the terms in the declarations and definitions of Γ by a one step e-reduction, then $\Gamma' \vdash_e A =_{\text{def}} B$.*

Proof. By induction on the derivation rules $\Gamma \vdash_e A =_{\text{def}} B$. □

Lemma 38 (Subject reduction SR, for $\vdash_e$). *If $\Gamma \vdash_e A : B$ and $A \twoheadrightarrow_e A'$, then $\Gamma \vdash_e A' : B$.*

Proof. We only prove the above lemma for $\rightarrow_e$. We prove by simultaneous induction on the derivation rules:

1. If $\Gamma \vdash_e A : B$ and $\Gamma \rightarrow_e \Gamma'$ (i.e., Γ' results from contracting one of the terms in the declarations and definitions of Γ by a one step e-reduction), then $\Gamma' \vdash_e A : B$.

2. If $\Gamma \vdash_e A : B$ and $A \to_e A'$ then $\Gamma \vdash_e A' : B$.

- (axiom), (start), (weak), (Π), (λ) are easy.
- (start-def) where $\Gamma, x = B : A \vdash_e x : A$ comes from $\Gamma \vdash_e A : s$, $\Gamma \vdash_e B : A$ and $x \notin \mathrm{DOM}(\Gamma)$. We only show the case for $A \to_e A'$. By induction hypothesis (IH) on $\Gamma \vdash_e A : s$, we have $\Gamma \vdash_e A' : s$. Hence by (new-conv) as $\Gamma \vdash_e A =_{\mathrm{def}} A'$ we get from $\Gamma \vdash_e B : A$ that $\Gamma \vdash_e B : A'$. Now use (start-def) on $\Gamma \vdash_e A' : s$ and $\Gamma \vdash_e B : A'$ to derive $\Gamma, x = B : A' \vdash_e x : A'$.
  Finally, as $\Gamma, x = B : A'$ is legal, $\Gamma \vdash_e A : s$ and $\Gamma \vdash_e A =_{\mathrm{def}} A'$, by Thinning Lemma 32, $\Gamma, x = B : A' \vdash_e A : s$ and $\Gamma, x = B : A' \vdash_e A =_{\mathrm{def}} A'$. Hence, by (new-conv) $\Gamma, x = B : A' \vdash_e x : A'$ gives $\Gamma, x = B : A' \vdash_e x : A$.
- (weak-def) Assume $\Gamma, y = D : C \vdash_e A : B$ comes from $\Gamma \vdash_e A : B$, $\Gamma \vdash_e C : s$, $\Gamma \vdash_e D : C$ and $x \notin \mathrm{DOM}(\Gamma)$. If $A \to_e A'$, $D \to_e D'$ or $\Gamma \to_e \Gamma'$ then simply use IH. Else, if $C \to_e C'$ then by IH on $\Gamma \vdash_e C : s$ we have $\Gamma \vdash_e C' : s$ and as $\Gamma \vdash_e C =_{\mathrm{def}} C'$ then by (new-conv) $\Gamma \vdash_e D : C'$. Now use (weak-def) to conclude $\Gamma, y = D : C' \vdash_e A : B$.
- (subst) Take $\Gamma \vdash_e C[x \leftarrow B] : D[x := B]$, where $\Gamma, x = B : A \vdash_e C : D$.
  - If $\Gamma \to_e \Gamma'$ or $C \to_e C'$, then easy use of IH.
  - If $B \to_e B'$ then by IH on $\Gamma, x = B : A \vdash_e C : D$ we get $\Gamma, x = B' : A \vdash_e C : D$. Hence by (subst) $\Gamma \vdash_e C[x \leftarrow B'] : D[x := B']$. By Type Correctness Lemma 36 on $\Gamma, x = B : A \vdash_e C : D$ we get either $D \equiv \Box$ or for some s $\Gamma, x = B : A \vdash_e D : s$. If $D \equiv \Box$, then $D[x := B'] \equiv D[x := B]$ and we are done. If $\Gamma, x = B : A \vdash_e D : s$, then by Substitution Lemma 28 $\Gamma \vdash_e D[x := B] : s$ and hence as $\Gamma \vdash_e D[x := B] =_{\mathrm{def}} D[x := B']$ we get by (new-conv) $\Gamma \vdash_e C[x \leftarrow B'] : D[x := B]$.
  - If $x \notin \mathrm{FV}(C)$, then $C[x \leftarrow B] \to_\sigma C$. Now use Substitution Lemma 28 on $\Gamma, x = B : A \vdash_e C : D$ to get $\Gamma \vdash_e C : D[x := B]$.
  - If $C \equiv x$ then $C[x \leftarrow B] \to_\sigma B$. As $\Gamma, x = B : A$ is legal, then by Context Lemma 31 $\Gamma \vdash_e B : A$. As $\Gamma, x = B : A \vdash_e x : D$ then by Generation Lemma 35, we have $\Gamma, x = B : A \vdash_e D : s$ and $\Gamma, x = B : A \vdash_e D =_{\mathrm{def}} A$. So by Substitution Lemma 28 $\Gamma \vdash_e D[x := B] : s$ and $\Gamma \vdash_e D[x := B] =_{\mathrm{def}} A[x := B]$. But by Free Variable Lemma 27 $x \notin \mathrm{FV}(A)$. Now use (new-conv) to derive $\Gamma \vdash_e B : D[x := B]$.
  - If $C \equiv EF$ where $C[x \leftarrow B] \to_\sigma E[x \leftarrow B]F[x \leftarrow B]$ then we check the case where $E \not\equiv \pi z : M.N$ and leave the case where $E \equiv \pi z : M.N$ to the reader. First note that $D[x := B] \not\equiv \Box$ and hence by Type Correctness Lemma 36 we have $\Gamma \vdash_e D[x := B] : s$ for some s. Now by Generation Lemma 35 we have for some H, G that $\Gamma, x = B : A \vdash_e E : \Pi y : H.G, \Gamma, x = B : A \vdash_e F : H$ and $\Gamma, x = B : A \vdash_e (\Pi y : H.G)F =_{\mathrm{def}} D$. Hence by (subst) $\Gamma \vdash_e E[x \leftarrow B] : (\Pi y : H.G)[x := B]$ and $\Gamma \vdash_e F[x \leftarrow B] : H[x := B]$ and by (new-appl) $\Gamma \vdash_e E[x \leftarrow B]F[x \leftarrow B] : (\Pi y : H[x := B].G[x := B])F[x \leftarrow B]$. Also, by Substitution Lemma 28 $\Gamma \vdash_e ((\Pi y : H.G)F)[x := B] =_{\mathrm{def}} D[x := B]$. But it is easy to show that $\Gamma \vdash_e ((\Pi y : H.G)F)[x := B] =_{\mathrm{def}} (\Pi y : H[x := B].G[x := B])F[x \leftarrow B]$. Hence, apply (new-conv) to get $\Gamma \vdash_e E[x \leftarrow B]F[x \leftarrow B] : D[x := B]$.
  - Case $C \equiv \pi y : E.F$ and $C[x \leftarrow B] \to_\sigma \pi y : E[x \leftarrow B]F[x \leftarrow B]$ is left to the reader.
- (def) Assume $\Gamma \vdash_e (\pi x : A.C)B : D[x := B]$ comes from $\Gamma, x = B : A \vdash_e C : D$.
  - The cases $\Gamma \to_e \Gamma'$ or $C \to_e C'$ or $A \to_e A'$ are easy.
  - If $B \to_e B'$ then by IH $\Gamma, x = B' : A \vdash_e C : D$ hence by (def) $\Gamma \vdash_e (\pi x : A.C)B' : D[x := B']$. But as $\Gamma, x = B : A \vdash_e C : D$ then by Correctness of types Lemma 36 either $D \equiv \Box$ or for some s $\Gamma, x = B : A \vdash_e D : s$. If $D \equiv \Box$ then $D[x := B'] \equiv$

$D[x := B]$ and we are done. If $\Gamma, x = B : A \vdash_e D : s$ then by Substitution Lemma 28 $\Gamma \vdash_e D[x := B] : s$ and hence as $\Gamma \vdash_e D[x := B] =_{\text{def}} D[x := B']$ we get by (new-conv) $\Gamma \vdash_e (\pi x : A.C)B' : D[x := B]$.

  - If $(\pi x : A.C)B \to_e C[x \leftarrow B]$ then use $\Gamma, x = B : A \vdash_e C : D$ to get by (subst) $\Gamma \vdash_e C[x \leftarrow B] : D[x := B]$.

- (new-conv) is an easy application of IH and Lemma 37.
- (new-appl) If $\Gamma \vdash_e Fa : (\Pi x : A.B)a$ where $\Gamma \vdash_e F : \Pi_{x:A}.B$ and $\Gamma \vdash_e a : A$, then the cases where $\Gamma \to_e \Gamma'$ or $F \to_e F'$ or $a \to_e a'$ are an easy use of IH. The interesting case is when $F \equiv (\lambda_{y:E}.F)$, $Fa \to_{\beta'} F[y \leftarrow a]$. Then by Generation Lemma 35 on $\Gamma \vdash_e \lambda_{y:E}.F : \Pi_{x:A}.B$ we have for some D, s: $\Gamma, y : E \vdash_e F : D$, $\Gamma \vdash_e (\Pi_{y:E}.D) : s$, $\Gamma \vdash_e (\Pi_{y:E}.D) =_{\text{def}} \Pi x : A.B$, and if $\Pi_{y:E}.D \not\equiv \Pi x : A.B$ then $\Gamma \vdash_e \Pi x : A.B : s'$ for some s'. As $\Gamma \vdash_e (\Pi_{y:E}.D) =_{\text{def}} \Pi_{x:A}.B$, we get $x \equiv y$, $\Gamma \vdash_e E =_{\text{def}} A$ and $\Gamma \vdash_e D =_{\text{def}} B$. Hence, $\Gamma \vdash_e D[y := a] =_{\text{def}} (\Pi x : A.B)a$.
  We now establish that $\Gamma, y = a : E$ is legal: as $\Gamma \vdash_e E =_{\text{def}} A$, $\Gamma \vdash_e a : A$ and $\Gamma \vdash_e E : s'$ then $\Gamma \vdash_e a : E$, hence by (start) $\Gamma, y = a : E \vdash_e y : E$. Hence by Thinning Lemma 32 as $\Gamma, y : E \vdash_e F : D$, we get $\Gamma, y = a : E \vdash_e F : D$. Now use (subst) to get $\Gamma \vdash_e F[y \leftarrow a] : D[y := a]$.
  We only need now that $\Gamma \vdash_e (\Pi x : A.B)a : s''$ for some sort s'' in order to apply (new-conv) and get the desired result: $\Gamma \vdash_e F[y \leftarrow a] : (\Pi x : A.B)a$. But we have $\Gamma \vdash_e (\Pi_{y:E}.D) : s$ and if $\Pi_{y:E}.D \not\equiv \Pi x : A.B$ then $\Gamma \vdash_e \Pi x : A.B : s'$ for some s'. Hence, in all cases, $\Gamma \vdash_e \Pi x : A.B : s''$ for some sort s'' and so by Generation Lemma 35 $\Gamma, x : A \vdash_e B : s''$. Now establish that $\Gamma, x = a : A$ is legal and use Thinning Lemma 32 to get $\Gamma, x = a : A \vdash_e B : s''$. Finally use (def) to get: $\Gamma \vdash_e (\Pi x : A.B)a : s''$. □

Lemma 39 (Uniqueness of Types for $\vdash_e$).

1. If $\Gamma \vdash_e A : B_1$ and $\Gamma \vdash_e A : B_2$, then $\Gamma \vdash_e B_1 =_{\text{def}} B_2$.

2. If $\Gamma \vdash_e A_1 : B_1$ and $\Gamma \vdash_e A_2 : B_2$ and $A_1 =_e A_2$, then $\Gamma \vdash_e B_1 =_{\text{def}} B_2$.

Proof. 1. is by induction on the structure of A using the Generation Lemma 35. 2. Assume $\Gamma \vdash_e A_1 : B_1$ and $\Gamma \vdash_e A_2 : B_2$ and $A_1 =_e A_2$. By Church Rosser (corollary of Theorem 26) we have for some A, $A_1 \twoheadrightarrow_e A$ and $A_2 \twoheadrightarrow_e A$. Now use Subject Reduction Lemma 38 to get $\Gamma \vdash_e A : B_1$ and $\Gamma \vdash_e A : B_2$ and apply 1. above to get $\Gamma \vdash_e B_1 =_{\text{def}} B_2$. □

In order to prove the strong normalisation of $\vdash_e$-legal terms with respect to e-reduction, we will restrict our attention to two simpler type systems where the (new-appl) rule is returned to the (appl) rule. In both systems, we also remove Π-reduction. In one of those systems, we also remove explicit substitution terms and σ-reduction. These new systems are defined as follows:

Definition 40 (e'-reduction, $\vdash_{e'}$, $\vdash_{e''}$).

