Parameters in pure type systems

Technische Universiteit Eindhoven
Department of Mathematics and Computing Science

Parameters in Pure Type Systems

by

Twan Laan, Roel Bloo, Fairouz Kamareddine, and Rob Nederpelt

00/18

ISSN 0926-4515

editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 00/18
Eindhoven, November 2000

Parameters in Pure Type Systems

Twan Laan*, Roel Bloo**, Fairouz Kamareddine***, and Rob Nederpelt†

Abstract. In this paper we add parameters to λ-calculus and type theory and show that the resulting systems have nice meta-theoretical properties. We illustrate that parameters allow for a better fine-tuning of the strength of type systems and hence allow for a better description of existing type systems in the framework of pure type systems.

1 What are parameters?

Functions play a fundamental role in logic, mathematics and computer science. 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. So given a set A and an 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 $f(x)$, and 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 and this is still the case in many introductory 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.

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.

The above 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

* email: twan.laan@wxs.nl
** Eindhoven University of Technology, Dept. of computing science, PO-Box 513, 5600 MB Eindhoven, The Netherlands, email: c.j.bloo@tue.nl
*** Heriot-Watt University, Dept. of electrical engineering, Edinburgh, Scotland, email: fairouz@cee.hw.ac.uk
† Eindhoven University of Technology, Dept. of computing science, PO-Box 513, 5600 MB Eindhoven, The Netherlands, email: r.p.nederpelt@tue.nl

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 (cf. Section 1.5). This makes it possible to *fine-tune* a theory by using parameters for some classes of functions. An advantage can be, as we show below, 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. It will also turn out that using parameters is a natural thing to do in many logical and mathematical applications and in programming languages and software construction.

Therefore, in this paper we stand up for a revaluation of the low-level approach, which has been lost sight of in the modern, Bourbaki-inspired style of doing mathematics. We show that this approach 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.

1.1 A different form of abstraction and application

The basis of the λ-calculus is a mechanism for abstraction and application. For abstraction, we use λ-abstraction, and application is implemented via function application. Abstraction and application form the basis of a type system. However, this view of abstraction and application is rigid and does not represent the development of logic in the 20th century. In particular, Frege and Russell's conceptions of functional abstraction, instantiation and application do not fit well with the λ-calculus approach. This is illustrated by $*\mathbf{9}\cdot\mathbf{14}$ and $*\mathbf{9}\cdot\mathbf{15}$ on page 133 of the *Principia Mathematica* (cf. [35]):

> $*\mathbf{9}\cdot\mathbf{14}$. If '$\phi x$' 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 $\phi\hat{x}$, and vice versa.

We see that the function ϕ is not mentioned as a separate entity but always has an argument, be it x, a or $\hat{x}$. Indeed, in the formalisation of propositional functions of the *Principia Mathematica* given by Laan (cf. [22], def. 2.3), we see that e.g. $R(x)$ and $S(x,y)$ are propositional functions but R and S alone are not. However, translation of the propositional function $R(x)$ into ordinary λ-calculus results in $\lambda x.Rx$, since λ-abstraction is the only abstraction mechanism available, and $\lambda x.Rx$ can only be typed if R can be typed on its own.

Allowing parameters in type theory enables one to study Frege and Russell's work from the perspective of modern type theory while staying close to the originally intended interpretation as was done in Laan's thesis [22].

1.2 Developers versus users of a type theory

The parameter mechanism enables us to describe the difference between *developers* and *users* of certain systems. We illustrate this by expressing 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.

Let's give an example in type theory. We adopt the PTS-style, where PTS stands for Pure Type System (cf. [3]). 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 is assumed to hold if and only if m is the successor of n) of type $\mathbb{N}\to\mathbb{N}\to *$, the induction axiom can be described by the following PTS-type (let's call it: Ind), abstracting over the variable p (a proposition ranging over the naturals):

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

in a PTS with sorts $*, \Box$, axiom $* : \Box$ and Π-formation rules $(*,*,*)$, $(*,\Box,\Box)$, $(\Box,*,*)$. With this type Ind one can introduce a variable ind of type 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 doesn't need to know the proof-theoretical backgrounds, this interpretation is too strong. Translating the mathematician's conduct to a PTS-like setting, we may express this as follows: The mathematician uses the term 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: he 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 much better described by the following, parametric, scheme (p, q and r are the *parameters* of the scheme):

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

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 sorts $*, \Box$, axiom $* : \Box$ and rules $(*,*,*)$, $(*,\Box,\Box)$, 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 gives the mathematician limited (but for his purposes sufficient) access to the induction scheme. Without parameter mechanism, this would not have been possible.

We see that the parameter mechanism enables us to describe the difference between a user of a system (in this example: the mathematician) and a developer of the same system (in this example: the logician).

1.3 Automath

In light of the previous section, 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 following format:

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

Here g is a new name, an abbreviation for the expression t of type T and $x_1, \ldots, x_n$ are the parameters of g, with respective types $A_1, \ldots, A_n$.

We see that parameters (and definitions as well) 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. [5]) revealed that this combined definition and parameter mechanism is vital for keeping proofs manageable and sufficiently readable for humans.

1.4 First-order logic in PAT needs parameters

The representation of mathematics in type theory as done in the AUTOMATH project is generally called the propositions-as-types (PAT) style. A mathematical proposition then corresponds to a type, a proof of the proposition corresponds to an inhabitant of the type. Implementations of first-order logic in PAT style usually use a type system that is a variant of the pure type system λP[1]. In λP, it is possible to construct types that are not in β-normal form. Hence, a derivation

[1] λP is due to Berardi [7].

in λP can have non-trivial applications of the conversion rule

$$\frac{\Gamma \vdash A : B_1 \qquad \Gamma \vdash B_2 : s \qquad B_1 =_\beta B_2}{\Gamma \vdash A : B_2}.$$

This can be problematic in implementations. In theory, it is always decidable whether two terms B_1, B_2 are β-equal or not (simply: check whether their β-normal forms are syntactically equal or not). In practice, such a calculation may take quite some time and memory. Therefore, it would be better to use a type system in which the conversion rule is superfluous. This is the case if all types in such a type system are in β-normal form. As all types in simply typed λ-calculus $\lambda{\to}$ (that is: λP without a rule for forming top-level function types) are in β-normal form, it would be a good candidate for an implementation of first-order predicate logic. Unfortunately, first-order predicate logic cannot be described in PAT-style in $\lambda{\to}$, since the introduction of the relation symbols in a first order language involves top-level function types.

But in a first-order language, a relation symbol R always has a fixed arity $\mathbf{ar}(R)$. This means that R itself is not a proposition. It can only be used to *construct* a proposition: if $t_1, \ldots, t_{\mathbf{ar}(R)}$ are terms, then $R(t_1, \ldots, t_{\mathbf{ar}(R)})$ is a proposition. Laan and Franssen show in [23] that with the use of parameters in a type system, it is possible to introduce the relation symbols without need for the top-level function types. This results in a system in which the conversion rule is superfluous, and therefore easier to handle in implementations.

1.5 Programming languages need parameters

Programming languages frequently use parameters. For example, consider the Pascal fragment P defining a function `double`:

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

The argument `(z : integer)` is a parameter in our sense: the function `double` can only be used when given an argument of type `integer`, ergo `double` is a function in the low-level approach. Pascal does not allow polymorphism on the type of the parameter `z`, but in Haskell we can even define

```
double :: Num a => a -> a
double z = z + z.
```

Now `double` may be used without an argument, e.g. in the composition

```
double . double,
```

so here `double` is a function in the high-level approach. But in this definition `double` can be said to have an implicit parameter `a`: when evaluating `double 3.0`, then `a` is instantiated to `Float`; when evaluating `double 3` however, `a` is instantiated to `Integer`.

Alas, parameters as explained above do not exist in the λ-calculus and type theory. In fact, the first `double` above can only be represented in the λ-calculus as (λz:`Int`.(z+z)). The second `double` is represented as: (λa:`Num`.(λz:a.(z+z))). In the first case, the representation in λ-calculus is unfaithful since `double` is a λ-term on its own, of 'high-level character' (it can be used without a parameter). In the second case the representation in λ-calculus of `double` is unfaithful since the parameter `a` is no longer implicit. Note the difference in order between the two representations: the Pascal representation of `double` is first order whereas the Haskell representation is second order.

Another use of parameters can be found in the prototype verification system PVS developed at SRI (cf. [29]). The type system of PVS is more or less similar to simply typed λ-calculus, polymorphic types are not allowed. But in addition, PVS has a mechanism for importing previously

defined theories with parameters. Together with overloading facilities this allows in practice for an almost polymorphic use of types since theories can be imported for each parameter type needed.

Let's discuss the first `double` (the one from Pascal) in some more detail. Note that another problem with representing `double` by (λz:Int.(z+z)) is that the representation in λ-calculus does not give a name to the function (λz:Int.(z+z)). Therefore every time we want to use `double`, we need to use the whole λ-expression that represents it. If we wanted to give a name (say `double`) to the λ-term (λz:Int.(z+z)), then a natural way of doing so is to use type systems with a definition mechanism. For example, in the definition systems of [8, 32], P can be represented by the following context declaration:

In [8]: ((λz:Int.(z+z)) δ)(λ double).[2]

In [32]: double = (λz:Int.(z+z)) : (Int $\rightarrow$ Int).

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

- The declaration of [32] has as subterm the type Int $\rightarrow$ Int. This subterm does not occur in the Pascal fragment P itself. More general, Pascal does not have a mechanism to construct types of the form $A \rightarrow B$. Note that this disadvantage is not faced by the account of [8].
- Due to the way in which `double` is defined in [8, 32], it is (again) a separate subterm in a PTS. But `double` itself is not a separate expression in Pascal; one can't write x := double in a program body. One may use the expression `double` in a program, provided that one specifies a parameter p that serves as an argument of `double`.

We conclude that the translation of P by means of the context declaration above is not fully to the point. Extending type systems with both a definition mechanism and a parameter mechanism allows us to translate P by the parametric context declaration double(z:Int) = (z+z) : Int. This declaration does not have the disadvantages described above:

- It doesn't have the subterm Int $\rightarrow$ Int;
- As we will show in this paper, `double` itself is not a term. We always have to specify an argument p for `double`, thus constructing a term double(p).

Similar remarks can be made about the representation of the Haskell-`double` by the λ-term λa : Num.(λz : a.z + z).

1.6 Extending pure type systems with parameters and definitions

We believe that the previous sections provide ample motivation for extending type theory with parameters and definitions. Our goal is to treat parameters as formal as ordinary abstractions. Therefore we study the so called pure type systems (PTSs) (cf. [3]) and extend these with both a parameter mechanism and the definition mechanism of [32].

One important choice to make is whether to restrict the use of parameters. For a parametrised term $t(p_1, \ldots, p_n)$ we might want to restrict its formation according to the type of t as well as according to the types of the parameters $p_1, \ldots, p_n$. Our choice is to first study the unrestricted use of parameters. This is done in Section 4.

But however elegant an unrestricted use of parameters may seem from a theoretical point of view, this is not custom in actual programming languages.

In many Pascal versions, for instance, parametric *terms* can only have parameters at *term level*, like the integer parameter z of double above. It is, however, not possible in Pascal to write a function `polydouble` that takes both a numeric type Z and an element z of type Z as parameters, and returns the value z+z.

In Section 5 we study PTSs with definitions and restrictions on the use of parameters. Typability of abstractions (or: the 'abstraction rights') in PTSs is governed by a set of triples of sorts.

[2] Note that in [8] we use the item notation (cf. [21]) where the application Fa is written as (a δ)F.

In eight PTSs which together form what is called the Barendregt Cube (cf. [3]), there are two sorts $*$ and $\Box$, and the various systems in the cube are determined by the various ways in which type abstractions can be made. If all constructions of Π-types are allowed, we obtain the Calculus of Constructions, with rules $(*,*,*)$, $(*,\Box,\Box)$, $(\Box,*,*)$ and $(\Box,\Box,\Box)$. If we do not allow all Π-type constructions, we get one of the subsystems of the Calculus of Constructions in the Barendregt Cube.

We propose to govern the 'parametrisation rights' of our PTSs extended with parameters in a similar way, by a set of pairs of sorts. The first elements of these pairs tell which kind of paramaters are allowed, the second elements of these pairs tell which kind of terms are allowed to have parameters.

The combination of the rules for parameter constructions with the well-known rules for the construction of abstractions in the Barendregt Cube leads in a natural way to a division of the Barendregt Cube into eight sub-cubes (we illustrate this in Figure 3 on page 31). As in the Barendregt Cube, one dimension in the cube still corresponds with one of the rules $(*,\Box)$, $(\Box,*)$ or $(\Box,\Box)$. Following an edge of the cube in dimension (s_1,s_2) can now be done in two ways:

- As was already possible, we can follow the edge to the end. This still corresponds to accepting the Π-formation rule (s_1,s_2,s_2);
- We can also follow the edge only half-way. This means that we do not accept the Π-formation rule (s_1,s_2,s_2), but that we do accept the parameter construction rule (s_1,s_2).

This viewpoint suggests that allowing the Π-formation rule (s_1,s_2,s_2) also allows the parameter construction rule (s_1,s_2). Formally, one can work with systems in which we do allow the Π-formation rule, but do not allow the parameter construction rule. We prove, however, that if the Π-construction rule (s_1,s_2,s_2) is allowed, a parameter construction involving rule (s_1,s_2) can be imitated by λ-abstractions (Theorem 100).

This paper is organised as follows:

In Section 2, we give definitions of PTSs extended with parametric constants and definitions. This definition includes an extension of the δ-reduction described in [32] (which unfolds definitions) to parametric definitions.

In Section 3 we show that the δ-reduction and $\beta\delta$-reductions have the Church-Rosser property, and that δ-reduction (under some reasonable conditions) is strongly normalising.

In Section 4, we show some elementary properties of the system introduced in Section 2, like a Generation Lemma, and the Subject Reduction property for $\beta\delta$-reduction. We also prove that $\beta\delta$-reduction is strongly normalising if a slightly stronger PTS is β-strongly normalising.

Section 5 is devoted to the various ways in which parameters can be added to a PTS (with or without definitions) in a more restricted way, with the refined Barendregt Cube of Figure 3 as a result.

In Section 6, we compare our system with some other type systems, like AUTOMATH. We place various AUTOMATH systems in the refined Barendregt Cube of Figure 3.

In Section 7 we see that the use of parameters can sometimes result in simpler and more realistic implementations of type systems.

2 Parametric constants and definitions

In this section, we extend Pure Type Systems (PTSs) (cf. [3]) with parametric constants and definitions.

2.1 Pure Type Systems

Pure Type Systems (PTSs) were introduced by Berardi [7] and Terlouw [33] as a general framework in which many current type systems can be described. The framework is a generalisation of the well-known Barendregt Cube.

Though PTSs were not introduced before 1988, they were already implicitly present in Nederpelt's thesis ([27], Chapter III, Definition 1.3) and many rules in PTSs are highly influenced by rules of known type systems like Church's Simple Theory of Types [12] and AUTOMATH (see 5.5.4. of [14]). The description below is based on [3].

Definition 1 (λ-terms) Let $\mathcal{V}$ be a set of variables and $\mathcal{C}$ a set of constants disjoint with $\mathcal{V}$. Terms of typed λ-calculus are defined by the following abstract syntax:

$$\mathcal{A} ::= \mathcal{V} \mid \mathcal{C} \mid (\mathcal{A}\mathcal{A}) \mid (\lambda\mathcal{V} : \mathcal{A}.\mathcal{A}) \mid (\Pi\mathcal{V} : \mathcal{A}.\mathcal{A}).$$

We omit brackets when possible and consider terms modulo renaming of bound variables (cf. [3]). The notions $FV(A)$ denoting the free variables in λ-term A and substitution $[x{:=}A]$ are defined as usual. Reduction on λ-terms is defined as the contextual closure of

$$(\lambda x : A.B)C \to_\beta B[x{:=}C].$$

Definition 2 (Specification) A *specification* is a triple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$, such that $\boldsymbol{S} \subseteq \mathcal{C}$, $\boldsymbol{A} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ and $\boldsymbol{R} \subseteq \boldsymbol{S} \times \boldsymbol{S} \times \boldsymbol{S}$. The specification is called *singly sorted* if $\boldsymbol{A}$ is a (partial) function $\boldsymbol{S} \to \boldsymbol{S}$, and $\boldsymbol{R}$ is a (partial) function $\boldsymbol{S} \times \boldsymbol{S} \to \boldsymbol{S}$. $\boldsymbol{S}$ is called the set of *sorts*, $\boldsymbol{A}$ is the set of *axioms*, and $\boldsymbol{R}$ is the set of (Π-formation) *rules* of the specification.

Definition 3 (Contexts) A *context* is a finite (possibly empty) list $x_1{:}A_1, \ldots, x_n{:}A_n$ (shorthand: $\vec{x}{:}\vec{A}$) of variable declarations. $\{x_1, \ldots, x_n\}$ is called the *domain* $Dom(\vec{x}{:}\vec{A})$ of the context. The *empty context* is denoted $\langle\rangle$. We use Γ, Δ as meta-variables for contexts.

Definition 4 We define $\Gamma[x{:=}A]$ by induction on the length of Γ:

- $\langle\rangle[x{:=}A] \equiv \langle\rangle$;
- $(\Gamma', y{:}B)[x{:=}A] \equiv \begin{cases} \Gamma'[x{:=}A] & \text{if } x \equiv y; \\ \Gamma'[x{:=}A], y{:}B[x{:=}A] & \text{if } x \not\equiv y. \end{cases}$

Definition 5 (Pure Type Systems) Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The Pure Type System $\lambda\mathcal{S}$ describes in which ways judgements $\Gamma \vdash_{\mathcal{S}} A : B$ (or $\Gamma \vdash A : B$, if it is clear which $\mathcal{S}$ is used) can be derived. $\Gamma \vdash A : B$ states that A has type B in context Γ. $\lambda\mathcal{S}$ consists of the following derivation rules.

$$\text{(axiom)} \quad \langle\rangle \vdash s_1 : s_2 \qquad (s_1, s_2) \in \boldsymbol{A}$$

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

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

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3} \qquad (s_1, s_2, s_3) \in \boldsymbol{R}$$

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

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

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

A context Γ is *legal* if there are A, B such that $\Gamma \vdash A : B$. A term A is *legal* if there are Γ, B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.

An important class of examples of PTSs is formed by the eight PTSs of the so-called Barendregt Cube. The Barendregt Cube (Figure 1 on page 8) is a three-dimensional presentation of eight well-known PTSs. All systems have sorts $\boldsymbol{S} = \{*, \Box\}$, and axioms $\boldsymbol{A} = \{(*, \Box)\}$. Moreover, all the systems have rule $(*, *, *)$. System $\lambda\!\!\rightarrow$ has no extra rules, but the other seven systems all have one or more of the rules $(*, \Box, \Box)$, $(\Box, *, *)$ and $(\Box, \Box, \Box)$:

- Going to the right in the cube means adding rule $(*, \Box, \Box)$;
- Going upwards in the cube means adding rule $(\Box, *, *)$;
- Going backward in the cube means adding rule $(\Box, \Box, \Box)$.

Thus, going to the right, going upwards and going backward means going to a stronger type system. The systems depicted in Figure 1 have the following Π-formation rules:

$\lambda\!\!\rightarrow$	$(*,*,*)$			
$\lambda 2$	$(*,*,*)$	$(\Box,*,*)$		
λP	$(*,*,*)$		$(*,\Box,\Box)$	
$\lambda\underline{\omega}$	$(*,*,*)$			$(\Box,\Box,\Box)$
$\lambda P2$	$(*,*,*)$	$(\Box,*,*)$	$(*,\Box,\Box)$	
$\lambda\omega$	$(*,*,*)$	$(\Box,*,*)$		$(\Box,\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*,*)$		$(*,\Box,\Box)$	$(\Box,\Box,\Box)$
λC	$(*,*,*)$	$(\Box,*,*)$	$(*,\Box,\Box)$	$(\Box,\Box,\Box)$

The dependencies between these systems are depicted in the Barendregt Cube (see Figure 1).

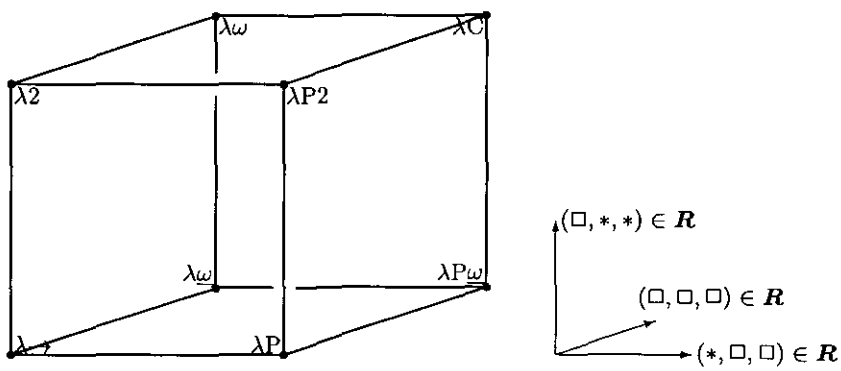


Fig. 1. The Barendregt Cube

The systems in the Cube are related to many other type systems. The overview below is taken from [3].

System	Related system	Names, references
$\lambda\!\!\rightarrow$	λ^{τ}	simply typed λ-calculus; [12], [2] (Appendix A), [20] (Chapter 14)
$\lambda 2$	F	second order typed λ-calculus; [18], [31]
λP	AUT-QE	[10]
	LF	[19]
$\lambda P2$		[24]
$\lambda\underline{\omega}$	POLYREC	[30]
$\lambda\omega$	Fω	[18]
λC	CC	Calculus of Constructions; [13]

Another PTS is the Extended Calculus of Constructions ECC (see [25]). This is a PTS with $\boldsymbol{S} = \mathbb{N}$; $\boldsymbol{A} = \{(n, n+1) \mid n \in \mathbb{N}\}$; and $\boldsymbol{R} = \{(m, 0, 0) \mid m \in \mathbb{N}\} \cup \{(m, n, r) \mid 0 \leq m, n \leq r\}$. This is indeed an extension of λC (write $*$ for 0 and $\Box$ for 1).

