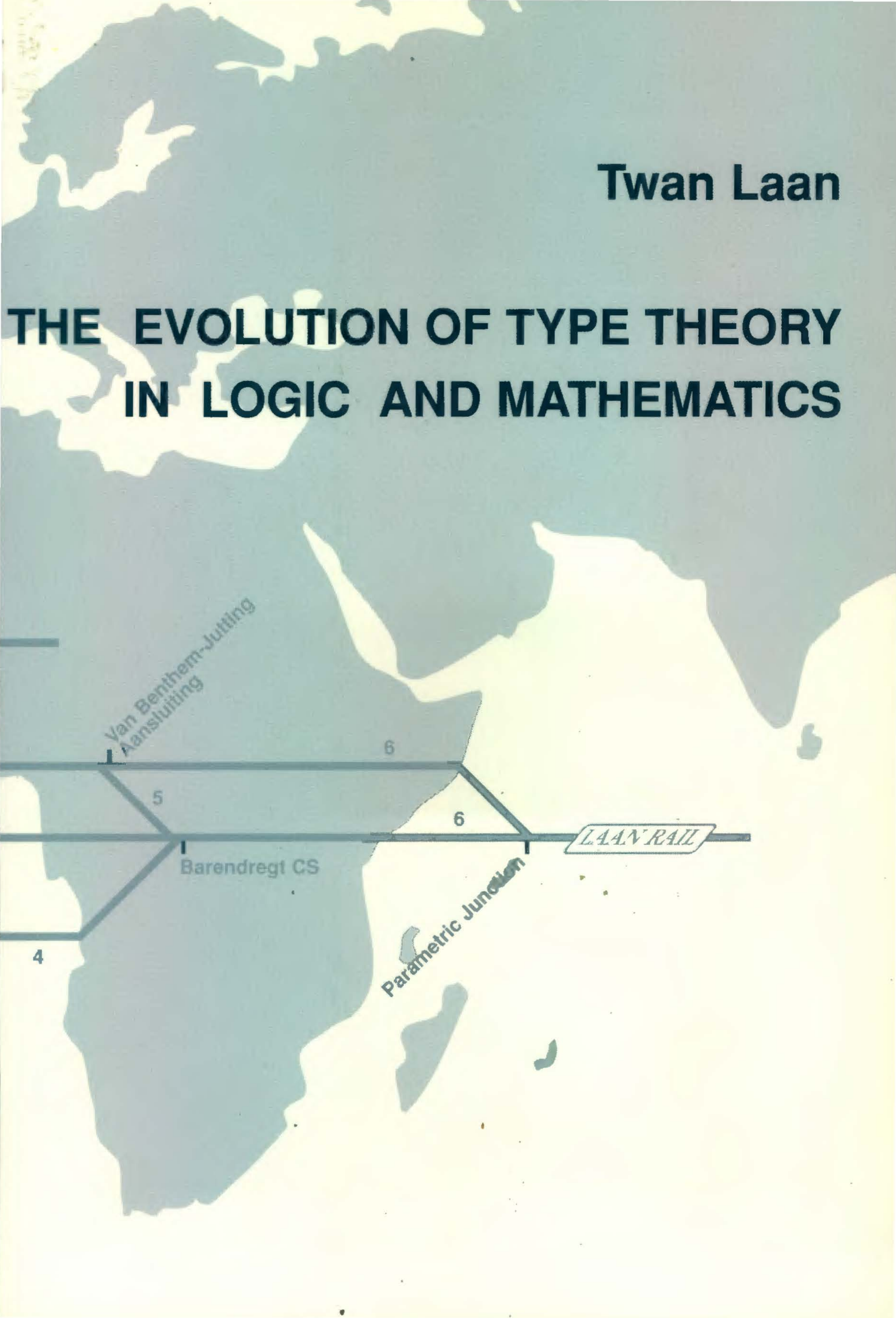
The evolution of type theory in logic and mathematics

Twan Laan

THE EVOLUTION OF TYPE THEORY IN LOGIC AND MATHEMATICS



The Evolution of Type Theory in Logic and Mathematics

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 29 september 1997 om 16.00 uur

door

Twan Dismas Laurens Laan

geboren te Etten-Leur

Dit proefschrift is goedgekeurd door de promotoren:
prof.dr. J.C.M. Baeten
prof.dr. H.C.M. de Swart
en de copromotor:
dr. R.P. Nederpelt

This thesis has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA) and the Cooperation Center Tilburg and Eindhoven Universities (SOBU).

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Laan, Twan Dismas Laurens

The evolution of type theory in logic and mathematics / by
Twan Dismas Laurens Laan. - Eindhoven : Technische
Universiteit Eindhoven, 1997. - xii, 344 p.
Proefschrift. - ISBN 90-386-0531-5
NUGI 811
Subject headings: programming languages ; data types /
mathematical logic / mathematical logic ; history
1991 Mathematics Subject Classification: 03B15, 03B40,
01-02, 01A60

IPA dissertation series 1997-04
Cover design by Elske Bleeker
Printed and bound by Print Partners Ipskamp, Enschede

The Evolution of Type Theory in Logic and Mathematics

Twan Laan

29 september 1997

Stellingen

behorende bij het proefschrift

The Evolution of Type Theory in Logic and Mathematics

van

Twan Laan

1. Type-theorie is zich pas gaan vertakken nadat bleek dat vertakking in de type-theorie niet noodzakelijk is (dit proefschrift, omslag).
2. De aanduiding "simple" in "simple type theory" verwijst niet naar het feit dat de simply typed λ-calculus een van de eenvoudigste getypeerde λ-calculi is, maar duidt op een type-systeem dat geen vertakte types ("ramified types") kent. Daarom zijn alle getypeerde λ-calculi in de kubus van Barendregt voorbeelden van "simple type theories".
3. Het toevoegen van een parameter-mechanisme aan Pure Type Systems is geen *uitbreiding* van dit framework maar een *verfijning* ervan (dit proefschrift, Hoofdstuk 6).
4. Het schrijven van een proefschrift is als het reduceren van een term in een termherschrijfsysteem dat niet noodzakelijk confluent of sterk normaliserend, maar (door de eindige levensduur van promovendi) wel zwak normaliserend is.
5. Een belangrijke eis aan een wetenschappelijk experiment is de *herhaalbaarheid* ervan. Daarom is het discutabel om informatica een wetenschap te noemen zolang één van de meest gegeven adviezen bij computernukken luidt: "uitzetten, aanzetten, en kijken of-ie dan wel werkt".
6. Het verkrijgen van een reisadvies voor een reis per openbaar vervoer is te vergelijken met het passen van kleding in een kledingwinkel en moet dus (net als het passen van kleding) gratis zijn.

7. Om "de automobilist in het openbaar vervoer te krijgen" is tariefintegratie tussen openbaar vervoer en personenauto minstens zo belangrijk als tariefintegratie tussen de verschillende vormen van openbaar vervoer onderling.

8. Het feit dat er beambten zijn die belast zijn met de controle en afgifte van vervoerbewijzen van het openbaar vervoer en die ten onrechte sommige zwembadabonnementen accepteren als geldig vervoerbewijs in bepaalde bussen, geeft aan dat er iets grondig mis is met de helderheid van de tariefstructuur (en daarmee: met de klantvriendelijkheid) van het Nederlandse openbaar vervoer.

9. Het verkrijgen van handtekeningen die noodzakelijk zijn voor het succesvol afleggen van het bureaucratische traject voorafgaande aan een promotieplechtigheid wordt aanmerkelijk vergemakkelijkt indien de eerste promotor tevens decaan van de betrokken faculteit is.

10. Engels en Frans zijn twee talen die niet vloeiend uit één mond kunnen komen.

Acknowledgements

This thesis is the main result of the project "De ontwikkeling van het Typebegrip", which was financially supported by the Co-operation Centre Tilburg and Eindhoven Universities (SOBU). NWO and Glasgow University provided funding for my trips to Glasgow in spring 1995 and winter 1996.

But a thesis cannot be written with money alone. Invaluable support was obtained from many persons in several ways.

Rob Nederpelt and Harrie de Swart came with the idea for the project. They wrote the research proposal, and in this way they made things generally happen.

It was a pleasure to work in the "types group" of Eindhoven University. Many suggestions and ideas of the people of this group found their way into this thesis. The results of the cooperation with Michael Franssen, Paula Severi and Jan Zwanenburg on Pure Type Systems with Parameters are reflected in Chapter 6. Bert van Benthem Jutting explained me some essentials of Automath, and Stuart Allan did the same with some details of the Russell paradox in Frege's *Grundgesetze der Arithmetik*.

Most ideas for Chapter 3, and many for Chapter 4, were born when I worked together with Fairouz Kamareddine at Glasgow University. To those who do not know Fairouz it may sound strange that the basics for almost two full chapters arose within three months. To others, who know her energy and the way in which she can motivate people, this will not come as a surprise.

Araminte Bleeker, Tijn Borghuis, Herman Geuvers, and Jan Zwanenburg read a manuscript of the manuscript and made it considerably less inconsistent than it initially was.

The less inconsistent manuscript was carefully read by the manuscript committee, consisting of my promotors prof.dr. J.C.M. Baeten and prof.dr. H.C.M. de Swart, my co-promotor Rob Nederpelt, and prof.dr. H.P. Barendregt, and prof.dr. D. van Dalen.

Elske Bleeker took care that the overall result got packed in a nice cover.

Roel Bloo managed to be my room-mate for the last four years. He was perfect company, and always open to discuss my work with me.

My colleagues of the Formal Methods group in Eindhoven and the Logic group in Tilburg provided a pleasant atmosphere, and even survived my railway-enthousiasm.

Sitting in front of my over-developed Turing-machine and watching all the pages that now form this thesis, writing a thesis seems to be a cool thing. In spring 1996 (when not even one character for this thesis was written) my opinion on writing thesises was quite a different one. It seemed to be something impossible. At that time, many people explained me that writing a thesis is one of the less important things in life. Strange enough, this gave me ample motivation to write and finish it. On this point I am particularly indebted to Rob Nederpelt, Annemarie Broekhuysen, my paranymphs Dion-ben Hendriks and Araminte Bleeker, and my parents Jan and Stefanie Laan for their open ears, clear explanation, and unconditional support.

Twan Laan,
Eindhoven,
29 September 1997

Contents

Introduction

Nowadays, type theory has many applications and is used in a lot of different disciplines. Even within logic and mathematics, there are a lot of different type systems. They serve several purposes, and are formulated in various ways. In this thesis we present a formal framework in which various type theories can be described. This framework is an extension of an already existing framework.

For the development of this framework, we follow the evolution of type theory throughout the past century. However, we do not only give a mere historical description. On the contrary: our goal is not to describe the various type systems that have been developed in their historical setting, but to present them in a modern framework. In this way it becomes clear how the various type systems are related to each other, even if originally those systems are described in very different ways. Moreover, we can make clear what is the essence, or the common basis, of the various modern type theories.

The historical line in this thesis is, therefore, only part of our *method* of research, and definitely not a *goal* of our research.

The approach

Following the historical line from Frege (1879) to today, we are confronted with various type systems. Often, such a system has already been described in a modern framework, but the relation between the modern description and the original system has not always been made clear. This is particularly the case when the original system is quite far from the modern framework with respect to notation, level of formality and/or purpose. We will focus on such type systems. We describe them within the framework in such a way that:

- We respect the ideas and the philosophy underlying the original system;
- We meet contemporary requirements on formality and accuracy.

As basis for our framework we choose typed λ-calculus, more specific the framework of Pure Type Systems. There are several reasons for this choice:

- Many type systems have already been placed in this framework (see Example 5.2.4 of [5]);
- PTSs meet contemporary requirements on formality and accuracy;
- PTSs focus on the heart of type theory: functionalisation and instantiation (see below). This makes it possible to compare type systems in a very fundamental way, without being hindered by things that do not touch the heart of the matter;
- Though PTSs focus on the heart of type theory, they are easily extendible in several ways. There are already many extensions described in the literature:
  - PTSs with definitions, introduced in [114];
  - PTSs with modalities, introduced in [18];
  - PTSs with sum types, see [6];
  - PTSs with quotient types, see [68], [6];
  - PTSs with subset types, see [6].

  Another extension (PTSs with parameters and parametric definitions) is studied in this thesis (see Chapter 6);
- The meta-theory that has been developed for PTSs makes it easier to access, develop and compare meta-theoretic properties of the various original type systems.

By placing several systems in the PTS framework, we also find some omissions in this framework. In particular, there is no extension of PTSs with parameters, while parameters play an important role in the type systems underlying the proof checker AUTOMATH [95]. Extending PTSs with parameters not only opens the possibility of placing AUTOMATH more accurately in the framework of PTSs, it makes it also possible to give a better classification of more modern type systems and their applications.

In the above, we claimed that PTSs focus on the heart of type theory: functionalisation and instantiation. We now describe what we mean by functionalisation and instantiation, and why these two notions are in our opinion the heart of the matter.

The heart of type theory

The explicit and formal use of types (and thus an early form of what is presently called "type theory") was originally intended to prevent the paradoxes that occurred in logic and mathematics at the end of the 19th and the beginning of the 20th century. But it was not the only method developed for this purpose. Another tool was the fine-tuning of Cantor's Set Theory [25, 26] by Zermelo [123]. The approach of type theory however, is completely different from the set-theoretical approach. Type theory focuses on the notion of *function* in logic and mathematics, and throughout the history of type theory, functions have remained one of the main objects of study for type theorists.

In the abstract theory of functions, there are only two important constructions: *functionalisation* and *instantiation*. We now discuss both construction principles.

Functionalisation

Consider the mathematical expression $2 + 3$. This expression indicates the addition of the number 2 to the number 3. Both 2 and 3 are fixed objects. But we cannot only think of the addition of 2 to 3, but also of the addition of any other number to 3. This means that we replace the object 2 by a symbol that denotes "any natural number": a variable (say: $\mathtt{x}$). This results in the expression $\mathtt{x} + 3$. This expression does not denote one specific natural number, but if we replace $\mathtt{x}$ by a natural number, then the resulting expression represents a natural number. This replacement-activity is similar for the various possibilities for x, and gives rise to an algorithm: We feed a natural number to the algorithm, the algorithm adds 3 to that natural number, and returns the result to us. This algorithm is called a *function*.

Notice that the function that returns $\mathrm{x}+3$ whenever we assign a natural number to x is more than just the expression $\mathrm{x}+3$: The expression $\mathrm{x}+3$ denotes some natural number, but the function denotes an algorithm. Moreover, the expression $\mathrm{y}+3$ is an expression that is different from $\mathrm{x}+3$. We can have two different natural numbers in mind for x and y. But if we construct functions from $\mathrm{x}+3$ and $\mathrm{y}+3$ by the method described above, we obtain the same algorithm.

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

- Frege ([45], 1879) simply wrote $\mathrm{x}+3$, and did not make any difference between the algorithm and the expression $\mathrm{x}+3$;[1]
- Russell ([121], 1910) was more clear on this point, and wrote $\hat{\mathrm{x}}+3$ to distinguish the algorithm from the expression $\mathrm{x}+3$;
- Church ([28], 1932) used $\lambda \mathrm{x}.\mathrm{x}+3$ where Russell wrote $\hat{\mathrm{x}}+3$;
- In the proof checker AUTOMATH ([95], 1968), the notation $[\mathrm{x}{:}\mathbb{N}]\mathrm{x}+3$ is used;
- In explicitly typed λ-calculi (also known as λ-calculi "in Church-style") one usually writes $\lambda \mathrm{x}{:}\mathbb{N}.\mathrm{x}+3$. The $\mathrm{x}{:}\mathbb{N}$ behind the λ denotes that the algorithm requires "special food". We cannot just feed it anything we want, it only eats natural numbers;
- In many mathematical texts, we find the notation $\mathrm{x} \mapsto \mathrm{x}+3$.

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

1. *Abstraction from a subexpression.* First, we replace an subexpression k in an expression f by a variable x, at one or more places where k occurs in f. Thus we obtain a new expression, f';

[1]Frege used the notation $\dot{\mathrm{x}}(\mathrm{x}+3)$ for what is called the *course-of-value* of the algorithm above, but not for the algorithm itself. See Section 1b1.

2. *Function construction.* Then we construct the function $\lambda x.f'$ that assigns $f'[x{:=}a]$ to a value a that we feed it. Here, $f'[x{:=}a]$ denotes f', in which a has been substituted for x. Sometimes, "substituted" denotes: "replaced" (as in λ-calculus). In other systems, like Russell's Ramified Theory of Types, substitution is a more complicated operation (see Section 2a4).

We call this process: *Functionalisation.*

The first part of the functionalisation process, abstraction from a subexpression, is already present in Frege's *Begriffsschrift*:

> "If in an expression, [...] a simple or a compound sign has one or more occurrences and if we regard that sign as replaceable in all or some of these occurrences by something else (but everywhere by the same thing), then we call the part that remains invariant in the expression a function, and the replaceable part the argument of the function."
>
> (*Begriffsschrift*, Section 9)

Frege, however, did not introduce separate notations for for example, the expression $\mathrm{x} + 3$ and the function $\lambda\mathrm{x}.\mathrm{x} + 3$. Hence, Frege did not employ the second part of functionalisation, the function *construction.*

However, both parts of the functionalisation process are present in *Principia Mathematica* by Whitehead and Russell [121]. The first part is represented by the "vice versa" part of $*9{\cdot}14$ below, and the combination of the first and the second part is represented by $*9{\cdot}15$ below:

> "**$*$9·14.** If 'φx' is significant, then if x is of the same type as a, 'φa' is significant, and vice versa.
> **$*$9·15.** If, for some a, there is a proposition φa, then there is a function $\varphi\hat{x}$, and vice versa"
>
> (*Principia Mathematica*, p. 133)

Here, φx denotes an expression in which a variable x occurs. Similarly, φa denotes an expression in which a sub-expression a occurs, and $\varphi\hat{x}$ denotes the function (algorithm) that assigns the value φa to an argument a.

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

Functions of more variables can be constructed by repeatedly applying functionalisation. This repetition process is often called "currying" after H.B. Curry, and is usually accredited to Schönfinkel ([109], 1924), but some of the ideas behind it are already present in Frege's *Begriffsschrift* (1879):

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

For Frege, this procedure of introducing several variables one by one results in the functions of more than one variable as used in ordinary mathematics. Schönfinkel's procedure results in curried functions as we know them from λ-calculus.

In λ-calculus, functionalisation focuses on the function construction. The abstraction from subexpressions can be omitted, as variables form the basic terms of the λ-calculus.

The notion of functionalisation in the works of Frege and Russell is inaccurate according to modern standards. There are many unexpected choices for the expression that is replaced by a variable in the first step of function construction. Examples will be given in Remark 2.10. In modern systems, which usually use λ-calculus, the second step of function construction is very clear: From a term f we can construct a term $\lambda x.f$. But also the

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

first step can be made clear. Take again the expression $2 + 3$. This term is β-equivalent to the λ-term $(\lambda \mathrm{x}.\mathrm{x} + 3)2$, which can be regarded as the term $\lambda \mathrm{x}.\mathrm{x} + 3$ (a *function*) applied to an *argument*, viz. 2. More precise, $(\lambda \mathrm{x}.\mathrm{x} + 3)2$ is a β-expansion of $2 + 3$. In this β-expansion, both the newly introduced variable, x, and the object that is replaced by x, namely 2, are present. They are linked via the λ-abstractor. The term $\lambda \mathrm{x}.\mathrm{x} + 3$ is a *function* which is applied to the *argument* 2. By removing the argument 2 from $(\lambda \mathrm{x}.\mathrm{x} + 3)2$, we obtain the function $\lambda \mathrm{x}.\mathrm{x} + 3$. More generally, we can construct a function from a λ-term f by first taking a β-expansion $(\lambda \mathrm{x}.f')a$ of f, and then removing the argument a. This is a much more precise description of functionalisation than the one that is given in the work of Frege.

This does not mean that the work on functionalisation has finished now. There are several variants of functionalisation that have not yet been studied completely. See the section on special forms of functionalisation below.

Instantiation

Instantiation is the inverse process of functionalisation. It consists of applying a function to an argument, and calculating the result of this application. As the function is an algorithm, it is prescribed how this calculation should be made. For example, if we instantiate the function $\lambda \mathrm{x}.\mathrm{x} + 3$ with the argument 2, we first apply this function to 2, obtaining $(\lambda \mathrm{x}.\mathrm{x} + 3)2$, and then calculate the result via β-reduction: $2 + 3$. Just like the functionalisation process, the instantiation process has two phases:

1. *Application construction.* Juxtaposing the function f to an argument a; the result is usually written as $f(a)$ or fa and denotes an *intended* function application;

2. *Concretisation to a subexpression.* Calculating the result of this intended application. f is a function, and therefore has been constructed from an expression f' with a free variable (say x). The calculation usually consists of a substitution of a for the free variable x.

These phases are clearly visible in the λ-calculus above. The calculation consists of removing the λ-abstraction from $\lambda \mathrm{x}.\mathrm{x} + 3$, and substituting the

$$\begin{tikzcd}
2+3 \arrow[r, shift left, "\text{abstraction from a subexpression}"] \arrow[rr, bend left=40, "\text{functionalisation}"] & (\lambda x.x+3)2 \arrow[l, shift left, "\text{concretisation to a subexpression}"] \arrow[r, shift left, "\text{function construction}"] & \lambda x.x+3 \arrow[l, shift left, "\text{application construction}"] \arrow[ll, bend left=40, "\text{initialisation}"]
\end{tikzcd}$$

Figure 1: Functionalisation and instantiation are each others inverse

argument 2 for the free variable that appears when the λ-abstraction is removed. Moreover, it becomes clear that function construction is the inverse procedure of application construction, and that abstraction from a subexpression is the inverse procedure of concretisation to a subexpression. See Figure 1.

It is not always that simple. Sometimes, the function f to which we apply an argument a is not a concrete object, but only a variable. For example, look at the function $\lambda y.zy$ that is applied to an argument 2. In that case, the instantiation cannot be carried out completely. We can apply the function to the argument, obtaining $(\lambda y.zy)2$, and this term β-reduces to $z2$. As we do not have a concrete object as function, but only the variable z, we cannot proceed with this calculation. If we substitute a function for z (say: $\lambda x.x+3$), we obtain $(\lambda x.x+3)2$. Then we continue the calculation by substituting 2 for x in $x+3$, obtaining $2+3$.

As with functionalisation, instantiation can now be precisely defined in terms of λ-calculus. In the works of Frege and Russell, we do not find such a precise description of instantiation. The application construction is well-described (for instance in the "vice-versa" part of ∗9·15 in *Principia Mathematica* — see the quotation at page 5), but there is no precise definition of the concretisation to subexpressions by means of substitution. This is not so very important as long as it is straightforward how the substitution should be carried out. However, we will see that substitution is not a straightforward procedure in *Principia Mathematica.*

The precise definition of substitution in λ-calculus is due to Curry and Feys [38] (1958). However, we must remark that "precise" is a relative notion here. The presentations of functionalisation and instantiation that were given by Frege and Russell were very precise for those days. And the definition of substitution by Curry and Feys in 1958 is not the last word to be said on the notion of substitution. Currently, there is quite some research on so-called *explicit* substitutions, which are refinements of the notion of substitution of Curry and Feys. See [15], [1], [71], [8].

Special forms of functionalisation

The mechanisms of functionalisation and instantiation that are used in λ-calculus are quite powerful, but have some disadvantages:

- In λ-calculus, the first step in the functionalisation process is not carried out. In particular, the functionalisation process in λ-calculus does not show from which term (object) has been abstracted. This is contrary to the systems of Frege and Russell, where it is clear from the functionalisation process from which object has been abstracted.

  However, there are also modern functionalisation processes in which it is essential to remind the original term from which has been abstracted. A good example is the use of *definitions*. If an subexpression k occurs in an expression f, it is sometimes practical to introduce an abbreviation for k, for several reasons:

  - The syntactical representation of the object k may be long. This makes manipulations with f a time-consuming and memory-consuming task, in particular when k occurs several times in f. Abbreviating k can make manipulations with f easier;
  - The object k may represent a structure that is particularly interesting. Abbreviating k opens the possibility to introduce a significant name for k. This makes the expression easier to understand for human beings.

  Abbreviating k can be seen as an functionalisation process: we replace all the occurrences of k by its definiendum (its name), and then have an *unfolding algorithm* that can be used to replace the definiendum by its definiens (that is: k) when the internal structure of k is needed in the term f. For example, if we develop the theory of natural numbers using the axioms of Peano, we have terms 0 (zero), S0 (the successor of zero, or: one), S(S0) (two), S(S(S0)), etc. In this notation, the expression $2+3$ looks like S(S0)+S(S(S0)). Introducing abbreviations

(1 for $\mathtt{S0}$, 2 for $\mathtt{S(S0)}$, etc.) makes the term shorter and more clear. The definiens of 2 can be stored in some context, but it can also be stored directly in front of the term $2 + 3$ like a λ-abstraction. We then obtain a new term

$$2\mathtt{=S(S0)}\ \textsc{in}\ 2 + 3.$$

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

- It is not always necessary to make "full functionalisation". For instance, have a look at the axiom for natural induction. This axiom can be written as a function:

  $$\lambda \mathtt{P.P0} \to \forall \mathtt{n[Pn} \to \mathtt{P(Sn)]} \to \forall \mathtt{n[Pn]}.$$

  This function takes one argument: a predicate on natural numbers P. A mathematician usually is not interested in the axiom presented in the above formulation. Often he is interested in instantiations of the axiom only. Therefore, a mathematician may prefer the induction axiom in the form of an axiom *scheme*, depending on a *parameter* for the predicate P. The scheme itself is not part of the formal language, but all the instantiations of the scheme do. As the scheme itself is not part of the language, this "parametric" presentation of the induction axiom is not as strong as the presentation with the λ-term: The latter is part of the formal language, so it is possible to discuss the axiom within the formal language. Nevertheless, the parametric presentation occurs very often in practice (for instance, in applications and implementations of mathematics), and therefore deserves a closer study (see Chapter 6 of this thesis).

Preliminaries

We assume that the reader is more or less familiar with the basics of typed λ-calculus and type theory. We give a survey of the most important topics concerning typed λ-calculus in Appendix A. General introductions to type theory are also available in the literature (e.g. [93], [99]).

Pure Type Systems play an important role in the thesis, especially in the chapters 4–6. At the point where PTSs enter the work, in Section 4b, we give a short explanation of the various PTS-rules. Again, we refer to Appendix A for a short summary of the theory regarding PTSs.

Overview of this thesis

The thesis is divided into six chapters.

In the first chapter we discuss the prehistory of type theory. That is, we study the way in which types implicitly occurred in logic and mathematics before there was an explicit theory of types. We pay special attention to the formalisation of logic that is made in Frege's *Begriffsschrift* [45] and *Grundgesetze der Arithmetik* [48, 52], as in this system many basic ideas are presented that are later used in type theory. Moreover, the system of *Grundgesetze der Arithmetik* is the one for which Russell derives his famous paradox, and this paradox has been the reason for Russell to introduce the first theory of types.

This first type theory is the subject of the second chapter. Whitehead and Russell present their theory, the Ramified Type Theory (RTT), in an informal way. Several rough descriptions of this theory have been given in the literature (see for instance [101], [64], [30] and [32]) but we present a formalisation of RTT that is directly based on the presentation of RTT in Whitehead and Russell's *Principia Mathematica* ([121], 1910-12). The construction of this formalisation is not a simple task. Whitehead and Russell do not present a clear syntax for their so-called *propositional functions* in [121], neither do they make a clear difference between syntax and semantics. We present a formal definition of propositional function that is faithful to the original ideas exposed in *Principia Mathematica.* A second technical problem is the notion of substitution, which is totally undefined in *Principia Mathematica.* The formalisation of the notion of propositional function makes it possible to express the notion of substitution of *Principia Mathematica* in terms of λ-calculus. We use techniques from typed and untyped λ-calculus to give a precise description of substitution, and to show that substitution is well-defined as long as we restrict ourselves to well-typed propositional functions of RTT.

In 1926, Ramsey [101] proposes an important simplification of RTT, the *simple* theory of types. This simple type theory has become the basis for many modern type systems, and for the simply typed λ-calculus of Church [30]. The simplification consisted of the removal of one of the two hierarchies from the RTT. The hierarchy of types is maintained, while the hierarchy of orders is removed. In Chapter 3 we discuss this process of so-called *deramification*. An important observation of this Chapter is that though the orders do not occur in the mainstream of type theories, they still provide an important intuition for logicians. We show that there is a close link between the hierarchy of orders in RTT and the hierarchy of truths that was introduced by Kripke [78]. We also show that Kripke's use of orders is more flexible than Russell's, and that this is due to the fact that orders occur at the semantical level in Kripke's theory, while they occur at the level of syntax in RTT.

Though type theory clearly served as a method to prevent certain logical paradoxes, the logical system stood apart from the type system until the 1950s. In Chapter 4 we study the ways in which logic can be included in a type system. The various methods are all based on the idea that the proof of a logical implication can be seen as a function. More precisely: A proof of the proposition $A \to B$ is implemented as a function that takes a proof of A as argument, and returns a proof of B. In this way, the *proposition* $A \to B$ can be seen as the *type* of all functions from (proofs of) A to (proofs of) B. Similarly, a *proof* of $A \to B$ becomes a *term* of type $A \to B$. One callsthis principle: *propositions as types*, or: *proofs as terms*. Both expressions are abbreviated by PAT. We illustrate the principle by giving a description of RTT in a PAT style.

One of the important applications of PAT is the mechanical verification of mathematical proofs. The first tool for such a verification was AUTOMATH. It was developed in 1968. The languages of the various AUTOMATH systems have been studied intensively. In [5], a description of two of the most important systems within the framework of Pure Type Systems is given, but without any explanation. In Chapter 5 we study the original language of AUTOMATH and translate it to a PTS format. In doing so, we obtain descriptions similar to the ones in [5].

The description in Chapter 5 is precise, but does not take into account two important mechanisms of AUTOMATH: the definition mechanism and the parameter mechanism. Many other type systems use these mechanisms as well. This motivates us to extend the framework of PTSs with definitions and parameters in Chapter 6. As far as definitions are concerned, this extension is based on the PTSs with definitions that were introduced by Severi and Poll [114]. This extension results in a refinement of the framework of PTSs. In this refined framework, various modern type systems (like LF and ML) can be described in a more precise way than in the PTS framework without definitions and parameters.

Chapter 1

Prehistory

In this chapter, we discuss the development of type theory before it was actually baptised. This may sound like a contradiction. But types have played an important (though not very apparent) role in mathematics even before the theory of types was explicitly introduced by Russell in 1908 [108]. Moreover, knowledge of the development of logic and mathematics before 1908, and especially of the occurrence of the logical paradoxes at the turn of the century, provides insight in the way in which Russell and others formulated their theories of types.

When the first formalisations of parts of mathematics and logic appeared, the types were left implicit. Cantor's Set Theory [25, 26], Peano's formalisation of the theory of natural numbers in [97], and Frege's *Begriffsschrift* [45] and *Grundgesetze der Arithmetik* [48, 52] did not have a formal type system. The type of an object is indicated by means of natural language ("Let a be a proposition") or is taken for granted. Types were informally present in the background of these theories, but a formal representation of the types was not incorporated: one could say that they were separated from logic and mathematics.

However, even without a formalisation of the notion of types, the introduction of formal language had considerable advantages in the description of mathematical notions. The formalisation made it easier to give a precise definition of important abstract concepts, like the concept of function. The precise formulation allowed for a generalisation of the notion of function to include not only functions that take numbers as an argument, and return a number, but also functions that can take and return other sorts of arguments (like propositions, but also functions). Unfortunately, this also allowed logical paradoxes to enter the formal theory, without the (informal) type mechanism being able to prevent that.

In this chapter we first argue that types have always been present in mathematics, though probably nobody was aware of it before the end of the 19th century (Section 1a). We proceed by describing how the logical paradoxes entered the formal systems of Frege, Cantor and Peano in Section 1b.

The historical remarks in this chapter have been taken from various resources. The most important ones are [14], [37], [61], [76], [99], [104] and [122].

1a Paradox threats

The most fundamental idea behind type theory is being able to distinguish between different classes of objects (*types*). Until the end of the 19th century it had hardly ever been necessary to make this ability explicit. The mathematical language itself was predominantly informal, and so was the use of classes of objects.

It is, however, difficult to argue that there were no types before Russell "invented" them in 1908. Already around 325 B.C., Euclid began his *Elements* [43] with the following primitive definitions:

1. A *point* is that which has no part;

2. A *line* is breadthless length.

From these two basic notions of "point" and "line", Euclid defined more complex notions, like the notion of "circle":

15. A *circle* is a plane figure contained by one line such that all the straight lines falling upon it from one point among those lying within the figure are equal to one another.

At first sight, these three observations are mere definitions. But these three pieces of text do not only *define* the notions of point, line and circle, they also show that Euclid *distinguished* between points, lines and circles.

Throughout the *Elements*, Euclid always mentioned to which class an object belonged (the class of points, the class of lines, etc.). In doing so, he prevented undesired situations, like the intersection of two points (instead of two lines).

Undesired results? Euclid himself would probably have said: *impossible* results. When talking of an intersection, intuition implicitly forced him to think about the *type* of the objects of which he wanted to construct the intersection. As the intersection of two points is not supported by intuition, he did not even *try* to undertake such a construction.

Euclid's attitude to, and implicit use of type theory was maintained by the mathematicians and logicians of the next twenty-one centuries. From the 19th century on, mathematical systems became less intuitive, for several reasons:

1. The system itself is complex, or abstract. An example is the theory of convergence in real analysis;
2. The system is a formal system, for example, the formalisation of logic in Frege's *Begriffsschrift*;
3. (In the second half of the 20th century:) It is not a human being working with the system, but something with less intuition, in particular: a computer.

We will call these three situations *paradox threats*. In all these cases, there is not enough intuition to activate the (implicitly present) type theory to warn against an impossible situation. One proceeds to reason within the impossible situation and then obtains a result that may be wrong or paradoxical: an *undesired* situation. We mention examples related to the three situations above:

ad 1. The controversial results on convergence of series in analysis obtained in the 17th and 18th century, due to lack of knowledge on what real numbers actually are;

ad 2. The logical paradoxes that arose from self-application of functions. Self-application is intuitively impossible, but this is easily forgotten when working in a formal system in which such self-application can be expressed. The result is undesirable: a logical paradox;

ad 3. An untyped computer program may receive instructions from a not too watchful user to add the number 3 to the string `four` (instead of the number 4). The computer, unaware of the fact that `four` is not a number, starts his calculation. It is not programmed to handle the calculation of 3+`four`. The result of this calculation is unpredictable. The computer may

- give an answer that is clearly wrong (for example, **);
- give no answer at all;
- give an answer that is not so clearly wrong (for example, 6).

Especially the last situation is highly undesirable.

The example ad 2 is the main subject of the next section.

1b Paradox threats in formal systems

In the 19th century, the need for a more precise style in mathematics arose. Controversial results had appeared in analysis. Many of these controversies were solved by the work of Cauchy. For instance, he introduced a precise definition of convergence in his *Cours d'Analyse* [27]. Due to the more exact definition of real numbers given by Dedekind [41], the rules for reasoning with real numbers became even more precise.

In 1879, Frege published his *Begriffsschrift* [45], in which he presented the first formalisation of logic. Frege's reasoning was uncommonly precise for those days. Until then, it had been possible to make mathematical and logical concepts more clear by textual refinement in the natural language in which they were described. Frege was not satisfied with this:

> " ... I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain the precision that my purpose required."
>
> (*Begriffsschrift*, Preface)

Frege therefore presented a completely formal system, whose

> "first purpose is to provide us with the most reliable test of the validity of a chain of inferences and to point out every presupposition that tries to sneak in unnoticed, so that its origin can be investigated."
>
> (*Begriffsschrift*, Preface)

1b1 Functions and their course of values

The introduction of a very general definition of function was the key to the formalisation of logic. Frege defined what we will call the *Abstraction Principle*:

Abstraction Principle 1.1

> *"If in an expression, [...] a simple or a compound sign has one or more occurrences and if we regard that sign as replaceable in all or some of these occurrences by something else (but everywhere by the same thing), then we call the part that remains invariant in the expression a function, and the replaceable part the argument of the function."*
>
> *(Begriffsschrift,* Section 9*)*

Frege put no restrictions on what could play the role of an argument. An argument could be a number (as was the situation in analysis), but also a proposition, or a function. Similarly, the result of applying a function to an argument did not necessarily have to be a number. Functions of more than one argument were constructed by a method that is very close to the method presented by Schönfinkel [109] in 1924:

Abstraction Principle 1.2

> *"If, given a function, we think of a sign*[1] *that was hitherto regarded as not replaceable as being replaceable at some or all of*

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

> *its occurrences, then by adopting this conception we obtain a function that has a new argument in addition to those it had before."*
>
> (*Begriffsschrift,* Section 9)

With this definition of function, two of the three possible paradox threats mentioned on p. 16 occurred:

1. The generalisation of the concept of function made the system more abstract and less intuitive. The fact that functions could have different types of arguments is at the basis of the Russell Paradox;

2. Frege introduced a *formal* system instead of the informal systems that were used up till then. Type theory, that would be helpful in distinguishing between the different types of arguments that a function might take, was left informal.

So, Frege had to proceed with caution. And so he did, at this stage. He remarked that

> "if the [...] letter [sign] occurs as a function sign, this circumstance [should] be taken into account."
>
> (*Begriffsschrift*, Section 11)

This could be interpreted as if Frege was aware of some typing rule that does not allow to substitute functions for object variables or objects for function variables. In his paper *Function and Concept* [47], Frege more explicitly stated:

> " Now just as functions are fundamentally different from objects, so also functions whose arguments are and must be functions are fundamentally different from functions whose arguments are objects and cannot be anything else. I call the latter first-level, the former second-level."
>
> (*Function and Concept*, pp. 26–27)

A few pages later he proceeded:

> "In regard to second-level functions with one argument, we must make a distinction, according as the role of this argument can be played by a function of one or of two arguments."
>
> (*Function and Concept*, p. 29)

Therefore, we may safely conclude that Frege avoided the two paradox threats in the *Begriffsschrift*. In *Function and Concept* we even see that he was aware of the fact that making a difference between first-level and second-level objects is essential in preventing certain paradoxes:

> "The ontological proof of God's existence suffers from the fallacy of treating existence as a first-level concept."
>
> (*Function and Concept*, p. 27, footnote)

The *Begriffsschrift*, however, was only a prelude to Frege's writings. In *Grundlagen der Arithmetik* [46] he argued that mathematics can be seen as a branch of logic. In *Grundgesetze der Arithmetik* [48, 52] he actually described the elementary parts of arithmetics within an extension of the logical framework that was presented in the *Begriffsschrift*.

Frege approached the paradox threats for a second time at the end of Section 2 of his *Grundgesetze*. There he defined the expression "the function $\Phi(x)$ has the same course-of-values as the function $\Psi(x)$" by

> "the functions $\Phi(x)$ and $\Psi(x)$ always have the same value for the same argument."
>
> (*Grundgesetze*, p. 7)

Note that functions $\Phi(x)$ and $\Psi(x)$ may have equal courses-of-values even if they have different definitions. For instance, let $\Phi(x)$ be $x \wedge \neg x$, and $\Psi(x)$ be $x \leftrightarrow \neg x$, for all propositions x. Then $\Phi(x) = \Psi(x)$ for all x. So $\Phi(x)$ and $\Psi(x)$ are different functions, but have the same course-of-values.

Frege denoted the course-of-values of a function $\Phi(x)$ by $\grave{\varepsilon}\Phi(\varepsilon)$.[2] The definition of equal courses-of-values could therefore be expressed as

$$\grave{\varepsilon}f(\varepsilon) = \grave{\varepsilon}g(\varepsilon) \longleftrightarrow \forall a[f(a) = g(a)]. \tag{1}$$

In modern terminology, we could say that the functions $\Phi(x)$ and $\Psi(x)$ have the same course-of-values if they have the same *graph.*

Frege did not provide a satisfying intuition for the formal notion of course-of-values of a function. He treated courses-of-values as ordinary objects. As a consequence, a function that takes objects as arguments could have its own course-of-values as an argument. In modern terminology: a function that takes objects as arguments can have its own graph as an argument. All essential information of a function is contained in its graph. So intuitively, a system in which a function can be applied to its own graph should have similar possibilities as a system in which a function can be applied to itself. Frege excluded the paradox threats from his system by forbidding self-application, but due to his treatment of courses-of-values these threats were able to enter his system through a back door.

1b2 The Russell Paradox in the *Grundgesetze*

In 1902, Russell wrote a letter to Frege [106], in which he informed Frege that he had discovered a paradox in Frege's *Begriffsschrift*. Russell gave his well-known argument, defining the propositional function $f(x)$ by $\neg x(x)$ (in Russell's words: "to be a predicate that cannot be predicated of itself"). He assumed $f(f)$. Then by definition of f, $\neg f(f)$, a contradiction. Therefore: $\neg f(f)$ holds. But then (again by definition of f), $f(f)$ holds. Russell concluded that both $f(f)$ and $\neg f(f)$ hold, a contradiction.

[2]This may be the origin of Russell's notation $\hat{x}\Phi(x)$ for the class of objects that have the property Φ. According to a paper by J. B. Rosser [105], the notation $\hat{x}\Phi(x)$ has been at the basis of the current notation $\lambda x.\Phi$ in λ-calculus. Church is supposed to have written $\wedge x\Phi(x)$ for the function $x \mapsto \Phi(x)$, writing the hat in front of the x in order to distinguish this function from the class $\hat{x}\Phi(x)$. For typographical reasons, the $\wedge$ is supposed to have changed into a λ. On the other hand, J. P. Seldin informed us [111] that he had asked Church about it in 1982, and that Church had answered that there was no particular reason for choosing λ, that some letter was needed and λ happened to have been chosen. Moreover, Curry had told him that Church had a manuscript in which there were many occurrences of λ already in 1929, so three years before the paper [28] appeared.

Only six days later, Frege answered Russell that Russell's derivation of the paradox was incorrect [51]. He explained that the self-application $f(f)$ is not possible in the *Begriffsschrift*. $f(x)$ is a function, which requires an *object* as an argument, and a function cannot be an object in the *Begriffsschrift* (see 1b1).

In the same letter, however, Frege explained that Russell's argument could be amended to a paradox in the system of his *Grundgesetze*, using the course-of-values of functions. Frege's amendment was shortly explained in that letter, but he added an appendix of eleven pages to the second volume of his *Grundgesetze* in which he provided a very detailed and correct description of the paradox.

The derivation goes as follows (using the same argument as Frege, though replacing Frege's two-dimensional notation by the nowadays more usual one-dimensional notation). First, define the function $f(x)$ by:

$$\neg\forall\varphi[(\grave{\alpha}\varphi(\alpha) = x) \longrightarrow \varphi(x)].$$

Write $K = \grave{\varepsilon}f(\varepsilon)$. By (1) we have, for any function $g(x)$,

$$\grave{\varepsilon}g(\varepsilon) = \grave{\varepsilon}f(\varepsilon) \longrightarrow g(K) = f(K)$$

and this implies

$$f(K) \longrightarrow ((\grave{\varepsilon}g(\varepsilon) = K) \longrightarrow g(K)). \tag{2}$$

As this holds for any function $g(x)$, we have

$$f(K) \longrightarrow \forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \to \varphi(K)]. \tag{3}$$

On the other hand, for any function g,

$$\forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \to \varphi(K)] \longrightarrow ((\grave{\varepsilon}g(\varepsilon) = K) \to g(K)).$$

Substituting $f(x)$ for $g(x)$ results in:

$$\forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \to \varphi(K)] \longrightarrow ((\grave{\varepsilon}f(\varepsilon) = K) \to f(K))$$

and as $\grave{\varepsilon}f(\varepsilon) = K$ by definition of K,

$$\forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \to \varphi(K)] \longrightarrow f(K).$$

Using the definition of f, we obtain

$$\forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \rightarrow \varphi(K)] \longrightarrow \neg\forall\varphi[\grave{\varepsilon}\varphi(\varepsilon) = K \rightarrow \varphi(K)],$$

hence by reductio ad absurdum,

$$\neg\forall\varphi[\grave{\alpha}\varphi(\alpha) = K \rightarrow \varphi(K)],$$

or shorthand:

$$f(K). \tag{4}$$

Applying (3) results in

$$\forall\varphi[\grave{\alpha}\varphi(\alpha) = K \rightarrow \varphi(K)],$$

which implies

$$\neg\neg\forall\varphi[\grave{\alpha}\varphi(\alpha) = K \rightarrow \varphi(K)],$$

or shorthand:

$$\neg f(K). \tag{5}$$

(4) and (5) contradict each other.

1b3 How wrong was Frege?

In the history of the Russell Paradox, Frege is often depicted as the pitiful person whose system was inconsistent. This suggests that Frege's system was the only one that was inconsistent, and that Frege was very inaccurate in his writings. On these points, history does Frege an injustice.

In fact, Frege's system was much more accurate than other systems of those days. Peano's work, for instance, was less precise on several points:

- Peano hardly paid any attention to logic, especially not to quantification theory;
- Peano did not make a strict distinction between his symbolism and the objects underlying this symbolism. Frege was much more accurate on this point (see also his paper *Über Sinn und Bedeutung* [49]);

- Frege made a strict distinction between a proposition (as an object of interest or discussion) and the *assertion* of a proposition. Frege denoted a proposition, in general, by $-A$, and the assertion of the proposition by $\vdash A$. The symbol $\vdash$ is still widely used in logic and type theory. Peano did not make this distinction and simply wrote A.

Nevertheless, Peano's work was very popular, for several reasons:

- Peano had able collaborators, and in general had a better eye for presentation and publicity. For instance, he bought his own press, so that he could supervise the printing of his journal *Rivista di Matematica* and *Formulaire* [98];
- Peano used a symbolism much more familiar to the notations that were used in those days by mathematicians (and many of his notations, like $\in$ for "is an element of", and $\supset$ for logical implication, are also used in Russell's *Principia Mathematica*, and are actually still in use).

Frege's work did not have these advantages and was hardly read before 1902[3]. In the last paragraph of [50], Frege concluded:

> "... I observe merely that the Peano notation is unquestionably more convenient for the typesetter, and in many cases takes up less room than mine, but that these advantages seem to me, due to the inferior perspicuity and logical defectiveness, to have

[3]When Peano published his formalisation of mathematics in 1889 [97] he clearly did not know Frege's *Begriffsschrift*, as he did not mention the work, and was not aware of Frege's formalisation of quantification theory. Peano considered quantification theory to be "abstruse" in [98], on which Frege proudly reacted:

> "In this respect my conceptual notion of 1879 is superior to the Peano one. Already, at that time, I specified all the laws necessary for my designation of generality, so that nothing fundamental remains to be examined. These laws are few in number, and I do not know why they should be said to be abstruse. If it is otherwise with the Peano conceptual notation, then this is due to the unsuitable notation."
>
> ([50], p. 376)

been paid for too dearly — at any rate for the purposes I want to pursue."

(*Ueber die Begriffschrift des Herrn Peano und meine eigene*, p. 378)

Frege's system was not the only paradoxical one. The Russell Paradox can be derived in Peano's system as well, by defining the class

$$K \stackrel{\text{def}}{=} \{x \mid x \notin x\}$$

and deriving $K \in K \longleftrightarrow K \notin K$. In Cantor's Set Theory one can derive the paradox via the same class (or *set*, in Cantor's terminology).

1b4 The importance of Russell's Paradox

Russell's paradox was certainly not the first or only paradox in history. Paradoxes were already widely known in antiquity. The first known paradox is the Achilles paradox of Zeno of Elea. It is a purely mathematical paradox. Due to a precise formulation of mathematics and especially the concept of real numbers, the paradox can now be satisfactorily solved.

The oldest logical paradox is probably the Liar's Paradox, also known as the Paradox of Epimenides. It can be very shortly formulated by the sentence "This sentence is not true". The paradox was widely known in antiquity. For instance, it is referred to in the Bible (Titus 1:12). It is based on the confusion between language and meta-language.

The Burali-Forti paradox ([24], 1897) is the first of the modern paradoxes. It is a paradox within Cantor's theory on ordinal numbers. Cantor's paradox on the largest cardinal number occurs in the same field. It must have been discovered by Cantor around 1895, but was not published before 1932.

The logicians considered these paradoxes to be out of the scope of logic: the paradoxes based on the Liar's Paradox could be regarded as a problem of linguistics, and the paradoxes of Cantor and Burali-Forti occurred in a in those days highly questionable part of mathematics: Cantor's Set Theory.

The Russell Paradox, however, was a paradox that could be formulated in all the systems that were presented at the end of the 19th century (except for Frege's *Begriffsschrift*). It was at the very basics of logic. It could not be disregarded, and a solution to it had to be found.

Chapter 2

Type theory in Principia Mathematica

When Russell proved Frege's *Grundgesetze* to be inconsistent, Frege was not the only person in trouble. In Russell's letter to Frege (1902), we read:

> "I am on the point of finishing a book on the principles of mathematics"
>
> (*Letter to Frege*, [106])

Therefore, Russell had to find a solution to the paradoxes, before he could finish his book.

His paper *Mathematical logic as based on the theory of types* [108] (1908), in which a first step is made towards the Ramified Theory of Types, started with a description of the most important contradictions that were known up till then, including Russell's own paradox. He then concluded:

> "In all the above contradictions there is a common characteristic, which we may describe as self-reference or reflexiveness. [...] In each contradiction something is said about *all* cases of some kind, and from what is said a new case seems to be generated, which both is and is not of the same kind as the cases of which *all* were concerned in what was said."
>
> (*Ibid.*)

Russell's plan was, therefore, to avoid the paradoxes by avoiding all possible self-references. He postulated the "vicious circle principle":

Vicious Circle Principle 2.1

> "*Whatever involves* all *of a collection must not be one of the collection.*"
>
> ([81], p. 28)

Russell applies this principle very strictly. He implemented it using types, in particular the so-called *ramified* types. The theory presented in *Mathematical logic as based on the theory of types* was elaborated in Chapter II of the Introduction to the famous *Principia Mathematica* [121] (1910-1912). In the *Principia*, Whitehead and Russell founded mathematics on logic, as far as possible. The result was a very formal and accurate build-up of mathematics, avoiding the logical paradoxes.

The logical part of the *Principia* was based on the works of Frege. This was acknowledged by Whitehead and Russell in the preface, and can also be seen throughout the description of Type Theory. The notion of function is based on Frege's Abstraction Principles 1.1 and 1.2, and the *Principia* notation $\hat{x}f(x)$ for a class looks very similar to Frege's $\grave{\varepsilon}f(\varepsilon)$ for course-of-values.

An important difference is that Whitehead and Russell treated functions as first-class citizens. Frege used courses-of-values as a way of speaking about functions (and was confronted with a paradox); in the *Principia* a direct approach was possible. Equality, for instance, was defined for objects as well as for functions by means of Leibniz equality ($x = y$ if and only if $f(x) \leftrightarrow f(y)$ for all propositional functions f — see [121], ∗13·11).

The description of the Ramified Theory of Types (RTT) in the *Principia* was, though extensive, still informal. It is clear that Type Theory had not yet become an independent subject. The theory

> "only recommended itself to us in the first instance by its ability to solve certain contradictions"
>
> (*Principia Mathematica*, p. 37)

And though

> "it has also a certain consonance with common sense which makes it inherently credible"
>
> (*Principia Mathematica*, p. 37)

(probably, Whitehead and Russell refer to the implicit, intuitive use of types by mathematicians. See Section 1a), Type Theory was not introduced because it was interesting on its own, but because it had to serve as a tool for logic and mathematics. A formalisation of Type Theory, therefore, was not considered in those days.

Though the description of RTT in the *Principia* was still informal, it was clearly present throughout the work. It was not mentioned very often, but when necessary, Russell made a remark on RTT. This is an important difference with the earlier writings of Frege, Peano and Cantor.

If we want to compare RTT with contemporary type systems, we have to make a formalisation of RTT. Though there are many descriptions of RTT available in the literature (like [30], [32], [64], [101] and Section 27 of [110]), none of these descriptions presents a formalisation that is both accurate and as close as possible to the ideas of the *Principia*. We will fill up this gap in the literature in the first part of this chapter.

Making such a formalisation is by no means easy:

- Important formal notions, especially the notion of substitution, remained completely unexplained in the *Principia*;

- The accuracy of Frege's work was not present in Russell's. This was already observed by Gödel, who said that the precision of Frege was lost in the writings of Russell, and who, due to the informality of some basic notions of the *Principia*, had to give his paper [57] the title *Über formal unentscheidbare Sätze der Principia Mathematica und* verwandter *Systeme*.

In 1b1 we saw that Frege generalised the notion of function from analysis. For Russell's formalisation of mathematics within logic, a special kind of these functions was needed: the so-called *propositional* functions. A propositional function (pf) always returns a *proposition* when it is applied to suitable arguments. In Section 2a, we introduce a formalised version of these pfs. This makes it possible to compare pfs with other formal systems, like λ-calculus, and to give a precise definition of substitution.

In Section 2b we give a formalisation of Russell's notion of ramified type (2b1), followed by a formal definition of the notion *the pf f is of type t* (2b2). We motivate this definition (2b3) by referring to passages in the *Principia*. As the formalisation of pf is precise enough to be translated to λ-calculus, we can make a comparison between RTT and current type systems.

Thanks to our formal notation and its relation with λ-calculus, we are able to prove properties of RTT in an easy way, using properties of modern type systems. This will be done in Section 2c. Due to the new notation it is relatively easy to see that we have proved variants of well-known theorems from Type Theory, like Strong Normalisation, Free Variable Lemma, Strengthening Lemma, Unicity of Types and Subterm Lemma.

In Section 2d we answer in full detail the question which pfs are typable. We also make a comparison between our notion of typable pf, and the corresponding notion in the *Principia*, and conclude that these two notions of typable pf coincide.

Parts of this chapter have been taken from earlier publications: [79], [80] and [82].

2a Propositional functions

In this section we present a formalisation of the propositional functions (pfs) of the *Principia*. In Section 2a1 we give a syntax that is as close as possible to the ideas of the *Principia*. Intuition about this syntax is provided in Section 2a2 by translating pfs into λ-terms. In Section 2a3 we define some related notions that are needed in the rest of the Chapter. We devote a special section 2a4 to the notion of substitution. This notion is clearly present in the *Principia*, but not formally defined. Due to the translation to λ-calculus of Section 2a2, we are able to give a precise definition.

2a1 Definition

The definition of propositional function in the *Principia* is as follows:

"By a "propositional function" we mean something which contains a variable x, and expresses a *proposition* as soon as a value is assigned to x."

(*Principia Mathematica*, p. 38)

Pfs are, however, *constructed* from propositions with the use of the Abstraction Principles: they arise when in a proposition one or more occurrences of a sign are replaced by a variable. Therefore we have to begin our formalisation with certain basic propositions, certain basic signs, and signs that indicate a replaceable object. For this purpose we use

- A set $\mathcal{A}$ of *individual symbols* (the basic signs);
- A set $\mathcal{V}$ of *variables* (the signs that indicate replaceable objects);
- A set $\mathcal{R}$ of *relation symbols* together with a map $\mathfrak{a} : \mathcal{R} \rightarrow \mathbb{N}^+$ indicating the *arity* of each relation-symbol (these are used to form the basic propositions).