- $\mathscr{T}_{e'}$ is the same as $\mathscr{T}_e$. e'-reduction on $\mathscr{T}_{e'}$ is defined as in Definition 16 but where $\to_{\Pi'}$ is removed.
- $\vdash_{e'}$ on $\mathscr{T}_{e'}$ is defined as in Definition 17 but where (new-appl) is replaced by (appl) of Definition 12 and $\Gamma \vdash_{e'} A =_{\text{def}} B$ is defined as $\Gamma \vdash_e A =_{\text{def}} B$ but where $=_e$ is replaced by $=_{e'}$. I.e. the rules of $\vdash_{e'}$ are (axiom), (start), (weak), (Π), (λ), (appl), (start-def), (subst), (weak-def), (def) restricted to λ, and (new-conv).
- $\mathscr{T}_{e''}$ is the same as $\mathscr{T}$. e''-reduction on $\mathscr{T}_{e''}$ is simply β-reduction.

- $\vdash_{e''}$ on $\mathscr{T}_{e''}$ is defined as in Definition 17 but where (subst) is removed, (new-appl) is replaced by (appl) of Definition 12 and $\Gamma \vdash_{e''} A =_{\mathrm{def}} B$ is defined as $\Gamma \vdash_e A =_{\mathrm{def}} B$ but where $=_e$ is replaced by $=_\beta$. I.e. the rules of $\vdash_{e''}$ are (axiom), (start), (weak), (Π), (λ), (appl), (start-def), (weak-def), (def) restricted to λ, and (new-conv).

Similar to the above proofs for $\vdash_e$ and e-reduction, one can establish the desirable properties for $\vdash_{e'}$ with e'-reduction and for $\vdash_{e''}$ with β-reduction. In particular, $\vdash_{e'}$ and $\vdash_{e''}$ satisfy Thinning (see Lemma 32), Subject Reduction (see Lemma 38) and Correctness of Types (see Lemma 36) with respect to e'-reduction resp. β-reduction.

$\vdash_{e''}$ with β-reduction appears in [9] where its strong normalisation has been established.

Theorem 41 (Strong Normalisation with respect to $\vdash_{e''}$ *and* $\rightarrow_\beta$, see [9]). *If A is $\vdash_{e''}$-legal, then $SN_{\rightarrow\!\!\!\rightarrow_\beta}(A)$; i.e. A is strongly normalising with respect to $\rightarrow_\beta$.*

For a context Γ of $\mathscr{T}_e$, we define $\sigma(\Gamma)$ by $\sigma(\emptyset) \equiv \emptyset$, $\sigma(\Gamma, x : A) \equiv \sigma(\Gamma), x : \sigma(A)$ and $\sigma(\Gamma, x = B : A) \equiv \sigma(\Gamma), x = \sigma(B) : \sigma(A)$.

In order to establish strong normalisation for $\vdash_{e'}$ with respect to $\rightarrow_{e'}$ we need the following:

Lemma 42

1. *If $\Gamma \vdash_{e'} A =_{\mathrm{def}} B$, then $\sigma(\Gamma) \vdash_{e''} \sigma(A) =_{\mathrm{def}} \sigma(B)$.*
2. *If $\Gamma \vdash_{e'} A : B$, then $\sigma(\Gamma) \vdash_{e''} \sigma(A) : \sigma(B)$.*

Proof

1. is by induction on the derivation $\Gamma \vdash_{e'} A =_{\mathrm{def}} B$.

2. is by induction on the derivation $\Gamma \vdash_{e'} A : B$ using 1. We only show (subst) where $\Gamma \vdash_{e'} C[x \leftarrow B] : D[x := B]$ comes from $\Gamma, x = B : A \vdash_{e'} C : D$.
Then by IH, $\sigma(\Gamma), x = \sigma(B) : \sigma(A) \vdash_{e''} \sigma(C) : \sigma(D)$.
Hence by (def) $\sigma(\Gamma) \vdash_{e''} (\lambda x : \sigma(A).\sigma(C))\sigma(B) : \sigma(D)[x := \sigma(B)]$. Now by Subject Reduction for $\vdash_{e''}$ and $\rightarrow_\beta$, $\sigma(\Gamma) \vdash_{e''} \sigma(C)[x := \sigma(B)] : \sigma(D)[x := \sigma(B)]$. Finally, by Lemma 21, $\sigma(\Gamma) \vdash_{e''} \sigma(C[x \leftarrow B]) : \sigma(D[x := B])$. □

Theorem 43 (Strong Normalisation with respect to $\vdash_{e'}$ and with respect to $\rightarrow_{e'}$). *If A is $\vdash_{e'}$-legal, then $SN_{\rightarrow_{e'}}(A)$; i.e. A is strongly normalising with respect to $\rightarrow_{e'}$.*

Proof. By correctness of types for $\vdash_{e'}$, we only show that if A is typable, then $\mathrm{SN}_{\rightarrow_{e'}}(A)$. Assume $\Gamma \vdash_{e'} A : B$ then by Lemma 42 $\sigma(\Gamma) \vdash_{e''} \sigma(A) =_{\mathrm{def}} \sigma(B)$. Hence by Theorem 41 $\sigma(A)$ is $\mathrm{SN}_{\rightarrow_\beta}(A)$. Now, one can use the well-established techniques for showing preservation of strong normalisation of calculi of explicit substitutions (as in [5,7,28]) in order to show that $\mathrm{SN}_{\rightarrow_{e'}}(A)$. □

In order to establish the strong normalisation of $\vdash_e$-terms, we change Π-redexes into λ-redexes.

Definition 44

- For all terms A we define $\widetilde{A}$ to be the term A where in all Π-redexes the Π-symbol has been changed into a λ-symbol, creating a λ-redex instead.
- For a context $\Gamma \equiv d_1, \ldots, d_n$ we define $\widetilde{\Gamma}$ to be $\widetilde{d_1}, \ldots, \widetilde{d_n}$, where $\widetilde{\langle x : A \rangle} \equiv \langle x : \widetilde{A} \rangle$ and $\widetilde{x = B : A} \equiv x = \widetilde{B} : \widetilde{A}$.

Lemma 45. *If $\Gamma \vdash_e A : B$ then $\widetilde{\Gamma} \vdash_{e'} \widetilde{A} : \widetilde{B}$.*

Proof. By induction on the rules of $\vdash_e$. All rules except the (new- appl) are trivial since they are also rules in $\vdash_{e'}$. If $\Gamma \vdash_e Fa : (\Pi x : A.B)a$ results from $\Gamma \vdash_e F : \Pi x : A.B$ and $\Gamma \vdash_e a : A$. Then by IH $\widetilde{\Gamma} \vdash_{e'} \widetilde{F} : \Pi x : \widetilde{A}.\widetilde{B}$ and $\widetilde{\Gamma} \vdash_{e'} \widetilde{a} : \widetilde{A}$, so by (appl) of $\vdash_{e'}$, $\widetilde{\Gamma} \vdash_{e'} \widetilde{F}\widetilde{a} : \widetilde{B}[x := \widetilde{a}]$. As $\widetilde{\Gamma} \vdash_{e'} \widetilde{F} : \Pi x : \widetilde{A}.\widetilde{B}$, then by Correctness of types for $\vdash_{e'}$, for some sort s, $\widetilde{\Gamma} \vdash_{e'} \Pi x : \widetilde{A}.\widetilde{B} : s$. Now by Generation for $\vdash_{e'}$, for some sort s', $\widetilde{\Gamma}, x : \widetilde{A} \vdash_{e'} \widetilde{B} : s$ and $\widetilde{\Gamma} \vdash_{e'} \widetilde{A} : s'$. Hence by (weak-def) $\widetilde{\Gamma}, x = \widetilde{a} : \widetilde{A} \vdash_{e'} x : \widetilde{A}$ and hence, $\widetilde{\Gamma}, x = \widetilde{a} : \widetilde{A}$ is legal. Now by Thinning for $\vdash_{e'}$, $\widetilde{\Gamma}, x = \widetilde{a} : \widetilde{A} \vdash_{e'} \widetilde{B} : s$ and by (def) $\widetilde{\Gamma} \vdash_{e'} (\lambda x : \widetilde{A}.\widetilde{B})\widetilde{a} : s$, so by conversion $\widetilde{\Gamma} \vdash_{e'} \widetilde{F}\widetilde{a} : (\lambda x : \widetilde{A}.\widetilde{B})\widetilde{a}$. Finally we need to show that $\widetilde{F}\widetilde{a} \equiv \widetilde{Fa}$, i.e., F is not a Π-term. Assume that F is a Π-term, then by Generation Lemma 35 on $\Gamma \vdash_e F : \Pi x : A.B$, $\Gamma \vdash_e \Pi x : A.B =_{\mathrm{def}} s$ for some sort s. This is a contradiction. □

Theorem 46 (Strong Normalisation with respect to $\vdash_e$ and $\to_e$). *If A is $\vdash_e$-legal, then $SN_{\to_e}(A)$; i.e. A is strongly normalising with respect to $\to_e$.*

Proof. Assume A is $\vdash_e$-legal then by Lemma 45 $\widetilde{A}$ is $\vdash_{e'}$-legal and hence $\mathrm{SN}_{\to_{e'}}(\widetilde{A})$. But, by induction on the derivations one can show that if $\Gamma \vdash_{e'} B : C$, then B is free of Π-redexes (hence by Correctness of types one can also conclude that C too is free of Π-redexes). Hence, by Subject Reduction of $\vdash_{e'}$, no Π-redexes can be created in the course of $\to_{e'}$-reduction of $\widetilde{A}$. Therefore $\mathrm{SN}_{\to_{e'}}(\widetilde{A})$ implies $\mathrm{SN}_{\to_e}(A)$ because otherwise, if there is an infinite e-reduction sequence starting at A then one can create an infinite e'-reduction sequence starting at $\widetilde{A}$. □

6. Refining the Barendregt cube with parameters

6.1. Low- versus high-level approach to functions via parameters: going only half way in the abstraction process

As said before, in the λ-calculus, it is not possible to go only half way in the abstraction process. That is, we cannot express abstractions from 2+3 to $x + 3$. We have to use the 'full λ-abstraction power', since we abstract from 2+3 to $x + 3$ via the β-expansion $(\lambda x.(x + 3))2$. This going half way versus going all the way in the functionalisation process represents a distinction in the functional world between *low-level* versus *high-level* functions.

In the nowadays accepted view on functions, they are 'first class citizens', being entities on their own which act on a par with sets, elements of sets and other basic objects. Historically, however, functions have long been treated as a kind of meta-objects. It is true, function *values* have always been important, but abstract functions as such have not been recognised in their own right until the middle of the previous century. In order to make clear what we are talking about, we distinguish between the following two approaches to the notion of function:

1. In the *low level approach* there are no functions as such, but only function values. Given a set A and an 'arbitrary' element a in A, then $f(a)$ is defined as an element of, say, set B. This is the *operational* view on functions. It is unimportant what the function *is*, as long as we know how it works: for each x of A we must be able to find a value

$f(x)$. That's all there is to say. In this view, the sine-function, for example, is always expressed together with a value: $\sin(\pi)$, $\sin(x)$, etc. This gives formulas like $\sin(2x) = 2\sin(x)\cos(x)$. (Note that it has long been usual to call $f(x)$—and not f—the function as is the case in many mathematics courses.)

2. In the *high level approach*, however, functions are objects in their own right. Given sets A and B, there are '*abstract*' functions f 'from' A 'to' B, which are objects of the function space B^A (also written as $A \to B$). These functions can be indefinite (named by a variable name, like f), or definite (i.e. uniquely defined, like sin).
   In this approach, a function f of type $A \to B$ can be treated just as any other object. It can even be the value of another function. For example, if f is a bijective function from A to B, then inverse(f) is a function from B to A. Hence, inverse is a function of type $(A \to B) \to (B \to A)$, taking functions of type $(A \to B)$ as arguments.

Convention 47. In concurrence with the usual terminology, we speak about *functions with parameters* when referring to functions with variable values in the *low-level* approach. So in this approach, the x in $f(x)$ is a parameter. In the *high-level* approach, such an x is called a (variable) *argument* of the ('abstract') function f.

This shows that an important difference between the low-level and high-level approach is whether functions are 'spectators' in the world under consideration which can be called upon for services but do not join the ongoing play, or 'participants' standing on stage just like the other players. This has important consequences for the theory in which functions participate. In the low-level approach, the corresponding theory can be of lower order than in the high-level case, e.g. first-order with parameters versus second-order without [31]. This makes it possible to *fine-tune* a theory by using parameters for some classes of functions. An advantage can be, as we show in the induction axiom example below and in Section 6, that some desirable properties of the lower order theory (think of decidability, easiness of calculations, typability) can be maintained, without losing the flexibility of the higher-order aspects. In fact, using parameters is a natural thing to do in many logical and mathematical applications and in programming languages and software construction.

Therefore, using the more elaborate view of instantiation and functionalisation, we stand up for a re-evaluation of the low-level approach, which has been lost sight of in the modern, Bourbaki-inspired style of doing mathematics. The low-level approach corresponds in an obvious manner to 'abstraction from a subexpression', as discussed in Section 2. The high-level approach is required for 'function construction' and the full 'functionalisation'. This more elaborate view of instantiation and functionalisation is still worthwhile for many exact disciplines. In fact, both in logic and in computer science it has certainly not been wiped out, and for good reasons.