Pure Type Systems have some important meta-properties, which we describe below. The proofs can be found in [17] and [16]. Throughout this section, $\vdash$ denotes derivability in a PTS with a certain specification $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

Lemma 6 (Free Variable Lemma) *Let $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ be legal, say $\Gamma \vdash B : C$. Then*

1. *The x_i are distinct;*
2. *$FV(B), FV(C) \subseteq Dom(\Gamma)$;*
3. *$FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $1 \leq i \leq n$.*

Lemma 7 (Start Lemma) *Let Γ be legal. Then*

1. *$\Gamma \vdash s_1 : s_2$ for all $(s_1, s_2) \in \boldsymbol{A}$;*
2. *$\Gamma \vdash x : A$ for all $(x{:}A) \in \Gamma$.*

Lemma 8 (Transitivity Lemma) *Let Γ, Δ be contexts. Assume Γ is legal, $\Gamma \vdash x{:}A$ for all $(x{:}A) \in \Delta$, and $\Delta \vdash B : C$. Then $\Gamma \vdash B : C$.*

Lemma 9 (Thinning Lemma) *Let $\Gamma_1, x{:}A, \Gamma_2$ be a legal context such that $\Gamma_1, \Gamma_2 \vdash B : C$. Then also $\Gamma_1, x{:}A, \Gamma_2 \vdash B : C$.*

Lemma 10 (Substitution Lemma) *If $\Gamma, x{:}A, \Delta \vdash B : C$ and $\Gamma \vdash D : A$ then $\Gamma, \Delta[x{:=}D] \vdash B[x{:=}D] : C[x{:=}D]$.*

Lemma 11 (Generation Lemma)

1. *If $\Gamma \vdash c : C$ for a $c \in \mathcal{C}$ then there is $s \in \boldsymbol{S}$ such that $C =_\beta s$ and $(c{:}s) \in \boldsymbol{A}$;*
2. *If $\Gamma \vdash x : C$ for an $x \in \mathcal{V}$ then there is $s \in \boldsymbol{S}$ and $B =_\beta C$ such that $\Gamma \vdash B : s$ and $(x{:}B) \in \Gamma$;*
3. *If $\Gamma \vdash (\Pi x{:}A.B) : C$ then there is $(s_1, s_2, s_3) \in \boldsymbol{R}$ such that $\Gamma \vdash A : s_1$, $\Gamma, x{:}A \vdash B : s_2$ and $C =_\beta s_3$;*
4. *If $\Gamma \vdash (\lambda x{:}A.b) : C$ then there is $s \in \boldsymbol{S}$ and B such that $\Gamma \vdash (\Pi x{:}A.B) : s$; $\Gamma, x{:}A \vdash b : B$; and $C =_\beta (\Pi x{:}A.B)$;*
5. *If $\Gamma \vdash Fa : C$ then there are A, B such that $\Gamma \vdash F : (\Pi x{:}A.B)$, $\Gamma \vdash a : A$ and $C =_\beta B[x{:=}a]$.*

Lemma 12 (Correctness of Types) *If $\Gamma \vdash A : B$ then $B \equiv s$ or $\Gamma \vdash B : s$ for some $s \in \boldsymbol{S}$.*

Lemma 13 (Subterm Lemma) *If A is legal and B is a subterm of A, then B is legal.*

Lemma 14 (Subject Reduction) *If $\Gamma \vdash A : B$ and $A \to_\beta A'$ then $\Gamma \vdash A' : B$.*

Lemma 15 (Strengthening Lemma) *If $\Gamma, x{:}A, \Delta \vdash B : C$ and $x \notin FV(\Delta) \cup FV(B) \cup FV(C)$, then $\Gamma, \Delta \vdash B : C$.*

The proof of the next lemma is due to van Benthem Jutting [6].

Lemma 16 (Unicity of Types) *If $\mathcal{S}$ is singly sorted, $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$, then $B_1 =_\beta B_2$.*

Lemma 17 (Strong Permutation Lemma) *If $\Gamma, x{:}A, y{:}B, \Delta \vdash C : D$ and $x \notin FV(B)$, then $\Gamma, y{:}B, x{:}A, \Delta \vdash C : D$.*

Definition 18 (Topsort) A sort s is a *topsort* if there is no $s' \in \boldsymbol{S}$ such that $(s, s') \in \boldsymbol{A}$.

Lemma 19 (Topsort Lemma) *If s is a topsort and $\Gamma \vdash A : s$ then A is not of the form $A_1 A_2$ or $\lambda x{:}A_1.A_2$.*

Theorem 20 (Strong Normalisation for ECC) *Let A be a legal term in the Extended Calculus of Constructions. Then A is strongly normalising.*

As the systems of the Barendregt Cube are subsystems of ECC, all legal terms in the systems of the Barendregt Cube are strongly normalising, too.

2.2 Extending PTSs with parametric constants and definitions

Definition 21 The set $\mathcal{T}_P$ of *parametric terms* is defined together with the set $\mathcal{L}_V$ of *lists of typed variables* and the set $\mathcal{L}_T$ of *lists of terms* as follows:

$$\begin{array}{l}\mathcal{T}_P ::= \mathcal{V} \mid \boldsymbol{S} \mid \mathcal{C}(\mathcal{L}_T) \mid \mathcal{T}_P\mathcal{T}_P \mid \lambda\mathcal{V}{:}\mathcal{T}_P.\mathcal{T}_P \mid \Pi\mathcal{V}{:}\mathcal{T}_P.\mathcal{T}_P \mid \mathcal{C}(\mathcal{L}_V){=}\mathcal{T}_P{:}\mathcal{T}_P \texttt{ in } \mathcal{T}_P;\\ \mathcal{L}_V ::= \varnothing \mid \langle \mathcal{L}_V, \mathcal{V}{:}\mathcal{T}_P\rangle;\\ \mathcal{L}_T ::= \varnothing \mid \langle \mathcal{L}_T, \mathcal{T}_P\rangle.\end{array}$$

where, as usual, $\mathcal{V}$ is a set of variables, $\mathcal{C}$ is a set of constants, and $\boldsymbol{S}$ is a set of sorts. Formally, lists of variables are of the form $\langle \ldots \langle\langle \varnothing, x_1{:}A_1\rangle, x_2{:}A_2\rangle \ldots x_n{:}A_n\rangle$. We usually write $\langle x_1{:}A_1, \ldots, x_n{:}A_n\rangle$ or even $x_1{:}A_1, \ldots, x_n{:}A_n$. A similar convention is adopted for lists of terms. In a parametric term of the form $c(b_1, \ldots, b_n)$, the subterms $b_1, \ldots, b_n$ are called the *parameters* of the term.

Terms of the form $\mathcal{C}(\mathcal{L}_V){=}\mathcal{T}_P{:}\mathcal{T}_P$ `in` $\mathcal{T}_P$ represent parametric local definitions. An example of such a term is $\texttt{double(x:}\mathbb{N}\texttt{)=(x+x):}\mathbb{N}$ `in` A. The term indicates that a subterm of A of the form $\texttt{double}(P)$ is to be interpreted as $P + P$, and has type $\mathbb{N}$. The definition is local, that is: the scope of the definition is the term A. Local definitions contrast with global definitions. Global definitions are given in a context Γ, and refer to any term that is considered within Γ (see the forthcoming Definition 28). The definition system in AUTOMATH can be compared to the system of global definitions in this paper. However, there are no local definitions in AUTOMATH.

Definition 22 Let $\vec{x}{:}\vec{A}$ denote $x_1{:}A_1, \ldots, x_n{:}A_n$. We extend the definition of $FV(A)$, the set of *free variables* of a term A, to parametric terms:

$$\begin{array}{rl} FV(c(a_1, \ldots, a_n)) &= \bigcup_{i=1}^{n} FV(a_i);\\ FV(c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C) &= \bigcup_{i=1}^{n}(FV(A_i) \setminus \{x_1, \ldots, x_{i-1}\})\\ &\quad \cup \big((FV(A) \cup FV(B)) \setminus \{x_1, \ldots, x_n\}\big) \cup FV(C);\end{array}$$

We similarly define $Cons(A)$, the set of *constants and global definitions* of A:

$$\begin{array}{rl} Cons(s) = Cons(x) &= \emptyset;\\ Cons(c(a_1, \ldots, a_n)) &= \{c\} \cup \bigcup_{i=1}^{n} Cons(a_i);\\ Cons(AB) &= Cons(A) \cup Cons(B);\\ Cons(\lambda x{:}A.B) = Cons(\Pi x{:}A.B) &= Cons(A) \cup Cons(B);\\ Cons(c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C) &= \bigcup_{i=1}^{n} Cons(A_i) \cup Cons(A) \cup Cons(B) \cup (Cons(C) \setminus \{c\}).\end{array}$$

$FV(A) \cup Cons(A)$ forms the *domain* $Dom(A)$ of A.

Remark 23 The definition of $FV(c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C)$ and $Cons(c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C)$ make clear what the binding structure in a term $c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C$ is:

- A variable declaration $x_i{:}A_i$ in the parameter list $\vec{x}{:}\vec{A}$ binds all the occurrences of x_i in A_k, for $k > i$. That is: the type of a parameter x_k may depend on earlier declared parameters;
- Moreover, the declaration $x_i{:}A_i$ binds all the occurrences of x_i in A and B. This corresponds to the intuitive idea of a parametric definition: x_i can serve as a parameter in the definiens A and in the type B of the definiens;
- However, the variable declaration $x_i{:}A_i$ does *not* bind any occurrence of x_i in C. The definiendum c will occur in C only with a list of parameters $a_1, \ldots, a_n$ behind it, so in the form $c(a_1, \ldots, a_n)$. The variables $x_1, \ldots, x_n$ in the definition of c only serve to indicate what the type of the a_is must be (below, we will see that a_i must have type $A_i[x_j{:=}a_j]_{j=1}^{i-1}$), and what the type of the term $c(a_1, \ldots, a_n)$ is (this turns out to be $B[x_j{:=}a_j]_{j=1}^{n}$);
- Moreover, we see that c is not included in the constants of $c(\vec{x}{:}\vec{A}){=}A{:}B \texttt{ in } C$. This is because c is a *local* definition, and acts as a binder for the occurrences of c in C.

Remark 24 Our reasons for including the type B in a local definition $c(\vec{x}:\vec{A})=A:B$ `in` C are:

- We want to remain consistent with other binders, such as λ and Π. In a term $\lambda x{:}A.B$ or $\Pi x{:}A.B$ we mention the type of the binder x, therefore we also mention the type of the binder c in a local definition $c(\vec{x}:\vec{A})=A:B$ `in` C;
- Sometimes $A : B$ indicates that the term A is a proof of a theorem B (using PAT). If we want to use B in the proof of a new theorem B', we must use the proof term A of B in the proof A' of B'. In that case it is attractive to abbreviate A by introducing a definition $c(\vec{x}:\vec{A})=A:B$ `in` A'. It is important to remember that c is (an abbreviation of) a proof of B, and that is a reason to mention B, the type of A, in the definition declaration;
- For practical purposes like proof assistants or proof checkers, it may seem to be problematic to have B in the definition declaration. However, the program does not always have to ask the user to explicitly mention the type of the abbreviation. Often it can find this type itself via a type inference algorithm. Of course, this also depends on whether type inference is decidable in the underlying type system.
  Sometimes, the user may wish to manually enter the type, because he/she may prefer a certain formulation of the type to a β-equivalent formulation that the program automatically offers.

As usual in PTSs, we do not distinguish terms that are equal up to renaming of bound variables: we consider these terms to be syntactically equal. Moreover, we assume the Barendregt variable convention:

Convention 25 Names of bound variables and constants will always be chosen such that they differ from the free ones in a term.

Hence, we do not write $(\lambda \mathtt{x}{:}A.\mathtt{x})\mathtt{x}$ but $(\lambda \mathtt{y}{:}A.\mathtt{y})\mathtt{x}$ instead.
Similarly, we write $\mathtt{c}(\mathtt{x}'{:}A){=}\mathtt{x}'+\mathtt{x}'{:}A$ `in` $\mathtt{c}(\mathtt{c}(\mathtt{c}(\mathtt{x})))$ instead of $\mathtt{c}(\mathtt{x}{:}A){=}\mathtt{x}+\mathtt{x}{:}A$ `in` $\mathtt{c}(\mathtt{c}(\mathtt{c}(\mathtt{x})))$.

Definition 26 We extend the definition of substitution of a term a for a variable x in a term b, $b[x{:=}a]$, to parametric terms, assuming that x is not a bound variable of either b or a:

$$c(b_1,\ldots,b_n)[x{:=}a] \equiv c(b_1[x{:=}a],\ldots,b_n[x{:=}a]);$$
$$(c(\vec{x}:\vec{A}) = A{:}B \texttt{ in } C)[x{:=}a] \equiv c(x_1{:}A_1[x{:=}a],\ldots,x_n{:}A_n[x{:=}a]){=}A[x{:=}a]{:}B[x{:=}a] \texttt{ in } C[x{:=}a].$$

We now define contexts for type systems with parameters and definitions.

Definition 27 The set of *contexts* which we denote by $\Gamma, \Gamma', \ldots$ is given by:

$$\mathcal{C}_P ::= \varnothing \mid \langle \mathcal{C}_P, \mathcal{V}{:}\mathcal{T}_P\rangle \mid \langle \mathcal{C}_P, \mathcal{C}(\mathcal{L}_V){=}\mathcal{T}_P{:}\mathcal{T}_P\rangle \mid \langle \mathcal{C}_P, \mathcal{C}(\mathcal{L}_V){:}\mathcal{T}_P\rangle.$$

Notice that $\mathcal{L}_V \subseteq \mathcal{C}_P$: all lists of variable declarations are contexts, as well.

Definition 28 Let Γ be a context. Elements $x{:}A$, $c(x_1{:}B_1,\ldots,x_n{:}B_n){:}A$, $c(x_1{:}B_1,\ldots,x_n{:}B_n){=}a{:}A$ of Γ are called *declarations*.

- $x{:}A$ is a *variable declaration*.
  - The variable x is the *subject* of the declaration;
  - A is the *type* or *predicate* of the declaration;
- A declaration of the form $c(x_1{:}B_1,\ldots,x_n{:}B_n){:}A$ is a *constant declaration*.
  - The constant c is the *subject* of the declaration. As c is introduced without further definition, c is called a *primitive constant* (cf. the primitive notions in AUTOMATH);
  - $x_1,\ldots,x_n$ are the *parameters* of the declaration;
  - A is the *type* (*predicate*) of the declaration;
- A declaration $c(x_1{:}B_1,\ldots,x_n{:}B_n){=}a{:}A$ is called a *global definition declaration* or shorthand *global definition* or *definition*.

- The constant c is the *subject* or *definiendum* of the declaration. c is called a *(globally) defined constant*;
- $x_1, \ldots, x_n$ are the *parameters* of the declaration;
- a is the *definiens* of the declaration;
- A is the *type* (*predicate*) of the declaration.

The reasons for including the type of a global definition or a parametric constant in its declaration are the same as for local definitions, cf. Remark 24.

In the rest of this paper, Δ denotes a context $x_1{:}B_1, \ldots, x_n{:}B_n$ consisting of variable declarations only. Such a context is typically used as a list of parameters in a definition $c(\Delta){=}a{:}A$. We write $\Delta_i \equiv x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$ for $i \leq n$.

We extend the definition of substitution to contexts:

Definition 29 Let $\Gamma \in \mathcal{C}_P$, $M \in \mathcal{T}_P$. We define $\Gamma[x{:=}M]$ as follows:

$$\begin{aligned}
\varnothing[x{:=}M] &\equiv \varnothing;\\
\langle \Gamma, x{:}A\rangle[x{:=}M] &\equiv \Gamma[x{:=}M];\\
\langle \Gamma, x'{:}A\rangle[x{:=}M] &\equiv \langle \Gamma[x{:=}M], x'{:}A[x{:=}M]\rangle \qquad \text{if } x \not\equiv x';\\
\langle \Gamma, c(\Delta){:}A\rangle[x{:=}M] &\equiv \langle \Gamma[x{:=}M], c(\Delta[x{:=}M]){:}A[x{:=}M]\rangle;\\
\langle \Gamma, c(\Delta){=}a{:}A\rangle[x{:=}M] &\equiv \langle \Gamma[x{:=}M], c(\Delta[x{:=}M]){=}a[x{:=}M]{:}A[x{:=}M]\rangle.
\end{aligned}$$

For a term A we defined $FV(A)$ and $Cons(A)$. For a context Γ we do not form one set $Cons(\Gamma)$, but we split this set into a set $Prim(\Gamma)$, containing the primitive constants of Γ, and a set $Def(\Gamma)$, containing the defined constants of Γ.

Definition 30 Let Γ be a context. We define the free variables, constants and definitions of Γ:

Γ	$FV(\Gamma)$	$Prim(\Gamma)$	$Def(\Gamma)$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\Gamma, x{:}A$	$FV(\Gamma) \cup \{x\}$	$Prim(\Gamma)$	$Def(\Gamma)$
$\Gamma, c(\Delta){:}A$	$FV(\Gamma)$	$Prim(\Gamma) \cup \{c\}$	$Def(\Gamma)$
$\Gamma, c(\Delta){=}a{:}A$	$FV(\Gamma)$	$Prim(\Gamma)$	$Def(\Gamma) \cup \{c\}$

Finally we define the *domain* of Γ, $Dom(\Gamma)$, by $FV(\Gamma) \cup Prim(\Gamma) \cup Def(\Gamma)$.

In ordinary Pure Type Systems we have that, for a legal term A in a legal context Γ, $FV(A) \subseteq FV(\Gamma)$. The type of a free variable in A, therefore, can always be determined via Γ. In our pure type systems with definitions and parameters we will have: $FV(A) \subseteq FV(\Gamma)$ and $Cons(A) \subseteq Prim(\Gamma) \cup Def(\Gamma)$. This has not only as an effect that the type of a free variable or a constant can be determined via Γ, but also that Γ determines whether a constant in A that is not serving as a *local* definition within A, is a defined constant or a primitive constant. We therefore define:

Definition 31 For a context Γ and a term A with $Dom(A) \subseteq Dom(\Gamma)$ we define:
$Def_\Gamma(A) = Cons(A) \cap Def(\Gamma)$; and $Prim_\Gamma(A) = Cons(A) \cap Prim(\Gamma)$.

We see that a constant $c \in C$ can play three roles in a term A, with respect to a context Γ:

- If c occurs in a subterm $(c(\Delta){=}b{:}B \texttt{ in } a)$ of A, then c is a *locally defined constant*;
- If $c \in Def_\Gamma(A)$, then c is a *globally defined constant*;
- If $c \in Prim_\Gamma(A)$ (or $c \notin Dom(\Gamma)$), then c is a *primitive constant.*

A natural condition on a context $\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$ is that all the free variables and constants of a and A are declared in either Γ_1 or Δ, and that all free variables and constants in a declaration $x_i{:}B_i \in \Delta$ are declared in Γ_1, Δ_i (recall that Δ is a standard context $x_1{:}B_1, \ldots, x_n{:}B_n$ and $\Delta_i \equiv x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$). We call such a context *sound*:

Definition 32 $\Gamma \in \mathcal{C}_P$ is *sound* if $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$ implies
$Dom(a) \cup Dom(A) \subseteq Dom(\Gamma_1) \cup Dom(\Delta)$ and $Dom(B_i) \subseteq Dom(\Gamma_1, \Delta_i)$.

The contexts occurring in the type systems proposed in this paper are all sound (see Lemma 44). This fact will be useful when proving properties of these systems.

We will consider several extensions of Pure Type Systems (PTSs).

- An extension that includes globally and locally defined constants is described and studied in [32]: 'PTSs with definitions' (D-PTSs);
- Orthogonally, we can extend PTSs with parameter-free primitive constants. Then we obtain C-PTSs. C-PTSs are not very interesting, as the role of parameter-free primitive constants can usually be imitated by variables.[3] One could agree that a parameter-free primitive constant is a special kind of variable, and promise not to make any (λ or Π) abstraction over such a variable;
- Our first real extension describes PTSs with *parametric* primitive constants, but without definitions (C^p-PTSs). The C^p-PTSs include the C-PTSs, as a parameter-free primitive constant can be seen as a parametric primitive constant with zero parameters;
- Another extension includes parametric defined constants, and can be seen as a generalisation of D-PTSs: D^p-PTSs;
- We can combine the extensions with primitive constants and defined constants, choosing between parametrised or parameter-free variants. For instance, we can make an extension that includes parameter-free defined constants, and parametric primitive constants. We call this extension C^pD-PTSs.

Combining the various extensions, we obtain a hierarchy that can be depicted as in Figure 2.

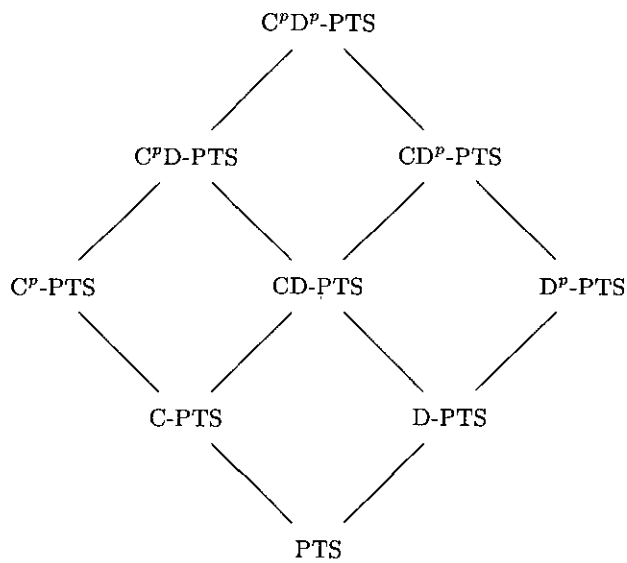


Fig. 2. The hierarchy of parameters and definitions

We give some examples of the possibilities of parameters and definitions.

Example 33 We illustrate the difference between PTSs, C-PTSs and C^p-PTSs.

- In the PTS $\lambda\!\!\rightarrow$ (with only one axiom $* : \Box$ and 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;

[3] There are, however, extensions of PTSs in which constants play an essential role. See for instance the Modal PTSs in the thesis of Borghuis [9], p. 28–29.

- 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 $N{:}*, o{:}N \vdash t : N$ for some term t, it is technically possible to make a λ-abstraction over the variable o and obtain $N{:}* \vdash \lambda o{:}N.t : N \to N$. This is permitted since o is introduced as a variable, but it is probably not what we had in mind.
  In C-PTSs 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 $\mathtt{proof}(\Sigma)$ of proofs of Σ. This cannot be done in the PTS $\lambda{\to}$ extended with (unparametrised) constants: such a constant `proof` should be of type $\mathtt{prop} \to \mathtt{type}$ and this type cannot be constructed in $\lambda{\to}$ (notice that $\mathtt{type} \equiv *$, so the construction of $\mathtt{prop} \to \mathtt{type}$ would involve the Π-formation rule $(*, \Box, \Box)$).
  However, the term `proof` will hardly ever be used on its own. It is usually used when applied to a proposition Σ. In C^p-PTSs it is possible to introduce a parametric version of `proof` by the following context declaration: $\mathtt{proof}(\mathtt{p{:}prop}) : \mathtt{type}$.
  This does not involve the construction of a type $\mathtt{prop} \to \mathtt{type}$. Nevertheless it is possible to construct the term $\mathtt{prop}(P)$ for any term $P : \mathtt{prop}$. 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 $T : \mathtt{prop}$.
  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. We give an example in Section 7.