We want to have a sufficient supply of individual symbols, variables and relation symbols and therefore assume that $\mathcal{A}$ and $\mathcal{V}$ are infinite (but countable), and that $\{R \in \mathcal{R} \mid \mathfrak{a}(R) = n\}$ is infinite (but countable) for each $n \in \mathbb{N}^+$. We assume that $\{\mathtt{a}_1, \mathtt{a}_2, \dots\} \subseteq \mathcal{A}$, $\{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{x}_1, \dots\} \subseteq \mathcal{V}$ and $\{\mathtt{R}, \mathtt{S}, \dots\} \subseteq \mathcal{R}$. We use $a_1, a_2, \dots$ as metavariables over $\mathcal{A}$; $x, y, z, x_1, \dots$ as metavariables over $\mathcal{V}$ and $R, S, \dots$ as metavariables over $\mathcal{R}$. For technical reasons we assume that there is an *order* (e.g. alphabetical) on $\mathcal{V}$. We write $x < y$ if x is ordered before y, and not equal to y (so: $<$ is *strict*). In particular, we assume that

$$\mathtt{x} < \mathtt{x}_1 < \dots \mathtt{y} < \mathtt{y}_1 < \dots \mathtt{z} < \mathtt{z}_1 \dots$$

and: for each x there is a y with $x < y$.

Definition 2.2 (Atomic propositions) A list of symbols of the form $R(a_1, \dots, a_{\mathfrak{a}(R)})$ is called an *atomic proposition*.

Other names used for these atomic propositions in the *Principia* are *elementary judgements* and *elementary propositions* (cf. [121], pp. xv, 43–45, and 91).

Propositional functions in *Principia Mathematica* are generated from atomic propositions by two means:

- The use of logical connectives and quantifiers;
- Abstraction from (earlier generated) propositional functions, using the abstraction principles.

This leads to the following formal definition of propositional function. Examples are given in 2.5 and intuition is provided in Section 2a2.

Definition 2.3 (Propositional functions) We define a collection $\mathcal{P}$ of *propositional functions* (pfs), and for each element f of $\mathcal{P}$ we simultaneously define the collection $\textsc{fv}(f)$ of *free variables* of f:

1. If $i_1, \ldots, i_{\mathfrak{a}(R)} \in \mathcal{A} \cup \mathcal{V}$ then $R(i_1, \ldots, i_{\mathfrak{a}(R)}) \in \mathcal{P}$.
   $\textsc{fv}(R(i_1, \ldots, i_{\mathfrak{a}(R)})) \stackrel{\text{def}}{=} \{i_1, \ldots, i_{\mathfrak{a}(R)}\} \cap \mathcal{V}$;
2. If $f, g \in \mathcal{P}$ then $f \vee g \in \mathcal{P}$ and $\neg f \in \mathcal{P}$.
   $\textsc{fv}(f \vee g) \stackrel{\text{def}}{=} \textsc{fv}(f) \cup \textsc{fv}(g)$; $\textsc{fv}(\neg f) \stackrel{\text{def}}{=} \textsc{fv}(f)$;
3. If $f \in \mathcal{P}$ and $x \in \textsc{fv}(f)$ then $\forall x[f] \in \mathcal{P}$.
   $\textsc{fv}(\forall x[f]) \stackrel{\text{def}}{=} \textsc{fv}(f) \setminus \{x\}$;
4. If $n \in \mathbb{N}$ and $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, then $z(k_1, \ldots, k_n) \in \mathcal{P}$.
   $\textsc{fv}(z(k_1, \ldots, k_n)) \stackrel{\text{def}}{=} \{z, k_1, \ldots, k_n\} \cap \mathcal{V}$.
   If $n = 0$ then we write $z()$ in order to distinguish the pf $z()$ from the variable z[1];
5. All pfs can be constructed by using the construction-rules 1, 2, 3 and 4 above.

We use the letters f, g, h as meta-variables over $\mathcal{P}$.

Definition 2.4 (Propositions) A propositional function f is a *proposition* if $\textsc{fv}(f) = \varnothing$.

Example 2.5 We give some examples of (higher-order) pfs of the form $z(k_1, \ldots, k_n)$ in ordinary mathematics. To keep the link with mathematics clear, we use some extra logical connectives like $\leftrightarrow$ and $\rightarrow$.

[1]It is important to note that a variable is not a pf. See for instance [107], Chapter VIII: "The variable", p. 94 of the 7th impression.

1. The pfs $\mathsf{z(x)}$ and $\mathsf{z(y)}$ in the definition of equality according to Leibniz: By definition $\mathsf{x} = \mathsf{y}$ if and only if
$$\forall \mathsf{z}[\mathsf{z(x)} \leftrightarrow \mathsf{z(y)}];$$
2. The pfs $\mathsf{z(0)}$, $\mathsf{z(x)}$ and $\mathsf{z(y)}$ in the formulation of the principle of mathematical induction:
$$\begin{aligned}\forall \mathsf{z}[\mathsf{z(0)} &\to (\forall \mathsf{x}\forall \mathsf{y}[\mathsf{z(x)} \to (\mathsf{S(x,y)} \to \mathsf{z(y)})]) \\ &\to \forall \mathsf{x}[\mathsf{z(x)}]]\end{aligned}$$
(we suppose that the relation symbol S represents the successor function: $\mathsf{S}(x,y)$ holds if and only if y is the successor of x);
3. $\mathsf{z()}$ in the formulation of the law of the excluded middle:
$$\forall \mathsf{z}[\mathsf{z()} \vee \neg \mathsf{z()}].$$

2a2 Propositional functions as λ-terms

The binding structure and the notion of free variable of pfs become more clear if we translate pfs to λ-terms. Moreover, such a translation will be useful at several places in this Chapter, for instance when we give a definition of substitution.

We first translate one of the examples of Example 2.5. Then we give a formal definition of the translation that we have in mind. After that we provide additional remarks and intuition on pfs.

Example 2.6 Consider the pf $f \equiv \forall \mathsf{z}[\mathsf{z(x)} \leftrightarrow \mathsf{z(y)}]$ of Example 2.5.1. Two objects x and y are Leibniz-equal if and only if they share the same properties. These objects are represented by the variables x and y. The variable z is a variable for properties of objects, in other words: predicates over objects. Such a predicate is a function that takes the object as argument, and returns a truth value. The expression $\mathsf{z(x)}$ indicates that the predicate that is taken for z must be applied to the object that is taken for x. Therefore, we translate $\mathsf{z(x)}$ by an application of z to x in λ-calculus: zx. Similarly we translate the expression $\mathsf{z(y)}$ by zy.

Just as in [30], we can interpret logical connectives as functions. Therefore we can translate $\mathbf{z}(\mathbf{x}) \leftrightarrow \mathbf{z}(\mathbf{y})$ by the λ-term $\leftrightarrow(\mathbf{zx})(\mathbf{zy})$. We handle the translation of universal quantification also as in [30], hence $\forall\mathbf{z}[\dots]$ translates to $\forall(\lambda\mathbf{z}.\dots)$. As an effect we get a λ-term $\forall(\lambda\mathbf{z}.\leftrightarrow(\mathbf{zx})(\mathbf{zy}))$ with two free variables, $\mathbf{x}$ and $\mathbf{y}$. But we want to have a *function* taking two *arguments*. This can be solved by a double λ-abstraction. The final result is $\lambda\mathbf{x}.\lambda\mathbf{y}.\forall(\lambda\mathbf{z}.(\leftrightarrow(\mathbf{zx})(\mathbf{zy})))$.

We remark that the pf f has two free variables, $\mathbf{x}$ and $\mathbf{y}$. These two free variables correspond to the two arguments that the propositional function takes, and therefore to the two λ-abstractions that are at the front of the translation of f.

In the following definition, we translate the propositional functions to λ-terms in a similar way as we did in Example 2.6. Let $f \in \mathcal{P}$ and let $x_1 < \cdots < x_m$ be the free variables of f. We define a λ-term $\overline{f}$. We do this in such a way that $\overline{f} \equiv \lambda x_1.\cdots\lambda x_m.F$, where F is a λ-term that is not of the form $\lambda x.F'$. To keep notations uniform, we also give translations $\overline{a}$ for $a \in \mathcal{A}$ and $\overline{x}$ for $x \in \mathcal{V}$. To keep notations short, we use $\lambda_{i=1}^{m} x_i.F$ as shorthand for $\lambda x_1.\cdots\lambda x_m.F$.

Definition 2.7

- $\overline{a} \stackrel{\text{def}}{=} a$ for $a \in \mathcal{A}$;
- $\overline{x} \stackrel{\text{def}}{=} x$ for $x \in \mathcal{V}$;
- Now assume $f \in \mathcal{P}$ has free variables $x_1 < \cdots < x_m$. Use induction on the structure of f:
  - $f \equiv R(i_1, \dots, i_{\mathfrak{a}(R)})$. Then $\overline{f} \stackrel{\text{def}}{=} \lambda_{i=1}^{m} x_i.Ri_1 \cdots i_{\mathfrak{a}(R)}$;
  - $f \equiv f_1 \vee f_2$. We can assume that for $j = 1,2$, $\overline{f_j} \equiv \lambda_{i=1}^{m_j} y_i^j.F_j$, where $y_1^j < \cdots < y_{m_j}^j$ are the free variables of f_j.
    Then $\overline{f} \stackrel{\text{def}}{=} \lambda_{i=1}^{m} x_i.\vee F_1 F_2$.
    If $f \equiv \neg f'$ then we can assume that $\overline{f'} \equiv \lambda_{i=1}^{m} x_i.F$, because $x_1 < \cdots < x_m$ are the free variables of f'. Let $\overline{f} \stackrel{\text{def}}{=} \lambda_{i=1}^{m} x_i.\neg F$;
  - $f \equiv z(k_1, \dots, k_n)$. Let $\overline{f} \equiv \lambda_{i=1}^{m} x_i.z\overline{k_1} \cdots \overline{k_n}$;

- $f \equiv \forall x[f']$. We can assume that $\overline{f'} \equiv \lambda_{i=1}^{j-1} x_i.\lambda x.\,\lambda_{i=j}^{m} x_i.F$, because $x_1, \ldots, x_m, x$ are the free variables of f'.
  Define $\overline{f} \equiv \lambda_{i=1}^{m} x_i.\forall(\lambda x.F)$.

Example 2.8

f	$\overline{f}$
R(x)	λx.Rx
z(R(x), S(a))	λz.z(λx.Rx)(Sa)
z_1(a) $\vee$ z_2()	$\lambda z_1.\lambda z_2.\vee(z_1 a)z_2$
z(y(R(x)))	λz.z(λy.y(λx.Rx))
$\forall$x[R(x)]	$\forall$(λx.Rx)

By induction on the structure of f one can prove the following properties of $\overline{f}$:

Lemma 2.9 (Properties of $\overline{\ }$) *Let $f \in \mathcal{P}$.*

1. $\text{FV}(\overline{f}) = \varnothing$;
2. $\overline{f}$ *is in β-normal form;*
3. $\overline{f}$ *is a λI-term;*
4. *If $x_1 < \cdots < x_m$ are the free variables of f, then $\overline{f} \equiv \lambda_{i=1}^{m} x_i.F$, where F is not of the form $\lambda x.F'$.*

$\boxtimes$

Observe that we use FV for indicating both the free variables of a pf and the free variables of a λ-term. We take care that it will always be clear in which meaning we use FV.

We make some remarks on the definition of propositional function 2.3.

Remark 2.10 We show that the propositional functions of Definition 2.3 are indeed objects that exist in the theory of Russell.

1. In Rule 1 we describe the atomic propositions, and the atomic propositions in which one or more individuals have been replaced by variables due to one or more applications of the abstraction principles. The abstraction principles are not only present in the works of Frege, but also in the *Principia* (cf. for instance *9·14 and *9·15);

2. Rule 2 describes the use of the logical connectives $\vee$ and $\neg$. These logical connectives are also used in the *Principia*. Implication[2], conjunction[3] and logical equivalence[4] are defined in terms of negation and disjunction. In examples, we sometimes use symbols for implication, conjunction and logical equivalence as abbreviations;

3. Rule 3 describes the use of the universal quantifier. It is explicitly stated in the *Principia* (cf. pp. 14–16) that the pf $\forall x[f]$ can only be constructed if f is a pf that contains x as a variable. Existential quantification[5] is defined in terms of negation and universal quantification;

4. Rule 4 is also an instantiation of the abstraction principle. The pfs that can be constructed by using the construction-rules 1–3 only are exactly the pfs of what in these days would be called first-order predicate logic. With rule 4, higher-order pfs can be constructed. This is based on the following idea. Let f be a (fixed) pf in which $k_1, \ldots, k_n$ occur. We can interpret f as an *instantiation* of a function that has taken arguments $k_1, \ldots, k_n$. We now generalise this to $z(k_1, \ldots, k_n)$, representing *any* function z taking these arguments. Such a construction is also explicitly present in the *Principia*:

   > "the first matrices[6] that occur are those whose values are of the forms $\varphi x, \psi(x,y), \chi(x,y,z,\ldots)$, *i.e.* where the arguments, however many there may be, are all individuals. Such [propositional] functions we will call '*first-order* functions.' We may now introduce a notation to express 'any first-order function.' "
   >
   > (*Principia Mathematica*, p. 51)

Remark 2.11 The definition of free variable needs some special attention. We must notice that, for instance,

$$\mathrm{FV}(\mathtt{z}(\mathtt{R}(\mathtt{x}),\mathtt{S}(\mathtt{a}))) = \{\mathtt{z}\}$$

[2]cf. *Principia*, ∗1·01, p. 94
[3]cf. *Principia*, ∗3·01, p. 107
[4]cf. *Principia*, ∗4·01, p. 117
[5]cf. *Principia*, ∗10·01, p. 140
[6]see Remark 2.13 [footnote of the author].

and not $\{x, z\}$. The reason for this is that the notion of free variable should harmonise with the intuitive notion of "argument place" of Frege and Russell. As was indicated in Remark 2.10.4, z represents an arbitrary function that takes $R(x)$ and $S(a)$ as arguments and returns a proposition. This means that we do not have to supply an argument for x "by hand". As soon as we feed a suitable[7] argument f to z in $z(R(x), S(a))$, f will take the arguments $R(x)$ and $S(a)$, and return a proposition.

This idea is also clearly reflected in the translation of $z(R(x), S(a))$ to the λ-term $\lambda z.z(\lambda x.Rx)(Sa)$. The variable x is bound in a subterm $\lambda x.Rx$ that is an argument to the variable z. The full λ-term is a function of z only.

Remark 2.12 In the Introduction we suggested that functionalisation can be represented in λ-calculus by first making a β-expansion, and then removing the argument. The translation of Definition 2.7 enables us to show that functionalisation in the theories of Russell and Frege (using the Abstraction Principles) is similar to functionalisation in λ-calculus. We do this by giving some examples. Consider the pf $R(a) \vee S(a)$. There are several ways to apply the abstraction possibilities to this pf:

1. The list of symbols $R(a)$ can be seen as an instance of the pf $z(a)$. $z(a)$ is a pf that takes a unary propositional function as an argument and returns the value of that pf for the argument a. Applying abstraction to $R(a)$ in $R(a) \vee S(a)$ results in $z(a) \vee S(a)$. In λ-calculus: expand the λ-term $\vee(Ra)(Sa)$ to $\vee((\lambda x.Rx)a)(Sa)$, and then to $(\lambda z.\vee(za)(Sa))(\lambda x.Rx)$. Remove the argument $\lambda x.Rx$ and we have the translation of $z(a) \vee S(a)$;

2. But one could also consider the expression $R(a) \vee S(a)$ as an instance of the pf that takes two propositions and returns their disjunction: the pf $z_1() \vee z_2()$. In λ-calculus: $\vee(Ra)(Sa)$ β-expands to $(\lambda z_2.\vee(Ra)z_2)(Sa)$. Removing the argument gives $\lambda z_2.\vee(Ra)z_2$. A similar operation on Ra results in $\lambda z_1.\lambda z_2.\vee z_1 z_2$;

3. More abstract: One could consider $R(a) \vee S(a)$ as an instance of the pf $z(R(a), S(a))$ for the argument $z_1() \vee z_2()$. The pf $z(R(a), S(a))$

[7] At this stage, we cannot provide a formalisation of "suitable". This can only be done after we have introduced types, and formalised the notion "the pf f is of type t"

takes one argument, say f, (for the variable z). Such an argument f in its turn, needs to be a pf taking two propositions as arguments. In λ-calculus, $\vee(\mathrm{Ra})(\mathrm{Sa})$ expands to $(\lambda z.z(\mathrm{Ra})(\mathrm{Sa}))\vee$. Removing the argument gives $\lambda z.z(\mathrm{Ra})(\mathrm{Sa})$.

Applying $z(R(a), S(a))$ to f results in the pf f, evaluated for the arguments $R(a)$ and $S(a)$;

4. And even more mind-bogglingly (for persevering readers only): $R(a) \vee S(a)$ is an instance of the pf $z(R(x), S(a))$ for the argument $z_1(a) \vee z_2()$. The pf $z(R(x), S(a))$ takes one argument (for z). Such an argument f in its turn, must be a pf that takes one pf and one proposition as arguments. The pf-argument of f must take one individual as an argument. If we evaluate $z(R(x), S(a))$ for the argument f, then we get as a result the value of f for the arguments $R(x)$ and $S(a)$.

   In λ-calculus: $\vee(\mathrm{Ra})(\mathrm{Sa})$ β-expands to $\vee((\lambda x.\mathrm{R}x)a)(\mathrm{Sa})$, then to

$$(\lambda z_1.\lambda z_2.\vee(z_1 a)z_2)(\lambda x.\mathrm{R}x)(\mathrm{Sa}),$$

   and finally to

$$(\lambda z.z(\lambda x.\mathrm{R}x)(\mathrm{Sa}))(\lambda z_1.\lambda z_2.\vee(z_1 a)z_2).$$

   Removing the argument gives $\lambda z.z(\lambda x.\mathrm{R}x)(\mathrm{Sa})$.

   Let us check what we have done by evaluating $z(R(x), S(a))$ for the argument $z_1(a) \vee z_2()$. Above, we argued that $R(a) \vee S(a)$ is an instance of $z(R(x), S(a))$ for the argument $z_1(a) \vee z_2()$, so as a result we must obtain $R(a) \vee S(a)$.

   According to the description of $z(R(x), S(a))$ above, assigning the value $z_1(a) \vee z_2()$ to z is equal to the value of $z_1(a) \vee z_2()$ for the arguments $R(x)$ and $S(a)$. Substituting $R(x)$ for z_1 gives the value of $R(x)$ for the argument a: $R(a)$. Substituting $S(a)$ for z_2 gives the proposition $S(a)$. The final result is indeed $R(a) \vee S(a)$. This calculation can also be carried out in λ-calculus by applying $\lambda z.z(\lambda x.\mathrm{R}x)(\mathrm{Sa}))$ to $\lambda z_1.\lambda z_2.\vee(z_1 a)z_2)$ and reducing to β-normal form.

All these abstractions are in line with 1.1 and 1.2 on page 18. Via the abstraction in the last two examples one also obtains pfs of the form

$z(k_1, \ldots, k_n)$ where some of the k_i are elements of $\mathcal{P}$. There is no formal definition of abstraction in the works of Frege and Russell. We could use a definition that is related to β-expansion. See Remark 2.29.

Remark 2.13 It appears that there is also an alternative way of constructing pfs in the *Principia.* Whitehead and Russell distinguish between quantifier-free pfs (so-called *matrices*, i.e. the pfs that can be constructed using construction-rules 1, 2 and 4). Then they form pfs by defining that

- Any matrix is a pf;
- If f is a pf and $x \in \text{FV}(f)$ then $\forall x[f]$ is a pf with free variables $\text{FV}(f) \setminus \{x\}$.

This definition is a little different from our Definition 2.3, as a pf of the form $z(\forall x[f])$ is not a matrix and therefore not a pf according to this alternative definition. Nevertheless we feel that Whitehead and Russell intended to give our Definition 2.3. In the *Principia* ([121], $*54$) they define the natural number 0 as the propositional function $\forall \mathrm{x}[\neg \mathrm{z}(\mathrm{x})]$[8]. In defining the principle of induction on natural numbers, one needs to express the property "0 has the property y", or: $y(0)$. But $y(0)$ is not a pf according to this alternative definition, as 0 contains quantifiers.

Therefore we feel that our Definition 2.3, which is also based on the definition of function by Frege and on the definition of propositional function on p. 38 of the *Principia*, is the definition that was meant by Whitehead and Russell.

Remark 2.14 Note that pfs as such do not yet obey to the vicious circle principle 2.1! For example, $\neg\mathrm{z}(\mathrm{z})$ (the pf that is at the basis of the Russell paradox) is a pf. In Section 2b we will assign types to some pfs, and it will be shown (Remark 2.66) that no type can be assigned to the pf $\neg\mathrm{z}(\mathrm{z})$.

[8]This definition is based on Frege's definition in *Grundlagen der Arithmetik* [46] (1884). See [121], vol. II, p. 4. In [46], the natural number n is defined as the class of predicates f for which there are exactly n objects a for which $f(a)$ holds. Hence 0 is the class of predicates f for which $f(a)$ does not hold for any object a. So 0 can be described by the pf $\forall \mathrm{x}[\neg \mathrm{z}(\mathrm{x})]$

Remark 2.15 Before we make further developments of the theory based on pfs, we must decide which of the two syntaxes introduced above shall be used in the sequel. It looks attractive to use the syntax of λ-calculus:

- This syntax is well-known;
- It is used for many other type systems, so it makes the comparison of ramified type theory with modern type systems easier;
- There is a lot of meta-theory on typed and untyped λ-calculus. This can be useful when proving certain properties of the formalisation of the ramified theory of types that is to be introduced in the next sections;
- The syntax of λ-calculus gives a better look on the notion of free variable than the syntax of pfs.

Nevertheless, we shall only indirectly use λ-calculus for our further study of the ramified type theory in this Chapter. We have several reasons for that:

- There are much more λ-terms than there are pfs. More precise, the mapping $\bar{}$ is not surjective. As we want to study the theory of *Principia Mathematica* as precise as possible, we only want to study the propositional functions, which are directly related to the syntax used by Russell and Whitehead. Not using pf-syntax may result in a system in which it is not clear which term belongs to the original ramified type theory and which term does not;
- The syntax of λ-calculus is strongly curried. This would give problems in the definition of substitution. In a pf $\mathtt{R}(\mathtt{x},\mathtt{y})$ we may want to substitute some object a for $\mathtt{y}$ without substituting anything for $\mathtt{x}$. In λ-calculus, substitution should be translated to application followed by β-reduction to β-normal form. If we want to substitute something for $\mathtt{y}$ in the translation $\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{Rxy}$ of $\mathtt{R}(\mathtt{x},\mathtt{y})$, we have to substitute something for $\mathtt{x}$ first. Choosing a different representation of propositional functions does not help: the representation $\lambda\mathtt{y}.\lambda\mathtt{x}.\mathtt{Ryx}$ would have given problems if we wanted to substitute something for $\mathtt{x}$ without substituting something for $\mathtt{y}$;

- The translation of pfs to λ-calculus makes it possible to use the meta-theory and the intuition of λ-calculus when we need it without losing control over the original system.

2a3 Related notions

We proceed our discussion of pfs by defining a number of related notions. If a pf $z(k_1, \ldots, k_n)$ takes an argument f for the variable z, the list $k_1, \ldots, k_n$ indicates what should be substituted for the free variables of f (cf. also Remark 2.10.4). We therefore call this list the list of *parameters* of $z(k_1, \ldots, k_n)$. A formal definition:

Definition 2.16 (Parameters) Assume f is a pf, and $k \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$. We define, inductively, the notion *k is a parameter of f*, and write $\text{PAR}(f)$ for the set of parameters of f.

- $\text{PAR}\big(R(i_1, \ldots, i_{\mathfrak{a}(R)})\big) \stackrel{\text{def}}{=} \{i_1, \ldots, i_{\mathfrak{a}(R)}\}$;
- $\text{PAR}(f_1 \vee f_2) \stackrel{\text{def}}{=} \text{PAR}(f_1) \cup \text{PAR}(f_2)$ and $\text{PAR}(\neg f) \stackrel{\text{def}}{=} \text{PAR}(f)$;
- $\text{PAR}(\forall x[f]) \stackrel{\text{def}}{=} \text{PAR}(f)$;
- $\text{PAR}(z(k_1, \ldots, k_n)) \stackrel{\text{def}}{=} \{k_1, \ldots, k_n\}$.

Note that x is not a parameter of $\mathtt{z}(\mathtt{R}(\mathtt{x}), \mathtt{S}(\mathtt{a}))$, but it is a *recursive* parameter according to the following definition:

Definition 2.17 (Recursive parameters) Assume f is a pf. We define, inductively, the set of *recursive parameters* of f, $\text{RP}(f)$:

- $\text{RP}\big(R(i_1, \ldots, i_{\mathfrak{a}(R)})\big) \stackrel{\text{def}}{=} \{i_1, \ldots, i_{\mathfrak{a}(R)}\}$;
- $\text{RP}(\neg f) \stackrel{\text{def}}{=} \text{RP}(f)$; $\text{RP}(f_1 \vee f_2) \stackrel{\text{def}}{=} \text{RP}(f_1) \cup \text{RP}(f_2)$;
- $\text{RP}(\forall x[f]) \stackrel{\text{def}}{=} \text{RP}(f)$;
- $\text{RP}(z(k_1, \ldots, k_n)) \stackrel{\text{def}}{=} \{k_1, \ldots, k_n\} \cup \bigcup_{k_i \in \mathcal{P}} \text{RP}(k_i)$.

Another important notion is the notion of α-equality[9]. We want the pfs R(x) and R(y) to be the same. However, we want the pfs S(x, y) and S(y, x) to be different. The reason for this is the alphabetical order of the variables x, y. As $x < y$, we will consider x to be the "first" variable of the pfs S(x, y) and S(y, x), and y the "second" variable . The place of the "first" variable in S(x, y), however, is different from the place of the "first" variable in S(y, x).[10] We therefore present the following definition of α-equality:

Definition 2.18 **(α-equality)** Let f and g be pfs. We say that f and g are *α-equal*, notation $f =_\alpha g$, if there is a bijection $\varphi : \mathcal{V} \to \mathcal{V}$ such that

- g can be obtained from f by replacing each variable that occurs in f by its φ-image;
- $x < y$ iff $\varphi(x) < \varphi(y)$.

This definition corresponds to the definition of α-equality in λ-calculus in the following way:

Lemma 2.19 *Let $f, g \in \mathcal{P}$. $f =_\alpha g$ if and only if $\overline{f} =_\alpha \overline{g}$.* ⊠

Sometimes, we are not that precise, and want the pfs S(x, y) and S(y, x) to be α-equal. This can be a consideration especially if we are not interested in which free variable is "first" and which is "second". We call this weakened notion of α-equality: α_P-equality (α-equality modulo permutation):

Definition 2.20 **(α-equality modulo permutation)** Let f and g be pfs. We say that f and g are *α_P-equal*, notation $f =_{\alpha_P} g$, if there is a bijection $\varphi : \mathcal{V} \to \mathcal{V}$ such that g can be obtained from f by replacing each variable that occurs in f by its φ-image.

[9]Historically, it is not correct to use this terminology when discussing Type Theory of the *Principia*, which dates from the first decade of this century. The term α-equality originates from Curry and Feys' book *Combinatory Logic* [38], which appeared only in 1958. In this book, conversion rules for the λ-calculus are numbered with Greek letters α, β. The rule numbered with α is now known as α-conversion; the rule numbered with β is now known as β-conversion. In earlier papers of Church, Rosser and Kleene, these rules were numbered with Roman capitals I, II, and the terminology α-conversion, β-conversion, was not used.

[10]Compare this with their equivalents in λ-calculus λx.λy.Sxy and λx.λy.Syx, which are not α-equal, either. We do not want to use the λ-notation for determining which variable is "first" and which is "second", for reasons to be explained in Remarks 2.15 and 2.32.

2a4 Substitution

In the Introduction we argued that instantiation is the inverse operator of function construction. In Remark 2.12 we saw that function construction in *Principia Mathematica* can be compared to β-expansion plus removing an argument in λ-calculus. This suggests that instantiation in the *Principia* must be comparable to application plus β-reduction in λ-calculus. In [79] we showed that this is indeed the case. There, we gave a laborious definition of instantiation using the syntax of and the intuition behind pfs. We showed that this definition is faithful to the original ideas of the *Principia* and that it can be imitated in λ-calculus using a translation similar to the one in Definition 2.7. This allows us to give a definition of substitution for pfs that is based on that imitation in λ-calculus.

As was argued in Remark 2.15, the mapping $f \mapsto \overline{f}$ is not perfectly suited for a definition of substitution. This was due to the currying of the λ-abstractions that are at the front of the term $\overline{f}$. We therefore take a slightly different notation and remove these front abstractions from $\overline{f}$:

Definition 2.21 Let $f \in \mathcal{P}$ with free variables $x_1 < \cdots < x_m$. Then $\overline{f} \equiv \lambda_{i=1}^{m} x_i.F$ for some λ-term F. Let $\widetilde{f} \stackrel{\text{def}}{=} F$.

Example 2.22

f	$\widetilde{f}$
$\mathsf{R(x)}$	Rx
$\mathsf{z(R(x), S(a))}$	$\mathsf{z(\lambda x.Rx)(Sa)}$
$\mathsf{z_1(a) \vee z_2()}$	$\mathsf{\vee(z_1a)z_2}$
$\mathsf{z(y(R(x)))}$	$\mathsf{z(\lambda y.y(\lambda x.Rx))}$
$\mathsf{\forall x[R(x)]}$	$\mathsf{\forall(\lambda x.Rx)}$

The mapping $f \mapsto \widetilde{f}$ has similar properties as $f \mapsto \overline{f}$ (cf. Lemma 2.9):

Lemma 2.23 (Properties of $\widetilde{\ }$)

1. $\mathrm{FV}(f) = \mathrm{FV}\left(\widetilde{f}\right)$;
2. $\widetilde{f}$ *is in* β*-normal form for all* f*;*
3. $\widetilde{f}$ *is a* λI*-term for all* f*;*

$$\begin{tikzcd}
f[x_1,\ldots,x_n{:=}g_1,\ldots,g_n] \arrow[d] \arrow[r, phantom, "="] & h \\
(\lambda x_1\cdots x_n.\widetilde{f})\overline{g_1}\cdots\overline{g_n} \arrow[r, two heads, "\mathrm{nf}"', "\beta"] & \widetilde{h} \arrow[u]
\end{tikzcd}$$

Figure 2: Substitution via β-reduction

4. $\overline{f}$ is a closure (see A.7) of $\widetilde{f}$;

5. If $\widetilde{f} =_\alpha \widetilde{g}$, then $f =_\alpha g$.

⊠

With the λ-notation we can rely on the notions of β-reduction and β-normal form to give the following definition of substitution:

Definition 2.24 (Substitution) Let $f \in \mathcal{P}$, assume $x_1,\ldots,x_n$ are distinct variables, and $g_1,\ldots,g_n \in \mathcal{A}\cup\mathcal{V}\cup\mathcal{P}$. Assume that the λ-term

$$(\lambda x_1\cdots x_n.\widetilde{f})\overline{g_1}\cdots\overline{g_n}$$

has a β-normal form H. Assume $h \in \mathcal{P}$ such that $\widetilde{h} \equiv H$ (If such an h exists, it is unique due to Lemma 2.23.5). Then $f[x_1,\ldots,x_n{:=}g_1,\ldots,g_n] \stackrel{\mathrm{def}}{=} h$.

We sometimes abbreviate $f[x_1,\ldots,x_n{:=}g_1,\ldots,g_n]$ to $f[x_i{:=}g_i]_{i=1}^n$ or $f[\vec{x}{:=}\vec{g}]$.

So substitution in RTT can be seen as application plus β-reduction to β-normal form in λ-calculus. Definition 2.24 is schematically reflected in Figure 2. Notice that $f[x_1,\ldots,x_n{:=}g_1,\ldots,g_n]$ should be seen as a *simultaneous* substitution of $g_1,\ldots,g_n$ for $x_1,\ldots,x_n$. As the $\overline{g_i}$s are either closed λ-terms, or individuals, or variables, it is no problem to define this simultaneous substitution via a list of applications that results in a list of *consecutive* substitutions.

Example 2.25

1. $\mathtt{S}(\mathtt{x}_1)[\mathtt{x}_1{:=}\mathtt{a}_1] \equiv \mathtt{S}(\mathtt{a}_1)$, as $(\lambda\mathtt{x}_1.\mathtt{S}\mathtt{x}_1)\mathtt{a}_1 \to_\beta^{\mathrm{nf}} \mathtt{S}\mathtt{a}_1$;

2. $\mathtt{S}(\mathtt{x}_1)[\mathtt{x}_2{:=}\mathtt{a}_2] \equiv \mathtt{S}(\mathtt{x}_1)$, as $(\lambda \mathtt{x}_2.\mathtt{S}\mathtt{x}_1)\mathtt{a}_2 \rightarrow_{\beta}^{\mathrm{nf}} \mathtt{S}\mathtt{x}_1$;

3. $\mathtt{z}(\mathtt{S}(\mathtt{x}_1), \mathtt{x}_2, \mathtt{a}_2)[\mathtt{x}_1{:=}\mathtt{a}_1] \equiv \mathtt{z}(\mathtt{S}(\mathtt{x}_1), \mathtt{x}_2, \mathtt{a}_2)$ as
$(\lambda \mathtt{x}_1.\mathtt{z}(\lambda \mathtt{x}_1.\mathtt{S}\mathtt{x}_1)\mathtt{x}_2\mathtt{a}_2)\mathtt{a}_1 \rightarrow_{\beta}^{\mathrm{nf}} \mathtt{z}(\lambda \mathtt{x}_1.\mathtt{S}\mathtt{x}_1)\mathtt{x}_2\mathtt{a}_2$.
This illustrates that the λ-notation is more precise and convenient with respect to free variables. In $\mathtt{z}(\mathtt{S}(\mathtt{x}_1), \mathtt{x}_2, \mathtt{a}_2)$, it is not immediately clear whether $\mathtt{x}_1$ is a free variable or not and one might tend to write $\mathtt{z}(\mathtt{S}(\mathtt{x}_1), \mathtt{x}_2, \mathtt{a}_2)[\mathtt{x}_1{:=}\mathtt{a}_1] \equiv \mathtt{z}(\mathtt{S}(\mathtt{a}_1), \mathtt{x}_2, \mathtt{a}_2)$. The λ-notation is more explicit in showing that $\mathtt{x}_1 \notin \mathrm{FV}(\mathtt{z}(\mathtt{S}(\mathtt{x}_1), \mathtt{x}_2, \mathtt{a}_2))$;

4. See Remarks 2.10.3.
$\mathtt{z}(\mathtt{R}(\mathtt{a}), \mathtt{S}(\mathtt{a}))[\mathtt{z}{:=}\mathtt{z}_1() \vee \mathtt{z}_2()] \equiv \mathtt{R}(\mathtt{a}) \vee \mathtt{S}(\mathtt{a})$, as
$(\lambda \mathtt{z}.\mathtt{z}(\mathtt{Ra})(\mathtt{Sa}))(\lambda \mathtt{z}_1\mathtt{z}_2.\vee \mathtt{z}_1\mathtt{z}_2) \rightarrow_{\beta}$
$(\lambda \mathtt{z}_1\mathtt{z}_2.\vee \mathtt{z}_1\mathtt{z}_2)(\mathtt{Ra})(\mathtt{Sa}) \twoheadrightarrow_{\beta}^{\mathrm{nf}} \vee(\mathtt{Ra})(\mathtt{Sa})$;

5. $\mathtt{x}_2(\mathtt{x}_1, \mathtt{R}(\mathtt{x}_1))[\mathtt{x}_2{:=}\mathtt{x}_4(\mathtt{x}_3)] \equiv \mathtt{R}(\mathtt{x}_1)$ as
$(\lambda \mathtt{x}_2.\mathtt{x}_2\mathtt{x}_1(\lambda \mathtt{x}_1.\mathtt{R}\mathtt{x}_1))(\lambda \mathtt{x}_3\mathtt{x}_4.\mathtt{x}_4\mathtt{x}_3) \twoheadrightarrow_{\beta}$
$(\lambda \mathtt{x}_3\mathtt{x}_4.\mathtt{x}_4\mathtt{x}_3)\mathtt{x}_1(\lambda \mathtt{x}_1.\mathtt{R}\mathtt{x}_1) \twoheadrightarrow_{\beta}$
$(\lambda \mathtt{x}_1.\mathtt{R}\mathtt{x}_1)\mathtt{x}_1 \rightarrow_{\beta}^{\mathrm{nf}} \mathtt{R}\mathtt{x}_1$.

Remark 2.26 $f[x_1, \ldots, x_n{:=}g_1, \ldots, g_n]$ is not always defined. For its existence we need:

- The existence of the normal form H in Definition 2.24. For instance, this normal form does not exist if we choose $n = 1$, $f \equiv \mathtt{x}_1(\mathtt{x}_1)$ and $g_1 \equiv \mathtt{x}_1(\mathtt{x}_1)$: then we obtain for the calculation of $f[\mathtt{x}_1{:=}g_1]$ the famous λ-term $(\lambda \mathtt{x}_1.\mathtt{x}_1\mathtt{x}_1)(\lambda \mathtt{x}_1.\mathtt{x}_1\mathtt{x}_1)$;

- The existence of a (unique) h such that $\widetilde{h} \equiv H$. For instance, if we take $n = 1$, $f \equiv \mathtt{z}(\mathtt{a})$ (with $\mathtt{z} \in \mathcal{V}$ and $\mathtt{a} \in \mathcal{A}$) and $g_1 \equiv \mathtt{a}$, then $H \equiv \mathtt{aa}$ and there is no $h \in \mathcal{P}$ such that $\widetilde{h} \equiv \mathtt{aa}$.

In Section 2c2 we will prove that, as long as we are within the type system RTT (to be introduced in Section 2b), both H and h always exist uniquely (Corollary 2.73). Until then, the notation $f[x_1, \ldots, x_n{:=}g_1, \ldots, g_n] = h$ implicitly assumes that the substitution *exists*.

Remark 2.27 If we compute a substitution $f[x_1, \ldots, x_n{:=}g_1, \ldots, g_n]$, we have to reduce the λ-term $(\lambda x_1 \cdots x_n.\widetilde{f})\widetilde{g_1} \cdots \widetilde{g_n}$ to its β-normal form (if there is any). One might wonder whether this is too restrictive: In a reduction path to this normal form, there may be an intermediate result H that could be interpreted as the final result of the substitution $f[\vec{x}{:=}\vec{g}]$. However, this never happens, as any term that can be interpreted as such a result is always of the form $\widetilde{h}$, and is therefore always in β-normal form (Lemma 2.23.2).

Remark 2.28 The alphabetical order of the variables plays a crucial role in the substitution process, as it determines in which order the free variables of a pf f are curried in the translation $\overline{f}$. For example, look at the substitutions $\mathtt{z(a,b)[z{:=}R(x,y)]}$ and $\mathtt{z(a,b)[z{:=}R(y,x)]}$. The result of the first one is obtained via the normal form of $(\lambda\mathtt{z.zab})(\lambda\mathtt{xy.Rxy})$, which is equal to $\mathtt{Rab}$, translated: $\mathtt{R(a,b)}$. The second one is calculated via $(\lambda\mathtt{z.zab})(\lambda\mathtt{xy.Ryx})$, resulting in $\mathtt{Rba}$ and $\mathtt{R(b,a)}$.

Remark 2.29 Now that substitution has been properly defined, we could define that f is an *abstraction* of g if there are $x_1,\ldots,x_n \in \mathrm{FV}(f)$ and $h_1,\ldots,h_n \in \mathcal{A}\cup\mathcal{P}$ such that $f[\vec{x}{:=}\vec{h}] \equiv g$, or, in λ-calculus notation: $(\lambda x_1\cdots x_n.\widetilde{f})\overline{h_1}\cdots\overline{h_n} \twoheadrightarrow_\beta \widetilde{g}$. The set of abstractions of a pf g is therefore comparable with the set of β-expansions of the λ-term $\widetilde{g}$.

Some elementary calculation with substitutions can be done using the following lemma:

Lemma 2.30

1. *Assume* $(f_1\vee f_2)[\vec{x}{:=}\vec{h}]$ *exists. Then* $f_j[\vec{x}{:=}\vec{h}]$ *exists for* $j=1,2$, *and*
$$(f_1\vee f_2)[\vec{x}{:=}\vec{h}] \equiv (f_1[\vec{x}{:=}\vec{h}])\vee(f_2[\vec{x}{:=}\vec{h}]);$$
2. *Assume* $(\neg f)[\vec{x}{:=}\vec{h}]$ *exists. Then* $f[\vec{x}{:=}\vec{h}]$ *exists, and*
$$(\neg f)[\vec{x}{:=}\vec{h}] \equiv \neg(f[\vec{x}{:=}\vec{h}]);$$
3. *Assume* $(\forall x{:}t^a[f])[\vec{x}{:=}\vec{h}]$ *exists, and* $x\notin\vec{x}$. *Then* $f[\vec{x}{:=}\vec{h}]$ *exists, and*
$$(\forall x{:}t^a[f])[\vec{x}{:=}\vec{h}] \equiv \forall x{:}t^a[f[\vec{x}{:=}\vec{h}]];$$

4. *Assume* $z(k_1,\ldots,k_n)[z{:=}f]$ *exists, and* $x_1 < \cdots < x_n$ *are the free variables of* f. *Then* $f[\vec{x}{:=}\vec{k}]$ *exists, and*

$$z(k_1,\ldots,k_n)[z{:=}f] \equiv f[\vec{x}{:=}\vec{k}];$$

5. *Assume* $z(k_1,\ldots,k_n)[\vec{x}{:=}\vec{h}]$ *exists,* $z \equiv x_p$, *and* $y_1 < \cdots < y_n$ *are the free variables of* $k_p \in \mathcal{P}$. *Define* $k_i' \equiv h_j$ *if* $k_i \equiv x_j$, *and* $k_i' \equiv k_i$ *otherwise. Then* $k_p[\vec{y}{:=}\vec{k'}]$ *exists, and*

$$z(k_1,\ldots,k_n)[\vec{x}{:=}\vec{h'}] \equiv k_p[y_j{:=}\vec{k'}].$$

PROOF: Directly from the definition of substitution. ⊠

2b The Ramified Theory of Types

After we have formalised the notion of propositional function in Section 2a we now give a precise description of the type theory underlying the *Principia*. First we explicitly introduce types (Section 2b1 — there is no such introduction in *Principia*), and then we formalise the notion "the propositional function f has type t" (Section 2b2).

2b1 Types

Types in the *Principia* have a double hierarchy: one of (simple) types and one of orders. In Section 2b1.1 we introduce the first hierarchy. In Section 2b1.2 we extend this hierarchy with orders, resulting in the ramified types of the *Principia*.

2b1.1 Simple types

As we saw in Section 1b, Frege already distinguished between objects, functions that take objects as arguments, and functions that take functions as arguments. He also made a distinction between functions that take one and functions that take two arguments (see the quotations from *Function and Concept* on p. 19). In the *Principia*, Whitehead and Russell use a similar principle. Whilst Frege's argument for this distinction was only that functions are fundamentally different from objects, and that functions taking objects as arguments are fundamentally different from functions taking functions as arguments, Whitehead and Russell are more precise:

> "[The difference between objects and propositional functions] arises from the fact that a [propositional] function is essentially an ambiguity, and that, if it is to occur in a definite proposition, it must occur in such a way that the ambiguity has disappeared, and a wholly unambiguous statement has resulted."
>
> (*Principia Mathematica*, p. 47)

There is no definition of "type" in the *Principia*, only a definition of "being of the same type":[11]

> "*Definition of "being of the same type."*" The following is a step-by-step definition, the definition for higher types presupposing that for lower types. We say that u and v "are of the same type" if
>
> 1. both are individuals,
> 2. both are elementary [propositional] functions[12] taking arguments of the same type,
> 3. u is a pf and v is its negation,
> 4. u is $\varphi\hat{x}$[13] or $\psi\hat{x}$, and v is $\varphi\hat{x} \vee \psi\hat{x}$, where $\varphi\hat{x}$ and $\psi\hat{x}$ are elementary pfs,
> 5. u is $(y).\varphi(\hat{x}, y)$[14] and v is $(z).\psi(\hat{x}, z)$, where $\varphi(\hat{x}, \hat{y})$, $\psi(\hat{x}, \hat{y})$ are of the same type,
> 6. both are elementary propositions,

[11]See Definition 2.2 for the notion of elementary proposition. In the *Principia*, an elementary pf is a pf that has elementary propositions as values, when it takes suitable arguments.

[12]The term *elementary functions* refers to a pf that has only elementary propositions as value, when it takes suitable (well-typed) arguments. See *Principia*, p. 92.

[13]Whitehead and Russell use $\varphi\hat{x}$ to denote that φ is a pf that has, amongst others, x as a free variable. Similarly, they use $\varphi(\hat{x}, \hat{y})$ to indicate that φ has x, y amongst its free variables.

[14]Whitehead and Russell write $(x).\varphi(x)$ where we would write $\forall x[\varphi]$.

7. u is a proposition and v is $\sim u$[15], or
8. u is $(x).\varphi x$ and v is $(y).\psi y$, where $\varphi\hat{x}$ and $\psi\hat{x}$ are of the same type."

(*Principia Mathematica*, $*9{\cdot}131$, p. 133)

The definition has to be seen as the definition of an equivalence relation. For instance, assume that $\varphi\hat{x}$, $\psi\hat{x}$ and $\chi\hat{x}$ are elementary pfs. By rule 4, $\varphi\hat{x}$ and $\varphi\hat{x} \vee \psi\hat{x}$ are of the same type, and so are $\varphi\hat{x}$ and $\varphi\hat{x} \vee \chi\hat{x}$. By (implicit) transitivity, $\varphi\hat{x} \vee \psi\hat{x}$ and $\varphi\hat{x} \vee \chi\hat{x}$ are of the same type.

The definition seems rather precise at first sight. But there are several remarks to be made:

- The notion "being of the same type" seems to be defined for pfs taking one argument only. On the other hand, rules 2 and 5 suggest that such a definition should be extended to pfs taking two arguments. How this should be done is not made explicit;

- According to this definition, $\mathtt{z}_1() \vee \neg\mathtt{z}_1()$ is not of the same type as $\mathtt{z}_1()$. The only rules by which could be derived that $\mathtt{z}_1()$ and $\mathtt{z}_1() \vee \neg\mathtt{z}_1()$ are of the same type, are rules 2 and 4. But if we want to use these rules, $\mathtt{z}_1()$ must be an elementary pf, which it is not: It can take the argument $\forall\mathtt{x}[\mathtt{R}(\mathtt{x})]$, which has as result the proposition $\forall\mathtt{x}[\mathtt{R}(\mathtt{x})]$. This is not an elementary proposition and therefore $\mathtt{z}_1()$ is not an elementary pf.

So there are quite some omissions in this definition. However, the intention of the definition is clear: pfs that take a different number of arguments, or that take arguments of different types, cannot be of the same type.

In order to make precise what is meant by "being of the same type", it is easier to explicate what these types "are". The notion "being of the same type" can then be replaced by "having the same type". The notion of simple type as defined below is due to Ramsey [101] (1926). Historically, it is incorrect to give Ramsey's definition of simple type before Russell's definition of ramified type, as Russell's definition is of an earlier date, and Ramsey's definition is in fact based on Russell's ideas and not the other way around. On the other hand, the ideas behind simple types were already

[15] $\sim u$ is *Principia* notation for $\neg u$.

explained by Frege (see the quotes from *Function and Concept* on page 19). Moreover, knowledge of the intuition behind simple types will make it easier to understand the ramified ones. Therefore we present Ramsey's definition first.

Definition 2.31 (Simple types)

1. $\mathtt{0}$ is a simple type;
2. If $t_1, \ldots, t_n$ are simple types, then also $(t_1, \ldots, t_n)$ is a simple type. $n = 0$ is allowed: then we obtain the simple type $()$;
3. All simple types can be constructed using the rules 1 and 2.

We use $t, u, t_1, \ldots$ as metavariables over simple types.

Here, $(t_1, \ldots, t_n)$ is the type of pfs that should take n arguments (have n free variables), the ith argument having type t_i. The type $()$ stands for the type of the propositions, and the type $\mathtt{0}$ stands for the type of the individuals.

Remark 2.32 To formalise the notion of ith argument that a pf takes, we use the alphabetical order on variables that was introduced in Section 2a. The ith argument taken by a pf will be substituted for the ith free variable of that pf, according to the alphabetical order.

Now it becomes clear why we considered the alphabetical order of variables in the definition of α-equality 2.18: we want α-equal pfs to have the same type. However, if f has type (t_1, t_2) and two free variables $x < y$, and g is the same as f except that the roles of x and y have been switched, then g will have type (t_2, t_1). Therefore we demand that the renaming of variables must maintain the alphabetical order. See also Remark 2.47.7.

Example 2.33 The propositional function $\mathtt{R(x)}$ should have type $(\mathtt{0})$, as it takes one individual as argument.

The propositional function $\mathtt{z(R(x), S(a))}$ (see Remark 2.10.4) takes one argument. This argument must be a pf that can take $\mathtt{R(x)}$ as its first argument (so this first argument must be of type $(\mathtt{0})$), and a proposition (of type $()$) as its second argument. We conclude that in $\mathtt{z(R(x), S(a))}$, we must substitute pfs of type $((\mathtt{0}), ())$ for $\mathtt{z}$. Therefore, $\mathtt{z(R(x), S(a))}$ has type $(((\mathtt{0}), ()))$.

The intuition presented in Remark 2.32 and Example 2.33 will be formalised in 2.45. Theorem 2.58 shows that this formalisation follows the intuition.

Just as propositional functions can be translated to λ-terms, simple types can be translated to types of the simply typed λ-calculus of Church (see [30], and Section Ab of the Appendix).

Definition 2.34 We define a type $T(t)$ for each simple type t by induction:

1. $T(0) \stackrel{\text{def}}{=} \iota$;
2. $T((t_1, \ldots, t_n)) \stackrel{\text{def}}{=} T(t_1) \to \cdots \to T(t_n) \to o$.

A simple type t of Definition 2.31 has the same interpretation as its translation $T(t)$. Moreover, T is injective:

Lemma 2.35 *If t and u are simple types, then $T(t) = T(u)$ if and only if $t = u$.*

PROOF: Induction on the definition of simple type. ⊠

Notation 2.36 From now on we will use a slightly different notation for quantification in pfs. Instead of $\forall x[f]$ we now explicitly mention the type (say: t) over which is quantified: $\forall x{:}t[f]$. We do the same with the translations of pfs to λ-calculus: instead of $\lambda x.F$ we write $\lambda x{:}T(t).F$.

2b1.2 Ramified types

Up to now, the type of a pf only depends on the types of the arguments that it can take. In the *Principia*, a second hierarchy is introduced by regarding also the types of the variables that are bound by a quantifier (see *Principia*, pp. 51–55). Whitehead and Russell consider, for instance, the propositions $\mathsf{R}(\mathsf{a})$ and $\forall \mathsf{z}{:}()[\mathsf{z}() \vee \neg\mathsf{z}()]$ to be of a different level. The first is an atomic proposition, while the latter is based on the pf $\mathsf{z}() \vee \neg\mathsf{z}()$. The pf $\mathsf{z}() \vee \neg\mathsf{z}()$ involves an arbitrary proposition z, therefore $\forall \mathsf{z}{:}()[\mathsf{z}() \vee \neg\mathsf{z}()]$ quantifies over all propositions z. According to the vicious circle principle 2.1, $\forall \mathsf{z}{:}()[\mathsf{z}() \vee \neg\mathsf{z}()]$ cannot belong to this collection of propositions.

This problem is solved by dividing types into *orders* (not to be confused with the *alphabetical order* on the variables). An order is simply a natural number. Basic propositions are of order 0, and in $\forall \mathsf{z}{:}()[\mathsf{z}() \vee \neg\mathsf{z}()]$ we must mention the order of the propositions over which is quantified. The pf $\forall \mathsf{z}{:}()^n[\mathsf{z}() \vee \neg\mathsf{z}()]$ quantifies over all propositions of order n, and has order $n+1$.

The division of types into orders gives *ramified* types.

Definition 2.37 (Ramified types)

1. 0^0 is a ramified type;
2. If $t_1^{a_1}, \ldots, t_n^{a_n}$ are ramified types, and $a \in \mathbb{N}$, $a > \max(a_1, \ldots, a_n)$, then $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ is a ramified type (if $n = 0$ then take $a \geq 0$);
3. All ramified types can be constructed using the rules 1 and 2.

If t^a is a ramified type, then a is called the *order* of t^a.

Remark 2.38 In $(t_1^{a_1}, \ldots, t_n^{a_n})^a$, we demand that $a > a_i$ for all i. This is because a pf of this type presupposes all the elements of type ${t_i}^{a_i}$, and therefore must be of an order that is higher than a_i.

Example 2.39 We give some examples of ramified types:

- 0^0;
- $(0^0)^1$;
- $\left((0^0)^1, (0^0)^4\right)^5$;
- $\left(0^0, ()^2, \left(0^0, (0^0)^1\right)^2\right)^7$.

$\left(0^0, \left(0^0, (0^0)^2\right)^2\right)^7$ is not a ramified type.

Ramified types can also be translated to types of the simply typed λ-calculus. However, we lose the orders if we do so.

Definition 2.40 We define a type $T(t)$ for each ramified type t by induction:

1. $T(0^0) \stackrel{\text{def}}{=} \iota$;

2. $T\left((t_1^{a_1}, \ldots, t_n^{a_n})^a\right) \stackrel{\text{def}}{=} T(t_1) \to \cdots \to T(t_n) \to o$.

In the rest of this Chapter we simply speak of *types* when we mean *ramified* types, as long as no confusion arises.

In the type $(0^0)^1$, all orders are "minimal", i.e., not higher than strictly necessary. This is, for instance, not the case in the type $(0^0)^2$. Types in which all orders are minimal are called *predicative* and play a special role in the Ramified Theory of Types. A formal definition:

Definition 2.41 (Predicative types)

1. 0^0 is a predicative type;

2. If $t_1{}^{a_1}, \ldots, t_n{}^{a_n}$ are predicative types, and $a = 1 + \max(a_1, \ldots, a_n)$ (take $a = 0$ if $n = 0$), then $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ is a predicative type;

3. All predicative types can be constructed using the rules 1 and 2 above.

The mapping T is injective when restricted to predicative types:

Lemma 2.42 *If t^a and u^b are predicative types, then $T(t^a) = T(u^b)$ if and only if $t^a = u^b$.*

PROOF: Induction on the definition of predicative type. ⊠

2b2 Formalisation of the Ramified Theory of Types

In this section we formalise the intuition on types presented in Example 2.33 and Definition 2.34 together with the intuition on orders that was given at the beginning of Section 2b1.2. Before we can do this we must introduce some additional terminology.