In order to illustrate how parameters (see Convention 47) are used in the low-level approach, opposing it to the use of abstract functions in the high-level approach, we consider the different attitudes of logicians and mathematicians towards the induction axiom for natural numbers. A logician is someone developing this axiom (or studying its properties), whilst the mathematician is usually only interested in applying (using) the axiom. Hence, the logician describes the axiom as a function which takes one argument: a predicate on natural numbers P:

$$\lambda \mathtt{P}.\mathtt{P}0 \to \forall n[\mathtt{P}n \to \mathtt{P}(\mathtt{S}n)] \to \forall n[\mathtt{P}n]. \tag{1}$$

A mathematician usually is not interested in the axiom presented in the above formulation. Often he is interested in instantiations of the axiom only. Therefore, a mathematician may prefer the induction axiom in the form of an axiom *scheme*, depending on a *parameter* for the predicate $\mathtt{P}$:

$$\mathtt{P}0 \to \forall n[\mathtt{P}n \to \mathtt{P}(\mathtt{S}n)] \to \forall n[\mathtt{P}n]. \tag{2}$$

The scheme itself is not part of the formal language, but all the instantiations of the scheme are. As the scheme itself is not part of the language, this "parametric" presentation of the induction axiom is not as strong as the presentation with the λ-term: The latter is part of the formal language, so it is possible to discuss the axiom within the formal language.

The advantage of not going all the way obtaining the λ-abstraction of (1) but simply using Schema (2) if you are a mathematician, can be obtained by noticing that the λ-abstraction in (1) needs a higher type system ($\lambda P2$ of Section 4), when actually a lower system (λP of Section 4) is sufficient for the mathematician's purposes. This can be seen as follows:

Assuming a variable $\mathbb{N}$ (the type of natural numbers) of type $*$, a variable 0 (representing the natural number zero) of type $\mathbb{N}$ and a variable S (an implementation of the successor function: Snm holds if and only if m is the successor of n) of type $\mathbb{N} \to \mathbb{N} \to *$, the induction axiom can be described in the system of the cube whose Π-formation rules are $(*, *)$, $(*, \Box)$, $(\Box, *)$, by the type (let's call it: $\mathtt{Ind}$) abstracting over the variable p (a proposition ranging over the naturals):

$$\mathtt{Ind} = \Pi p{:}(\mathbb{N}\to *).p0 \to (\Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.pn \to Snm \to pm) \to \Pi n{:}\mathbb{N}.pn. \tag{3}$$

With this type $\mathtt{Ind}$ one can introduce a variable $\mathtt{ind}$ of type $\mathtt{Ind}$ that may serve as a proof term for any application of the induction axiom. This is the logician's approach.

For a mathematician, who only *applies* the induction axiom and does not need to know the proof-theoretical backgrounds, this interpretation is too strong. Translating the mathematician's conduct to this setting, we may express this as follows: The mathematician uses the term $\mathtt{ind}$ only in combination with terms $P : \mathbb{N}\to *$, $Q : P0$ and $R : \Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}. Pn \to Snm \to Pm$ to form a term $\mathtt{ind}PQR$ of type $\Pi n{:}\mathbb{N}.Pn$. In other words: the mathematician is only interested in the *application* of the induction axiom, and treats it as an induction *scheme* in which values P, Q, R have to be substituted to use it. The use of the induction axiom by the mathematician is therefore better described by the following, parametric, scheme (p, q and r are the *parameters* of the scheme):

$$\mathtt{ind}(p{:}\mathbb{N}\to *, q{:}p0, r{:}(\Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.pn \to Snm \to pm)) : \Pi n{:}\mathbb{N}.pn. \tag{4}$$

If now $P : \mathbb{N}\to *$, $Q : P0$ and $R : \Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.Pn \to Snm \to Pm$, then one can form the term $\mathtt{ind}(P, Q, R)$ of type $\Pi n{:}\mathbb{N}.Pn$. The types that occur in this scheme can all be constructed using the Π-formation rules $(*, *)$, $(*, \Box)$, hence the rule $(\Box, *)$ is not needed (in the logician's approach, this rule was needed to form the Π-abstraction $\Pi p{:}(\mathbb{N} \to *) \cdots$).

Consequently, the type system that is used to describe the mathematician's use of the induction axiom can be weaker than the one for the logician. Nevertheless, the parameter mechanism as in (4) gives the mathematician limited (but for his purposes sufficient) access to the induction scheme. Without parameter mechanism, this would not have been possible.

6.2. Other uses of parameters

Many well-known type systems, like AUTOMATH [34], LF [20], and ML [33] can be more or less related to one of the systems of the cube. However, the relations between systems from "practice", and systems of the cube are not always perfect. Here are some examples illustrating this point:

Example 48 (AUTOMATH). It is interesting to note that the first tool for mechanical representation and verification of mathematical proofs, AUTOMATH, has a parameter mechanism and was developed from the viewpoint of mathematicians (see [11]). The representation of a mathematical text in AUTOMATH consists of a finite list of *lines* where every line has the format:

$$x_1 : A_1, \dots, x_n : A_n \vdash g(x_1, \dots, x_n) = t : T$$

Here g is a new name, an abbreviation for the expression t of type T (t may exceptionally be the symbol PN, for primitive notion; such as an axiom) and $x_1, \dots, x_n$ are the parameters of g, with respective types $A_1, \dots, A_n$. (In shorthand, the above AUTOMATH-line is written $(\Gamma; g; t; T)$.) We see that parameters (and definitions such as $g(x_1, \dots, x_n) = t$) are a very substantial part of AUTOMATH since each line introduces a new definition which is inherently parametrised by the variables occurring in the context needed for it.

Actual development of ordinary mathematical theory in the AUTOMATH system by e.g. van Benthem Jutting (cf. [6]) revealed that this combined definition and parameter mechanism is vital for keeping proofs manageable and sufficiently readable for humans.

All the AUTOMATH systems have a restricted typed λ-calculus. But they are more expressive than their λ-calculus suggests at first sight. This is due to a strong parameter mechanism. Even if one removes the typed λ-calculus from AUTOMATH, a quite expressive system "PAL", fully based on parameters, remains (cf. [34]). On the other hand, both AUT-68 and AUT-QE have been related to the cube. But the corresponding cube-systems are too weak to properly describe these AUTOMATH-systems (see below). We will be able to place both AUT-68 and AUT-QE on our refined cube with parameters.

Example 49 (*LF*). The system LF (see [20]) is often described as the system λP of the Barendregt cube. However, Geuvers [18] shows that the use of the Π-formation rule $(*, \Box)$ is very restricted in the practical use of LF. We will see that this use is in fact based on a parametric construct rather than on a Π-formation rule. Here again, we will be able to find a more precise position of LF on the cube which will be the center of the line whose ends are $\lambda{\to}$ and λP.

Example 50 (*ML*). In ML (see [33]), types are written implicitly à la Curry. For example, instead of writing $\lambda x{:}A.B$, one writes $\lambda x.B$ and the type checker in ML looks for the type. It is well-known however from [4] that the implicit and explicit type schemes can be related. For the purposes of our paper, we only consider an explicit version of a subset of ML. Furthermore, we do not treat recursive types nor the Y combinator. In ML, one can define the polymorphic identity by:

$$\mathtt{Id}(\alpha{:}*) = (\lambda x{:}\alpha.x) : (\alpha \to \alpha). \tag{5}$$

But in ML, it is not possible to make an explicit λ-abstraction over $\alpha : *$ by

$$\mathtt{Id} = (\lambda\alpha{:}*.\lambda x{:}\alpha.x) : (\Pi\alpha{:}*.\alpha \to \alpha). \tag{6}$$

Those familiar with ML know that the type $\Pi\alpha{:}\ast.\alpha \to \alpha$ does not belong to the language of ML and hence the λ-abstraction of equation (6) is not possible in ML. Therefore, we can state that ML does not have a Π-formation rule $(\Box, \ast)$. Nevertheless, it clearly has some parameter mechanism (α acting as parameter of `Id`) and hence ML has limited access to the rule $(\Box, \ast)$ enabling Eq. (5) to be defined. This means that ML's type system is none of those of the eight systems of the cube. We will find a place for the type system of ML on our refined cube. That place will be the intersection of the diagonals of the square (of the Barendregt cube) whose corners are $\lambda{\to}$, $\lambda 2$, $\lambda\underline{\omega}$, and $\lambda\omega$ (cf. Fig. 6).

The above examples show that the Barendregt cube of [4] cannot accommodate well-known and practical type systems in a precise manner. We will refine the Barendregt cube by extending it with a parameter mechanism. Such a mechanism allows the construction of terms of the form $c(b_1, \ldots, b_n)$, where c is a constant and $b_1, \ldots, b_n$ are terms. In traditional typed λ-calculus such a term would be written as $cb_1 \ldots b_n$. This last term is constructed step by step. First, c gets typed, then it is applied to b_1, then the result is applied to b_2, and so on. This means that c, cb_1, cb_1b_2, …, $cb_1 \ldots b_n$ are all legal terms of the system. Hence, the attempt to internalise the parameter mechanism into typed λ-calculus as described above, is going too far. In the parametric situation, only $c(b_1, \ldots, b_n)$ is a term. Partial constructions of this term like $c(b_1, \ldots, b_i)$ (for $i < n$) are not a part of the syntax.

Adding parameters *is* an extension, and a useful one, since parametric constructs occur in many practical systems.

Example 51. As explained in Example 48, AUTOMATH has a parametric system.

Example 52. First-order predicate logic has no λ-calculus. It only has parametric constructs. Laan and Frunssen [31] shows that parametric constructs make it possible to give a more accurate description of first-order predicate logic in type theory than that given in the traditional approach of typed λ-calculus.

Example 53. Parameters occur in many parts of computer science. For example, look at the following Pascal fragment P with the function `double`:

```
function double(z:integer):integer;
begin
double := z + z
end;
```

P could be represented by the definition

$$\texttt{double} = (\lambda z{:}\texttt{Int}.(z{+}z)) : (\texttt{Int} \to \texttt{Int}). \qquad (7)$$

Of course, this declaration can imitate the behaviour of the function perfectly well. But the construction has the following disadvantages:

- The declaration has as subterm the type `Int` → `Int`. This subterm does not occur in P itself. More general, Pascal does not have a mechanism to construct types of the form $A \to B$. Hence, the representation contains terms that do not occur in Pascal;
- `double` itself is not a separate expression in Pascal: you canot write x := `double` in a program body. One may only use the expression `double` in a program, if one specifies a parameter p that serves as an argument of `double`.

We conclude that the translation of P as given above is not fully to the point. A parameter mechanism allows us to translate P in the parametric form

$$\texttt{double}(z : \texttt{Int}) = (z + z) : \texttt{Int}. \tag{8}$$

This declaration in (8) does not have the disadvantages of (7) described above.

Example 54. Parameters are also useful in *Inductive Type Systems.* For example, Stefanova [41], uses parameters of three different forms:

1. *Parametric types* of the form $X(x_1 : T_1, \ldots, x_n : T_n)$, where X is a type with $X(x_1 : T_1, \ldots, x_n : T_n) : s$ (s is a sort). Parametric types are useful when extending a PTS with *families* of inductive types.
2. *Parametric elements* of the form $c(a_1 : A_1, \ldots, a_n : A_n)$. In [41] , these are only used for constructors of inductive types. For example: $s(x : \mathit{Nat}) : \mathit{Nat}$ is a parameterized version of the successor function.
3. *Implicit parameterization.* This refers to terms which are parameterized over some of their free variables, to be used in the elimination rules for inductive types.

So for an optimal description of practical systems it may be an advantage to study the "mild" extension with parametric constructs only (cf. [25]). In Section 6.3, we extend the syntax of the cube with parametric constructs, and propose type systems that can type these new constructs. In Section 6.4 we establish the meta-theoretical properties of this extension. In Section 6.5 we show that this extension in fact leads to a refinement of the Barendregt cube: it is split into eight smaller cubes. Section 6.6 places systems like LF, ML, and AUTOMATH in the Refined Barendregt cube.

6.3. The extension with parameters

We extend the eight systems of the Barendregt cube with parametric constructs. Parametric constructs are of the form $c(b_1, \ldots, b_n)$, where $b_1, \ldots, b_n$ are terms of certain prescribed types. Just as we can allow several kinds of Π-constructs (via the set $\boldsymbol{R}$) in the Barendregt cube, we can also allow several kinds of parametric constructs. This is indicated by a set $\boldsymbol{P}$, consisting of tuples (s_1, s_2), where $s_1, s_2 \in \{*, \Box\}$. $(s_1, s_2) \in \boldsymbol{P}$ means that we allow parametric constructs $c(b_1, \ldots, b_n) : A$ where $b_1, \ldots, b_n$ have types $B_1, \ldots, B_n$ of sort s_1, and A is of type s_2. However, if both $(*, s_2) \in \boldsymbol{P}$ and $(\Box, s_2) \in \boldsymbol{P}$, then combinations of parameters are possible. For example, it is allowed that B_1 has type $*$, whilst B_2 has type $\Box$.