Example 34 We illustrate the difference between PTSs, D-PTSs and D^p-PTSs.

- In a simple PTS like $\lambda{\to}$ one can derive the following statement for an identity function: $\alpha{:}* \vdash (\lambda \mathtt{x}{:}\alpha.\mathtt{x}) : \alpha \to \alpha$;
- The same derivation can be made in the corresponding D-PTS, but in that D-PTS we have the possibility of abbreviating the term $\lambda \mathtt{x}{:}\alpha.\mathtt{x}$. We can do this in two ways:
  1. We can introduce this definition in the context: $\alpha{:}*, \mathtt{id}{=}(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha) \vdash \mathtt{id} : \alpha{\to}\alpha$.
  2. If we use this definition in the context and derive some judgement $\alpha : *, \mathtt{id}{=}(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha) \vdash t : T$, then the rules of D-PTSs permit us to introduce a local definition resulting in derivation of the judgement $\alpha : * \vdash \mathtt{id}{=}(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha)\ \mathtt{in}\ t : \mathtt{id}{=}(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha)\ \mathtt{in}\ T$. We see that the definition of `id` now appears both in the term and in the type of the term, but no longer in the context. The advantages of definitions are:
     - We can abbreviate long expressions. This makes terms more *surveyable*: `id` is shorter than $\lambda \mathtt{x}{:}\alpha.\mathtt{x}$;
     - We can give names to important expressions. This makes terms more *understandable*: `id` expresses that we are dealing with the identity function, whilst $\lambda \mathtt{x}{:}\alpha.\mathtt{x}$ does not express this fact;
- In a D^p-PTS we have more options for abbreviating the identity function.

- First of all, we can make the same derivation as in the D-PTS. Formally, there is a small difference: we cannot use `id` but must work with `id()`, a parametric term with zero parameters (as in D^p-PTSs we can only work with parametric definitions). We obtain (in the case of the global definition): $\alpha{:}*, \mathtt{id}()=(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha) \vdash \mathtt{id}() : \alpha{\to}\alpha$;
- But we could also decide to use parameters in the definition of `id`. For instance, we could parametrise the variable α resulting in the declaration $\mathtt{id}(\alpha{:}*)=(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha)$.

  If we want to use this declaration, we must have a term T of type $*$. Assuming that we have such a term T, we can derive: $\mathtt{id}(\alpha{:}*)=(\lambda \mathtt{x}{:}\alpha.\mathtt{x}){:}(\alpha{\to}\alpha) \vdash \mathtt{id}(T) : T \to T$.

  We see that we obtain a restricted form of polymorphism in this way. The type system may not allow the construction of $\lambda\alpha{:}*.\lambda \mathtt{x}{:}\alpha.\mathtt{x}$; nevertheless the parameter mechanism makes it possible to express $\mathtt{id}(T)$ for any type $T : *$;
- We could also decide to parametrise the variable `x`, and leave the variable α unparametrised. This yields a context $\alpha{:}*, \mathtt{id}(\mathtt{x}{:}\alpha)=\mathtt{x}{:}\alpha$. We see that the λ-abstraction $\lambda \mathtt{x}{:}\alpha.\mathtt{x}$ is parametrised now. The definition declaration means: For any term t of type α, the term $\mathtt{id}(t)$ of type α is defined by t. If we have such a term t, then we can derive $\alpha{:}*, \mathtt{id}(\mathtt{x}{:}\alpha)=\mathtt{x}{:}\alpha \vdash \mathtt{id}(t) : \alpha$. Observe that $\mathtt{id}(t)$ does not have type $\alpha \to \alpha$ (as was the case with `id`) but type α (which would also be the type of `id` t if we had used the identity $\mathtt{id}{=}\lambda \mathtt{x}{:}\alpha.\mathtt{x}$ from λ-calculus);
- Finally, one could parametrise both α and `x`. This results in a declaration

  $\mathtt{id}(\alpha{:}*, \mathtt{x}{:}\alpha)=\mathtt{x}{:}\alpha$ in the context. If we have a term T of type $*$ and a term t of type T, we can derive $\mathtt{id}(\alpha{:}*, \mathtt{x}{:}\alpha)=\mathtt{x}{:}\alpha \vdash \mathtt{id}(T, t) : T$.

The global definitions given in the D^p-PTS case could also be made local, as was done in the D-PTS case.

We now start a more detailed description of the various extensions of PTSs with definitions and parameters. We define two reduction relations, namely the δ- and β-reduction. β-reduction is defined as usual, and we use $\to_\beta$, $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\beta^+$, and $=_\beta$ as usual. As far as global definitions are concerned, δ-reduction is comparable to δ-reduction in AUTOMATH. This is reflected in rule (δ1) in the definition below. But now, a δ-reduction step can also unfold *local* definitions. Therefore, two new reduction steps are introduced. Rule (δ2) below removes the declaration of a local definition if there is no position within its scope where it can be unfolded ('removal of void local definitions'). Rule (δ3) shows how one can treat a local definition as a global definition, and thus how the problem of unfolding local definitions can be reduced to unfolding global definitions ('localisation of global definitions'). Remember that $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$.

Definition 35 We define the following three reduction rules:

$$\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n) \to_\delta a[x_i{:=}b_i]_{i=1}^n \quad (\delta 1)$$

$$\Gamma \vdash c(\Delta){=}a{:}A \ \mathtt{in}\ b \to_\delta b \quad \text{if } c \notin \mathit{Cons}(b) \quad (\delta 2)$$

$$\frac{\Gamma, c(\Delta){=}a{:}A \vdash b \to_\delta b'}{\Gamma \vdash c(\Delta){=}a{:}A \ \mathtt{in}\ b \to_\delta c(\Delta){=}a{:}A \ \mathtt{in}\ b'} \quad (\delta 3)$$

Furthermore, we have some compatibility rules. These rules are not very complicated, but unfortunately we need quite a lot of them.

Definition 36 We define the following compatibility rules:

$$\frac{\Gamma,\Delta \vdash a \to_\delta a'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(\Delta){=}a'{:}A \text{ in } b} \qquad \frac{\Gamma,\Delta \vdash A \to_\delta A'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(\Delta){=}a{:}A' \text{ in } b}$$

$$\frac{\Gamma,\Delta_i \vdash B_i \to_\delta B_i'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(x_1{:}B_1,\ldots,x_i{:}B_i',\ldots,x_n{:}B_n){=}a{:}A \text{ in } b}$$

$$\frac{\Gamma \vdash a \to_\delta a'}{\Gamma \vdash ab \to_\delta a'b} \qquad \frac{\Gamma \vdash b \to_\delta b'}{\Gamma \vdash ab \to_\delta ab'}$$

$$\frac{\Gamma,x{:}A \vdash a \to_\delta a'}{\Gamma \vdash \lambda x{:}A.a \to_\delta \lambda x{:}A.a'} \qquad \frac{\Gamma \vdash A \to_\delta A'}{\Gamma \vdash \lambda x{:}A.a \to_\delta \lambda x{:}A'.a}$$

$$\frac{\Gamma,x{:}A \vdash a \to_\delta a'}{\Gamma \vdash \Pi x{:}A.a \to_\delta \Pi x{:}A.a'} \qquad \frac{\Gamma \vdash A \to_\delta A'}{\Gamma \vdash \Pi x{:}A.a \to_\delta \Pi x{:}A'.a}$$

$$\frac{\Gamma \vdash a_j \to_\delta a_j'}{\Gamma \vdash c(a_1,\ldots,a_n) \to_\delta c(a_1,\ldots,a_j',\ldots,a_n)}$$

Remark 37 One might also expect a compatibility rule

$$\frac{\Gamma \vdash b \to_\delta b'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(\Delta){=}a{:}A \text{ in } b'}.$$

However, this rule is a derived rule (see the forthcoming Lemma 47).

Now we can give a formal definition of δ-reduction:

Definition 38 δ-reduction is defined as the smallest relation $\to_\delta$ on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules (δ1), (δ2) and (δ3) of Definition 35 and under the compatibility rules of Definition 36.
$\Gamma \vdash \cdot =_\delta \cdot$ denotes the reflexive, symmetric and transitive closure of $\Gamma \vdash \cdot \to_\delta \cdot$.

When Γ is the empty context, we write $a \to_\delta a'$ instead of $\Gamma \vdash a \to_\delta a'$.
We extend $\to_\delta$ to contexts:

Definition 39 δ-reduction between contexts is the smallest relation $\to_\delta$ on $\mathcal{C}_P \times \mathcal{C}_P$ closed under the following rules:

$$\frac{\Gamma_1 \vdash A \to_\delta A'}{\Gamma_1,x{:}A,\Gamma_2 \to_\delta \Gamma_1,x{:}A',\Gamma_2} \qquad \frac{\Gamma_1,\Delta \to_\delta \Gamma_1,\Delta'}{\Gamma_1,c(\Delta){:}A,\Gamma_2 \to_\delta \Gamma_1,c(\Delta'){:}A,\Gamma_2}$$

$$\frac{\Gamma_1,\Delta \vdash A \to_\delta A'}{\Gamma_1,c(\Delta){:}A,\Gamma_2 \to_\delta \Gamma_1,c(\Delta){:}A',\Gamma_2} \qquad \frac{\Gamma_1,\Delta \vdash a \to_\delta a'}{\Gamma_1,c(\Delta){=}a{:}A,\Gamma_2 \to_\delta \Gamma_1,c(\Delta){=}a'{:}A,\Gamma_2}$$

$$\frac{\Gamma_1,\Delta \to_\delta \Gamma_1,\Delta'}{\Gamma_1,c(\Delta){=}a{:}A,\Gamma_2 \to_\delta \Gamma_1,c(\Delta'){=}a{:}A,\Gamma_2} \qquad \frac{\Gamma_1,\Delta \vdash A \to_\delta A'}{\Gamma_1,c(\Delta){=}a{:}A,\Gamma_2 \to_\delta \Gamma_1,c(\Delta){=}a{:}A',\Gamma_2}$$

We now describe the extensions to PTSs that are needed to obtain C^p-PTSs and D^p-PTSs. We don't discuss D-PTSs and C^{p}D-PTSs: D-PTSs are introduced in [32] and C^{p}D-PTSs can be constructed by extending D-PTSs with the additional rules for C^p-PTSs.

Definition 40 (C^p-PTS: Pure type systems with parametric constants) The *typing relation* $\vdash^{C^p}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition 5 and the following ones (we still write $\Delta \equiv x_1{:}B_1,\ldots,x_n{:}B_n$):

$$\textbf{(C}^p\textbf{-weak)} \qquad \frac{\Gamma \vdash^{C^p} b : B \qquad \Gamma,\Delta \vdash^{C^p} A : s}{\Gamma,c(\Delta) : A \vdash^{C^p} b : B}$$

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

where $s \in \boldsymbol{S}$ and the c that is introduced in the C^p-weakening rule is assumed to be Γ-fresh (i.e., it does not occur in Γ).

At first sight one might miss a C^p-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 C^p-application rule.

The extra condition $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^{C^p} A : s$ in the C^p-application rule 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 only needed to assure that the context in the conclusion is a legal context.

Remark 41 If we have a parametric constant in the context, for instance $plus(x : \mathbb{N}, y : \mathbb{N}) : \mathbb{N}$, then it is tempting to think of *plus* as a parametric function. Note however that in PTS-terms it is not a function anymore since the only way to obtain a legal term with it is in its parameterised form $plus(x, y)$ which has type $\mathbb{N}$; $plus(x : \mathbb{N}, y : \mathbb{N})$ itself is not a legal term. In order to talk about properties of *plus* 'as a function' we are forced to consider $\lambda x : \mathbb{N}.\lambda y : \mathbb{N}.plus(x, y)$.

Adapting the rules for $\vdash^{C^p}$ and the rules for definitions of [32] results in rules for *parametric definitions*:

Definition 42 (D^p-PTS: Pure type systems with parametric definitions) The *typing relation* $\vdash^{D^p}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition 5 and the following ones:

$$\textbf{(D^p-weak)} \qquad \frac{\Gamma \vdash^{D^p} b : B \qquad \Gamma, \Delta \vdash^{D^p} a : A}{\Gamma, c(\Delta){=}a{:}A \vdash^{D^p} b : B}$$

$$\textbf{(D^p-app)} \qquad \frac{\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash^{D^p} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}\ (i = 1, \ldots, n) \quad \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash^{D^p} a : A \quad (\text{if } n = 0)}{\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash^{D^p} c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}}$$

$$\textbf{(D^p-form)} \qquad \frac{\Gamma, c(\Delta){=}a{:}A \vdash^{D^p} B : s}{\Gamma \vdash^{D^p} c(\Delta){=}a{:}A \text{ in } B : s}$$

$$\textbf{(D^p-intro)} \qquad \frac{\Gamma, c(\Delta){=}a{:}A \vdash^{D^p} b : B \qquad \Gamma \vdash^{D^p} c(\Delta){=}a{:}A \text{ in } B : s}{\Gamma \vdash^{D^p} c(\Delta){=}a{:}A \text{ in } b : c(\Delta){=}a{:}A \text{ in } B}$$

$$\textbf{(D^p-conv)} \qquad \frac{\Gamma \vdash^{D^p} b : B \qquad \Gamma \vdash^{D^p} B' : s \qquad \Gamma \vdash B =_\delta B'}{\Gamma \vdash^{D^p} b : B'}$$

where $s \in \boldsymbol{S}$, and the c that is introduced in the D^p-weakening rule is assumed to be Γ-fresh.

$\vdash^{D^p}$ includes the definition system of [32]: The D^p-application rule for $n = 0$ can be seen as the δ-start rule of D-PTSs. $\vdash^{C^p D^p}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ that is closed under the rules of Definition 5 and the rules of $\vdash^{C^p}$ and $\vdash^{D^p}$.

Definition 43 (Pure Type Systems with (parametric) constants and (parametric) definitions) Let $\mathcal{S}$ be a specification (see Definition 2).

- A *pure type system with (parametric) constants* C^p-PTS is denoted as $\lambda^{C^p}(\mathcal{S})$ and consists of a set of terms $\mathcal{T}_P$, a set of contexts $\mathcal{C}_P$, the β-reduction rule and the typing relation $\vdash^{C^p}$;
- A *pure type system with (parametric) definitions* D^p-PTS is denoted as $\lambda^{D^p}(\mathcal{S})$ and consists of a set of terms $\mathcal{T}_P$, a set of contexts $\mathcal{C}_P$, β and δ-reduction and the typing relation $\vdash^{D^p}$;
- A *pure type system with (parametric) constants and (parametric) definitions* C^pD^p-PTS is denoted as $\lambda^{C^p D^p}(\mathcal{S})$ and consists of a set of terms $\mathcal{T}_P$, a set of contexts $\mathcal{C}_P$, β and δ-reduction and the typing relation $\vdash^{C^p D^p}$.

A term a is *legal* (with respect to a certain type system) if there are Γ, b such that either $\Gamma \vdash a : b$ or $\Gamma \vdash b : a$ is derivable (in that type system). Similarly, a context Γ is *legal* if there are a, b such that $\Gamma \vdash a : b$.

All contexts occurring in C^pD^p-PTSs are sound (see Definition 32). As C^pD^p-PTSs are clearly extensions of PTSs, C^p-PTSs and D^p-PTSs, this implies that all contexts occurring in PTSs, C^p-PTSs and D^p-PTSs are sound. We need this fact in many proofs in the next sections.

Lemma 44 *Assume $\Gamma \vdash^{C^pD^p} b : B$. The following holds:*
1. $Dom(b), Dom(B) \subseteq Dom(\Gamma)$; *2. Γ is sound.*

PROOF: We prove the statements (1) and (2) simultaneously by induction on the derivation of $\Gamma \vdash^{C^pD^p} b : B$. We treat the two most important cases:
- (D^p-weakening) $\Gamma, c(\Delta){:=}a{:}A \vdash b{:}B$ because $\Gamma \vdash b{:}B$ and $\Gamma, \Delta \vdash a{:}A$.
  (1) is trivial; (2) follows from the induction hypothesis for (1);
- (D^p-formation) $\Gamma \vdash (c(\Delta){:=}a{:}A \texttt{ in } B) : s$ because $\Gamma, c(\Delta){:=}a{:}A \vdash B : s$.
  (1) follows from the induction hypothesis for (2); (2) is trivial.

⊠

3 Properties of terms

In this section, we prove properties of terms without wondering whether these terms are legal or not. In Section 3.1 we discuss some basic properties, such as a Substitution Lemma, and substitutivity. Section 3.2 is devoted to the Church-Rosser property for $\beta\delta$-reduction, and in Section 3.3 we prove strong normalisation for δ-reduction.

Though we do not restrict ourselves to legal terms in this section, we often demand that the free variables and constants of a term are contained in the domain of a sound context.

3.1 Basic properties

In the following lemma we show that a δ-reduction step remains invariant if we enlarge the context. The proof is done by induction on the definition of $\to_\delta$.

Lemma 45 ($\to_\delta$-weakening)
Let $\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \in \mathcal{C}_P$ be such that $\Gamma_1, \Gamma_3 \vdash b \to_\delta b'$. Then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash b \to_\delta b'$. ⊠

The implications from left to right of the following lemma are a particular case of Lemma 45.

The implications from right to left allow to make the context shorter. The first two parts state that declarations of the form $c(\Delta){:}A$ and $x{:}A$ in a context do not have any influence on the reduction relation $\to_{\beta\delta}$. The last part states that declarations of the form $c(\Delta){:=}a{:}A$ in a context do not have any influence on the $\to_{\beta\delta}$ reduction behaviour of terms $b \in \mathcal{T}_P$ with $c \notin Cons(b)$. This allows to remove definition declarations, as rule ($\delta 2$) of the definition of δ-reduction does for local definitions. The lemma is proved by induction on the definition of $\to_\beta$ and $\to_\delta$.

Lemma 46

1. *Let $\langle \Gamma_1, x{:}A, \Gamma_2 \rangle \in \mathcal{C}_P$ and $b \in \mathcal{T}_P$. $\Gamma_1, \Gamma_2 \vdash b \to_{\beta\delta} b'$ if and only if $\Gamma_1, x{:}A, \Gamma_2 \vdash b \to_{\beta\delta} b'$;*
2. *Let $\langle \Gamma_1, c(\Delta){:}A, \Gamma_2 \rangle \in \mathcal{C}_P$ and $b \in \mathcal{T}_P$. $\Gamma_1, \Gamma_2 \vdash b \to_{\beta\delta} b'$ if and only if $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash b \to_{\beta\delta} b'$;*
3. *Let $\langle \Gamma_1, c(\Delta){:=}a{:}A, \Gamma_2 \rangle \in \mathcal{C}_P$ and $b \in \mathcal{T}_P$ be such that $c \notin Cons(b)$.*
   $\Gamma_1, \Gamma_2 \vdash b \to_{\beta\delta} b'$ if and only if $\Gamma_1, c(\Delta){:=}a{:}A, \Gamma_2 \vdash b \to_{\beta\delta} b'$. ⊠

Now we show that the compatibility rule for $c(\Delta){:=}a{:}A \texttt{ in } b$ when we reduce inside b is a derived rule (and therefore not included in the list of compatibility rules in Definition 36).

Lemma 47 *The following rule is derivable from the ones in the definition of* $\to_\delta$*:*

$$\frac{\Gamma \vdash b \to_\delta b'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(\Delta){=}a{:}A \text{ in } b'}.$$

PROOF: Suppose $\Gamma \vdash b \to_\delta b'$. By Lemma 45, $\Gamma, c(\Delta){=}a{:}A \vdash b \to_\delta b'$. By definition of $\to_\delta$, it follows that $\Gamma \vdash c(\Delta){=}a{:}A \text{ in } b \to_\delta c(\Delta){=}a{:}A \text{ in } b'$. ⊠

The following lemma is proved by induction on the structure of a.

Lemma 48 (Substitution Lemma)
Suppose $x \not\equiv y$ *and* $x \notin FV(d)$*. Then* $a[x{:=}b][y{:=}d] \equiv a[y{:=}d][x{:=}b[y{:=}d]]$. ⊠

The following lemma shows that $\to_\beta$ is substitutive. It is proved by induction on the generation of $\to_\beta$ and by the Substitution Lemma.

Lemma 49 (Substitutivity for $\to_\beta$) *If* $a \to_\beta a'$ *then* $a[x{:=}b] \to_\beta a'[x{:=}b]$. ⊠

The relation $\to_\delta$ is not substitutive when considered as a binary relation (disregarding the context). For example, let $\Gamma \equiv \mathtt{x}{:}\alpha, \mathtt{x}'{:}\alpha, \mathtt{c}() {=} \mathtt{x}{:}\alpha$. We have $\Gamma \vdash^{DP} \mathtt{c}() : \alpha$ and $\Gamma \vdash \mathtt{c}() \to_\delta \mathtt{x}$, but not $\Gamma \vdash \mathtt{c}()[\mathtt{x}{:=}\mathtt{x}'] \to_\delta \mathtt{x}[\mathtt{x}{:=}\mathtt{x}']$.
The reason for this is to be found in the δ-weakening rule. When we introduce a new parametric definition $c(\Delta){=}a{:}A$, the term a may contain free variables that are not in the domain of Δ but in the domain of Γ. When unfolding the definition c, these new variables can appear, thus destroying substitutivity. However, we do have the substitutivity property below. The proof is by induction on the derivation of $\Gamma \vdash a \to_\delta a'$.

Lemma 50 (Substitutivity for $\to_\delta$) *If* $\Gamma \vdash a \to_\delta a'$ *then* $\Gamma[x{:=}b] \vdash a[x{:=}b] \to_\delta a'[x{:=}b]$.

PROOF: Induction on the derivation of $\Gamma \vdash a \to_\delta a'$. ⊠

In the following lemma we reduce inside the term b of $a[x{:=}b]$. The proof is by induction on the structure of a.

Lemma 51 *If* $\Gamma \vdash b \to_{\beta\delta} b'$ *then* $\Gamma \vdash a[x{:=}b] \twoheadrightarrow_{\beta\delta} a[x{:=}b']$. ⊠

3.2 Church-Rosser for $\to_{\beta\delta}$

In this section we prove the Church-Rosser theorem for $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\delta$ and $\twoheadrightarrow_{\beta\delta}$. The proof is of a rather technical nature. We suffice by mentioning the necessary lemmas, for proofs the reader is referred to the appendix.