In the pf R(x) we implicitly assume that x is a variable for which objects of type 0 must be substituted. For our formalisation we want to make the information on the type of a variable explicit. We do this by storing this information in so-called *contexts*. Contexts, common in modern type systems, are not used in the *Principia*.

Definition 2.43 (Contexts) Let $x_1, \dots, x_n \in \mathcal{V}$ be distinct variables, and assume $t_1^{a_1}, \dots, t_n^{a_n}$ are ramified types. Then $\{x_1{:}t_1^{a_1}, \dots, x_n{:}t_n^{a_n}\}$ is a *context*. The set $\{x_1, \dots, x_n\}$ is called the *domain* of the context and is denoted by $\mathrm{dom}(\{x_1{:}t_1^{a_1}, \dots, x_n{:}t_n^{a_n}\})$. We will use Greek capitals Γ, Δ as meta-variables over contexts.

The pfs $\mathbf{z}_1(\mathbf{y}_1)$ and $\mathbf{z}_2(\mathbf{y}_2)$ are α-equal, according to Definition 2.18. But in a context $\Gamma \equiv \left\{\mathbf{y}_1{:}0^0, \mathbf{z}_1{:}(0^0)^1, \mathbf{y}_2{:}(0^0)^1, \mathbf{z}_2{:}\left((0^0)^1\right)^2\right\}$ one does not want to see $\mathbf{z}_1(\mathbf{y}_1)$ and $\mathbf{z}_2(\mathbf{y}_2)$ as equal, as the types of $\mathbf{y}_1$ and $\mathbf{y}_2$ differ, and the types of $\mathbf{z}_1$ and $\mathbf{z}_2$ differ as well. Therefore, we introduce a more restricted version of α-equality:

Definition 2.44 Let Γ be a context and f and g pfs. We say that f and g are *α_Γ-equal*, notation $f =_{\alpha,\Gamma} g$, if there is a bijection $\varphi : \mathcal{V} \to \mathcal{V}$ such that

- g can be obtained from f by replacing each variable that occurs in f by its φ-image;
- $x < y$ iff $\varphi(x) < \varphi(y)$;
- $x{:}t \in \Gamma$ iff $\varphi(x){:}t \in \Gamma$.

We will now define what we mean by $\Gamma \vdash f : t^a$, or, in words: f is of type t^a in the context Γ.[16] In this definition we will try to follow the line of the Principia as much as possible. If $\Gamma \equiv \varnothing$ then we will write $\vdash f : t^a$.

We explain some aspects of the following definition in Section 2b3.

Definition 2.45 (Ramified Theory of Types: RTT) The *judgements* $\Gamma \vdash f : t^a$ is inductively defined as follows:

1. **(start)** For all a:
$$\vdash a : 0^0.$$
For all atomic pfs f:
$$\vdash f : ()^0;$$

[16]The symbol $\vdash$ in $\Gamma \vdash f : t^a$ is the same symbol that Frege used to assert a proposition. It enters Type Theory in 1934 [36], via Curry's combinatory logic. Curry defines a functionality combinator F in such a way that $\mathsf{F}XYf$ holds, exactly if f is a function from X to Y. To denote the assertion of $\mathsf{F}XYf$, Curry uses Frege's symbol $\vdash$.

2. **(connectives)** Assume $\Gamma \vdash f{:}(t_1^{a_1},\ldots,t_n^{a_n})^a$, $\Delta \vdash g{:}(u_1^{b_1},\ldots,u_m^{b_m})^b$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$ and $y \in \mathrm{dom}(\Delta)$. Then

$$\Gamma \cup \Delta \vdash f \vee g : \left(t_1^{a_1},\ldots,t_n^{a_n},u_1^{b_1},\ldots,u_m^{b_m}\right)^{\max(a,b)};$$

$$\Gamma \vdash \neg f : (t_1^{a_1},\ldots,t_n^{a_n})^a;$$

3. **(abstraction from parameters)** If $\Gamma \vdash f : (t_1^{a_1},\ldots,t_m^{a_m})^a$, $t_{m+1}^{a_{m+1}}$ is a *predicative* type[17], $g \in \mathcal{A} \cup \mathcal{P}$ is a parameter of f, $\Gamma \vdash g : t_{m+1}^{a_{m+1}}$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$, then

$$\Gamma' \vdash h : (t_1^{a_1},\ldots,t_{m+1}^{a_{m+1}})^{\max(a,a_{m+1}+1)}.$$

Here, h is a pf obtained by replacing all parameters g' of f which are α_Γ-equal to g by y. Moreover, Γ' is the subset of the context $\Gamma \cup \{y : t_{m+1}^{a_{m+1}}\}$ such that $\mathrm{dom}(\Gamma')$ contains exactly all the variables that occur in h[18];

4. **(abstraction from pfs)** If $(t_1^{a_1},\ldots,t_m^{a_m})^a$ is a *predicative* type[17], $\Gamma \vdash f : (t_1^{a_1},\ldots,t_m^{a_m})^a$, $x < z$ for all $x \in \mathrm{dom}(\Gamma)$, and $y_1 < \cdots < y_n$ are the free variables of f, then

$$\Gamma' \vdash z(y_1,\ldots,y_n) : (t_1^{a_1},\ldots,t_m^{a_m},(t_1^{a_1},\ldots,t_m^{a_m})^a)^{a+1},$$

where Γ' is the subset of $\Gamma \cup \{z{:}(t_1^{a_1},\ldots,t_m^{a_m})^a\}$ such that $\mathrm{dom}(\Gamma') = \{y_1,\ldots,y_n,z\}$[18];

5. **(weakening)** If Γ, Δ are contexts, $\Gamma \subseteq \Delta$, and $\Gamma \vdash f : t^a$, then also $\Delta \vdash f : t^a$;

6. **(substitution)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y : t_i^{a_i}\} \vdash f : (t_1^{a_1},\ldots,t_n^{a_n})^a$, and $\Gamma \vdash k : t_i^{a_i}$ then

$$\Gamma' \vdash f[y{:=}k] : (t_1^{a_1},\ldots,t_{i-1}^{a_{i-1}},t_{i+1}^{a_{i+1}},\ldots,t_n^{a_n})^b.$$

[17]The restriction to predicative types only is based on *Principia*, pp. 53–54.

[18]In Lemma 2.56 we prove that this context always exists.

Here, $b = 1 + \max(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c)$, and

$$c = \max\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\}$$

(if $n = 1$ and $\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\} = \varnothing$ then take $b = 0$) and once more, Γ' is the subset of $\Gamma \cup \{y : t_i^{a_i}\}$ such that $\mathrm{dom}(\Gamma')$ contains exactly all the variables that occur in $f[y{:=}k]$[18];

7. **(permutation)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $x < y'$ for all $x \in \mathrm{dom}(\Gamma)$, then

$$\Gamma' \vdash f[y{:=}y'] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n}, t_i^{a_i})^a.$$

Γ' is the subset of $\Gamma \cup \{y{:}t_i^{a_i}, y'{:}t_i^{a_i}\}$ such that $\mathrm{dom}\Gamma'$ contains exactly all the variables that occur in $f[y{:=}y']$[18];

8. **(quantification)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, then

$$\Gamma \vdash \forall y{:}t_i^{a_i}[f] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n})^a.$$

Definition 2.46 A pf f is called *legal*, if there is a context Γ and a ramified type t^a such that $\Gamma \vdash f : t^a$.

2b3 Discussion and examples

We will make some remarks on Definition 2.45. First of all, we motivate the eight rules of 2.45 by referring to passages in the *Principia*. Then we make some technical remarks, and give some examples of how the rules work. It will be made clear that the substitution rule is problematic, because substitution is not clearly defined in the *Principia*.

Remark 2.47 We will motivate RTT (Definition 2.45) by referring to the Principia:

1. Individuals and elementary judgements (atomic propositions) are, also in the *Principia*, the basic ingredients for creating legal pfs;[19]

[19]As for individuals: see *Principia*, ∗9, p. 132, where "Individual" is presented as a primitive idea. As for elementary judgements: See *Principia*, Introduction, pp. 43-45.

2. We can see rule 2 "at work" in $*12$, p. 163 of the *Principia*[20]:

   > "We can build up a number of new formulas, such as [...] $\varphi!x \vee \varphi!y$, $\varphi!x \vee \psi!x$, $\varphi!x \vee \psi!y$, [...] and so on."
   >
   > (*Principia Mathematica*, $*12$, p. 163))

   The restriction about contexts that we make in rule 2 has technical reasons and is not made in the *Principia*. It will be discussed in 2.49;

3. Rule 3 is justified by $*9{\cdot}14$ and $*9{\cdot}15$ in the *Principia*. It is an instantiation of the abstraction principles 1.1 and 1.2 for functions that was already proposed by Frege. In Frege's definition one does not have to replace all parameters g' that are α_{Γ}-equal to g, but one can also take some of these parameters. In Section 2d we show that this is not a serious restriction.

   The restriction to *predicative* types is in line with the *Principia* (cf. *Principia*, pp. 53–54);

4. Rule 4 is based on the Introduction of the *Principia*. There, pfs are constructed, and

   > "the first matrices that occur are those whose values are of the forms $\varphi x, \psi(x,y), \chi(x,y,z,\dots)$, *i.e.* where the arguments, however many there may be, are all individuals. Such [propositional] functions we will call '*first-order* functions.' We may now introduce a notation to express 'any first-order function.' "
   >
   > (*Principia Mathematica*, p. 51)

   This quote from the *Principia* is an instance of Frege's abstraction principles, and so is rule 4 of our formalisation. It results in second order pfs, and the process can be iterated to obtain pfs of higher orders.

   Rule 4 makes it possible to introduce variables of higher order. In fact, leaving out rule 4 would lead to first-order predicate logic, as

[20]In the *Principia*, Whitehead and Russell write $\varphi!x$ instead of φx to indicate that φx is not only (what we would call) a pf, but even a *legal* pf.

without rule 4 it is impossible to introduce variables of types that differ from 0^0.

The use of predicative types only is inspired by the *Principia*, again;

5. The weakening rule cannot be found in the *Principia*, because no formal contexts are used there. It is implicitly present, however: the addition of an extra variable to the set of variables does not affect the well-typedness of pfs that were already constructed;

6. The rule of substitution is based on ∗9·14 and ∗9·15 of the *Principia*, and can be seen as an inverse of the abstraction operators in rule 3 and 4. Notice that we do not know yet whether the substitution $f[y{:=}k]$ exists or not. Therefore, we limit the use of rule 6 to the cases in which the substitution exists. In Section 2c2 we show that it always exists if the premises of rule 6 are fulfilled;

7. In the system above, the (sequential) order of the t_is is related to the alphabetic order of the free variables of the pf f that has type $(t_1, \ldots, t_n)$ (see the remark before Definition 2.18, Remark 2.32, and Theorem 2.58). This alphabetic order plays a role in the clear presentation of results like Theorem 2.58, and in the definition of substitution (see Remark 2.28).

   With rule 7 we want to express that the order of the t_is in $(t_1, \ldots, t_n)$ and the alphabetic order of the variables are not characteristics of the *Principia*, but are only introduced for the technical reasons explained in this remark. This is worked out in Corollary 2.59;

8. Notice that in the quantification rule, both f and $\forall x{:}t_i^{a_i}.f$ have order a. The intuition is that the order of a propositional function f equals one plus the maximum of the orders of *all* the variables (either free or bound by a quantifier) in f. This is in line with the *Principia*: see [121], page 53. See also the introduction to Definition 2.37, and the proof of Lemma 2.60 below.

Remark 2.48 Rules 3 and 4 are a restricted version of the abstraction principles of Frege, with less power. It is, for instance, not possible to imitate all the abstractions of Remark 2.10 by using rules 3 and 4 only. But in combination with the other rules, rule 3 and 4 are sufficient (see

Example 2.54 for the cases of Remark 2.10, and Section 2d, especially Theorem 2.84).

Remark 2.49 In rule 2 of RTT, we make the assumption that the variables of Γ must all come before the variables of Δ. The reason for this is that we want to prevent undesired results like

$$\mathtt{x}_1{:}0^0 \vdash \mathtt{R}_1(\mathtt{x}_1) \vee \mathtt{R}_2(\mathtt{x}_1) : (0^0, 0^0)^1.$$

In fact, $\mathtt{R}_1(\mathtt{x}_1) \vee \mathtt{R}_2(\mathtt{x}_1)$ has *only one* free variable, so its type should be $(0^0)^1$ and not $(0^0, 0^0)^1$ (see Example 2.53, second part). For technical reasons (the order of the $t_i^{a_i}$s; see also Theorem 2.58) we strengthen the assumption such that for $x \in \mathrm{dom}(\Gamma)$ and $y \in \mathrm{dom}(\Delta)$, $x < y$ must hold.

As Whitehead and Russell do not have a formal notation for types, they do not forbid this kind of constructions in the *Principia*. In 2.82 we show that our limitation to contexts with disjoint domains as made in rule 2 is not a real limitation: all the desired judgements can still be derived for contexts with non-disjoint domains.

Remark 2.50 In both rules 3 and 4 we see that it is necessary to introduce at least one new variable. It is, for instance, not possible to interpret the proposition $\mathtt{R}(\mathtt{a})$ as a (constant) pf of type $(0^0)^1$. This is in line with the abstraction principles of Frege and Russell. In Frege's definition 1.1, for example, it is explicitly mentioned that the object that is to be replaced occurs at least once in the expression.

Translated to λ-calculus this means that the *Principia* have λI-terms, only. See also Lemma 2.9.3 and Lemma 2.23.3.

Remark 2.51 Contexts as used in RTT contain, in a sense, too much information: not only information on all free variables, but also information on non-free variables (Cf. rules 3, 6 and 7. The set of non-free variables contains more than only the variables that are bound by a quantifier. For example, in the pf $\mathtt{z}(\mathtt{R}(\mathtt{x}))$, $\mathtt{x}$ is neither free, nor bound by a quantifier).

Remark 2.52 The system is based on the abstraction principles of Frege. In a context Γ, one cannot introduce a variable of a certain type t unless one has a pf (or an individual) f that has type t in Γ. This is different from modern, λ-calculus based systems, where one can introduce a variable of a type u without knowing whether or not there are terms of this type u.

We give some examples, in order to illustrate how our system works. Example 2.53 shows applications of the rules. Example 2.54 makes a link between the intuitive notion of abstraction that was explained in Remark 2.10 and the abstraction rules 3 and 4 of our system.

We will use a notation of the form $\dfrac{X_1 \cdots X_n}{Y}N$, indicating that from the judgements $X_1, \ldots, X_n$, we can infer the judgement Y by using the RTT-rule of Definition 2.45 with number N. As usual, this is called a *derivation step*. Subsequent derivation steps give a *derivation*. A derivation *of a judgements* Y is a derivation tree with Y as root (the final conclusion). The types in the examples below are all predicative (as a pf of impredicative type must have a quantifier, and the examples below are quantifier-free). To avoid too much notation, we omit the orders.

Example 2.53

- $\vdash \mathtt{S}(\mathtt{a}_1, \mathtt{a}_2) : ()$;
- $$\dfrac{\vdash \mathtt{R}_1(\mathtt{a}_1) : () \qquad \vdash \mathtt{R}_2(\mathtt{a}_1) : ()}{\vdash \mathtt{R}_1(\mathtt{a}_1) \vee \mathtt{R}_2(\mathtt{a}_1) : ()}2$$

  but *not*:

  $$\dfrac{\mathtt{x}_1 : 0 \vdash \mathtt{R}_1(\mathtt{x}_1) : (0) \qquad \mathtt{x}_1 : 0 \vdash \mathtt{R}_2(\mathtt{x}_1) : (0)}{\mathtt{x}_1 : 0 \vdash \mathtt{R}_1(\mathtt{x}_1) \vee \mathtt{R}_2(\mathtt{x}_1) : (0,0)}2$$

  ($\mathtt{x}_1 \not< \mathtt{x}_1$ because $<$ is strict). To obtain $\mathtt{R}_1(\mathtt{x}_1) \vee \mathtt{R}_2(\mathtt{x}_1)$ we must make a different start:

  $$\dfrac{\dfrac{\vdash \mathtt{R}_1(\mathtt{a}_1) : () \qquad \vdash \mathtt{R}_2(\mathtt{a}_1) : ()}{\vdash \mathtt{R}_1(\mathtt{a}_1) \vee \mathtt{R}_2(\mathtt{a}_1) : ()}2 \qquad \vdash \mathtt{a}_1 : 0}{\mathtt{x}_1 : 0 \vdash \mathtt{R}_1(\mathtt{x}_1) \vee \mathtt{R}_2(\mathtt{x}_1) : (0)}3;$$

- $$\dfrac{\begin{array}{lcl} \mathtt{x}_1 : 0, \mathtt{x}_2 : 0, \mathtt{z}_1 : ((0),(0)) & \vdash & \mathtt{z}_1(\mathtt{R}(\mathtt{x}_1), \mathtt{R}(\mathtt{x}_2)) : (((0),(0))) \\ \mathtt{x}_1 : 0, \mathtt{x}_2 : 0, \mathtt{z}_1 : ((0),(0)) & \vdash & \mathtt{R}(\mathtt{x}_1) : (0) \end{array}}{\mathtt{z}_1 : ((0),(0)), \mathtt{z}_2 : (0) \vdash \mathtt{z}_1(\mathtt{z}_2, \mathtt{z}_2) : (((0),(0)),(0))}3$$

  As $\mathtt{R}(\mathtt{x}_1)$ is α-equal to $\mathtt{R}(\mathtt{x}_2)$ in the context, both $\mathtt{R}(\mathtt{x}_1)$ and $\mathtt{R}(\mathtt{x}_2)$ are replaced by the newly introduced variable $\mathtt{z}_2$;

- $$\dfrac{\mathtt{x}_1 : 0, \mathtt{x}_2 : 0 \vdash \mathtt{S}(\mathtt{x}_1, \mathtt{x}_2) : (0,0)}{\mathtt{x}_1 : 0, \mathtt{x}_2 : 0, \mathtt{z} : (0,0) \vdash \mathtt{z}(\mathtt{x}_1, \mathtt{x}_2) : (0,0,(0,0))}4;$$

- $$\dfrac{\dfrac{x_1 : 0 \vdash R_1(x_1) \vee R_2(x_1) : (0) \qquad \vdash a_1 : 0}{\vdash R_1(a_1) \vee R_2(a_1) : ()}\,6;}{x_1 : 0 \vdash R_1(a_1) \vee R_2(a_1) : ()}\,5$$

- $$\dfrac{\begin{array}{rcl} x_1 : 0, x_2 : 0, x_3 : (0,0) & \vdash & R(x_1) \vee \neg x_3(x_1, x_2) : (0,0,(0,0)) \\ x_1 : 0, x_2 : 0 & \vdash & T(x_1, x_1, x_2) : (0,0) \end{array}}{x_1 : 0, x_2 : 0 \vdash R(x_1) \vee \neg T(x_1, x_1, x_2) : (0,0)}\,6$$

  $T(x_1, x_1, x_2)$ is substituted for x_3.

Example 2.54 We give a formal derivation of the examples of the abstraction rules that were given in Remark 2.10. Again, we omit the orders.

- Constructing $z(a) \vee S(a)$ from $R(a) \vee S(a)$ cannot be done with the use of rule 4 only. The following derivation is correct:

  $$\dfrac{\dfrac{\dfrac{\vdash a{:}0}{z{:}(0) \vdash a{:}0}\,5 \qquad \dfrac{\dfrac{\vdash a{:}0 \qquad \vdash R(a){:}()}{x{:}0 \vdash R(x){:}(0)}\,3}{x{:}0, z{:}(0) \vdash z(x){:}(0,(0))}\,4}{z{:}(0) \vdash z(a){:}((0))}\,6 \qquad \vdash S(a){:}()}{z{:}(0) \vdash z(a) \vee S(a) : ((0))}\,2.$$

  To obtain $z(a)$ instead of $z()$, we must transform $R(a)$ into a pf $R(x)$ by abstracting from a. Then we can construct $z(x)$ by abstraction from pfs (rule 4). In this way, the "frame" for $z(a)$ is of the right form. Substituting a for x gives $z(a)$ (and "neutralises" the application of rule 3 at the top of the derivation). Simply applying rule 4 on the judgement $\vdash R(a) : ()$ does not work: it results in $z() \vdash z() : (())$;

- Constructing $z_1() \vee z_2()$ is easier: $z_1()$ can be obtained by abstracting from $R(a)$, and $z_2()$ similarly from $S(a)$. Result:

  $$\dfrac{\dfrac{\vdash R(a){:}()}{z_1{:}() \vdash z_1(){:}(())}\,4 \qquad \dfrac{\vdash S(a){:}()}{z_2{:}() \vdash z_2(){:}(())}\,4}{z_1{:}(), z_2{:}() \vdash z_1() \vee z_2() : ((),())}\,2.$$

  We see that in fact two abstractions are needed to construct this pf: we must abstract from $R(a)$ as an instance of the pf $z_1()$, and from $S(a)$ as an instance of the pf $z_2()$. As rule 4 does not work on parts of pfs, these abstractions have to be made before we use rule 2. Applying rule 4 on $\vdash R(a) \vee S(a) : ()^0$ would result in $z : () \vdash z() : (())$;

- We can extend the derivation of $z_1{:}(), z_2{:}() \vdash z_1() \vee z_2() : ((),())$ to obtain a type for $z(R(a),S(a))$:

$$\dfrac{\dfrac{\dfrac{x_1{:}(), x_2{:}() \vdash x_1() \vee x_2() : ((),())}{x_1{:}(), x_2{:}(), z{:}((),()) \vdash z(x_1,x_2){:}((),(),((),()))}\,4}{x_2{:}(), z{:}((),()) \vdash z(R(a),x_2) : ((),((),()))}\,6}{z{:}((),()) \vdash z(R(a),S(a)) : (((),()))}\,6$$

  (for reasons of space, we omitted the premises $z{:}((),()), x_2{:}() \vdash R(a){:}()$ and $z{:}((),()) \vdash S(a){:}()$ of the first and second application of the substitution rule);

- For the derivation of the type of $z(R(x),S(a))$ we first make a derivation of the "frame" $z(y_1,y_2)$ of this pf:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash a{:}0}{y_1{:}(0) \vdash a{:}0}\,5 \qquad \dfrac{\dfrac{\vdash a{:}0 \quad \vdash R(a){:}()}{x{:}0 \vdash R(x){:}(0)}\,3}{x{:}0, y_1{:}(0) \vdash y_1(x){:}(0,(0))}\,4}{y_1{:}(0) \vdash y_1(a){:}((0))}\,6 \qquad \dfrac{\vdash R(a){:}()}{y_2{:}() \vdash y_2(){:}(())}\,4}{y_1{:}(0), y_2{:}() \vdash y_1(a) \vee y_2() : ((0),())}\,2}{y_1{:}(0), y_2{:}(), z{:}((0),()) \vdash z(y_1,y_2) : ((0),(),((0),()))}\,4.$$

  Then we derive $x{:}0 \vdash R(x){:}(0)$ and $\vdash S(a){:}()$, and after applying the weakening rule, we can substitute $R(x)$ for y_1 and $S(a)$ for y_2. As a result, we get

$$z{:}((0),()), x{:}0 \vdash z(R(x),S(a)) : (((0),())).$$

Example 2.55 In the example below, the orders are important:

$$\dfrac{\dfrac{\dfrac{\vdash R(a) : ()^0 \qquad \dfrac{\vdash R(a) : ()^0}{\vdash \neg R(a) : ()^0}\,2}{\vdash R(a) \vee \neg R(a) : ()^0}\,2}{z{:}()^0 \vdash z() \vee \neg z : (()^0)^1}\,4}{\vdash \forall z{:}()^0[z() \vee \neg z()] : ()^1}\,8.$$

We see that $\forall z{:}()^0[z()^0 \vee \neg z()]$ does not have a predicative type. This is the case because this pf has a bound variable z that is of a higher order than the order of any free variable (as there are no free variables here). Therefore, the order of this pf is determined by the order bound variable z.

We still need to prove that the contexts in the conclusions of rules 3, 4 and 6 exist. This follows from the following Lemma:

Lemma 2.56 *Assume* $\Gamma \vdash f : t^a$. *Then*

1. **(Free variable lemma)** *All variables of* f *that are not bound by a quantifier are in* $\mathrm{dom}(\Gamma)$*;*

2. **(Strengthening lemma)** *If* Δ *is the (unique) subset of* Γ *such that* $\mathrm{dom}(\Delta)$ *contains exactly all the variables of* f *that are not bound by a quantifier, then* $\Delta \vdash f : t^a$.

PROOF: An easy induction on the definition of $\Gamma \vdash f : t^a$. ⊠

2c Properties of RTT

2c1 Types and free variables

In this section we treat some meta-properties of RTT. Using the λ-notation for pfs, we can often refer to known results in typed λ-calculus[21].

Theorem 2.57 (First Free Variable Theorem)
Let $f \in \mathcal{P}$*;* $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$.

$$\mathrm{FV}(f[x_1, \ldots, x_n{:=}k_1, \ldots, k_n]) =$$

$$(\mathrm{FV}(f) \setminus \{x_1, \ldots, x_n\}) \cup \{k_i \in \mathcal{V} \mid x_i \in \mathrm{FV}(f)\}.$$

PROOF: Write $h \equiv f[x_1, \ldots, x_n{:=}k_1, \ldots, k_n]$. We know that $\widetilde{h}$ is the β-normal form of the λ-term $(\lambda_{i=1}^n x_i.\widetilde{f})\overline{k_1} \cdots \overline{k_n}$. By Lemma 2.23.3 and Lemma 2.9.4 we know that $\widetilde{f}, \overline{k_1}, \ldots, \overline{k_n}$ are all λI-terms. We conclude that $\widetilde{f}[x_1{:=}\overline{k_1}] \cdots [x_n{:=}\overline{k_n}]$ is also a λI-term. As

$$\left(\mathop{\lambda}_{i=1}^{n} x_i.\widetilde{f}\right) \overline{k_1} \cdots \overline{k_n} \twoheadrightarrow_\beta \widetilde{f}[x_1{:=}\overline{k_1}] \cdots [x_n{:=}\overline{k_n}]$$

[21]The meta-properties can also be proved directly, without λ-calculus: see [79].

we have by the Church-Rosser Theorem that $\widetilde{f}[x_1{:=}\overline{k_1}]\cdots[x_n{:=}\overline{k_n}] \twoheadrightarrow_\beta \widetilde{h}$, and therefore:

$$\begin{array}{rcl}
\mathrm{FV}(h) & \overset{(2.23.1)}{=} & \mathrm{FV}(\widetilde{h}) \\
 & \overset{(1)}{=} & \mathrm{FV}\Big(\widetilde{f}[x_1{:=}\overline{k_1}]\cdots[x_n{:=}\overline{k_n}]\Big) \\
 & = & \Big(\mathrm{FV}\big(\widetilde{f}\big)\setminus\{x_1,\ldots,x_n\}\Big)\cup\bigcup\Big\{\mathrm{FV}\big(\overline{k_i}\big) \mid x_i\in\mathrm{FV}\big(\widetilde{f}\big)\Big\} \\
 & \overset{(2)}{=} & \Big(\mathrm{FV}\big(\widetilde{f}\big)\setminus\{x_1,\ldots,x_n\}\Big)\cup\Big\{\overline{k_i}\in V \mid x_i\in\mathrm{FV}\big(\widetilde{f}\big)\Big\} \\
 & \overset{(2.23.1)}{=} & (\mathrm{FV}(f)\setminus\{x_1,\ldots,x_n\})\cup\{k_i\in\mathcal{V} \mid x_i\in\mathrm{FV}(f)\}.
\end{array}$$

At (1) we use that $\widetilde{f}[x_1{:=}\overline{k_1}]\cdots[x_n{:=}\overline{k_n}]$ is a λI-term that β-reduces to $\widetilde{h}$; at (2) we use the fact that $\mathrm{FV}\big(\overline{k_i}\big)=\varnothing$ whenever $k_i\in\mathcal{A}\cup\mathcal{P}$ (by definition of $\overline{k_i}$). ⊠

Theorem 2.58 (Second Free Variable Theorem) *Assume that we can derive $\Gamma\vdash f:(t_1^{a_1},\ldots,t_n^{a_n})^a$, and $x_1<\cdots<x_m$ are the free variables of f. Then $m=n$ and $x_i:t_i^{a_i}\in\Gamma$ for all $i\le n$.*

PROOF: An easy induction on $\Gamma\vdash f:(t_1^{a_1},\ldots,t_n^{a_n})^a$. For rules 6 and 7, use Theorem 2.57. ⊠

We can now prove a corollary that we promised in Remark 2.47.7:

Corollary 2.59 *If $\Gamma\vdash f:(t_1^{a_1},\ldots,t_n^{a_n})^a$ and φ is a bijection $\{1,\ldots,n\}\to\{1,\ldots,n\}$ then there is a context Γ' and a pf f' which is α_P-equal to f such that*

$$\Gamma'\vdash f':\left(t_{\varphi(1)}^{a_{\varphi(1)}},\ldots,t_{\varphi(n)}^{a_{\varphi(n)}}\right)^a.$$

PROOF: By the second Free Variable Theorem, we can assume that f has n free variables $x_1<\cdots<x_n$, and that $x_i{:}t_i^{a_i}\in\Gamma$ for all $i\in\{1,\ldots,n\}$. Take n new free variables $z_1<\cdots<z_n$ such that $z_i>y$ for all $y\in\mathrm{dom}(\Gamma)$. Now apply rule 7 of RTT n times. ⊠

We can also prove unicity of types and unicity of orders. Orders are unique in the following sense:

Lemma 2.60 *Assume $\Gamma \vdash f : t^a$. If x occurs in f and $x : u^b \in \Gamma$, then u^b is predicative. Moreover, if also $\Gamma \vdash f : t'^{a'}$, then $a = a'$.*

PROOF: By induction on the derivation of $\Gamma \vdash f : t^a$ one shows that a variable x that occurs in f always has a predicative type in Γ, and that both a and a' equal one plus the maximum of the orders of all the (free and non-free) variables that occur in f. ⊠

Corollary 2.61 (Unicity of types for pfs) *Assume Γ is a context, f is a pf, $\Gamma \vdash f : t^a$ and $\Gamma \vdash f : u^b$. Then $t^a \equiv u^b$.*

PROOF: $t \equiv u$ follows from Theorem 2.58; $a = b$ from Lemma 2.60. ⊠

Remark 2.62 We cannot omit the context Γ in Corollary 2.61. For example, the pf $\mathsf{z}(\mathsf{x})$ can have different types in different contexts, as is illustrated by the following derivations (we have omitted the orders as they can be calculated via Lemma 2.60):

$$\cfrac{\cfrac{\vdash \mathsf{R}(\mathsf{a}_1) : () \qquad \vdash \mathsf{a}_1 : 0}{\mathsf{x} : 0 \vdash \mathsf{R}(\mathsf{x}) : (0)}\,3}{\mathsf{x} : 0, \mathsf{z} : (0) \vdash \mathsf{z}(\mathsf{x}) : (0, (0))}\,4$$

versus

$$\cfrac{\cfrac{\vdash \mathsf{R}(\mathsf{a}_1) : ()}{\mathsf{x} : () \vdash \mathsf{x}() : (())}\,4}{\mathsf{x} : (), \mathsf{z} : (()) \vdash \mathsf{z}(\mathsf{x}) : ((), (()))}\,4.$$

Theorem 2.58 and Corollary 2.61 show that our system RTT makes sense, in a certain way: The type of a pf only depends on the context and does not depend on the way in which we derived the type of that pf.

As a corollary of 2.61 we find:

Corollary 2.63 *If $\Gamma \vdash f : t^a$, $\Gamma \vdash k : u^b$, $x{:}u^b \in \Gamma$ and $\Gamma \vdash f[x{:=}k] : t'^{a'}$ then $a \geq a'$.*

PROOF: If $x \notin \text{FV}(f)$ then $f \equiv f[x{:=}k]$ and the corollary follows from Unicity of Types 2.61. If $x \in \text{FV}(f)$ then the variables that occur in $f[x{:=}k]$, occur either in f or in k, and as the order of k is smaller than the order of f ($x \in \text{FV}(f)$, so $b < a$), the corollary follows from the proof of Lemma 2.60. ⊠

2c2 Strong normalisation

We investigate the problem whether there exists (in the situation of Definition 2.45.6) a pf h such that $h \equiv f[y{:=}k]$. We show that this is the case, in Corollary 2.73.

The nonexistence of $f[\vec{x}{:=}\vec{k}]$ can have two reasons:

- The λ-term $(\lambda_{i=1}^{n} x_i.\widetilde{f})\overline{k_1}\cdots\overline{k_n}$ has no β-normal form;
- The λ-term $(\lambda_{i=1}^{n} x_i.\widetilde{f})\overline{k_1}\cdots\overline{k_n}$ has β-normal form H, but there is no $h \in \mathcal{P}$ such that $h \equiv H$.

However, these two things do not occur if we use substitution under the restrictions of Definition 2.45.6. For a proof of this we use the simply typed λ-calculus of Church [30]. This is not only of help for the existence-proof of h, but also shows that RTT can be seen as a subsystem of $\lambda{\rightarrow}$. However, we remark that the orders of RTT are lost in the embedding to $\lambda{\rightarrow}$. A definition of Church's calculus is given in Section Ab of the Appendix. We translated the propositional functions and the ramified types of RTT to terms and types of $\lambda{\rightarrow}$ in Definitions 2.7, 2.21, and 2.40. We now extend the mapping T of 2.40 to contexts:

Definition 2.64 We define a standard context Γ_0 in $\lambda{\rightarrow}$, in which type information on $\vee$, $\neg$, $\forall$ and elements of $\mathcal{A}$ and $\mathcal{R}$ is stored:

$$\begin{aligned}
\Gamma_0 \stackrel{\text{def}}{=} & \{\neg : o \rightarrow o, \vee : o \rightarrow o \rightarrow o\} \cup \\
& \{a : \iota \mid a \in \mathcal{A}\} \cup \\
& \{\forall_{t^a} : T(t^a) \rightarrow o \mid t^a \text{ is a ramified type}\} \cup \\
& \{R : \underbrace{\iota \rightarrow \ldots \rightarrow \iota}_{m \text{ times } \iota} \rightarrow o \mid R \in \mathcal{R}, \mathfrak{a}(R) = m\}.
\end{aligned}$$

If Γ is a context in RTT then $T(\Gamma) \stackrel{\text{def}}{=} \Gamma_0 \cup \{x : T(t^a) \mid x{:}t^a \in \Gamma\}$.

In particular, $T(\varnothing) = \Gamma_0$.

Theorem 2.65 *If* $\Gamma \vdash f : t^a$ *then*

1. $T(\Gamma) \vdash_{\lambda\rightarrow} \widetilde{f} : o$;
2. $T(\varnothing) \vdash_{\lambda\rightarrow} \overline{f} : T(t^a)$.

PROOF: A straightforward induction on $\Gamma \vdash f : t^a$ with the use of Theorem 2.58 and the Subject Reduction property for λ-Church (A.30). ⊠

Remark 2.66 Observe that the above theorem immediately excludes the pf that leads to the Russell Paradox from the well-typed pfs: If $\neg\mathbf{z}(\mathbf{z})$ were legal then the λ-term $\neg(\mathbf{zz})$ would be typable in λ-Church, which is not the case (see [5]).

Using the strong normalisation of λ-Church (A.36), it is easy to solve the first problem:

Theorem 2.67 *Take $i \leq n$. Assume $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$ and $\Gamma \vdash k : t_i^{a_i}$ (so: the preconditions of rule 6 of* RTT *are fulfilled). Then $(\lambda y{:}T(t_i^{a_i}).\widetilde{f})\overline{k}$ is strongly normalising.*

PROOF: The theorem is easy for $k \in \mathcal{A}$, so assume $k \in \mathcal{P}$. By Theorem 2.65.2 we know that $T(\varnothing) \vdash \overline{k} : T(t_i^{a_i})$, hence $T(\Gamma) \vdash \overline{k} : T(t_i^{a_i})$ by weakening. As, by Theorem 2.65.1, $T(\Gamma) \cup \{y{:}T(t_i^{a_i})\} \vdash \widetilde{f} : o$ and therefore $T(\Gamma) \vdash \lambda y{:}T(t_i^{a_i}).\widetilde{f} : T(t_i^{a_i}) \to o$, we have $T(\Gamma) \vdash (\lambda y{:}T(t_i^{a_i}).\widetilde{f})\overline{k} : o$; so the term $(\lambda y{:}T(t_i^{a_i}).\widetilde{f})\overline{k}$, being a typable term in $\lambda\!\to$, is strongly normalising. ⊠

The second problem is harder to tackle: substitution (Definition 2.24) is defined in terms of λ-calculus, and not every λ-term H has an equivalent h in $\mathcal{P}$ with $\widetilde{h} \equiv H$. This makes it hard to see what happens, especially in case of a substitution

$$z(h_1, \ldots, h_m)[x_1, \ldots, z, \ldots, x_n{:=}k_1, \ldots, k, \ldots, k_n].$$

For this substitution we must calculate the β-normal form of

$$(\lambda x_1{:}\tau_1 \cdots z{:}\tau \cdots x_n{:}\tau_n.z\overline{h_1} \cdots \overline{h_m})\overline{k_1} \cdots \overline{k} \cdots \overline{k_n}.$$

This term reduces to $\overline{k}H_1 \cdots H_m$ for some β-normal H_js. If $k \in \mathcal{P}$ then this new term may not be in β-normal form, and it is not clear what will be the final result (cf. Examples 2.25.4 and 2.25.5.).

The problem clearly has to do with the special structure of λ-terms H for which there is a (legal) $h \in \mathcal{P}$ with $\widetilde{h} \equiv H$. Such terms $\widetilde{h}$ have one important property: All variables are either arguments of functions, or they are applied to the maximal number of arguments that is possible according to the type of that variable. For instance: if a term $\widetilde{h}$ is of the form $zH_1 \cdots H_m$, then the type of z will be of the form $\tau_1 \to \cdots \to \tau_m \to o$.

We call such terms *fully applied* and give the following formal definition:

Definition 2.68 (Fully applied λ-terms) Let Γ be a $\lambda{\to}$-context, and let M be a Γ-legal term of type t. Write $M \equiv M_0 M_1 \cdots M_m$, where M_0 is either a variable or a term of the form $\lambda x{:}\tau.M_0'$. We define the notion *M is Γ-fully applied* by induction on the length of M:

- If M_0 is a variable then M is Γ-fully applied if M has type o in Γ, and for $1 \leq i \leq m$, either M_i is Γ-fully applied, or M_i is a variable;
- If $M_0 \equiv \lambda x{:}\tau.M_0'$ then M is Γ-fully applied if M_0' is $(\Gamma, x{:}\tau)$-fully applied, and for $1 \leq i \leq m$, either M_i is Γ-fully applied, or M_i is a variable.

If it is clear which context Γ is used, we just write *fully applied* instead of *Γ-fully applied.*

It will be shown that for each legal propositional function f, $\widetilde{f}$ and $\overline{f}$ are fully applied. This can be done by induction on the derivation of $\Gamma \vdash f : t^a$. For the substitution case, we need some additional properties of fully applied terms.

Lemma 2.69 *If M is $(\Gamma, y{:}\tau, \Delta)$-fully applied, and N is (Γ, Δ)-fully applied, then $M[y{:=}N]$ is (Γ, Δ)-fully applied.*

PROOF: Induction on the structure of M.

- $M \equiv xM_1 \cdots M_m$.
  - $x \equiv y$. Then $M[y{:=}N] \equiv NM_1[y{:=}N] \cdots M_m[y{:=}N]$. Distinguish:
    * $N \equiv zN_1 \cdots N_n$. Notice that $\Gamma, \Delta \vdash N : o$. This means that $\Gamma, y{:}\tau, \Delta \vdash y : o$, and therefore $m = 0$ and $M[y{:=}N] \equiv N$, thus: $M[y{:=}N]$ is fully applied;

    * $N \equiv (\lambda x{:}v.N')N_1 \cdots N_n$. As N is fully applied, N' is fully applied, and the N_js are either variables or they are fully applied. By induction, $M_i[y{:=}N]$ is either a variable or fully applied for $1 \leq i \leq m$. This means that

$$(\lambda x{:}v.N')N_1 \cdots N_n M_1[y{:=}N] \cdots M_m[y{:=}N]$$

      is fully applied;

  - $x \not\equiv y$. By induction, $M_i[y{:=}N]$ is either a variable or fully applied for $1 \leq i \leq m$. By the Substitution Lemma (A.26), $xM_1[y{:=}N] \cdots M_m[y{:=}N]$ has type o in (Γ, Δ). This means that $xM_1[y{:=}N] \cdots M_m[y{:=}N]$ is fully applied;

- $M \equiv (\lambda x{:}v.M')M_1 \cdots M_m$. By induction, $M'[y{:=}N]$ is $(\Gamma, \Delta, x{:}v)$-fully applied, and $M_i[y{:=}N]$ is either a variable, or fully applied. Therefore $M[y{:=}N]$ is fully applied.

⊠

Lemma 2.70 *If M is Γ-fully applied and $M \to_\beta M'$, then M' is Γ-fully applied.*

PROOF: Induction on the structure of M.

- $M \equiv xM_1 \cdots M_m$. The reduction must occur in a term M_i, say: $M_i \to_\beta M_i'$. As M_i cannot be a variable, M_i is fully applied. By the induction hypothesis, M_i' is fully applied. M has type o, so by Subject Reduction A.30, M' has type o. Hence M' is fully applied;

- $M \equiv (\lambda x{:}v.M_0)M_1 \cdots M_m$. If the reduction occurs within $M_1, \ldots,$ M_m or M_0 then we can give a similar argument as in (1). Now assume $M' \equiv M_0[x{:=}M_1]M_2 \cdots M_m$. Observe: M_0 is $(\Gamma, x{:}v)$-fully applied. If M_1 is a variable, then clearly $M_0[x{:=}M_1]$ is Γ-fully applied. If M_1 is not a variable then M_1 is Γ-fully applied, so by Lemma 2.69, $M_0[x{:=}M_1]$ is Γ-fully applied. Distinguish:

  - $M_0[x{:=}M_1] \equiv yN_1 \cdots N_n$. As $M_0[x{:=}M_1]$ is fully applied, it has type o. This means that $m = 1$, and that $M' \equiv M_0[x{:=}M_1]$ is fully applied;

- $M_0[x{:=}M_1] \equiv (\lambda y{:}\tau.N)N_1 \cdots N_n$. As $M_0[x{:=}M_1]$ is fully applied, N is fully applied, and the N_js are either variables or fully applied terms. Therefore $M' \equiv (\lambda y{:}\tau.N)N_1 \cdots N_n M_2 \cdots M_m$ is fully applied.

⊠

Now we can prove:

Lemma 2.71 *Let $f \in \mathcal{P}$. If $\Gamma \vdash f : t^a$ then $\widetilde{f}$ is $T(\Gamma)$-fully applied, and $\overline{f}$ is $T(\varnothing)$-fully applied.*

PROOF: We prove that $\widetilde{f}$ is $T(\Gamma)$-fully applied. Then it easily follows that $\overline{f}$ is $T(\varnothing)$-fully applied. We use induction on the derivation of $\Gamma \vdash t^a$. All cases are easy to check, except for the abstraction-from-parameters and the substitution cases:

3. (abstraction from parameters) We use notations as in Definition 2.45.3. By induction on the structure of f it is shown that if $\widetilde{f}$ and $\widetilde{g}$ are fully applied, then $\widetilde{h}$ is fully applied;
6. (substitution) We use notations as in Definition 2.45.6. Let $h \equiv f[y{:=}k]$. If $k \in \mathcal{A}$ then h is just f in which all free occurrences of y have been replaced by k. It is then easy to see that $\widetilde{h}$ is fully applied. Now suppose $k \in \mathcal{P}$. By the induction hypothesis, $\widetilde{f}$ and $\widetilde{g}$ are fully applied. This means that $(\lambda y{:}\widetilde{f})\widetilde{g}$ is fully applied. This term β-reduces to $\widetilde{h}$. By Lemma 2.70, $\widetilde{h}$ is fully applied.

⊠

We now know that each legal pf f gives rise to fully applied λ-terms $\widetilde{f}$ and $\overline{f}$. This is of great help in showing that substitutions always exist in case of the application of rule 6 of Definition 2.45. We first show that each substitution in RTT that gives rise to a β-reduction path starting with a fully applied λ-term, really exists. Then it is easy to show that substitutions always exist in the situation of Definition 2.45.6.

Lemma 2.72 *Let $f \in \mathcal{P}$, let $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, and assume that $(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f})\overline{k_1} \cdots \overline{k_n}$ is $T(\Gamma)$-fully applied, where Γ is a* RTT*-context. Then there is $h \in \mathcal{P}$ such that $h \equiv f[x_1, \ldots, x_n{:=}k_1, \ldots, k_n]$.*

PROOF: Notice that $(\lambda_{i=1}^n x_i{:}t_i.\widetilde{f})\overline{k_1}\cdots\overline{k_n}$ is a legal term of $\lambda{\rightarrow}$, and therefore strongly normalising. Let q be the length of the longest reduction path of this term. Use induction on q, so assume that the lemma has been proved for all $q' < q$ (write IH1 for the induction hypothesis). We use induction on the structure of f (and write IH2 for this induction hypothesis). Some cases can be handled directly, for other cases we need help of Lemma 2.70.

1. $f \equiv R(i_1,\dots,i_{\mathfrak{a}(R)})$. Define

$$i'_j \equiv \begin{cases} k_\ell & \text{if } i_j \equiv x_\ell; \\ i_j & \text{if } i_j \notin \{x_1,\dots,x_n\}. \end{cases}$$

Observe that $i'_j \in \mathcal{A} \cup \mathcal{V}$: otherwise there would have been $k_\ell \in \mathcal{P}$ such that $i_j \equiv x_\ell$. But as R has type $\iota \to \dots \to \iota \to o$, x_ℓ must have type ι, and therefore $\overline{k_\ell}$ has type ι, which means that k_ℓ cannot be a pf (Theorem 2.58, Theorem 2.65).

Let $f' \equiv R(i'_1,\dots,i'_{\mathfrak{a}(R)})$. As none of the i'_js is a pf, we have $f' \in \mathcal{P}$. Observe:

$$\left(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n} \twoheadrightarrow_\beta \widetilde{f'},$$

so $f[x_1,\dots,x_n{:=}k_1,\dots,k_n]$ exists and is equal to f';

2. $f \equiv f_1 \vee f_2$. Notice: $\widetilde{f}$ is fully applied and $\widetilde{f} \equiv \underline{\vee}\widetilde{f_1}\widetilde{f_2}$, so $\widetilde{f_1}$ and $\widetilde{f_2}$ are fully applied. Therefore, $(\lambda_{i=1}^n x_i{:}t_i.\widetilde{f_j})\overline{k_1}\cdots\overline{k_n}$ is fully applied for $j = 1,2$. By the induction hypothesis[22], there are $h_1, h_2 \in \mathcal{P}$ such that

$$f_j[x_1,\dots,x_n{:=}k_1,\dots,k_n] \equiv h_j$$

for $j = 1,2$. This means that

$$\left(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n} \twoheadrightarrow_\beta \underline{\vee}\widetilde{h_1}\widetilde{h_2}$$

and therefore $f[x_1,\dots,x_n{:=}k_1,\dots,k_n] \equiv h_1 \vee h_2$.

A similar proof can be given for $f \equiv \neg f'$;

[22]Observe that the longest reduction path of $(\lambda x_1{:}t_1\cdots x_n{:}t_n.\widetilde{f_j})\overline{k_1}\cdots\overline{k_n}$ has a length $\leq q$. If the length is equal to q then use IH2; otherwise use IH1.

3. $f \equiv \forall x{:}t^a[f']$. Notice that $\widetilde{f} \equiv \forall_{t^a}(\lambda x{:}T(t^a).\widetilde{f'})$. As $\widetilde{f}$ is fully applied, $\widetilde{f'}$ is fully applied as well. This means that $(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f'})\overline{k_1}\cdots\overline{k_n}$ is fully applied. By the induction hypothesis[22], there is $h \in \mathcal{P}$ such that $f'[x_1,\ldots,x_n{:=}k_1,\ldots,k_n] \equiv h$. This means that

$$\left(\mathop{\lambda}_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n} \twoheadrightarrow_\beta \forall_{t^a}\lambda x{:}t^a.\widetilde{h}$$

and therefore $f[x_1,\ldots,x_n{:=}k_1,\ldots,k_n] \equiv \forall x{:}t^a[h]$. As $\widetilde{f},\widetilde{k_1},\ldots,\widetilde{k_n}$ are all λI-terms, $x \in \text{FV}(\widetilde{h})$, so $x \in \text{FV}(h)$, which means that $\forall x{:}t^a[h]$ is indeed a pf;

4. $f \equiv z(h_1,\ldots,h_m)$. If $z \notin \{x_1,\ldots,x_n\}$ then we can give a proof similar to the case $f \equiv R(i_1,\ldots,i_{\mathfrak{a}(R)})$. Now assume $z \equiv x_p$. Define

$$h'_j \equiv \begin{cases} k_\ell & \text{if } h_j \equiv x_\ell; \\ h_j & \text{if } h_j \notin \{x_1,\ldots,x_n\}. \end{cases}$$

Notice that $\widetilde{f}$ is fully applied. As $\widetilde{f}$ starts with a variable (this is due to the definition of $\widetilde{f}$), it has type o. Therefore,

$$\left(\mathop{\lambda}_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n}$$

has type o as well. Observe:

$$\left(\mathop{\lambda}_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n} \twoheadrightarrow_\beta \overline{k_p}\overline{h'_1}\cdots\overline{h'_m}.$$

Write $K \equiv \overline{k_p}\overline{h'_1}\cdots\overline{h'_m}$. K is fully applied (Lemma 2.70), and has type o (Subject Reduction A.30). Observe that $\overline{k_p} \equiv \lambda_{y=1}^{q} y_j{:}v_j.\widetilde{k_p}$. Hence $\widetilde{k_p}$ is fully applied. By definition of $\widetilde{k_p}$, $\widetilde{k_p}$ starts with a variable. Therefore, $\widetilde{k_p}$ has type o. As K has type o as well, we have that $q = m$, and that K represents the substitution

$$k_p[y_1,\ldots,y_m{:=}h'_1,\ldots,h'_m].$$

The longest reduction path of K is shorter than the longest reduction path of

$$\left(\mathop{\lambda}_{i=1}^{n} x_i{:}t_i.\widetilde{f}\right)\overline{k_1}\cdots\overline{k_n}$$

so we can apply the induction hypothesis IH1 and conclude that there is $h \in \mathcal{P}$ such that $k_p[y_1, \ldots, y_m{:=}h'_1, \ldots, h'_m] \equiv h$. But then also $f[x_1, \ldots, x_n{:=}k_1, \ldots, k_n] \equiv h$.

$\boxtimes$

With this lemma it is easy to show that substitution always exists in the case of RTT-rule 2.45.6.

Theorem 2.73 (Existence of substitution) *If $f \in \mathcal{P}$, y is the ith free variable in f, $\Gamma \cup \{y : t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $\Gamma \vdash k : t_i^{a_i}$, then $f[y{:=}k]$ exists.*

PROOF: Notice that $\widetilde{f}$ and $\overline{k}$ are fully applied. Therefore $(\lambda y{:}T(t_i^{a_i}).\widetilde{f})\overline{k}$ is fully applied. By Lemma 2.72, $f[y{:=}k]$ exists. $\boxtimes$

2c3 Subterm property

The technique of fully applied λ-terms that was used in Section 2c2 to prove the existence of substitution can also be used to prove another important property of type systems for RTT: the Subterm Property. This property states that if a propositional function is typable, then its recursive parameters (see Definition 2.17) are typable as well. If all recursive parameters of a legal pf f are typable, we say that f has the subterm property:

Definition 2.74 Assume $\Gamma \vdash f : t^a$. If for all $h \in \mathrm{RP}(f) \cap \mathcal{P}$ there is $\Delta \supseteq \Gamma$ and a predicative type u^b such that $\Delta \vdash h : u^b$, then f has the *subterm property*. Notation: $\mathrm{SP}(f)$.

Just as in Section 2c2, we prove by induction on the derivation $\Gamma \vdash f : t^a$ that all legal pfs have the subterm property. Again, all cases are easy, except for the substitution rule 2.45.6. This case can be solved using similar techniques as in Section 2c2.

Lemma 2.75 *Let $f \in \mathcal{P}$, let $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, and assume that $(\lambda_{i=1}^n x_i{:}t_i.\widetilde{f})\overline{k_1} \cdots \overline{k_n}$ is fully applied with respect to $T(\Gamma)$, where Γ is a RTT-context. If $\mathrm{SP}(f)$ and $\mathrm{SP}(k_i)$ for all $k_i \in \mathcal{P}$, then*

1. $\mathrm{RP}\Big(f[\vec{x}{:=}\vec{k}]\Big) \subseteq \mathrm{RP}(f) \cup \{k_i \mid 1 \leq i \leq n\} \cup \bigcup_{k_i \in \mathcal{P}} \mathrm{RP}(k_i)$;

2. $\mathrm{SP}\Big(f[\vec{x}{:=}\vec{k}]\Big)$.

PROOF: Clearly, (2) follows from (1). We prove (1) by induction on the length of the reduction path of $(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f})\overline{k_1}\cdots\overline{k_n}$. We use induction on the structure of f, and only treat the interesting case: $f \equiv z(h_1,\ldots,h_m)$, and $z \equiv x_p$. As in the proof of Lemma 2.72, we define

$$h'_j \equiv \begin{cases} k_\ell & \text{if } h_j \equiv x_\ell; \\ h_j & \text{if } h_j \notin \{x_1,\ldots,x_n\}, \end{cases}$$

and prove that $f[x_1,\ldots,x_n{:=}k_1,\ldots,k_n] \equiv k_p[y_1,\ldots,y_m{:=}h'_1,\ldots,h'_m]$. As the reduction path of $(\lambda_{j=1}^{m} y_j{:}u_j.\widetilde{k_p})\overline{h'_1}\cdots\overline{h'_m}$ is shorter than the reduction path of $(\lambda_{i=1}^{n} x_i{:}t_i.\widetilde{f})\overline{k_1}\cdots\overline{k_n}$, we can use the induction hypothesis:[23]

$$\begin{aligned} \mathrm{RP}\Big(f[\vec{x}{:=}\vec{k}]\Big) &\equiv \mathrm{RP}\Big(k_p[\vec{y}{:=}\vec{h}]\Big) \\ &\subseteq \mathrm{RP}(k_p) \cup \{h'_j \mid 1 \leq j \leq m\} \cup \bigcup_{h'_j \in \mathcal{P}} \mathrm{RP}\Big(h'_j\Big) \\ &\subseteq \mathrm{RP}(f) \cup \{k_i \mid 1 \leq i \leq n\} \cup \bigcup_{k_i \in \mathcal{P}} \mathrm{RP}(k_i). \end{aligned}$$

⊠

As a corollary we get:

Corollary 2.76 (Subterm Lemma) *If* $\Gamma \vdash f : t^a$ *then* $\mathrm{SP}(f)$.