Definition 8 is changed to deal with parametric terms as follows:

Definition 55 (Terms of the cube extended with parameters). The set $\mathscr{T}_P$ of *parametric terms* is defined together with the set $\mathscr{L}_T$ of *lists of terms* as follows:

$$\begin{aligned} \mathscr{T}_P &::= * \mid \Box \mid \mathscr{V} \mid \mathscr{C}(\mathscr{L}_T) \mid \mathscr{T}_P\mathscr{T}_P \mid \pi\mathscr{V}{:}\mathscr{T}_P.\mathscr{T}_P; \ \text{(where } \pi \in \{\lambda, \Pi\}\text{)} \\ \mathscr{L}_T &::= \varnothing \mid \mathscr{L}_T, \mathscr{T}_P. \end{aligned}$$

$\mathscr{V}$ is the set of variables and $\mathscr{C}$ (over which $c, c', \ldots$ range) is a set of constants disjoint from $\mathscr{V}$. In a parametric term $c(b_1, \ldots, b_n)$, the subterms $b_1, \ldots, b_n$ are called the *parameters* of the term.

In Notations and Conventions 9, Barendregt Convention's BC is extended in the obvious way to include parametric terms. The notions of $\mathrm{FV}\,(A)$, $\mathrm{BV}\,(A)$, implicit substitution $A[x := B]$ and compatibility are also extended to take into account the new parametric terms of the form $c(b_1, \ldots, b_n)$. In particular, $\mathrm{FV}\,(c(a_1, \ldots, a_n)) = \bigcup_{i=1}^{n} \mathrm{FV}\,(a_i)$ and $c(b_1, \ldots, b_n)[x{:=}A] \equiv c(b_1[x{:=}A], \ldots, b_n[x{:=}A])$.

Similarly, compatibility of β-reduction is extended to parametric terms in the obvious way:

if $b_i \to_\beta b_i'$ then $c(b_1, \ldots, b_i, \ldots, b_n) \to_\beta c(b_1, \ldots, b_i', \ldots, b_n)$ for $1 \leqslant i \leqslant n$.

This is the only way in which β-reduction on $\mathscr{T}_P$ differs from β-reduction on $\mathscr{T}$.

Definition 56 (Constants of terms). Let $A \in \mathscr{T}_P$. Define $\mathrm{CONS}\,(A)$, the set of *constants* of A by

$$\mathrm{CONS}\,(*) = \mathrm{CONS}\,(\Box) = \mathrm{CONS}\,(x) = \emptyset;$$
$$\mathrm{CONS}\,(c(a_1, \ldots, a_n)) = \{c\} \cup \bigcup_{i=1}^{n} \mathrm{CONS}\,(a_i)\,;$$
$$\mathrm{CONS}\,(AB) = \mathrm{CONS}\,(\lambda x{:}A.B) = \mathrm{CONS}\,(\Pi x{:}A.B) = \mathrm{CONS}\,(A) \cup \mathrm{CONS}\,(B)\,;$$

The definition of declarations of a context is now extended to deal with constant declarations:

Definition 57 (declarations, contexts, $\subseteq'$). Let $A, B_1, \ldots, B_n \in \mathscr{T}_P$, $x, x_1, \ldots, x_n \in \mathscr{V}$ and $c \in \mathscr{C}$.

- A *variable declaration* d is of the form $x : A$. We define $\mathtt{var}(d) \equiv x$, $\mathtt{type}(d) \equiv A$, $\mathrm{FV}\,(d) \equiv \mathrm{FV}\,(A)$ and $\mathrm{CONS}\,(d) \equiv \mathrm{CONS}\,(A)$.
- A *constant declaration* d is of the form $c(x_1{:}B_1, \ldots, x_n{:}B_n){:}A$. We define $\mathtt{type}(d) \equiv A$ and $\mathit{dec\text{-}cons}\,(d) \equiv c$. c is called a *primitive constant* (cf. the primitive notions in AUTOMATH). $x_1, \ldots, x_n$ are the *parameters* of d. We define $\mathrm{FV}\,(d)$ to be $\mathrm{FV}\,(A) \cup \mathrm{FV}\,(B_1) \ldots \cup \mathrm{FV}\,(B_n)$ and $\mathrm{CONS}\,(d)$ to be $\mathrm{CONS}\,(A) \cup \mathrm{CONS}\,(B_1) \ldots \cup \mathrm{CONS}\,(B_n)$.
- We let d, d', d_1, etc. range over declarations (both variable and constant declarations).
- We define the set $\mathscr{C}_P$ of *parametric contexts* (which we denote by $\Gamma, \Gamma', \ldots$) and the set $\mathscr{L}_V$ of *lists of variable declarations* as follows:

$$\mathscr{C}_P \ ::= \ \emptyset \mid \mathscr{C}_P, \mathscr{V}{:}\mathscr{T}_P \mid \mathscr{C}_P, \mathscr{C}(\mathscr{L}_V){:}\mathscr{T}_P \qquad \mathscr{L}_V \ ::= \ \emptyset \mid \mathscr{L}_V, \mathscr{V}{:}\mathscr{T}_P.$$

  Notice that $\mathscr{L}_V \subseteq \mathscr{C}_P$ (all lists of variable declarations are contexts, as well).
- We use $\mathrm{DOM}\,(\Gamma)$ to denote the set $\{\mathtt{var}(d) \mid d \text{ is a variable declaration in } \Gamma\}$.
  We define $\mathrm{CONS}\,(\Gamma)$ to denote the set $\{\mathit{dec\text{-}cons}\,(d) \mid d \text{ is a constant declaration in } \Gamma\}$.
- We define substitutions on contexts by: $\emptyset[x := A] \equiv \emptyset$, $(\Gamma, y : B)[x := A] \equiv \Gamma[x := A], y : B[x := A]$, and $(\Gamma, c(x_1 : A_1, \ldots, x_n : A_n) : C)[x := A] \equiv \Gamma[x := A], c(x_1 : A_1[x := A], \ldots, x_n : A_n[x := A]) : C[x := A]$.
- $\subseteq'$ between contexts is defined in a similar way to that of Definition 10.

In the ordinary cube we have that, for a legal term A in a legal context Γ, $\mathrm{FV}\,(A) \subseteq \mathrm{DOM}\,(\Gamma)$. In our cube extended with parameters we have: $\mathrm{FV}\,(A) \subseteq \mathrm{DOM}\,(\Gamma)$ and $\mathrm{CONS}\,(A) \subseteq \mathrm{CONS}\,(\Gamma)$.

Now we extend the typing rules of the cube as follows:

Definition 58 (The Barendregt cube with parametric constants). Let $\boldsymbol{R}$ be as in Definition 12 and let $\boldsymbol{P}$ be a subset of $\{(*,*), (*,\square), (\square,*), (\square,\square)\}$, such that $(*,*) \in \boldsymbol{P}$. The judgements that are derivable in $\lambda\boldsymbol{RP}$ are determined by the rules for $\lambda\boldsymbol{R}$ of Definition 12 and the following two rules where $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$ and $\Delta_i \equiv x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$ (the new relation is called $\vdash^p$):

$$(\vec{\mathbf{C}}\ \mathbf{-weak})\qquad \frac{\Gamma \vdash^p b : B \quad \Gamma, \Delta_i \vdash^p B_i : s_i \quad \Gamma, \Delta \vdash^p A : s}{\Gamma, c(\Delta) : A \vdash^p b : B}\,(s_i, s) \in \boldsymbol{P}, c \notin \mathrm{CONS}\,(\Gamma)$$

$$(\vec{\mathbf{C}}\ \mathbf{-app})\qquad \frac{\begin{array}{c}\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1} (i = 1, \ldots, n)\\ \Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A : s \qquad (\text{if } n = 0)\end{array}}{\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}}$$

At first sight one might miss a $\vec{\mathrm{C}}$-introduction rule. Such a rule, however, is not necessary, as c (on its own) is not a term. c can only be (part of) a term in the form $c(b_1, \ldots, b_n)$, and such terms can be typed by the ($\vec{\mathrm{C}}$-app) rule.

Constant weakening ($\vec{\mathrm{C}}$-weak) explains how we can introduce a declaration of a parametric constant in the context. The context Δ indicates the arity of the parametric constants (the number of declarations in Δ), and of which type each parameter must be ($x_j : B_j$ in Δ means the jth parameter must be of type B_j).

The extra condition $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A : s$ in the ($\vec{\mathrm{C}}$-app) for $n = 0$ is necessary to prevent an empty list of premises. Such an empty list of premises would make it possible to have almost arbitrary contexts in the conclusion. The extra condition is needed to assure that the context in the conclusion is legal.

Definition 59 (Statements, judgements, legal terms and contexts). Definition 13 is extended to $\mathscr{T}_P$ and $\vdash^p$ by changing everywhere in Definition 13, $\mathscr{T}$ by $\mathscr{T}_P$, $\vdash$ by $\vdash^p$ and by changing item 5 to:
5. If d is a variable declaration, then $\Gamma \vdash^p d$ is defined as $\Gamma \vdash^p \mathtt{var}(d) : \mathtt{type}(d)$.
If $d \equiv c(x_1 : B_1, \ldots, x_n : B_n){:}A$ and $n = 0$, then $\Gamma \vdash^p d$ is defined as $\Gamma \vdash^p c : A$.
If $d \equiv c(x_1 : B_1, \ldots, x_n : B_n){:}A$ and $n \neq 0$, then $\Gamma \vdash^p d$ is defined as $\Gamma \vdash^p c(b_1, \ldots, b_n)$: $A[x_j{:=}b_j]_{j=1}^{n}$ whenever $\Gamma \vdash^p b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$ for $1 \leqslant i \leqslant n$.

Now we illustrate the difference between the cube without and with parameters.

Example 60
- In the cube system $\lambda{\to}$ (with one Π-formation rule $(*,*)$) we could introduce a type variable $N : *$ and a variable $o : N$ when we want to work with natural numbers. N represents the type of natural numbers and o represents the natural number zero;
- Though the representation of objects like the type of natural numbers and the natural number zero as a variable works fine in practice, there is a philosophical problem with such a representation. We do not consider the set $\mathbb{N}$ and the number $0 \in \mathbb{N}$ to be variables, because these objects 'do not vary'. If we have a derivation of $N{:}*, o{:}N \vdash^p t : N$

for some term t, it is technically possible to make a λ-abstraction over the variable o and obtain $N{:}{*} \vdash^p \lambda o{:}N.t : N \to N$. This is permitted since o is introduced as a variable, but it is probably not what we had in mind. In systems with parameters, we can distinguish between constants and variables. If o is introduced as a constant, it is not possible to form a λ-abstraction $\lambda o{:}N.t$;

- In some cases, we may need to introduce for each proposition Σ the type $\texttt{proof}(\Sigma)$ of proofs of Σ. This cannot be done in the cube system $\lambda\!\to$ extended with (unparametrised) constants: such a constant `proof` should be of type $\texttt{prop} \to \texttt{type}$ and this type cannot be constructed in $\lambda\!\to$ (notice that $\texttt{type} \equiv *$, so the construction of $\texttt{prop} \to \texttt{type}$ would involve the Π-formation rule $(*, \Box)$).
  However, the term `proof` will hardly ever be used on its own. It is usually used when applied to a proposition Σ. With parameters, it is possible to introduce a parametric version of `proof` by the following context declaration: `proof(p:prop) : type`.
  This does not involve the construction of a type $\texttt{prop} \to \texttt{type}$. Nevertheless it is possible to construct the term $\texttt{prop}(P)$ for any term $P : \texttt{prop}$. We obtain a form of polymorphism without using the polymorphism of λ-calculus.
  A disadvantage may be that we cannot speak about the term `proof` 'as it is'. When using `proof` in the syntax, it must always be applied to a parameter $T : \texttt{prop}$.
  However, an advantage is that we can restrict ourselves to a much more simple type system. In the situation above we remain within the types of the system $\lambda\!\to$. We do not need to use types of the system λP. This may have advantages in implementations of type systems. For instance, the system $\lambda\!\to$ does not involve the conversion rule

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$$

  while λP does involve such a rule. The conversion rule involves β-equality of terms, and though it is decidable whether two λ-terms of λP are β-equal or not, it may take a lot of time and/or memory to establish such a fact. This may cause serious problems when implementing certain type systems. Using parameters whenever possible may therefore simplify implementations.

6.4. Properties of the cube with parameters

We now establish the meta-theoretical properties of the parametric type system of Definition 58.

Lemma 61 (Free variable Lemma for $\vdash^p$).