As for ordinary λ-terms, we have:

Theorem 52 (Church-Rosser theorem for β-reduction) *If* $a \twoheadrightarrow_\beta a_1$ *and* $a \twoheadrightarrow_\beta a_2$ *then there exists a term* a_3 *such that* $a_1 \twoheadrightarrow_\beta a_3$ *and* $a_2 \twoheadrightarrow_\beta a_3$. ⊠

The proof is similar to the proof for λ-terms without definitions and parameters.

For a context Γ and a term b we define $|b|_\Gamma$, which is, intuitively, b in which all definitions are unfolded. That is: both the local definitions inside b, and the global definitions given in Γ. The definition is by induction on the total number of symbols occurring in Γ and b.

Definition 53 For $a \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$ we define a term $|a|_\Gamma \in \mathcal{T}_P$ as follows:

$$\begin{aligned}
|a|_\Gamma &\equiv a \text{ (for } a \equiv x \in \mathcal{V} \text{ or } a \equiv s \in \boldsymbol{S});\\
|c(b_1,\ldots,b_n)|_\Gamma &\equiv \begin{cases} |a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n & \text{if } \Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle;\\ c(|b_1|_\Gamma,\ldots,|b_n|_\Gamma) & \text{if } c \notin \mathit{Def}(\Gamma); \end{cases}\\
|ab|_\Gamma &\equiv |a|_\Gamma |b|_\Gamma;\\
|\mathcal{O}x{:}A.B|_\Gamma &\equiv \mathcal{O}x{:}|A|_\Gamma.|B|_{\Gamma,x:A} \quad \text{if } \mathcal{O} \text{ is } \lambda \text{ or } \Pi;\\
|c(\Delta){=}a{:}A \text{ in } b|_\Gamma &\equiv |b|_{\Gamma,c(\Delta)=a:A} \text{ (where } c \text{ is } \Gamma\text{-fresh).}
\end{aligned}$$

The following lemma shows that $|b|_\Gamma$ is independent from variable declarations $x{:}A$ and (primitive) constant declarations $c(\Delta){:}A$ in Γ. The proof is by induction on the definition of $|b|_{\Gamma_1,\Gamma_2}$.

Lemma 54 *$|b|_{\Gamma_1,\Gamma_2} \equiv |b|_{\Gamma_1,x:A,\Gamma_2}$; and $|b|_{\Gamma_1,\Gamma_2} \equiv |b|_{\Gamma_1,c(\Delta):A,\Gamma_2}$.* ⊠

By induction on the definition of $|b|_\Gamma$ one shows that $|b|_\Gamma$ does not contain any local definitions.

Lemma 55 *For all $b \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$, $|b|_\Gamma$ has no subterms of the form $(c(\Delta){=}a{:}A \texttt{ in } d)$.* ⊠

The intuition on $|b|_\Gamma$ suggests that all definitions of b are unfolded in $|b|_\Gamma$. However, there may be global definitions in Γ that have not been unfolded in $|b|_\Gamma$. Take, for example, $\Gamma \equiv \langle \mathtt{c}()=\mathtt{c}'():*, \mathtt{c}'()=\mathtt{c}''():*\rangle$. Then $|\mathtt{c}()|_\Gamma \equiv |\mathtt{c}'()|_\varnothing \equiv \mathtt{c}'()$, but $\mathtt{c}'()$ is not in δ-normal form with respect to Γ. This is due to the fact that Γ is not a sound context (see Definition 32).

By induction on the definition of $|b|_\Gamma$, we show that if Γ is sound, $|b|_\Gamma$ is equal to b with all the definitions in b and Γ unfolded. It is no serious restriction to consider only sound contexts, as all contexts that appear in $\mathrm{C}^p\mathrm{D}^p$-PTSs are sound (Lemma 44).

Lemma 56 *Let Γ be a sound context such that $Dom(b) \subseteq Dom(\Gamma)$. Then $Dom(|b|_\Gamma) \subseteq Dom(\Gamma) \setminus Def(\Gamma)$.* ⊠

With the above we can show:

Lemma 57 *If Γ is sound and $Dom(d) \subseteq Dom(\Gamma)$, then $\Gamma \vdash d \twoheadrightarrow_\delta |d|_\Gamma$.* ⊠

Corollary 58 *In any C^pD^p-PTS, the relation $\to_\delta$ is weakly normalising, i.e., each legal term has a δ-normal form.*

PROOF: $|b|_\Gamma$ is in δ-normal form (Lemmas 55 and 56) and $|b|_\Gamma$ is a δ-normal form of b (Lemma 57). ⊠

The mapping $|-|\_$ also helps us to show that $\to_{\beta\delta}$ is confluent (Theorem 63). For the proof we use some lemmas:

Lemma 59 *Assume $\langle \Gamma_1, \Gamma_3\rangle$ is sound and $Dom(b) \subseteq Dom(\Gamma_1, \Gamma_3)$. Then $|b|_{\Gamma_1,\Gamma_2,\Gamma_3} \equiv |b|_{\Gamma_1,\Gamma_3}$.* ⊠

Lemma 60 *Assume $\langle \Gamma_1, \Gamma_2\rangle$ is sound, and $Dom(a) \subseteq Dom(\Gamma_1, \Gamma_2)$, $Dom(b) \subseteq Dom(\Gamma_1)$ and $x \notin Dom(\Gamma_1)$. Then $|a|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}] \equiv |a[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}$.* ⊠

Lemma 61 *If $\Gamma \vdash d \to_\delta d'$, Γ is sound, and $Dom(d) \subseteq Dom(\Gamma)$, then $|d|_\Gamma \equiv |d'|_\Gamma$ and $Dom(d') \subseteq Dom(\Gamma)$.*

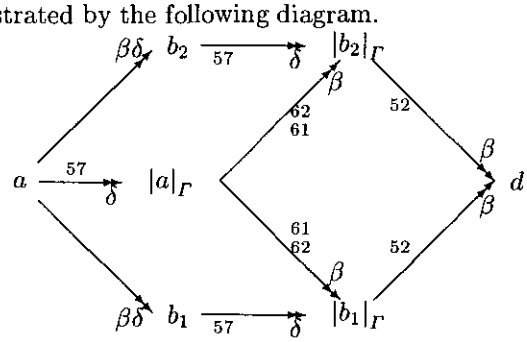
PROOF: We prove the following two statements simultaneously by induction on the definition of $|d|_\Gamma$: 1. If $\Gamma \vdash d \to_\delta d'$ then $|d|_\Gamma \equiv |d'|_\Gamma$; and 2. If $\Gamma \to_\delta \Gamma'$ then $|d|_\Gamma \equiv |d|_{\Gamma'}$. ⊠

Lemma 62 *If Γ is sound, $Dom(d) \subseteq Dom(\Gamma)$ and $d \to_\beta d'$, then $|d|_\Gamma \twoheadrightarrow_\beta |d'|_\Gamma$.* ⊠

The proof is similar to the proof of Lemma 61.

Theorem 63 (Confluence for $\to_{\beta\delta}$) *If Γ is sound, $\Gamma \vdash a \twoheadrightarrow_{\beta\delta} b_1$ and $\Gamma \vdash a \twoheadrightarrow_{\beta\delta} b_2$ then there exists a term d such that $\Gamma \vdash b_1 \twoheadrightarrow_{\beta\delta} d$ and $\Gamma \vdash b_2 \twoheadrightarrow_{\beta\delta} d$.*

PROOF: The proof is illustrated by the following diagram.



⊠

3.3 Strong normalisation for $\rightarrow_\delta$

In [15], van Daalen presents a proof (originally due to de Bruijn) of strong normalisation for a definition system that is at the basis of AUTOMATH. De Vrijer uses a similar technique to prove the finite developments theorem [34]. A similar technique to the one of de Vrijer is also used in [32] to prove strong normalisation for δ-reduction in D^p-PTSs. We extend these techniques to prove strong normalisation for δ-reduction in $\mathrm{C}^p\mathrm{D}^p$-PTSs.

First we define the multiplicity $\mathtt{M}_z(\Gamma, a)$ of a variable z in a term a, depending on a context Γ.

Definition 64 For $z \in \mathcal{V}$, $\Gamma \in \mathcal{C}_P$ and $a \in \mathcal{T}_P$ we define a natural number $\mathtt{M}_z(\Gamma, a)$ by induction on the total number of symbols in Γ and a.

$$
\begin{aligned}
\mathtt{M}_z(\Gamma, z) &= 1;\\
\mathtt{M}_z(\Gamma, a) &= 0 \text{ if } a \equiv x \not\equiv z \text{ or } a \equiv s \in \boldsymbol{S};\\
\mathtt{M}_z(\Gamma, c(b_1, \ldots, b_n)) &= \begin{cases} \mathtt{M}_z(\langle \Gamma_1, \Delta\rangle, a) + \sum_{i=1}^n \mathtt{M}_z(\Gamma, b_i) \cdot \max(1, \mathtt{M}_{x_i}(\langle \Gamma_1, \Delta\rangle, a)) \\ \text{if } \Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle; \\ \sum_{i=1}^n \mathtt{M}_z(\Gamma, b_i) \text{ otherwise;} \end{cases}\\
\mathtt{M}_z(\Gamma, c(\Delta){=}a{:}A \text{ in } b) &= \mathtt{M}_z(\langle \Gamma, \Delta\rangle, a) + \mathtt{M}_z(\langle \Gamma, \Delta\rangle, A) + \textstyle\sum_{i=1}^n \mathtt{M}_z(\langle \Gamma, \Delta_i\rangle, B_i) + \mathtt{M}_z(\langle \Gamma, c(\Delta){=}a{:}A\rangle, b);\\
\mathtt{M}_z(\Gamma, ab) &= \mathtt{M}_z(\Gamma, a) + \mathtt{M}_z(\Gamma, b);\\
\mathtt{M}_z(\Gamma, \mathcal{O}x{:}A.a) &= \mathtt{M}_z(\langle \Gamma, x{:}A\rangle, a) + \mathtt{M}_z(\Gamma, A); \text{ if } \mathcal{O} \text{ is } \lambda \text{ or } \Pi.
\end{aligned}
$$

Following the line of [34] one can prove the following lemma (using induction on the definitions of $\mathtt{M}_-(-,-)$):

Lemma 65

1. *If Γ is sound, $Dom(a) \subseteq Dom(\Gamma)$ and $x \notin FV(a) \cup FV(\Gamma)$, then $\mathtt{M}_x(\Gamma, a) = 0$;*
2. *If $\langle \Gamma_1, \Gamma_3\rangle$ is sound and $Dom(a) \subseteq Dom(\langle \Gamma_1, \Gamma_3\rangle)$, then $\mathtt{M}_x(\langle \Gamma_1, \Gamma_3\rangle, a) = \mathtt{M}_x(\langle \Gamma_1, \Gamma_2, \Gamma_3\rangle, a)$;*
3. *If $\langle \Gamma_1, \Gamma_2\rangle$ is sound, $Dom(b) \subseteq Dom(\langle \Gamma_1, \Gamma_2\rangle)$, $Dom(a) \subseteq Dom(\Gamma_1)$, $x \not\equiv z$ and $x \notin FV(\Gamma_1)$, then $\mathtt{M}_z(\langle \Gamma_1, \Gamma_2[x{:=}a]\rangle, b[x{:=}a]) = \mathtt{M}_z(\langle \Gamma_1, \Gamma_2\rangle, b) + \mathtt{M}_z(\Gamma_1, a) \cdot \mathtt{M}_x(\langle \Gamma_1, \Gamma_2\rangle, b)$.* ⊠

The following lemma requires a somewhat more complicated proof than in [34], as contexts are involved in our situation.

Lemma 66 *Let Γ be sound, $Dom(a) \subseteq Dom(\Gamma)$. If $\Gamma \vdash a \rightarrow_\delta b$, then $\mathtt{M}_x(\Gamma, a) \geq \mathtt{M}_x(\Gamma, b)$.*

PROOF: We simultaneously prove, using induction on the total number of symbols in Γ and a, the following two statements:
1. If $\Gamma \vdash a \rightarrow_\delta b$, then $\mathtt{M}_x(\Gamma, a) \geq \mathtt{M}_x(\Gamma, b)$; 2. If $\Gamma \rightarrow_\delta \Gamma'$, then $\mathtt{M}_x(\Gamma, a) \geq \mathtt{M}_x(\Gamma', a)$.
The proof is straightforward, using the lemma above. ⊠

Next we define, for $\Gamma \in \mathcal{C}_P$ and $a \in \mathcal{T}_P$, a natural number $\mathtt{L}_\Gamma(a)$ that decreases with each δ reduction step. It is similar to the mappings defined in [34] (used to prove the finite developments theorem), in [15] and in [32] (used to prove strong normalisation of δ-reduction). This function $\mathtt{L}_-(-)$ computes an upper bound for the length of the longest δ-reduction path from a term to its δ-normal form.

Definition 67 For $\Gamma \in \mathcal{C}_P$ and $a \in \mathcal{T}_P$ we define $\mathtt{L}_\Gamma(a)$ by induction on the total number of symbols in Γ and a:

$$
\begin{aligned}
\mathtt{L}_\Gamma(a) &= 0 \text{ if } a \equiv x \in \mathcal{V} \text{ or } a \equiv s \in \boldsymbol{S};\\
\mathtt{L}_\Gamma(c(b_1, \ldots, b_n)) &= \begin{cases} \mathtt{L}_{\langle \Gamma_1, \Delta\rangle}(a) + \sum_{i=1}^n \mathtt{L}_\Gamma(b_i) \cdot \max(1, \mathtt{M}_{x_i}(\langle \Gamma_1, \Delta\rangle, a)) + 1 \\ \text{if } \Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle; \\ \sum_{i=1}^n \mathtt{L}_\Gamma(b_i) \text{ otherwise;} \end{cases}\\
\mathtt{L}_\Gamma(c(\Delta){=}a{:}A \text{ in } b) &= \mathtt{L}_{\langle \Gamma, \Delta\rangle}(a) + \mathtt{L}_{\langle \Gamma, \Delta\rangle}(A) + \textstyle\sum_{i=1}^n \mathtt{L}_{\langle \Gamma, \Delta_i\rangle}(B_i) + \mathtt{L}_{\langle \Gamma, c(\Delta){=}a{:}A\rangle}(b) + 1;\\
\mathtt{L}_\Gamma(ab) &= \mathtt{L}_\Gamma(a) + \mathtt{L}_\Gamma(b);\\
\mathtt{L}_\Gamma(\mathcal{O}x{:}A.a) &= \mathtt{L}_{\langle \Gamma, x{:}A\rangle}(a) + \mathtt{L}_\Gamma(A); \quad \text{if } \mathcal{O} \text{ is } \lambda \text{ or } \Pi
\end{aligned}
$$

Similar properties as in Lemma 65 and Lemma 66 hold for $\mathtt{L}_{\_}(-)$:

Lemma 68

1. *If* $\langle \Gamma_1, \Gamma_3 \rangle$ *is sound,* $Dom(a) \subseteq Dom(\langle \Gamma_1, \Gamma_3 \rangle)$, *then* $\mathtt{L}_{\langle \Gamma_1, \Gamma_3 \rangle}(a) = \mathtt{L}_{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle}(a)$;
2. *If* $\langle \Gamma_1, \Gamma_2 \rangle$ *is sound,* $Dom(b) \subseteq Dom(\langle \Gamma_1, \Gamma_2 \rangle)$, $Dom(a) \subseteq Dom(\Gamma_1)$, *and* $x \notin FV(\Gamma_1)$, *then* $\mathtt{L}_{\langle \Gamma_1, \Gamma_2[x:=a] \rangle}(b[x:=a]) = \mathtt{L}_{\langle \Gamma_1, \Gamma_2 \rangle}(b) + \mathtt{L}_{\Gamma_1}(a) \cdot \mathtt{M}_x(\langle \Gamma_1, \Gamma_2 \rangle, b)$.

The lemma above is used to prove the crucial property of $\mathtt{L}_{\_}(-)$:

Lemma 69 *If* Γ *is sound,* $Dom(a) \subseteq Dom(\Gamma)$ *and* $\Gamma \vdash a \to_\delta b$, *then* $\mathtt{L}_\Gamma(a) > \mathtt{L}_\Gamma(b)$.

PROOF: Similar to the proof of Lemma 66. ⊠

Theorem 70 (Strong Normalisation for δ) *The reduction* δ *(when restricted to sound contexts* Γ *and terms* a *with* $Dom(a) \subseteq Dom(\Gamma)$*) is strongly normalising, i.e. there are no infinite* δ*-reduction paths.*

PROOF: This follows from lemma 69. ⊠

Without the restriction to sound contexts Γ and terms a with $Dom(a) \subseteq Dom(\Gamma)$, we do not even have weak normalisation: take $\Gamma \equiv \langle \mathtt{c}() = \mathtt{d}() : \mathtt{A}, \mathtt{d}() = \mathtt{c}() : \mathtt{A} \rangle$. The term $\mathtt{c}()$ does not have a Γ-normal form.

4 Properties of legal terms

The properties in this section are proved for all terms that are legal in a pure type system with parameters, i.e. for terms a for which there are A, Γ such that $\Gamma \vdash^{C^p D^p} a : A$ or $\Gamma \vdash^{C^p D^p} A : a$. The main property we prove is that strong normalisation of a PTS is preserved by certain extensions.

Many of the standard properties of PTSs in [3], [16] hold for C^pD^p-PTSs as well. In the same way as in [3], [16] we can prove the Substitution Lemma, Correctness of Types, Subject Reduction (for $\beta\delta$-reduction) and Uniqueness of Types (for singly sorted C^pD^p-PTSs):

Theorem 71 *Let* $\mathcal{S}$ *be a specification. The type system* $\lambda^{C^p D^p}(\mathcal{S})$ *has the following properties:*

- *Substitution Lemma;*
- *Correctness of Types;*
- *Subject Reduction (for* $\to_{\beta\delta}$*).*

Moreover, if $\mathcal{S}$ *is singly sorted then* $\lambda^{C^p D^p}(\mathcal{S})$ *has Uniqueness of Types.* ⊠

The Generation Lemma is extended with two extra cases:

Lemma 72 (Generation Lemma, extension)

1. *If* $\Gamma \vdash^{C^p D^p} c(b_1, \ldots, b_n) : D$ *then there exist sort* s, $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$ *and term* A *such that* $\Gamma \vdash D =_{\beta\delta} A[x_i{:=}b_i]_{i=1}^n$, *and* $\Gamma \vdash^{C^p D^p} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$. *Besides we have one of these two possibilities:*
   (a) *Either* $\Gamma = \langle \Gamma_1, c(\Delta){:}A, \Gamma_2 \rangle$ *and* $\Gamma_1, \Delta \vdash^{C^p D^p} A : s$;
   (b) *Or* $\Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \rangle$ *and* $\Gamma_1, \Delta \vdash^{C^p D^p} a : A$;
2. *If* $\Gamma \vdash^{C^p D^p} c(\Delta){=}a{:}A$ `in` $b : D$ *then we have two possibilities:*
   (a) *Either* $\Gamma, c(\Delta){=}a{:}A \vdash^{C^p D^p} b : B$, $\Gamma \vdash^{C^p D^p} (c(\Delta){=}a{:}A$ `in` $B) : s$ *and* $\Gamma \vdash D =_{\beta\delta} c(\Delta){=}a{:}A$ `in` B;
   (b) *Or* $\Gamma, c(\Delta){=}a{:}A \vdash^{C^p D^p} b : s$ *and* $\Gamma \vdash D =_{\beta\delta} s$.

⊠

In case 1(b) we do not necessarily have $\Gamma_1, \Delta \vdash^{C^p D^p} A : s$. For instance, in the $C^p D^p$-PTSs of the Barendregt Cube one can abbreviate terms a of type $\Box$, whilst $\Box$ is not typable in these systems.

Also Correctness of Contexts has some extra cases compared to usual PTSs. Recall that Γ is *legal* if there are b, B such that $\Gamma \vdash^{C^p D^p} b : B$.

Lemma 73 (Correctness of Contexts)

1. *If $\Gamma, x{:}A, \Gamma'$ is legal then there exists a sort s such that $\Gamma \vdash^{C^p D^p} A : s$;*
2. *If $\Gamma, c(\Delta){:}A, \Gamma'$ is legal then $\Gamma, \Delta \vdash^{C^p D^p} A : s$;*
3. *If $\Gamma, c(\Delta){=}a{:}A, \Gamma'$ is legal then $\Gamma, \Delta \vdash^{C^p D^p} a : A$.* ⊠

Again, in case 3 we do not necessarily have $\Gamma, \Delta \vdash^{C^p D^p} A : s$.

Now we prove that $\lambda^{C^p D^p}(\mathcal{S})$ is $\beta\delta$-strongly normalising if a slightly larger PTS $\lambda(\mathcal{S}')$ is β-strongly normalising. The proof follows the same ideas of [32] to prove that a PTS extended with definitions is $\beta\delta$-strongly normalising.

For legal terms $a \in \mathcal{T}_P$ in a context Γ, we define a lambda term $\|a\|_\Gamma$ without definitions and without parameters. If a is typable in a $C^p D^p$-PTS $\lambda^{C^p D^p}(\mathcal{S})$, then $\|a\|_\Gamma$ will be typable in a PTS $\lambda(\mathcal{S}')$, where $\mathcal{S}'$ is a so-called *completion* (see Definition 81) of the specification $\mathcal{S}$. Moreover, we take care that if $a \to_\beta a'$, then $\|a\|_\Gamma \twoheadrightarrow_\beta^+ \|a'\|_\Gamma$ (that is: $\|a\|_\Gamma \twoheadrightarrow_\beta \|a'\|_\Gamma$ and $\|a\|_\Gamma \not\equiv \|a'\|_\Gamma$). Together with strong normalisation of δ-reduction (Theorem 70), this guarantees that $\lambda^{C^p D^p}(\mathcal{S})$ is $\beta\delta$-strongly normalising whenever $\lambda(\mathcal{S}')$ is β-strongly normalising.

We suppose that $\mathcal{V} \cup \mathcal{C}$, the set of variables and constants that are used to define $\mathcal{T}_P$, is included in the set of variables that is used to define $\mathcal{T}$, the set of terms used for the PTS $\lambda(\mathcal{S}')$.

Δ still denotes a list of variables with types $x_1{:}B_1, \ldots, x_n{:}B_n$ and Δ_i is an abbreviation for $x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$. $\lambda\,\Delta.a$ denotes $\lambda_{i=1}^n x_i{:}B_i.a$ and $\prod \Delta.A$ denotes $\prod_{i=1}^n x_i{:}B_i.A$.