PROOF: Induction on $\Gamma \vdash f : t^a$. All cases are easily checked except for the substitution rule, which is proved with Lemma 2.75. ⊠

[23]Note that $\mathrm{RP}(k_p) \subseteq \bigcup_{k_i \in \mathcal{P}} \mathrm{RP}(k_i)$, $\{h'_j \mid 1 \leq j \leq m\} \subseteq \mathrm{RP}(f) \cup \{k_i \mid 1 \leq i \leq n\}$ and $\bigcup_{h'_j \in \mathcal{P}} \mathrm{RP}(h'_j) \subseteq \mathrm{RP}(f) \cup \bigcup_{k_i \in \mathcal{P}} \mathrm{RP}(k_i)$.

2d Legal propositional functions

We recall Definition 2.46: a pf f is called *legal* if $\Gamma \vdash f : t^a$ for some Γ and t^a. We will check whether this definition of legal pf coincides with the definition of formula that was given in the *Principia*. For this purpose we prove a number of lemmas concerning the relation between legal pfs and predicative types.

We do not distinguish between pfs that are α_P-equal, nor between types $(t_1, \ldots, t_n)$ and $(t_{\varphi(1)}, \ldots, t_{\varphi(n)})$ for a bijection φ. This is justified by Corollary 2.59 and by the fact, that pfs that are α_P-equal are supposed to be the same in the *Principia* too.

We define the notion "up to α_P-equality" formally:

Definition 2.77 Let $f \in \mathcal{P}$, Γ a context, t^a a type. *f is of type t^a in the context Γ up to α_P-equality*, notation $\Gamma \vdash f : t^a(\text{mod } \alpha_P)$, if there is $f' \in \mathcal{P}$, a context Γ' and a bijection $\varphi : \mathcal{V} \to \mathcal{V}$ such that

- $\Gamma' \vdash f' : t^a$;
- f' and f are α_P-equal via the bijection φ;
- $\Gamma' = \{\varphi(x){:}u^b \mid x{:}u^b \in \Gamma\}$.

We say that f is *legal* in the context Γ *up to α_P-equality* if there is a type u^b such that $\Gamma \vdash f : u^b(\text{mod } \alpha_P)$. We say that f is *legal up to α_P-equality* if there is a context Γ such that f is legal in Γ up to α_P-equality.

The following lemma states that all predicative types are "inhabited":

Lemma 2.78 *If t^a is predicative then there are f, Γ such that $\Gamma \vdash f : t^a$.*

PROOF: We use induction on predicative types.

The case $t = 0^0$ is trivial.

Now assume $t = (t_1^{a_1}, \ldots, t_m^{a_m})^a$. By induction there are f_i and Γ_i such that $\Gamma_i \vdash f_i : t_i^{a_i}$ for all $i \leq m$. Take a fixed i. We shall find a context Δ_i and a legal pf g_i such that $\Delta_i \vdash g_i{:}(t_i^{a_i})^{a_i+1}$. Distinguish two cases:

- $t_i^{a_i} = 0^0$. Then make the following derivation:

$$\frac{\Gamma_i \vdash \mathtt{R}(f_i) : ()^0 \qquad \Gamma_i \vdash f_i : 0^0}{\Gamma_i, z_i{:}0^0 \vdash \mathtt{R}(z_i) : (0^0)^1} 3.$$

Write $\Delta_i \equiv \Gamma_i \cup \{z_i{:}0^0\}$, and $g_i \equiv \mathrm{R}(z_i)$, then $\Delta_i \vdash g_i{:}(t_i^{a_i})^{a_i+1}$;

- $t_i \neq 0^0$, say $t_i^{a_i} = (u_1^{b_1}, \dots, u_n^{b_n})^{a_i}$. Because of Theorem 2.58, f_i has n free variables, say $x_1 < \cdots < x_n$, such that $x_j{:}u_j^{b_j} \in \Gamma_i$. Now use rule 4 of RTT:

$$\frac{\Gamma_i \vdash f_i : t_i^{a_i}}{\Gamma_i' \vdash z_i(x_1, \dots, x_n) : (u_1^{b_1}, \dots, u_n^{b_n}, t_i^{a_i})^{a_i+1}}4 \qquad (1)$$

where $\Gamma_i' \equiv \{x_j{:}u_j^{b_j} \mid 1 \leq j \leq n\} \cup \{z_i{:}t_i^{a_i}\}$. Use rule 8 n times:

$$\{z_i{:}t_i^{a_i}\} \vdash \forall x_1{:}u_1^{b_1}[\cdots \forall x_n{:}u_n^{b_n}[z_i(x_1, \dots, x_n)] \cdots] : (t_i^{a_i})^{a_i+1}.$$

Write $\Delta_i \equiv \{z_i{:}t_i^{a_i}\}$ and

$$g_i = \forall x_1{:}u_1^{b_1}[\cdots \forall x_n{:}u_n^{b_n}[z_i(x_1, \dots, x_n)] \cdots].$$

For arbitrary $i = 1, \dots, m$ we now have: $\Delta_i \vdash g_i : (t_i^{a_i})^{a_i+1}$.

We can assume that $x < y$ for $x \in \mathrm{dom}(\Delta_i)$, $y \in \mathrm{dom}(\Delta_j)$ with $i < j$. Write $\Delta \equiv \Delta_1 \cup \dots \cup \Delta_m$. Now apply rule 2 consecutively $m-1$ times, to obtain:

$$\Delta \vdash g_1 \vee (g_2 \vee \dots \vee (g_{m-1} \vee g_m) \dots) : (t_1^{a_1}, \dots, t_m^{a_m})^{\max(a_1, \dots, a_m)+1}.$$

Notice that $(t_1^{a_1}, \dots, t_m^{a_m})^a$ is predicative, so $a = \max(a_1, \dots, a_m) + 1$. ⊠

Remark 2.79 From a modern point of view, this is a remarkable lemma. Many modern type systems are based on the principle of propositions-as-types (see Chapter 4). In such systems types represent propositions, and terms inhabiting such a type represent proofs of that proposition. In a propositions-as-types based system in which all types are inhabited, all propositions are provable. Such a system would be (logically) inconsistent. RTT is not based on propositions-as-types, and there is nothing paradoxical or inconsistent in the fact that all RTT-types are inhabited.

This lemma can be generalised to some non-predicative types:

Corollary 2.80 *If $(t_1^{a_1}, \dots, t_m^{a_m})^a$ is a type such that the $t_i^{a_i}$ are all predicative, then there are f and Γ such that $\Gamma \vdash f : (t_1^{a_1}, \dots, t_m^{a_m})^a$.*

PROOF: With Lemma 2.78 we can construct g_1 and Δ_1 such that

$$\Delta_1 \vdash g_1 : (t_1^{a_1}, \dots, t_m^{a_m})^{\max(a_1,\dots,a_m)+1}.$$

Let u^a be a predicative type of order a. Determine, again with Lemma 2.78, g_2 and Δ_2 such that $\Delta_2 \vdash g_2 : u^a$. Assume $x_1 < \cdots < x_n$ are the free variables of g_2 and $x_j{:}u_j^{b_j} \in \Delta_2$. Notice that $u^a = (u_1^{b_1}, \dots, u_n^{b_n})^a$ (Theorem 2.58). Apply rule 8 and weakening n times to obtain:

$$\Delta_2 \vdash \forall x_1{:}u_1^{b_1}[\cdots \forall x_n{:}u_n^{b_n}[g_2]\cdots] : ()^a.$$

We can assume that $x < y$ for all $x \in \text{DOM}(\Delta_1)$ and all $y \in \text{DOM}(\Delta_2)$, so we can use rule 2 to conclude:

$$\Delta_1 \cup \Delta_2 \vdash g_1 \vee \forall x_1{:}u_1^{b_1}[\cdots \forall x_n{:}u_n^{b_n}[g_2]\cdots] : (t_1^{a_1}, \dots, t_m^{a_m})^a.$$

⊠

We can use the same techniques as in the preceding proof to show that $z(k_1, \dots, k_m)$ is legal if $k_1, \dots, k_m$ are either legal pfs or variables, and z is "fresh".

Lemma 2.81 *If $k_1, \dots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, $t^a = (t_1^{a_1}, \dots, t_n^{a_n})^a$ is a predicative type, $\Gamma \vdash k_i : t_i^{a_i}$ for all $k_i \in \mathcal{A} \cup \mathcal{P}$ and $k_i : t_i^{a_i} \in \Gamma$ for all $k_i \in \mathcal{V}$, and $z \in \mathcal{V} \setminus \text{dom}(\Gamma)$, then $z(k_1, \dots, k_n)$ is legal in the context $\Gamma \cup \{z : t^a\}$ (up to $\alpha_{\mathcal{P}}$-equality).*

PROOF: First, we make a derivation of

$$\{z{:}t^a\} \cup \{x_i{:}t_i^{a_i} \mid 1 \le i \le n\} \vdash z(x_1, \dots, x_n) : (t_1^{a_1}, \dots, t_n^{a_n}, t^a)^{a+1} (\text{mod } \alpha_{\mathcal{P}}),$$

similarly to the derivation of (1) in the proof of Lemma 2.78. Next, find (with Lemma 2.78) $k_1', \dots, k_n'$ such that

- $k_i' \equiv k_i$ if $k_i \in \mathcal{A} \cup \mathcal{P}$;
- $k_i' \in \mathcal{A} \cup \mathcal{P}$ has type $t_i^{a_i}$ in a context Δ_i if $k_i \in \mathcal{V}$;
- Δ, the union of the contexts Γ and the Δ_is, is a context;

- For $k_i \in \mathcal{V}$: k'_i and k'_j are α_Γ-equal if and only if $k_i \equiv k_j$.[24]

Apply rule 6 n times (as in the proof of Lemma 2.78), and where necessary the weakening rule, to obtain:

$$\{z{:}t^a\} \cup \Delta \vdash z(k'_1, \ldots, k'_n) : (t^a)^{a+1}(\text{mod } \alpha_P).$$

Now introduce, with rule 3, new variables for the k'_i which are not equal to k_i, to obtain a legal pf that is α_P-equal to $z(k_1, \ldots, k_m)$. ⊠

It is also not hard to show that $f \vee g$ is legal if f and g are (see also Remark 2.49):

Lemma 2.82 *If f and g are legal in contexts Γ_1 and Γ_2, respectively, and $\Gamma_1 \cup \Gamma_2$ is a context, then $f \vee g$ is legal in the context $\Gamma_1 \cup \Gamma_2$ (up to α_P-equality).*

PROOF: For reasons of clarity, we again leave out the orders of the ramified types. We can not simply apply rule 2 of RTT, as the contexts Γ_1 and Γ_2 may not obey to the condition on them in rule 2. Assume $\Gamma_1 \vdash f : (t_1, \ldots, t_m)$, $\Gamma_2 \vdash g : (u_1, \ldots, u_n)$, and $x_1 < \cdots < x_m$ and $y_1 < \cdots < y_n$ are the free variables of f and g, respectively. Write $t = (t_1, \ldots, t_m)$ and $u = (u_1, \ldots, u_n)$. Take variables $x'_1 < \ldots x'_m < z_1 < y'_1 < \cdots < y'_n < z_2$ not occurring in the domain of $\Gamma_1 \cup \Gamma_2$; let

$$\Delta_1 = \{z_1{:}t, x'_1{:}t_1, \ldots x'_m{:}t_m\};$$

$$\Delta_2 = \{z_2{:}u, y'_1{:}u_1, \ldots y'_n{:}u_n\}.$$

Similar to the derivation (1) in the proof of Lemma 2.78, we can derive

$$\Delta_1 \vdash z_1(x'_1, \ldots, x'_m) : (t_1, \ldots, t_m, t);$$

$$\Delta_2 \vdash z_2(y'_1, \ldots, y'_n) : (u_1, \ldots, u_n, u).$$

As Δ_1 and Δ_2 obey to the conditions of rule 2 of RTT, we can derive

$$\Delta_1 \cup \Delta_2 \vdash z_1(x'_1, \ldots, x'_m) \vee z_2(y'_1, \ldots, y'_n) : (t_1, \ldots, t_m, t, u_1, \ldots, u_n, u).$$

[24]One might wonder whether there are enough pfs of one type that are not α_Γ-equal. Lemma 2.78 provides only one pf for each type. But if we have that pf, say k, then we use rule 2 of RTT to create $\neg k$, $\neg\neg k$, $\neg\neg\neg k$, etc. k, $\neg k$, $\neg\neg k$, ... are all α_Γ-different and of the same type as k.

With similar techniques as in the proof of Lemma 2.81 we can now derive

$$\Gamma_1 \cup \Gamma_2 \cup \{z_1{:}t, z_2{:}u\} \vdash z_1(x_1, \ldots, x_m) \vee z_2(y_1, \ldots, y_n) : v(\text{mod } \alpha_P)$$

for a certain type v (notice that the sets $\{x_1', \ldots, x_m'\}$ and $\{y_1', \ldots, y_n'\}$ do not overlap, whilst the sets $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_n\}$ may overlap). Use rule 6 twice: Substitute f for z_1 and substitute g for z_2. This gives a derivation of $f \vee g$ in the context $\Gamma_1 \cup \Gamma_2$ (mod α_P). ⊠

The following lemma is easy to prove and will be used in the proof of the main result of this section.

Lemma 2.83 *If $R(i_1, \ldots, i_{\mathfrak{a}(R)})$ is a pf with free variables $x_1 < \cdots < x_m$, then it is legal in the context $\{x_j{:}0 \mid 1 \leq j \leq m\}$.*

PROOF: Write $f = R(i_1, \ldots, i_{\mathfrak{a}(R)})$. Let $a_1, \ldots, a_m \in \mathcal{A}$ be m different individuals that do not occur in f, and replace each variable x_j in f by a_j, calling the result f'. By the first rule of RTT, f' is legal in the empty context. Re-introducing the variables $x_1, \ldots, x_m$ (by applying rule 3 of RTT m times) for the individuals $a_1, \ldots, a_m$, respectively, we obtain that f is legal in the context $\{x_j{:}0 \mid 1 \leq j \leq m\}$. ⊠

Finally, we can give a characterisation of the legal pfs:

Theorem 2.84 *Let $f \in \mathcal{P}$. f is legal (mod α_P) if and only if:*

- *$f \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$, or*
- *$f \equiv z(k_1, \ldots, k_n)$, $z \neq k_j$ for all $k_j \in \mathcal{V}$ and z does not occur in any $k_j \in \mathcal{P}$, and there is Γ with $\text{FV}(f) \subseteq \text{DOM}(\Gamma)$ and for all $k_j \in \mathcal{P}$, $\Gamma \vdash k_j{:}t_j^{a_j}$ for some predicative type $t_j^{a_j}$, or*
- *$f \equiv \neg f'$ and f' is legal (mod α_P) or*
- *$f \equiv f_1 \vee f_2$ and there are Γ_i and $t_i^{a_i}$ such that $\Gamma_i \vdash f_i{:}t_i^{a_i}$ (mod α_P) for $i = 1, 2$ and $\Gamma_1 \cup \Gamma_2$ is a context, or*
- *$f \equiv \forall x{:}t^a.f'$ and f' is legal.*

PROOF: Use induction on the structure of f:

- $f \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$. This is Lemma 2.83;
- $f \equiv z(k_1, \ldots, k_n)$. "$\Leftarrow$" is Lemma 2.81. "$\Rightarrow$": f is legal, so there is Γ with $\Gamma \vdash f : t^a$. As $\mathcal{T}(\Gamma) \vdash_{\lambda\rightarrow} z\overline{k_1}\cdots\overline{k_n} : o$ (Theorem 2.65), $z : (u_1^{b_1}, \ldots, u_n^{b_n})^b$ for a predicative type $(u_1^{b_1}, \ldots, u_n^{b_n})^b$, and $\mathcal{T}(\Gamma) \vdash_{\lambda\rightarrow} \overline{k_j} : \mathcal{T}(u_j^{b_j})$. If $z \equiv k_j$ then $z \equiv \overline{k_j}$ and therefore $z{:}u_j^{b_j} \in \Gamma$, which is impossible. By Corollary 2.76, each $k_j \in \mathcal{P}$ is typable in Γ, and as $\mathcal{T}(\Gamma) \vdash_{\lambda\rightarrow} \overline{k_j} : \mathcal{T}(u_j^{b_j})$ and the type of k_j is predicative, $\Gamma \vdash k_j{:}u_j^{b_j}$. Notice that $b_j < b$, so it is impossible that z occurs in a $k_j \in \mathcal{P}$;
- "$\Leftarrow$" is Rule 2 of RTT (for $\neg$) and Lemma 2.82 (for $\vee$). "$\Rightarrow$" (for $\vee$; the proof for $\neg$ is similar): Let Δ be the context containing all the variables of f (also those that are bound by a quantifier; we can assume that different quantifiers bind different variables) and their types. f is built from several pfs of the form $R(i_1, \ldots, i_{\mathfrak{a}(R)})$ and $z(k_1, \ldots, k_m)$ (we will call these pfs the *constituents* of f), and the logical connectives $\neg$, $\vee$ and $\forall$. Reasoning as in the "$\Rightarrow$" part of the first two cases of the proof of this lemma, we can show the preconditions for Lemma 2.83 (for constituents of the form $R(i_1, \ldots, i_{\mathfrak{a}(R)})$) and Lemma 2.81 (for constituents of the form $z(k_1, \ldots, k_m)$). Applying these Lemmas, we find that any constituent h of f is typable in Δ. Using Rule 2 of RTT (for $\neg$), Rule 8 of RTT (for $\forall$) and Lemma 2.82 (for $\vee$), we find that $f_1 \vee f_2$ itself is typable;
- "$\Leftarrow$" is Rule 8 of RTT. "$\Rightarrow$" is similar to "$\Rightarrow$" in the previous case.

⊠

We can now answer the question whether our legal pfs (as defined in 2.46) are the same as the formulas of the *Principia*.

First of all, we must notice that all the legal pfs from Definition 2.46 are also formulas of the *Principia*: This was motivated in Remark 2.47.

Moreover, we proved (in 2.84) that if f is a pf, then the only reasons why f cannot be legal (according to Definition 2.46) are:

- There is a constituent $z(k_1, \ldots, k_m)$ of f in which z occurs in one of the k_i's;
- There is a constituent $z(k_1, \ldots, k_m)$ of f and a $j \in \{1, \ldots, m\}$ such that k_j is a pf, but not a legal pf;

- f contains two non-overlapping constituents f_1, f_2 that cannot be typed in one and the same context.

Pfs of the first type cannot be legal in the *Principia*, because of the vicious circle principle. The same holds for pfs of the second type, because also in the *Principia*, parameters cannot be untyped. The third problem is a non-issue in the *Principia*. Formal contexts are not present in the *Principia*, but have been introduced in this Chapter to make a precise analysis of RTT possible. Propositional functions of the *Principia* are always constructed in one, implicitly defined, context. A formula, therefore, cannot contain two non-overlapping constituents that cannot be typed in the same context. This excludes pfs of the third type.

We conclude that we have described the legal pfs of the *Principia Mathematica* with the formal system RTT.

We present some refinements of Theorem 2.84 that will be useful in future chapters of this thesis:

Theorem 2.85 *Assume* $\Gamma \vdash f : t^a$.

- *If* $f \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$ *and* $x \in \mathrm{FV}(f)$ *then* $x{:}0^0 \in \Gamma$;
- *If* $f \equiv z(k_1, \ldots, k_m)$ *then there are* $u_1^{b_1}, \ldots, u_m^{b_m}, b$ *such that*
  - $z{:}\left(u_1^{b_1}, \ldots, u_m^{b_m}\right)^b \in \Gamma$;
  - $\Gamma \vdash k_i{:}u_i^{b_i}$ *for* $k_i \in \mathcal{A} \cup \mathcal{P}$;
  - $k_i{:}u_i^{b_i} \in \Gamma$ *for* $k_i \in \mathcal{V}$.

PROOF:

- By Theorem 2.65, $T(\Gamma) \vdash_{\lambda\to} Ri_1 \cdots i_{\mathfrak{a}(R)} : o$. This means that $x{:}\iota \in T(\Gamma)$. Therefore, $x{:}0^0 \in \Gamma$;
- Let $u_1^{b_1}, \ldots, u_m^{b_m}, b$ be as in the proof of Theorem 2.84. We only need to check that $k_i{:}u_i^{b_i} \in \Gamma$ for $k_i \in \mathcal{V}$. We already know that $k_i{:}T(u_i^{b_i}) \in T(\Gamma)$, and as $u_i^{b_i}$ is predicative, and the type of the variable k_i in Γ must be predicative as well, we have $k_i{:}u_i^{b_i} \in \Gamma$.

⊠

	in RTT	in λ-calculus
Strengthening Lemma	2.56.2	A.31
Free Variable Lemma	2.56.1	A.22
Unicity of Types	2.61	A.32
Subterm Lemma	2.76	A.29
Strong Normalisation	2.73	A.36

Figure 3: Comparison of the properties of RTT and modern typed λ-calculus

Conclusions

In this chapter we gave a formalisation of the Ramified Theory of Types. Some of the main ideas underlying this theory were already present in Frege's Abstraction Principles 1.1 and 1.2.

RTT not only prevents the paradoxes of Frege's *Grundgesetze der Arithmetik*, but also guarantees the well-definedness of substitution, as we have shown in Corollary 2.73. This second problem was not realized in the *Principia*, where substitution did not even have a proper definition.

There is a close relation between substitution in *Principia* and β-reduction in λ-calculus (Definition 2.24). RTT has characteristics that are also the basic properties of modern type systems for λ-calculus. See Figure 3. As there is no real reduction in RTT, we don't have an equivalent of the Subject Reduction theorem. However, the fact that the Free Variable property 2.58 is maintained under substitution can be seen as a (very weak) form of Subject Reduction A.30.

Expressing Russell's propositional functions in λ-calculus has made it possible to compare these pfs with λ-terms. We found that pfs can be seen as λ-terms, but in a rather simple way:

- A pf is always a λI-term, i.e. if $\lambda x{:}A.B$ is a subterm of the translation $\widetilde{f}$ of a pf f, then $x \in \text{FV}(B)$;
- The translation of a pf always results in a fully applied λ-term in β-normal form;
- Substitution in the *Principia* can be seen as application plus β-reduction to normal form.

Although the description of the Ramified Theory of Types in the *Principia* is very informal, it is remarkable that an accurate formalisation of this system can be made (see Theorem 2.84 and the discussion that follows it). The formalisation shows that Russell and Whitehead's ideas on the notion of types, though very informal to modern standards, must have been very thorough and to the point.

Apart from the orders, RTT is a subsystem of $\lambda{\rightarrow}$, [30] via the embeddings $\bar{\ }$ of Section 2a2 and T of Section 2c2. There are, however, important differences between the way in which the type of a pf is determined in RTT, and the way in which the type of a λ-term is determined in λ-Church. The rules of RTT, and the method of deriving the types of pfs that was presented in Section 2d, have a bottom-up character: one can only introduce a variable of a certain type in a context Γ, if there is a pf that has that type in Γ. In $\lambda{\rightarrow}$, one can introduce variables of any type without wondering whether such a type is inhabited or not.

Church's $\lambda{\rightarrow}$ is more general than RTT in the sense that Church does not only describe (typable) *propositional* functions. In $\lambda{\rightarrow}$, also functions of type $\tau \rightarrow \iota$ (where ι is the type of individuals) can be described, and functions that take such functions as arguments, etc..

A characteristic of RTT that is maintained in many modern type systems is the syntactic nature of the system: type and order of a pf are determined on purely syntactical grounds. No attention is paid to the interpretation of such a pf. This is remarkable, as the propositions $\forall \mathtt{x}{:}0^0[\mathtt{R}(\mathtt{x})]$ and $\forall \mathtt{x}{:}0^0[\mathtt{R}(\mathtt{x})] \vee \forall \mathtt{z}{:}()^9[\mathtt{z}() \wedge \neg\mathtt{z}()]$ are logically equivalent in most logics[25], though they are of different type (the former pf has type $()^1$ and the latter has type $()^{10}$). In Section 3c we show that other viewpoints are possible besides this concentration on syntax.

[25]At least in all the logical systems that Russell had in mind when he wrote the *Principia*

Chapter 3

Deramification

In this chapter we discuss the development of type theory in the period between the appearance of *Principia Mathematica* (1910-1912) and Church's formulation of the Simple Theory of Types [30] in 1940.

In Section 3a we show that RTT was not a very easy system to work with. Ramsey [101], and Hilbert and Ackermann [64], simplified the system by removing the orders. The result is known as the *Simple Theory of Types* (STT).

Nowadays, STT is known via Church's formalisation in λ-calculus. However, STT already existed (1926) before λ-calculus did (1932), and is therefore not inextricably bound up with λ-calculus. In Section 3b we show how we can obtain a formalisation of STT directly from the formalisation of RTT that was presented in Chapter 2 by simply removing the orders. Most of the properties that were proved for RTT hold for STT as well, including Unicity of Types and Strong Normalisation. The proofs are all similar to the proofs that were given for RTT. We also make a comparison between Church's formalisation in λ-calculus and the formalisation of STT that is obtained from RTT. It appears that Church's system is much more than only a formalisation. Because of the λ-calculus it is more expressive.

The removal of orders from type theory may suggest that orders are to be blamed for the restrictiveness of RTT, and that the concept of order is problematic. In Section 3c we show that this is not necessarily the case. We introduce a system KTT, based on Kripke's Hierarchy of Truths [78], that has an approach completely opposite to STT. Whilst STT is *order*-free, and *types* play the main role, Kripke's Hierarchy of Truths is *type*-free, and *orders* play an important, though not a restrictive, role. The main difference between Kripke's and Russell's notion of order is that Russell's classification is purely syntactical, whilst Kripke's is essentially semantical. We show that RTT can be embedded in KTT (3c2), and that there is a straightforward relation between the orders in RTT and the hierarchy of truths of KTT.

3a History of the deramification

3a1 The problematic character of RTT

The main part of the *Principia* is devoted to the development of logic and mathematics using the legal pfs of the ramified type theory. It appears that RTT is not easy to use. The main reason for this is the implementation of the so-called *ramification*: the division of simple types into orders. We illustrate this with two examples:

Example 3.1 (Equality) One tends to define the notion of *equality* in the style of Leibniz ([53]):

$$\mathrm{x} =_{\mathrm{L}} \mathrm{y} \overset{\text{def}}{\leftrightarrow} \forall \mathrm{z}[\mathrm{z}(\mathrm{x}) \leftrightarrow \mathrm{z}(\mathrm{y})],$$

or in words: Two individuals are equal if and only if they have exactly the same properties.

Unfortunately, in order to express this general notion in our formal system, we have to incorporate *all* pfs $\forall \mathrm{z} : (0^0)^n[\mathrm{z}(\mathrm{x}) \leftrightarrow \mathrm{z}(\mathrm{y})]$ for $n > 1$, and this cannot be expressed in one pf.

The ramification does not only influence definitions in logic. Some important mathematical concepts cannot be defined any more:

Example 3.2 Dedekind constructed the real numbers from the rationals using so-called Dedekind cuts. In this construction, a real number is a set r of rationals such that

- $r \neq \varnothing$;
- $r \neq \mathbb{Q}$;

- If $x \in r$ and $y < x$ then $y \in r$;
- If $x \in r$ then there is $y \in r$ with $x < y$.

For instance, the real number $\frac{1}{2}$ is represented by the set $\{x \in \mathbb{Q} \mid 2x < 1\}$, and the real number $\sqrt{2}$ is represented by the set $\{x \in \mathbb{Q} \mid x < 0 \text{ or } x^2 < 2\}$.

If we take $\mathbb{Q}$ as the set of individuals $\mathcal{A}$, and assume that the binary relation $<$ on $\mathbb{Q}$ is an element of $\mathcal{R}$, the set of relations, we can see real numbers as unary predicates f over $\mathbb{Q}$ such that

$$\begin{gathered}\exists \mathtt{x}{:}0^0[\mathtt{z}(\mathtt{x})] \wedge \exists \mathtt{x}{:}0^0[\neg \mathtt{z}(\mathtt{x})] \wedge \\ \forall \mathtt{x}{:}0^0[\forall \mathtt{y}{:}0^0[\mathtt{z}(\mathtt{x}) \rightarrow \mathtt{y} < \mathtt{x} \rightarrow \mathtt{z}(\mathtt{y})]] \wedge \\ \forall \mathtt{x}{:}0^0[\mathtt{z}(\mathtt{x}) \rightarrow \exists \mathtt{y}{:}0^0[\mathtt{z}(\mathtt{y}) \wedge \mathtt{x} < \mathtt{y}]]\end{gathered} \tag{1}$$

holds if we substitute f for $\mathtt{z}$. We will abbreviate the predicate (1) (with the free variable $\mathtt{z}$) as $\mathbb{R}$. It has type $\left((0^0)^1\right)^2$, and real numbers can be seen as pfs of type $(0^0)^1$. We will, for shortness of notation, write $\mathbb{R}(f)$ for $\mathbb{R}[\mathtt{z}{:=}f]$, so $\mathbb{R} \equiv \mathbb{R}(\mathtt{z})$. A real number r is smaller than or equal to another real number r' if for all x with $r(x)$, also $r'(x)$ holds. We write, shorthand, $r \leq r'$ if r is smaller than or equal to r'.

In traditional mathematics, the above would define a system that obeys the traditional axioms for real numbers. In particular, the theorem of the least upper bound holds for this system. This theorem states that each non-empty subset of $\mathbb{R}$ with an upper bound has a least upper bound. In our formalism:

$$\forall \mathtt{v}{\subseteq}\mathbb{R}\left[\begin{array}{c}\left(\begin{array}{c}\exists \mathtt{z}_1{\in}\mathbb{R}[\mathtt{v}(\mathtt{z}_1)] \wedge \\ \exists \mathtt{z}_2{\in}\mathbb{R}\forall \mathtt{z}_3{\in}\mathbb{R}[\mathtt{v}(\mathtt{z}_3) \rightarrow \mathtt{z}_3 \leq \mathtt{z}_2]\end{array}\right) \\ \rightarrow \\ \exists z_1 \in \mathbb{R}\left[\begin{array}{c}\forall \mathtt{z}_2{\in}\mathbb{R}[\mathtt{v}(\mathtt{z}_2) \rightarrow \mathtt{z}_2 \leq \mathtt{z}_1] \wedge \\ \forall \mathtt{z}_3{\in}\mathbb{R}\left[\begin{array}{c}\forall \mathtt{z}_4{\in}\mathbb{R}[\mathtt{v}(\mathtt{z}_4) \rightarrow \mathtt{z}_4 \leq \mathtt{z}_3] \\ \rightarrow \mathtt{z}_1 \leq \mathtt{z}_3\end{array}\right]\end{array}\right]\end{array}\right]$$

(We write, shorthand, $\forall v{\subseteq}\mathbb{R}[g]$ to denote

$$\forall v{:}\left((0^0)^1\right)^2[\forall \mathtt{u}{:}(0^0)^1[v(\mathtt{u}) \rightarrow \mathbb{R}(\mathtt{u})] \rightarrow g],$$

and $\forall z{\in}\mathbb{R}[g]$ to denote $\forall z{:}(0^0)^1[\mathbb{R}(z) \to g]$). If we try to prove this theorem within the system of Dedekind as formulated in the *Principia*-language RTT, we have to specify a type t^a for the variable $\mathbf{z}_1$. As $\mathbf{z}_1$ must be a real number, its type must be $(0^0)^1$. If we give a proof of the theorem, and construct some object f that should be the least upper bound of a set of real numbers V, f will depend on V. Therefore, a general description of f will have a variable v for V in it. As v is of order 2, f must be of order 3 or more. Therefore, f cannot be a real number, since real numbers have order 1. This makes it impossible to give a constructive proof of the theorem of the least upper bound within a ramified type theory.

This is a consequence of the fact that it is not possible in RTT to give a definition of an object that refers to the class to which this object belongs (because of the Vicious Circle Principle). Such a definition is called an *impredicative* definition. The relation with the notion of impredicative *type* is immediate: an object defined by an impredicative definition is of a higher order than the order of the elements of the class to which this object should belong. This means that the defined object has an *impredicative* type.

Nowadays we would consider the use of the Vicious Circle Principle too strict. The impredicative definition of f is a matter of *syntax*, whilst the existence of the object f has to do with *semantics*. The fact that we are not able to give a predicate definition of f does not imply that such an object does not exist. Here we must remark that Russell and Whitehead did not make a distinction between syntax and semantics in the *Principia*.[1] Therefore they had to interpret the Vicious Circle Principle in the strict way above.

3a2 The Axiom of Reducibility

Russell and Whitehead tried to solve these problems with the so-called *axiom of reducibility*.

Axiom 3.3 (Axiom of Reducibility) *For each formula f, there is a formula g with a* predicative *type such that f and g are (logically) equivalent.*

[1]Though the basic ideas for this were already present in the works of Frege. See for instance *Über Sinn und Bedeutung* [49].

Accepting this axiom, one may define equality on formulas of order 1 only:

$$\mathsf{x} =_1 \mathsf{y} \overset{\text{def}}{=} \forall \mathsf{z} : (0^0)^1[\mathsf{z}(\mathsf{x}) \leftrightarrow \mathsf{z}(\mathsf{y})].$$

If f is a function of type $(0^0)^n$ for some $n > 1$, and a and b are individuals for which the Leibniz equality $\mathsf{a} =_{\mathrm{L}} \mathsf{b}$ holds then $f(\mathsf{a}) \leftrightarrow f(\mathsf{b})$ holds: With the Axiom of Reducibility we can determine a predicative function g (so of type $(0^0)^1$), equivalent to f. As g has order 1, $g(\mathsf{a}) \leftrightarrow g(\mathsf{b})$ holds. And because f and g are equivalent, also $f(\mathsf{a}) \leftrightarrow f(\mathsf{b})$ holds. This solves the problem of Example 3.1. A similar solution gives, in Example 3.2, the proof of the theorem of the least upper bound.

The validity of the Axiom of Reducibility has been questioned from the moment it was introduced. In the introduction to the 2nd edition of the *Principia*, Whitehead and Russell admit:

> "This axiom has a purely pragmatic justification: it leads to the desired results, and to no others. But clearly it is not the sort of axiom with which we can rest content."
>
> (*Principia Mathematica*, p. xiv)

Moreover, Weyl states that

> "if the properties are constructed there is no room for an axiom here; it is a question which ought to be decided on ground of the construction"
>
> (Mathematics and Logic: A brief survey serving as preface to a review of "The Philosophy of Bertrand Russell", p. 5)

and that

> "with his axiom of reducibility Russell therefore abandoned the road of logical analysis and turned from the constructive to the existential-axiomatic standpoint."
>
> (*Ibid.*, p. 6)

With the more modern developments of logic in our mind, we could add the following objection, associated to Weyl's argument above, against the Axiom. The Axiom of Reducibility states that for each f, there is a predicative g that is logically equivalent to f. The function g is something at *object* level, but the statement "f is logically equivalent to g" is a statement at a higher level than the object level. Pfs exist (at least, in the syntactic construction) independently from the existence of the notion of logical equivalence.

Moreover, there is more than one notion of logical equivalence, corresponding to the various kinds of logic that have been developed, or could have been developed. It would be remarkable if one Axiom of Reducibility would provide predicative pfs f for any kind of logic that is available, or can be thought of, and indeed, this is not true as shown by the following trivial example:

Example 3.4 We consider so-called "bureaucratic logic". This logic has a set of axioms A, and no derivation rules at all. In short, a proposition g is true if and only if $g \in A$. Take, for the sake of the argument,

$$A = \{g \in \mathcal{P} \mid \vdash g{:}()^0\},$$

so A is the set of all predicative propositions. In this system, a proposition g is true if and only if it is predicative. If f is an impredicative proposition, then so is $f \leftrightarrow g$, for any proposition g. Therefore, $f \leftrightarrow g$ is false for any proposition g, in particular for any predicative proposition g. So the Axiom of Reducibility does not hold in bureaucratic logic.

Though Weyl [120] made an effort to develop analysis within the Ramified Theory of Types (but without the Axiom of Reducibility), and various parts of mathematics can be developed within RTT and without the Axiom[2], the general attitude towards RTT (without the axiom) was that the system was too restrictive, and that a better solution had to be found.

[2]See [67], where many algebraic notions are developed within the Nuprl Proof Development System, a proof checker based on the hierarchy of types and orders of RTT without the Axiom of Reducibility.

3a3 Deramification

The first impulse to such a solution was given by Ramsey in 1926 [101]. He recalls that the Vicious Circle Principle 2.1 was postulated in order to prevent the paradoxes. Though all the paradoxes were prevented by this Principle, Ramsey considers it essential to divide them into two parts:

1. One group of paradoxes is removed

   > "by pointing out that a propositional function cannot significantly take itself as argument, and by dividing functions and classes into a hierarchy of types according to their possible arguments."
   >
   > (The Foundations of Mathematics, p. 356)

   This means that a class can never be a member of itself. The paradoxes solved by introducing the hierarchy of types (but not orders), like the Russell paradox, and the Burali-Forti paradox, are called *logical* or *syntactical* paradoxes;

2. The second group of paradoxes is excluded by the hierarchy of orders. These paradoxes (like the Liar's paradox, and the Richard Paradox) are based on the confusion of language and meta-language. These paradoxes are, therefore, not of a purely mathematical or logical nature. When a proper distinction between object language (the pfs of the system RTT, for example) and meta-language is made, these so-called *semantical* paradoxes disappear immediately.

Ramsey agrees with the part of the theory that eliminates the syntactic paradoxes. This part is in fact RTT without the orders of the types. The second part, the hierarchy of orders, does not gain Ramsey's support: if a proper distinction between object-language and meta-language is made, the semantic paradoxes disappear. Moreover, by accepting the hierarchy in its full extent one either has to accept the Axiom of Reducibility or reject ordinary real analysis. Ramsey is supported in his view by Hilbert and Ackermann [64]. They all suggest a *deramification* of the theory, i.e. leaving out the orders of the types. When making a proper distinction between language and meta-language, the deramification will not lead to a re-introduction of the (semantic) paradoxes.

The solution proposed by Ramsey, and Hilbert and Ackermann, looks better than the Axiom of Reducibility. Nevertheless, both deramification and the Axiom of Reducibility are violations of the Vicious Circle Principle, and reasons (of a more fundamental character than "they do not lead to a re-introduction of the semantic paradoxes" and "it leads to the desired results, and to no others") why these violations can be harmlessly made must be given. Gödel [58] fills in this gap. He points out that whether one accepts this second principle or not, depends on the philosophical point of view that one has with respect to logical and mathematical objects:

> "it seems that the vicious circle principle [...] applies only if the entities involved are constructed by ourselves. In this case there must clearly exist a definition (namely the description of the construction) which does not refer to a totality to which the object defined belongs, because the construction of a thing can certainly not be based on a totality of things to which the thing to be constructed itself belongs. If, however, it is a question of objects that exist independently of our constructions, there is nothing in the least absurd in the existence of totalities containing members, which can be described only by reference to this totality."
>
> (Russell's mathematical logic)

The remark puts the Vicious Circle Principle back from a proposition (a statement that is either true or false, without any doubt) to a philosophical principle that will be easily accepted by, for instance, intuitionists (for whom mathematics is a pure mental construction) or constructivists, but that will be rejected, at least in its full strength, by mathematicians with a more platonic point of view.

Gödel is supported in his ideas by Quine [100], sections 34 and 35. Quine's criticism on impredicative definitions (for instance, the definition of the least upper bound of a nonempty subset of the real numbers with an upper bound) is not on the definition of a special symbol, but rather on the very assumption of the *existence* of such an object at all. Quine continues by stating that even for Poincaré, who was an opponent of impredicative definitions and deramification, one of the doctrines of classes is that they are there "from the beginning". So, even for Poincaré there should be no evident fallacy in impredicative definitions.

The deramification has played an important role in the development of type theory. In 1932 and 1933, Church presented his (untyped) λ-calculus [28, 29]. In 1940 he combined this theory with a deramified version of Russell's theory of types to the system that is known as the *simply typed λ-calculus*[3].

3b The Simple Theory of Types

3b1 Constructing the Simple Theory of Types from RTT

It is straightforward to carry out the deramification as it was originally proposed by Ramsey, Hilbert and Ackermann: We take the formalisation of RTT that was presented in Chapter 2, and leave out all the orders and the references to orders (including the notions of predicative and impredicative types). The system we obtain in this way will be denoted STT. The types used in the system are the simple types of Definition 2.31.

The following definitions, lemmas, theorems and corollaries, including their proofs, can be adapted to STT without any problems: 2.43, 2.44, 2.45, 2.46, 2.56, 2.57 (first free variable theorem), 2.58 (second free variable theorem), 2.59, 2.61 (unicity of types), 2.64, 2.65, 2.67, 2.71, 2.72, 2.73 (existence of substitution), 2.74, 2.75 and 2.76 (subterm lemma).

The description of legal pfs for STT follows the same line as in Section 2d, with straightforward adaptions of 2.77, 2.78 (now, all simple types are inhabited), 2.81, 2.82, 2.83, and finally 2.84 (characterisation of legal pfs):

Theorem 3.5 *Let $f \in \mathcal{P}$. f is legal (mod α) if and only if:*

- $f \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$, *or*
- $f \equiv z(k_1, \ldots, k_n)$, $z \not\equiv k_j$ *for all* $k_j \in \mathcal{V}$ *and* z *does not occur in any* $k_j \in \mathcal{P}$, *and there is* Γ *with* $\text{FV}(f) \subseteq \text{DOM}(\Gamma)$ *and for all* $k_j \in \mathcal{P}$, $\Gamma \vdash k_j{:}t_j$, *or*

[3]Thus, the adjective *simple* is used to distinguish the theory from the more complicated — both in its construction with a double hierarchy and in its use — *ramified* theory. The classification "simple", therefore, has nothing to do with the fact that STT, formulated with λ-calculus as described in [30], is the simplest system of the Barendregt Cube (see Ac).

- *$f \equiv \neg f'$ and f' is legal (mod α) or $f \equiv f_1 \vee f_2$, there are Γ_i and t_i such that $\Gamma_i \vdash f_i{:}t_i$ (mod α) and $\Gamma_1 \cup \Gamma_2$ is a context, or*
- *$f \equiv \forall x{:}t.f'$ and f' is legal.*

A comparison between the formalisations of STT and RTT can easily be made using Theorems 3.5 and 2.84. We find that

- All RTT-legal pfs are (when the ramified types behind the quantifiers are replaced by their corresponding simple types) STT-legal;
- A STT-legal pf f is RTT-legal, except when f contains a subformula of the form $z(k_1, \ldots, k_n)$, where one or more of the k_js are not RTT-legal or can only be typed in RTT by an impredicative type.

3b2 Comparison of STT with Church's $\lambda\!\rightarrow$

Nowadays, the Simple Theory of Types is often identified with Church's formalisation of it in [30]. The definition of $\lambda\!\rightarrow$ that was given there is repeated in Section Ab of the Appendix.

We make the following remarks with respect to $\lambda\!\rightarrow$ and the Simple Theory of Types.

Remark 3.6 We see that the constants $\neg$, $\wedge$, $\forall_\alpha$ and $\imath_\alpha$ are terms. This may need some explanation for the modern reader.

- Church considers $\neg$ and $\wedge$ to be functions. The function $\neg$ takes a proposition as argument, and returns a proposition; similarly $\wedge$ takes two propositions as arguments, and returns a proposition. In Definition A.16, we see that $\neg$ and $\wedge$ are assigned the corresponding types $o \rightarrow o$ and $o \rightarrow o \rightarrow o$;
- More remarkable: $\forall_\alpha$ and $\imath_\alpha$ are just terms, and do not act as binding operators. The usual variable binding of $\forall_\alpha$ and $\imath_\alpha$ is obtained via λ-abstraction: instead of $\forall x{:}\alpha[f]$, Church writes $\forall_\alpha(\lambda x{:}\alpha.f)$. In this way, $\forall_\alpha$ is a function that takes a propositional function of type $\alpha \rightarrow o$ as argument, and returns a proposition (a term of type o). In Definition A.16, $\forall_\alpha$ obtains the corresponding type $(\alpha \rightarrow o) \rightarrow o$. Similarly, the choice operator $\imath_\alpha$ takes a propositional function of type $\alpha \rightarrow o$ as argument, and returns a term of type α. The term

$\imath x{:}\alpha.f$, or in Church's notation: $\imath_\alpha(\lambda x{:}\alpha.f)$, has as interpretation: the (unique) object t of type α for which $f[x{:=}t]$ holds. Correspondingly, the type of $\imath_\alpha$ is $(\alpha \to o) \to \alpha$.

The mappings T for types and $\bar{\ }$ for terms (see Definition 2.40 and Definition 2.7), adapted for STT, make it possible to compare STT with $\lambda\!\!\to$.

Regarding the types, we find that T gives an injective correspondence between types of STT and $\lambda\!\!\to$. T is clearly not surjective, as $T(t)$ is never of the form $\alpha \to \iota$ (this follows directly from Definition 2.34). This indicates an important difference between STT and $\lambda\!\!\to$. In RTT and STT, functions (other than propositional functions) have to be defined via relations (and this is the way it is done in *Principia Mathematica*). The value of such a function f, described via the relation R, for a certain value a is described using the $\imath$-operator: $\imath y.R(a,y)$ (to be interpreted as: the unique y for which $R(a,y)$ holds). Things get even more complicated if one realizes that the $\imath$-operator is not a part of the syntax used in *Principia Mathematica*, but an abbreviation with a not so straightforward translation (see [121], pp. 66–71). In $\lambda\!\!\to$, as everywhere in λ-calculus, functions (both propositional functions and other ones) are first-class citizens, which means that the construction with the $\imath$-operator is not the first tool to be used when constructing a function. If one has an algorithm (a λ-term) that describes the function f, the value of f for the argument a can be easily described via the term fa. And even if such an algorithm is not at hand, one can use the $\imath$-operator, which is part of the syntax of $\lambda\!\!\to$. This makes $\lambda\!\!\to$ much easier to use for the formalisation of logic and mathematics than RTT and STT.

Regarding the terms, $\bar{\ }$ provides an injective correspondence between terms of STT and $\lambda\!\!\to$. Again, this mapping is not surjective, for several reasons:

- T is not surjective. As there is no t with $T(t) = \iota \to \iota$, there cannot be a legal pf f such that $\overline{f} \equiv \lambda \mathrm{x}{:}\iota.\mathrm{x}$ (cf. Theorem 2.65.2 adapted for STT);
- We already observed that $\overline{f}$ is a λI-term for all $f \in \mathcal{P}$. $\lambda\!\!\to$ also allows terms like $\lambda \mathrm{x}{:}\alpha.\mathrm{y}$;
- If $\overline{f} \equiv zH_1 \cdots H_n$ for some $z \in \mathcal{V}$ and some terms $H_1, \ldots, H_n$, the H_is must be either *closed* λ-terms, or variables, or individuals. This

means that there is no $f \in \mathcal{P}$ such that $\overline{f} \equiv \lambda \mathbf{z}{:}o{\to}o.\lambda \mathbf{x}{:}\iota.\mathbf{z}(\mathbf{Rx})$, since $\mathbf{Rx}$ contains the free variable $\mathbf{x}$ and is neither a variable nor an individual;

- We remark that $\overline{f}$ is always a closed λ-term, so there is no $f \in \mathcal{P}$ such that $\overline{f} \equiv \mathbf{x}$;

- It has already been remarked that the $\imath$-operator is part of the syntax of $\lambda{\to}$, and this is not the case in STT and RTT.

The discussion above makes clear that $\lambda{\to}$ is a far more expressive system than RTT and STT. Type-theoretically, it generalises the idea of function types of Frege and Russell from propositional functions to more general functions.

Philosophically, there is another important difference between STT and $\lambda{\to}$. The systems STT and RTT have a strong bottom-up approach: To type a higher-order pf one has to start with propositions of order 0. Only by applying the abstraction principles, it is possible to obtain higher-order pfs. In $\lambda{\to}$, one can introduce a variable of a higher-order type at once, without having to refer to terms of lower order.

3c Are the orders to be blamed?

The historical success of the deramification makes it attractive to conclude that the ramification of Russell's theory is to be "blamed" for the restrictiveness of RTT: the orders were just an emergency measure, and by removing them from the theory, everything works fine. This reasoning, however, is a bit too fast. Orders still play a role in logic, and they provide a useful intuition to describe how complicated a certain proposition is (for example "first-order", or "second-order").

Moreover, we feel that there are reasons to criticise not the *concept* of order, but Russell's *definition* of order. Russell's classification of pfs in types and orders is purely syntactic. This is quite harmless as far as (simple) types are concerned: the number of arguments that a propositional function takes is a notion that can be reasonably described by syntactic methods.[4]

[4]The only criticism that one could have is that Russell's method excludes so-called "constant" functions, i.e. functions of which the outcome is independent of one or more

For orders, the syntactic classification is more questionable. Look for instance at the pfs $f \equiv \forall \mathtt{x}{:}0^0[\mathtt{R}(\mathtt{x})]$ and $g \equiv \forall \mathtt{x}{:}0^0[\mathtt{R}(\mathtt{x})] \vee \forall \mathtt{z}{:}()^9[\mathtt{z}() \vee \neg\mathtt{z}()]$. According to the *Principia*, g is a proposition of order 10, because g contains a variable of order 9. On the other hand, f, a proposition of order 1, is logically equivalent to g. So, the *interpretation* of g does not essentially involve the variable $\mathtt{z}$ of order 9. We could therefore argue that the order of g, for semantic reasons, should not be higher than 1, the order of f.

In the forthcoming sections we show that there are workable systems that do have an order-like hierarchy and nevertheless are not restrictive. The system that we present however, does not make a syntactic classification of propositional functions into orders, but a semantic one.

The system is based on Kripke's paper [78]. In 1936, Tarski [117] proved that introducing a truth-predicate T in a first-order language leads to contradictions. Such a predicate T is true for objects that are encodings of true propositions, and false for objects that are encodings of false propositions. For this reason, Tarski distinguishes between object-language and meta-language, and the truth predicate for propositions of the object language occurs only at the meta-level. For a truth predicate for propositions in meta-language one needs a meta-meta-language, etcetera.

Kripke however, allows a restricted truth-predicate in a first-order language. The restrictions on this predicate are such that no contradictions occur. The construction of Kripke's truth predicate has remarkable similarities with the use of orders in the Ramified Theory of Types, and we show that RTT can be embedded within Kripke's Theory of Truths KTT. It is even possible to define a notion of order for pfs of RTT, based on the construction of Kripke's truth predicate. An important difference is that this new notion of order is (partially) based on the interpretation of a pf, whilst Russell's definition of order is purely syntactic.

In Section 3c1 we describe KTT. In Section 3c2 we embed RTT in KTT and show that Russell's syntactic approach is much more restrictive than Kripke's semantic approach.

Parts of this section are based on [70].

of the arguments it takes. We saw this in our translation of pfs to λ-terms: all the translations were λI-terms (see Lemma 2.23.3).

3c1 Kripke's Theory of Truth KTT

In this section, we shortly describe Kripke's Theory of Truth: KTT (see [78]). Kripke expresses higher-order formulas within a first-order language, using the fact that many interesting languages are rich enough to express their own syntax via a Gödel Numbering.

In the rest of this Section 3c, L is a first-order language. $\mathcal{A}$ is the set of constant symbols in L, $\mathcal{F}$ is the set of function symbols in L, and $\mathcal{R}$ represents the set of predicate symbols of L. We assume that $\mathfrak{M} = \langle \boldsymbol{A}, [\![\ ]\!] \rangle$ is a model for L, where $[\![\ ]\!]$ is an interpretation function for the symbols of L.

Let us also assume two subsets S_1 and S_2 of $\boldsymbol{A}$ such that $S_1 \cap S_2 = \varnothing$. Kripke extends L by adding a monadic predicate T. The main idea is to interpret T as a unary "truth predicate" $\boldsymbol{T}$ such that S_1 contains the elements $\boldsymbol{a}$ of $\boldsymbol{A}$ that are (codes of) formulas which we consider to be "true", and S_2 contains those $\boldsymbol{a} \in \boldsymbol{A}$ that are (codes of) formulas which we consider to be "false". This extension of the model is denoted as $\mathfrak{M}(S_1, S_2)$. We do not demand that $S_1 \cup S_2 = \boldsymbol{A}$, hence $\boldsymbol{T}$ may be a *partially* defined predicate.

Definition 3.7 (Logical Truth for KTT) Let s be an assignment function $\mathcal{V} \to \boldsymbol{A}$. We define $\mathfrak{M} \models f[s]$ as follows[5]:

f	$\mathfrak{M} \models f[s]$	$\mathfrak{M} \models \neg f[s]$
$R(a_1, \ldots, a_m)$	$(a_1(s), \ldots, a_m(s)) \in [\![R]\!]$	$(a_1(s), \ldots, a_m(s)) \notin [\![R]\!]$
$g_1 \wedge g_2$	$\mathfrak{M} \models g_1[s]$ and $\mathfrak{M} \models g_2[s]$	$\mathfrak{M} \models ((\neg g_1) \vee (\neg g_2))[s]$
$g_1 \vee g_2$	$\mathfrak{M} \models g_1[s]$ or $\mathfrak{M} \models g_2[s]$	$\mathfrak{M} \models ((\neg g_1) \wedge (\neg g_2))[s]$
$\forall x[g]$	$\mathfrak{M} \models g[s[x{:=}\boldsymbol{a}]]$ for all $\boldsymbol{a} \in \boldsymbol{A}$	$\mathfrak{M} \models (\exists x[\neg g])[s]$
$\exists x[g]$	$\mathfrak{M} \models g[s[x{:=}\boldsymbol{a}]]$ for an $\boldsymbol{a} \in \boldsymbol{A}$	$\mathfrak{M} \models (\forall x[\neg g])[s]$
$\neg\neg g$	$\mathfrak{M} \models g[s]$	$\mathfrak{M} \models (\neg g)[s]$

Here, $R \in \mathcal{R}$ has arity m; $a, a_1, \ldots, a_m$ are terms of L, and g, g_1, g_2 are formulas of L. Moreover, $s[x{:=}\boldsymbol{a}]$ is the assignment function that assigns $\boldsymbol{a}$ to x, and $s(y)$ to any variable $y \in \mathcal{V} \setminus \{x\}$. Now let $S_1, S_2 \subseteq \boldsymbol{A}$ such that

[5]Notice that even though this definition is different from Tarski's definition, especially with respect to the definition of $\mathfrak{M} \models \neg f$, it is easy to prove the equivalence of both definitions. This is because all primitive predicates of L are totally defined. We took this definition however, as we need to extend it for the partial predicate T.