1. *If d and d' are different variable declarations in a legal context Γ, then* $\texttt{var}(d) \not\equiv \texttt{var}(d')$.
2. *If d and d' are different constant declarations in a legal context Γ, then $dec\text{-}cons\ (d) \not\equiv dec\text{-}cons(d')$.*
3. *If $\Gamma \equiv \Gamma_1, d, \Gamma_2$ and $\Gamma \vdash B : C$, then*
   - $\mathrm{CONS}\,(d) \subseteq \mathrm{CONS}\,(\Gamma_1)$,
   - $\mathrm{FV}\,(d) \subseteq \begin{cases} \mathrm{DOM}\,(\Gamma_1) & \textit{if } d \textit{ is a variable declaration} \\ \mathrm{DOM}\,(\Gamma_1, x_1{:}B_1, \ldots, x_n{:}B_n) & \textit{if } d \equiv c(x_1{:}B_1, \ldots, x_n{:}B_n){:}A \end{cases}$
   - $\mathrm{FV}\,(B)\,, \mathrm{FV}\,(C) \subseteq \mathrm{DOM}\,(\Gamma)$ *and* $\mathrm{CONS}\,(B)\,, \mathrm{CONS}\,(C) \subseteq \mathrm{CONS}\,(\Gamma)$.

Proof. In 1. and 2. as Γ is legal, $\Gamma \vdash_e B : C$ for some B, C. Now, each of 1. and 2. is by induction on the derivation of $\Gamma \vdash_e B : C$. 3. is by induction on the derivation of $\Gamma \vdash_e B : C$. $\square$

Lemma 62 (Substitution Lemma for $\vdash^p$). *If $\Gamma_1, x : C, \Gamma_2 \vdash^p A : B$ and $\Gamma_1 \vdash^p D : C$, then $\Gamma_1, \Gamma_2[x := D] \vdash^p A[x := D] : B[x := D]$.*

Proof. Induction on the derivations $\Gamma, x : C, \Gamma_2 \vdash^p A : B$. We only show the $(\overrightarrow{\text{C}}\text{-weak})$ case. Assume Δ is $x_1 : B_1, \ldots, x_n : B_n$ and $\Gamma_1, x : C, \Gamma_2, c(\Delta) : E \vdash^p A : B$ comes from $(s_i, s) \in \boldsymbol{P}$, $c \notin \text{CONS}\,(\Gamma_1, x : C, \Gamma_2)$, $\Gamma_1, x : C, \Gamma_2 \vdash^p A : B$, $\Gamma_1, x : C, \Gamma_2, \Delta_i \vdash^p B_i : s_i$ and $\Gamma_1, x : C, \Gamma_2, \Delta \vdash^p E : s$. By IH, $\Gamma_1, \Gamma_2[x := D] \vdash^p A[x := D] : B[x := D]$, $\Gamma_1, \Gamma_2[x := D], \Delta_i[x := D] \vdash^p B_i[x := D] : s_i$ and $\Gamma_1, \Gamma_2[x := D], \Delta[x := D] \vdash^p E[x := D] : s$. Hence by $(\overrightarrow{\text{C}}\text{-weak})$ $\Gamma_1, \Gamma_2[x := D], c(\Delta[x := D]) : E[x := D] \vdash^p A[x := D] : B[x := D]$. $\square$

Lemma 63 (Context Lemma for $\vdash^p$). *Let Γ be a $\vdash^p$-legal context. Then $\Gamma \vdash^p * : \square$ and if $\Gamma \equiv \Gamma_1, d, \Gamma_2$, then*
- *If $d \equiv x : A$, then $\Gamma_1, x : A \vdash^p x : A$ and $\Gamma_1 \vdash^p A : s$ for some sort s.*
- *If $d \equiv c(x_1 : B_1, \ldots, x_n : B_n){:}A$, then for some sort s, $\Gamma_1, x_1 : B_1, \ldots, x_n : B_n \vdash^p A : s$ and for some sorts s_i, for $1 \leqslant i \leqslant n$ where $(s_i, s) \in \boldsymbol{P}$, we have $\Gamma_1, x_1 : B_1, \ldots, x_{i-1} : B_{i-1} \vdash^p B_i : s_i$.*

Proof. If Γ is legal, then for some terms B, C: $\Gamma \vdash_e B : C$; now use induction on the derivation of $\Gamma \vdash_e B : C$ to show that $\Gamma \vdash^p * : \square$.
If $\Gamma \equiv \Gamma_1, d, \Gamma_2$ is legal, then for some terms B, C we have $\Gamma_1, d, \Gamma_2 \vdash_e B : C$; now use induction on the derivation of $\Gamma_1, d, \Gamma_2 \vdash_e B : C$. $\square$

Lemma 64 (Thinning Lemma for $\vdash^p$). *Let Γ_1, d, Γ_2 be a legal context. If $\Gamma_1, \Gamma_2 \vdash^p A : B$, then $\Gamma_1, d, \Gamma_2 \vdash^p A : B$.*

Proof. By induction on the derivation $\Gamma_1, \Gamma_2 \vdash_e A : B$. We only show the two interesting cases of $(\overrightarrow{\text{C}}\text{-weak})$ where $\Delta' \equiv x'_1{:}B'_1, \ldots, x'_m{:}B'_m$, $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$, $d \equiv c(\Delta) : A$, $\Gamma, c'(\Delta') : A' \vdash^p b : B$ comes from $\Gamma \vdash^p b : B$, $\Gamma, \Delta'_i \vdash^p B'_i : s'_i$, $\Gamma, \Delta' \vdash^p A' : s'$, $(s'_i, s') \in \boldsymbol{P}$ and $c' \notin \text{CONS}\,(\Gamma)$ and
- either $\Gamma, c'(\Delta') : A', c(\Delta) : A$ is legal then by Context Lemma 63, for some sorts s_i, s where $(s_i, s) \in \boldsymbol{P}$, $\Gamma, c'(\Delta') : A', \Delta \vdash^p A : s$ and $\Gamma, c'(\Delta') : A', \Delta_i \vdash^p B_i : s_i$. Hence, by $(\overrightarrow{\text{C}}\text{-weak})$ $\Gamma, c'(\Delta') : A', c(\Delta) : A \vdash^p b : B$.
- or $\Gamma, c(\Delta) : A, c'(\Delta') : A'$ is legal then by IH (because by Context Lemma 63, $\Gamma, c(\Delta) : A, \Gamma, c(\Delta) : A, \Delta'$ and $\Gamma, c(\Delta) : A, \Delta'_i$ are legal) we have: $\Gamma, c(\Delta) : A \vdash^p b : B$, $\Gamma, c(\Delta) : A, \Delta' \vdash^p A' : s'$ and $\Gamma, c(\Delta) : A, \Delta'_i \vdash^p B'_i : s'_i$. Hence, by $(\overrightarrow{\text{C}}\text{-weak})$ $\Gamma, c(\Delta) : A, c'(\Delta') : A' \vdash^p b : B$. $\square$

Corollary 65. *Let Γ and Δ be legal contexts such that $\Gamma \subseteq' \Delta$. If $\Gamma \vdash_e A : B$, then $\Delta \vdash_e A : B$.*

Lemma 66 (Generation Lemma for $\vdash^p$).
1. *If $\Gamma \vdash^p s : C$, then $s \equiv *$, $C =_\beta \square$ and if $C \not\equiv \square$, then $\Gamma \vdash^p C : s'$ for some sort s'.*
2. *If $\Gamma \vdash^p x : C$, then there is $s \in \boldsymbol{S}$ and $B =_\beta C$ such that $\Gamma \vdash^p B : s$ and $(x{:}B) \in \Gamma$;*

3. *If* $\Gamma \vdash^p (\Pi x{:}A.B) : C$, *then there is* $(s_1, s_2) \in \boldsymbol{R}$ *such that* $\Gamma \vdash^p A : s_1, \Gamma, x{:}A \vdash^p B : s_2$ *and* $C =_\beta s_2$;
4. *If* $\Gamma \vdash^p (\lambda x{:}A.b) : C$, *then there is* $s \in \boldsymbol{S}$ *and* B *such that* $\Gamma \vdash^p (\Pi x{:}A.B) : s$; $\Gamma, x{:}A \vdash^p b : B$; *and* $C =_\beta (\Pi x{:}A.B)$;
5. *If* $\Gamma \vdash^p Fa : C$, *then there are* A, B *such that* $\Gamma \vdash^p F : (\Pi x{:}A.B)$, $\Gamma \vdash^p a : A$ *and* $C =_\beta B[x{:=}a]$.
6. *If* $\Gamma \vdash^p c(b_1, \ldots, b_n){:}D$, *then there exist* $s, \Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$ *and* A *such that* $D =_\beta A[x_j{:=}b_j]_{j=1}^{n}$, *and* $\Gamma \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$. *Moreover*, $\Gamma \equiv \Gamma_1, c(\Delta){:} A, \Gamma_2$ *and* $\Gamma_1, \Delta \vdash^p A : s$. *Finally, there are* $s_i \in S$ *such that* $\Gamma_1, \Delta_i \vdash^p B_i{:}s_i$ *and* $(s_i, s) \in \boldsymbol{P}$.

Proof
1. By induction on the derivation $\Gamma \vdash^p s : C$.
2. By induction on the derivation $\Gamma \vdash^p x : C$.
3. By induction on the derivation $\Gamma \vdash^p (\Pi x{:}A.B) : C$ using Thinning Lemma 64.
4. By induction on the derivation $\Gamma \vdash^p (\lambda x{:}A.b) : C$ using Thinning Lemma 64 and 3. above. We only show the case ($\vec{\text{C}}$-weak) where $\Gamma, c(\Delta) : E \vdash^p (\lambda x{:}A.b) : C$ comes from $\Gamma \vdash^p (\lambda x{:}A.b) : C, \Gamma, \Delta_i \vdash^p B_i : s_i$, $\Gamma, \Delta \vdash^p E : s$ for $(s_i, s) \in \boldsymbol{P}$ and $c \notin$ CONS (Γ). By IH on $\Gamma \vdash^p (\lambda x{:}A.b) : C$, for some s', B, we have $\Gamma \vdash^p (\Pi x{:}A.B) : s'$, $\Gamma, x : A \vdash^p b : B$ and $(\Pi x{:}A.B) =_\beta C$. Hence, by ($\vec{\text{C}}$-weak) we have $\Gamma, c(\Delta) : E \vdash^p (\Pi x{:}A.b) : s'$. Using the above case 3, we get $\Gamma, c(\Delta) : E \vdash^p A : s''$. Hence, Γ, $c(\Delta) : E, x : A \vdash^p x : A$ and so $\Gamma, c(\Delta) : E, x : A$ is legal. Now using Thinning Lemma 64 on $\Gamma, x : A \vdash^p b : B$ we get $\Gamma, c(\Delta) : E, x : A \vdash^p b : B$.
5. By induction on the derivation $\Gamma \vdash^p Fa : C$.
6. By induction on the derivation $\Gamma \vdash^p c(b_1, \ldots, b_n) : D$ using Context Lemma 63 in the ($\vec{\text{C}}$-app) case. We only show the ($\vec{\text{C}}$-app) case. Assume $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}$ comes from $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ $i = 1, \ldots, n$, if $n \neq 0$ and from $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A : s$ if $n = 0$. Then obviously all goals hold except the last two. But, as $\Gamma_1, c(\Delta){:}A, \Gamma_2$ is legal then by Context Lemma 63 there are s, s_i such that $(s_i, s) \in \boldsymbol{P}, \Gamma_1, \Delta \vdash^p A : s$ and $\Gamma_1, \Delta_i \vdash^p B_i : s_i$. □

Lemma 67 (Correctness of types for $\vdash^p$). *If* $\Gamma \vdash^p A : B$, *then* ($B \equiv \Box$ *or* $\Gamma \vdash^p B : s$ *for some sort* s).