Definition 74 For $a \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$ we define $\|a\|_\Gamma$ as follows:

$$
\begin{aligned}
\|a\|_\Gamma &\equiv a \text{ if } a \equiv s \in \boldsymbol{S} \text{ or } a \equiv x \in \mathcal{V}\,;\\
\|c(b_1, \ldots, b_n)\|_\Gamma &\equiv \begin{cases} \|\lambda\,\Delta.a\|_{\Gamma_1}\, \|b_1\|_\Gamma \cdots \|b_n\|_\Gamma \text{ if } \Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle; \\ c\,\|b_1\|_\Gamma, \cdots, \|b_n\|_\Gamma \text{ otherwise;} \end{cases}\\
\|ab\|_\Gamma &\equiv \|a\|_\Gamma\, \|b\|_\Gamma\,;\\
\|\mathcal{O}x{:}A.b\|_\Gamma &\equiv \mathcal{O}x{:}\|A\|_\Gamma \,.\, \|b\|_{\Gamma, x:A}\,; \text{ if } \mathcal{O} \text{ is } \lambda \text{ or } \Pi\\
\|c(\Delta){=}a{:}A \texttt{ in } b\|_\Gamma &\equiv \left(\lambda c{:}(\|\textstyle\prod \Delta.A\|_\Gamma) \,.\, \|b\|_{\Gamma, c(\Delta)=a:A}\right) \|\lambda\,\Delta.a\|_\Gamma\,.
\end{aligned}
$$

The mapping $\|\_\|_\_$ is slightly different from the mapping $|\_|_\_$. This is because we want $\|\_\|_\_$ to maintain β-reductions. In a term $c(\Delta){=}a{:}A \texttt{ in } b$, there may be β-redexes in Δ, a or A. These redexes may be lost in $|c(\Delta){=}a{:}A \texttt{ in } b|_\Gamma \equiv |b|_{\Gamma, c(\Delta)=a:A}$. Due to the extra λ-abstraction in the definition of $\|c(\Delta){=}a{:}A \texttt{ in } b\|_\Gamma$, the possible β-redexes in Δ, a and A are maintained.

The mapping $\|\_\|_\_$ is extended to contexts.

Definition 75 For a context $\Gamma \in \mathcal{C}_P$ we define $\|\Gamma\|$ as follows:

$$
\begin{aligned}
\|\varnothing\| &\equiv \varnothing;\\
\|\Gamma, x{:}A\| &\equiv \|\Gamma\|, x{:}\|A\|_\Gamma\,;\\
\|\Gamma, c(\Delta){:}A\| &\equiv \|\Gamma\|, c{:}(\|\textstyle\prod \Delta.A\|_\Gamma)\,;\\
\|\Gamma, c(\Delta){=}a{:}A\| &\equiv \|\Gamma\|, c{:}(\|\textstyle\prod \Delta.A\|_\Gamma)\,.
\end{aligned}
$$

We have similar properties for $\|\_\|_\_$ as for $|\_|_\_$:

Lemma 76 *If $\langle \Gamma_1, \Gamma_3\rangle$ is sound and $Dom(a) \subseteq Dom(\langle \Gamma_1, \Gamma_3\rangle)$, then $\|a\|_{\Gamma_1, \Gamma_2, \Gamma_3} \equiv \|a\|_{\Gamma_1, \Gamma_3}$.* ⊠

The proof is similar to the proof of Lemma 59.

Lemma 77 *Assume $\langle \Gamma_1, \Gamma_2 \rangle$ is sound, and $Dom(a) \subseteq Dom(\langle \Gamma_1, \Gamma_2 \rangle)$, $Dom(b) \subseteq Dom(\Gamma_1)$, and $x \notin Dom(\Gamma_1)$. Then $\|a\|_{\Gamma_1,\Gamma_2}[x:=\|b\|_{\Gamma_1}] \equiv \|a[x:=b]\|_{\Gamma_1,\Gamma_2[x:=b]}$.* ⊠

The proof is similar to the proof of Lemma 60.

We now show that $\|\_\|_{\_}$ translates a δ-reduction into zero or more β-reductions, and that it translates a β-reduction into one or more β-reductions.

Lemma 78 *Let Γ be sound, and assume $Dom(a) \subseteq Dom(\Gamma)$. If $\Gamma \vdash a \to_\delta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta \|b\|_\Gamma$.* ⊠

Lemma 79 *Let Γ be sound, and assume $Dom(a) \subseteq Dom(\Gamma)$. If $a \to_\beta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta^+ \|b\|_\Gamma$.* ⊠

The proof for the cases $c(b_1, \ldots, b_n)$ and $c(\Delta)$=a:A `in` b shows that this lemma does not hold if we use $|\_|_{\_}$ instead of $\|\_\|_{\_}$. The proof for the case $c(\Delta)$=a:A `in` b shows the need to prove that $\|a\|_\Gamma \twoheadrightarrow_\beta \|a\|_{\Gamma'}$ if $\Gamma \to_\beta \Gamma'$.

Definition 80 The specification $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is called *quasi full* if for all $s_1, s_2 \in \boldsymbol{S}$ there exists $s_3 \in \boldsymbol{S}$ such that $(s_1, s_2, s_3) \in \boldsymbol{R}$.

Definition 81 A specification $\mathcal{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ is a *completion* of $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ if

1. $\boldsymbol{S} \subseteq \boldsymbol{S}'$, $\boldsymbol{A} \subseteq \boldsymbol{A}'$, and $\boldsymbol{R} \subseteq \boldsymbol{R}'$;
2. $\boldsymbol{S}'$ is quasi full;
3. for all $s \in \boldsymbol{S}$ there is a sort $s' \in \boldsymbol{S}'$ such that $(s, s') \in \boldsymbol{A}'$.

Theorem 82 *Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $\mathcal{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that $\mathcal{S}'$ is a completion of $\mathcal{S}$. If $\Gamma \vdash_{\mathcal{S}}^{C^p D^p} a : A$ then $\|\Gamma\| \vdash_{\mathcal{S}'} \|a\|_\Gamma : \|A\|_\Gamma$.* ⊠

Now we can prove our normalisation result for C^pD^p-PTSs.

Theorem 83 *Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $\mathcal{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that $\mathcal{S}'$ is a completion of $\mathcal{S}$. If the PTS $\lambda(\mathcal{S}')$ is β-strongly normalising, then the C^pD^p-PTS $\lambda^{C^pD^p}(\mathcal{S})$ is $\beta\delta$-strongly normalising.*

PROOF: Suppose that $\lambda(\mathcal{S}')$ is β-strongly normalising, and suppose towards a contradiction that $\lambda^{C^pD^p}(\mathcal{S})$ is not $\beta\delta$-strongly normalising, i.e. there is an infinite $\beta\delta$-reduction sequence $a_1 \to_{\beta\delta} a_2 \to_{\beta\delta} \ldots$, starting at $a \equiv a_1$ and $\Gamma \vdash^{C^pD^p} a : A$.

Observe that the number of β-reductions in this sequence is infinite. Otherwise there would be $n \in \mathbb{N}$ such that $\Gamma \vdash a_n \to_\delta a_{n+1} \to_\delta a_{n+2} \ldots$, which contradicts the fact that δ-reduction is strongly normalising (Theorem 70).

We conclude that the reduction sequence is of the form $\Gamma \vdash a \twoheadrightarrow_\delta a_{n_1} \to_\beta a_{n_2} \twoheadrightarrow_\delta a_{n_3} \to_\beta a_{n_4} \to \to_\delta \ldots$ By lemmas 78 and 79 there is an infinite β-reduction sequence starting at $\|a\|_\Gamma$:

$$\|a\|_\Gamma \twoheadrightarrow_\beta \|a_{n_1}\|_\Gamma \twoheadrightarrow_\beta^+ \|a_{n_2}\|_\Gamma \twoheadrightarrow_\beta \|a_{n_3}\|_\Gamma \twoheadrightarrow_\beta^+ \|a_{n_4}\|_\Gamma \twoheadrightarrow_\beta \ldots$$

and by Theorem 82, $\|\Gamma\| \vdash_{\mathcal{S}'} \|a\|_\Gamma : \|A\|_\Gamma$, which contradicts the assumption that $\lambda(\mathcal{S}')$ is β-strongly normalising. ⊠

Since ECC is β-strongly normalising and a completion of all systems of the λ-cube, Theorem 83 guarantees that all systems of the λ-cube are $\beta\delta$-strongly normalising. Note that λC itself is not a completion since it has a topsort $\Box$.

5 Restrictive use of parameters

In the extension of PTSs to C^pD^p-PTSs presented in Sections 2-4, we did not put any serious restrictions on the use of parameters:

1. If $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is a specification, then the introduction of a parametric constant c in $\lambda^{C^pD^p}(\mathcal{S})$ only requires that its intended type A is of type s, for some sort $s \in \boldsymbol{S}$. Similarly, for the introduction of a parametric definition we only require that its definiens a is of a certain type A. By correctness of types, either $A \equiv s$, or A has type s, for some $s \in \boldsymbol{S}$;
2. Similarly, if $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$, or $\Gamma \equiv \Gamma_1, c(\Delta){:}A, \Gamma_2$, the only restrictions we put on Δ are that Δ must contain only variable declarations, and that Γ_1, Δ must be legal. There are no additional restrictions on the types B_i of the declarations $x_i{:}B_i$ in Δ.

Something similar is the case with Π-formation rules in a (parameter-free) PTS in which there is no restriction on the use of Π-formation rules: $(s_1, s_2, s_3) \in \boldsymbol{R}$ for any $s_1, s_2, s_3 \in \boldsymbol{S}$. In the specific situation that $\boldsymbol{S} = \{(*, \Box)\}$ and $\boldsymbol{A} = \{*{:}\Box\}$, this would give a non-functional PTS even stronger than λC, the system on top of the Barendregt Cube. λC and the other systems of the Barendregt Cube cannot be constructed if we do not put restrictions on the rules that are allowed. It is the variation in the set of Π-formation rules that makes it possible to distinguish the various type systems in the Cube (and the various logical systems that are behind it, via the PAT-isomorphism).

In this section we study C^pD^p-PTSs in which we put restrictions on the types of parametric constants and definitions, and their parameters. These restrictions can be described in a set $\boldsymbol{P}$ of parametric rules, just as the restrictions on Π-formation rules are described in $\boldsymbol{R}$. The effect of the rules in $\boldsymbol{P}$ is as follows.

- Assume we have a constant declaration $c(\Delta) : A$ that is part of a legal context Γ. By Correctness of Contexts, A has type s for some $s \in \boldsymbol{S}$. Similarly, for each declaration $x_i{:}B_i$ in Δ there is a sort s_i such that B_i has type s_i. The use of parameters is restricted by demanding that $(s_i, s) \in \boldsymbol{P}$ for $i = 1, \ldots, n$;
- In principle, the same holds for a definition declaration $c(\Delta){=}a{:}A$. However, there is a small difference on this point. It is not necessary that A has type s for some sort $s \in \boldsymbol{S}$: it can be the case that $A \equiv s$ and that $s : s'$ does not hold for any $s' \in \boldsymbol{S}$. This is a feature that occurs in the DPTSs of Severi and Poll. To keep our system compatible with the DPTSs, we want to maintain this feature.
  To cover this case, we do not only introduce rules of the form (s_i, s), but also rules of the form (s_i, TOP). If the use of parameters is restricted by a set $\boldsymbol{P}$, then either $(s_i, s) \in \boldsymbol{P}$ for $i = 1, \ldots, n$, or A is a topsort, and $(s_i, \text{TOP}) \in \boldsymbol{P}$ for $i = 1, \ldots, n$.

In the specific case of the Barendregt Cube, the combination of $\boldsymbol{R}$ and $\boldsymbol{P}$ leads to a refinement of the Cube, thus making it possible to classify more type systems within one and the same framework.

The similarity of restricting the use of parameters by a set $\boldsymbol{P}$ with restricting the use of Π-formation by a set $\boldsymbol{R}$ gives us a theoretical motivation for the work in this section. But there are also some practical motivations, as several type systems can be described using restriction of parameters.

Example 84 Consider the Pascal function `double` that was presented in Section 1.

- Remark that `double` only takes object variables as parameters. In Pascal, it is not possible to have functions with type variables as parameters;
- Moreover, `double` returns an object. It is not possible in Pascal to construct functions that return a type as result.

So the use of parameters is restricted to the object level.

Other examples (ML, LF, AUTOMATH) are discussed in Section 6.

5.1 C^pD^p-PTSs with restricted parameters

We now give a formal definition of pure type systems with restricted parameters and restricted parametric definitions.

Definition 85 (Parametric Specification) A *parametric specification* is a quadruple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ such that $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is a specification (cf. Definition 2), and $\boldsymbol{P} \subseteq \boldsymbol{S} \times (\boldsymbol{S} \cup \{\text{TOP}\})$. The parametric specification is called *singly sorted* if the specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is singly sorted.

The set $\boldsymbol{P}$ enables us to present a restricted version of the C^p-weakening rule of Definition 40. We call this rule *restricted* C^p-weakening ($\hat{C}$-weak):

$$\frac{\Gamma \vdash^{\hat{C}} b : B \qquad \Gamma, \Delta_i \vdash^{\hat{C}} B_i : s_i \qquad \Gamma, \Delta \vdash^{\hat{C}} A : s}{\Gamma, c(\Delta) : A \vdash^{\hat{C}} b : B} \qquad (s_i, s) \in \boldsymbol{P}$$

The condition $(s_i, s) \in \boldsymbol{P}$ must hold for all $i \in \{1, \ldots, n\}$. But it is not necessary that all the s_i are equal: in one application of rule ($\hat{C}$-weak) it is possible to rely on more than one element of $\boldsymbol{P}$.

Definition 86 The *typing relation* $\vdash^{\hat{C}}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition 5, (C^p-app) (see Definition 40), and ($\hat{C}$-weak).

Similarly, we present a restricted version of the δ-weakening rule of Definition 42. We call this rule *restricted* D^p-weakening ($\hat{D}$-weak):

$$\frac{\Gamma \vdash^{\hat{D}} b : B \qquad \Gamma, \Delta_i \vdash^{\hat{D}} B_i : s_i \qquad \Gamma, \Delta \vdash^{\hat{D}} a : A : s}{\Gamma, c(\Delta){=}a{:}A \vdash^{\hat{D}} b : B} \qquad (s_i, s) \in \boldsymbol{P}$$

Again, $(s_i, s) \in \boldsymbol{P}$ must hold for all $i \in \{1, \ldots, n\}$, and again it is not necessary that all the s_i are equal.

For the case that A is a topsort, we present a special version of this ($\hat{D}$-weak) rule. By $A : \text{TOP}$ we denote that $A \equiv s$ and that there is no $s' \in \boldsymbol{S}$ such that $(s : s') \in \boldsymbol{A}$.

$$\frac{\Gamma \vdash^{\hat{D}} b : B \qquad \Gamma, \Delta_i \vdash^{\hat{D}} B_i : s_i \qquad \Gamma, \Delta \vdash^{\hat{D}} a : A : \text{TOP}}{\Gamma, c(\Delta){=}a{:}A \vdash^{\hat{D}} b : B} \qquad (s_i, \text{TOP}) \in \boldsymbol{P}$$

For all $i \in \{1, \ldots, n\}$, $(s_i, \text{TOP}) \in \boldsymbol{P}$ must hold, but the s_i may, again, be different.

Definition 87 The *typing relation* $\vdash^{\hat{D}}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition 5, (D^p-app), both versions of ($\hat{D}$-weak), (D^p-form), (D^p-intro), and (D^p-conv) (see Definition 42).

Definition 88 The *typing relation* $\vdash^{\hat{C}\hat{D}}$ is obtained from the relation $\vdash^{C^pD^p}$ by replacing rule (C^p-weak) by rule ($\hat{C}$-weak) and rule (D^p-weak) by rules ($\hat{D}$-weak).

Definition 89 (Pure Type Systems with Restricted Parameters and Restricted Parametric Definitions) Let $\mathcal{S}$ be a parametric specification. The pure type system with restricted parameters and restricted parametric definitions ($\hat{C}\hat{D}$-PTS) and parametric specification $\mathcal{S}$ is denoted $\lambda^{\hat{C}\hat{D}}(\mathcal{S})$. The system consists of the set of terms $\mathcal{T}_P$, the set of contexts $\mathcal{C}_P$, β-reduction, δ-reduction, and the typing relation $\vdash^{\hat{C}\hat{D}}$.

We do not extensively discuss the various meta-properties of $\hat{C}\hat{D}$-PTSs. This is because a $\hat{C}\hat{D}$-PTS with parametric specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ is a subsystem of the C^pD^p-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. We only give a stronger formulation of the extension of the Generation Lemma 72

Lemma 90 (Generation Lemma, second extension)
If $\Gamma \vdash^{\hat{C}\hat{D}} c(b_1, \ldots, b_n) : D$ *then there exist* s, Δ *and* A *such that* $\Gamma \vdash D =_{\beta\delta} A[x_i{:=}b_i]_{i=1}^n$, *and* $\Gamma \vdash b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$. *Besides we have one of these three possibilities:*

1. *Either we have that* $\Gamma = \langle \Gamma_1, c(\Delta){:}A, \Gamma_2\rangle$ *and* $\Gamma_1, \Delta \vdash^{\hat{C}\hat{D}} A : s$, *and for each* i *there is* s_i *with* $(s_i, s) \in \boldsymbol{P}$ *and* $\Gamma, \Delta_i \vdash^{\hat{C}\hat{D}} B_i : s_i$;
2. *Or we have that* $\Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle$, *and* $\Gamma_1, \Delta \vdash^{\hat{C}\hat{D}} a : A : s$, *and for each* i *there is* s_i *with* $(s_i, s) \in \boldsymbol{P}$ *and* $\Gamma, \Delta_i \vdash^{\hat{C}\hat{D}} B_i : s_i$;
3. *Or we have that* $\Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle$, *and* $\Gamma_1, \Delta \vdash^{\hat{C}\hat{D}} a : A : \mathrm{TOP}$, *and for each* i *there is* s_i *with* $(s_i, \mathrm{TOP}) \in \boldsymbol{P}$ *and* $\Gamma, \Delta_i \vdash^{\hat{C}\hat{D}} B_i : s_i$.

An important observation is the following one.

Remark 91 Our systems with restricted parameters cover the PTSs with Definitions (D-PTSs) that were introduced by Severi and Poll in [32]. Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification, and observe the parametric specification $\mathcal{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. The fact that the set of parametric rules is empty does not exclude the existence of definitions: it is still possible to apply the rules $\hat{\mathrm{D}}$-weak for $n = 0$. In that case, we obtain only definitions without parameters, and the rules of the parametric system reduce precisely to the rules of a D-PTS with specification $\mathcal{S}$.[4]

For the comparison of $\hat{\mathrm{C}}\hat{\mathrm{D}}$-PTSs with other PTSs, we introduce some terminology.

In the introduction to this paper, we argued that a parameter mechanism can be seen as a system for abstraction and application that is weaker than the λ-calculus mechanism. We will make this precise by proving (in Theorem 100) that a D-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is as powerful as any $\hat{\mathrm{C}}\hat{\mathrm{D}}$-PTS with parametric specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ for which $(s_1, s_2) \in \boldsymbol{P}$ implies $(s_1, s_2, s_2) \in \boldsymbol{R}$. We call such a $\hat{\mathrm{C}}\hat{\mathrm{D}}$-PTS *parametrically conservative*:

Definition 92 Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. $\mathcal{S}$ is *parametrically conservative* if for all $s_1, s_2 \in \boldsymbol{S}$, $(s_1, s_2) \in \boldsymbol{P}$ implies $(s_1, s_2, s_2) \in \boldsymbol{R}$.

Each $\hat{\mathrm{C}}\hat{\mathrm{D}}$-PTS can be extended to a parametrically conservative one by taking its *parametric closure*:

Definition 93 Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. We define $\mathrm{CL}(\mathcal{S})$, the *parametric closure* of $\mathcal{S}$, by $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}', \boldsymbol{P})$, where $\boldsymbol{R}' = \boldsymbol{R} \cup \{(s_1, s_2, s_2) \mid (s_1, s_2) \in \boldsymbol{P}\}$.

The Lemma below follows immediately from the definitions above.

Lemma 94 *Let* $\mathcal{S}$ *be a parametric specification. The following holds:*
1. $\mathrm{CL}(\mathcal{S})$ *is parametrically conservative; and 2.* $\mathrm{CL}(\mathrm{CL}(\mathcal{S})) = \mathrm{CL}(\mathcal{S})$. ⊠

5.2 Imitating parameters by λ-abstractions

Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. If $\mathcal{S}$ is parametrically conservative, then each parametric rule (s_1, s_2) of $\mathcal{S}$ has a corresponding Π-formation rule (s_1, s_2, s_2). In this section we show that this Π-formation rule can indeed take over the role of the parametric rule (s_1, s_2). This means that $\mathcal{S}$ has the same 'power' (see Theorem 100) as $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. With Remark 91 in mind, this even means that $\mathcal{S}$ has the same power as the D-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

In order to compare $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ with $\mathcal{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$, we need to remove the parameters from the syntax of $\lambda^{\hat{C}\hat{D}}(\mathcal{S})$. This can be easily obtained as follows:

[4] The parametric system with specification $\mathcal{S}'$ has a $\hat{\mathrm{C}}$-weakening rule while the systems of Severi and Poll do not. But the $\hat{\mathrm{C}}$-weakening rule can only be used for $n = 0$, and in that case $\hat{\mathrm{C}}$-weakening can be imitated by the normal weakening rule of PTSs: a parametric constant with zero parameters is in fact a parameter-free constant, and for such a constant one can use a variable as well.

- The parametric application in a term $c(b_1,\ldots,b_n)$ is replaced by function application $cb_1\cdots b_n$;
- A local parametric definition is translated by a parameter-free local definition, and the parameters are replaced by λ-abstractions;
- A global parametric definition is translated by a parameter-free global definition, and the parameters are replaced by λ-abstractions.