$S_1 \cap S_2 = \varnothing$. $L(\mathtt{T})$ is the extension of L with the monadic predicate $\mathtt{T}$. We extend the definition of $\mathfrak{M} \models f$ to $\mathfrak{M}(S_1, S_2) \models f$ by putting

$$\mathfrak{M}(S_1, S_2) \models (\mathtt{T}(a))[s] \text{ iff } a(s) \in S_1$$

and

$$\mathfrak{M}(S_1, S_2) \models (\neg\mathtt{T}(a))[s] \text{ iff } a(s) \in S_2.$$

It is important (and easy) to notice that the extension of L and $\mathfrak{M}$ to $L(\mathtt{T})$ and $\mathfrak{M}(S_1, S_2)$ is conservative:

Lemma 3.8 *Assume f is a sentence in L, and $S_1, S_2 \subseteq \boldsymbol{A}$ such that $S_1 \cap S_2 = \varnothing$. Then $\mathfrak{M} \models f$ if and only if $\mathfrak{M}(S_1, S_2) \models f$.* ⊠

The predicate $\mathtt{T}$ cannot express truth in a direct way. This is because $\mathtt{T}$ is a predicate in a first order language, and therefore can only take terms (objects) as arguments, and not propositions. However, there is an indirect way in which $\mathtt{T}$ can express truth: enumerate all formulas of $L(\mathtt{T})$.

From now on, we assume that we have an injective, primitive recursive function $\langle\ \rangle$ from the formulas (including non-closed formulas) of the first order language $L(\mathtt{T})$ to the terms of L. If f is a sentence then we can form the proposition $\mathtt{T}(\langle f \rangle)$, which expresses the truth of f. But we can also form the proposition $\mathtt{T}(\langle \mathtt{T}(\langle f \rangle) \rangle)$, so we can discuss the truth of the truth of f, etcetera. This makes it possible to express higher-order propositions.

As announced, we use the sets S_1 and S_2 to establish the truth of such higher-order propositions. Actually, we build a hierarchy of sets $S_{\alpha,1}$ and $S_{\alpha,2}$ for ordinals α. We will see that this hierarchy has much in common with Russell's hierarchy of orders.

Definition 3.9 For any ordinal α we define a pair of sets $(S_{\alpha,1}, S_{\alpha,2})$ and a model $\mathfrak{M}_\alpha$.

- $S_{0,1} \stackrel{\text{def}}{=} \varnothing$; $S_{0,2} \stackrel{\text{def}}{=} \varnothing$; $\mathfrak{M}_0 \stackrel{\text{def}}{=} \mathfrak{M}(S_{0,1}, S_{0,2})$;
- If $S_{\alpha,1}$, $S_{\alpha,2}$ and $\mathfrak{M}_\alpha$ have been defined, then we define:

$$\begin{array}{lcl} S_{\alpha+1,1} & \stackrel{\text{def}}{=} & \{[\![\langle f \rangle]\!] \mid f \text{ is a sentence and } \mathfrak{M}_\alpha \models f\} \\ S_{\alpha+1,2} & \stackrel{\text{def}}{=} & \{[\![\langle f \rangle]\!] \mid f \text{ is a sentence and } \mathfrak{M}_\alpha \models \neg f\} \\ \mathfrak{M}_{\alpha+1} & \stackrel{\text{def}}{=} & \mathfrak{M}(S_{\alpha+1,1}, S_{\alpha+1,2}); \end{array}$$

- If α is a limit ordinal and $S_{\beta,1}$, $S_{\beta,2}$ and $\mathfrak{M}_\beta$ have been defined for all $\beta < \alpha$, then

$$\begin{aligned} S_{\alpha,i} &\overset{\text{def}}{=} \bigcup_{\beta<\alpha} S_{\beta,i}; \\ \mathfrak{M}_\alpha &\overset{\text{def}}{=} \mathfrak{M}(S_{\alpha,1}, S_{\alpha,2}). \end{aligned}$$

The proof of the following lemma is not difficult. Yet, the lemma plays a crucial role in the rest of this Section.

Lemma 3.10 (Conservation of Knowledge (1))
If $\alpha < \beta$ then $S_{\alpha,1} \subseteq S_{\beta,1}$ and $S_{\alpha,2} \subseteq S_{\beta,2}$.

PROOF: Induction on β. The only non-trivial case is $\beta = \beta' + 1$. By the induction hypothesis, $S_{\alpha,i} \subseteq S_{\beta',i}$ for all $\alpha < \beta'$, so it suffices to prove that $S_{\beta',i} \subseteq S_{\beta,i}$.

We only give the proof for the case $S_{\beta,1}$; the proof for $S_{\beta,2}$ is similar.

So assume $\boldsymbol{a} \in S_{\beta',1}$. Determine $\gamma < \beta'$ and a sentence f such that $[\![\langle f \rangle]\!] = \boldsymbol{a}$ and $\mathfrak{M}_\gamma \models f$. Use induction on the definition of $\mathfrak{M}_\gamma \models f$. We treat only one case, the others are trivial: Assume $f \equiv \mathrm{T}(a')$ for some term a' of $L(\mathrm{T})$. Then $[\![a']\!] \in S_{\gamma,1}$ by definition of $\mathfrak{M}_\gamma \models f$. By the induction hypothesis on α we know: $S_{\gamma,1} \subseteq S_{\beta',1}$, so: $[\![a']\!] \in S_{\beta',1}$. By definition, this means $\mathfrak{M}_{\beta'} \models f$. Hence $[\![\langle f \rangle]\!] \in S_{\beta'+1,1}$, and this means $\boldsymbol{a} \in S_{\beta,1}$. ⊠

Corollary 3.11 (Conservation of Knowledge (2))
If $\alpha < \beta$ and $\mathfrak{M}_\alpha \models f$ then $\mathfrak{M}_\beta \models f$. ⊠

Remark 3.12 It is not the case that $\mathfrak{M}_\alpha \not\models f$ implies $\mathfrak{M}_\beta \not\models f$ for $\alpha < \beta$. For instance, let $f = \forall \mathrm{x}[\mathrm{R}(\mathrm{x}) \vee \neg\mathrm{R}(\mathrm{x})]$. Notice that $\mathfrak{M}_0 \models f$. Therefore, $\langle f \rangle \in S_{1,1}$, so $\mathfrak{M}_1 \models \mathrm{T}(\langle f \rangle)$. But $S_{0,1} = \varnothing$, so $\mathfrak{M}_0 \not\models \mathrm{T}(\langle f \rangle)$.

We prove that the theories with the new predicate T are all consistent:

Lemma 3.13 *Let α be an ordinal.*

1. *For all formulas f of $L(\mathrm{T})$ and for all assignments s,*

$$\mathfrak{M}_\alpha \not\models (f \wedge \neg f)[s];$$

2. $S_{\alpha,1} \cap S_{\alpha,2} = \varnothing$.

PROOF: We use induction on α. So assume the lemma has been proven for all $\beta < \alpha$ (IH1).

1. Use induction on the structure of f. So assume the lemma has been proven for all subformulas g of f (IH2). We treat three cases only (the other cases are similar).
   - $f = R(a_1, \ldots, a_m)$. If $\mathfrak{M}_\alpha \models (f \wedge \neg f)[s]$ then $(a_1(s), \ldots, a_m(s)) \in [\![R]\!]$ and $(a_1(s), \ldots, a_m(s)) \notin [\![R]\!]$, which is impossible;
   - $f = g_1 \wedge g_2$. If $\mathfrak{M}_\alpha \models (f \wedge \neg f)[s]$ then $\mathfrak{M}_\alpha \models (g_1 \wedge g_2)[s]$, so $\mathfrak{M}_\alpha \models g_j[s]$ for $j = 1, 2$. Also: $\mathfrak{M}_\alpha \models (\neg(g_1 \wedge g_s))[s]$, so $\mathfrak{M}_\alpha \models ((\neg g_1) \vee (\neg g_2))[s]$, so $\mathfrak{M}_\alpha \models (\neg g_j)[s]$ for $j = 1$ or $j = 2$. Therefore $\mathfrak{M}_\alpha \models (g_j \wedge \neg g_j)[s]$ for $j = 1$ or $j = 2$, which contradicts (IH2);
   - $f = \mathtt{T}(a)$ for a term a. If $\mathfrak{M}_\alpha \models (f \wedge \neg f)[s]$ then $a(s) \in S_{\alpha,1}$ and $a(s) \in S_{\alpha,2}$, which contradicts (IH1);
2. If $\boldsymbol{a} \in S_{\alpha_1} \cap S_{\alpha_2}$, then determine a formula g such that $[\![\langle g \rangle]\!] = \boldsymbol{a}$, and $\beta_1, \beta_2 < \alpha$ such that $\mathfrak{M}_{\beta_1} \models g$ and $\mathfrak{M}_{\beta_2} \models \neg g$. Let $\beta = \max(\beta_1, \beta_2)$. Then $\beta < \alpha$ and because of Conservation of Knowledge 3.11, $\mathfrak{M}_\beta \models g \wedge \neg g$. This contradicts (IH1).

⊠

We can see the construction of the models $\mathfrak{M}_\alpha$ as a process of obtaining knowledge. At the initial stage 0, $\mathtt{T}(\langle f \rangle)$ is not defined for any formula f. There is no knowledge at all.

By applying the definition of truth given for $\mathfrak{M}_0$, we obtain knowledge. Some sentences f can be judged true: $\mathfrak{M}_0 \models f$. We store the code of f in $S_{1,1}$. Some other sentences g can be judged false: $\mathfrak{M}_0 \models \neg g$. The code of g is stored in $S_{1,2}$. It is not possible to judge all sentences. For instance, neither $\mathfrak{M}_0 \models \forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})]$ nor $\mathfrak{M}_0 \models \neg \forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})]$ holds, so $[\![\langle \forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})] \rangle]\!]$ neither belongs to $S_{1,1}$, nor to $S_{1,2}$.

The knowledge we have obtained is expressed by the behaviour of the predicate $\mathtt{T}$ in $\mathfrak{M}_1$. At stage 1 we know more about $\mathtt{T}$ than at stage 0 $S_{0,1} = S_{0,2} = \varnothing$, but $S_{1,1} \neq \varnothing$ and $S_{1,2} \neq \varnothing$. Hence more sentences can be judged true or false. We store the codes of sentences that were judged "true" at level 1 in $S_{2,1}$ and codes of the sentences that were judged "false" at level 1 in $S_{2,2}$. The Lemma on Conservation of Knowledge 3.10 guarantees that this process only extends our knowledge, i.e.:

- Sentences that were judged to be true at level 0 remain true at level 1;
- Sentences that were judged to be false at level 0 remain false at level 1.

By iterating this process we arrive at the levels $2, 3, \ldots, \omega, \omega + 1, \ldots$. One might expect that for each sentence f there is an ordinal α for which $f \in S_{\alpha,1} \cup S_{\alpha_2}$. This is not the case. There are sentences of which the truth will never be established. See the forthcoming example 3.30.

3c2 RTT in KTT

Both in RTT and in KTT we are confronted with a hierarchy. Russell constructs a hierarchy by dividing propositions and propositional functions into different orders, taking care that a propositional function f can only depend on objects of a lower order than the order of f.

Kripke does not make this distinction beforehand. He has only one truth-predicate (T), but decisions about the truth of propositions are split into different levels: At the first level only decisions about propositions that do not involve any knowledge about T are made (for example: the proposition $\mathtt{R}(\mathtt{a}_1)$, but also the proposition $(\mathtt{R}(\mathtt{a}_1) \vee \neg\mathtt{R}(\mathtt{a}_1)) \vee \mathtt{T}(\mathtt{a}_2))$. At the second level decisions about propositions involving T for codes of first-level propositions are made, and so on.

Before we can compare RTT with KTT, we must give a formal definition of logical truth for pfs of RTT. After that, we investigate the similarity between both hierarchies in subsection 3c2.2 by describing RTT within KTT. In subsection 3c2.3 we investigate in which way RTT is more restrictive with respect to self-reference than KTT.

3c2.1 Logical truth for RTT in Tarski's style

As KTT uses Tarski's notion of logical truth, we use a similar notion for RTT. This definition of logical truth is quite informal. For example, the first clause "If $(a_1, \ldots, a_m) \in R$ then RTT $\models R(a_1, \ldots, a_m)$" requires the symbol R to be already fully interpreted and to denote a relation independently of any Tarskian assignment function. This is in line with Russell, who did not make distinction between the syntactic symbol R in "$\models R(a_1, \ldots, a_m)$" and the semantic use of R in "$(a_1, \ldots, a_m) \in R$". We take care that it will always be clear whether we use a symbol in its syntactic or in its semantic way.

We must remark that the definition of logical truth for RTT is not due to Russell and cannot be found in *Principia Mathematica.* In *Principia*, Russell and Whitehead use a notion of truth based on derivations in natural deduction style. But in order to make clear the similarities between RTT and KTT, we must use the notion of truth used in KTT in RTT as well.

Definition 3.14 (Logical Truth for RTT) Let Γ be an RTT-context with domain $\mathcal{V}$. Let $f \in \mathcal{P}$ be a legal pf in Γ with free variables $x_1, \ldots, x_n$ of types $t_1^{a_1}, \ldots, t_n^{a_n}$. Let $s : \mathcal{V} \to \mathcal{P}$ be such that $s(x_i) : t_i^{a_i} \in \Gamma$. We define RTT $\models_\Gamma f[s]$ by induction on the order of f. We give this definition by induction on the structure of f.

- RTT $\models_\Gamma R(i_1, \ldots, i_{\mathfrak{a}(R)})[s]$ if $(i_1(s), \ldots, i_{\mathfrak{a}(R)}(s)) \in R$;
- RTT $\models_\Gamma (g_1 \vee g_2)[s]$ if RTT $\models_\Gamma g_1[s]$ or RTT $\models_\Gamma g_2[s]$;
- RTT $\models_\Gamma (\neg g)[s]$ if RTT $\not\models_\Gamma g[s]$;
- $f = z(k_1, \ldots, k_n)$. The order of $f[z{:=}s(z)]$ is lower than the order of f. Therefore we can define: RTT $\models_\Gamma f[s]$ if RTT $\models_\Gamma (f[z{:=}s(z)])[s]$;
- $f = \forall x{:}t^a[g]$. The order of g is equal to the order of $\forall x{:}t^a[g]$, so we can assume that RTT $\models_{\Gamma'} g[s']$ has already been defined for contexts Γ' and valuations s'. Therefore we can define RTT $\models_\Gamma (\forall x{:}t^a[g])\,[s]$ if for all $h \in \mathcal{P} \cup \mathcal{A}$ for which $\Gamma \setminus \{x{:}t^a\} \vdash h : t^a$, RTT $\models_{\Gamma[x:t^a]} g[s[x{:=}h]]$.

Here $\Gamma[x{:}t^a]$ is the same context as Γ, except that we now assign the type t^a to the variable x. We write RTT $\models f$ instead of RTT $\models_\Gamma f$ if it is clear which context Γ is used.

3c2.2 RTT embedded in KTT

To embed RTT in a first order language L, we have to cope with two technical problems:

1. We need to encode the notion of and the manipulation with (higher-order) propositional functions into a first-order language. The manipulation is particularly important with respect to substitution, which in the higher-order situation is much more complicated than in the first order case (cf. the definition of substitution 2.24);

2. In Russell's theory, it is only possible to quantify over a part of all propositions. This makes it impossible to translate, for instance, the proposition $\forall \mathrm{p}{:}()^1[\mathrm{p}() \vee \neg \mathrm{p}()]$ directly to $\forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})]$, as the quantifier in the latter also quantifies over (codes of) higher-order propositions.

As we do not want RTT-contexts to be involved in this coding, we assume that each variable in RTT (implicitly) has a superscript t, indicating its ramified type. We only consider the legal propositional functions of RTT, and given a context Γ it is always possible to assign a unique type to each free variable in such a pf (cf. Lemma 2.56.1). Therefore we can do without contexts, as the types of the variables are now clear from the function in which they occur. For reasons of clarity, we will not explicitly write this superscript, as long as no confusion arises.

We propose the following solutions to the problems sketched above (we first give the definition and afterwards explain our thoughts behind it):

Definition 3.15 Extend the language $L(\mathrm{T})$ with for each ramified type t a monadic predicate Typ_t, and for each $n \in \mathbb{N}$ a $(n{+}1)$-ary function App_n. We code the typable propositional functions f of $\mathcal{P}$ as formulas $\overline{f}$ in this extension.[6] We do this by induction on the structure of f.

- If $f \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$, then f is present in the original language L and we take $\overline{f} \stackrel{\text{def}}{=} f$;
- If $f \equiv f_1 \vee f_2$, then $\overline{f} \stackrel{\text{def}}{=} \overline{f_1} \vee \overline{f_2}$;
- If $f \equiv \neg f'$, then $\overline{f} \stackrel{\text{def}}{=} \neg \overline{f'}$;
- If $f \equiv \forall x{:}u[f']$, then $\overline{f} \stackrel{\text{def}}{=} \forall x[\neg \mathsf{Typ}_u(x) \vee \overline{f'}]$;

[6]This mapping $\overline{}$ is different from the mapping $\overline{}$ that was used in Chapter 2.

- If $f \equiv z(k_1, \ldots, k_m)$, write $K_i \equiv \langle \overline{k_i} \rangle$ for $k_i \in \mathcal{P}$[7], and $K_i \equiv k_i$ for $k_i \in \mathcal{A} \cup \mathcal{V}$. Define $\overline{f} \stackrel{\text{def}}{=} \mathtt{T}(\mathtt{App}_m(z, K_1, \ldots, K_m))$.

Notation 3.16 To keep notations uniform, we sometimes want to speak about $\langle \overline{x} \rangle$ when we only intend to mention x, for $x \in \mathcal{V}$, and about $\langle \overline{a} \rangle$ when only meaning a, for $a \in \mathcal{A}$. Hence, we formally define: $\langle \overline{x} \rangle \stackrel{\text{def}}{=} x$ and $\langle \overline{a} \rangle \stackrel{\text{def}}{=} a$ for all $x \in \mathcal{V}$ and all $a \in \mathcal{A}$.

We now give a formal interpretation of the newly introduced predicate symbol $\mathtt{Typ}_t$ and function symbol $\mathtt{App}_n$. We take $\mathcal{A}$ as domain of our model, so: $\boldsymbol{A} = \mathcal{A}$. This corresponds with the fact that Russell did not make a distinction between syntax and semantics. The following definition is also based on this fact:

Definition 3.17 We define the function $[\![\ ]\!]$:

- $[\![a]\!] = a$ for all $a \in \mathcal{A}$;
- $[\![R]\!] = R$ for all $R \in \mathcal{R}$;
- $[\![\mathtt{Typ}_{0^0}]\!] \stackrel{\text{def}}{=} \mathcal{A}$, and for $t \neq 0^0$, $[\![\mathtt{Typ}_t]\!] = \{[\![\langle \overline{f} \rangle]\!] \mid f \in \mathcal{P}, f : t\}$;
- We do not give a full semantics of the function $\mathtt{App}_n$. This is because we need $\mathtt{App}_n$ and its semantics only in some special cases. Assume $n \in \mathbb{N}$, $f \in \mathcal{P}$ is of type $(t_1, \ldots, t_n)$ and has free variables $x_1 < \cdots < x_n$. Also assume $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{P}$, $k_i : t_i$ We define:

$$[\![\mathtt{App}_n]\!]([\![\langle \overline{f} \rangle]\!], [\![\langle \overline{k_1} \rangle]\!], \ldots, [\![\langle \overline{k_n} \rangle]\!]) \stackrel{\text{def}}{=} [\![\left\langle \overline{f[x_1, \ldots, x_n{:=}k_1, \ldots, k_n]} \right\rangle]\!].$$

Together, $\mathcal{A}$ and $[\![\ ]\!]$ form a model $\mathfrak{N}$ for the translation of RTT in KTT.

We make some remarks with respect to these definitions.

Remark 3.18 It is clear that the newly introduced functions $\mathtt{App}_n$ can be used for carrying out substitutions, thus solving the first of the technical problems stated at the beginning of this subsection. The predicates $\mathtt{Typ}_t$ (typability with type t) solve the second problem, as can be seen in the definition of $\overline{\forall x{:}u[f]}$.

[7]Observe: $\langle$ and $\rangle$ do not belong to the syntax of L extended with $\mathtt{T}$, $\mathtt{Typ}_t$ and $\mathtt{App}_n$. We define K_i to be the encoding of the translation of k_i, and not the list of symbols started with $\langle$, followed by the list of symbols that represent $\overline{k_i}$, and closed by $\rangle$.

Remark 3.19 At this point, our work is related to (but independent of) Paul Gilmore's work on NaDSet 1. NaDSet 1 is a theory of generalised abstraction which makes n-ary predication a primitive of the system, with the unary truth predicate being trivially definable upon this basis. For a useful connection between KTT and NaDSet 1, see [44].

Remark 3.20 The extensions of $L(\mathrm{T})$ with the relation symbol $\mathtt{Typ}_t$ and the function symbol $\mathtt{App}_n$ are of a mere technical character. Therefore, we think that we can still speak of an *embedding* of RTT within KTT.

Below, we work in two systems: RTT and KTT. These systems have a different notion of substitution, though they use the same notation for expressing substitution. From the context however, it will always be clear which kind of substitution is meant.

The language $L(\mathrm{T})$ extended with $\mathtt{Typ}_t$ and $\mathtt{App}_n$ is an example of the languages described in Section 3c1, and we can construct $\mathfrak{N}_\alpha$ for each ordinal α as described in that section.

Substitution in the language KTT is ordinary first-order substitution. Higher-order substitutions in KTT can be obtained via the application operators $\mathtt{App}_n$. For future results, it is essential to know that the combination of first-order substitution and the operators $\mathtt{App}_n$ in KTT is compatible with the higher-order substitution for RTT that was defined in Definition 2.24. This is shown in the following Lemma (we write $f[x_i{:=}g_i]_{i=1}^n$ as shorthand for $f[x_1,\ldots,x_n{:=}g_1,\ldots,g_n]$):

Lemma 3.21 (Substitution Lemma) *Let g be a legal propositional function such that* $\mathrm{FV}(g) = \{x_1,\ldots,x_m\}$, *and $k_1,\ldots,k_m \in \mathcal{A}\cup\mathcal{P}$ such that x_i and k_i have the same type for all i. Let p be at least the order of $g[x_i{:=}k_i]_{i=1}^{n-1}$, and q at least the order of $g[x_i{:=}k_i]_{i=1}^{n}$. Then*

$$\mathfrak{N}_p \models \overline{g[x_i{:=}k_i]_{i=1}^{n-1}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m}$$

if and only if

$$\mathfrak{N}_q \models \overline{g[x_i{:=}k_i]_{i=1}^{n}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}.$$

PROOF: We prove the following two statements

1.
$$\mathfrak{N}_p \models \overline{g[x_i{:=}k_i]_{i=1}^{n-1}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m}$$
if and only if
$$\mathfrak{N}_q \models \overline{g[x_i{:=}k_i]_{i=1}^{n}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$

2.
$$\mathfrak{N}_p \models \overline{\neg g[x_i{:=}k_i]_{i=1}^{n-1}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m}$$
if and only if
$$\mathfrak{N}_q \models \overline{\neg g[x_i{:=}k_i]_{i=1}^{n}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}.$$

It is necessary to prove these two statements instead of the single statement of the Lemma because of the special way in which we defined $\mathfrak{N}_\alpha \models \neg f$. We use induction on the order of g. So assume (induction hypothesis A) that the substitution lemma is proved for all g' with an order smaller than the order of g. Use induction on the structure of $g[x_i{:=}k_i]_{i=1}^{n-1}$ (induction hypothesis B).

- $g[x_i{:=}k_i]_{i=1}^{n-1} \equiv R(i_1, \ldots, i_{\mathfrak{a}(R)})$.
  1. Notice: $\overline{g[x_i{:=}k_i]_{i=1}^{n-1}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m} \equiv \overline{g[x_i{:=}k_i]_{i=1}^{n}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}$. As the truth of $R(a_1, \ldots, a_{\mathfrak{a}(R)})$ can always be established at level 0 (so in $\mathfrak{N}_0$), there is nothing to be proved;
  2. Similar;
- $g[x_i{:=}k_i]_{i=1}^{n-1} \equiv g_1 \vee g_2$.
  1. The following statements are equivalent:

$$\begin{aligned}
&\mathfrak{N}_p \models \overline{g_1 \vee g_2}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m};\\
&\mathfrak{N}_p \models (\overline{g_1} \vee \overline{g_2})\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m};\\
&\mathfrak{N}_p \models \overline{g_1}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m} \vee \overline{g_2}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m};\\
&\mathfrak{N}_p \models \overline{g_j}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m} \text{ for some } j \in \{1,2\}. \qquad (2)
\end{aligned}$$

As the order of g_j is at most the order of $g[x_i{:=}k_i]_{i=1}^{n-1}$, and the order of $g_j[x_n{:=}k_n]$ is at most the order of $g[x_i{:=}k_i]_{i=1}^{n}$, we can apply induction hypothesis B. Therefore, (2) is equivalent to the following statements:

$$\begin{aligned}
\mathfrak{R}_q &\models \overline{g_j[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m} \text{ for some } j \in \{1,2\};\\
\mathfrak{R}_q &\models \overline{g_1[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m} \vee\\
&\ \overline{g_2[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};\\
\mathfrak{R}_q &\models \Big(\overline{g_1[x_n{:=}k_n]} \vee \overline{g_2[x_n{:=}k_n]}\Big)\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};\\
\mathfrak{R}_q &\models \overline{(g_1 \vee g_2)[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};
\end{aligned}$$

2. Similar;

- $g[x_i{:=}k_i]_{i=1}^{n-1} \equiv \neg g'$.
  1. This is induction hypothesis B(2) on g';
  2. The following statements are equivalent:

$$\begin{aligned}
\mathfrak{R}_p &\models \overline{\neg\neg g'}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m};\\
\mathfrak{R}_p &\models \neg\neg\overline{g'}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m};\\
\mathfrak{R}_p &\models \overline{g'}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m}. \qquad (3)
\end{aligned}$$

By induction hypothesis B(1) on g', (3) is equivalent to the following statements:

$$\begin{aligned}
\mathfrak{R}_q &\models \overline{g'[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};\\
\mathfrak{R}_q &\models \neg\neg\overline{g'[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};\\
\mathfrak{R}_q &\models \overline{\neg\neg g'[x_n{:=}k_n]}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n+1}^{m};
\end{aligned}$$

- $g[x_i{:=}k_i]_{i=1}^{n-1} \equiv \forall x{:}t[g']$.
  1. The following statements are equivalent:

$$\begin{aligned}
\mathfrak{R}_p &\models \overline{\forall x{:}t[g']}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m};\\
\mathfrak{R}_p &\models \forall x[\neg\mathrm{Typ}_t(x) \vee \overline{g'}]\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m};\\
\mathfrak{R}_p &\models \forall x\left[\neg\mathrm{Typ}_t(x) \vee \overline{g'}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m}\right];\\
\mathfrak{R}_p &\models \overline{g'}\,\big[x_i{:=}\langle\overline{k_i}\rangle\big]_{i=n}^{m}\ \big[x{:=}\langle\overline{h}\rangle\big] \text{ for all } h : t. \qquad (4)
\end{aligned}$$

By induction hypothesis B(1) on g', (4) is equivalent to the following statements:

$$\mathfrak{R}_q \models \overline{g'[x_n{:=}k_n]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}\left[x{:=}\langle\overline{h}\rangle\right] \text{ for all } h : t;$$
$$\mathfrak{R}_q \models \forall x\left[\neg\mathtt{Typ}_t(x) \vee \overline{g'[x_n{:=}k_n]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}\right];$$
$$\mathfrak{R}_q \models \forall x\left[\neg\mathtt{Typ}_t(x) \vee \overline{g'[x_n{:=}k_n]}\right]\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$
$$\mathfrak{R}_q \models \overline{\forall x{:}t[g'[x_n{:=}k_n]]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$
$$\mathfrak{R}_q \models \overline{\forall x{:}t[g'][x_n{:=}k_n]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$

2. Similar;

- $g[x_i{:=}k_i]_{i=1}^{n-1} \equiv z(h_1, \ldots, h_r)$.
  1. If $x_n \not\equiv z$, then
  $$\overline{g}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m} \equiv \overline{g[x_n{:=}k_n]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m},$$
  and there is nothing to be proved. So assume $z \equiv x_n$. Let $y_1 < \cdots < y_r$ be the free variables of k_n. The following statements are equivalent:
  $$\mathfrak{R}_p \models \overline{x_n(h_1, \ldots, h_r)}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m};$$
  $$\mathfrak{R}_p \models \mathtt{T}\left(\mathtt{App}_r\left(x_n, \langle\overline{h_1}\rangle, \ldots, \langle\overline{h_r}\rangle\right)\right)\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n}^{m};$$
  $$\mathfrak{R}_p \models \mathtt{T}\left(\mathtt{App}_r\left(\langle\overline{k_n}\rangle, \langle\overline{h_1}\rangle, \ldots, \langle\overline{h_r}\rangle\right)\right)\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m}. \quad (5)$$
  Let $h'_j \equiv k_s$ if $h_j \equiv x_s$ and $h'_j \equiv h_j$ if $h_j \notin \{x_{n+1}, \ldots, x_m\}$. Then (5) is equivalent to the following statements:
  $$\mathfrak{R}_p \models \mathtt{T}\left(\mathtt{App}_r\left(\langle\overline{k_n}\rangle, \left\langle\overline{h'_1}\right\rangle, \ldots, \langle\overline{h'_r}\rangle\right)\right);$$
  $$\mathfrak{R}_{p-1} \models \overline{k_n[y_j{:=}h'_j]_{j=1}^{r}};$$
  $$\mathfrak{R}_{p-1} \models \overline{k_n[y_j{:=}h_j]_{j=1}^{r}[x_i{:=}k_i]_{i=n+1}^{m}}. \quad (6)$$
  We can use induction hypothesis A: the order of k_n is smaller than the order of $z(h_1, \ldots, h_r)$ and therefore smaller than the order of g (see Corollary 2.63).). Thus (6) is equivalent to the following statements:
  $$\mathfrak{R}_q \models \overline{k_n[y_j{:=}h_j]_{j=1}^{r}}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$
  $$\mathfrak{R}_q \models \overline{z(h_1, \ldots, h_r)[x_n{:=}k_n]}\left[x_i{:=}\langle\overline{k_i}\rangle\right]_{i=n+1}^{m};$$

2. Similar.

⊠

Corollary 3.22 *Assume g is a pf of order p and $g[x{:=}k]$ is a proposition of order q. If $\mathfrak{N}_q \models \overline{g[x{:=}k]}$ then $\mathfrak{N}_p \models \overline{g}\,[x{:=}\langle\overline{k}\rangle]$.* ⊠

Remark 3.23 We have actually proved a stronger fact: *Assume g is a propositional function of order m and $g[x{:=}k]$ is a proposition of order n. If $\mathfrak{N}_n \models \overline{g[x{:=}k]}$ then $\mathfrak{N}_p \models \overline{g}[x{:=}\langle\overline{k}\rangle]$, where $p = min(m, n+1)$.* This tells us more about the role of the predicate T: Although a substitution may lower the order of a propositional function by more than one, only one application of the T-predicate is involved (hence only one level in the hierarchy of truths). However, in the theorem below we only need the (weaker) form in which we presented the Substitution Lemma originally.

We now prove the main theorem of this section.

Theorem 3.24 (Embedding of RTT in KTT) *Let Γ be an RTT-context with domain $\mathcal{V}$. Let $f \in \mathcal{P}$ be a legal pf in Γ with free variables $x_1, \ldots, x_n$ of types $t_1^{a_1}, \ldots, t_n^{a_n}$. Let $s : \mathcal{V} \to \mathcal{P}$ be an assignment function such that $s(x_i) : t_i^{a_i} \in \Gamma$. Let n be the order of $f[s] \equiv f[x_i{:=}s(x_i)]_{i=1}^m$.*
Then RTT $\models f[s]$ if and only if $\mathfrak{N}_n \models \overline{f[s]}$.

PROOF:

⇒ As in the proof of Lemma 3.21, we have to deal with the special way in which we defined KTT$_n \models \neg f$. Therefore, we simultaneously prove:

1. If RTT $\models f[s]$ then $\mathfrak{N}_n \models \overline{f[s]}$;
2. If RTT $\models \neg f[s]$ then $\mathfrak{N}_n \models \overline{\neg f[s]}$.

The proof follows the same induction structure as the definition of RTT $\models f[s]$ (3.14).

- $f \equiv R(a_1, \ldots, a_{\mathfrak{a}(R)})$ for some $R \in \mathcal{R}$ and some $a_1, \ldots, a_{\mathfrak{a}(R)} \in \mathcal{A} \cup \mathcal{V}$. Notice that $\overline{f} \equiv f$ and that $\overline{f[s]} \equiv f[s]$. Write

$$f[s] \equiv R(a'_1, \ldots, a'_{\mathfrak{a}(R)})$$

for certain $a'_1, \ldots, a'_{\mathfrak{a}(R)} \in \mathcal{A}$. As RTT $\models R(a'_1, \ldots, a'_{\mathfrak{a}(R)})$, we know that $(a'_1, \ldots, a'_{\mathfrak{a}(R)}) \in R$, hence $\mathfrak{N}_n \models R(a'_1, \ldots, a'_{\mathfrak{a}(R)})$. The proof is similar for $\neg f$;

- $f \equiv g_1 \vee g_2$.
  Then g_1 and g_2 are legal. Assume n_j is the order of $g_j[s]$. Notice that $n_j \leq n$.
  First assume RTT $\models f[s]$. As
  $$\begin{array}{rcl} f[s] & \equiv & (g_1 \vee g_2)[s] \\ & \overset{(2.30.1)}{\equiv} & g_1[s] \vee g_2[s], \end{array}$$
  we have RTT $\models g_j[s]$ for $j = 1$ or $j = 2$. By the induction hypothesis on the structure of f: $\mathfrak{N}_{n_j} \models \overline{g_j[s]}$, and as $n_j \leq n$: $\mathfrak{N}_n \models \overline{g_j[s]}$. Therefore, $\mathfrak{N}_n \models \overline{g_1[s]} \vee \overline{g_2[s]}$. Now observe that
  $$\begin{array}{rcl} \overline{f[s]} & \equiv & \overline{(g_1 \vee g_2)[s]} \\ & \overset{(2.30.1)}{\equiv} & \overline{g_1[s]} \vee \overline{g_2[s]}. \end{array}$$
  Now assume RTT $\models \neg f[s]$. By a similar argument, we find that RTT $\models \neg g_j[s]$ for $j = 1$ and $j = 2$. By the induction hypothesis on the structure of f, $\mathfrak{N}_{n_j} \models \overline{\neg g_j[s]}$ for $j = 1, 2$, so $\mathfrak{N}_n \models \overline{\neg g_1[s]} \wedge \overline{\neg g_2[s]}$, or in other words:
  $$\mathfrak{N}_n \models \neg\left(\overline{g_1[s]} \vee \overline{g_2[s]}\right).$$
  Observe that $\neg\left(\overline{g_1[s]} \vee \overline{g_2[s]}\right) \equiv \overline{\neg f[s]}$;
- $f \equiv \neg g$.
  If RTT $\models f[s]$ then use the induction hypothesis on the structure of g to get $\mathfrak{N}_n \models \overline{\neg g[s]}$, hence $\mathfrak{N}_n \models \overline{f[s]}$.
  If RTT $\models \neg f[s]$, then RTT $\models g[s]$, so by induction on the structure of g, $\mathfrak{N}_n \models \overline{g[s]}$, so $\mathfrak{N}_n \models \neg\neg\overline{g[s]}$, so $\mathfrak{N}_n \models \overline{\neg f[s]}$;
- $f \equiv \forall x{:}t[g]$.
  If RTT $\models f[s]$ then for all $k{:}t$, RTT $\models g[s[x{:=}k]]$, where $s[x{:=}k]$ is the assignment function that assigns k to x, and $s(y)$ to all $y \in \mathcal{V} \setminus \{x\}$. By the induction hypothesis on the structure of f, we know that for all $k : t$, $\mathfrak{N}_{n_k} \models \overline{g[s[x{:=}k]]}$, where n_k is the order of $g[s[x{:=}k]] \equiv g[x_i{:=}s(x_i)][x{:=}k]$. By Corollary 3.22 we have: For all $k : t$,
  $$\mathfrak{N}_n \models \overline{g[x_i{:=}s(x_i)]_{i=1}^{m}[x{:=}\langle\overline{k}\rangle]}.$$

Hence, for all $a \in \mathcal{A}$,

$$\mathfrak{N}_n \models \left(\neg\mathrm{Typ}_t(x) \vee \overline{g[x_i{:=}s(x_i)]_{i=1}^m}\right)[x{:=}a].$$

So $\mathfrak{N}_n \models \forall x \left[\neg\mathrm{Typ}_t(x) \vee \overline{g[x_i{:=}s(x_i)]_{i=1}^m}\right]$. Observe:

$$\begin{aligned}\overline{f[s]} &\equiv \overline{(\forall x{:}t[g])\,[s]}\\ &\equiv \overline{\forall x{:}t\,[g[x_i{:=}s(x_i)]_{i=1}^m]}\\ &\equiv \forall x\left[\neg\mathrm{Typ}_t(x) \vee \overline{g[x_i{:=}s(x_i)]_{i=1}^m}\right].\end{aligned}$$

The argument for RTT $\models \neg f$ is similar;

- $f \equiv z(h_1,\ldots,h_p)$. Determine q such that $z \equiv x_q$. Write $h'_j \equiv h_j$ if $h_j \notin \{x_1,\ldots,x_m\}$; $h'_j \equiv s(x_\ell)$ if $h_j \equiv x_\ell$. Assume that $s(x_q)$ has free variables $y_1 < \cdots < y_r$. Observe:

$$f[x_i{:=}s(x_i)]_{i=1}^m \equiv s(x_q)[y_j{:=}h'_j]_{j=1}^r.$$

So if RTT $\models f[x_i{:=}s(x_i)]_{i=1}^m$ then RTT $\models s(x_q)[y_j{:=}h'_j]_{j=1}^r$. As the order of $s(x_q)$ is smaller than the order of f, we can use induction on the order of f to obtain (with Conservation of Knowledge): $\mathfrak{N}_n \models \overline{s(x_q)[y_j{:=}h'_j]_{j=1}^r}$, which is equivalent to: $\mathfrak{N}_n \models \overline{f[x_i{:=}s(x_i)]_{i=1}^m}$. The proof for RTT $\models \neg f[x_i{:=}s(x_i)]_{i=1}^m$ is similar;

$\Leftarrow$ This is easily shown now by contraposition. Assume, for the sake of the argument, that RTT $\models f$ does not hold. Then RTT $\models \neg f$, so by the $\Rightarrow$ part of the theorem (that was proved above), $\mathfrak{N}_n \models \neg\overline{f}$. So, if $\mathfrak{N}_n \models \overline{f}$ then RTT $\models f$.

⊠

This theorem clearly shows the relation between the orders in RTT and the levels of truth in KTT. The heart of the proof of Theorem 3.24 is in the proof of case $x_n(h_1,\ldots,h_r)$ of the Substitution Lemma 3.21 (via its corollary 3.22). This is the only place in the proof where the properties of the predicate T are used. It is understandable that these properties must be used at exactly this place when we look at the definition of propositional functions and the typing rules for propositional functions.

Exactly the possibility of constructing a propositional function of the form $x_n(h_1, \ldots, h_r)$ makes it possible to arrive at higher-order propositional functions and higher-order propositions. So exactly at this spot, Kripke's predicate T must appear, in order to raise one level in KTT as well.

One might expect to need the properties of T in the proof of the case $f \equiv z(h_1, \ldots, h_p)$ of Theorem 3.24 as well. But we see that this is not the case. This is understandable: we do not consider the truth of $z(h_1, \ldots, h_p)$ itself, but of $z(h_1, \ldots, h_p)[x_i{:=}s(x_i)]_{i=1}^m$. And we do not work with the order of $z(h_1, \ldots, h_p)$, but with n, the order of $z(h_1, \ldots, h_p)[x_i{:=}s(x_i)]_{i=1}^m$. The shift to a lower order has to do with the orders of $z(h_1, \ldots, h_p)$ and $s(x_q)$, but not with the orders of $z(h_1, \ldots, h_p)[x_i{:=}s(x_i)]_{i=1}^m$ and $s(x_q)[y_j{:=}h'_j]_{j=1}^r$. These last two propositions are syntactically equivalent and therefore of the same order.

Corollary 3.25 *If $f \in \mathcal{P}$ is a legal proposition of order n, then* RTT $\models f$ *if and only if* $\mathfrak{N}_n \models \overline{f}$. ⊠

Corollary 3.26 (Conservativity of $\mathfrak{N}_\omega$ over RTT) RTT $\models f$ *if and only if* $\mathfrak{N}_\omega \models \overline{f}$. ⊠

We cannot improve the result of Theorem 3.24 in general: For all n, there are propositions f of order n in RTT whose code is provable at level $\mathfrak{N}_n$ in KTT, but not at any lower level.

Theorem 3.27 *Let $n > 0$, and let f_n be the nth-order-proposition*

$$\forall \mathtt{p}{:}()^{n-1}[\mathtt{p}() \vee \neg\mathtt{p}()].$$

Then:

$$\mathfrak{N}_m \models \overline{f_n} \textit{ if and only if } m \geq n.$$

PROOF:

$\Leftarrow$ follows from Theorem 3.24 and Lemma 3.10;

$\Rightarrow$ is by induction on n. Observe that

$$\overline{f_n} \equiv \forall \mathtt{p}[\neg\mathtt{Typ}_{()^{n-1}}(\mathtt{p}) \vee (\mathtt{T}(\mathtt{App}_0(\mathtt{p})) \vee \neg\mathtt{T}(\mathtt{App}_0(\mathtt{p})))].$$

- $n = 1$. Let g be any proposition of order 0 in RTT. Then $\mathfrak{N}_0 \models \mathtt{Typ}_{()^0}(\langle \overline{g} \rangle)$ but as T is completely undefined at level 0 ($S_{0,1} = S_{0,2} = \varnothing$),
$$\mathfrak{N}_0 \not\models \mathtt{T}(\mathtt{App_0}(\langle \overline{g} \rangle)) \vee \neg \mathtt{T}(\mathtt{App_0}(\langle \overline{g} \rangle)).$$
Hence, $\mathfrak{N}_0 \not\models \overline{f_1}$;
- Assume the theorem has been proved for $n-1$. Assume $m < n$ and $\mathfrak{N}_m \models \overline{f_n}$. By definition of $\models$, we have:
$$\mathfrak{N}_{n-1} \models \mathtt{T}(\mathtt{App_0}(\langle \overline{f_{n-1}} \rangle)) \vee \neg \mathtt{T}(\mathtt{App_0}(\langle \overline{f_{n-1}} \rangle)),$$
and for reasons of consistency: $\mathfrak{N}_m \models \mathtt{T}(\mathtt{App_0}(\langle \overline{f_{n-1}} \rangle))$. Therefore, $\mathfrak{N}_m \models \mathtt{T}(\overline{\langle f_{n-1} \rangle})$, so, by the definition of T: $\mathfrak{N}_{m-1} \models \overline{f_{n-1}}$, which contradicts the induction hypothesis, as $m - 1 < n - 1$.

⊠

There are however, for any n, propositions f of order n in RTT for which $\mathfrak{N}_m \models \overline{f}$ or $\mathfrak{N}_m \models \overline{\neg f}$ can already be established for $m < n$.

Example 3.28 Consider a proposition $g \equiv g_1 \vee g_2$ where g_1 is a true proposition of order m and g_2 is any proposition of order $n > m$. As g_1 is true in RTT, we have $\mathfrak{N}_m \models \overline{g_1}$, and therefore $\mathfrak{N}_m \models \overline{g}$.

3c2.3 The restrictiveness of Russell's theory

We illustrate the different approaches of Russell and Kripke by an example given by Kripke himself in [78].

Example 3.29 Two politicians, Wim and Frits[8], are quarrelling about who is telling the truth and who is lying. Of course, Wim states that anything said by Frits is untrue (A), and Frits argues that any statement of Wim is false (B). The utterances (A) and (B) can be complete nonsense, but they can also be meaningful. This does not only depend on the (syntactic) structure of (A) and (B), but also on their semantics, that is: on the utterances of Wim and Frits (which may be more than only (A) and (B)):

[8]Any correspondence with existing Dutch politicians is purely coincidental.

1. Assume, (A) is the only statement that Wim makes, and (B) is the only statement that was made by Frits. Then (A) and (B) are nonsense. More precise, there is no reason to believe that (A) is true, and there is no reason either to believe that (B) is true. Namely: if we want to prove that (A) is true, we must show first that all statements of Frits are false, in other words: that (B) is false. But in order to establish the falsehood of (B) we must first find a true statement of Wim, that is: we must prove (A). Summarising: The truth of (A) can only be established if the truth of (A) has already been established before. So the truth of (A) will never be established.

   Similarly, we show that the falsehood of (A) will never be established, and that neither truth nor falsehood of (B) will ever be established;

2. Now assume that (B) is still the only statement made by Frits, but that Wim has not only uttered (A), but also argues that that one equals one (C). Statement (C) is clearly true. This means that Frits has been lying. Therefore, Frits' only statement (B) is false, hence Wim's statement (A) is true.

We formalise this situation as follows. We assume that the first order language L contains at least three relation symbols, $\mathtt{W}$, $\mathtt{F}$ and $=$. Moreover, we assume to have an individual symbols $\mathtt{1}$. $\mathtt{W}$ will be interpreted as the set of (codes of the) utterances of Wim, $\mathtt{F}$ shall represent the set of (codes of the) utterances of Frits. In this way, we can encode the expression (A) by

$$A \equiv \forall \mathtt{x}[\neg \mathtt{F}(\mathtt{x}) \vee \neg \mathtt{T}(\mathtt{x})].$$

Here, $\mathtt{T}$ is the truth predicate as introduced earlier in this Section. Similarly, (B) is encoded by

$$B \equiv \forall \mathtt{x}[\neg \mathtt{W}(\mathtt{x}) \vee \neg \mathtt{T}(\mathtt{x})]$$

and (C) is encoded by

$$C \equiv \mathtt{1} = \mathtt{1}.$$

We model the situations 1 and 2 above as follows:

1. In the first situation we take $\mathbb{N}$ as our domain. The semantics of $\mathtt{W}$ must represent the set of utterances of Wim, so we let $\boldsymbol{W} = [\![\mathtt{W}]\!] = \{[\![\langle A\rangle]\!]\}$. Similarly, $\boldsymbol{F} = [\![\mathtt{F}]\!] = \{[\![\langle B\rangle]\!]\}$. Further we let $[\![\mathtt{1}]\!] = 1$. This

gives us a model $\mathfrak{M}^1$ for L. We can build a hierarchy $\mathfrak{M}^1_\alpha$ (for ordinals α) as explained in Section 3c1. We show that there is no ordinal α for which $\mathfrak{M}^1_\alpha \models A$.

Assume, α is the smallest ordinal for which $\mathfrak{M}^1_\alpha \models A$. Then $\mathfrak{M}^1_\alpha \models (\neg\mathtt{F}(\mathtt{x}) \vee \neg\mathtt{T}(\mathtt{x}))[s[\mathtt{x}{:=}n]]$ for all $n \in \mathbb{N}$ and all assignment functions $s : \mathcal{V} \to \mathbb{N}$. This means that for all $n \in \mathbb{N}$, either $n \notin \boldsymbol{F}$ or $n \in S_{\alpha,2}$. Notice that $[\![\langle B\rangle]\!] \in \boldsymbol{F}$. Therefore: $[\![\langle B\rangle]\!] \in S_{\alpha,2}$. But then there is $\beta < \alpha$ such that $\mathfrak{M}^1_\beta \models \neg B$. Using the definition of $\mathfrak{M}^1_\beta \models \neg B$, this means that there is $n \in \mathbb{N}$ such that $n \in \boldsymbol{W}$ and $n \in S_{\beta,1}$. The only candidate for such n is $[\![\langle A\rangle]\!]$, as this is the only element of $\boldsymbol{W}$. Hence: $[\![\langle A\rangle]\!] \in S_{\beta,1}$. Therefore, there is $\gamma < \beta$ for which $\mathfrak{M}^1_\gamma \models A$. This is a contradiction because $\gamma < \alpha$, and α is the smallest ordinal for which $\mathfrak{M}^1_\alpha \models A$.

In a similar way we can show that for all α, $\mathfrak{M}^1_\alpha \not\models \neg A$, $\mathfrak{M}^1_\alpha \not\models B$, and $\mathfrak{M}^1_\alpha \not\models \neg B$. This corresponds to our earlier conclusions that the sentences (A) and (B) are nonsense if they are the only utterances of Wim and Frits;

2. For the second situation, we change the model $\mathfrak{M}^1$ to a model $\mathfrak{M}^2$ by replacing $\boldsymbol{W}$ by $\{[\![\langle A\rangle]\!], [\![\langle C\rangle]\!]\}$. Notice that $\mathfrak{M}^2_0 \models C$, because $1 = 1$. Therefore, $[\![\langle C\rangle]\!] \in S_{1,1}$, so $\mathfrak{M}^2_1 \models \mathtt{T}(\langle C\rangle)$. As $[\![\langle C\rangle]\!] \in \boldsymbol{W}$, we also have $\mathfrak{M}^2_1 \models \mathtt{W}(\langle\mathtt{C}\rangle)$. Therefore, $\mathfrak{M}^2_1 \models \neg(\neg\mathtt{W}(\mathtt{x}) \vee \neg\mathtt{T}(\mathtt{x}))[s[\mathtt{x}{:=}[\![\langle C\rangle]\!]]$, for any assignment function $s : \mathcal{V} \to \mathbb{N}$. Hence $\mathfrak{M}^2_1 \models \neg B$.

   This shows that Frits' statement is indeed false, but it also shows that Wim's statement (A) is true at level 2: As $\mathfrak{M}^2_1 \models \neg B$, $[\![\langle B\rangle]\!] \in S_{2,2}$, and this implies that $\mathfrak{M}^2_2 \models A$.

In Russell's terminology it would not be possible to type expressions like A and B at all. The expression A involves B, and therefore has to be of higher order than B. Similarly, B involves A, so it has to be of a higher order than A.

This indicates an important difference between RTT and KTT: Kripke allows much more expressions to be included in the system. In some situations these expressions will never obtain any truth-value (like A and B in the first example), but in other situations (so with other definitions of the primitive predicates) the same expressions will get a truth-value. Kripke concludes:

"it would be fruitless to look for an *intrinsic*[9] criterion that will enable us to sieve out – as meaningless, or ill-formed – those sentences which lead to paradox"

([78], p. 692)

Example 3.30 Another, more formal, example of a proposition f in KTT for which there is no $g \in \mathcal{P}$ with $\overline{g} \equiv f$ is the proposition

$$f \stackrel{\text{def}}{=} \forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})].$$

Notice that this is an impredicative proposition. It expresses that all propositions are either true or false, including f itself.

Assume, for the sake of the argument, that $\overline{g} \equiv f$. Let m be the order of g. Determine whether RTT $\models g$ or RTT $\models \neg g$. We give the argument for the case RTT $\models g$; the argument for $\neg g$ is easy. If RTT $\models g$ then by Corollary 3.25, $\mathfrak{N}_m \models \forall \mathrm{x}[\mathrm{T}(\mathrm{x}) \vee \neg \mathrm{T}(\mathrm{x})]$. This implies $\mathfrak{N}_m \models \mathrm{T}(f_m) \vee \neg \mathrm{T}(f_m)$, where f_m is as in Theorem 3.27. By definition of T this means $\mathfrak{N}_{m-1} \models f_m$ or $\mathfrak{N}_{m-1} \models \neg f_m$, both contradicting Theorem 3.27.

3c3 Orders and types

RTT is based on a double hierarchy: One of types and one of orders. This double hierarchy is too restrictive. It is possible to develop Logic and Mathematics within RTT, but we saw that the proof of the theorem of the least upper bound, which is fundamental in real analysis, cannot be given. The origin of the problem is the use of the so-called predicative and impredicative propositional functions.

It is therefore interesting to notice the relation between orders in RTT and levels of truth in KTT, as formulated in Theorem 3.24. It shows that Kripke's system can be regarded as a system based on RTT of which not the *orders*, but the *types* have been removed. In this way, KTT can be seen as a system that is dual to the simple theory of types.

KTT however, has a more subtle approach than many type theories as it does not exclude any, possibly "paradoxical", expression from the syntax,

[9]Italics of Kripke

which is the usual type-theoretic approach. If an expression is paradoxical, it will not get a truth value at any level α of the hierarchy of Truths. Whether an expression is paradoxical or not does not only depend on its syntactic structure, but also on the domain $\boldsymbol{A}$ and the relations of $\boldsymbol{R}$ on $\boldsymbol{A}$ (see Example 3.29). So paradoxes are only excluded at the level of semantics.

The discussion above shows that the orders of RTT are not to be blamed for the restrictiveness of RTT. KTT is a system which contains orders but has only few restrictions towards self-application. It is the *combination* of orders and types that makes RTT restrictive.

The special structure of KTT makes it possible to define a notion of *semantic* order in RTT:

Definition 3.31 Let $f \in \mathcal{P}$ have type t^a. The *semantic order* of f is the smallest natural number n for which either $\mathfrak{N}_n \models \overline{f}$ or $\mathfrak{N}_n \models \neg\overline{f}$.

By Theorem 3.24, the semantic order of f is always smaller than or equal to its (syntactic) order.

Conclusions

We saw in Section 3a1 that the Ramified Theory of Types is very restrictive for the description of mathematics within logic, because it is not possible to formulate impredicative definitions in RTT.

This was already realised by Russell and Whitehead, who tried to solve this by postulating the Axiom of Reducibility 3.3. This axiom has been criticised from the moment it was written down, both by Russell and Whitehead themselves and by others. Ramsey, Hilbert and Ackermann deramify RTT: They remove the orders. They observe that this does not lead to known paradoxes as long as a proper distinction between language and metalanguage is made.

Gödel and Quine observe that the deramification does not violate the Vicious Circle Principle, as long as one accepts that objects and pfs exist independently of our constructions.

So historically speaking, one could say that the orders were blamed for the restrictiveness of RTT. In Section 3c we showed that this is not correct. We used the formalisation of RTT that was given in Chapter 2 to compare RTT with Kripke's Theory of Truth KTT. We established the relation between Russell's hierarchy of orders and Kripke's hierarchy of truth-levels. In particular we showed that:

1. A proposition of RTT of order n is true if and only if its interpretation is true at level n in Kripke's Truth Hierarchy (Theorem 3.24);

2. The truth of some propositions of order n of RTT cannot be established in KTT at a level of truth hierarchy smaller than n (Theorem 3.27). Yet for some other propositions, it can be established at an earlier level (Example 3.28).

We also saw that Russell's theory is more restrictive than Kripke's. On the one hand, all propositional functions of RTT can be coded in Kripke's Truth Theory; on the other hand there are formulas of Kripke's theory that cannot be expressed in RTT, respecting both hierarchies.

We feel that the orders are not to be blamed (alone) for the restrictiveness of RTT. KTT clearly has a structure with orders (see Definition 3.31); nevertheless it is possible to give impredicative definitions (see Example 3.30).

Russell excludes all propositional functions that might lead to paradoxical situations beforehand. Kripke does not exclude them, though it is not guaranteed that each proposition gets a truth value. This may depend on the model chosen (see Example 3.29).