Proof. By induction on the derivation $\Gamma \vdash^p A : B$. We only show (appl) and ($\vec{\text{C}}$-app).
- (appl): If $\Gamma \vdash Fa : B[x{:=}a]$ comes from $\Gamma \vdash^p F : (\Pi x{:}A.B)$ and $\Gamma \vdash^p a : A$ then by IH, $\Gamma \vdash^p (\Pi x{:}A.B) : s$ (it cannot be $\Box$). By Generation Lemma 66, $\Gamma, x : A \vdash^p B : s$. As $\Gamma \vdash a : A$, use Substitution Lemma 62 to get $\Gamma \vdash^p B[x := a] : s$.
- ($\vec{\text{C}}$-app): Assume $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}$ comes from Γ_1, $c(\Delta){:}A, \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ $i = 1, \ldots, n$, if $n \neq 0$ and from $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A : s$ if $n = 0$. If $n = 0$, then obviously $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A[x_j{:=}b_j]_{j=1}^{n} : s$. Else, if $n \neq 0$, then: by Generation Lemma 66 we have $\Gamma_1, \Delta \vdash^p A : s$. It is easy to show that all of $\Gamma_1, c(\Delta){:}A, \Gamma_2\Delta$, and $\Gamma_1, c(\Delta){:}A, \Gamma_2\Delta_i$ for $i = 1, \ldots, n$ are legal contexts. Hence, we can use Thinning Lemma 64 on $\Gamma_1, \Delta \vdash^p A : s$ and $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p$

$b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ to get for $\Delta' \in \{\Delta, \Delta_1, \ldots, \Delta_n\}$, any of $\Gamma_1, c(\Delta){:}A, \Gamma_2\Delta' \vdash^p b_i{:}B_i$ $[x_j{:=}b_j]_{j=1}^{i-1}$ and $\Gamma_1, c(\Delta){:}A, \Gamma_2\Delta' \vdash^p A : s$. Now, use Substitution Lemma 62 to get $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A[x_j{:=}b_j]_{j=1}^{n} : s$. $\square$

Lemma 68 (Subterm Lemma for $\vdash^p$). *If A is legal and A' is a subterm of A, then A' is legal.*

Proof. If A is legal then for some Γ, B we have either $\Gamma \vdash^p A : B$ or $\Gamma \vdash^p B : A$. By Correctness of Types Lemma 67, if $\Gamma \vdash^p B : A$ then $A \equiv \square$ or $\Gamma \vdash^p A : s$ for some s. Hence, it is enough to show that if $\Gamma \vdash^p A : B$ and if A' is a subterm of A then for some Γ', B', $\Gamma' \vdash^p A' : B'$. We do this by induction on the structure of A using Generation Lemma 66. $\square$

Lemma 69 (Subject Reduction for $\vdash^p$). *If $\Gamma \vdash^p A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash^p A' : B$.*

Proof. We only prove the above lemma for $\to_\beta$. We prove by simultaneous induction on the derivation rules:

- If $\Gamma \vdash^p A : B$ and $A \to_\beta A'$ then $\Gamma \vdash^p A' : B$.
- If $\Gamma \vdash^p A : B$ and $\Gamma \to_\beta \Gamma'$ then $\Gamma' \vdash^p A : B$ where $\Gamma \to_\beta \Gamma'$ iff $\Gamma = d_1, \ldots, d_i, \ldots d_n$, $\Gamma' = d_1, \ldots, d_i', \ldots d_n$ and
  - $d_i = x : A$, $d_i' = x : A'$ and $A \to_\beta A'$ or
  - $d_i = c(\Delta) : A$ and $((d_i' = c(\Delta') : A$ where $\Delta \to_\beta \Delta')$ or $(d_i' = c(\Delta) : A'$ where $A \to_\beta A'))$.

We will only show ($\vec{\mathrm{C}}$-app). Assume $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}$ comes from:

- $n = 0$ and $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p A : s$. If $\Gamma_1 \to_\beta \Gamma_1'$ or $\Gamma_2 \to_\beta \Gamma_2'$ use IH. If $A \to_\beta A'$ then by IH, $\Gamma_1, c() : A', \Gamma_2 \vdash^p A : s$. Hence by Context Lemma 63 $\Gamma_1 \vdash^p A' : s'$ for some s'. By Thinning Lemma 64 $\Gamma_1, c() : A', \Gamma_2 \vdash^p A' : s'$. Now use ($\vec{\mathrm{C}}$-app) to get $\Gamma_1, c() : A', \Gamma_2 \vdash^p c() : A'$ and then use (conv) to get $\Gamma_1, c() : A', \Gamma_2 \vdash^p c() : A$.
- $n \neq 0$ and $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ for $i = 1, \ldots, n$. If $\Gamma_1 \to_\beta \Gamma_1'$ or $\Gamma_2 \to_\beta \Gamma_2'$ use IH. If $A_1 \to_\beta A'$ then by IH $\Gamma_1, c(\Delta){:}A', \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ for $i = 1, \ldots, n$ and hence by ($\vec{\mathrm{C}}$-app) $\Gamma_1, c(\Delta){:}A', \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A'[x_j{:=}b_j]_{j=1}^{n}$. But $A'[x_j{:=}b_j]_{j=1}^{n} =_\beta A[x_j{:=}b_j]_{j=1}^{n}$. Morevover, by Correctness of types Lemma 67 Γ_1, $c(\Delta){:}A', \Gamma_2 \vdash^p A[x_j{:=}b_j]_{j=1}^{n} : s$. Now use (conv) to get $\Gamma_1, c(\Delta){:}A', \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}$.

  If $B_r \to_\beta B_r'$ for some $1 \leqslant r \leqslant n$ then by IH, $\Gamma_1, c(\Delta'){:}A, \Gamma_2 \vdash^p b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ for $i = 1, \ldots, n$. (Note that $\Delta_i \equiv \Delta_i'$ for $1 \leqslant i \leqslant r$). By Context Lemma 63, $\Gamma_1, \Delta_r' \vdash^p B_r' : s_r$ for some sort s_r. We can easily show that $\Gamma_1, c(\Delta'){:}A, \Gamma_2, \Delta_r'$ is legal and hence by Thinning Lemma 64 $\Gamma_1, c(\Delta'){:}A, \Gamma_2, \Delta_r' \vdash^p B_r' : s_r$. Now by Substitution Lemma 62 $\Gamma_1, c(\Delta'){:}A, \Gamma_2 \vdash^p B_r'[x_j{:=}b_j]_{j=1}^{r-1} : s_r$. But $B_r'[x_j{:=}b_j]_{j=1}^{r-1} = B_r[x_j{:=}b_j]_{j=1}^{r-1}$ and Γ_1, $c(\Delta'){:}A, \Gamma_2 \vdash^p b_r{:}B_r[x_j{:=}b_j]_{j=1}^{r-1}$ hence by (conv) $\Gamma_1, c(\Delta'){:}A, \Gamma_2 \vdash^p b_r{:}B_r'[x_j{:=}b_j]_{j=1}^{r-1}$. Finally, by ($\vec{\mathrm{C}}$-app) $\Gamma_1, c(\Delta'){:}A, \Gamma_2 \vdash^p c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}$. $\square$

Lemma 70 (Uniqueness of Types for $\vdash^p$).
1. *If* $\Gamma \vdash^p A : B_1$ *and* $\Gamma \vdash^p A : B_2$, *then* $B_1 =_\beta B_2$.
2. *If* $A_1 = {}_\beta A_2, \Gamma \vdash^p A_1 : B_1$ *and* $\Gamma \vdash^p A_2 : B_2$, *then* $B_1 =_\beta B_2$.

Proof. 1. is by induction on the structure of A using the Generation Lemma 66. 2. Assume $\Gamma \vdash^p A_1 : B_1$ and $\Gamma \vdash^p A_2 : B_2$ and $A_1 =_\beta A_2$. By Church Rosser we have for some A, $A_1 \twoheadrightarrow_e A$ and $A_2 \twoheadrightarrow_e A$. Now use Subject Reduction Lemma 69 to get $\Gamma \vdash_e A : B_1$ and $\Gamma \vdash_e A : B_2$ and apply 1. above to get $\Gamma \vdash_e B_1 =_{\mathrm{def}} B_2$. $\square$

Theorem 71 (Strong Normalisation with respect to $\vdash^p$ and $\rightarrow_p$). *If A is $\vdash^p$-legal, then $SN_{\rightarrow_\beta}(A)$; i.e. A is strongly normalising with respect to $\rightarrow_\beta$.*

Proof. In [30], a type system $\vdash_{\beta\delta}$ was presented which accommodates definitions both in the contexts and in the terms, and where parameters were also present and intertwined with definitions (unlike the system of this section where we studied the advantages of parameters on their own right). In the system of [30], reduction was not only β-reduction but also included δ-reduction which unfolds the definitions inside the terms step by step. We call the reduction relation of [30] $\beta\delta$-reduction. Obviously, terms in $\mathscr{T}_P$ are also terms of the system of [30]. Moreover, as $\rightarrow_\beta \subset \rightarrow_{\beta\delta}$ then if $SN_{\rightarrow_{\beta\delta}}(A)$ then $SN_{\rightarrow_\beta}(A)$. Furthermore, we can establish by induction on the derivations that if A is $\vdash^p$-legal then A is $\vdash_{\beta\delta}$-legal.

Now assume A is $\vdash^p$-legal, then by above, A is $\vdash_{\beta\delta}$-legal. But by Strong Normalisation for $\vdash_{\beta\delta}$ with respect to $\beta\delta$-reduction (see [30]) we have $SN_{\rightarrow_{\beta\delta}}(A)$. As we discussed above, because $SN_{\rightarrow_{\beta\delta}}(A)$ then $SN_{\rightarrow_\beta}(A)$ and we are done. $\square$

6.5. The refined Barendregt cube

The systems of Definition 58 have six degrees of freedom: three for the possible choices of $(*, \square)$, $(\square, *)$ and $(\square, \square) \in \boldsymbol{R}$ and three for the possible choices of $(*, \square)$, $(\square, *)$, and $(\square, \square) \in \boldsymbol{P}$. However, these choices are not independent since constructs that can be made with $\boldsymbol{P}$-rule (s_1, s_2) can be imitated in a typed λ-calculus with $\boldsymbol{R}$-rule (s_1, s_2). This means that the parameter-free type system with $\boldsymbol{R} = \{(*, *), (*, \square)\}$ is at least as strong as the type system with parameters with the same set $\boldsymbol{R}$, but with $\boldsymbol{P} = \{(*, *), (*, \square)\}$. We make this precise in Theorem 77.

The insight of Theorem 77 can be expressed by depicting the systems with parameters of Definition 58 as a *refinement* of the Barendregt cube. As in the Barendregt cube, we start with the system $\lambda\!\rightarrow$, which has $\boldsymbol{R} = \{(*, *)\}$ and $\boldsymbol{P} = \{(*, *)\}$. Adding an extra element (s_1, s_2) to $\boldsymbol{R}$ still corresponds to moving in one dimension in the cube. Now we add the possibility of moving in one dimension in the cube but stopping half-way. We let this movement correspond to extending $\boldsymbol{P}$ with (s_1, s_2). This "going only half-way" is in line with the intuition that Π-formation with (s_1, s_2) can imitate the construction of a parametric construct with (s_1, s_2). In other words, the system obtained by "going all the way" is at least as strong as the system obtained by "going only half-way". This refinement of the Barendregt cube is depicted in Fig. 5.

We now make the above intuition that "$\boldsymbol{R}$ can imitate $\boldsymbol{P}$" precise.

Definition 72. Consider the system $\lambda\boldsymbol{RP}$. We call this system *parametrically conservative* if $(s_1, s_2) \in \boldsymbol{P}$ implies $(s_1, s_2) \in \boldsymbol{R}$.

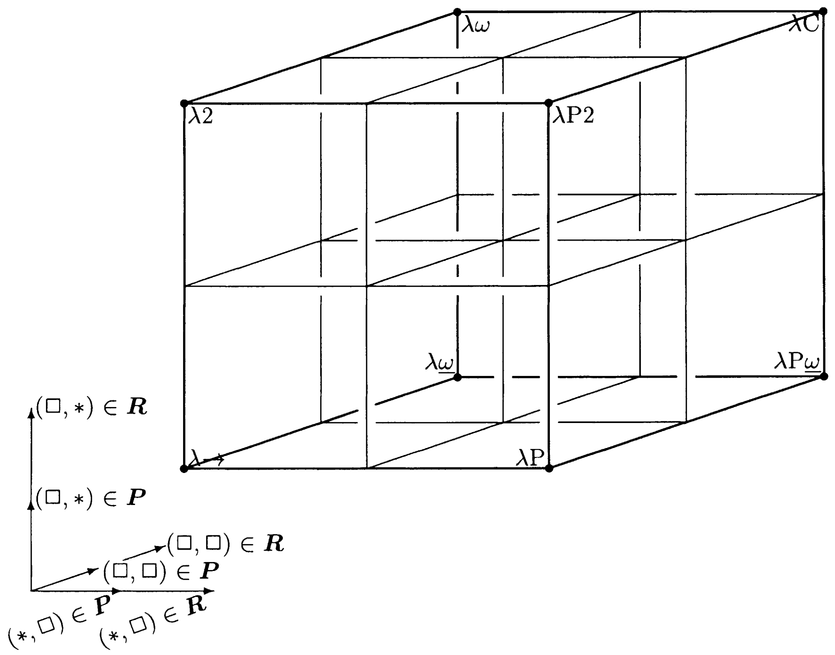

Fig. 5. The refined Barendregt cube.

Let λ***RP*** be parametrically conservative. In order to show that the parameter-free system λ***R*** is at least as powerful as λ***RP***, we need to remove the parameters from the syntax of λ***RP***. To do so, we replace the parametric application in a term $c(b_1, \ldots, b_n)$ by function application $cb_1 \ldots b_n$:

Definition 73. Define the parameter-free translation $\{A\}$ of a term $A \in \mathscr{T}_P$ by:

$$\begin{aligned}\{A\} &\equiv A \quad \text{if } A \equiv x \text{ or } A \equiv s;\\ \{c(b_1, \ldots, b_n)\} &\equiv c\,\{b_1\} \cdots \{b_n\};\\ \{AB\} &\equiv \{A\}\,\{B\};\\ \{\pi x{:}A.B\} &\equiv \pi x{:}\,\{A\}\,.\,\{B\} \quad \text{if } \pi \text{ is } \lambda \text{ or } \Pi.\end{aligned}$$

Definition 74. We extend the definition of {_} to contexts:

$$\begin{aligned}\{\langle\rangle\} &\equiv \langle\rangle;\\ \{\Gamma, x{:}A\} &\equiv \{\Gamma\}\,, x{:}\,\{A\}\,;\\ \{\Gamma, c(\Delta){:}A\} &\equiv \{\Gamma\}\,, c(){:}\,\left\{\prod \Delta.A\right\}.\end{aligned}$$

Here, $\Delta \equiv x_1 : B_1, \ldots, x_n : B_n$, and $\prod \Delta.A$ is shorthand for $\prod_{i=1}^{n} x_n : B_i.A$.