This leads to the following definitions:

Definition 95 We define the parameter-free translation $\{t\}$ of a term $t \in \mathcal{T}_P$ as follows:

$$\begin{aligned}
\{a\} &\equiv a \quad \text{if } a \equiv x \text{ or } a \equiv s;\\
\{c(b_1,\ldots,b_n)\} &\equiv c\,\{b_1\}\cdots\{b_n\};\\
\{ab\} &\equiv \{a\}\,\{b\};\\
\{\mathcal{O}x{:}A.B\} &\equiv \mathcal{O}x{:}\{A\}.\{B\} \quad \text{if } \mathcal{O} \text{ is } \lambda \text{ or } \Pi\\
\{c(\Delta){=}a{:}A \texttt{ in } b\} &\equiv c()=\{\lambda\Delta.a\}:\{\textstyle\prod\Delta.A\} \texttt{ in } \{b\}.
\end{aligned}$$

Definition 96 We extend the definition of $\{\_\}$ to contexts:

$$\begin{aligned}
\{\langle\rangle\} &\equiv \langle\rangle;\\
\{\Gamma, x{:}A\} &\equiv \{\Gamma\}, x{:}\{A\};\\
\{\Gamma, c(\Delta){:}A\} &\equiv \{\Gamma\}, c(){:}\{\textstyle\prod\Delta.A\};\\
\{\Gamma, c(\Delta){=}a{:}A\} &\equiv \{\Gamma\}, c()=\{\lambda\Delta.a\}:\{\textstyle\prod\Delta.A\}.
\end{aligned}$$

To demonstrate the behaviour of $\{\_\}$ under $\beta\delta$-reduction, we need a lemma that shows how to manipulate with substitutions and $\{\_\}$. The proof is straightforward, using induction on the structure of a.

Lemma 97 *For $a, b \in \mathcal{T}_P$: $\{a[x{:=}b]\} \equiv \{a\}[x{:=}\{b\}]$.* ⊠

The mapping $\{\_\}$ maintains β-reduction. A δ-reduction is translated into a δ-reduction followed by zero or more β-reductions. These β-reductions take over the n substitutions that are needed in a δ-reduction $c(b_1,\ldots,b_n) \to_\delta a[x_i{:=}b_i]_{i=1}^n$.

Lemma 98

1. *If $a \to_\beta a'$ then $\{a\} \twoheadrightarrow_\beta^+ \{a'\}$;*
2. *If $\Gamma \vdash a \to_\delta a'$ then there is a'' such that $\{\Gamma\} \vdash \{a\} \twoheadrightarrow_\delta^+ a'' \twoheadrightarrow_\beta \{a'\}$;*
3. *If $\Gamma \vdash a \twoheadrightarrow_{\beta\delta} a'$ then $\{\Gamma\} \vdash \{a\} \twoheadrightarrow_{\beta\delta} \{a'\}$.*

PROOF: (1) follows easily by induction on the structure of a, and Lemma 97. (3) follows from (1) and (2). We only show (2) by induction on the definition of $\Gamma \vdash a \to_\delta a'$. We treat only one case. Assume $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$ and $\Gamma \vdash c(b_1,\ldots,b_n) \to_\delta a[x_i{:=}b_i]_{i=1}^n$. Observe that $\{\Gamma\} \equiv \{\Gamma_1\}, c()=\{\lambda\Delta.a\}:\{\prod\Delta.A\}, \{\Gamma_2\}$, so

$$\{\Gamma\} \vdash c()\,\{b_1\}\cdots\{b_n\} \to_\delta \{\lambda\Delta.a\}\,\{b_1\}\cdots\{b_n\} \twoheadrightarrow_\beta \{a\}\,[x_i{:=}\{b_i\}]_{i=1}^n \overset{(97)}{\equiv} \{a[x_i{:=}b_i]_{i=1}^n\}.$$ ⊠

Remark 99 In 98.1, we cannot replace $\twoheadrightarrow_\beta^+$ by $\to_\beta$. This has to do with the definition of $\{c(\Delta){=}a{:}A \texttt{ in } b\}$. One β-reduction in Δ gives rise to (at least) two β-reductions in $c()=\{\lambda\Delta.a\}:\{\prod\Delta.A\} \texttt{ in } \{b\}$. Similarly, we cannot replace the $\twoheadrightarrow_\delta^+$ in 98.2 by $\to_\delta$.

Now we show that $\{\_\}$ embeds the $\hat{\text{C}}\hat{\text{D}}$-PTS with parametric specification $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ in the $\hat{\text{C}}\hat{\text{D}}$-PTS with parametric specification $\mathcal{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$, provided that $\mathcal{S}$ is parametrically conservative.

Theorem 100 *Let $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. Assume that $\mathcal{S}$ is parametrically conservative. Let $\mathcal{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. Then $\Gamma \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} a : A$ implies $\{\Gamma\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{a\} : \{A\}$.*

PROOF: Induction on the derivation of $\Gamma \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} a : A$. With the help of Lemma 97 and Lemma 98.3, all cases are straightforward except for the (Ĉ-weak) and (D̂-weak) rules. We only treat the (D̂-weak) rule; the proof for (Ĉ-weak) is similar. So: assume the last step of the derivation was

$$\frac{\Gamma \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} b : B \qquad \Gamma, \Delta_i \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} B_i : s_i \qquad \Gamma, \Delta \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} a : A : s}{\Gamma, c(\Delta){=}a{:}A \vdash_{\mathcal{S}}^{\hat{C}\hat{D}} b : B}(s_i, s) \in \boldsymbol{P}.$$

By the induction hypothesis, we have:

$$\{\Gamma\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{b\} : \{B\}; \tag{1}$$

$$\{\Gamma, \Delta_i\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{B_i\} : s_i; \tag{2}$$

$$\{\Gamma, \Delta\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{a\} : \{A\}; \tag{3}$$

$$\{\Gamma, \Delta\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{A\} : s. \tag{4}$$

$\mathcal{S}$ is parametrically conservative, so $(s_i, s, s) \in \boldsymbol{R}$ for $i = 1, \ldots, n$. Therefore, we can repeatedly use the Π-formation rule, starting with (4) and (2), obtaining

$$\{\Gamma\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \textstyle\prod_{i=1}^{n} x_i{:}\{B_i\}.\{A\} : s. \tag{5}$$

Notice: $\prod_{i=1}^{n} x_i{:}\{B_i\}.\{A\} \equiv \{\prod \Delta.A\}$. Repeatedly using λ-formation, (3) and (5), results in

$$\{\Gamma\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \overset{n}{\underset{i=1}{\lambda}} x_i{:}\{B_i\}.\{a\} : \{\textstyle\prod \Delta.A\}. \tag{6}$$

Similarly, $\lambda_{i=1}^{n} x_i{:}\{B_i\}.\{a\} \equiv \{\lambda\, \Delta.a\}$. Using (D̂-weak) (for the specification $\mathcal{S}'$) on (1), (2), (5) and (6) results in $\{\Gamma\}, c() = \{\lambda\, \Delta.a\} : \{\prod \Delta.A\} \vdash_{\mathcal{S}'}^{\hat{C}\hat{D}} \{b\} : \{B\}$. ⊠

Remark 101 The results in this section are presented for ĈD̂-PTSs. The same result, however, can be obtained for Ĉ-PTSs, that is: for PTSs with restricted parameters, but without definitions. We can also give an alternative formulation of Remark 91, stating that a Ĉ-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$ is in fact nothing more than a C-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

5.3 Refined Barendregt Cubes

Theorem 100 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 ĈD̂-PTS with parametric specification $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$ can be extended with any set of parametric rules without extending its logical power, as long as the parametric specification obtained remains parametrically conservative.

In this section, we will apply the insight obtained in Section 5.2 to a concrete situation: the Barendregt Cube of Section 2.1. This cube can be constructed not only for PTSs, but also for C-PTSs, D-PTSs, C^p-PTSs, D^p-PTSs, and their combinations (see Figure 2 on page 13).

With Theorem 100, we can place certain ĈD̂-PTSs in the cube of D-PTSs (and, with Remark 101 in mind, certain Ĉ-PTSs can be placed in the cube of C-PTSs). Let us, for example, have a look at the following parametric specifications (where $\boldsymbol{S} = \{*, \Box\}$ and $\boldsymbol{A} = \{(*, \Box)\}$):

$$(\boldsymbol{S}, \boldsymbol{A}, \{(*,*,*), (*,\Box,\Box)\}, \varnothing);$$
$$(\boldsymbol{S}, \boldsymbol{A}, \{(*,*,*), (*,\Box,\Box)\}, \{(*,*)\});$$
$$(\boldsymbol{S}, \boldsymbol{A}, \{(*,*,*), (*,\Box,\Box)\}, \{(*,\Box)\});$$
$$(\boldsymbol{S}, \boldsymbol{A}, \{(*,*,*), (*,\Box,\Box)\}, \{(*,*), (*,\Box)\}).$$

According to Theorem 100, the ĈD̂-PTSs with the above specifications are all equal in power, and according to Remark 91, they are all equal in power to the D-PTS with the specification of λP.

Now look at the parametric specification $\mathcal{S} = (\boldsymbol{S}, \boldsymbol{A}, \{(*,*,*)\}, \{(*,*), (*,\Box)\})$. The $\hat{\text{C}}$-PTS $\lambda^{\hat{C}}(\mathcal{S})$ is clearly stronger than the PTS $\lambda{\to}$, as in $\lambda^{\hat{C}}(\mathcal{S})$ it is possible (in a restricted way) to talk about predicates. For instance, we can have the following context:

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

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 PTS $\lambda{\to}$ without any parameter mechanism. On the other hand, $\lambda^{\hat{C}}(\mathcal{S})$ is weaker than the PTS λP: 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 (cf. the 'philosophical argument' in the introduction to this paper).

Altogether, this puts the $\hat{\text{C}}$-PTS $\lambda^{\hat{C}}(\mathcal{S})$ clearly between the PTSs $\lambda{\to}$ and λP. Similarly, the $\hat{\text{C}}\hat{\text{D}}$-PTS $\lambda^{\hat{C}\hat{D}}(\mathcal{S})$ is between the D-PTSs $\lambda{\to}$ and λP. We can illustrate this in the Barendregt Cube by putting the specification $\mathcal{S}$ in the middle of the edge that connects the systems $\lambda{\to}$ and λP.

This idea can be generalised to obtain a refinement of the Barendregt Cube. We start with the system $\lambda{\to}$. Adding an extra Π-formation rule (s_1, s_2, s_2) to $\lambda{\to}$ corresponds to moving in one dimension (to the right, upward, or backward) in the Cube. We add the possibility of moving in one dimension in the Cube, but stopping half-way the Cube, and we let this movement correspond to extending the system with the parameter rule (s_1, s_2). This 'going only half-way' is in line with Theorem 100, which says that Π-formation rule (s_1, s_2, s_2) can mimic the parameter rule (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'.

The refinement of the Barendregt Cube is depicted in Figure 3.

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 ML, LF, λ68, and λQE can be seen as systems in the Refined Barendregt Cube. This is depicted in Figure 4 on page 33, and motivated in the four subsections below.

6.1 ML

In ML (cf. [26]) one can define the polymorphic identity by (we use the notation of this paper. In ML, the types and the parameters are left implicit): $\mathtt{Id}(\alpha{:}*) = (\lambda\mathtt{x}{:}\alpha.\mathtt{x}) : (\alpha \to \alpha)$. But it is not possible to make an explicit λ-abstraction over $\alpha{:}*$. That is, the expression $\mathtt{Id} = (\lambda\alpha{:}*.\lambda\mathtt{x}{:}\alpha.\mathtt{x}) : (\Pi\alpha{:}*.\alpha \to \alpha)$ 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 elementary operations in ML as follows: $\mathtt{List}(\alpha{:}*) : *$; $\mathtt{nil}(\alpha{:}*) : \mathtt{List}(\alpha)$; $\mathtt{cons}(\alpha{:}*) : \alpha \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$,
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 $(\Box, \Box, \Box)$, 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 $\lambda{\to}$ and $\lambda\omega$.

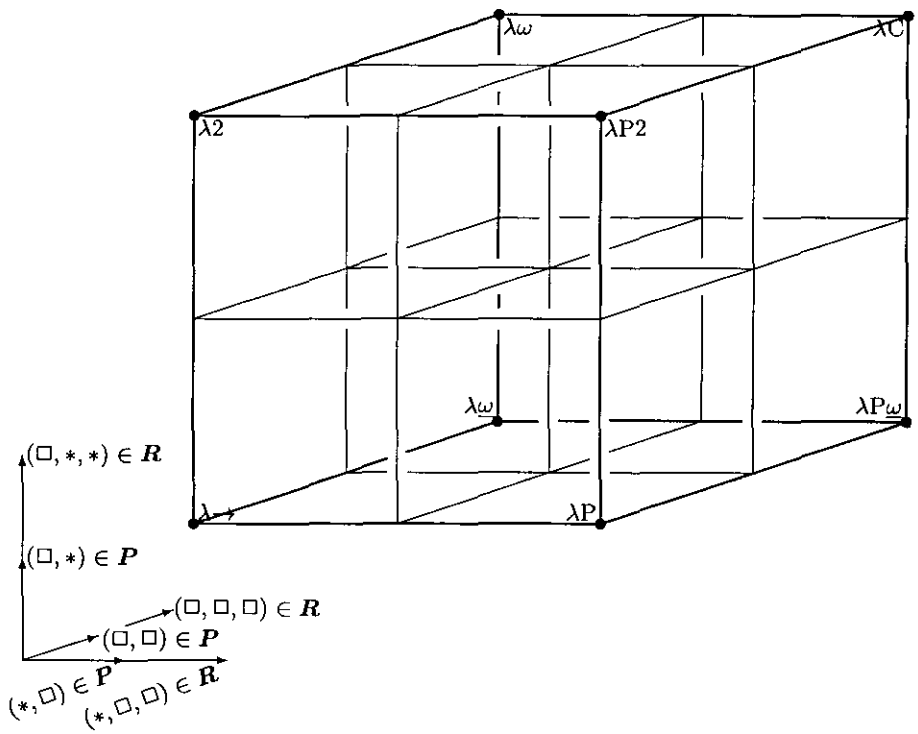


Fig. 3. The refined Barendregt Cube

6.2 LF

Geuvers [16] initially describes the system LF (see [19]) as the PTS λP. However, the use of the Π-formation rule $(*, \Box, \Box)$ is quite restrictive in most applications of LF. Geuvers splits the λ-formation rule in two rules:

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

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

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 for the bool-style implementation of PAT, a term is needed to form, given a proposition, the type of proofs of that proposition. Since the resulting term is only used when it is applied to a proposition, this means that the practical use of LF would not be restricted if we introduced the type-of-proofs-term in a parametric form, and replaced the Π-formation rule $(*, \Box, \Box)$ by a parameter rule $(*, \Box)$.

This puts (the practical applications of) LF in between the systems λ→ and λP in the Refined Barendregt Cube.

6.3 λ68 and AUT-68

The basic AUTOMATH system AUT-68 and its λ-calculus variant λ68 (see [22], section 5c for a detailed description) have a parameter mechanism and a mechanism for global parametric definitions:

- A line $(\Gamma; k; \text{PN}; \texttt{type})$ in a book is nothing more than the declaration of a parametric constant $k(\Gamma)$:$*$, and a line $(\Gamma; k; \Sigma_1; \texttt{type})$ is the declaration of a global parametric definition $k(\Gamma)=\Sigma_1$:$*$.[5] There are no demands on the context Γ, and this means that for a declaration x:$A \in \Gamma$ we can have either $A \equiv \texttt{type}$ (in PTS-terminology: $A \equiv *$, so $A : \Box$) or A:type (in PTS-terminology: $A : *$). We conclude that AUT-68 has the parameter rules $(*, \Box)$ and $(\Box, \Box)$;
- Similarly, lines of the form $(\Gamma; k; \text{PN}; \Sigma_2)$ and $(\Gamma; k; \Sigma_1; \Sigma_2)$, where Σ_2:type, represent parametric constants and global parametric definitions that are constructed using the parameter rules $(*, *)$ and $(\Box, *)$.

Moreover, AUT-68 has a λ-calculus mechanism with as only Π-formation rule $(*, *, *)$.

This suggests that AUT-68 can be represented by a $\hat{\text{C}}\hat{\text{D}}$-PTS with specification

$$\mathcal{S}_{68} = (\boldsymbol{S}, \boldsymbol{A}, \{(*, *, *)\}, \boldsymbol{S} \times \boldsymbol{S})$$

where $\boldsymbol{S} = \{*, \Box\}$ and $\boldsymbol{A} = \{(*, \Box)\}$. This system can be found in the exact middle of the refined Barendregt Cube.

As for the structure of abstraction and application, this gives a good description of AUT-68. The position of AUT-68 in the Refined Barendregt Cube gives a far better idea of the force of AUT-68 than, for instance, the description of AUT-68 in [3], where it cannot be clearly positioned in the Barendregt Cube. Another advantage is that $\lambda^{\hat{C}\hat{D}}(\mathcal{S}_{68})$ has parameters. Thus, it is closer to the original system AUT-68 than the system that was described in [3]. On the other hand, we should not say that AUT-68 is exactly the system $\lambda^{\hat{C}\hat{D}}(\mathcal{S}_{68})$. There are several differences:

- DPTSs have both global and local definitions. AUTOMATH has only global definitions;
- In DPTSs, the type B of a definition x=T:B does not have to be typable itself (B can be a topsort). In AUTOMATH, B has to be typable;
- The δ-reduction of DPTSs is not substitutive; δ-reduction of AUTOMATH is substitutive.

6.4 λQE and AUT-QE

In the more sophisticated AUTOMATH system AUT-QE and its λ-calculus variant λQE we have a Π-formation rule $(*, \Box, \Box)$ additionally to the rules of λ68. This means that the applicational and abstractional behaviour can be described by the $\hat{\text{C}}\hat{\text{D}}$-PTS with Π-formation rules $(*, *, *)$ and $(*, \Box, \Box)$, and parametric rules (s_1, s_2) for $s_1, s_2 \in \boldsymbol{S}$. This system is located in the middle of the right side of the Refined Barendregt Cube, exactly in between λC and λP.

6.5 PAL

The AUTOMATH languages are all based on two concepts: typed λ-calculus and a combined 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), 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 $\hat{\text{C}}\hat{\text{D}}$-PTS with the following parametric specification:
$\boldsymbol{S} = \{*, \Box\}$; $\boldsymbol{A} = \{(*, \Box)\}$; $\boldsymbol{R} = \varnothing$; $\boldsymbol{P} = \{(*, *), (*, \Box), (\Box, *), (\Box, \Box)\}$.

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

[5] The latter corresponds to the AUTOMATH line $\Gamma \vdash k(x_1, \ldots, x_n) = \Sigma_1 : *$, as discussed in Section 1.3.

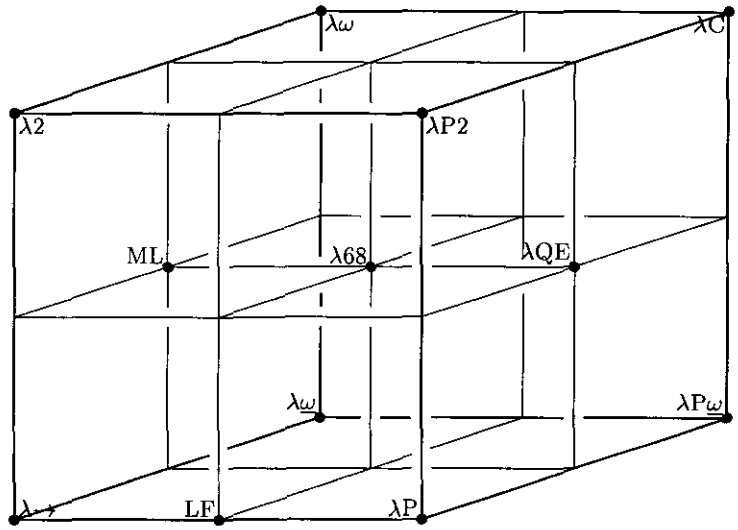


Fig. 4. LF, ML, λ68, and λQE in the refined Barendregt Cube

7 First-order predicate logic

A standard way to code first-order predicate logic in PAT-style (Curry-Howard variant) uses a type system that looks familiar to λP. It is due to Berardi, and presented in Definition 5.4.5 of [3].

In order to keep objects and object types separated from proofs and propositions, the sorts $*$ and $\Box$ of λP are replaced by $*_s, *_p, *_f, \Box_s$ and $\Box_p$. Here, $*_s$ and $\Box_s$ handle the objects and object types, whilst $\Box_p, *_p$ are used for propositions and their proofs. The sort $*_f$ is used to store the types of the function symbols of the first-order language. For the construction of logical implication and universal quantification, the Π-formation rules $(*_p, *_p, *_p)$ and $(*_s, *_p, *_p)$ are used. The Π-formation rule $(*_s, *_s, *_f)$ allows the formation of a function space between object types, and the Π-formation rule $(*_s, *_f, *_f)$ makes it possible to form functions of several arguments between object types. There is no sort $\Box_f$, as free variables for function spaces are not allowed. The construction of relation symbols requires Π-formation rule $(*_s, \Box_p, \Box_p)$.

Thus, we find a PTS (or a D-PTS) with the following specification:

$$\boldsymbol{S} = \{*_s, *_p, *_f, \Box_s, \Box_p\};$$
$$\boldsymbol{A} = \{(*_s, \Box_s), (*_p, \Box_p)\};$$
$$\boldsymbol{R} = \{(*_s, *_s, *_f), (*_s, *_f, *_f), (*_s, *_p, *_p), (*_p, *_p, *_p), (*_s, \Box_p, \Box_p)\}.$$

Due to the Π-formation rule $(*_s, \Box_p, \Box_p)$ in the PTS-representation of first-order logic, there are types that are not in β-normal form:

Example 102 For a term $A : *_s$ we can form the type $\Pi x{:}A.*_p$. If b is a term of type $*_p$ in which a variable $x{:}A$ may occur free, we can form $\lambda x{:}A.b$ of type $\Pi x{:}A.*_p$. Applying this term to a term a of type A results in $(\lambda x{:}A.b)a$ of type $*_p$. This term is a type (because it has type $*_p$) and is not in β-normal form.

If a PTS has types that are not in β-normal form, it is possible that there are applications of the conversion rule

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

in a deduction in such a PTS. The conversion rule has as a disadvantage that its implementation in computer systems makes the system slow. This is because it may be very time-consuming (or memory-consuming) to establish whether two λ-terms are β-equal or not. Hence, it would be useful to have a type system in which all types are in β-normal form.

In the formulation of first-order predicate logic above, it is only the rule $(*_s, \Box_p, \Box_p)$ which allows to form types that are not in β-normal form. We show this as follows. Assume, $\Gamma \vdash P : s$, P is not in β-normal form, and all the subterms P' of P that are a type are in β-normal form. Then P cannot be a sort or a variable. As P has type s, P cannot be of the form $\lambda x{:}P_1.P_2$, either. If $P \equiv \Pi x{:}P_1.P_2$ then either P_1 or P_2 are not in β-normal form. As P_1 and P_2 are both types, this does not occur. So P must be an application term P_1P_2. By the Generation Lemma for PTSs, there is a type A and a sort s such that $\Gamma \vdash P_1 : (\Pi x{:}A.s)$. By Correctness of Types, there is a sort s' such that $\Gamma \vdash (\Pi x{:}A.s) : s'$. By the Generation Lemma, there is $(s_1, s_2, s') \in \boldsymbol{R}$ such that $\Gamma \vdash A : s_1$ and $\Gamma, x{:}A \vdash s : s_2$. This means that (s, s_2) is an axiom, and therefore $s_2 \in \{\Box_s, \Box_p\}$. Hence, $(s_1, s_2, s_3) = (*_s, \Box_p, \Box_p)$.