Whether the orders should be blamed or not, the main line in the history continues with non-ramified theories. For example, Church's combination of λ-calculus with simple type theory, the basis for most modern type systems, has no orders.

Chapter 4

Propositions as Types and Proofs as Terms

In this chapter we discuss the notions of Propositions as Types and Proofs as Terms (both abbreviated as PAT). These notions have played an important role in the development of Type Theory after the Second World War. They opened the possibility to use Type Theory not only as a restrictive method (to prevent paradoxes) but also as a constructive method. Many proof checkers and theorem provers, like AUTOMATH [95], Coq [42], Nuprl [34], LEGO [87], LF [59], use the PAT principle.

PAT was discovered independently by different persons. In Section 4a we give a historical sketch. In the next two sections we describe how the two most important type systems of the pre-PAT-era can be described in a PAT style. This gives insight in the various ways in which PAT-implementations can be made.

4a The discovery of PAT

In the first three chapters we discussed type systems the way they were initially designed, namely to prevent the logical paradoxes. But although the systems of both Russell and Church have some logical symbols in them (like $\vee$, $\forall$), these theories themselves cannot be seen as a logical system. If one wants to make logical derivations, one has to build a logical system on top of one of these type systems.

However, type theory nowadays also plays an important role in logic in a different way: It can be used as a logical system itself. This use of type theory is generally known as "propositions as types" or "proofs as terms". As we will see in this section, both expressions only partially cover the idea of using type theory as a logical system. As they both abbreviate to PAT, we will use this abbreviation to indicate both "propositions as types" and "proofs as terms".

"Proofs as terms" already suggests an important advantage of using type theory as a logical system: In this method proofs are first-class citizens of the logical system, whilst for many other logical systems, proofs are rather complex objects outside the logic (for example: derivation trees), and therefore cannot be easily manipulated.

Below we mention some origins of the PAT principle.

4a1 Intuitionistic logic

The idea of PAT originates in the formulation of intuitionistic logic. Though it is not correct that "intuitionistic logic" is simply the logic that is used in intuitionistic mathematics[1], there are frequently occurring constructions

[1] "Intuitionistic logic" is standard terminology for "logic without the law of the excluded middle". The terminology suggests that it is "the logic that is used in intuitionism". However, intuitionism, that is: the philosophy of Brouwer and the mathematics based on that philosophy, declares mathematics to be independent of logic. According to that philosophy, a proof of a mathematical theorem is a method to read that theorem as a tautology. The fact that one needs a list of tautologies before the proof of more complicated theorems becomes clear, only indicates that the constructions we make are too complicated to be comprehended immediately. Mathematics itself however, is a construction in one's mind, independent of logic:

> "Een logische opbouw der wiskunde, onafhankelijk van de wiskundige intuïtie, is onmogelijk — daar op die manier slechts een taalgebouw wordt verkregen, dat van de eigenlijke wiskunde onherroepelijk gescheiden blijft — en bovendien een contradictio in terminis — daar een logisch systeem, zoo goed als de wiskunde zelf, de wiskundige oer-intuïtie nodig heeft"
>
> (*Over de Grondslagen der Wiskunde* [19], p. 180)

(A logical construction of mathematics, independent of the mathematical intuition, is impossible — for by this method no more is obtained than a linguistic structure, which irrevocably remains separated from mathematics — and moreover it is a contradictio in terminis — because a logical system needs the basic intuition of mathematics as much as mathematics itself needs it. [Translation from [63]]).

in intuitionistic mathematics that have a logical counterpart. One of these constructions is the proof of an implication. Heyting [62] describes the proof of an implication $a \Rightarrow b$ as: Deriving a solution for the problem b from the problem a. Kolmogorov [77] is even more explicit, and describes a proof of $a \Rightarrow b$ as the construction of a method that transforms each proof of a into a proof of b. This means that a proof of $a \Rightarrow b$ can be seen as a *(constructive) function* from the proofs of a to the proofs of b. In other words, the proofs of the proposition $a \Rightarrow b$ form exactly the set of functions from the set of proofs of a to the set of proofs of b. This suggests to identify a proposition with the set of its proofs. Now *types* are used to represent these sets of proofs. An element of such a set of proofs is represented as a *term* of the corresponding type. This means that propositions are interpreted as *types*, and proofs of a proposition a as *terms of type a*.

4a2 Curry

PAT was, independently from Heyting and Kolmogorov, discovered by Curry and Feys [38]. In paragraph 8C of [38], Curry describes so-called F-objects, which correspond more or less to the simple types of Church in [30]. As a basis, a list of primitive objects $\vartheta_1, \vartheta_2, \ldots$ is chosen. All these primitive objects are F-objects. Moreover, if α and β are F-objects, then so is $\mathsf{F}\alpha\beta$. Here, F is a new symbol. $\mathsf{F}\alpha\beta$ must be interpreted as the class of functions from α to β. If α is an F-object, then the statement $\vdash \alpha X$ must be interpreted as "the object X belongs to α". The following *rule*-F is adopted: If $\vdash \mathsf{F}XYZ$ and $\vdash XU$ then $\vdash Y(ZU)$. The intuitive meaning of this rule is: If Z belongs to $\mathsf{F}XY$ and U belongs to X, then ZU belongs to Y. This rule immediately corresponds to the application-rule of λ-Church (see Ab).

Earlier in [38], Curry has introduced the combinator P, which is the implication combinator. $\mathsf{P}XY$ can be interpreted as the proposition "if X then Y". The combinator P comes together with a *rule*-P: If $\vdash \mathsf{P}XY$ and $\vdash X$ then $\vdash Y$. Curry notices that this rule has similar behaviour as rule-F.

Curry is the first one to give a formalisation of PAT. For each F-object α he defines a corresponding proposition α^P as follows: $\vartheta_i^P \equiv \vartheta_i$ and $(\mathsf{F}\alpha\beta)^P \equiv \mathsf{P}\alpha^P\beta^P$. Remark that Curry's function $\alpha \mapsto \alpha^P$ is in fact an embedding of types in propositions (so a types-as-propositions embedding instead of a propositions-as-types embedding).

Curry then derives the following theorem, where $\mathsf{F}_m X_1 \cdots X_m Y$ is an abbreviation of $\mathsf{F}X_1(\mathsf{F}X_2(\ldots(\mathsf{F}X_m Y)\ldots))$:

> "If $\vdash \mathsf{F}_m \xi_1 \cdots \xi_m \eta X$ then $\vdash (\mathsf{F}_m \xi_1 \cdots \xi_m \eta)^P$.
> Moreover, if $\vdash \mathsf{F}_m \xi_1 \cdots \xi_m \eta X$ is derivable from the premises $\vdash \alpha_i a_i$ $(i = 1, \ldots, p)$ then $\vdash (\mathsf{F}_m \xi_1 \cdots \xi_m \eta)^P$ is derivable from the premises $\vdash \alpha_i^P$ $(i = 1, \ldots, p)$."
>
> ([38], paragraph 9E, Theorem 1)

In other words: If there is (under certain type conditions $a_i{:}\alpha_i$) an object X that is a function taking arguments of types $\xi_1, \ldots, \xi_m$, resulting in an object of type η, then the corresponding theorem is derivable (if we presuppose α_i^P). Or in short: The types-as-propositions embedding $\alpha \mapsto \alpha^P$ is sound.

The converse of the theorem holds as well:

> "If $\vdash (\mathsf{F}_m \xi_1 \cdots \xi_m \eta)^P$ is derivable by rule-P from the premises $\vdash \alpha_i^P$, then for each derivation of this fact and each assignment of $a_1, \ldots, a_p$ to $\alpha_1, \ldots, \alpha_p$ respectively there exists an X such that $\vdash \mathsf{F}_m \xi_1 \cdots \xi_m \eta X$ is derivable from the premises $\vdash \alpha_i a_i$ $(i = 1, \ldots, p)$ by rule-F alone."
>
> ([38], paragraph 9E, Theorem 2)

In other words: The types-as-propositions embedding $\alpha \mapsto \alpha^P$ is complete.

The treatment of PAT in [38] is mainly directed towards Propositions as Types. Proofs as terms are implicitly present in the theory of [38]: The term X in the proof of Theorem 1 of [38] can be seen as a proof of the proposition $(\mathsf{F}_m \xi_1 \cdots \xi_m \eta)^P$. But this is not made explicit in [38].

Example 4.1 As an example, we show the deduction of the proposition $A \to A$ from the logical axioms $X \to Y \to X$[2] (the K-*axiom*) and $(X \to Y \to Z) \to (X \to Y) \to X \to Z$ (the S-*axiom*), both in the style of the

[2] We assume that $\to$ is associative to the right, i.e. $X \to Y \to Z$ denotes $X \to (Y \to Z)$ and not $(X \to Y) \to Z$.

combinator P and in the PAT-style. Both derivations correspond to the derivation of the proposition $A \to A$ in natural deduction style, with the use of modus ponens, and axioms $X \to Y \to X$ and $(X \to Y \to Z) \to (X \to Y) \to X \to Z$ only:

$$\dfrac{\dfrac{\vdash (A \to (A \to A) \to A) \to (A \to A \to A) \to A \to A \qquad \vdash A \to (A \to A) \to A}{\vdash (A \to A \to A) \to A \to A} \qquad \vdash A \to A \to A}{\vdash A \to A}.$$

- We use $\mathsf{P}_m X_1 \cdots X_m Y$ as an abbreviation for

$$\mathsf{P}X_1(\mathsf{P}X_2(\ldots(\mathsf{P}X_mY)\ldots)).$$

So $\mathsf{P}_m X_1 \cdots X_m Y$ can be interpreted as the proposition

$$X_1 \to X_2 \to \cdots X_m \to Y.$$

In this notation, Rule-P looks as follows:

$$\frac{\vdash \mathsf{P}_{m+1}X_0 \cdots X_m Y \qquad \vdash X_0}{\vdash \mathsf{P}_m X_1 \cdots X_m Y}.$$

For terms X, Y, Z, we take the following axioms:

(K): $\vdash \mathsf{P}_2 XYX$;

(S): $\vdash \mathsf{P}_3(\mathsf{P}_2XYZ)(\mathsf{P}XY)XZ$.

Let A be a term. From the axioms we derive $\vdash \mathsf{P}AA$, using rule-P:

$$\dfrac{\dfrac{\vdash \mathsf{P}_3(\mathsf{P}_2A(\mathsf{P}AA)A)(\mathsf{P}A(\mathsf{P}AA))AA \qquad \vdash \mathsf{P}_2A(\mathsf{P}AA)A}{\vdash \mathsf{P}_2(\mathsf{P}A(\mathsf{P}AA))AA} \qquad \vdash \mathsf{P}A(\mathsf{P}AA)}{\vdash \mathsf{P}AA};$$

- In PAT-style, the situation is similar. Now we do not use any axioms, but we use some standard combinators. The combinator K (which can be compared to the λ-term λxy.x) has type F_2XYX, for arbitrary F-objects X, Y (a term can have more than one type in Curry's theory). K can be seen as a "proof" of the axiom $(\mathsf{F}_2XYX)^P$. This is indicated by putting K behind the axiom:

$$(\mathsf{F}_2XYX)^P\mathsf{K}.$$

The combinator S, comparable to the λ-term $\lambda \mathtt{xyz.xz(yz)}$, has type $\mathsf{F}_3(\mathsf{F}_2XYZ)(\mathsf{F}XY)XZ$ for arbitrary F-objects X, Y, Z. S is a "proof" of the axiom $(\mathsf{F}_3(\mathsf{F}_2XYZ)(\mathsf{F}XY)XZ)^P$. This is denoted as

$$(\mathsf{F}_3(\mathsf{F}_2XYZ)(\mathsf{F}XY)XZ)^P\mathsf{S}.$$

The derivation above now translates to:

$$\dfrac{\dfrac{\begin{array}{c}\vdash \mathsf{F}_3(\mathsf{F}_2A(\mathsf{F}AA)A)(\mathsf{F}A(\mathsf{F}AA))AA\mathsf{S}\\ \vdash \mathsf{F}_2A(\mathsf{F}AA)A\mathsf{K}\end{array}}{\vdash \mathsf{F}_2(\mathsf{F}A(\mathsf{F}AA))AA(\mathsf{SK})} \qquad \vdash \mathsf{F}A(\mathsf{F}AA)\mathsf{K}}{\vdash \mathsf{F}AA(\mathsf{SKK})}.$$

The conclusion of this derivation can be read as: SKK is a function from A to A, or, with PAT in mind: SKK is a proof of the proposition $A \to A$.

Both derivations correspond to the derivation of the proposition $A \to A$ in natural deduction style, with the use of modus ponens, and axioms $X \to Y \to X$ and $(X \to Y \to Z) \to (X \to Y) \to X \to Z$ only:

$$\dfrac{\dfrac{\begin{array}{c}\vdash (A \to (A \to A) \to A) \to (A \to A \to A) \to A \to A\\ \vdash A \to (A \to A) \to A\end{array}}{\vdash (A \to A \to A) \to A \to A} \qquad \vdash A \to A \to A}{\vdash A \to A}.$$

4a3 Howard

Howard [66] follows the argument of Curry and Feys [38] and combines it with Tait's discovery of the correspondence between cut elimination and β-reduction of λ-terms [116].

Example 4.2 The idea is as follows. Consider the following derivation in natural deduction style of a proposition B:

$$\dfrac{\dfrac{\begin{array}{c}[A]\\ \boxed{\mathfrak{D}_1}\\ B\end{array}}{A \to B} \qquad \begin{array}{c}\boxed{\mathfrak{D}_2}\\ A\end{array}}{B}$$

Here, $[A]$ denotes that the assumption A has been discharged at the point where we concluded $A \to B$ from B. $\mathfrak{D}_1$ is a derivation with some assumptions of A, and conclusion B, whilst $\mathfrak{D}_2$ is a derivation with conclusion A. The derivation $\mathfrak{D}_2$ can be used to replace the assumptions of A in derivation $\mathfrak{D}_1$. This means that we can transform the derivation to:

$$\begin{array}{c}\boxed{\mathfrak{D}_2}\\ A\\ \boxed{\mathfrak{D}_1}\\ B\end{array}$$

where copies of $\mathfrak{D}_2$ have replaced the assumptions A in $\mathfrak{D}_1$.

We can decorate the two derivations above with λ-terms that represent proofs. This results in the following two deductions:

$$\dfrac{\dfrac{\begin{array}{c}[x{:}A]\\ \boxed{\mathfrak{D}_1}\\ T : B\end{array}}{(\lambda x{:}A.T) : (A \to B)} \qquad \begin{array}{c}\boxed{\mathfrak{D}_2}\\ S : A\end{array}}{((\lambda x{:}A.T)S) : B}$$

and

$$\begin{array}{c}\boxed{\mathfrak{D}_2}\\ S : A\\ \boxed{\mathfrak{D}_1}\\ T[x{:=}S] : B\end{array}$$

The assumption of A is represented by a variable x of type A. This is a natural idea: the variable expresses the idea "assume we have some proof of A". The derivation $\mathfrak{D}_1$ is represented by a λ-term T, in which the variable x may occur (we can use the assumption A in derivation $\mathfrak{D}_1$). Then the term $\lambda x{:}A.T$ exactly represents a proof of $A \to B$: it is a function that transforms any proof x of A into a proof T of B. As $\mathfrak{D}_2$ is a derivation of A (assume, S is a proof term of A), we can apply $\lambda x{:}A.T$ to S, obtaining a proof $(\lambda x{:}A.T)S$ of B.

Now substituting the derivation $\mathfrak{D}_2$ for the assumptions of A in $\mathfrak{D}_1$ is nothing more than replacing the assumption "assume we have some proof of A" by the explicit proof S, or in other words: substituting S for x. This results in a term T, where each occurrence of x has been replaced by S: the λ-term $T[x{:=}S]$.

We see that the proof transformation exactly corresponds to the β-reduction $(\lambda x{:}A.T)S \rightarrow_\beta T[x{:=}s]$.

This is the first time that proofs are treated as λ-terms. Howard doesn't call these λ-terms "proofs" but "constructions".

Moreover, Howard's treatment of PAT pays attention to both Propositions as Types (following the line of Curry and Feys) and Proofs as Terms (by using λ-terms to represent proofs, thus following the interpretation of logical implication as given by Heyting).

Howard's discovery dates from 1969, but was not published until 1980.

4a4 De Bruijn

Independently of Curry and Feys and Howard, we find a variant of PAT in the first AUTOMATH system of De Bruijn (AUT-68 [95], [21]). Though De Bruijn was probably influenced by Heyting (see [23] in [95], p. 211), his ideas arose independently from Curry, Feys and Howard. This can be clearly seen in Section 2.4 of [20], where propositions as types (or better: Proofs as terms) is implemented in the following way, differing from the method of Curry and Howard.

First, a constant `bool` is introduced. `bool` is a type: The type of propositions. If b is a term of type `bool` (so b is a proposition), then `true`(b) is a primitive notion of type `type`. `true`(b) represents the type of the proofs of b. So, a proof of proposition b is of type `true`(b) and not of type b (since propositions themselves are no types) With this "bool-style" implementation (as it was called by De Bruijn in [23]) in mind, it becomes clear why De Bruijn prefers the terminology "proofs as terms" to "propositions as types": In the bool-style implementation, propositions are not represented as types. Only the class of proofs of such a proposition is represented as a type. Proofs however, are represented as terms, just as in Howard's implementation of PAT. So in the bool-style implementation, the link between proposition and type is not as direct as the link between proof and term. The implementation of Howard (called "prop-style" by De Bruijn) does not make any distinction between a proposition and the type of its proofs.

The bool-style implementation has as advantage that one does not need a higher order lambda calculus to construct predicate logic. In relatively weak AUTOMATH systems such as AUT-68 one usually finds a "bool-style" implementation of PAT. It would be impossible to give a "prop-style" implementation in such a system as its λ-calculus is not strong enough to support it. In AUTOMATH systems with a more powerful λ-calculus we also find "prop-style" implementations. See [92] for a global description of how prop-style implementations are made in AUTOMATH.

Another advantage of the bool-style implementation is that one does not depend on a fixed interpretation of the logical connectives. One is free to define ones own logical system (and it is possible to base that system on the Brouwer-Heyting-Kolmogorov interpretation of the logical connectives, just like the prop-style implementation of PAT). This has been one of the reasons for De Bruijn to implement PAT in a bool-style way (see [23]).

Though the bool-style implementation has disappeared from later AUTOMATH systems, it is still in use in the Edinburgh Logical Framework [59], and the systems proposed in certain formulations of the Calculus of Constructions by Luo [86], Streicher [115], and Altenkirch [3]. Luo defines a class of proofs, called $\mathrm{Prf}(P)$, for each proposition P, and explicitly defines how to construct a proof of a proposition of the form $\forall x{:}A.P$ by giving a rule

$$\frac{\Gamma, x{:}A \vdash M : \mathrm{Prf}(P)}{\Gamma \vdash (\lambda x{:}A.M) : \mathrm{Prf}(\forall x{:}A.P)}.$$

Streicher and Altenkirch have a somewhat different approach. They also define a class of proofs for each proposition P (in Altenkirch's thesis this class is denoted $\mathrm{El}(P)$), but the proofs are constructed in a somewhat different way: An equality relation $\simeq$ is introduced, and the class of proofs $\mathrm{El}(\forall x{:}A.P)$ is explicitly declared equivalent to the type $\Pi x{:}A.\mathrm{El}(P)$:

$$\mathrm{El}(\forall x{:}A.P) \simeq \Pi x{:}A.\mathrm{El}(P).$$

This type $\Pi x{:}A.\mathrm{El}(P)$ represents the class of functions f with domain A, and such that for each a in A, fa is an element of $\mathrm{El}(P)[x{:=}a]$, so a proof of $P[x{:=}a]$.

4b RTT in PAT style

In this section we show that the system RTT of Chapter 2 can be described in a PAT style using the prop-style implementation of Curry and Howard. This will give us a better view on the various ways in which PAT can be implemented.

Before we can give a description, we must make the following observations:

- Russell and Whitehead designed their system for classical logic. As the PAT principle in prop style is based on intuitionistic logic, we need to supply extra logical axioms to obtain the classical logic of Russell and Whitehead;

- RTT is constructed with the logical connectives $\vee$, $\neg$, $\forall$, while the PAT principle is strongly based on the interpretation of $\rightarrow$ and $\forall$ as function types. In the sequel of this section we will work with the symbols $\forall$, $\rightarrow$, and an additional symbol $\perp$, representing falsum. This makes it possible to interpret a proposition $\neg f$ by $f \rightarrow \perp$, and a proposition $f \vee g$ by $(f \rightarrow \perp) \rightarrow g$;

- As RTT distinguishes between propositions of various orders, it is not enough to provide one class of types. We must distinguish between several classes of types, corresponding to the orders of RTT.[3]

In Section 4b1–4b3 we present a type system λRTT for a PAT representation of RTT. The type system is (almost) a so-called *pure* type system. Pure type systems (PTSs) were invented by Terlouw [118] and Berardi [13] as a framework in which many type systems can be described. While defining

[3]Something similar is done in the proof checker Nuprl, where type universes $\mathcal{U}_1, \mathcal{U}_2, \ldots$ are introduced. The type universe $\mathcal{U}_n$ contains all objects of order $\leq n$. The approach below is not exactly similar to the Nuprl approach. In Nuprl, $\mathcal{U}_n$ is not only a subset of $\mathcal{U}_{n+1}$, but also an element of $\mathcal{U}_{n+1}$. See [69] (where these type universes are denoted $*_1, *_2, \ldots$), [67], [34]. This is the case neither in RTT nor in system λRTT below.

λRTT we try to give sufficient intuition about PTSs in order to understand the ideas behind λRTT. A short overview of PTSs is given in Appendix A. More details can be found in [5], [54] and [55].

In Section 4b4 we make a comparison between λRTT and RTT, and in Section 4b5 we give some examples on how to "do" logic of the *Principia* in λRTT.

In Section 4b6 we discuss in which ways PAT can be implemented. The various implementations lead to different level structures within the resulting PTS.

4b1 An introduction to λRTT

We now present a system λRTT that will be suitable for a PAT representation of RTT.

Definition 4.3 (Terms of λRTT) Let $\mathcal{A}$, $\mathcal{V}$ and $\mathcal{R}$ be as in Chapter 2. Define the set $\mathcal{T}$ of terms of λRTT by:

$$\begin{aligned} \mathcal{T} \;::=\; & *_s \mid *_{\mathbb{N}^+} \mid \Box_{\mathbb{N}^+} \mid \mathcal{A} \mid \mathcal{V} \mid \mathcal{R} \mid \iota \mid \bot \mid \\ & \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}. \end{aligned}$$

Remark 4.4 In this definition, $*_s$, $*_n$ (for $n \in \mathbb{N}^+$) and $\Box_n$ (for $n \in \mathbb{N}^+$) are so-called *sorts*.

- $*_s$ is the sort of object types. There will be only one object type in λRTT, namely the type of individuals ι. An individual a has type ι, this is denoted: $a : \iota$;
- $*_n$ (for $n \in \mathbb{N}^+$) is the sort containing the propositions of order $\leq n$. Notice that these propositions will be represented as *types*: We are presenting a PAT version of RTT;
- $\Box_n$ (for $n \in \mathbb{N}^+$) contains $*_n$, and (translations of) the ramified types of order n as they occur in RTT.

We write $\boldsymbol{S}$ for the set of sorts $\{*_s, *_1, \Box_1, *_2, \Box_2, \dots\}$.

Remark 4.5 A term of the form $\Pi x{:}A.B$ denotes a (dependent) function type. That is: $\Pi x{:}A.B$ is the type that contains functions f with domain

A, and range $\bigcup_{x:A} B$ (it is possible that x occurs free in B), such that fa has type $B[x{:=}a]$ for all a of type A. Such function types can occur at several places:

- First of all, the translations of the propositional functions of a certain ramified type will belong to a function type. Look for instance at the ramified type $(0^0)^1$. Pfs of this type are functions that take an individual as argument, and return a proposition of order ≤ 1 as result.[4] This suggests to translate the ramified type $(0^0)^1$ by the function type $\Pi \mathrm{x}{:}\iota.*_1$ (see the forthcoming Definition 4.13);

- Secondly, certain propositions will be represented as function types. This has its origin in Heyting's description of the proof of an implication as a function. If the proof of an implication is a function, then the implication itself (a proposition, or equivalently, the type of all the proofs of a proposition) must be a function type. For example, the RTT proposition $\mathtt{R(a)} \to \mathtt{R(b)}$ will (in 4.13) be translated into the type $\Pi \mathrm{x}{:}\mathtt{Ra}.\mathtt{Rb}$.

  A universal quantification will also be translated as a function type. According to Heyting, the proof of a proposition $\forall x{:}A[B]$ is a function that takes elements a of A as arguments, and returns proofs of $B(a)$. For example, the RTT proposition $\forall \mathrm{x}{:}0^0[\mathrm{R(x)}]$ will (again: In 4.13) be translated by $\Pi \mathrm{x}{:}\iota.\mathrm{Rx}$. Here we see an example of a type $\Pi x{:}A.B$ where B depends on the variable x. The intuition of a function of type $\Pi \mathrm{x}{:}\iota.\mathrm{Rx}$ coincides with the intuition of a proof of the proposition $\forall \mathrm{x}{:}0^0[\mathrm{R(x)}]$: A function taking individuals a as arguments, and returning a proof of $\mathtt{R}(a)$.

The type system λRTT will have a rule to introduce terms of type $\Pi x{:}A.B$ (provided we have a term of type B): If b is a term of type B (possibly x occurs free in b), then $\lambda x{:}A.b$ is of type $\Pi x{:}A.B$. This can be understood with the above interpretations of $\Pi x{:}A.B$ in mind. For instance, we represented the RTT-proposition $\forall \mathrm{x}{:}0^0[\mathrm{R(x)}]$ by $\Pi \mathrm{x}{:}\iota.\mathrm{Rx}$. If we

[4]The proposition that is returned can be of order 1, for instance in the case we substitute the individual a for x in the pf $\forall \mathrm{y}{:}0^0[\mathtt{R(x,y)}]$, or of order 0, for instance in the case we substitute a for x in the pf $\mathtt{R(x,x)}$. However, the returned proposition can never be of order > 1, due to Corollary 2.63. See also Remark 4.7.

have a term b of type $\mathtt{R}\mathtt{x}$ (so b proves $\mathtt{R}\mathtt{x}$ for an arbitrary individual $\mathtt{x}$) then the function $\lambda \mathtt{x}{:}\iota.b$, assigning $b[\mathtt{x}{:=}a]$ to each $a \in \iota$, is indeed a proof of the proposition $\forall \mathtt{x}{:}0^0[\mathtt{R}(\mathtt{x})]$.

The system λRTT also has a rule that shows what can be done with a term of type $\Pi x{:}A.B$. If a is a term of type ι, and f a proof of $\Pi \mathtt{x}{:}0^0.\mathtt{R}\mathtt{x}$, then we can apply f to a, thus obtaining fa of type $\mathtt{R}\mathtt{x}[\mathtt{x}{:=}a]$, which is $\mathtt{R}a$. Indeed applying the function f to an individual a gives a proof of $\mathtt{R}a$.

Remark 4.6 It is usual to write $A \to B$ for $\Pi x{:}A.B$, if x does not occur free in B. Hence, in *notation* there is hardly any difference between the RTT proposition $\mathtt{R}(\mathtt{a}) \to \mathtt{R}(\mathtt{b})$ (where $\to$ denotes logical implication) and the λRTT type $\mathtt{Ra} \to \mathtt{Rb}$ (where $\to$ is used to form a function type).

Remark 4.7 One may wonder why we chose $*_n$ to be the type of all propositions of order $\leq n$ instead of the type of all propositions of order $= n$. This has a technical reason that already popped up in footnote 4, namely the translation of the ramified types of RTT into λ-terms. Consider a ramified type $(t_1^{a_1}, \ldots, t_n^{a_n})^a$. If $\tau_1, \ldots, \tau_n$ are appropriate translations of $t_1^{a_1}, \ldots, t_n^{a_n}$, then one would like to translate $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ into $\tau_1 \to \cdots \to \tau_n \to *_m$, where $*_m$ represents the type of propositions of some order m. This would suggest that a propositional function f of type $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ always results in a proposition of (fixed) order m as soon as values of types $t_1^{a_1}, \ldots, t_n^{a_n}$ are substituted for its free variables. However, it is impossible to determine such m:

- Consider the pfs $f \equiv \mathtt{z}(\mathtt{a})$ and $g \equiv \mathtt{z}(\mathtt{a}) \vee \forall \mathtt{z}'{:}(0^0)^1[\mathtt{z}'(\mathtt{b})]$ in a context $\Gamma = \left\{\mathtt{z}{:}(0^0)^1\right\}$. Observe: $\Gamma \vdash f : \left((0^0)^1\right)^2$ and $\Gamma \vdash g : \left((0^0)^1\right)^2$. But if we substitute $\mathtt{R}(\mathtt{x})$ for $\mathtt{z}$ in both f and g, we obtain $\mathtt{R}(\mathtt{a})$ (a proposition of order 0) and $\mathtt{R}(\mathtt{a}) \vee \forall \mathtt{z}'{:}(0^0)^1[\mathtt{z}'(\mathtt{b})]$ (a proposition of order 1).

  So substituting the same propositional function $\mathtt{R}(\mathtt{x})$ in two different propositional functions f and g of the same type, may result in propositions of different orders;

- Let f be as above, and extend Γ with the declaration $\mathtt{x}{:}0^0$. Notice: Both $h_1 \equiv \mathtt{R}(\mathtt{x})$ and $h_2 \equiv \mathtt{R}(\mathtt{x}) \vee \forall \mathtt{y}{:}0^0[\mathtt{S}(\mathtt{y})]$ are pfs of type $(0^0)^1$, so they can be substituted for $\mathtt{z}$ in f. As a result we obtain $\mathtt{R}(\mathtt{a})$, a proposition of order 0, and $\mathtt{R(a)} \vee \forall \mathtt{y{:}0^0[S(y)]}$, a proposition of order 1. So substituting different propositional functions h_1, h_2 of the same type in one and the same propositional function f may result in propositions of different orders.

Therefore, we cannot interpret $*_m$ in $\tau_1 \to \cdots \to \tau_n \to *_m$ as the type of propositions of order m. However, by Corollary 2.63, the order of the proposition $f[x_1, \ldots, x_n{:=}g_1, \ldots, g_n]$ cannot be higher than the order of f. Hence, it is safe to translate the ramified type $(t_1^{a_1}, \ldots, t_n^{a_n})^m$ into $\tau_1 \to \cdots \to \tau_n \to *_m$, if we interpret $*_m$ as the type of propositions of order $\leq m$.

One might wonder whether this somewhat unusual interpretation of $*_m$ makes λRTT much different from the original RTT. The idea of "propositions of order $\leq m$" however, is not very different from the idea of "propositions of order m", as all propositions of order $< m$ have a logically equivalent proposition of order m (at least in the logic that Russell and Whitehead had in mind): If f has order $< m$ then

$$f \vee \forall \mathtt{z{:}()}^{m-1}[\mathtt{z()} \vee \neg \mathtt{z()}]$$

is of order m, and logically equivalent to f. The system λRTT will have a special rule (Incl) stating that any proposition of order $\leq m$ is also a proposition of order $\leq m+1$.

Contexts, as in the situation of Chapter 2, contain information on the types of variables. These variables can be of different nature: First of all, we still have the variables of Chapter 2. These variables live at the level of types, as propositions are interpreted as types. But now we can also have variables that serve as assumptions: If A is a proposition, then a variable of type A refers to an arbitrary proof of A. If a context contains such a declaration $x{:}A$, then this can be read as: It is supposed that A holds (and that x is some proof of A).

In Chapter 2, contexts were sets. In particular, the order in which the various declarations were mentioned, did not matter: $\{\mathtt{x{:}0^0}, \mathtt{y{:}()^1}\}$ is not different from $\{\mathtt{y{:}()^1}, \mathtt{x{:}0^0}\}$. Now, types that occur in a context can depend on variables that are declared somewhere else in that context. We do not want a variable to occur in a context before its type has been declared. Therefore, we present contexts as lists. The order in which the variables are declared is determined by the order in the list: In a context $\langle \mathtt{x}{:}\tau, \mathtt{y}{:}\upsilon\rangle$, the variable $\mathtt{x}$ was declared before $\mathtt{y}$. Thus it is possible that υ depends on $\mathtt{x}$, i.e. $\mathtt{x}$ may occur as a free variable in υ. On the other hand, $\mathtt{y}$ must not occur in τ, as $\mathtt{y}$ is declared after τ has appeared. Therefore, the context $\langle \mathtt{x}{:}\tau, \mathtt{y}{:}\upsilon\rangle$ is different from the context $\langle \mathtt{y}{:}\upsilon, \mathtt{x}{:}\tau\rangle$.

The presented intuition leads to the type system in Definition 4.9 below, which is in fact almost a PTS. The Π-form rule determines which types (and, with the PAT principle in mind: Which propositions) can be constructed in the system. The general form of the Π-formation rule is

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

where s_1, s_2, s_3 are sorts. By specifying for which combinations of triples (s_1, s_2, s_3) the Π-formation rule can be applied, one can control which Π-types can be constructed. We now informally discuss which Π-formation rules we need in a PAT description of RTT. After that, we give a formal presentation of the system λRTT.

4b1.1 The translations of the ramified types

To translate and type the ramified types, we need the rules $(*_s, \Box_n, \Box_n)$ and $(\Box_m, \Box_n, \Box_n)$ for $n \geq 1$ and $m < n$.

Assume $(t_1^{a_1}, \ldots, t_p^{a_p})^a$ is a ramified type, and that we already found proper translations $\tau_1, \ldots, \tau_p$ for the ramified types $t_1^{a_1}, \ldots, t_p^{a_p}$ such that

- $\vdash_{\lambda\mathrm{RTT}} \tau_i : *_s$ if $t_i^{a_i} \equiv \iota$;
- $\vdash_{\lambda\mathrm{RTT}} \tau_i : \Box_{a_i}$ if $t_i^{a_i} \not\equiv \iota$.

(we use $\vdash_{\lambda\mathrm{RTT}}$ to denote derivability in λRTT, the PAT version of RTT). The type system λRTT will have an axiom rule that declares that $*_a$ has type $\Box_a$, therefore we have: $\mathtt{x}_p{:}\tau_p \vdash_{\lambda\mathrm{RTT}} *_a : \Box_a$. Now distinguish:

- If $t_p^{a_p} \equiv \iota$ then we use Π-formation rule $(*_s, \Box_a, \Box_a)$:

$$\frac{\vdash_{\lambda\mathrm{RTT}} \tau_p : *_s \qquad \mathtt{x}_p{:}\tau_p \vdash_{\lambda\mathrm{RTT}} *_a : \Box_a}{\vdash_{\lambda\mathrm{RTT}} (\Pi \mathtt{x}_p{:}\tau_p.*_a) : \Box_a};$$

- If $t_p^{a_p} \not\equiv \iota$ then we use Π-formation rule $(\Box_{a_p}, \Box_a, \Box_a)$ (notice that $a_p < a$):
$$\frac{\vdash_{\lambda\mathrm{RTT}} \tau_p : \Box_{a_p} \qquad \mathbf{x}_p{:}\tau_p \vdash_{\lambda\mathrm{RTT}} *_a : \Box_a}{\vdash_{\lambda\mathrm{RTT}} (\Pi \mathbf{x}_p{:}\tau_p.*_a) : \Box_a}.$$

In λRTT there will be a weakening rule so that we can weaken this conclusion by adding a variable $\mathbf{x}_{p-1}$:

$$\mathbf{x}_{p-1}{:}\tau_{p-1} \vdash_{\lambda\mathrm{RTT}} (\Pi \mathbf{x}_p{:}\tau_p.*_a) : \Box_a.$$

In a similar way, we can now deduce:

$$\vdash_{\lambda\mathrm{RTT}} (\Pi \mathbf{x}_{p-1}{:}\tau_{p-1}.\Pi \mathbf{x}_p{:}\tau_p.*_a) : \Box_a$$

and so on until we have

$$\vdash_{\lambda\mathrm{RTT}} (\Pi \mathbf{x}_1{:}\tau_1.\cdots.\Pi \mathbf{x}_p{:}\tau_p.*_a) : \Box_a.$$

Notice that the variables $\mathbf{x}_i$ are only dummy variables, and do not occur in any of the τ_j. We can therefore write

$$\tau_1 \to \cdots \to \tau_p \to *_a$$

instead of $(\Pi \mathbf{x}_1{:}\tau_1.\cdots.\Pi \mathbf{x}_p{:}\tau_p.*_a) : \Box_a$. This $\tau_1 \to \cdots \to \tau_p \to *_a$ is the translation of the RTT-type $(t_1^{a_1}, \ldots, t_p^{a_p})^a$.

4b1.2 The translation of the logical implication

The translation of propositional functions will have a lot of similarities with the mapping $\bar{}$ of Definition 2.7. A propositional function f with free variables $x_1 < \cdots < x_m$ will be translated into a λ-term $\lambda x_1{:}\tau_1 \ldots x_m{:}\tau_m.F$, where F is a λ-term of type $*_n$, n is the order of f, and $\tau_1, \ldots, \tau_m$ are translations of the types of the variables $x_1, \ldots, x_m$. In this way, the translation of f will have type $\tau_1 \to \ldots \to \tau_m \to *_n$, which is exactly the translation of the type of f. In this subsection, we focus on the translation of propositions of the form $f \to g$; in the next subsection we focus on the translation of propositions of the form $\forall x{:}A[f]$.

If f and g are propositions, we must be able to form the proposition $f \to g$. According to Russell's definition in *Principia Mathematica*, $f \to g$

is only shorthand for $(\neg f) \vee g$. In a system in PAT style, the implication plays a more central role, as the logical connectives $\neg$ and $\vee$ are defined with the use of implication.

If f is a proposition of order m, and g has order n, then $f \to g$ clearly has order $\max(m,n)$. In PAT, we can obtain this effect via a rule $(*_m, *_n, *_{\max(m,n)})$. We can assume that F is a translation of f, and G is a translation of g, such that $\vdash_{\lambda\text{RTT}} F : *_m$ and $\vdash_{\lambda\text{RTT}} G : *_n$. Introducing a variable $\mathtt{x}$ of type F does not affect the fact that G is a type of sort $*_n$: $\mathtt{x}{:}F \vdash_{\lambda\text{RTT}} G : *_n$ (the system λRTT will have a so-called *weakening rule* that formalises this intuition). Applying the Π-formation rule $(*_m, *_n, *_{\max(m,n)})$ results in

$$\frac{\vdash_{\lambda\text{RTT}} F : *_m \qquad \mathtt{x}{:}F \vdash_{\lambda\text{RTT}} G : *_n}{\vdash_{\lambda\text{RTT}} (\Pi \mathtt{x}{:}F.G) : *_{\max(m,n)}}.$$

Notice that $\mathtt{x}$ is only a dummy variable here, and does not occur in G. Therefore we can write $F \to G$ instead of $\Pi\mathtt{x}{:}F.G$. This $F \to G$ is the translation of the RTT-proposition $f \to g$.

4b1.3 The translation of the universal quantifier

If f is a pf with x as one and only free variable, then we want to translate f by some term $\lambda x{:}\tau.F$. With PAT in mind, we want to translate $\forall x{:}t[f]$ by $\Pi x{:}\tau.F$: A proof of $\forall x{:}t[f]$ must be a function that assigns a proof of Fa to each a of type τ.

The construction of $\Pi x{:}\tau.F$ can be done in the following way. We can assume that $\lambda x{:}\tau.F$ has a type that corresponds to a propositional function with one variable: A type of the form $\tau \to *_n$, where n is the order of f. This means that F itself must be of type $*_n$, where n is the order of f. The term $\Pi x{:}\tau.F$ (a translation of $\forall x{:}t[f]$) should have type $*_n$, as it represents a proposition of order n. Now τ is a translation of a ramified type, therefore it is of sort $*_s$ (if $\tau \equiv \iota$) or sort $\square_m$ (if $\tau \not\equiv \iota$). For the construction of $\Pi x{:}\tau.F$ we need the Π-formation rule $(*_s, *_n, *_n)$ or $(\square_m, *_n, *_n)$. Notice that x is a free variable of f, so its order is smaller than the order of f. In other words: $m < n$.

The rules $(*_s, *_n, *_n)$ can also be used to represent universal quantification over pfs with more than one free variable. We discuss the situation for two free variables; for three or more free variables the procedure is similar.

Let g be a pf with two free variables $x_1 < x_2$, having type $(t_1^{a_1}, t_2^{a_2})^a$, where $t_i^{a_i}$ is the type of x_i. We assume that g has been translated into a term of the form $\lambda x_1{:}T(t_1^{a_1}).\lambda x_2{:}T(t_2^{a_2}).G$, of type $T(t_1^{a_1}) \to T(t_2^{a_2}) \to *_n$, where n is the order of g. With the definition of $^-$ in mind, we translate $\forall x_1{:}t_1^{a_1}[g]$ into

$$\lambda x_2{:}T(t_2^{a_2}).\Pi x_1{:}T(t_1^{a_1}).G,$$

and $\forall x_2{:}t_2^{a_2}[g]$ by

$$\lambda x_1{:}T(t_1^{a_1}).\Pi x_2{:}T(t_2^{a_2}).G.$$

We can form $\Pi x_1{:}T(t_1^{a_1}).G$ in a similar way as $\Pi x{:}\tau.F$ above, using Π-formation rule $(*_s, *_n, *_n)$ or $(\Box_m, *_n, *_n)$ for an $n \geq 1$ and an $m < n$. The term $\Pi x_1{:}T(t_1^{a_1}).G$ has type $*_n$, and $\lambda x_2{:}T(t_2^{a_2}).\Pi x_1{:}T(t_1^{a_1}).G$ will have type $T(t_2^{a_2}) \to *_n$, which is the appropriate type for a translation of $\forall x_1{:}t_1^{a_1}[g]$. Similarly, $\Pi x_2{:}T(t_2^{a_2}).G$ has type $*_n$, and $\lambda x_1{:}T(t_1^{a_1}).\Pi x_2{:}T(t_2^{a_2}).G$ has type $T(t_1^{a_1}) \to *_n$, which is appropriate for a translation of $\forall x_2{:}t_2^{a_2}[g]$.

4b2 The system λRTT

Now that we have explained the way in which types are constructed in λRTT, we present the system in a formal way. As announced, it will (almost) have the form of a Pure Type System (PTS).

A PTS always has five fixed rules, and two "flexible" rules (axioms and Π-formation rules):

(Axioms) Axioms provide the types of certain constants that are used in the system. This rule is flexible: The axioms may vary from one PTS to another. In λRTT, we have the following axioms:

- $*_n : \Box_n$ for $n \geq 1$. See Remark 4.4;
- $\bot : *_1$. This means that we consider falsum to be a proposition of the lowest order;
- $\iota : *_s$. See Remark 4.4;
- $a : \iota$ for $a \in \mathcal{A}$. Each individual belongs to the type ι of individuals;
- $R : \underbrace{\iota \to \cdots \to \iota}_{\mathfrak{a}(R) \text{ times } \iota} \to *_1$ for $R \in \mathcal{R}$. This illustrates that R is an $\mathfrak{a}(R)$-ary relation on individuals.

All axioms are derivable in an empty context;

(Start) If we want to introduce a variable, we can only do this if we assign a *type* to such a variable. In the type systems that we saw before (like RTT and $\lambda\!\rightarrow$), it was always clear what the types of a certain type system are. There were always two definitions: First a definition that describes which types are allowed, and then a definition that describes which terms have what type. In a PTS however, terms and types are mixed up in one derivation system. Types are recognised as follows:

- Each sort $s \in \boldsymbol{S}$ is a type;
- If $\Gamma \vdash A : s$ for a $s \in \boldsymbol{S}$ then A is a type (within the context Γ).

In the second case, we see that the type A has also a type itself, namely: s. For these "typable" types we make it possible to introduce variables via the start rule:

$$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A}.$$

This means that we can only introduce a variable of type A if A itself has a type $s \in \boldsymbol{S}$. In λRTT, this is possible for the sort $*_n$ (because this sort has type $\Box_n$). This means that it is possible to introduce variables for propositions of order n. However, $\Box_n$ cannot be typed itself in λRTT. This is not harmful, as we do not want to introduce variables of type $\Box_n$. Such a variable would be a variable for a ramified type of order $\leq n$, and such variables do not occur in RTT, either.

As the introduced variable must be seen as a new object, we demand that x is "fresh": It must not occur anywhere in Γ or A;

(Weakening) Once we have derived $\Gamma \vdash M : N$, we want to be able to add variables to the context Γ. Compare this to the weakening rule 2.45.5 for RTT. To add such a variable to the context, we take the same precautions as in the start rule, so we have a premise $\Gamma \vdash A : s$ if we want to add a variable of type A to Γ. The weakening rule now becomes:

$$\frac{\Gamma \vdash M : N \qquad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash M : N}.$$

Again, we demand that x is fresh: It must not occur in Γ, M, N, or A;

(Π-formation) This rule describes which Π-types can be constructed. It is flexible: In different PTSs, different Π-types may be allowed. We already discussed this rule in Section 4b1. In Subsections 4b1.1-4b1.3 we described which Π-formation rules are needed for λRTT:

- $(*_s, \Box_n, \Box_n)$ and $(\Box_m, \Box_n, \Box_n)$ $(m < n)$ to translate the ramified types;
- $(*_m, *_n, *_{\max(m,n)})$ to translate the logical implication;
- $(*_s, *_n, *_n)$ and $(\Box_m, *_n, *_n)$ $(m < n)$ to translate the universal quantification;

(Π-introduction) This rule describes how we can form terms of type $\Pi x{:}A.B$, once we have established that the type $\Pi x{:}A.B$ can be constructed. If $\Gamma \vdash (\Pi x{:}A.B) : s$ has been derived (using the Π-formation rules), we can not only introduce variables of this type with the start and weakening rules, but we can also form terms of this type by λ-abstraction:

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

This rule is a modern version of the Abstraction Principles 1.1 and 1.2, and the abstraction rules 2.45.3 and 2.45.4 for RTT.

The Π-introduction rule is also called λ-*formation*-rule;

(Π-elimination) This rule describes what can be done with a term f of type $\Pi x{:}A.B$. The intuition is clear: f is a function that takes arguments of type A, so it should be possible to apply f to any a of type A. And indeed:

$$\frac{\Gamma \vdash f : \Pi x{:}A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x{:=}a]}.$$

The substitution in the type of the result is necessary. This can be seen if we interpret $\Pi x{:}A.B$ as a proposition $\forall x{:}A.B$. Then f is a proof of $\forall x{:}A.B$, and fa is a proof of B where x has been replaced by

a, so $B[x{:=}a]$. The Π-elimination rule together with β-reduction can be seen as a modern substitution rule. We already saw in Chapter 2 that substitution in RTT can be seen as function application plus β-reduction to normal form in λ-calculus. Indeed: If we apply a term $\lambda x{:}A.b$ to a term a, we get $(\lambda x{:}A.b)a$, which β-reduces to $b[x{:=}a]$.

The Π-elimination rule is also called *application*-rule;

(Conversion) If a term A β-reduces to a term A', we consider A and A' to be equal, in some way. A property of PTSs is that if $\Gamma \vdash A : B$ and $A \to_\beta A'$, then also $\Gamma \vdash A' : B$ (the so-called "Subject Reduction" property). However, we do not have the "Type Reduction" property that $\Gamma \vdash A : B'$ whenever $\Gamma \vdash A : B$ and $B \to_\beta B'$. This property does not even hold if we demand that $\Gamma \vdash B' : s$ for some $s \in \boldsymbol{S}$. As we want to have that β-equal types have the same inhabitants, we introduce a conversion rule:

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

As was already observed in the motivation for the start rule, there is an important difference between PTSs and the other type systems that have appeared in this thesis up till now. The other systems always made a clear distinction between *terms* and *types*. The definition of types is usually given first, and then a second definition indicates which terms are of what type. This distinction is not made in PTSs. There, types and terms are defined in one system. The jargon used may be confusing to the reader that is not familiar with PTSs, and therefore we give the following definition:

Definition 4.8 (Terms and Types)

Terms

- A term A is a *legal term* in a context Γ, if there is B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$;
- *A is a term of type B* in a context Γ, if $\Gamma \vdash A : B$;

Types

- A term A is a *type* in a context Γ if there is a sort $s \in \boldsymbol{S}$ such that $\Gamma \vdash A : s$;

- A is *the type of* B in a context Γ if $\Gamma \vdash B : A$;
- A type A is *inhabited* in a context Γ if there is a term B such that $\Gamma \vdash B : A$.

So: a type is always a term. And: a term can sometimes act as a type. Look for instance at the Π-introduction rule. In the right-hand premise, $\Pi x{:}A.B$ is a term of type s. But in the conclusion, it acts as a type: the term $\lambda x{:}A.b$ is of type $\Pi x{:}A.B$.

For λRTT we need a rule in addition to the seven rules that were stated for PTSs above. That is why we said that λRTT is *almost* a PTS. This is the so-called *inclusion*-rule, which describes the intuition behind $*_n$ as the class of propositions of order $\leq n$ (and not only the class of propositions of order n). We can formulate this rule as follows:

$$\frac{\Gamma \vdash A : *_n}{\Gamma \vdash A : *_{n+1}}.$$

We summarise the eight rules in the following definition:

Definition 4.9 (Derivation Rules for λRTT) Let s, s_1, s_2 range over $\boldsymbol{S} = \{*_s, \Box_1, \Box_2, \ldots, *_1, *_2, \ldots\}$, and let

$$\begin{aligned}\boldsymbol{R} \quad = \quad & \{(*_s, \Box_n, \Box_n) \mid n \geq 1\} \cup \\ & \{(\Box_m, \Box_n, \Box_n) \mid 1 \leq m < n\} \cup \\ & \{(*_m, *_n, *_{\max(m,n)}) \mid m, n \geq 1\} \cup \\ & \{(*_s, *_n, *_n) \mid n \geq 1\} \cup \\ & \{(\Box_m, *_n, *_n) \mid 1 \leq m < n\}.\end{aligned}$$

The derivation rules for λRTT are as follows:

$$\begin{array}{lrl} \textbf{(Axioms)} & \vdash *_n : \Box_n & (n \geq 1) \\ & \vdash \iota : *_s & \\ & \vdash \bot : *_1 & \\ & \vdash a : \iota & (a \in \mathcal{A}) \\ & \vdash R : \underbrace{\iota \to \cdots \to \iota}_{\mathfrak{a}(R) \text{ times } \iota} \to *_1 & (R \in \mathcal{R}) \end{array}$$

(Start) $$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A}$$

(Weak) $$\frac{\Gamma \vdash M : N \qquad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash M : N}$$

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

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

(Π-el) $$\frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x{:=}N]}$$

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

(Incl) $$\frac{\Gamma \vdash A : *_n}{\Gamma \vdash A : *_{n+1}}$$

The introduced variables x in the Start and Weakening rules are assumed to be fresh. If confusion with derivation rules of other type systems might arise, we use $\vdash_{\lambda\text{RTT}}$ instead of $\vdash$ to indicate derivability in λRTT.

4b3 Meta-properties of λRTT

We now describe some meta-properties of λRTT. Their formulation is very close to the formulation of the usual meta-properties of PTS, as described in Section Ac of the Appendix. However, there are a few deviations. This is due to the rule (Incl), which is not a rule in PTSs. The proofs of the meta-properties below are as in the standard literature on PTSs ([5], [55]). In Remark 4.12 we provide some intuition behind these meta-theorems and compare them with the meta-theorems we found for RTT in Chapter 2.

Theorem 4.10 (Properties of λRTT)

1. **(Church-Rosser)** *If $A \twoheadrightarrow_\beta B_1$ and $A \twoheadrightarrow_\beta B_2$ then there is a C such that $B_1 \twoheadrightarrow_\beta C$ and $B_2 \twoheadrightarrow_\beta C$;*

2. **(Free Variables)** *Let $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$, and assume $\Gamma \vdash A : B$. Then:*

   - $\text{FV}(A) \cup \text{FV}(B) \subseteq \text{DOM}(\Gamma)$;
   - *For all $1 \leq i \leq n$ there is $s_i \in \boldsymbol{S}$ such that $x_1{:}A_1, \ldots, x_{i-1}{:}A_{i-1} \vdash A_i : s_i$;*

3. **(Substitution)** *Assume* $\Gamma, x{:}A, \Delta \vdash B{:}C$ *and* $\Gamma \vdash D{:}A$. *Then* $\Gamma, \Delta[x{:=}D] \vdash B[x{:=}D] : C[x{:=}D]$;

4. **(Thinning)** *Assume* Γ, Δ *are legal and* $\Gamma \subseteq \Delta$. *Then* $\Gamma \vdash A : B \Rightarrow \Delta \vdash A : B$;

5. **(Generation)**

   (a) *If* $\Gamma \vdash *_n : C$ *then* $C \equiv \Box_n$;
   If $\Gamma \vdash \iota : C$ *then* $C \equiv *_s$;
   If $\Gamma \vdash \bot : C$ *then* $C \equiv *_n$ *for some* $n \in \mathbb{N}$;
   For $a \in \mathcal{A}$, *if* $\Gamma \vdash a : C$ *then* $C \equiv \iota$;
   For $R \in \mathcal{R}$, *if* $\Gamma \vdash R : C$ *then* $C \equiv \underbrace{\iota \to \cdots \to \iota}_{\mathfrak{a}(R) \text{ times } \iota} \to *_1$ *or* $\mathfrak{a}(R) = 0$ *and* $C \equiv *_n$ *for some* $n \in \mathbb{N}$;

   (b) *If* $\Gamma \vdash x : C$ *then there are* B, s *such that* $\Gamma \vdash B : s$, $x{:}B \in \Gamma$, *and either* $B =_\beta C$, *or there are* m, n *with* $m < n$ *and* $B \equiv *_m$, $C \equiv *_n$;

   (c) *If* $\Gamma \vdash (\Pi x{:}A.B){:}C$ *then there are* s_1, s_2, s_3 *such that* $(s_1, s_2, s_3) \in \boldsymbol{R}$, $\Gamma \vdash A : s_1$, *and* $\Gamma, x{:}A \vdash B : s_2$. *Moreover,* $C \equiv s_3$ *or there are* m, n *such that* $m < n$, $s_3 \equiv *_m$ *and* $C \equiv *_n$;

   (d) *If* $\Gamma \vdash (\lambda x{:}A.b) : C$ *then there are* B, s *such that* $\Gamma \vdash (\Pi x{:}A.B) : s$; $\Gamma, x{:}A \vdash b : B$ *and* $C =_\beta \Pi x{:}A.B$;

   (e) *If* $\Gamma \vdash (AB) : C$ *then there are* x, P, Q *such that* $\Gamma \vdash A : (\Pi x{:}P.Q)$, $\Gamma \vdash B : P$, *and either* $C =_\beta Q[x{:=}B]$ *or there are* m, n *with* $m < n$ *and* $Q[x{:=}B] \equiv *_m$, $C \equiv *_n$;

6. **(Correctness of Types)** *If* $\Gamma \vdash A : B$ *then there is* $s \in \boldsymbol{S}$ *such that* $\Gamma \vdash B : s$ *or* $B \equiv s$;

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

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

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

Theorem 4.11 (Strong Normalisation for λRTT) *If* $\Gamma \vdash A : B$ *then* A *is strongly normalising.*

PROOF: We embed λRTT into system λC of the Barendregt cube by mapping $*_n$ to $*$ (for all $n \in \mathbb{N}$), $*_s$ to $*$, and $\Box_n$ to $\Box$. In this way, λRTT becomes a pure type system (as rule (Incl) disappears) that is a subsystem of λC. As all terms of λC are strongly normalising, λRTT is strongly normalising as well. ⊠

Remark 4.12 We provide some intuition for the properties in Theorems 4.10 and 4.11.