To demonstrate the behaviour of {_} under β-reduction, we need a lemma that shows how to manipulate with substitutions and {_}.

Lemma 75. *For* $A, B \in \mathscr{T}_P$: $\{A[x{:=}B]\} \equiv \{A\}\,[x{:=}\,\{B\}]$.

Proof. The proof is straightforward, using induction on the structure of A. $\square$

The mapping $\{\_\}$ maintains β-reduction:

Lemma 76. *$A \to_\beta A'$ if and only if $\{A\} \to_\beta \left\{A'\right\}$.*

Proof. Follows easily by induction on the structure of A, and Lemma 75. $\square$

Now we show that $\{\_\}$ embeds the parametrically conservative $\lambda\boldsymbol{RP}$ in the parameter-free $\lambda\boldsymbol{R}$:

Theorem 77. *Let $\lambda\boldsymbol{RP}$ be parametrically conservative. If $\Gamma \vdash^p_{\boldsymbol{RP}} A : B$ then $\{\Gamma\} \vdash_{\boldsymbol{R}} \{A\} : \{B\}$.*

Proof. Induction on the derivation of $\Gamma \vdash^p_{\boldsymbol{RP}} A : B$. By Lemma 75, all cases are easy except for ($\overrightarrow{\text{C}}$-weak). So: assume the last step of the derivation was

$$\frac{\Gamma \vdash_{\boldsymbol{RP}} b : B \quad \Gamma, \Delta_i \vdash_{\boldsymbol{RP}} B_i : s_i \quad \Gamma, \Delta \vdash_{\boldsymbol{RP}} A : s}{\Gamma, c(\Delta){:}A \vdash_{\boldsymbol{RP}} b : B} \quad (s_i, s) \in \boldsymbol{P}.$$

By the induction hypothesis, we have:

$$\{\Gamma\} \vdash_{\boldsymbol{R}} \{b\} : \{B\}; \tag{9}$$

$$\{\Gamma, \Delta_i\} \vdash_{\boldsymbol{R}} \{B_i\} : s_i; \tag{10}$$

$$\{\Gamma, \Delta\} \vdash_{\boldsymbol{R}} \{A\} : s. \tag{11}$$

$\lambda\boldsymbol{RP}$ is parametrically conservative, so $(s_i, s) \in \boldsymbol{R}$ for $i = 1, \ldots, n$. Therefore, we can repeatedly use the Π-formation rule, starting with (11) and (10), obtaining

$$\{\Gamma\} \vdash_{\boldsymbol{R}} \prod_{i=1}^{n} x_i{:}\{B_i\}\,.\,\{A\} : s. \tag{12}$$

Notice that $\prod_{i=1}^n x_i{:}\{B_i\}\,.\,\{A\} \equiv \left\{\prod \Delta.A\right\}$. Using ($\overrightarrow{\text{C}}$-weak) on (9) and (12) gives $\{\Gamma\}, c(){:}\left\{\prod \Delta.A\right\} \vdash_{\boldsymbol{R}} \{b\} : \{B\}$. $\square$

Theorem 77 has important consequences. The mapping $\{\_\}$ is fairly simple. It only translates some parametric abstractions and applications into λ-calculus style abstractions and applications. Hence a system of the cube with parametric specification having Π-formation rules $\boldsymbol{R}$ and no parametric formation rules, can be extended with any set of parametric rules without extending its logical power, as long as the parametric specification obtained remains parametrically conservative.

Let us, for example, have a look at the following parametric specifications

$\boldsymbol{R}_1 = \{(*, *), (*, \square)\}, \boldsymbol{P}_1 = \varnothing;$
$\boldsymbol{R}_2 = \{(*, *), (*, \square)\}, \boldsymbol{P}_2 = \{(*, *)\};$
$\boldsymbol{R}_3 = \{(*, *), (*, \square)\}, \boldsymbol{P}_3 = \{(*, \square)\};$
$\boldsymbol{R}_4 = \{(*, *), (*, \square)\}, \boldsymbol{P}_4 = \{(*, *), (*, \square)\}.$

According to Theorem 77, the systems $\lambda\boldsymbol{R}_i\boldsymbol{P}_i$ for $1 \leqslant i \leqslant 4$ with the above specifications are all equal in power.

Now look at the parametric specification $\boldsymbol{R}_5 = \{(*, *)\}, \boldsymbol{P}_5 = \{(*, *), (*, \Box)\}$. The system $\lambda\boldsymbol{R}_5\boldsymbol{P}_5$ is clearly stronger than the system $\lambda{\rightarrow}$, as in $\lambda\boldsymbol{R}_5\boldsymbol{P}_5$ it is possible (in a restricted way) to talk about predicates. For instance, we can have the following context:

$$\begin{array}{rcl} \alpha & : & *, \\ \mathtt{eq}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha) & : & *, \\ \mathtt{refl}(\mathtt{x}{:}\alpha) & : & \mathtt{eq}(\mathtt{x}, \mathtt{x}), \\ \mathtt{symm}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha, \mathtt{p}{:}\mathtt{eq}(\mathtt{x}, \mathtt{y})) & : & \mathtt{eq}(\mathtt{y}, \mathtt{x}), \\ \mathtt{trans}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha, \mathtt{z}{:}\alpha, \mathtt{p}{:}\mathtt{eq}(\mathtt{x}, \mathtt{y}), \mathtt{q}{:}\mathtt{eq}(\mathtt{y}, \mathtt{z})) & : & \mathtt{eq}(\mathtt{x}, \mathtt{z}) \end{array}$$

This context introduces an equality predicate `eq` on objects of type α, and axioms `refl`, `symm`, `trans` for the reflexivity, symmetry and transitivity of `eq`. It is not possible to introduce such a predicate `eq` in the cube system $\lambda{\rightarrow}$ without any parameter mechanism. On the other hand, $\lambda\boldsymbol{R}_5\boldsymbol{P}_5$ is weaker than the cube system λP. In fact, in λP we can construct the type $\Pi\mathtt{x}{:}\alpha.\Pi\mathtt{y}{:}\alpha.*$, which allows us to introduce variables `eq` of type $\Pi\mathtt{x}{:}\alpha.\Pi\mathtt{y}{:}\alpha.*$. This makes it possible to speak about *any* binary predicate, instead of one fixed predicate `eq`. It also gives us the possibility to speak about the term `eq` without the need to apply two terms of type α to it.

This puts the parametric system $\lambda\boldsymbol{R}_5\boldsymbol{P}_5$ clearly between the cube systems $\lambda{\rightarrow}$ and λP.

6.6. Systems in the Refined Barendregt cube

In this section, we show that the Refined Barendregt cube enables us to compare some well-known type systems with systems from the Barendregt cube. In particular, we show that AUT-68, and AUT-QE, LF, and ML, can be seen as systems in the Refined Barendregt cube. This is depicted in Fig. 6, and motivated in the three subsections below.

AUTOMATH: The AUTOMATH-systems (see [34]) all heavily rely on parametric constructs.
1. AUT-68: The typed λ-calculus of one of the most elementary systems of AUTOMATH, AUT-68, is relatively simple and corresponds to $\lambda{\rightarrow}$: it has only $(*, *)$ as a Π-formation rule. This should suggest that AUT-68 has comparable expressiveness to $\lambda{\rightarrow}$. But for the parametrical constructions there are no limitations in AUT-68 whose parameter mechanism has the following features:

- A line $(\Gamma; k; PN; \mathtt{type})$ in a book is nothing more than the declaration of a parametric constant $k(\Gamma){:}*$. There are no demands on the context Γ, and this means that for a declaration $x{:}A \in \Gamma$ we can have either $A \equiv \mathtt{type}$ (in cube-terminology: $A \equiv *$, so $A : \Box$) or $A{:}\mathtt{type}$ (in cube-terminology: $A : *$). We conclude that AUT-68 has the parameter rules $(*, \Box)$ and $(\Box, \Box)$;
- Similarly, lines of the form $(\Gamma; k; PN; \Sigma_2)$ where $\Sigma_2{:}\mathtt{type}$, represent parametric constants that are constructed using the parameter rules $(*, *)$ and $(\Box, *)$.

This suggests that AUT-68 can be represented by the parametric system with $\boldsymbol{R} = \{(*, *)\}$ and $\boldsymbol{P} = \{*, \Box\} \times \{*, \Box\}$. The AUT-68 system can be found in the exact middle of the refined cube.
2. AUT-QE: Something similar holds for the more extensive system AUT-QE. This system has an extra Π-formation rule: $(*, \Box)$ additionally to the rules of AUT-68. This means that for representing this system, we need the Π-formation rules $\boldsymbol{R} = \{(*, *), (*, \Box)\}$, and parametric rules (s_1, s_2) for $s_1, s_2 \in \{*, \Box\}$. This system is located in the middle of the right side of the Refined Barendregt cube, exactly in between λC and λP.

3. PAL: It should be noted that the AUTOMATH languages are all based on two concepts: typed λ-calculus and a parameter/definition mechanism. Both concepts can be isolated: it is possible to study λ-calculus without a parameter/definition mechanism (for instance via the format of Pure Type Systems or the Barendregt cube of [4]), but one can also isolate the parameter/definition mechanism from AUTOMATH. One then obtains a language that is called PAL, the "Primitive AUTOMATH Language". It cannot be described within the Refined Barendregt cube (as all the systems in that cube have at least some basic λ-calculus in it), but it can be described as a system with the following parametric specification: $\boldsymbol{R} = \varnothing$; $\boldsymbol{P} = \{(*, *), (*, \Box), (\Box, *), (\Box, \Box)\}$.

This parametric specification corresponds to the parametric specifications that were given for the AUTOMATH systems above, from which the Π-formation rules are removed.

LF: Geuvers [18] initially describes the system LF (see [20]) as the system λP of the cube. However, the use of the Π-formation rule $(*, \Box)$ is quite restrictive in most applications of LF. Geuvers splits the λ-formation rule in two:

$$(\lambda_0)\frac{\Gamma, x{:}A \vdash M : B \qquad \Gamma \vdash \Pi x{:}A.B : *}{\Gamma \vdash \lambda_0 x{:}A.M : \Pi x{:}A.B};$$

$$(\lambda_P)\frac{\Gamma, x{:}A \vdash M : B \qquad \Gamma \vdash \Pi x{:}A.B : \Box}{\Gamma \vdash \lambda_P x{:}A.M : \Pi x{:}A.B}.$$

System LF without rule (λ_P) is called LF$^-$. β-reduction is split into β_0-reduction and β_P-reduction:

$$(\lambda_0 x{:}A.M)N \to_{\beta_0} M[x{:=}N];$$

$$(\lambda_P x{:}A.M)N \to_{\beta_P} M[x{:=}N].$$

Geuvers then shows that
- If $M : *$ or $M : A : *$ in LF, then the β_P-normal form of M contains no λ_P;
- If $\Gamma \vdash_{\mathrm{LF}} M : A$, and Γ, M, A do not contain a λ_P, then $\Gamma \vdash^-_{\mathrm{LF}} M : A$;
- If $\Gamma \vdash M : A(: *)$, all in β_P-normal form, then $\Gamma \vdash^-_{\mathrm{LF}} M : A(: *)$.

This means that the only real need for a type $\Pi x{:}A.B : \Box$ is to be able to declare a variable in it. The only point at which this is really done is where the bool-style implementation of the Propositions-As-Types principle PAT is made: the construction of the type of the operator Prf (in an unparameterised form) has to be made as follows:

$$\frac{\mathtt{prop}{:}* \vdash \mathtt{prop}{:}\, * \qquad \mathtt{prop}{:}*, \alpha{:}\mathtt{prop} \vdash *{:}\Box}{\mathtt{prop}{:}* \vdash (\Pi\alpha{:}\mathtt{prop}.*) : \Box}.$$

In the practical use of LF, this is the only point where the Π-formation rule $(*, \Box)$ is used. No λ_P-abstractions are used, either, and the term Prf is only used when it is applied to a term $p{:}\mathtt{prop}$. This means that the practical use of LF would not be restricted if we introduced Prf in a parametric form, and replaced the Π-formation rule $(*, \Box)$ by a parameter rule $(*, \Box)$. This puts (the practical applications of) LF in between the systems λ→ and λP in the Refined Barendregt cube.