We conclude that implementations of first-order predicate logic in type theory would be more efficient if it were possible to avoid rule $(*_s, \Box_p, \Box_p)$. With the use of parameters, it is easy to avoid that rule. This is because rule $(*_s, \Box_p, \Box_p)$ is only necessary to type the relation symbols of the first-order language. And as relation symbols in a first-order language are always introduced with parameters, it is no restriction to introduce them in the type system in a parametrised way. This can be done with parameter-rule $(*_s, \Box_p)$: if we want to introduce a n-ary relation symbol R with arguments of type $U_1, \ldots, U_n$ (where the U_i's are of type $*_s$), we apply C^p-weakening (let $\Delta \equiv x_1{:}U_1, \ldots, x_n{:}U_n$ and $\Delta_i \equiv x_1{:}U_1, \ldots, x_{i-1}{:}U_{i-1}$):

$$\frac{\Gamma \vdash b{:}B \qquad \Gamma, \Delta_i \vdash U_i{:}\, *_s \qquad \Gamma, \Delta \vdash *_p{:}\Box_p}{\Gamma, R(\Delta) : *_p \vdash b : B}.$$

This involves the use of the parameter-rule $(*_s, \Box_p)$.

Hence, replacing rule $(*_s, \Box_p, \Box_p)$ by parameter-rule $(*_s, \Box_p)$ enables one to remove the conversion rule in the type-theoretic representation of first-order predicate logic, making it more efficient (see the forthcoming Theorem 105). It is reasonable to replace even more rules by parameter-rules in the case of first-order predicate logic, as we presently explain.

Function symbols in a first-order language are also of a parametric nature. The sort $*_f$, the Π-formation rules $(*_s, *_s, *_f)$ and $(*_s, *_f, *_f)$ are only used to construct the types of these function symbols. We can introduce these function symbols in a more realistic way by using the parametric rule $(*_s, *_s)$ instead of the Π-formation rules $(*_s, *_s, *_f)$ and $(*_s, *_f, *_f)$:

$$\frac{\Gamma \vdash b{:}B \qquad \Gamma, \Delta_i \vdash U_i{:}\, *_s \qquad \Gamma, \Delta \vdash U{:}*_s}{\Gamma, f(\Delta) : U \vdash b : B}.$$

We have now obtained a $\hat{\mathrm{C}}$-PTS with parametric specification $\mathcal{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}', \boldsymbol{P}')$, where:
$\boldsymbol{S}' = \{*_s, *_p, \Box_s, \Box_p\}$; $\boldsymbol{A}' = \{(*_s, \Box_s), (*_p, \Box_p)\}$;
$\boldsymbol{R}' = \{(*_s, *_p, *_p), (*_p, *_p, *_p)\}$; $\boldsymbol{P}' = \{(*_s, *_s), (*_s, \Box_p)\}$.

We now prove that types in this $\hat{\mathrm{C}}$-PTS are always in β-normal form. For the proof we need as a lemma that any object term (that is: a term P such that there is Q with $P : Q : *_s$) is in β-normal form.

Lemma 103 *If $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} P : Q : *_s$ then P is in β-normal form.*

PROOF: Induction on the structure of P.

- The cases $P \in \mathcal{V}$ and $P \in \boldsymbol{S}'$ are trivial;
- If $P \equiv c(b_1, \ldots, b_n)$ then we use the second extension of the Generation Lemma, 90, to find $B_1, \ldots, B_n, B$ and $s_1, \ldots, s_n, s$ such that $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ and $\Gamma, x_1{:}B_1, \ldots x_{i-1}{:}B_{i-1} \vdash^{\hat{C}}_{\mathcal{S}'} B_i{:}s_i$, and $(s_i, s) \in \boldsymbol{P}'$. Due to the definition of $\boldsymbol{P}'$, $s_i \equiv *_s$ for all i. By the Substitution Lemma, $\Gamma \vdash B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s$, and therefore $\Gamma \vdash b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s$. By the induction hypothesis, the b_i are in β-normal form. Therefore, $c(b_1, \ldots, b_n)$ is in β-normal form;
- If $P \equiv P_1P_2$ then there are (Generation Lemma) R_1, R_2 such that $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} P_1 : \Pi x{:}R_1.R_2$, and $Q =_\beta R_2[x{:=}P_2]$. By Correctness of Types there is $s \in \boldsymbol{S}'$ such that $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} (\Pi x{:}R_1.R_2) : s$. By the Generation Lemma and the definition of $\boldsymbol{R}'$, $\Gamma, x{:}R_1 \vdash^{\hat{C}}_{\mathcal{S}'} R_2{:}*_p$. By the Substitution

Lemma, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} R_2[x{:=}P_2]{:}*_p$. Let Q' be a common β-reduct of Q and $R_2[x{:=}P_2]$. By Subject Reduction, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} Q' : *_s$ and $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} Q' : *_p$, which contradicts Unicity of Types. We conclude that the case $P \equiv P_1P_2$ does not occur;
- If $P \equiv \lambda x{:}P_1.P_2$ then there are R_1, R_2 such that $Q =_\beta \Pi x{:}R_1.R_2$. Let Q' be a common β-reduct of Q and $\Pi x{:}R_1.R_2$. There are R_1', R_2' such that $Q' \equiv \Pi x{:}R_1'.R_2'$. By Subject Reduction, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} \Pi x{:}R_1'.R_2' : *_s$. By the Generation Lemma, there are s_1, s_2 such that $(s_1, s_2, *_s) \in \boldsymbol{R}'$. This is not the case. So the case $P \equiv \lambda x{:}P_1.P_2$ does not occur;
- If $P \equiv \Pi x{:}P_1.P_2$ then there is s such that $Q \equiv s$. By the Generation Lemma, this would mean that $s : *_s$ is an axiom, which is not the case. So the case $P \equiv \Pi x{:}P_1.P_2$ does not occur.

⊠

Remark 104 The proof of this lemma not only shows that a P for which $P : Q : *_s$ is always in normal form. It also shows that P can only be a variable or an expression of the form $c(b_1, \ldots, b_n)$ such that there are $B_1, \ldots, B_n$ with $b_i : B_i : *_s$. This corresponds exactly to the definition of terms in first-order logic. We conclude that our specification $\mathcal{S}'$ results in an exact description of the terms of first-order logic.

Theorem 105 *Assume $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} P : s$. Then P is in β-normal form.*

PROOF: Induction on the structure of P.
- The cases $P \in \mathcal{V}$ and $P \in \boldsymbol{S}'$ are trivial;
- $P \equiv c(b_1, \ldots, b_n)$. By the second extension of the Generation Lemma 90, there are sorts $s_1, \ldots, s_n$ and terms $B_1, \ldots, B_n$ such that
  $(s_i, s) \in \boldsymbol{P}'$, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$ and $\Gamma, x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1} \vdash^{\hat{C}}_{\mathcal{S}'} B_i{:}s_i$.
  By the definition of $\boldsymbol{P}'$, $s_i \equiv *_s$ for all i. By the Substitution Lemma, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s$. By Lemma 103, the b_i are in β-normal form. Therefore $c(b_1, \ldots, b_n)$ is in β-normal form;
- $P \equiv P_1P_2$. By the Generation Lemma, there are R_1, R_2 such that $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} P_1 : \Pi x{:}R_1.R_2$ and $s =_\beta R_2[x{:=}P_2]$. By Correctness of Types, the Generation Lemma and the definition of $\boldsymbol{R}'$, $\Gamma, x{:}R_1 \vdash^{\hat{C}}_{\mathcal{S}'} R_2 : *_p$. By the Substitution Lemma, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} R_2[x{:=}P_2] : *_p$. By Subject Reduction, $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} s : *_p$. This means that $(s, *_p)$ is an axiom, which is not the case. We conclude that the case $P \equiv P_1P_2$ does not occur;
- $P \equiv \lambda x{:}P_1.P_2$. By the Generation Lemma, $s =_\beta \Pi x{:}R_1.R_2$ for some R_1, R_2. This is impossible. We conclude that the case $P \equiv \lambda x{:}P_1.P_2$ does not occur;
- $P \equiv \Pi x{:}P_1.P_2$. By the Generation Lemma, there are s_1, s_2 such that $\Gamma \vdash^{\hat{C}}_{\mathcal{S}'} P_1 : s_1$ and $\Gamma, x{:}P_1 \vdash^{\hat{C}}_{\mathcal{S}'} P_2 : s_2$. By the induction hypothesis, P_1 and P_2 are in β-normal form. So P is in β-normal form.

⊠

We conclude that replacing the Π-formation rules $(*_s, *_s, *_f)$, $(*_s, *_f, *_f)$ and $(*_s, \Box_p, \Box_p)$ by parametric rules $(*_s, *_s)$ and $(*_s, \Box_p)$ makes the implementations of first-order languages in type theory

- easier to implement, as the conversion rule becomes superfluous;
- more realistic; it gives, for example, an exact description of the terms in first-order logic, something that cannot be done in the parameter-free PTS proposed by Berardi.

8 Conclusions: Yet another extension of PTSs?

Since PTSs have been introduced, many extensions have been proposed (see [4] for a non-exhaustive list). The reader may wonder why we propose yet another extension of PTSs, and whether it is more interesting than those other extensions or not. Here we answer to these questions.

- Our extension is compatible with (and can be seen as an extension of) the extension of PTSs with definitions as proposed by Poll and Severi, which is considered to be a standard way to introduce definitions in PTSs. In fact, allowing only parametric constants with zero parameters results in the D-PTSs of [32];
- Parameters and parametric definitions occur in many implementations of type systems and programming languages. The Pascal-function `double` given at the beginning of this paper can be described in our formalism by the context declaration `double(z:Int)=z+z:Int`;
- The AUTOMATH systems, which form the basis for most modern proof checkers that are based on type theory, can be described in our system. A description of AUTOMATH that includes parameters does justice to that system and places it in a more general framework, so that it can more easily be compared with other type systems (see Figure 4 on page 33);
- Modern type systems, like LF and ML, have already been described as one of the systems of the Barendregt Cube (Figure 1 on page 8). In Section 6 we showed that a more detailed description can be given in the refined Barendregt Cube of Figure 4;
- As argued in Section 7, parameters are useful when describing first-order logic in type theory. Compared to the traditional PTS-representation (systems related to λP of the Barendregt Cube) of first-order logic, parametric representations are
  - easier to implement (because the conversion rule is not needed);
  - closer to the first-order language and therefore closer to the intuition;
- As argued in the beginning of this paper, parameters make it possible to distinguish the attitude of users and developers of a system. Often, the user only needs a (partially) parametrised version of the system, whilst the developer wants to have the possibilities of full λ-abstractions.

Future work

There are several issues concerning parametric type systems that deserve to be studied in the future:

- The meta-theoretical properties may have easier proofs than the ones presented in this paper. In particular, the proof of strong normalisation for a parametric type system is based on strong normalisation for a PTS that may have more Π-formation rules. It would be interesting to know whether (and to what extent) these rather strong demands can be weakened;
- There may be a relation between the parameter mechanism of this paper and AUTOMATH, and the use of parameters in the representation of higher order propositional functions in the ramified theory of types of Russell and Whitehead.

References

1. S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science, Volume 2: Background: Computational Structures.* Oxford University Press, 1992.
2. H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics **103**. North-Holland, Amsterdam, revised edition, 1984.
3. H.P. Barendregt. Lambda calculi with types. In [1], pages 117–309. Oxford University Press, 1992.
4. G. Barthe. Extensions of pure type systems. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, pages 16–31, Edinburgh, 1995. Springer Verlag, Heidelberg.
5. L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* PhD thesis, Eindhoven University of Technology, 1977. Published as Mathematical Centre Tracts nr. 83 (Amsterdam, Mathematisch Centrum, 1979).
6. L.S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.
7. S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

8. R. Bloo, F. Kamareddine, and R. P. Nederpelt. The Barendregt Cube with Definitions and Generalised Reduction. *Information and Computation*, 126 (2):123–143, 1996.
9. V.A.J. Borghuis. *Coming to Terms with Modal Logic: On the interpretation of modalities in typed λ-calculus.* PhD thesis, Technische Universiteit Eindhoven, 1994.
10. N.G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, IRIA, Versailles, 1968. Springer Verlag, Berlin, 1970. Lecture Notes in Mathematics **125**; also in [28], pages 73–100.
11. N.G. de Bruijn. Reflections on Automath. Eindhoven University of Technology, 1990. Also in [28], pages 201–228.
12. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.
13. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
14. D.T. van Daalen. A description of Automath and some aspects of its language theory. In P. Braffort, editor, *Proceedings of the Symposium APLASM*, volume I, pages 48–77, 1973. Also in [28], pages 101–126.
15. D.T. van Daalen. *The Language Theory of Automath.* PhD thesis, Eindhoven University of Technology, 1980.
16. J.H. Geuvers. *Logics and Type Systems.* PhD thesis, Catholic University of Nijmegen, 1993.
17. J.H. Geuvers and M.J. Nederhof. A modular proof of strong normalization for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.
18. J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.
19. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings Second Symposium on Logic in Computer Science*, pages 194–204, Washington D.C., 1987. IEEE.
20. J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.
21. F. Kamareddine and R. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.
22. T. Laan. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Eindhoven University of Technology, 1997.
23. Twan Laan and Michael Franssen. Parameters for first order logic. *Logic and Computation*, 2001.
24. G. Longo and E. Moggi. Constructive natural deduction and its modest interpretation. Technical Report CMU-CS-88-131, Carnegie Mellon University, Pittsburgh, USA, 1988.
25. Z. Luo. A problem of adequacy: conservativity of calculus of constructions over higher-order logic. Technical Report ECS-LFCS-90-121, University of Edinburgh, 1990.
26. R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML.* MIT Press, Cambridge (Massachusetts)/London, 1990.
27. R.P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types.* PhD thesis, Eindhoven University of Technology, 1973. Also in [28], pages 389–468.
28. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath.* Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam, 1994.
29. The PVS specification and verification system. *SRI International's Computer Science Laboratory*, see `http://pvs.csl.sri.com/`.
30. G.R. Renardel de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Information and Computation*, 99:154–177, 1991.
31. J.C. Reynolds. *Towards a theory of type structure*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.
32. P. Severi and E. Poll. Pure type systems with definitions. In A. Nerode and Yu.V. Matiyasevich, editors, *Proceedings of LFCS'94 (LNCS* **813***)*, pages 316–328, New York, 1994. LFCS'94, St. Petersburg, Russia, Springer Verlag.
33. J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.
34. R. de Vrijer. A direct proof of the finite developments theorem. *The Journal of Symbolic Logic*, 50(2):339–343, 1985.
35. A.N. Whitehead and B. Russell. *Principia Mathematica*, volume I, II, III. Cambridge University Press, 1910[1], 1927[2]. All references are to the first volume, unless otherwise stated.

A Proofs

PROOF OF LEMMA 56: Induction on the definition of $|b|_\Gamma$. We treat the two most interesting cases (at (IH) we use the induction hypothesis):

- $b \equiv c(b_1, \ldots, b_n)$ and $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \rangle$.

$$\begin{aligned} Dom(|b|_\Gamma) &= Dom(|a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n) \\ &\subseteq \Big(Dom(|a|_{\Gamma_1,\Delta}) \setminus \{x_1, \ldots, x_n\}\Big) \cup \bigcup_{i=1}^n Dom(|b_i|_\Gamma) \\ &\overset{(IH)}{\subseteq} (Dom(\Gamma_1, \Delta) \setminus Def(\Gamma_1)) \setminus \{x_1, \ldots, x_n\} \cup (Dom(\Gamma) \setminus Def(\Gamma)) \\ &= Dom(\Gamma) \setminus Def(\Gamma). \end{aligned}$$

  We can use the induction hypothesis at (IH) because Γ is sound, and therefore $Dom(a) \subseteq Dom(\Gamma_1, \Delta)$;

- If $b \equiv c(\Delta){=}a{:}A$ `in` b' then $Dom(|b|_\Gamma) = Dom(|b'|_{\Gamma, c(\Delta){=}a:A})$

$$\overset{(IH)}{\subseteq} Dom(\Gamma, c(\Delta){=}a{:}A) \setminus Def(\Gamma, c(\Delta){=}a{:}A) = Dom(\Gamma) \setminus Def(\Gamma).$$

⊠

PROOF OF LEMMA 57: Induction on the total number of symbols occurring in Γ and d. We treat two cases:

- $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \rangle$ and $d \equiv c(b_1, \ldots, b_n)$. Notice that $\Gamma \vdash d \to_\delta a[x_i{:=}b_i]_{i=1}^n$.
  By induction, $\Gamma_1, \Delta \vdash a \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}$, so $\Gamma \vdash a \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}$ (Lemma 45), so by Lemma 50,

$$\Gamma[x_i{:=}b_i]_{i=1}^n \vdash a[x_i{:=}b_i]_{i=1}^n \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}[x_i{:=}b_i]_{i=1}^n.$$

  As the x_i are bound in $c(\Delta){=}a{:}A$, they do not occur free in Γ, so $\Gamma[x_i{:=}b_i]_{i=1}^n \equiv \Gamma$. Therefore $\Gamma \vdash a[x_i{:=}b_i]_{i=1}^n \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}[x_i{:=}b_i]_{i=1}^n \overset{(IH,\,51)}{\twoheadrightarrow_\delta} |a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n \equiv |c(b_1, \ldots, b_n)|_\Gamma$;
- $d \equiv c(\Delta){=}a{:}A$ `in` b. By the induction hypothesis, $\Gamma, c(\Delta){=}a{:}A \vdash b \twoheadrightarrow_\delta |b|_{\Gamma, c(\Delta){=}a:A}$
  hence $\Gamma \vdash c(\Delta){=}a{:}A$ `in` $b \twoheadrightarrow_\delta c(\Delta){=}a{:}A$ `in` $|b|_{\Gamma, c(\Delta){=}a:A}$.
  Now $Dom(d) \subseteq Dom(\Gamma)$, so $Dom(b) \subseteq Dom(\Gamma, c(\Delta){=}a{:}A)$, so by Lemma 56,
  $Dom(|b|_{\Gamma, c(\Delta){=}a:A}) \subseteq Dom(\Gamma, c(\Delta){=}a{:}A) \setminus Def(\Gamma, c(\Delta){=}a{:}A)$,
  so $c \notin Cons(|b|_{\Gamma, c(\Delta){=}a:A})$, so $\Gamma \vdash d \twoheadrightarrow_\delta c(\Delta){=}a{:}A$ `in` $|b|_{\Gamma, c(\Delta){=}a:A} \to_\delta |b|_{\Gamma, c(\Delta){=}a:A}$.

⊠

PROOF OF LEMMA 59: Induction on the definition of $|b|_{\Gamma_1,\Gamma_3}$. We consider only a few non-trivial cases:

- $b \equiv c(b_1, \ldots, b_n)$ and $\Gamma_3 \equiv \langle \Gamma_{31}, c(\Delta){=}a{:}A, \Gamma_{32} \rangle$.

$$\begin{aligned} |c(b_1, \ldots, b_n)|_{\Gamma_1,\Gamma_3} &\equiv |a|_{\Gamma_1,\Gamma_{31},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_3}] \\ &\overset{(IH,51)}{\equiv} |a|_{\Gamma_1,\Gamma_2,\Gamma_{31},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_2,\Gamma_3}] \equiv |c(b_1, \ldots, b_n)|_{\Gamma_1,\Gamma_2,\Gamma_3}; \end{aligned}$$

- $b \equiv c(\Delta){=}a{:}A$ `in` b. Notice that

$$\begin{aligned} |c(\Delta){=}a{:}A \text{ in } b|_{\Gamma_1,\Gamma_3} &\equiv |b|_{\Gamma_1,\Gamma_3,c(\Delta){=}a:A} \\ &\overset{(IH)}{\equiv} |b|_{\Gamma_1,\Gamma_2,\Gamma_3,c(\Delta){=}a:A} \equiv |c(\Delta){=}a{:}A \text{ in } b|_{\Gamma_1,\Gamma_2,\Gamma_3}. \end{aligned}$$

⊠

PROOF OF LEMMA 60: Induction on the definition of $|a|_{\Gamma_1,\Gamma_2}$. We treat only a few non-trivial cases:

- $a \equiv c(b_1, \ldots, b_n)$ and $\Gamma_2 \equiv \Gamma_{21}, c(\Delta){=}c'{:}C, \Gamma_{22}$.

$$\begin{aligned} &|a|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}] \\ &\equiv |c'|_{\Gamma_1,\Gamma_{21},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_2}]_{i=1}^n[x{:=}|b|_{\Gamma_1}] \\ &\overset{(48)}{\equiv} |c'|_{\Gamma_1,\Gamma_{21},\Delta}[x{:=}|b|_{\Gamma_1}][x_i{:=}|b_i|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}]]_{i=1}^n \\ &\overset{(IH)}{\equiv} |c'[x{:=}b]|_{\Gamma_1,\Gamma_{21}[x:=b],\Delta[x:=b]}[x_i{:=}|b_i[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}]_{i=1}^n \\ &\equiv |c(b_1, \ldots, b_n)[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}; \end{aligned}$$

– $a \equiv c(\Delta){=}c'{:}C \texttt{ in } d$.

$$\begin{aligned} |a|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}] &\equiv |d|_{\Gamma_1,\Gamma_2,c(\Delta)=c':C}[x{:=}|b|_{\Gamma_1}] \\ &\overset{\text{(IH)}}{\equiv} |d[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b],c(\Delta[x:=b])=c'[x:=b]:C[x:=b]} \\ &\equiv |(c(\Delta){=}c'{:}C \texttt{ in } d)[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}. \end{aligned}$$

⊠

PROOF OF LEMMA 78: Using induction on the definition of $\Gamma \vdash a \to_\delta b$, we simultaneously prove: 1. If $\Gamma \vdash a \to_\delta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta \|b\|_\Gamma$; and 2. If $\Gamma \to_\delta \Gamma'$ then $\|a\|_\Gamma \twoheadrightarrow_\beta \|a\|_{\Gamma'}$.
We only treat two non-trivial cases.