1. The Church-Rosser theorem is a basic theorem on λ-calculus. It indicates that it does not matter in which way one makes a calculation (list of β-reductions): The result will always be the same (or, more precisely: The results can be *coerced* to be the same). The various proofs that are known even present a constructive method to find a common reduct C of two β-equal terms B_1 and B_2;

2. The first part of the Free Variables Lemma is comparable to Lemma 2.56.1 of RTT.

   The second part has no counterpart in RTT. This is because in RTT the set of types is determined by a separate definition (2.37), and types do not have a type themselves. In λRTT, the derivation rules determine which types are "allowed", and which types are not: An allowed type has a sort as type. The second part of the Free Variables Lemma shows that types that occur in a context are, indeed, always typable by a sort;

3. The Substitution Lemma can be compared to the Substitution Rule of RTT (see 2.45.6). Observe that the substitution $[x{:=}D]$ is now not only carried out in B (as was the case in RTT), but also in C and Δ. This is due to the fact that in λRTT, types may have free variables;

4. The Thinning Lemma is comparable to the Weakening Rule of RTT (see 2.45.5);

5. The Generation Lemma (called "Stripping lemma" in [54]) is one of the most important meta-properties of a PTS. The derivation rules

of λRTT are, as is the case with most usual formulations of PTSs, not syntax directed, i.e. the last rule in a derivation is not necessarily determined by the structure of the term and the context of the conclusion of the derivation.[5] This is due to rules like (Weak), (Conv) and (Incl). If the conclusion of the derivation is $\Gamma \vdash A : C$, then the Generation Lemma provides information on the type of the subterms of A, and on the structure of C.

The Generation Lemma for λRTT is comparable to Theorems 2.84 and 2.85 for RTT.

Case (a) of the Generation Lemma might raise the question why a relation symbol R of arity > 0 cannot have type $\iota \to \cdots \to \iota \to *_n$ for $n > 1$, while a relation symbol R' of arity 0 can have type $*_n$ for $n > 1$. This has to do with the way in which we implemented type inclusion. We only declared that $*_n$ is a subtype of $*_{n+1}$ for all n, but did not extend this to types of the form $\tau_1 \to \tau_2$. It is quite possible to work with type systems that have an extensive subtyping relation (see for instance [33]), but as we do not need an extension of subtyping in this Section, we do not introduce it here;

6. Correctness of Types shows that every term B for which there are Γ, A such that $\Gamma \vdash A : B$ is typable by a sort. Compare this to the second part of the Free Variable Lemma, that proves a similar thing for types occurring in a context;

7. Subject Reduction shows that the type of an expression does not change during a calculation. As there is no real reduction in RTT, we do not have an equivalent statement in RTT. One could see the fact that the Free Variable property 2.58 is maintained under substitution as a weak form of Subject Reduction;

8. Permutation is closely related to the Permutation Rule 2.45.7;

9. This lemma shows that the topsorts (see Definition A.34. In λRTT, the topsorts are $*_s$ and $\square_1, \square_2, \ldots$.) are only inhabited by types (i.e. constants or terms of the form $\Pi x{:}A.B$).

[5]It is possible to present PTSs in a syntax directed way. See Severi's thesis [113].

Strong Normalisation for λRTT can be compared to the theorem on Existence of Substitution for RTT, 2.73. However, in λRTT much more reductions are possible. For instance, the proof terms of λRTT do not have an equivalent in RTT, and these proof terms are also λ-terms that might β-reduce.

4b4 Interpreting RTT in λRTT

In this section we formally prove our claim that λRTT indeed is a PAT interpretation of RTT. We translate the ramified types of RTT to types of λRTT:

Definition 4.13 We define a type $T(t^a)$ for each ramified type t^a:

- $T(0^0) \equiv \iota$;
- $T((t_1^{a_1}, \ldots, t_m^{a_m})^a) \equiv T(t_1^{a_1}) \to \cdots \to T(t_m^{a_m}) \to *_{\max(a,1)}$.

All the translations of the ramified types are typable in λRTT. As announced in Subsection 4b1.1, we use the Π-formation rules $(*_s, \Box_n, \Box_n)$ and $(\Box_m, \Box_n, \Box_n)$ for $m < n$ and $n \geq 1$.

Lemma 4.14 *In* λRTT *we can derive:*

- $\vdash T(0^0) : *_s$;
- $\vdash T((t_1^{a_1}, \ldots, t_m^{a_m})^a) : \Box_a$.

PROOF: Induction on the definition of T; use the rules $(*_s, \Box_n, \Box_n)$ and $(\Box_m, \Box_n, \Box_n)$ of λRTT as sketched in Subsection 4b1.1. ⊠

On the other hand: The inhabitants of $*_s$ and $\Box_n$ are all translations of ramified types:

Lemma 4.15

1. *If* $\Gamma \vdash A : *_s$ *then* $A \equiv \iota$;
2. *If* $\Gamma \vdash A : \Box_n$ *then there is a ramified type* t^n *with* $T(t^n) \equiv A$.

PROOF:

1. As $*_s$ is a topsort, A cannot be a variable, or of the form A_1A_2 or $\lambda x{:}A_1.A_2$ (see Theorem 4.10.9). As there are no $s_1, s_2 \in \boldsymbol{S}$ for which $(s_1, s_2, *_s) \in \boldsymbol{R}$, A cannot be of the form $\Pi x{:}A_1.A_2$. Therefore, A must be a constant. By the Generation Lemma, 4.10.5(a), $A \equiv \iota$;
2. Use induction on the length of A. As $\Box_n$ is a topsort, A cannot be a variable, a λ-abstraction, or an application. Considering the Generation Lemma, 4.10.5, we conclude that $A \equiv *_n$ (and in that case we are done: Take $t^n \equiv ()^n$) or $A \equiv \Pi x{:}B_1.B_2$.

   In the last case, we use the Generation Lemma to obtain: $\Gamma \vdash B_1 : s_1$ and $\Gamma, x{:}B_1 \vdash B_2{:}s_2$, where $(s_1, s_2, \Box_n) \in \boldsymbol{R}$.

   Due to the definition of $\boldsymbol{R}$, $s_2 \equiv \Box_n$. Hence $\Gamma, x{:}B_1 \vdash B_2{:}\Box_n$. Using the induction hypothesis, we can find $u_2^{a_2}, \ldots, u_m^{a_m}$ such that $B_2 \equiv T((u_2^{a_2}, \ldots, u_m^{a_m})^n)$ (it is not the case that $B_2 \equiv T(0^0)$, otherwise we would have $\Gamma, x{:}B_1 \vdash \iota{:}\Box_n$).

   Due to the definition of $\boldsymbol{R}$ we also have $s_1 \equiv *_s$ or $s_1 \equiv \Box_{a_1}$ for some a_1. By induction, $B_1 \equiv \iota$, or $B_1 \equiv T(u_1^{a_1})$ for a ramified type $u_1^{a_1}$.

   Hence $A \equiv T((\iota, u_2^{a_2}, \ldots, u_m^{a_m})^n)$ or $A \equiv T((u_1^{a_1}, \ldots, u_m^{a_m})^n)$ (notice that $a_1 < n$, as $s_1 \equiv \Box_{a_1}$ and $(s_1, s_2, \Box_n) \in \boldsymbol{R}$).

⊠

We extend the mapping T to propositional functions:

Definition 4.16 Let Γ be a RTT-context. We define a term $T(i)$ for all $i \in \mathcal{V} \cup \mathcal{A}$, and a term $T(f)$ for each $f \in \mathcal{P}$ for which the variables of f are contained in DOM(Γ):

- $T(x) \equiv x$ for $x \in \mathcal{V}$;
- $T(a) \equiv a$ for $a \in \mathcal{A}$;
- $T(R(i_1, \ldots, i_{\mathfrak{a}(R)})) \equiv \lambda x_1{:}T(t_1) \ldots \lambda x_n{:}T(t_n).R(T(i_1)) \cdots (T(i_{\mathfrak{a}(R)}))$.

  Here, $x_1 < \cdots < x_n$ are the free variables of $R(i_1, \ldots, i_{\mathfrak{a}(R)})$, and $x_i{:}t_i \in \Gamma$ for $1 \leq i \leq n$;

- If $f_1, f_2 \in \mathcal{P}$ and $T(f_j) \equiv \lambda x_{j1}{:}A_{j1} \dots \lambda x_{jm}{:}A_{jm}.F_j$, where $x_{j1} < \dots < x_{jm}$ are the free variables of f_j, then define $T(f_1 \to f_2) \equiv$

  $$\lambda x_1{:}T(t_1) \dots \lambda x_n{:}T(t_n).(F_1 \to F_2).$$

  Here, $x_1 < \dots < x_n$ are the free variables of $f_1 \to f_2$, and $x_i{:}t_i \in \Gamma$ for $1 \leq i \leq n$;

- If $f \in \mathcal{P}$ and $T(f) \equiv \lambda x_1{:}A_1 \dots \lambda x_n{:}A_n.F$, where $x_1 < \dots < x_k < \dots < x_n$ are the free variables of f, then define $T(\forall x_k{:}t_k[f]) \equiv$

  $$\lambda x_1{:}A_1 \dots \lambda x_{k-1}{:}A_{k-1} \lambda x_{k+1}{:}A_{k+1} \dots \lambda x_n{:}A_n.\Pi x_k{:}T(t_k).F;$$

- If $z \in \mathcal{V}$ and $k_1, \dots, k_m \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, then define $T(z(k_1, \dots, k_m)) \equiv$

  $$\lambda x_1{:}T(t_1) \dots \lambda x_n{:}T(t_n).z(T(k_1)) \cdots (T(k_m)),$$

  where $x_1 < \dots < x_n$ are the free variables of $z(k_1, \dots, k_m)$ and $x_i{:}t_i \in \Gamma$.

We show that the legal pfs of RTT are legal terms in λRTT. For one step in the proof, we need a Lemma:

Lemma 4.17 *If* $\Gamma \vdash A : \Box_n$ *or* $\Gamma \vdash A : *_s$ *then* $\text{FV}(A) = \varnothing$.

PROOF: By Lemma 4.15, there is a ramified type t^a such that $A \equiv T(t^a)$. From the definition of $T(t^a)$ we conclude that $\text{FV}(T(t^a)) = \varnothing$. ⊠

Lemma 4.18 *Let* $f \in \mathcal{P}$ *and assume* $\Gamma \vdash f : t^a$ *in* RTT. *Then* $\vdash T(f) : T(t^a)$ *in* λRTT.

PROOF: Induction on the structure of f. Though the proof is rather straightforward, we treat all cases, in order to show where which rules of λRTT are used.

1. $f \equiv R(i_1, \dots, i_{\mathfrak{a}(R)})$, and $x_1 < \dots < x_m$ are the free variables of f. Notice: $x_i{:}0^0 \in \Gamma$ for $1 \leq i \leq m$ (Theorem 2.85). Therefore: $T(f) \equiv \lambda x_1{:}\iota \dots \lambda x_m{:}\iota.Ri_1 \cdots i_{\mathfrak{a}(R)}$. Notice that $x_1{:}\iota, \dots, x_m{:}\iota \vdash Ri_1 \cdots i_{\mathfrak{a}(R)} : *_1$. By abstraction, $\vdash T(f) : \underbrace{\iota \to \cdots \to \iota}_{\mathfrak{a}(R) \text{ times } \iota} \to *_1$;

2. $f \equiv f_1 \to f_2$; $x_1 < \cdots < x_m$ are the free variables of f, and $x_{j1} < \cdots < x_{jm_j}$ are the free variables of f_j. Assume $x_i{:}t_i^{a_i} \in \Gamma$ such that $x_{ji}{:}t_{ji}^{a_{ji}}$. By Theorem 2.84, the f_j are legal, and by Theorem 2.58, $\Gamma \vdash f_j : \left(t_{j1}^{a_{j1}}, \ldots, t_{jm_j}^{a_{jm_j}}\right)^{b_j}$ for some b_j. Observe that we can write (due to Definition 4.16)
$$T(f_j) \equiv \lambda x_{j1}{:}T\left(t_{j1}^{a_{j1}}\right) \ldots \lambda x_{jm_j}{:}T\left(t_{jm_j}^{a_{jm_j}}\right).F_j,$$
and that, by the induction hypothesis,
$$\vdash T(f_j) : T\left(t_{j1}^{a_{j1}}\right) \to \cdots \to T\left(t_{jm_j}^{a_{jm_j}}\right) \to *_{b_j}.$$
This means (Generation Lemma):
$$x_{j1}{:}T\left(t_{j1}^{a_{j1}}\right), \ldots, x_{jm_j}{:}T\left(t_{jm_j}^{a_{jm_j}}\right) \vdash F_j : *_{b_j},$$
and therefore (Weakening):
$$x_1{:}T\left(t_1^{a_1}\right), \ldots, x_m{:}T\left(t_m^{a_m}\right) \vdash F_1 : *_{b_1},$$
and
$$x_1{:}T\left(t_1^{a_1}\right), \ldots, x_m{:}T\left(t_m^{a_m}\right), x{:}F_1 \vdash F_2 : *_{b_2}$$
(notice that $\{x_1, \ldots, x_m\} \supseteq \{x_{j1}, \ldots, x_{jm_j}\}$).
With rule $(*_{b_1}, *_{b_2}, *_{\max(b_1,b_2)})$,
$$x_1{:}T\left(t_1^{a_1}\right), \ldots, x_m{:}T\left(t_m^{a_m}\right) \vdash F_1 \to F_2 : *_{\max(b_1,b_2)}.$$
Now notice that $a = \max(b_1, b_2)$, and use λ-abstraction m times:
$$\vdash T(f) : T(t^a);$$

3. $f \equiv \forall x{:}u^b[f']$, $x_1 < \cdots < x_m$ are the free variables of f, and $x_i{:}t_i^{a_i} \in \Gamma$. For simplicity of notation, we will assume that $x < x_1$. As $x \in \text{FV}(f')$, we can write $T(f') \equiv \lambda x{:}T(u^b).\lambda x_1{:}T(t_1^{a_1}) \ldots \lambda x_m{:}T(t_m^{a_m}).F'$. By Theorem 2.84 and Lemma 2.56, we have that f' is legal in $\Gamma \cup \{x{:}u^b\}$, and by Theorem 2.58 and Corollary 2.61: $\Gamma \cup \{x{:}u^b\} \vdash f' : (u^b, t_1^{a_1}, \ldots, t_m^{a_m})^a$. By the induction hypothesis,
$$\vdash T(f') : T(u^b) \to T(t_1^{a_1}) \to \cdots \to T(t_m^{a_m}) \to *_a.$$

Therefore (Generation Lemma),

$$x{:}T(u^b), x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash F' : *_a,$$

so by the permutation lemma and Lemma 4.17

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}), x{:}T(u^b) \vdash F' : *_a.$$

As $T(u^b)$ has either type $\Box_b$ or type $*_s$ (Lemma 4.14), and $b < a$, we can use rule $(\Box_b, *_a, *_a)$ or $(*_s, *_a, *_a)$ to derive

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash (\Pi x{:}T(u^b).F') : *_a.$$

By λ-abstraction, we find $\vdash T(f) : T(t^a)$;

4. $f \equiv z(k_1, \ldots, k_n)$, $x_1 < \cdots < x_m$ are the free variables of f, and $x_i{:}t_i^{a_i} \in \Gamma$. By Theorem 2.84, the k_j are either legal pfs of predicative type in Γ, or variables (so one of the x_is), or individuals (of type 0^0). Let $u_j^{b_j}$ be the type of k_j in Γ. By Theorem 2.85: $z{:}(u_1^{b_1}, \ldots, u_n^{b_n})^{a-1} \in \Gamma$. Using the induction hypothesis for the $k_j \in \mathcal{P}$, we have that

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash T(k_j) : T(u^{b_j})$$

for $1 \leq j \leq n$. We also have

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash z : T(u_1^{b_1}) \to \ldots \to T(u_n^{b_n}) \to *_{a-1}.$$

Therefore,

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash z(T(k_1)) \cdots (T(k_n)) : *_{a-1},$$

hence (by the (Incl) rule)

$$x_1{:}T(t_1^{a_1}), \ldots, x_m{:}T(t_m^{a_m}) \vdash z(T(k_1)) \cdots (T(k_n)) : *_a.$$

By λ-abstraction, $\vdash T(f) : T(t^a)$.

⊠

Remark 4.19 The use of the (Incl) rule in case 4 of the above proof is essential. There, it is shown that

$$x_1{:}T(t_1^{a_1}),\ldots,x_m{:}T(t_m^{a_m}) \vdash z : T(u_1^{b_1}) \to \ldots \to T(u_n^{b_n}) \to *_{a-1},$$

and one could try to form λ-abstractions without using the (Incl) rule first. However, $z \in \mathrm{FV}(f)$, so at a certain point one has to construct a λ-abstraction over z. Let t_z be such that $z : t_z \in \Gamma$. The resulting term $\lambda z{:}T(t_z).G$ has type $T(t_z) \to \ldots \to *_{a-1}$. As z has order $a-1$, $T(t_z)$ is a term of type $\Box_{a-1}$ (Lemma 4.14). One needs a rule $(\Box_{a-1}, \Box_p, \Box_p)$ with $p > a-1$ (as for $p \leq a-1$ such a rule is not present) for the construction of $T(t_z) \to \ldots \to *_{a-1}$. Hence, one has to use the rule (Incl) to replace $*_{a-1}$ by $*_a$, which has type $\Box_a$. This makes it possible to use $p = a$.

4b5 Logic in RTT and λRTT

Before we can use λRTT as a system in which we can prove theorems, we must add some logical axioms to it. These axioms mainly have to do with the symbol $\bot$. For $\to$ and $\forall$ the needed derivation rules are already provided by the type theory (via the PAT principle a la Curry-Howard).

- The $\neg$-introduction rule of natural deduction systems is already incorporated in the translation of $\neg A$ to $A \to \bot$. If we have a proof T of $\bot$ under the assumption that x is a proof of A, then $\lambda x{:}A.T$ is a proof of $A \to \bot$;

- For the rule "ex falso sequitur quodlibet" the type system does not provide a natural equivalent. We therefore introduce an axiom

  $$\mathtt{ExFalso}_n : \Pi \mathtt{f}{:}*_n.\Pi \mathtt{p}{:}\bot.\mathtt{f}$$

  for each $n \in \mathbb{N}^+$. We will store these axioms in some basic context Γ_0.

  We remark that the type $\Pi\mathtt{f}{:}\,*_n\,.\Pi\mathtt{p}{:}\bot.\mathtt{f}$ is indeed a type in λRTT. It is straightforward to derive that it is a type of sort $*_{n+1}$.

  We also remark that it is necessary to introduce separate axioms $\mathtt{ExFalso}_1, \mathtt{ExFalso}_2, \ldots$. If we want to conclude the proposition f

using the ExFalso-axiom, we must provide the type of f, and in that type the order of f is also mentioned. This is a usual thing in ramified type systems, and such constructions occur also in *Principia* (cf. [121], pp. 41–43);

- RTT is based on classical logic, and PAT on intuitionistic logic. Therefore we must add a "classical" axiom. We prefer to add the "law of double negation", and introduce axioms

  $$\mathtt{DblNeg}_n : \Pi \mathtt{f}{:}*_n.\Pi \mathtt{p}{:}(\mathtt{f}\to\bot)\to\bot.\mathtt{f}.$$

  It is easy to show that the type of this axiom is of sort $*_{n+1}$. We store the axioms $\mathtt{DblNeg}_n$ in the same context Γ_0.

We compare the obtained system with the original logical system that was proposed in *Principia Mathematica.* That system is presented in what we would now call a natural deduction style. It has one derivation rule, modus ponens (cf. *Principia*, ∗1·1), and the following axioms:

$$(p \vee p) \to p \qquad (*1{\cdot}2);$$

$$q \to (p \vee q) \qquad (*1{\cdot}3);$$

$$(p \vee q) \to (q \vee p) \qquad (*1{\cdot}4);$$

$$(p \vee (q \vee r)) \to (q \vee (p \vee r)) \qquad (*1{\cdot}5);$$

$$(q \to r) \to ((p \vee q) \to (p \vee r)) \qquad (*1{\cdot}6);$$

$$f(x) \to \exists z[f(z)] \qquad (*9{\cdot}1);$$

$$f(x) \vee f(y) \to \exists z[f(z)] \qquad (*9{\cdot}11);$$

In any assertion containing a free variable, this free variable may be turned into an apparent variable of which all possible values are asserted to satisfy the function in question (∗9·13).

The formulation of the last axiom is not as precise as the other ones. In later formulations of logic (as the ones by Gödel [57] and Church [30]) we see that the axioms for propositional logic are mostly maintained, but that the axioms on predicate logic are replaced by two other ones:

$$\forall x[f] \to f[x{:=}a]$$

$$\forall x[f \vee g] \to f \vee \forall x[g]$$

where we assume that $x \notin \mathrm{FV}(f)$ and that a does not contain any variable that is bound in f at a place where x is free in f. These new axioms are theorems in the *Principia* (∗9·2 and ∗9·25), and Russell's axioms are proved in Church's system [30].

We must take into account that Gödel and Church both use simple type theory instead of ramified type theory. But Russell's system, by accepting the axiom of reducibility, is in fact also based on simple type theory.

Clearly, λRTT has also modus ponens (function application). We now show that all the axioms of Russell's system can be derived in λRTT as well:

Theorem 4.20 *In* λRTT *with the axioms* $\mathtt{ExFalso}_n$ *and* $\mathtt{DblNeg}_n$*, one can construct terms of the following types:*

$$T(\forall \mathrm{p}{:}*_k[(\mathrm{p} \vee \mathrm{p}) \to \mathrm{p}]) \qquad (*1{\cdot}2);$$

$$T(\forall \mathrm{p}{:}*_k \forall \mathrm{q}{:}*_m[\mathrm{q} \to (\mathrm{p} \vee \mathrm{q})]) \qquad (*1{\cdot}3);$$

$$T(\forall \mathrm{p}{:}*_k \forall \mathrm{q}{:}*_m[(\mathrm{p} \vee \mathrm{q}) \to (\mathrm{q} \vee \mathrm{p})]) \qquad (*1{\cdot}4);$$

$$T(\forall \mathrm{p}{:}*_k \forall \mathrm{q}{:}*_m \forall \mathrm{r}{:}*_n[(\mathrm{p} \vee (\mathrm{q} \vee \mathrm{r})) \to (\mathrm{q} \vee (\mathrm{p} \vee \mathrm{r}))]) \qquad (*1{\cdot}5);$$

$$T(\forall \mathrm{p}{:}*_k \forall \mathrm{q}{:}*_m \forall \mathrm{r}{:}*_n[(\mathrm{q} \to \mathrm{r}) \to ((\mathrm{p} \vee \mathrm{q}) \to (\mathrm{p} \vee \mathrm{r}))]) \qquad (*1{\cdot}6);$$

$$T(\forall \mathrm{f}{:}\iota \to *_m \forall \mathrm{x}{:}\iota[f(x) \to \exists z{:}\iota[f(z)]]) \qquad (*9{\cdot}1);$$

$$T(\forall \mathrm{f}{:}\iota \to *_m \forall \mathrm{x}{:}\iota \forall \mathrm{y}{:}\iota[(f(x) \vee f(y) \to \exists z{:}\iota[f(z)]]) \qquad (*9{\cdot}11).$$

PROOF: The following terms are inhabitants of the types above:

$$\lambda \mathtt{p}{:}*_k.\lambda \mathtt{x}{:}(\mathtt{p} \to \bot) \to \mathtt{p}.\mathtt{DblNeg}_k\mathtt{p}(\lambda \mathtt{y}{:}\mathtt{p} \to \bot.\mathtt{y}(\mathtt{xy})) \qquad (*1{\cdot}2);$$

$$\lambda \mathtt{p}{:}*_k.\lambda \mathtt{q}{:}*_m.\lambda \mathtt{y}{:}\mathtt{q}.\lambda \mathtt{x}{:}\mathtt{p} \to \bot.\mathtt{y} \qquad (*1{\cdot}3);$$

$$\begin{array}{l}\lambda \mathtt{p}{:}*_k.\lambda \mathtt{q}{:}*_m.\lambda \mathtt{x}{:}(\mathtt{p} \to \bot) \to \mathtt{q}.\\ \quad \lambda \mathtt{y}{:}\mathtt{q} \to \bot.\mathtt{DblNeg}_k p(\lambda \mathtt{z}{:}\mathtt{p} \to \bot.\mathtt{y}(\mathtt{xz}))\end{array} \qquad (*1{\cdot}4);$$

$$\begin{array}{l}\lambda \mathtt{p}{:}*_k.\lambda \mathtt{q}{:}*_m.\lambda \mathtt{r}{:}*_n.\lambda \mathtt{x}{:}(\mathtt{p} \to \bot) \to ((\mathtt{q} \to \bot) \to \mathtt{r}).\\ \quad \lambda \mathtt{y}{:}\mathtt{q} \to \bot.\lambda \mathtt{z}{:}\mathtt{p} \to \bot.\mathtt{xzy}\end{array} \qquad (*1{\cdot}5);$$

$$\begin{array}{l}\lambda \mathtt{p}{:}*_k.\lambda \mathtt{q}{:}*_m.\lambda \mathtt{r}{:}*_n.\lambda \mathtt{x}{:}\mathtt{q} \to \mathtt{r}.\\ \quad \lambda \mathtt{y}{:}(\mathtt{p} \to \bot) \to \mathtt{q}.\lambda \mathtt{z}{:}\mathtt{p} \to \bot.\mathtt{x}(\mathtt{yz})\end{array} \qquad (*1{\cdot}6);$$

$$\lambda \mathtt{x}{:}\iota.\lambda \mathtt{f}{:}\iota \to *_k.\lambda \mathtt{p}{:}\mathtt{fx}.\lambda \mathtt{q}{:}(\Pi \mathtt{y}{:}\iota.\mathtt{fy}\to\bot).\mathtt{qxp} \qquad (*9{\cdot}1);$$

$$\begin{array}{l}\lambda \mathtt{x}{:}\iota.\lambda \mathtt{y}{:}\iota.\lambda \mathtt{f}{:}\iota \to *_k.\lambda \mathtt{p}{:}(\mathtt{fx} \to \iota) \to \mathtt{fy}.\\ \quad \lambda \mathtt{q}{:}\Pi \mathtt{z}{:}\iota.\mathtt{fz} \to \bot.\mathtt{qy}(\mathtt{p}(\mathtt{qx}))\end{array} \qquad (*9{\cdot}11).$$

⊠

The last axiom of Russell's logical system is implemented in λRTT by the Π-elimination rule.

We conclude that the embedding T is sound with respect to the logics that are used in RTT and λRTT.

4b6 Various implementations of PAT

As was explained in Definition 4.8, types and terms are mixed up in one system λRTT. As a consequence we can spot a hierarchy of *levels* in λRTT. The hierarchies in RTT and λRTT are depicted in Figures 4 and 5. Some of the levels in these figures are empty. Later, we will compare the systems RTT and λRTT with other systems. For this comparison, we will draw similar pictures, in which we need the levels that are empty in the presentation of RTT and λRTT.

We have the level of *propositional types*: This is formed by the terms that have a type of the form $\square_n$. Below the level of propositional types we find *propositions and propositional functions*. These have a propositional type as type. Under the level of propositions and propositional functions we find the level of *proofs*. A proof has always a proposition as its type.

	Ramified Types	
0^0	$()^2$	$(0^0)^1$
Objects	Propositions	Prop. functions
a	$\forall x{:}()^1[x() \to x()]$	R(x)

Figure 4: Levels within RTT

	Topsorts	
$*_s$	$\Box_2$	$\Box_1$
	Ramified Types	
ι	$*_2$	$\iota \to *_1$
Objects	Propositions	Prop. functions
a	$\Pi x{:}*_1.\Pi y{:}x.x$	$\lambda x{:}\iota.Rx$
	Proofs	
	$\lambda x{:}*_1.\lambda y{:}x.y$	

Figure 5: Levels within λRTT

There is a second hierarchy of levels in λRTT. At its top we find $*_s$, the type of individual types. It has as its only inhabitant ι, the type of individuals. One could imagine situations in which $*_s$ has more inhabitants. For instance, if we would allow several sets of individuals. Or if the Π-formation rule $(*_s, *_s, *_s)$ is allowed, so that also types like $\iota \to \iota$ can be constructed. Below the type of individuals, we find the individuals themselves.

We see that the transformation of RTT to λRTT has introduced some new term levels:

- A level of *topsorts*. These topsorts are needed to type the ramified types. Such typing is needed for two reasons:

  Variable introduction If we want to introduce a variable of a certain type τ in a PTS, we have to establish that τ is an allowed type. This is done by requiring that τ itself must have a certain type;

  Type construction To control the construction of types with the Π-formation rules, Π-formation is only allowed with certain types. This is determined by the type of that type. The ramified types of RTT however, do not have a type in RTT. We use the topsorts $\Box_1, \Box_2, \ldots$ to type the translations of these ramified types in λRTT. This also gives us a good way to check the order of a type: A type of order n has type $\Box_n$;

- A level of *proofs*.This level was empty in RTT, as proofs are not part of the theory of RTT.

From our PTS-point of view it is remarkable that sorts of RTT which are denoted by the symbol $*$, are not all at the same level. The sort $*_s$ occurs at the level of topsorts, while the sorts $*_n$ live at the level of the ramified types. Moreover, we have already seen that each Π-formation rule of the form $(\Box_n, s_1, s_2)$ also has a variant of the form $(*_s, s_1, s_2)$, and vice versa. With this in mind it would have been more clear to write $\Box_s$ instead of $*_s$, and use $*_s$ for ι.

The reason that we chose the symbol $*_s$ (instead of $\Box_s$) has to do with traditions within the discipline of Pure Type Systems: The levels are

	Logical Topsorts	
	$\Box_2$	$\Box_1$
Object Topsorts $*_s$	Logical Ramified Types $*_2$	$\iota \longrightarrow *_1$
Object Types ι	Propositions $\Pi x{:}*_1.\Pi y{:}x.x$	Prop. functions $\lambda x{:}\iota.Rx$
Objects a	Proofs $\lambda x{:}*_1.\lambda y{:}x.y$	

Figure 6: Levels of λRTT in PTS tradition

Obj. Tops. $*_s$	Proof Topsorts $*_2$	Logical Topsorts $\Box_2$	$\Box_1$
Obj. Types ι	Proof Types `true`$(\forall x{:}()^1.x \rightarrow x)$	Logical Ramified Types $()^2$	$(0^0)^1$
Objects a	Proofs $\lambda x{:}\ *_1\ .\lambda y{:}x.y$	Propositions $\forall x{:}()^1.x \rightarrow x$	Prop. functions R(x)

Figure 7: Levels of λRTT in bool-style PAT

usually partitioned in such a manner that $*_s$ and $*_n$ live at the same level. See Figure 6.

Let's have a closer look at the traditional situation. The PAT principle within PTSs is often implemented by lifting the propositions and propositional functions from term level to type level, but leaving the individuals at term level. We see that proofs are not introduced at a new level below term level, but that the type level (as far as propositions and propositional functions are concerned) is lifted, and that the proofs are put at the term level that was originally occupied by the propositions.

The treatment of propositions at a higher level than individuals can be understood if we take a look at first-order logic. In systems for first-order logic, quantification over individuals is possible, but quantification over propositions and propositional functions is not allowed. This leads to the treatment of propositions and propositional functions at a higher level.

Contrary to the PTS tradition, we did not lift propositions from term level to type level when we constructed a PAT implementation for RTT. Instead, we built a new level below the level of propositional functions, propositions and individuals: The level of proofs. In this way the double role of propositions is more clear:

- They are terms, as they live at the same level as the individuals;

- They are types, as they can have inhabitants (their proofs).

The PAT implementation à la De Bruijn can in various ways be seen as a compromise between the two different points of view above (though it has been developed independently). A PAT implementation of RTT à la De Bruijn could be depicted as in Figure 7. There are three hierarchies now: The two well-known hierarchies of objects and propositions/propositional functions, plus a new hierarchy for proofs. The hierarchies of propositional functions and proofs are connected via the operator `true` (see Section 4a4), which assigns a type of proofs to each proposition. This picture:

- Respects the wish to treat propositions at term level;

- Respects the wish to treat proof classes as types;

- Can also be seen, in retrospect, as a compromise from a historical point of view. Though AUTOMATH and the PAT notion in De Bruijn style are mainly independent of other developments in logic and type theory, the bool-style PAT notion (1968) historically fits between the style of Figure 5 for the Ramified Theory of Types (1908-1912) and the style of Pure Type Systems in Figure 6 (1988).

4c STT in PAT style

From the description of RTT in PAT style it is easy to make a description λSTT of STT in PAT style: Simply remove all references to orders. This means that $*_n$ has to be replaced by $*$, and $\square_n$ by $\square$ (for all $n \in \mathbb{N}$). In fact, the same procedure is followed by Ramsey [101], Gödel [57] and Church [30] in their presentations of simple type theory.

One of the consequences is that rule (Incl) disappears. We obtain a pure type system with axiom $*{:}\square$ and rules $(*_s, \square, \square), (\square, \square, \square), (*_s, *, *), (*, *, *)$ and $(\square, *, *)$. This looks familiar to the Calculus of Constructions λC. In λC there is the same axiom $*{:}\square$, and rules $(*, *, *), (*, \square, \square), (\square, *, *), (\square, \square, \square)$. But λSTT is more restricted than λC. More specifically, rule $(*_s, \square, \square)$ is not as powerful as rule $(*, \square, \square)$ in λC. As in λC, we do have higher order logic, but we do not have the higher order functions that are present in λC. This is due to the fact that $*_s$ is a topsort in λSTT, while $*$ has type $\square$. Therefore, λSTT is a system somewhere in between $\lambda\omega$ and λC.

Notice that we have given a PAT version of the simple theory of types, and not of the simply typed λ-calculus of Church [30]. In [30] there are more things formalised than in STT:

- In the simply typed λ-calculus there are more types. For instance, $\iota \to \iota$ is a type, and so is $* \to \iota$ (in [30] this type is denoted $o \to \iota$). More precise: For a PAT implementation of Church's theory we should add a rule $(*_s, *_s, *_s)$, and a rule $(\square, *_s, *_s)$;
- Church has an additional logical operator ($\imath$) in his system. This operator also occurs in Russell's RTT, but only as an abbreviation and not as a new syntactical object (see [121], pp. 66–68 and pp. 173–175).

Remark 4.21 Together, the rules $(*_s, \square, \square)$ and $(\square, \square, \square)$ form a version of the simply typed lambda calculus of Church. The identification would have been complete if we had identified $*_s$ with $\square$.

Conclusions

We saw that there are various ways in which PAT can be implemented in type theories. There are two main streams:

Curry-Howard approach: This approach treats propositions as types, and a proof of a proposition is an inhabitant of the type that represents that proposition. The implementation is based on the Brouwer-Heyting-Kolmogorov interpretation of the logical connectives. In particular, a proof of an implication $A \to B$ is represented as a function that transforms proofs of the proposition A (terms of the type A) to proofs of B (terms of type B);

De Bruijn approach: For each proposition P we create a type $\texttt{bool}(P)$. A proof of P in this approach is not a term of type P (as in the Curry-Howard style), but a term of type $\texttt{bool}(P)$.

In Curry-Howard style implementations, logic is already part of the system. The logical connective $\to$ and the quantifier $\forall$ immediately translate to the construction of function types. Using higher-order logic, other logical connectives can be defined in terms of $\to$ and $\forall$. De Bruijn style implementations have more possibilities. One can implement the logical system independent from the type system. But it is also possible to use function types for the translation of $\to$ and/or $\forall$ as is done in the Curry-Howard style.

The various implementations lead to various levels in type systems. This was depicted in figures 6 and 7. In Curry-Howard style and the PTS tradition, propositions are at the same level as types, and therefore, proofs are at the same level as objects (terms). In De Bruijn style (bool-style), proofs, propositions, and propositional functions all live at term level.

A third division into levels appeared when we gave a description of RTT in PAT-style. See Figure 5. On the one hand, λRTT uses a Curry-Howard style implementation. There is no difference between a proposition and the type of its proofs. Therefore, proofs and propositions do not live at the same level (as is the case in PAT a la De Bruijn). On the other hand, objects and propositions live at the same level.

The implementation of RTT in PAT-style not only serves as an elaborate example of PAT, but also shows that ramified types can be placed in the framework of Pure Type Systems without too many problems.

Chapter 5

Automath

The first practical use of the propositions-as-types principle sketched in Chapter 4 is found in the AUTOMATH project [95]. The AUTOMATH systems are the first examples of proof checkers, and in this way they are predecessors of modern proof checkers like Coq [42] and Nuprl [34].

The project was started in 1967 by N.G. de Bruijn, and

> "it was not just meant as a technical system for verification of mathematical texts, it was rather a life style with its attitudes towards understanding, developing and teaching mathematics."
>
> ([23]; see [95] p. 201)

Thus, the roots of AUTOMATH are not to be found in logic or type theory, but in mathematics and the mathematical vernacular [22]. This is also clearly reflected in the goals of the AUTOMATH project:

> "1. The system should be able to verify entire mathematical theories.
>
> 2. The system should remain very general, tied as little as possible to any particular set of rules for logic and foundations of mathematics. Such basic rules should preferably belong to material that can be presented for verification, on the same level with things like mathematical axioms that have

to be explained to the reader.[1]

3. The way mathematical material is to be presented to the system should correspond to the usual way we write mathematics. The only things to be added should be details that are usually omitted in standard mathematics."

([23]; see [95] pp. 209–210)

Goal 1 was definitely achieved: Van Benthem Jutting translated and verified Landau's "Grundlagen der Analysis" [83] in AUTOMATH (see [9], [10]) and Zucker formalised classical real analysis in AUTOMATH (see [124]).

A consequence of goal 2 has already been discussed in Section 4a4. There, we saw that de Bruijn used a PAT principle that was somewhat different from Curry and Howard's. Curry and Howard identified the logical implication and the universal quantifier with function types, following Heyting's intuitionistic interpretation of logical connectives. In doing so, they do not leave a possibility for a different interpretation of implication and universal quantification. Using PAT in de Bruijn's style, the rules for manipulating the logical connectives always have to be made explicit by the user (an example of such a specification can be found in Section 12 and 13 of [11]). This makes it possible to give interpretations of logical connectives that are not based on interpreting implication and universal quantification by a function type.

De Bruijn has spent a lot of effort in achieving goal 3. He has studied the language of mathematics in great depth (see [22]), and many of his insights are reflected in AUTOMATH. We mention some AUTOMATH features that help to achieve goal 3:

- The use of books. Just like a mathematical text, AUTOMATH is written line by line, where each line may refer to definitions or results given in earlier lines;

- The use of definitions. Without definitions, expressions very soon become too long. Moreover, a definition gives a name to a certain

[1]So: the logical rules should be treated on the same level as mathematics. A logical rule can be introduced as an axiom in the same way a mathematical axiom can be introduced. Other logical rules can be derived from existing rules, like mathematical theorems can be derived from existing theorems and axioms. [remark by the author]

expression, and this name makes it easier for the user to remember (or understand) what the use of the definiens is;

- The use of a parameter mechanism together with a default mechanism. We discuss the advantages of these mechanisms in Section 5a.

As AUTOMATH was developed quite independently from other developments in the world of type theory and λ-calculus, there are many things to be explained in the relation between the various AUTOMATH languages and other type theories. In this chapter we focus on the relation between AUTOMATH and Pure Type Systems (PTSs). Both [5] and [54] mention this relation in a few lines, but as far as we know a satisfactory explanation of the relation between AUTOMATH and PTSs is not available. Moreover, both works consider AUTOMATH without one of its most important mechanisms: The definition system. Even the system PAL, which roughly consists of the definition system of AUTOMATH only, is able to express some simple mathematical reasoning (see for instance Section 5 of [21]). Moreover, recent developments on the use of definitions in Pure Type Systems by Bloo, Kamareddine and Nederpelt [17, 16] and Severi and Poll [114] justify renewed research on the relation between AUTOMATH and PTSs. The combination of the work of Severi and Poll [114] and the parameter mechanism of AUTOMATH leads to a white spot in the theory of PTSs, and this spot will be filled up in Chapter 6.

In Section 5a we give a description of AUT-68, which is one of the most elementary AUTOMATH system. In Section 5b we discuss how we can transform AUT-68 into a PTS. In doing so, we must notice that AUT-68 has some properties that are not usual for PTSs:

- AUT-68 has η-reduction;
- AUT-68 has Π-application and Π-reduction (as it does not make any difference between λ and Π);
- AUT-68 has a definition system;
- AUT-68 has a parameter mechanism.

η-reduction is the reduction relation generated by $(\lambda x.Rx) \rightarrow_\eta R$, where $x \notin \mathrm{FV}(R)$. In systems with Π-application, a term $\Pi x{:}A.B$ can be applied to a term N (of type A). This results in $(\Pi x{:}A.B)N$. The usual application rule of Pure Type Systems then changes to

$$\frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : (\Pi x{:}A.B)N}.$$

In such systems, Π behaves like λ, and as a consequence, there also is a rule of Π-reduction

$$(\Pi x{:}A.B)N \to_{\Pi} B[x{:=}N].$$

In AUTOMATH, one does not even make any distinction between the terms $\Pi x{:}A.B$ and $\lambda x{:}A.B$. They are both denoted $[x{:}A]B$. It is not always easy to see whether a term $[x{:}A]B$ represents (in notation of PTSs) $\lambda x{:}A.B$ or $\Pi x{:}A.B$.

We pay more attention to Π-application and Π-reduction at the end of this Chapter; for more details see [17] and the literature on AUTOMATH [95].

We consider η-reduction not as one of the essential features of AUTOMATH, and prefer to focus on the definition and parameter mechanisms, which are the most characteristic type-theoretical features of AUTOMATH.

In Section 5c, we present a system $\lambda 68$ that is (almost) a PTS. We show that it has the usual properties of PTSs and we prove that $\lambda 68$ can be seen as AUT-68 without η-reduction, Π-application and Π-reduction. There is no direct parameter system in $\lambda 68$ either, but this parameter system is hidden in the rules for the construction of product types. In Section 5d we compare the definition system of AUT-68 with several other, more modern, type systems with definitions.

5a Description of AUTOMATH

During the AUTOMATH-project, several AUTOMATH-languages have been developed. They all have two mechanisms for describing mathematics. One of them essentially is a typed λ-calculus, with the important features of λ-abstraction, λ-application and β-reduction. The other mechanism is the use of definitions and parameters. The latter is the same for most AUTOMATH-systems, and the difference between the various systems is mainly caused by different λ-calculi that are included. In this section we will describe the system AUT-68 which not only is one of the first AUTOMATH-systems, but also a system with a relatively simple typed λ-calculus, which makes it easier to focus on the (less known) mechanism for definitions and parameters.

A more extensive description of AUT-68 on which our description below is based, can be found in [11], [20] or [40].

5a1 Books, lines and expressions

In the conception underlying the AUTOMATH-systems, a mathematical text is thought of as being a series of consecutive "clauses". Each clause is expressed in AUTOMATH as a *line*. Lines are stored in so-called *books*. For writing lines and books in AUT-68 we need

- The symbol `type`;
- A set $\mathcal{V}$ of variables;
- A set $\mathcal{C}$ of constants;
- The symbols ( ) [ ] : — , .

We assume that $\mathcal{V}$ and $\mathcal{C}$ are infinite, or at least offer us as many different elements as needed. We also assume that $\mathcal{V} \cap \mathcal{C} = \varnothing$ and that $\mathtt{type} \notin \mathcal{V} \cup \mathcal{C}$.

The elements of $\mathcal{V}$ are called *block openers*, the elements of $\mathcal{V} \cup \mathcal{C}$ are called *identifiers* in [21].

Definition 5.1 (Expressions) We define the set $\mathcal{E}$ of AUT-*68-expressions* (or, in short, *expressions*) inductively:

(variable) If $x \in \mathcal{V}$ then $x \in \mathcal{E}$;

(parameter) If $a \in \mathcal{C}$, $n \in \mathbb{N}$ ($n = 0$ is allowed) and $\Sigma_1, \dots, \Sigma_n \in \mathcal{E}$ then $a(\Sigma_1, \dots, \Sigma_n) \in \mathcal{E}$. $\Sigma_1, \dots, \Sigma_n$ are called the *parameters* of $a(\Sigma_1, \dots, \Sigma_n)$;

(abstraction) If $x \in \mathcal{V}$, $\Sigma \in \mathcal{E} \cup \{\mathtt{type}\}$ and $\Omega \in \mathcal{E}$ then $[x{:}\Sigma]\Omega \in \mathcal{E}$;

(application) If $\Sigma_1, \Sigma_2 \in \mathcal{E}$ then $\langle\Sigma_2\rangle\Sigma_1 \in \mathcal{E}$.

Sometimes we will consider the set $\mathcal{E}^+ \overset{\text{def}}{=} \mathcal{E} \cup \{\mathtt{type}\}$.

Remark 5.2 The AUT-68-expression $[x{:}\Sigma]\Omega$ is AUTOMATH-notation for abstraction terms. In PTS-notation one would write either $\lambda x{:}\Sigma.\Omega$ or $\Pi x{:}\Sigma.\Omega$. In a relatively simple AUTOMATH-system like AUT-68, it is easy to determine whether $\lambda x{:}\Sigma.\Omega$ or $\Pi x{:}\Sigma.\Omega$ is the correct interpretation for $[x{:}\Sigma]\Omega$. This is harder in AUTOMATH-systems with a more complex λ-calculus, like AUT-QE.

Remark 5.3 The AUT-68-expression $\langle\Sigma_2\rangle\Sigma_1$ is AUTOMATH-notation for the intended application of the "function" Σ_1 to the "argument" Σ_2. In PTS-notation: $\Sigma_1\Sigma_2$.

Note the unusual *order* of "function" Σ_1 and "argument" Σ_2 in $\langle\Sigma_2\rangle\Sigma_1$. An advantage of this notation with respect to the classical notation becomes clear if we assume that Σ_1 is a function $[x{:}\Omega_1]\Omega_2$. In that case, $\langle\Sigma_2\rangle\Sigma_1 \equiv \langle\Sigma_2\rangle[x{:}\Omega_1]\Omega_2$. The argument Σ_2 and the abstraction $[x{:}\Omega_1]$ belong together: As soon as the intended application of the function Σ_1 to its argument is carried out, Σ_2 is substituted for x everywhere in Ω_2. It is convenient to put expressions that belong together next to each other. In the usual, classical notation, we would write $([x{:}\Omega_1]\Omega_2)\Sigma_2$, where Σ_2 and $[x{:}\Omega_1]$ are separated from each other by the expression Ω_2. This makes the structure of the expression less clear, in particular if Ω_2 is a very long expression. The advantages of writing $\langle\Sigma_2\rangle\Sigma_1$ instead of the classical $\Sigma_1\Sigma_2$ are extensively discussed in the works of Kamareddine and Nederpelt — see for instance [94], [72], [73].

Definition 5.4 (Free variables)

- $\mathrm{FV}(x) \overset{\text{def}}{=} \{x\}$;
- $\mathrm{FV}(a(\Sigma_1,\ldots,\Sigma_n)) \overset{\text{def}}{=} \bigcup_{i=1}^{n}\mathrm{FV}(\Sigma_i)$;
- $\mathrm{FV}([x{:}\Sigma]\Omega) \overset{\text{def}}{=} \mathrm{FV}(\Sigma)\cup(\mathrm{FV}(\Omega)\setminus\{x\})$;
- $\mathrm{FV}(\langle\Sigma_2\rangle\Sigma_1) \overset{\text{def}}{=} \mathrm{FV}(\Sigma_1)\cup\mathrm{FV}(\Sigma_2)$.

Convention 5.5 We adhere to the usual convention that names of bound variables in an expression differ from the free variables in that expression.

We use $\equiv$ to denote syntactical equivalence (up to renaming of bound variables) on expressions.

Definition 5.6 If $\Omega, \Sigma_1, \ldots, \Sigma_n$ are expressions (in $\mathcal{E}$), and $x_1, \ldots, x_n$ are distinct variables, then

$$\Omega[x_1, \ldots, x_n{:=}\Sigma_1, \ldots, \Sigma_n]$$

denotes the expression Ω in which all free occurrences of $x_1, \ldots, x_n$ have simultaneously been replaced by $\Sigma_1, \ldots, \Sigma_n$. This, again, is an expression in $\mathcal{E}$ (this can be proved by induction on the structure of Ω).

$\texttt{type}[x_1, \ldots, x_n{:=}\Sigma_1, \ldots, \Sigma_n]$ is defined as $\texttt{type}$.

Definition 5.7 (Books and lines) An AUT-68-*book* (or *book* if no confusion arises) is a finite list (possibly empty) of (AUT-68)-lines (to be defined next). If $l_1, \ldots, l_n$ are the lines of book $\mathfrak{B}$, we write $\mathfrak{B} \equiv l_1, \ldots, l_n$.

An AUT-68-*line* (or *line* if no confusion arises) is a 4-tuple $(\Gamma; k; \Sigma_1; \Sigma_2)$. Here,

- Γ is a context, i.e. a finite (possibly empty) list $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$, where the x_is are different elements of $\mathcal{V}$ and the α_is are elements of $\mathcal{E}^+$;
- k is an element of $\mathcal{V} \cup \mathcal{C}$;
- Σ_1 can be (only):
  - The symbol — (if $k \in \mathcal{V}$);
  - The symbol PN (if $k \in \mathcal{C}$) (PN stands for "primitive notion");
  - An element of $\mathcal{E}$ (if $k \in \mathcal{C}$);
- Σ_2 is an element of $\mathcal{E}^+$.

Remark 5.8 As regards the intended meaning of an AUTOMATH-line, we note the following. There are three sorts of lines:

1. $(\Gamma; k; —; \Sigma_2)$ with $k \in \mathcal{V}$. This is a *variable declaration* of the variable k having type Σ_2. This does not really add a new statement to the book, but these declarations are needed to form contexts.

   Variables can play two roles. First of all they can represent an unspecified object of a certain type (compare this to the mathematical way of speaking: "let x be a natural number"). Secondly, a variable can act as a logical assumption. This happens if the variable has as type the proof of a certain proposition A. The usual mathematical way of speaking in such a situation is not "let x be a proof of A", but: "assume A";

2. $(\Gamma; k; \text{PN}; \Sigma_2)$ with $k \in \mathcal{C}$. This line introduces a *primitive notion*: A constant k of type Σ_2. This constant can act as a primitive notion (for instance introducing the type of natural numbers, or introducing the number 0), or as an axiom (to be precise, a postulated inhabitant of the set of proofs of the proposition expressing the axiom).

   The introduction of k is *parametrised* by the context Γ. For instance, if we want to introduce the primitive notion of "logical conjunction", we do not want to have a separate primitive notion for each possible conjunction $\mathtt{and}(A, B)$.[2] Instead, we want to have one primitive notion **and**, to which we can add two propositions A and B as parameters when we want to form the proposition $\mathtt{and}(A, B)$. Therefore, we introduce **and** in a context $\Gamma \equiv \mathtt{x{:}prop}, \mathtt{y{:}prop}$. Given certain propositions A, B this enables us to form the AUT-68-expression $\mathtt{and}(A, B)$;

3. $(\Gamma; k; \Sigma_1; \Sigma_2)$ with $k \in \mathcal{C}$ and $\Sigma_1 \in \mathcal{E}$. This line introduces a *definition*. The *definiendum* k is defined by the *definiens* Σ_1 and has *type* Σ_2. Definitions can be parametrised in a similar way as primitive definitions. Definitions have two important applications:

   - They make it possible to abbreviate long expressions, thus keeping the structure of a book clear, and making manipulations with expressions more efficient;
   - They make it possible to give a name to an expression. For instance, we can abbreviate `S(S(S(S(S(S(S(0)))))))` by `7`.

Example 5.9 In Figure 8 we give an example of an AUTOMATH-book that introduces some elementary notions of propositional logic. We have numbered each line in the example, and use these line numbers for reference in

[2]Contrary to the habit in mathematics to use only one character (possibly indexed) for a variable, AUTOMATH adopts the convention of computer science to use variables that consist of more than one character. So **and** represents only one variable, and not the application of **a** to **n** and **d**.

our comments below. To keep things clear, we have omitted the types of the variables in the context. The book consists of three parts:

- In lines 1–5 we introduce some basic material:

  1. We take the type `prop` as a primitive notion. This type can be interpreted as the type of propositions;
  2. We declare a variable `x` of type `prop`. This variable will be used in the sequel of the book;
  3. We similarly define a variable `y` of type `prop`. We do this within the context `x:prop`. For reasons of space, we do not explicitly mention the type of `x` in the context; if necessary we can find that type in line 2;
  4. Given propositions `x` and `y`, we introduce a new primitive notion, the conjunction `and(x,y)` of `x` and `y`;
  5. Given a proposition `x` we introduce the type `proof(x)` of the proofs of `x` as a primitive notion. In this way, we can use the PAT principle à la de Bruijn (cf. Section 4a4);

- In lines 6–11 we show how we can construct proofs of propositions of the form and(x, y), and how we can use proofs of such propositions:

  6. Given propositions `x` and `y`, we assume that we have a expression `px` $\in \mathcal{V}$ of type `proof(x)`. In other words, the variable `px` represents an arbitrary proof of the proposition `x`;
  7. We also assume a proof `py` of `y`;
  8. Given the propositions `x` and `y`, and proofs `px` and `py` of `x` and `y`, we want to conclude that `and(x,y)` holds. This is an axiom of natural deduction, and we call this axiom `and-I` (and-introduction) in our book. An expression `and-I(x,y,px,py)` is a proof of `and(x,y)`, so of type `proof(and(x,y))`.

     In line 8, we see `proof(and)` instead of `proof(and(x,y))` as the type of `and-I`. This is usual notation in AUTOMATH, and keeps lines short. To be precise, this "default mechanism" works as follows. From line 4, we conclude that `and` should always carry two parameters. This is because the context of line 4 has two

∅	prop	PN	type	(1)
∅	x	—	prop	(2)
x	y	—	prop	(3)
x,y	and	PN	prop	(4)
x	proof	PN	type	(5)
x,y	px	—	proof(x)	(6)
x,y,px	py	—	proof(y)	(7)
x,y,px,py	and-I	PN	proof(and)	(8)
x,y	pxy	—	proof(and)	(9)
x,y,pxy	and-01	PN	proof(x)	(10)
x,y,pxy	and-02	PN	proof(y)	(11)
x	prx	—	proof(x)	(12)
x,prx	and-R	and-I(x,x,prx,prx)	proof(and(x,x))	(13)
x,y,pxy	and-S	and-I(y,x,and-02,and-01)	proof(and(y,x))	(14)

Figure 8: Example of an AUTOMATH-book

variables x and y. In the expression proof(and) in line 8, no parameters are provided for and. It is then implicitly assumed that the first two variables of the context of line 8 are used as "default parameters". The first two variables of the context of line 8 are x and y. Therefore, proof(and) in line 8 should be read as proof(and(x,y)).