ML: In ML (cf. [33]) one can define the polymorphic identity by (we use the notation of this paper, whereas in ML, the types and the parameters are left implicit):

$$\mathtt{Id}(\alpha{:}*) = (\lambda \mathtt{x}{:}\alpha.\mathtt{x}) : (\alpha \to \alpha).$$

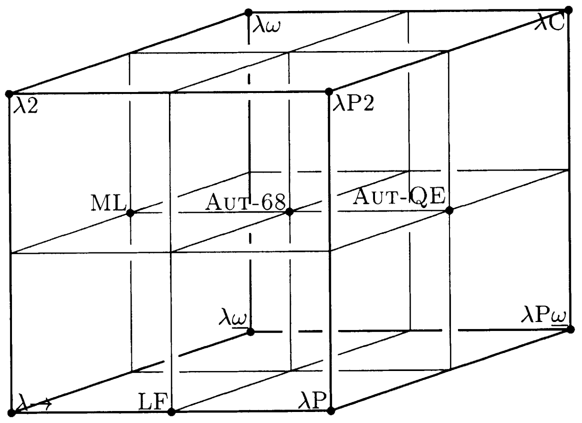

Fig. 6. LF, ML, AUT-68, and AUT-QE in the refined Barendregt cube.

But we cannot make an explicit λ-abstraction over $\alpha{:}*$. That is, the expression

$$\mathtt{Id} = (\lambda\alpha{:}*.\lambda x{:}\alpha.x) : (\Pi\alpha{:}*.\alpha \to \alpha)$$

cannot be constructed in ML, as the type $\Pi\alpha{:}*.\alpha \to \alpha$ does not belong to the language of ML. Therefore, we can state that ML does not have a Π-formation rule $(\Box, *)$, but that it does have the parametric rule $(\Box, *)$.

Similarly, one can introduce the type of lists and some operations by:

$\mathtt{List}(\alpha{:}*)$	:	$*$;
$\mathtt{nil}(\alpha{:}*)$	:	$\mathtt{List}(\alpha)$;
$\mathtt{cons}(\alpha{:}*)$	:	$\alpha \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$,

but the expression $\Pi\alpha{:}*.*$ does not belong to ML, so introducing `List` by

$$\mathtt{List} : \Pi\alpha{:}*.*$$

is not possible in ML. We conclude that ML does not have a Π-formation rule $(\Box, \Box)$, but only the parametric rule $(\Box, \Box)$. Together with the fact that ML has a Π-formation rule $(*, *)$, this places ML in the middle of the left side of the refined Barendregt cube, exactly in between λ→ and λω.

7. Conclusions

In this paper, we presented a detailed description of two important notions in logic, type theory and computation: functionalisation and instantiation. Both processes were split into two sub-processes. Functionalisation consists of abstraction from a subexpression followed by function construction, and instantiation consists of application construction followed by concretisation to a subexpression. Moreover, abstraction from a subexpression and concretisation to a subexpression are each others inverses, and so are function construction and application construction.

We concluded that functionalisation and instantiation can indeed be called "the heart of logic and computation". Our conclusions are strengthened by the fact that functionalisation and instantiation are not only the basic construction principles in the earliest systems (the systems of Frege and Russell), but can also be translated to the notions of β-reduction,

β-expansion and function application, which play a central role in the λ-calculus. However, the part of functionalisation which is the abstraction from a subexpression is missing in the λ-calculus. We argue that this missing part of abstraction from a subexpression was behind various extensions in type theory in order to avoid the disadvantages. We concentrated on the extensions of features that are present heavily in programming languages and theorem provers and that may get confused with the λ-calculus notion of functionalisation (λ-abstraction) but are in fact abstraction from a subexpression which is not present in the λ-calculus. In particular, we discussed the following implications of not having abstraction from a subexpression in the λ-calculus:

- In the λ-calculus, it is not straightforward to identify the original term from which a function has been abstracted. In Section 5.1, we saw that *definitions* and *let expressions* have been introduced in programming languages and theorem provers in order to avoid this disadvantage.
- In the λ-calculus, it is not possible to go only half way in the abstraction process. That is, we cannot express abstractions from $2+3$ to $x+3$. We have to use the 'full λ-abstraction power', since we abstract from $2+3$ to $x+3$ via the β-expansion $(\lambda x.(x+3))2$. In Sections 6.1 and 6.3 we discussed that going half way only, without intermediate β-expansion, results in various advantages and that this going half way is a useful feature, incorporated in programming languages and theorem provers.
- In the λ-calculus, although we could artificially view $x+3$ as an abstraction from a subexpression (say $2+3$), this is not useful because we cannot apply $x+3$ to an argument (say 2 to get $(x+3)2 = 2+3$). Hence, $x+3$ cannot be treated as a function. The explosion of works in explicit substitutions in the λ-calculus in the past two decades could be viewed as an attempt to treat expressions like $x+3$ as functions and apply them to arguments. Hence, in these accounts, the following holds in the λ-calculus extended with explicit substitutions: $(x+3)[x := 2] \rightarrow_\sigma 2+3$ where $(x+3)[x := 2]$ is internal to the λ-calculus with explicit substitutions rather than being an external (meta-) operation as in the usual λ-calculus. In Section 5.2 we argued that it is useful to add the explicit substitutions form of functionalisations and to combine it with another form of functionalisation (the Π-reduction) and with the definitions described in Section 5.1.

Armed with the disadvantages of functionalisation in the ordinary λ-calculus and type theory, we set out to create two extensions of the Barendregt Cube with combinations of the above mentioned missing features of the λ-calculus. The first extension accommodates in one framework, the three notions of explicit substitutions, definitions and Π-reductions. All of these notions are forms of functionalisation and there has been much interest in each of them in the last decade. We argued that these features carry out complementary advantages and hence combining them in one system brings about all of their advantages together. We established that the cube extended with all of these notions satisfies the desirable meta-properties including Subject Reduction and Strong Normalisation.

Our second extension concentrated on the parameter aspect that is found in AUTOMATH. We observed that many existing type systems do not fit exactly in the Barendregt cube. In particular, we explained that previous attempts to describe LF and AUTOMATH were not very successful. We noted that AUTOMATH uses parameters heavily, and that there are some types that are only used in special situations by LF and that those types and situations could be covered by parameters. In addition, we considered an explicitly typed version of ML and noted that there too, ML cannot occupy any of the corners of the cube. The reason being that, ML (as well as LF and AUTOMATH) allows Π-types, but not all of them. In any corner of the cube, as soon as an abstraction of a sort is allowed, all abstractions of that sort are allowed too.

Our above reasoning led us to propose a refinement of the cube where not only the eight corners can be inhabited, but also points half way between these corners. This way, AUTOMATH, LF, and ML find more accurate locations on the cube to represent their typing systems. We described an extension of the Barendregt cube with parameters. This is more a *refinement* than an *extension*, as new systems that are introduced can be depicted by dividing the traditional Barendregt cube into eight sub-cubes. This is due to the fact that parametric constructs can be imitated by constructions of typed λ-calculus (see Theorem 77) but not the other way around.

We showed that our refinement makes it possible to:

- Give a better description of practical type systems like LF and ML than those of the cube.
- Position systems that could not be placed in the usual cube (several AUTOMATH-systems).

This allows a more detailed comparison between the expressiveness of several type systems.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitutions, J. Funct. Programming 1 (4) (1991) 375–416.

[2] S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 2 Background: Computational structures, Oxford University Press, Oxford, 1992.

[3] H.P. Barendregt, The lambda calculus: its syntax and semantics, Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1984.

[4] H.P. Barendregt, Lambda calculi with types, In Ref. [2, pp. 117–309].

[5] Z.E.A. Benaissa, D. Briaud, P. Lescanne, J. Rouyer-Degli, $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation (revised edition), J. Funct. Programming 6 (5) (1996) 699–722.

[6] L.S. van Benthem Jutting, Checking Landau's "Grundlagen" in the Automath system, Ph.D. Thesis, Eindhoven University of Technology, 1977. Published as Mathematical Centre Tracts no. 83, Amsterdam, Mathematisch Centrum, 1979.

[7] R. Bloo, Preservation of termination for explicit substitutions, Ph.D. Thesis, Eindhoven University of Technology, 1997.

[8] R. Bloo, Explicit substitutions in the Barendregt cube, Math. Struct. Comput. Sci. 11 (2001) 3–19.

[9] R. Bloo, F. Kamareddine, R.P. Nederpelt, The Barendregt cube with definitions and generalised reduction, Inform. and Comput. 126 (2) (1996) 123–143.

[10] N.G. de Bruijn, The mathematical language AUTOMATH, its usage and some of its extensions, in: M. Laudet, D. Lacombe, M. Schuetzenberger (Eds.), Proceedings of the Symposium on Automatic Demonstration, IRIA, Versailles, 1968, pp. 29–61, Springer, Berlin, 1970, Lecture Notes in Mathematics 125; also in Ref. [34, pp. 73–100].

[11] N.G. de Bruijn, Reflections on Automath, Eindhoven University of Technology, 1990. Also in Ref. [34, pp. 201–228].

[12] A. Church, A set of postulates for the foundation of logic (1), Ann. Math. 33 (1932) 346–366.

[13] A. Church, A set of postulates for the foundation of logic (2), Ann. Math. 34 (1933) 839–864.

[14] A. Church, A formulation of the simple theory of types, J. Symbolic Logic 5 (1940) 56–68.

[15] T. Coquand, G. Huet, The calculus of constructions, Inform. and Comput. 76 (1988) 95–120.

[16] H.B. Curry, R. Feys, Combinatory Logic I. Studies in Logic and the Foundations of MathematicNorth-Holland, North-Holland, Amsterdam, 1958.

[17] G. Frege, Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens, Nebert, Halle, 1879. Also in Ref. [21, pp. 1–82].

[18] J.H. Geuvers. Logics and type systems, Ph.D. Thesis, Catholic University of Nijmegen, 1993.

[19] J.-Y. Girard, Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, Ph.D. Thesis, Université Paris VII, 1972.

[20] R. Harper, F. Honsell, G. Plotkin, A framework for defining logics, in: Proceedings of the Second Symposium on Logic in Computer Science, IEEE, Washington, DC, 1987, pp. 194–204.

[21] J. van Heijenoort (Ed.), From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931, Harvard University Press, Cambridge, MA, 1967.

[22] J.R. Hindley, J.P. Seldin, Introduction to Combinators and λ-Calculus Vol. 1, London Mathematical Society Student Texts, Cambridge University Press, Cambridge, 1986.

[23] W.A. Howard, The formulas-as-types notion of construction, In Ref. [40, pp. 479–490].

[24] F. Kamareddine, R. Bloo, R.P. Nederpelt, On π-conversion in the λ-cube and the combination with abbreviations, Ann. Pure Appl. Logic 97 (1999) 27–45.

[25] F. Kamareddine, L. Laan, R.P. Nederpelt, Refining the Barendregt cube using parameters, in: Proceedings of the Fifth International Symposium on Functional and Logic Programming, FLOPS 2001, Lecture Notes in Computer Science, 2001, pp. 375–389.

[26] F. Kamareddine, R.P. Nederpelt, Canonical typing and Π-conversion in the barendregt cube, J. Funct. Programming 6 (2) (1996) 245–267.

[27] F. Kamareddine, R.P. Nederpelt, A useful λ-notation, Theoret. Comput. Sci. 155 (1996) 85–109.

[28] F. Kamareddine, A. Ríos, A λ-calculus à la de Bruijn with explicit substitution, Programming Language Implementation and the Logic of Programs PLILP (1995) 45–62.

[29] F. Kamareddine, A. Ríos, Relating the $\lambda\sigma$- and λs-styles of explicit substitutions, J. Logic and Comput. 10 (3) (2000) 399–431.

[30] T. Laan, The evolution of type theory in logic and mathematics, Ph.D. Thesis, Eindhoven University of Technology, 1997.

[31] T. Laan, M. Franssen, Parameters for first order logic, Logic and Comput. (2001).

[32] G. Longo, E. Moggi, Constructive natural deduction and its modest interpretation. Technical Report CMU-CS-88-131, Carnegie Mellono University, Pittsburgh, USA, 1988.

[33] R. Milner, M. Tofte, R. Harper, Definition of Standard ML, MIT Press, Cambridge (MA)/London, 1990.

[34] R.P. Nederpelt, J.H. Geuvers, R.C. de Vrijer (Eds.), Selected Papers on Automath. Studies in Logic and the Foundations of Mathematics, 133, North-Holland, Amsterdam, 1994.

[35] S. Peyton-Jones, The Implementation of Functional Programming Languages, Prentice/Hall International, Englewood Cliff, NJ, 1987.

[36] S. Peyton-Jones, E. Meijer, Henk: a typed intermediate language, Types in Compilation Workshop (1997).

[37] G.R. Renardel de Lavalette, Strictness analysis via abstract interpretation for recursively defined types, Inform. and Comput. 99 (1991) 154–177.

[38] J.C. Reynolds, Towards a Theory of Type Structure, vol. 19, Lecture Notes in Computer Science, Springer, Berlin, 1974, pp. 408–425.

[39] M. Schönfinkel, Über die Bausteine der mathematischen Logik. Mathematische Annalen, 92 (1924) 305–316, Also in Ref. [21, pp. 355–366].

[40] J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic Lambda Calculus and Formalism, Academic Press, New York, 1980.

[41] M.T. Stefanova, Properties of typing systems, Ph.D. Thesis, University of Nijmegen, 1999.

[42] A.N. Whitehead, B. Russell, Principia Mathematica, volume I, II, III. Cambridge University Press, 1910,1912,1913^1, 1925,1925,1927^2. All references are to the first volume, unless otherwise stated.