– $\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash c(b_1,\ldots,b_n) \to_\delta a[x_i{:=}b_i]_{i=1}^n$. Observe:

$$\begin{aligned} \|c(b_1,\ldots,b_n)\|_\Gamma &\equiv \|\lambda\,\Delta.a\|_{\Gamma_1}\,\|b_1\|_\Gamma \cdots \|b_n\|_\Gamma \\ &\equiv \left(\underset{i=1}{\overset{n}{\lambda}} x_i{:}\|B_i\|_{\Gamma_1,\Delta_i}.\|a\|_{\Gamma,\Delta}\right)\|b_1\|_\Gamma \cdots \|b_n\|_\Gamma \\ &\twoheadrightarrow_\beta^n \|a\|_{\Gamma_1,\Delta}\,[x_i{:=}\|b_i\|_\Gamma]_{i=1}^n \overset{(76)}{\equiv} \|a\|_{\Gamma_1}\,[x_i{:=}\|b_i\|_\Gamma]_{i=1}^n \\ &\overset{(76)}{\equiv} \|a\|_\Gamma\,[x_i{:=}\|b_i\|_\Gamma]_{i=1}^n \overset{(77)}{\equiv} \|a[x_i{:=}b_i]_{i=1}^n\|_\Gamma\,; \end{aligned}$$

– $\Gamma \vdash c(\Delta){=}a{:}A \texttt{ in } b \to_\delta b$ because $c \notin \mathit{Cons}(b)$. Then $c \notin \mathit{FV}(\|b\|_\Gamma)$. Hence

$$\begin{aligned} \|c(\Delta){=}a{:}A \texttt{ in } b\|_\Gamma &\equiv \left(\lambda c : \|\textstyle\prod \Delta.A\|_\Gamma . \|b\|_{\Gamma,c(\Delta)=a:A}\right)\|\lambda\,\Delta.a\|_\Gamma \\ &\to_\beta \|b\|_{\Gamma,c(\Delta)=a:A}\,[c{:=}\|\lambda\,\Delta.a\|_\Gamma] \equiv \|b\|_\Gamma\,. \end{aligned}$$

⊠

PROOF OF LEMMA 79: The following statements are proved simultaneously by induction on the structure of a: 1. If $a \to_\beta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta^+ \|b\|_\Gamma$; and 2. If $\Gamma \to_\beta \Gamma'$ then $\|a\|_\Gamma \twoheadrightarrow_\beta \|a\|_{\Gamma'}$.
(IH 1) refers to the induction hypothesis on 1, (IH 2) to the induction hypothesis on 2. We do not treat all cases, and only prove the first statement.

– $c(b_1,\ldots,b_n) \to_\beta c(b_1,\ldots,b'_j,\ldots,b_n)$, where $b_j \to_\beta b'_j$, and $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle$. We have:

$$\begin{aligned} \|c(b_1,\ldots,b_n)\|_\Gamma &\equiv \|\lambda\,\Delta.a\|_{\Gamma_1}\,\|b_1\|_\Gamma \cdots \|b_n\|_\Gamma \\ &\overset{\text{(IH 1)}}{\twoheadrightarrow_\beta^+} \|\lambda\,\Delta.a\|_{\Gamma_1}\,\|b_1\|_\Gamma \cdots \|b'_j\|_\Gamma \cdots \|b_n\|_\Gamma \equiv \|c(b_1,\ldots,b'_j,\ldots,b_n)\|_\Gamma\,; \end{aligned}$$

– $(\lambda x{:}p.q)r \to_\beta q[x{:=}r]$. Observe:

$$\begin{aligned} \|(\lambda x{:}p.q)r\|_\Gamma &\equiv \left(\lambda x{:}\|p\|_\Gamma . \|q\|_{\Gamma,x:p}\right)\|r\|_\Gamma \\ &\to_\beta \|q\|_{\Gamma,x:p}\,[x{:=}\|r\|_\Gamma] \overset{(77)}{\equiv} \|q[x{:=}r]\|_\Gamma\,; \end{aligned}$$

– $c(\Delta){=}a{:}A \texttt{ in } b \to_\beta c(\Delta'){=}a{:}A \texttt{ in } b$, where $\Delta' \equiv x_1{:}B_1,\ldots,x_j{:}B'_j,\ldots,x_n : B_n$; and $B_j \to_\beta B'_j$.
Write $C_i \equiv B_i$ if $i \neq j$ and $C_j \equiv B'_j$ and let $\Delta'_i \equiv x_1{:}C_1,\ldots,x_{i-1}{:}C_{i-1}$. Observe that

$$\begin{aligned} &\|c(\Delta){=}a{:}A \texttt{ in } b\|_\Gamma \\ &\equiv \left(\lambda c : (\|\textstyle\prod \Delta.A\|_\Gamma) . \|b\|_{\Gamma,c(\Delta)=a:A}\right)(\|\lambda\,\Delta.a\|_\Gamma) \\ &\overset{\text{(IH 1)}}{\twoheadrightarrow_\beta^+} \left(\lambda c : (\|\textstyle\prod \Delta'.A\|_\Gamma) . \|b\|_{\Gamma,c(\Delta)=a:A}\right)(\|\lambda\,\Delta'.a\|_\Gamma) \\ &\overset{\text{(IH 2)}}{\twoheadrightarrow_\beta} \left(\lambda c : (\|\textstyle\prod \Delta'.A\|_\Gamma) . \|b\|_{\Gamma,c(\Delta')=a:A}\right)(\|\lambda\,\Delta'.a\|_\Gamma) \equiv \|c(\Delta'){=}a{:}A \texttt{ in } b\|_\Gamma\,. \end{aligned}$$

⊠

PROOF OF THEOREM 82: Induction on the derivation of $\Gamma \vdash_{\mathcal{S}}^{C^p D^p} a : A$. The rules of normal PTSs do not cause any problem, and the proof for the rules for parametric constants are simplifications of the proofs for the rules for parametric definitions. We therefore only focus on the rules for parametric definitions.

- δ-application:

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

  Write $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$. If $n = 0$ then we know by induction that $\|\Gamma\| \vdash_{\mathcal{S}'} \|a\|_\Gamma : \|A\|_\Gamma$ and we are done because $\|c(b_1, \ldots, b_n)\|_\Gamma \equiv \|a\|_{\Gamma_1} \overset{(76)}{\equiv} \|a\|_\Gamma$.
  Now assume $n > 0$. As we have a derivation of $\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash^{C^p D^p} b_1{:}B_1$, we can use Correctness of Contexts to find a (shorter) derivation of $\Gamma_1, \Delta \vdash^{C^p D^p} a{:}A$. By the induction hypothesis, we have

$$\|\Gamma_1, \Delta\| \vdash_{\mathcal{S}'} \|a\|_{\Gamma_1,\Delta} : \|A\|_{\Gamma_1,\Delta'}. \tag{7}$$

  Moreover, we can use the induction hypothesis to find

$$\|\Gamma_1\|, c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1}, \|\Gamma_2\| \vdash_{\mathcal{S}'} \|b_i\|_\Gamma : \left\|B_i[x_j{:=}b_j]_{j=1}^{i-1}\right\|_\Gamma. \tag{8}$$

  We can use Correctness of Types for the PTS $\lambda\mathcal{S}'$ to find $s \in \boldsymbol{S}'$ with

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \|\textstyle\prod \Delta.A\|_{\Gamma_1} : s. \tag{9}$$

  Using rule (λ), (7) and (9) result in $\|\Gamma_1\| \vdash_{\mathcal{S}'} \|\lambda\Delta.a\|_{\Gamma_1} : \|\prod \Delta.A\|_{\Gamma_1}$. By definition of $\|\prod \Delta.A\|_{\Gamma_1}$, this means

$$\|\Gamma\| \vdash_{\mathcal{S}'} \|\lambda\Delta.a\|_{\Gamma_1} : \textstyle\prod_{i=1}^{n} x_i{:}\, \|B_i\|_{\Gamma_1,\Delta} \cdot \|A\|_{\Gamma_1,\Delta}. \tag{10}$$

  By Lemma 77, $\left\|B_i[x_j{:=}b_j]_{j=1}^{i-1}\right\|_\Gamma \equiv \|B_i\|_{\Gamma,\Delta_i}[x_j{:=}\|b_j\|_\Gamma]_{j=1}^{i-1}$. Using (8) and the application rule, we can derive from (10) that:
  $\|\Gamma\| \vdash_{\mathcal{S}'} \left(\|\lambda\Delta.a\|_{\Gamma_1} \|b_1\|_\Gamma \cdots \|b_n\|_\Gamma\right) : \left(\|A\|_{\Gamma_1,\Delta}[x_j{:=}\|b_j\|_\Gamma]_{j=1}^{n}\right)$.
  We are done because $\|c(b_1, \ldots, b_n)\|_\Gamma \equiv \|\lambda\Delta.a\|_{\Gamma_1} \|b_1\|_\Gamma \cdots \|b_n\|_\Gamma$ and
  $\|A\|_{\Gamma_1,\Delta}[x_j{:=}\|b_j\|_\Gamma]_{j=1}^{n} \overset{(76)}{\equiv} \|A\|_{\Gamma,\Delta}[x_j{:=}\|b_j\|_\Gamma]_{j=1}^{n} \overset{(77)}{\equiv} \left\|A[x_j{:=}b_j]_{j=1}^{n}\right\|_\Gamma$;

- δ-weakening: $\dfrac{\Gamma \vdash^{C^p D^p} b{:}B \qquad \Gamma, \Delta \vdash^{C^p D^p} a{:}A}{\Gamma, c(\Delta){=}a{:}A \vdash^{C^p D^p} b{:}B}$.
  By induction, $\|\Gamma, \Delta\| \vdash_{\mathcal{S}'} \|a\|_{\Gamma,\Delta} : \|A\|_{\Gamma,\Delta}$, so

$$\|\Gamma\|, x_1{:}\, \|B_1\|_{\Gamma,\Delta_1}, \ldots, x_n{:}\, \|B_n\|_{\Gamma,\Delta_n} \vdash_{\mathcal{S}'} \|a\|_{\Gamma,\Delta} : \|A\|_{\Gamma,\Delta}. \tag{11}$$

  By Correctness of Contexts for $\lambda\mathcal{S}'$, there are $s_1, \ldots, s_n \in \boldsymbol{S}'$ such that

$$\|\Gamma\|, x_1{:}\, \|B_1\|_{\Gamma,\Delta_1}, \ldots, x_{i-1}{:}\, \|B_{i-1}\|_{\Gamma,\Delta_{i-1}} \vdash_{\mathcal{S}'} \|B_i\|_{\Gamma,\Delta_i} : s_i. \tag{12}$$

  By Correctness of Types for $\lambda_{\mathcal{S}}^{C^p D^p}$, there are two possibilities:
  - There is $s \in \boldsymbol{S}$ such that $A \equiv s$. As $\mathcal{S}'$ is a completion of $\mathcal{S}$, there is $s' \in \boldsymbol{S}'$ such that $\|\Gamma, \Delta\| \vdash_{\mathcal{S}'} s : s'$.
  - There is $s' \in \boldsymbol{S}$ such that $\Gamma, \Delta \vdash^{C^p D^p} A : s'$. Then by induction, $\|\Gamma, \Delta\| \vdash \|A\|_{\Gamma,\Delta} : s'$.

In any case: we can determine $s_0' \in \boldsymbol{S}'$ such that

$$\|\Gamma\|, x_1{:}\|B_1\|_{\Gamma,\Delta_1}, \ldots, x_n{:}\|B_n\|_{\Gamma,\Delta_n} \vdash_{\mathcal{S}'} \|A\|_{\Gamma,\Delta} : s_0'. \tag{13}$$

As $\mathcal{S}'$ is quasi-full, we can subsequently determine $s_1', \ldots, s_n'$ such that $(s_i, s_{i-1}', s_i') \in \boldsymbol{R}'$ for $i = 1, \ldots, n$. This allows us to apply Π-formation n times, with as premises (12) and (14), and as conclusion: $\|\Gamma\| \vdash_{\mathcal{S}'} \prod_{i=1}^{n} x_i{:}\|B_i\|_{\Gamma,\Delta_i}.\|A\|_{\Gamma,\Delta} : s_n'$.
Notice that $\prod_{i=1}^{n} x_i{:}\|B_i\|_{\Gamma,\Delta_i}.\|A\|_{\Gamma,\Delta} \equiv \|\prod \Delta.A\|_\Gamma$. As the induction hypothesis gives us also $\|\Gamma\| \vdash_{\mathcal{S}'} \|b\|_\Gamma : \|B\|_\Gamma$, we can use the weakening rule of $\lambda\mathcal{S}'$ to obtain

$$\|\Gamma\|, c{:}\|\textstyle\prod \Delta.A\|_\Gamma \vdash_{\mathcal{S}'} \|b\|_\Gamma : \|B\|_\Gamma .$$

We are done because $\|b\|_\Gamma \equiv \|b\|_{\Gamma,c(\Delta)=a:A}$ and $\|B\|_\Gamma \equiv \|B\|_{\Gamma,c(\Delta)=a:A}$ (Lemma 76);

– δ-formation:

$$\frac{\Gamma_1, c(\Delta){=}a{:}A \vdash^{\mathcal{C}^p\mathcal{D}^p} B : s}{\Gamma_1 \vdash^{\mathcal{C}^p\mathcal{D}^p} c(\Delta){=}a{:}A \text{ in } B : s}.$$

Write $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A$. By the induction hypothesis, we have $\|\Gamma\| \vdash_{\mathcal{S}'} \|B\|_\Gamma : s$, so

$$\|\Gamma_1\|, c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} \vdash_{\mathcal{S}'} \|B\|_\Gamma : s. \tag{14}$$

By Correctness of Contexts on (14) there is $s_1 \in \boldsymbol{S}'$ such that

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \|\textstyle\prod \Delta.A\|_{\Gamma_1} : s_1. \tag{15}$$

Moreover: As $\mathcal{S}'$ is a completion of $\mathcal{S}$, there is $s_2 \in \boldsymbol{S}'$ such that $(s{:}s_2) \in \boldsymbol{A}'$. By the Start Lemma,

$$\|\Gamma_1\|, c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} \vdash_{\mathcal{S}'} s : s_2. \tag{16}$$

As $\mathcal{S}'$ is quasi-full, there is $s_3 \in \boldsymbol{S}'$ such that $(s_1, s_2, s_3) \in \boldsymbol{R}'$. Hence we can apply Π-formation:

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \Pi c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} .s : s_3. \tag{17}$$

We can now apply λ-formation on (14) and (17):

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \left(\lambda c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) : \left(\Pi c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} .s\right). \tag{18}$$

As we have a derivation of $\Gamma_1, c(\Delta){=}a{:}A \vdash^{\mathcal{C}^p\mathcal{D}^p} B : s$, we can apply Correctness of Contexts to find a (shorter) derivation of $\Gamma_1, \Delta \vdash^{\mathcal{C}^p\mathcal{D}^p} a{:}A$, so by induction: $\|\Gamma_1, \Delta\| \vdash_{\mathcal{S}'} \|a\|_{\Gamma_1,\Delta} : \|A\|_{\Gamma_1,\Delta}$. Using (15), we can repeatedly apply λ-abstraction and obtain

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \|\lambda\, \Delta.a\|_{\Gamma_1} : \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \tag{19}$$

Using (18) and the application rule, we find: $\|\Gamma_1\| \vdash_{\mathcal{S}'} \left(\lambda c{:}\|\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : s$;

– δ-introduction:

$$\frac{\Gamma_1, c(\Delta){=}a{:}A \vdash^{\mathcal{C}^p\mathcal{D}^p} b{:}B \qquad \Gamma_1 \vdash^{\mathcal{C}^p\mathcal{D}^p} c(\Delta){=}a{:}A \text{ in } B : s}{\Gamma_1 \vdash^{\mathcal{C}^p\mathcal{D}^p} (c(\Delta){=}a{:}A \text{ in } b) : (c(\Delta){=}a{:}A \text{ in } B)}.$$

In a similar way as in the previous case, we can find derivations of (19) and

$$\|\Gamma_1\| \vdash_{\mathcal{S}'} \left(\lambda c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|b\|_\Gamma\right) : \left(\Pi c{:}\|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right). \tag{20}$$

Using (19), (20) and the application rule, we find
$\|\Gamma_1\| \vdash_{\mathcal{S}'} \left(\lambda c{:}\|\prod \Delta.A\|_{\Gamma_1} . \|b\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : \|B\|_\Gamma \left[c{:=}\|\lambda\, \Delta.a\|_{\Gamma_1}\right]$.
By the induction hypothesis, $\|\Gamma_1\| \vdash_{\mathcal{S}'} \left(\lambda c{:}\|\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : s$, so we can apply the conversion rule to find $\|\Gamma_1\| \vdash_{\mathcal{S}'} \|c(\Delta){=}a{:}A \text{ in } b\|_{\Gamma_1} : \|c(\Delta){=}a{:}A \text{ in } B\|_{\Gamma_1}$;

– δ-conversion:

$$\frac{\Gamma \vdash^{\mathcal{C}^p\mathcal{D}^p} b : B \qquad \Gamma \vdash^{\mathcal{C}^p\mathcal{D}^p} B' : s \qquad \Gamma \vdash B =_\delta B'}{\Gamma \vdash^{\mathcal{C}^p\mathcal{D}^p} b : B'}.$$

By induction, $\|\Gamma\| \vdash_{\mathcal{S}'} \|b\|_\Gamma : \|B\|_\Gamma$ and $\|\Gamma\| \vdash_{\mathcal{S}'} \|B'\|_\Gamma : s$. By Lemma 78, $\|B\|_\Gamma =_\beta \|B'\|_\Gamma$. By Conversion, $\|\Gamma\| \vdash_{\mathcal{S}'} \|b\|_\Gamma : \|B'\|_\Gamma$.

⊠

Computing Science Reports **Department of Mathematics and Computing Science**
Eindhoven University of Technology

If you want to receive reports, send an email to: m.m.j.l.philips@tue.nl (we cannot guarantee the availability of the requested reports)

In this series appeared:

97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.
97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209
98/08	Informal proceedings of the Workshop on User Interfaces for Theorem Provers. Eindhoven University of Technology ,13-15 July 1998	

		edited by R.C. Backhouse, p. 180
98/09	K.M. van Hee and H.A. Reijers	An analytical method for assessing business processes, p. 29.
98/10	T. Basten and J. Hooman	Process Algebra in PVS
98/11	J. Zwanenburg	The Proof-assistemt Yarrow, p. 15
98/12	Ninth ACM Conference on Hypertext and Hypermedia Hypertext '98 Pittsburgh, USA, June 20-24, 1998 Proceedings of the second workshop on Adaptive Hypertext and Hypermedia.	Edited by P. Brusilovsky and P. De Bra, p. 95.
98/13	J.F. Groote, F. Monin and J. v.d. Pol	Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.
98/14	T. Verhoeff (artikel volgt)	
99/01	V. Bos and J.J.T. Kleijn	Structured Operational Semantics of χ , p. 27
99/02	H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst	Diagnosing Workflow Processes using Woflan, p. 44
99/03	R.C. Backhouse and P. Hoogendijk	Final Dialgebras: From Categories to Allegories, p. 26
99/04	S. Andova	Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81
99/05	M. Franssen, R.C. Veltkamp and W. Wesselink	Efficient Evaluation of Triangular B-splines, p. 13
99/06	T. Basten and W. v.d. Aalst	Inheritance of Workflows: An Approach to tackling problems related to change, p. 66
99/07	P. Brusilovsky and P. De Bra	Second Workshop on Adaptive Systems and User Modeling on the World Wide Web, p. 119.
99/08	D. Bosnacki, S. Mauw, and T. Willemse	Proceedings of the first international syposium on Visual Formal Methods - VFM'99
99/09	J. v.d. Pol, J. Hooman and E. de Jong	Requirements Specification and Analysis of Command and Control Systems
99/10	T.A.C. Willemse	The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μ CRL, p. 44.
99/11	J.C.M. Baeten and C.A. Middelburg	Process Algebra with Timing: Real Time and Discrete Time, p. 50.
99/12	S. Andova	Process Algebra with Probabilistic Choice, p. 38.
99/13	K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen	A Framework for Component Based Software Architectures, p. 19
99/14	A. Engels and S. Mauw	Why men (and octopuses) cannot juggle a four ball cascade, p. 10
99/15	J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen	An algorithm for the asynchronous *Write-All* problem based on process collision*, p. 11.
99/16	G.J. Houben, P. Lemmens	A Software Architecture for Generating Hypermedia Applications for Ad-Hoc Database Output, p. 13.
99/17	T. Basten, W.M.P. v.d. Aalst	Inheritance of Behavior, p.83
99/18	J.C.M. Baeten and T. Basten	Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79
99/19	J.C.M. Baeten and C.A. Middelburg	Real Time Process Algebra with Time-dependent Conditions, p.33.
99/20	Proceedings Conferentie Informatiewetenschap 1999 Centrum voor Wiskunde en Informatica 12 november 1999, p.98	edited by P. de Bra and L. Hardman
00/01	J.C.M. Baeten and J.A. Bergstra	Mode Transfer in process Algebra, p. 14
00/02	J.C.M. Baeten	Process Algebra with Explicit Termination, p. 17.
00/03	S. Mauw and M.A. Reniers	A process algebra for interworkings, p. 63.
00/04	R. Bloo, J. Hooman and E. de Jong	Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.

00/05	J.F. Groote and M.A. Reniers	Algebraic Process Verification, p. 65.
00/06	J.F. Groote and J. v. Wamel	The Parallel Composition of Uniform Processes wit Data, p. 19
00/07	C.A. Middelburg	Variable Binding Operators in Transition System Specifications, p. 27.
00/08	I.D. van den Ende	Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.
00/09	R.R. Hoogerwoord	A Formal Development of Distributed Summation, p. 35
00/10	T. Willemse, J. Tretmans and A. Klomp	A Case Study in Formal Methods: Specification and Validation on the OM/RR Protocol, p. 14.
00/11	T. Basten and D. Bošnački	Enhancing Partial-Order Reduction via Process Clustering, p. 14
00/12	S. Mauw, M.A. Reniers and T.A.C. Willemse	Message Sequence Charts in the Software Engineering Process, p. 26
00/13	J.C.M. Baeten, M.A. Reniers	Termination in Timed Process Algebra, p. 36
00/14	M. Voorhoeve, S. Mauw	Impossible Futures and Determinism, p. 14
00/15	M. Oostdijk	An Interactive Viewer for Mathematical Content based on Type Theory, p. 24.
00/16	F. Kamareddine, R. Bloo, R. Nederpelt	Characterizing λ-terms with equal reduction behavior, p. 12
00/17	T. Borghuis, R. Nederpelt	Belief Revision with Explicit Justifications: an Exploration in Type Theory, p. 30.