In a similar way, we could write proof instead of proof(x) in line 6. From line 5 (where proof is introduced) we find that proof carries one parameter. Writing just proof in line 6 means that we must use the first variable of the context of line 6, x, as a default parameter. We must write proof(y) in line 7. Writing just proof would give proof(x);

9. We also want to express how we can use a proof of and(x,y). Therefore we introduce a variable pxy that represents an arbitrary proof of and(x,y);
10. First of all, we want that x holds whenever and(x,y) holds. Therefore we introduce an axiom and-O1 (and-out, first and-elimination). Given propositions x,y and a proof pxy of the proposition and(x,y), and-O1(x,y,pxy) is a proof of x;
11. Similarly, we introduce an axiom and-O2 that represents a proof of y;

- We can now derive some elementary theorems:

12. We want to prove that we can derive and(x,x) from x. That is: Whenever we have a proof of x, we can construct a proof of and(x,x). In line 6, we already introduced a variable for a proof of x: px. However, we declared this variable in the context x,y. As we do not want a second proposition y to occur in this theorem, we declare a new proof variable prx, in the context x;
13. We derive our first small theorem: The reflexivity of the logical conjunction. Given a proposition x, and a proof prx of x, we can use the axiom and-I to find a proof of and(x,x): we can use the expression and-I(x,x,px,px) thanks to line 8. We give a name to this proof: and-R. If, anywhere in the sequel of the book, Σ is a proposition, and Ω is a proof of Σ, we can write

and-R(Σ, Ω) for a proof of and(Σ, Σ). This is shorter, and more expressive, than the original expression and-I$(\Sigma, \Sigma, \Omega, \Omega)$;

14. We can also show that and is symmetric. That is: Whenever `and(x,y)` holds, we also have `and(y,x)`. The idea is as follows. Given propositions `x,y` and a proof `pxy` of `and(x,y)`, we can form proofs `and-O1(x,y,pxy)` of `x` and `and-O2(x,y,pxy)` of `y`. We can feed these proofs "in reverse order" to the axiom `and-I`: The expression `and-I(y,x,and-O2,and-O1)` represents a proof of `and(y,x)`. The expression `and-O2` should be read as `and-O2(x,y,pxy)` due to the "default parameter" mechanism. Similarly, `and-O1` must be read as `and-O1(x,y,pxy)`.

5a2 Correct books

Not all books are good books. If $(\Gamma; k; \Sigma_1; \Sigma_2)$ is a line of a book $\mathfrak{B}$, the expressions Σ_1 and Σ_2 (as long as Σ_1 is not PN or —, and Σ_2 is not `type`) must be well-defined, i.e. the elements of $\mathcal{V} \cup \mathcal{C}$ occurring in them must have been established (as variables, primitive notions, or defined constants) in previous parts of $\mathfrak{B}$. The same holds for the type assignments $x_i{:}\alpha_i$ that occur in Γ. Moreover, if Σ_1 is not PN or —, then Σ_1 must be of the same type as k, hence Σ_1 must be of type Σ_2 (within the context Γ). Finally, there should be only one definition of any object in a book, so k should not occur in the preceding lines of the book.

Hence we need notions of correctness (with respect to a book and/or a context) and we need a definition of the notion "Σ_1 is of type Σ_2" (within a book and a context).

We write $\mathfrak{B}; \varnothing \vdash \text{OK}$ to indicate that a book $\mathfrak{B}$ is correct, and $\mathfrak{B}; \Gamma \vdash \text{OK}$ to indicate that the context Γ is correct with respect to the (correct) book $\mathfrak{B}$. As the empty context will be correct with respect to any correct book, this does not lead to misunderstandings.

We write $\mathfrak{B}; \Gamma \vdash \Sigma$ (or $\mathfrak{B}; \Gamma \vdash_{\text{AUT-68}} \Sigma$ if confusion with other derivation systems might arise) to indicate that Σ is a correct expression with respect to $\mathfrak{B}$ and Γ. We write $\mathfrak{B}; \Gamma \vdash \Sigma_1 : \Sigma_2$ (or $\mathfrak{B}; \Gamma \vdash_{\text{AUT-68}} \Sigma_1 : \Sigma_2$) to indicate that Σ_1 is a correct expression of type Σ_2 with respect to $\mathfrak{B}$ and Γ. We also say: $\Sigma_1 : \Sigma_2$ is a correct *statement* with respect to $\mathfrak{B}$ and Γ.

The following two interrelated definitions are based on [40].

Definition 5.10 (Correct books and contexts) A book $\mathfrak{B}$ and a context Γ are *correct* if $\mathfrak{B};\Gamma \vdash \text{OK}$ can be derived with the following rules (The relation $=_{\beta\text{d}}$ ("definitional equality") will be explained in Section 5a3. The rules use the notion of *correct statement* as given in Definition 5.11).

(axiom) $$\varnothing;\varnothing \vdash \text{OK}$$

(context ext.) $$\frac{\mathfrak{B}_1,(\Gamma;x;\text{—};\alpha),\mathfrak{B}_2;\Gamma \vdash \text{OK}}{\mathfrak{B}_1,(\Gamma;x;\text{—};\alpha),\mathfrak{B}_2;\Gamma,x{:}\alpha \vdash \text{OK}}$$

(book ext.: var1) $$\frac{\mathfrak{B};\Gamma \vdash \text{OK}}{\mathfrak{B},(\Gamma;x;\text{—};\mathtt{type});\varnothing \vdash \text{OK}}$$

(book ext.: var2) $$\frac{\mathfrak{B};\Gamma \vdash \Sigma_2 : \mathtt{type}}{\mathfrak{B},(\Gamma;x;\text{—};\Sigma_2);\varnothing \vdash \text{OK}}$$

(book ext.: pn1) $$\frac{\mathfrak{B};\Gamma \vdash \text{OK}}{\mathfrak{B},(\Gamma;k;\text{PN};\mathtt{type});\varnothing \vdash \text{OK}}$$

(book ext.: pn2) $$\frac{\mathfrak{B};\Gamma \vdash \Sigma_2 : \mathtt{type}}{\mathfrak{B},(\Gamma;k;\text{PN};\Sigma_2);\varnothing \vdash \text{OK}}$$

(book ext.: def1) $$\frac{\mathfrak{B};\Gamma \vdash \Sigma_1 : \mathtt{type}}{\mathfrak{B},(\Gamma;k;\Sigma_1;\mathtt{type});\varnothing \vdash \text{OK}}$$

(book ext.: def2)
$$\frac{\mathfrak{B};\Gamma \vdash \Sigma_2 : \mathtt{type} \qquad \mathfrak{B};\Gamma \vdash \Sigma_1 : \Sigma_2' \qquad \mathfrak{B};\Gamma \vdash \Sigma_2 =_{\beta\text{d}} \Sigma_2'}{\mathfrak{B},(\Gamma;k;\Sigma_1;\Sigma_2);\varnothing \vdash \text{OK}}$$

For the (book ext.) rules, we assume that the introduced identifiers $x \in \mathcal{V}$ and $k \in \mathcal{C}$ do not occur anywhere in $\mathfrak{B}$ and Γ.

Definition 5.11 (Correct statements) A statement $\mathfrak{B};\Gamma \vdash \Sigma : \Omega$ is *correct* if it can be derived with the rules below (the start rule uses the notions of correct context and correct book as given in Definition 5.10).

(start) $$\frac{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2 \vdash \text{OK}}{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2 \vdash x{:}\alpha}$$

(parameters)
$$\mathfrak{B} \equiv \mathfrak{B}_1,(x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n;b;\Omega_1;\Omega_2),\mathfrak{B}_2$$
$$\frac{\mathfrak{B};\Gamma \vdash \Sigma_i{:}\alpha_i[x_1,\ldots,x_{i-1}{:=}\Sigma_1,\ldots,\Sigma_{i-1}]\,(i=1,\ldots,n)}{\mathfrak{B};\Gamma \vdash b(\Sigma_1,\ldots,\Sigma_n) : \Omega_2[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]}$$

(abstr.1) $$\frac{\mathfrak{B};\Gamma \vdash \Sigma_1{:}\mathtt{type} \qquad \mathfrak{B};\Gamma,x{:}\Sigma_1 \vdash \Omega_1{:}\mathtt{type}}{\mathfrak{B};\Gamma \vdash [x{:}\Sigma_1]\Omega_1 : \mathtt{type}}$$

(abstr.2)

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}\mathtt{type}\qquad \mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Omega_1{:}\mathtt{type}\qquad \mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Sigma_2{:}\Omega_1}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Sigma_2:[x{:}\Sigma_1]\Omega_1}$$

(application)

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1:[x{:}\Omega_1]\Omega_2\qquad \mathfrak{B};\Gamma\vdash\Sigma_2:\Omega_1}{\mathfrak{B};\Gamma\vdash\langle\Sigma_2\rangle\Sigma_1:\Omega_2[x{:=}\Sigma_2]}$$

(conversion)

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma:\Omega_1\qquad \mathfrak{B};\Gamma\vdash\Omega_2{:}\mathtt{type}\qquad \mathfrak{B};\Gamma\vdash\Omega_1=_{\beta\mathrm{d}}\Omega_2}{\mathfrak{B};\Gamma\vdash\Sigma:\Omega_2}$$

When using the parameter rule, we assume that $\mathfrak{B};\Gamma\vdash$ OK, even if $n=0$.

Example 5.12 The book of Example 5.9 (see Figure 8) is correct. We prove this line by line for the first four lines (the reader is invited to check lines 5–14 for himself). We write $(m\text{–}n)$ to denote the book that consists of lines m to n of Example 5.9.

1. By (axiom), $\varnothing;\varnothing\vdash$ OK, so by (book ext.: pn1),

   $$(\varnothing;\mathtt{prop};\mathrm{PN};\mathtt{type});\varnothing\vdash \mathrm{OK}.$$

2. By (parameters), $(1\text{–}1);\varnothing\vdash\mathtt{prop}:\mathtt{type}$. Therefore by (book ext.: var1), $(1\text{–}1),(\varnothing,\mathtt{x},\text{—},\mathtt{prop});\varnothing\vdash$ OK.

3. By (context ext.), $(1\text{–}2);\mathtt{x}{:}\mathtt{prop}\vdash$ OK. Therefore by (book ext.: var1), $(1\text{–}2),(\mathtt{x}{:}\mathtt{prop};\mathtt{y};\text{—};\mathtt{prop})\vdash$ OK.

4. By two applications of (context ext.), $(1\text{–}3);\mathtt{x}{:}\mathtt{prop},\mathtt{y}{:}\mathtt{prop}\vdash$ OK. By (parameters), $(1\text{–}3);\mathtt{x}{:}\mathtt{prop},\mathtt{y}{:}\mathtt{prop}\vdash\mathtt{prop}{:}\mathtt{type}$. Therefore by (book ext.: pn2), $(1\text{–}4);\varnothing\vdash$ OK.

5a3 Definitional equality

We still need to describe the relation $=_{\beta\mathrm{d}}$ ("definitional equality"). This notion is based on both the definition mechanism and the abstraction/application mechanism of AUT-68. The abstraction/application mechanism provides the well-known notion of β-equality, originating from the rule of β-conversion:

$$\langle\Sigma\rangle[x{:}\Omega_2]\Omega_1 \quad\to_\beta\quad \Omega_1[x{:=}\Sigma].$$

We will use notations like $\twoheadrightarrow_\beta$, $=_\beta$, and $\rightarrow^+_\beta$ as usual (see A.11).

We now describe the definition mechanism of AUT-68 via the notion of d-*equality*. This definition depends on the definition of derivability, and the definition of derivability given in the previous subsection depends on the definition of definitional equality. In fact, the definitions of correct book, correct line, correct context, correct expression and definitional equality should be given within one definition, using induction on the length of the book. This would lead to a correct but very long definition, and that is probably the reason why the definitions are split into smaller parts (in this thesis as well as in [40]).

Definition 5.13 (d-equality) Assume, $\mathfrak{B};\Gamma \vdash \Sigma : \Sigma'$. We define the d-*normal form* $\mathrm{nf_d}(\Sigma)$ of Σ with respect to $\mathfrak{B}$ by induction on the length of the book $\mathfrak{B}$. So, assume $\mathrm{nf_d}(\Sigma)$ has already been defined for all books $\mathfrak{B}'$ with less lines than $\mathfrak{B}$, and all expressions Σ that are correct with respect to $\mathfrak{B}'$ and a context Γ. Use induction on the structure of Σ:

- If Σ is a variable x, then $\mathrm{nf_d}(\Sigma) \overset{\mathrm{def}}{=} x$;
- Now assume $\Sigma \equiv b(\Omega_1, \ldots, \Omega_n)$, and assume that the normal forms of the Ω_is have already been defined.

  Determine a line $(\Delta; b; \Xi_1; \Xi_2)$ in the book $\mathfrak{B}$ (there is exactly one such line, and this line is determined by b).

  Write $\Delta \equiv x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$. Distinguish:

  - $\Xi_1 \equiv$ —. This case doesn't occur, as $b \in \mathcal{C}$;
  - $\Xi_1 \equiv$ PN. Then define $\mathrm{nf_d}(\Sigma) \overset{\mathrm{def}}{=} b(\mathrm{nf_d}(\Omega_1), \ldots, \mathrm{nf_d}(\Omega_n))$;
  - Ξ_1 is an expression. Then Ξ_1 is correct with respect to a book $\mathfrak{B}'$ that contains less lines than $\mathfrak{B}$ ($\mathfrak{B}'$ doesn't contain the line $(\Delta; b; \Xi_1; \Xi_2)$, and all lines of $\mathfrak{B}'$ are also lines of $\mathfrak{B}$), hence we can assume that $\mathrm{nf_d}(\Xi_1)$ has already been defined. Now define

$$\mathrm{nf_d}(\Sigma) \overset{\mathrm{def}}{=} \mathrm{nf_d}(\Xi_1)[x_1, \ldots, x_n{:=}\mathrm{nf_d}(\Omega_1), \ldots, \mathrm{nf_d}(\Omega_n)];$$

- If $\Sigma \equiv [x{:}\Omega_1]\Omega_2$ then $\mathrm{nf_d}(\Sigma) \overset{\mathrm{def}}{=} [x{:}\mathrm{nf_d}(\Omega_1)]\mathrm{nf_d}(\Omega_2)$;
- If $\Sigma \equiv \langle\Omega_2\rangle\Omega_1$ then $\mathrm{nf_d}(\Sigma) \overset{\mathrm{def}}{=} \langle\mathrm{nf_d}(\Omega_2)\rangle\mathrm{nf_d}(\Omega_1)$.

We write $\Sigma_1 =_{\mathrm{d}} \Sigma_2$ when $\mathrm{nf_d}(\Sigma_1) \equiv \mathrm{nf_d}(\Sigma_2)$.

As we see, the d-normal form $\mathrm{nf_d}(\Sigma)$ of a correct expression Σ depends on the book $\mathfrak{B}$, and in order to be completely correct we should write $\mathrm{nf}_{\mathrm{d}\mathfrak{B}}(\Sigma)$ instead of only $\mathrm{nf_d}(\Sigma)$. We will, however, omit the subscript $\mathfrak{B}$ as long as no confusion arises.

We write $=_{\beta\mathrm{d}}$ for the smallest equivalence relation that contains both $=_\beta$ and $=_{\mathrm{d}}$.

Definition 5.14 (Definitional equality) Σ_1 and Σ_2 are called *definitionally equal* (with respect to a book $\mathfrak{B}$) if $\Sigma_1 =_{\beta\mathrm{d}} \Sigma_2$.

This definition completes the description of AUT-68. Again, definitional equality of expressions Σ_1 and Σ_2 depends on the book $\mathfrak{B}$, so we should write $=_{\beta\mathrm{d}\mathfrak{B}}$ instead of $=_{\beta\mathrm{d}}$. Also in this case we leave out the subscript $\mathfrak{B}$ as long as no confusion arises.

As an alternative to Definition 5.13, we describe the notion of d-equality via a reduction relation.

Definition 5.15 (δ-reduction) Let $\mathfrak{B}$ be a book, Γ a correct context with respect to $\mathfrak{B}$, and Σ a correct expression with respect to $\mathfrak{B};\Gamma$. We define $\Sigma \to_\delta \Omega$ by the usual compatibility rules, and

(δ) If $\Sigma = b(\Sigma_1, \ldots, \Sigma_n)$, and $\mathfrak{B}$ contains a line $(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$ where $\Xi_1 \in \mathcal{E}$, then

$$\Sigma \to_\delta \Xi_1[x_1, \ldots, x_n{:=}\Sigma_1, \ldots, \Sigma_n].$$

We say that Σ is in δ-normal form if for no expression Ω, $\Sigma \to_\delta \Omega$, and use notations like $\twoheadrightarrow_\delta$, $\twoheadrightarrow_\delta^+$ and $=_\delta$ as usual. $\to_\delta$ depends on $\mathfrak{B}$, but as we did before with $\mathrm{nf_d}$ and $=_{\mathrm{d}}$, we only explicitly mention this if it is not clear in relation to which book $\mathfrak{B}$ $\to_\delta$ is considered.

The relations $=_{\mathrm{d}}$ and $=_\delta$ are the same:

Lemma 5.16

1. *(Church-Rosser) If $A_1 =_\delta A_2$ then there is B such that $A_1 \twoheadrightarrow_\delta B$ and $A_2 \twoheadrightarrow_\delta B$;*

2. $\mathrm{nf_d}(\Sigma)$ *is the (unique)* δ*-normal form of* Σ*;*

3. $\Sigma =_\delta \Omega$ *if and only if* $\Sigma =_\mathrm{d} \Omega$*.*

PROOF: AUT-68 with $\rightarrow_\delta$ can be seen as an orthogonal term rewrite system (see [75]).

1. Such a term rewrite system has the Church-Rosser property (see [75]);
2. It is not hard to show that $\Sigma \twoheadrightarrow_\delta \mathrm{nf_d}(\Sigma)$. By induction on the definition of $\mathrm{nf_d}(\Sigma)$ one shows that $\mathrm{nf_d}(\Sigma)$ is in δ-normal form. The uniqueness of this normal form follows from the Church-Rosser property;
3. If $\Sigma =_\delta \Omega$ then by (1) there is Ψ such that $\Sigma \rightarrow_\delta \Psi$ and $\Omega \rightarrow_\delta \Psi$. This means that the δ-normal forms of Σ and Ω are equal, so by (2), $\mathrm{nf_d}(\Sigma) \equiv \mathrm{nf_d}(\Omega)$.
   On the other hand, if $\mathrm{nf_d}(\Sigma) \equiv \mathrm{nf_d}(\Omega)$, then Σ and Ω have the same δ-normal forms (by (2)), so $\Sigma =_\delta \Omega$.

⊠

Lemma 5.17 *The relation* $\rightarrow_\delta$ *is strongly normalising.*

PROOF: We already know that $\rightarrow_\delta$ is weakly normalising (by 2). Moreover, the definition of $\mathrm{nf_d}(\Sigma)$ in 5.13 induces an innermost reduction strategy. By a theorem of O'Donnell (see [96], or pp. 75–76 of [75]), $\rightarrow_\delta$ is strongly normalising. ⊠

5a4 Some elementary properties

Although we do not want to give a complete overview of all the metatheoretical properties of AUTOMATH (these are studied in [91] and [40]), we do present some properties that we will need at a later stage.

Definition 5.18 A book $\mathfrak{B}$ is part of another book $\mathfrak{B}'$, denoted as $\mathfrak{B} \subseteq \mathfrak{B}'$, if all lines of $\mathfrak{B}$ are lines of $\mathfrak{B}'$ as well. Similarly, a context Γ is part of another context Γ', notation $\Gamma \subseteq \Gamma'$, if all declarations $x{:}\alpha$ of Γ are declarations in Γ' as well.

Lemma 5.19 (Weakening for AUT-68) *If* $\mathfrak{B};\Gamma\vdash\Sigma:\Omega$, $\mathfrak{B}\subseteq\mathfrak{B}'$, $\Gamma\subseteq\Gamma'$ *and* $\mathfrak{B}';\Gamma'\vdash\text{OK}$ *then* $\mathfrak{B}';\Gamma'\vdash\Sigma:\Omega$.

PROOF: By induction on the derivation of $\mathfrak{B};\Gamma\vdash\Sigma:\Omega$. ⊠

5b From AUT-68 towards a PTS

We want to give a description of AUT-68 within the framework of the Pure Type Systems. There are several ways to do this. One of the most important choices to be made is whether or not to maintain the parameter mechanism (that is: To allow expressions with parameters, as in the second clause of Definition 5.1). On the one hand, the parameter mechanism is an important feature of AUTOMATH. On the other hand PTSs do not have a parameter mechanism, and the parameter mechanism can be easily imitated by function application (cf. the second clause of the forthcoming Definition 5.20). Moreover, the description by van Benthem Jutting in [5] of the systems AUT-68 and AUT-QE in a PTS style does not use parameters.

In this chapter, we provide a translation to PTSs without parameters. In doing so, we can explain van Benthem Jutting's description of AUT-68 and AUT-QE.

We will see, however, that the way in which we must handle parameters in the resulting PTS is a bit artificial. Moreover, we think that parameters play an important role in the AUTOMATH systems, and that they could play a similar role in other PTSs. Therefore, we will present an extension of PTSs with parameters in Chapter 6. This extension is based on the way in which parameters are handled in AUTOMATH, and it will be shown that AUTOMATH can be described very well within these PTSs with parameters.

For a description of AUT-68 in PTSs without parameters, we must first make a translation of the expressions in AUT-68 to typed λ-terms. This translation is very straightforward:

Definition 5.20 We define a mapping $\overline{[\ldots]}$ from the correct expressions in $\mathcal{E}$ (relative to a book $\mathfrak{B}$ and a context Γ) to $\mathbb{T}$, the set of terms for PTSs. We assume that $\mathcal{C}\cup\mathcal{V}\subseteq\mathbb{V}$ ($\mathbb{V}$ is the set of variables for PTS-terms).

- $\overline{x}\stackrel{\text{def}}{=}x$ for $x\in\mathcal{V}$;

- $\overline{b(\Sigma_1,\ldots,\Sigma_n)} \stackrel{\text{def}}{=} b\overline{\Sigma_1}\cdots\overline{\Sigma_n}$;
- $\overline{[x{:}\Sigma]\Omega} \stackrel{\text{def}}{=} \Pi x{:}\overline{\Sigma}.\overline{\Omega}$ if $[x{:}\Sigma]\Omega$ has type type, otherwise $\overline{[x{:}\Sigma]\Omega} \stackrel{\text{def}}{=} \lambda x{:}\overline{\Sigma}.\overline{\Omega}$;
- $\overline{\langle\Omega\rangle\Sigma} \stackrel{\text{def}}{=} \overline{\Sigma}\,\overline{\Omega}$.

Moreover, we define: $\overline{\mathtt{type}} \stackrel{\text{def}}{=} *$.

In the second clause of this definition we see that the parameter mechanism of Definition 5.1 is replaced by repeated function application in PTSs.

With this translation in mind, we want to find a type system $\lambda 68$ that "suits" AUT68, i.e. if Σ is a correct expression of type Ω with respect to a book $\mathfrak{B}$ and a context Γ, then we want $\mathfrak{B}',\Gamma' \vdash \overline{\Sigma} : \overline{\Omega}$ to be derivable in $\lambda 68$, and vice versa. Here, $\mathfrak{B}'$ and Γ' are some suitable translations of $\mathfrak{B}$ and Γ. The search for a suitable $\lambda 68$ will concentrate on three points, which we first discuss informally. In the next section we give a formal definition of $\lambda 68$, and prove that it has the desired property.

5b1 The choice of the correct formation (Π) rules

When we keep in mind that $\overline{\mathtt{type}} \equiv *$, the definition of correct expressions 5.11 gives a clear answer to the question of which Π-rules are implied by the abstraction mechanism of AUT-68. The rule

$$\frac{\mathfrak{B};\Gamma \vdash \Sigma_1{:}\mathtt{type} \qquad \mathfrak{B};\Gamma, x{:}\Sigma_1 \vdash \Omega_1{:}\mathtt{type}}{\mathfrak{B};\Gamma \vdash [x{:}\Sigma_1]\Omega_1 : \mathtt{type}}$$

immediately translates into Π-rule $(*,*,*)$ for PTSs:

$$\frac{\overline{\mathfrak{B}},\overline{\Gamma} \vdash \overline{\Sigma_1}{:}\, * \qquad \overline{\mathfrak{B}},\overline{\Gamma}, x{:}\overline{\Sigma_1} \vdash \overline{\Omega_1}{:}*}{\overline{\mathfrak{B}},\overline{\Gamma} \vdash (\Pi x{:}\overline{\Sigma_1}.\overline{\Omega_1}) : *},$$

where $\overline{\mathfrak{B}}$ and $\overline{\Gamma}$ are suitable translations of $\mathfrak{B}$ and Γ.

It is, however, not immediately clear which Π-rules are induced by the parameter mechanism of AUT-68.

Let $\Sigma \equiv b(\Sigma_1, \dots, \Sigma_n)$ be a correct expression of type Ω with respect to a book $\mathfrak{B}$ and a context Γ. rhere is a line (see Definition 5.10)

$$(x_1{:}\alpha_1, \dots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$$

in $\mathfrak{B}$ such that each Σ_i is a correct expression with respect to $\mathfrak{B}$ and Γ, and has a type that is definitionally equal to $\alpha_i[x_1, \dots, x_{i-1}{:=}\Sigma_1, \dots, \Sigma_{i-1}]$. We also know that $\Omega =_{\beta\mathrm{d}} \Xi_2[x_1, \dots, x_n{:=}\Sigma_1, \dots \Sigma_n]$.

Now $\overline{\Sigma} \equiv b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$, and, assuming that we can derive in $\lambda 68$ that $\overline{\Sigma_i}$ has type

$$\overline{\alpha_i}[x_1, \dots, x_{i-1}{:=}\overline{\Sigma_1}, \dots, \overline{\Sigma_{i-1}}],$$

it is not unreasonable to assign the type $\Pi x_1{:}\overline{\alpha_1} \cdots \Pi x_n{:}\overline{\alpha_n} to b.\overline{\Xi_2}$. We will abbreviate this last term by $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$. Then we can derive (using n times the application rule that we will introduce for $\lambda 68$) that $\overline{\Sigma}$ has type $\overline{\Omega}$ in $\lambda 68$.

It is important to notice that the type of b, $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, does not necessarily have an equivalent in AUT-68, as in AUT-68 abstractions over `type` are not allowed (only abstractions over expressions Σ that have `type` as type are possible — cf. Definition 5.11). In other words, the type of b, $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, is not necessarily a first-class citizen of AUT-68 and should therefore have special treatment in $\lambda 68$. This is the reason to create a special sort $\triangle$, in which these types of AUT-68 constants and definitions are stored. This idea originates from Van Benthem Jutting, and was firstly presented in [5].

If we construct $\Pi x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$ from $\overline{\Xi_2}$, we must use a rule (s_1, s_2, s_3), where s_1, s_2, s_3 are sorts. Sort s_1 must be the type of $\overline{\alpha_n}$. As $\alpha_n \equiv$ `type` or α_n has type `type`, we must allow the possibilities $s_1 \equiv *$ and $s_1 \equiv \Box$. Similarly, $\Xi_2 \equiv$ `type` or Ξ_2 has type `type`, so we also allow $s_2 \equiv *$ and $s_2 \equiv \Box$. As we intended to store the new type in sort $\triangle$, we take $s_3 \equiv \triangle$.

For similar reasons, we introduce rules $(*, \triangle, \triangle)$ and $(\Box, \triangle, \triangle)$ to construct $\prod_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$ from $\Pi x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$ for $n > 1$.

As a result, we have the following Π-rules:

$$\begin{array}{cc} \multicolumn{2}{c}{(*, *, *);} \\ (*, *, \triangle); & (\Box, *, \triangle); \\ (*, \Box, \triangle); & (\Box, \Box, \triangle); \\ (*, \triangle, \triangle); & (\Box, \triangle, \triangle). \end{array}$$

We do not have rules of the form $(\triangle, s_2, s_3)$ or $(s_1, \triangle, s_3)$ with $s_3 \equiv *$ or $s_3 \equiv \Box$. So types of sort $\triangle$ cannot be used to construct types of other sorts. In this way, we can keep the types of the λ-calculus part of AUT-68 separated from the types of the parameter mechanism: The last ones are stored in $\triangle$.

In Example 5.2.4.8 of [5], there is no rule $(*, *, \triangle)$. In principle, this rule is superfluous, as each application of rule $(*, *, \triangle)$ can be replaced by an application of rule $(*, *, *)$. Nevertheless we want to maintain this rule:

- First of all, the presence of both $(*, *, *)$ and $(*, *, \triangle)$ in the system stresses the fact that AUT-68 has two type mechanisms: One provided by the parameter mechanism and one by the λ-abstraction mechanism;

- Secondly, there are technical arguments to make a distinction between types formed by the abstraction mechanism and types that appear via the parameter mechanism. In this thesis, we will denote product types constructed by the abstraction mechanism in the usual way (so: $\Pi x{:}A.B$), whilst we will (from now on) use the notation $\P x{:}A.B$ for a type constructed by the parameter mechanism. Hence, we have for the constant b above that $b : \P_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$[3]. As an additional advantage, the resulting system will maintain Unicity of Types.[4] This would have been lost if we had introduced rules $(*, *, *)$ and $(*, *, \triangle)$ without making this difference, as we can then derive both

$$\frac{\alpha{:}* \vdash \alpha : * \qquad \alpha{:}*, x{:}\alpha \vdash \alpha{:}*}{\alpha{:}* \vdash (\Pi x{:}\alpha.\alpha) : *} \text{ (rule } (*, *, *))$$

and

$$\frac{\alpha{:}* \vdash \alpha : * \qquad \alpha{:}*, x{:}\alpha \vdash \alpha{:}*}{\alpha{:}* \vdash (\Pi x{:}\alpha.\alpha) : \triangle} \text{ (rule } (*, *, \triangle));$$

- There is another reason to make a distinction between types formed by the abstraction mechanism and types that appear in the translation via the definition mechanism. For the moment, we consider AUT-68 *without* so-called Π-application. In AUT-68 with Π-application

[3]we use $\P_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$ as an abbreviation for $\P x_1{:}\overline{\alpha_1} \cdots \P x_n{:}\overline{\alpha_n}.\overline{\Xi_2}$

[4]The system as presented in [5] has Unicity of Types as well, because it does not have the Π-formation rule $(*, *, \triangle)$ and is therefore singly sorted.

(call this system AUT-68Π for the moment; see also Section 5d3) the application rule of Definition 5.11

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}[x{:}\Omega_1]\Omega_2 \qquad \mathfrak{B};\Gamma\vdash\Sigma_2{:}\Omega_1}{\mathfrak{B};\Gamma\vdash\langle\Sigma_2\rangle\Sigma_1 : \Omega_2[x{:=}\Sigma_2]}$$

is replaced by

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}[x{:}\Omega_1]\Omega_2 \qquad \mathfrak{B};\Gamma\vdash\Sigma_2{:}\Omega_1}{\mathfrak{B};\Gamma\vdash\langle\Sigma_2\rangle\Sigma_1 : \langle\Sigma_2\rangle\Omega_2},$$

but the rule describing the type of $b(\Sigma_1,\dots,\Sigma_n)$ is the same as the rule in Definition 5.11 (parameters).

So if we want to make a translation of AUT-68Π, the application rule for Π-terms has to be different from the application rule for ¶-terms. Without distinction between Π-terms and ¶-terms, it would be impossible to amend the system to represent AUT-68Π. Distinguishing between Π-terms and ¶-terms makes it possible to obtain a translation of AUT-68Π from the translation of AUT-68 in a simple way.

5b2 The different treatment of constants and variables

When we seek for a translation in λ68 of the AUT-68 judgement $\mathfrak{B};\Gamma\vdash\Sigma:\Omega$, we must pay extra attention to the translation of $\mathfrak{B}$, as there is no equivalent of books in PTSs. Our solution is to store the information on identifiers of $\mathfrak{B}$ in a PTS-context. Therefore, contexts of λ68 will have the form $\Delta;\Gamma$. The left part Δ contains type information on primitive notions and definitions, and can be seen as the translation of the information on primitive notions and definitions in $\mathfrak{B}$. In the right part Γ we find the usual type information on variables.

The idea to store the constant information of $\mathfrak{B}$ in the left part of the context arises in a natural way. Let $\mathfrak{B}$ be a correct AUT-68 book, to which we add a line $(\Gamma; b; \text{PN}; \Xi_2)$. Then $\Gamma\equiv x_1{:}\alpha_1,\dots,x_n{:}\alpha_n$ is a correct context with respect to $\mathfrak{B}$, and $\mathfrak{B};\Gamma\vdash\Xi_2{:}\mathtt{type}$ or $\Xi_2\equiv\mathtt{type}$. In λ68 we can work as follows. Assume the information on constants in $\mathfrak{B}$ has been translated into the left part Δ of a λ68 context. We have (assuming that λ68 is a type system that behaves like AUT-68, and writing $\overline{\Gamma}$ for the translation $x_1{:}\overline{\alpha_1},\dots,x_n{:}\overline{\alpha_n}$ of Γ):

$$\Delta;\overline{\Gamma}\vdash\overline{\Xi_2}{:}s$$

($s \equiv *$ if $\mathfrak{B}; \Gamma \vdash \Xi_2{:}\mathtt{type}$; $s \equiv \Box$ if $\Xi_2 \equiv \mathtt{type}$). Applying the ¶-formation rule n times, we obtain

$$\Delta; \varnothing \vdash \P\,\overline{\Gamma}.\overline{\Xi_2} : \Delta$$

(If Γ is the empty context, then $\P\,\overline{\Gamma}.\overline{\Xi_2} \equiv \overline{\Xi_2}$, and $\overline{\Xi_2}$ has type $*$ or $\Box$ instead of Δ. We write $\P\,\overline{\Gamma}$ for $\P_{i=1}^{n}\, x_i{:}\overline{\alpha_i}$). As $\P\,\overline{\Gamma}.\overline{\Xi_2}$ is exactly the type that we want to give to b (see the discussion in Subsection 5b1), we use this statement as premise for the start rule that introduces b. As the right part $\overline{\Gamma}$ of the original context has disappeared when we applied the ¶-formation rules, $b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2}$ is automatically placed at the righthand end of Δ: The conclusion of the start rule is

$$\Delta, b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2} \vdash b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2}.$$

Adding $b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2}$ at the end of Δ can be compared with adding the line $(\Gamma; b; \text{PN}; \Xi_2)$ at the end of $\mathfrak{B}$.

The process above can be captured in one rule:

$$\frac{\Delta; \overline{\Gamma} \vdash \overline{\Xi_2}{:}s_1 \qquad \Delta; \vdash \P\,\overline{\Gamma}.\overline{\Xi_2}{:}s_2}{\Delta, b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2}; \vdash b{:}\,\P\,\overline{\Gamma}.\overline{\Xi_2}}.$$

Here $s_1 \in \{*, \Box\}$ (compare: $\Xi_2{:}\mathtt{type}$ or $\Xi_2 \equiv \mathtt{type}$) and $s_2 \in \{*, \Box, \Delta\}$ (usually, $s_2 \equiv \Delta$; the cases $s_2 \equiv *, \Box$ only occur if Γ is empty).

5b3 The definition system

A line $(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$, in which b is a constant and $\Xi_1 \in \mathcal{E}$, represents a definition. It should be read as: For all expressions $\Omega_1, \ldots, \Omega_n$ (obeying certain type conditions), $b(\Omega_1, \ldots, \Omega_n)$ is an abbreviation for $\Xi_1[x_1, \ldots, x_n{:=}\Omega_1, \ldots, \Omega_n]$, and has type

$$\Xi_2[x_1, \ldots, x_n{:=}\Omega_1, \ldots, \Omega_n].$$

So in $\lambda 68$, the context should also mention that $bX_1 \cdots X_n$ "is equal to" $\Xi_1[x_1, \ldots, x_n{:=}X_1, \ldots, X_n]$, for all terms $X_1, \ldots, X_n$. The most straightforward way to do this, is to write

$$b{:=} \left(\mathop{\lambda}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1} \right) : \left(\P_{i=1}^{n}\, x_i{:}\overline{\alpha_i}.\overline{\Xi_2} \right)$$

in the context instead of only b: $\P_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, and adding a δ-reduction rule that allows to unfold the definition of b:

$$\Delta \vdash b \to_\delta \mathop{\lambda}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$$

whenever $b{:=} \left(\lambda_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\P_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right) \in \Delta$.

Unfolding the definition of b in a term $b\overline{\Sigma_1}\cdots\overline{\Sigma_n}$ and applying β-reduction n times results in $\overline{\Xi_1}[x_1{:=}\overline{\Sigma_1}]\cdots[x_n{:=}\overline{\Sigma_n}]$. This procedure corresponds exactly to the δ-reduction

$$\Delta \vdash b(\Sigma_1, \dots, \Sigma_n) \to_\delta \Xi_1[x_1, \dots, x_n{:=}\Sigma_1, \dots, \Sigma_n]$$

in AUT-68[5].

This method, however, has some disadvantages.

- Look again at a line $(x_1{:}\alpha_1, \dots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2)$ in an AUT-68 book. Then $b(\Sigma_1, \dots, \Sigma_n)$ has $b\overline{\Sigma_1}\cdots\overline{\Sigma_n}$ as its equivalent in $\lambda 68$. If $n > 0$, the latter $\lambda 68$-term has $B \equiv b\overline{\Sigma_1}\cdots\overline{\Sigma_m}$ as a subterm for any $m < n$. But B has no equivalent in AUT-68: Only after B has been applied to suitable terms $\overline{\Sigma_{m+1}}, \dots, \overline{\Sigma_n}$ the resulting term $B\overline{\Sigma_{m+1}}\cdots\overline{\Sigma_n}$ has $b(\Sigma_1, \dots, \Sigma_n)$ as its equivalent in AUT-68. Hence B must not be seen as a term directly translatable into AUTOMATH, but only as an intermediate result that is necessary to construct the equivalent of the expression $b(\Sigma_1, \dots, \Sigma_n)$. B is recognisable as an intermediate result via its type $\P_{i=m+1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_2}$, which has sort Δ (instead of $*$ or $\Box$).

  The method above allows to unfold the definition of b already in B, because $b\overline{\Sigma_1}\cdots\overline{\Sigma_m}$ can reduce to $\left(\lambda_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right)\overline{\Sigma_1}\cdots\overline{\Sigma_m}$, and we can β-reduce this term m times to $\left(\lambda_{i=m+1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right)[x_j{:=}\overline{\Sigma_j}]_{j=1}^m$. It is more in line with AUT-68 to make such unfolding not possible before *all* n arguments $\overline{\Sigma_1}, \dots, \overline{\Sigma_n}$ have been applied to b, so only when the construction of the equivalent of $b(\Sigma_1, \dots, \Sigma_n)$ has been completed;

- Moreover, $\lambda_{i=1}^n x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$ not necessarily has an equivalent in AUT-68. Consider for instance the constant b in the line

$$(\alpha{:}\mathtt{type}; b; [x{:}\alpha]x; [x{:}\alpha]\alpha).$$

[5]We can assume that the x_i do not occur in the Σ_j, so the simultaneous substitution $\Xi_1[x_1, \dots, x_n{:=}\Sigma_1, \dots, \Sigma_n]$ is equal to $\Xi_1[x_1{:=}\Sigma_1]\cdots[x_n{:=}\Sigma_n]$.

In this case, $\lambda_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1} \equiv \lambda\alpha{:}{*}.\lambda x{:}\alpha.x$. Its equivalent in AUT-68 would be $[\alpha{:}\mathtt{type}][x{:}\alpha]x$, but an abstraction $[\alpha{:}\mathtt{type}]$ cannot be made in AUT-68.[6] This is the reason why we do not incorporate $\lambda_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}$ as a citizen of $\lambda68$; we feel that this is better than making it a (first-class or second-class) citizen of $\lambda68$.

Therefore we choose a different translation. The line

$$(x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n; b; \Xi_1; \Xi_2),$$

where $\Xi_1 \in \mathcal{E}$, will be translated by putting

$$b{:=}\left(\mathop{\S}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right)$$

instead of

$$b{:=}\left(\mathop{\lambda}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_1}\right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}\overline{\alpha_i}.\overline{\Xi_2}\right)$$

in the left part of the translated context Δ. And a reduction rule

$$bX_1 \cdots X_n \to_{\delta} \overline{\Xi_1}[x_1, \ldots, x_n{:=}X_1, \ldots, X_n]$$

is added for all terms $X_1, \ldots, X_n$. The symbol § is used instead of λ. This is to emphasise that, though both $\S x{:}A$ and $\lambda x{:}A$ are abstractions, they are not the same kind of abstraction.

5c $\lambda68$

5c1 Definition and elementary properties

We give the formal definition of $\lambda68$, based on the motivation in Section 5b.

Definition 5.21 ($\lambda68$)

[6]This situation can be compared to the situation in Section 5b1, where we found that the type of b is not necessarily a first-class citizen of AUT-68. There, we could not avoid that the type of b became a citizen of $\lambda68$ (though we made it a second-class citizen by storing it in the sort Δ).

1. The terms of $\lambda 68$ form a set $\mathcal{T}$ defined by

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

   where $\boldsymbol{S}$ is the set of sorts $\{*, \square, \triangle\}$.

   We also define the sets of free variables $\textsc{fv}(T)$ and ("free")[7] constants $\textsc{fc}(T)$ of a term T in the straightforward way;

2. We define the notion of context inductively:

   - $\varnothing;\varnothing$ is a context; $\textsc{dom}(\varnothing;\varnothing) = \varnothing$;
   - If $\Delta;\Gamma$ is a context, $x \in \mathcal{V}$, x does not occur in $\Delta;\Gamma$ and $A \in \mathcal{T}$, then $\Delta;\Gamma, x{:}A$ is a context (x is a newly introduced variable); $\textsc{dom}(\Delta;\Gamma) = \textsc{dom}(\Delta;\Gamma) \cup \{x\}$;
   - If $\Delta;\Gamma$ is a context, $b \in \mathcal{C}$, b does not occur in $\Delta;\Gamma$ and $A \in \mathcal{T}$ then $\Delta, b{:}A;\Gamma$ is a context (in this case b is a *primitive* constant; cf. the primitive notions of AUTOMATH in Section 5a1); $\textsc{dom}(\Delta, b{:}A;\Gamma) = \textsc{dom}(\Delta;\Gamma) \cup \{b\}$;
   - If $\Delta;\Gamma$ is a context, $b \in \mathcal{C}$, b does not occur in $\Delta;\Gamma$, $A \in \mathcal{T}$, and $T \in \mathcal{T}$, then $\Delta, b{:=}T{:}A;\Gamma$ is a context (in this case b is a *defined* constant; cf. the definitions of AUTOMATH in Section 5a1); $\textsc{dom}(\Delta, b{:=}T{:}A;\Gamma) = \textsc{dom}(\Delta;\Gamma) \cup \{b\}$.

   Observe that a semicolon is used as the separation mark between the two parts of the context, and that a comma is used to separate the different expressions within each of these parts.

   We define

   $$\begin{aligned} \textsc{primcons}(\Delta;\Gamma) &= \{b \in \textsc{dom}(\Delta;\Gamma) \mid b \text{ is a primitive constant}\}; \\ \textsc{defcons}(\Delta;\Gamma) &= \{b \in \textsc{dom}(\Delta;\Gamma) \mid b \text{ is a defined constant}\}; \\ \textsc{fv}(\Delta;\Gamma) &= \textsc{dom}(;\Gamma); \end{aligned}$$

3. We define the notion of δ-reduction on terms. Let Δ be the left part of a context. If $(b{:=}(\S_{i=1}^{n} x_i{:}A_i.T) : (\P_{i=1}^{n} x_i{:}A_i.B)) \in \Delta$, where B is not of the form $\P y{:}B_1.B_2$, then

   $$\Delta \vdash bX_1 \cdots X_n \to_\delta T[x_1, \ldots, x_n{:=}X_1, \ldots, X_n]$$

[7]Of course, to call a constant "free" is a bit peculiar, since there are no *bound* constants.

for all $X_1, \ldots X_n \in \mathcal{T}$.

We also have the usual compatibility rules on δ-reduction. We use notations like $\twoheadrightarrow_\delta, \twoheadrightarrow_\delta^+, =_\delta$ as usual. When there is no confusion about which Δ is considered, we simply write

$$bX_1 \cdots X_n \rightarrow_\delta T[x_1, \ldots, x_n{:=}X_1, \ldots, X_n];$$

4. We use the usual notion of β-reduction;

5. Judgements in $\lambda 68$ have the form $\Delta; \Gamma \vdash A : B$, where $\Delta; \Gamma$ is a context and A and B are terms. In the case that a judgement $\Delta; \Gamma \vdash A : B$ is derivable according to the rules below, $\Delta; \Gamma$ is a *legal* context and A and B are *legal* terms. We write $\Delta; \Gamma \vdash A : B : C$ if both $\Delta; \Gamma \vdash A : B$ and $\Delta; \Gamma \vdash B : C$ are derivable in $\lambda 68$.

   Here are the rules for $\lambda 68$ (v, pc, and dc are shorthand for variable, primitive constant, and defined constant, respectively):

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

(Start: v) $$\frac{\Delta; \Gamma \vdash A : s}{\Delta; \Gamma, x{:}A \vdash x : A} \quad \text{where } s \equiv *, \square$$

(Start: pc) $$\frac{\Delta; \Gamma \vdash B : s_1 \qquad \Delta; \vdash \P\,\Gamma.B : s_2}{\Delta, b{:}\,\P\,\Gamma.B; \vdash b : \P\,\Gamma.B} \quad \text{where } s_1 \equiv *, \square$$

(Start: dc) $$\frac{\Delta; \Gamma \vdash T : B : s_1 \qquad \Delta; \vdash \P\,\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B); \vdash b : \P\,\Gamma.B} \quad \text{where } s_1 \equiv *, \square$$

(Weak: v) $$\frac{\Delta; \Gamma \vdash M : N \qquad \Delta; \Gamma \vdash A : s}{\Delta; \Gamma, x{:}A \vdash M : N} \quad \text{where } s \equiv *, \square$$

(Weak: pc) $$\frac{\Delta; \vdash M : N \qquad \Delta; \Gamma \vdash B : s_1 \qquad \Delta; \vdash \P\,\Gamma.B : s_2}{\Delta, b{:}\,\P\,\Gamma.B; \vdash M : N} \quad \text{where } s_1 \equiv *, \square$$

(Weak: dc)
$$\frac{\Delta; \vdash M : N \qquad \Delta; \Gamma \vdash T : B : s_1 \qquad \Delta; \vdash \P\,\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B); \vdash M : N}$$

where $s_1 \equiv *, \Box$

$$(\Pi\text{-form}) \quad \frac{\Delta;\Gamma \vdash A : * \qquad \Delta;\Gamma, x{:}A \vdash B : *}{\Delta;\Gamma \vdash (\Pi x{:}A.B) : *}$$

$$(\P\text{-form}) \quad \frac{\Delta;\Gamma \vdash A : s_1 \qquad \Delta;\Gamma, x{:}A \vdash B : s_2}{\Delta;\Gamma \vdash (\P x{:}A.B) : \Delta}$$

where $s_1 \equiv *, \Box$

$$(\lambda) \quad \frac{\Delta;\Gamma \vdash \Pi x{:}A.B : * \qquad \Delta;\Gamma, x{:}A \vdash F : B}{\Delta;\Gamma \vdash (\lambda x{:}A.F) : (\Pi x{:}A.B)}$$

$$(\mathrm{App}_1) \quad \frac{\Delta;\Gamma \vdash M : \Pi x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x{:=}N]}$$

$$(\mathrm{App}_2) \quad \frac{\Delta;\Gamma \vdash M : \P x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x{:=}N]}$$

$$(\mathrm{Conv}) \quad \frac{\Delta;\Gamma \vdash M : A \qquad \Delta;\Gamma \vdash B : s \qquad \Delta \vdash A =_{\beta\delta} B}{\Delta;\Gamma \vdash M : B}.$$

The newly introduced variables in the Start-rules and Weakening-rules are assumed to be fresh. Moreover, when introducing a variable x with a "pc"-rule or a "dc"-rule, we assume $x \in \mathcal{C}$, and when introducing x via a "v"-rule, we assume $x \in \mathcal{V}$.

We write $\Delta;\Gamma \vdash_{\lambda 68} A : B$ instead of $\Delta;\Gamma \vdash A : B$ if the latter gives rise to confusion with other derivation systems.

Notice that there is no rule (§). This is because we do not want that terms of the form $\S x{:}A.B$ are first-class citizens of λ68: they do not have an equivalent in AUTOMATH.

Many basic properties for Pure Type Systems also hold for λ68 and can be proved by the same methods as in the standard literature on PTSs. Due to the split of contexts and the different treatment of constants and variables, these properties are on some points differently formulated than usual (see Section Ad of the Appendix).

Lemma 5.22 (Free Variable Lemma) *Assume* $\Delta;\Gamma \vdash M : N$. *Write* $\Delta \equiv b_1{:}B_1, \ldots, b_m{:}B_m$; $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ *(in* Δ*, also expressions* $b_i{:=}T_i{:}B_i$ *may occur, but for uniformity of notation we leave out the* $:=T_i$*-part). Then:*

- *The* $b_1, \ldots, b_m \in \mathcal{C}$ *and* $x_1, \ldots, x_n \in \mathcal{V}$ *are all distinct;*

- $\mathrm{FC}(M), \mathrm{FC}(N) \subseteq \{b_1, \dots, b_m\}$; $\mathrm{FV}(M), \mathrm{FV}(N) \subseteq \{x_1, \dots, x_n\}$;
- $b_1{:}B_1, \dots, b_{i-1}{:}B_{i-1}; \vdash B_i{:}s_i$ *for some* $s_i \in \{*, \Box, \Delta\}$;
  $\Delta; x_1{:}A_1, \dots, x_{j-1}{:}A_{j-1} \vdash A_j{:}t_j$ *for some* $t_j \in \{*, \Box\}$.

$\boxtimes$

Lemma 5.23 (Start Lemma) *Let* $\Delta; \Gamma$ *be a legal context. Then* $\Delta; \Gamma \vdash * : \Box$, *and if* $b{:}A \in \Delta; \Gamma$, *or* $c{:=}T{:}A \in \Delta$, *then* $\Delta; \Gamma \vdash c : A$. $\boxtimes$

The following lemma is not a basic PTS-property. However, it can be seen as an extension of the Start Lemma.

Lemma 5.24 (Definition Lemma) *Assume*

$$\Delta_1, b{:=}\left(\mathop{\S}_{i=1}^{n} x_i{:}A_i.T\right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}A_i.B\right), \Delta_2; \Gamma \vdash M : N,$$

where B *is not of the form* $\P y{:}B_1.B_2$. *Then* $\Delta_1; x_1{:}A_1, \dots, x_n{:}A_n \vdash T : B : s$ *for an* $s \in \{*, \Box\}$. $\boxtimes$

The Transitivity Lemma must be formulated in a somewhat different way than usual (cf. A.24). This has to do with the fact that contexts may contain definitions. To the usual formulation

> *"Let* $\Delta_1; \Gamma_1$ *and* $\Delta_2; \Gamma_2$ *be contexts, of which* $\Delta_1; \Gamma_1$ *is legal. Assume that for all* $b{:}A \in \Delta_2; \Gamma_2$ *and for all* $b{:=}T{:}A \in \Delta_2; \Gamma_2$, $\Delta_1; \Gamma_1 \vdash b{:}A$. *Then* $\Delta_2; \Gamma_2 \vdash B : C \Rightarrow \Delta_1; \Gamma_1 \vdash B : C$."

we must add an extra clause that b is defined in $\Delta_1; \Gamma_1$ in a similar way as it has been defined in $\Delta_2; \Gamma_2$. In the following example we show that things go wrong otherwise:

Example 5.25 Let

$$\Delta_1 \equiv \mathsf{b}_1{:}*, \mathsf{b}_2{:}*, \mathsf{b}_3{:=}\mathsf{b}_1{:}*;$$
$$\Delta_2 \equiv \mathsf{b}_1{:}*, \mathsf{b}_2{:}*, \mathsf{b}_3{:=}\mathsf{b}_2{:}*$$

and $\Gamma_1 \equiv \Gamma_2 \equiv \mathsf{x}_3{:}\mathsf{b}_3$. Notice that all the assumptions of the traditional formulation of the Transitivity Lemma (see above) hold for $\Delta_1; \Gamma_1$ and $\Delta_2; \Gamma_2$. Nevertheless, we can derive

$$\Delta_2; \Gamma_2 \vdash \mathsf{x}_3 : \mathsf{b}_2$$

(because $\Delta_2; \Gamma_2 \vdash \mathsf{x}{:}\mathsf{b}_3$ and according to Δ_2, $\mathsf{b}_3 =_{\beta\mathrm{d}} \mathsf{b}_2$, so we can use the conversion rule). But we cannot derive

$$\Delta_1; \Gamma_1 \vdash \mathsf{x}_3 : \mathsf{b}_2$$

(because b_3 and b_2 are *not* definitionally equal according to Δ_1).

The following formulation of the Transitivity Lemma is correct:

Definition 5.26 We define: $\Delta_1; \Gamma_1 \vdash \Delta_2; \Gamma_2$ if and only if

- If $b{:}A \in \Delta_2; \Gamma_2$ then $\Delta_1; \Gamma_1 \vdash b{:}A$;
- If $b{:=}T{:}A \in \Delta_2$ then $\Delta_1; \Gamma_1 \vdash b{:}A$;
- If $b{:=}(\S_{i=1}^{n} x_i : A_i.U){:}B \in \Delta_2$ and $U \not\equiv \S\, y{:}B.A'$ then $\Delta_1 \vdash b x_1 \cdots x_n =_{\beta\delta} U$.

Lemma 5.27 (Transitivity Lemma)
Assume $\Delta_1; \Gamma_1 \vdash \Delta_2; \Gamma_2$ *and* $\Delta_2; \Gamma_2 \vdash B : C$. *Then* $\Delta_1; \Gamma_1 \vdash B : C$. ⊠

Lemma 5.28 (Substitution Lemma)
Assume $\Delta; \Gamma_1, x{:}A, \Gamma_2 \vdash B : C$ *and* $\Delta; \Gamma_1 \vdash D : A$.
Then $\Delta; \Gamma_1, \Gamma_2[x{:=}D] \vdash B[x{:=}D] : C[x{:=}D]$. ⊠

Lemma 5.29 (Thinning Lemma) *Let* $\Delta_1; \Gamma_1$ *be a legal context, and let* $\Delta_2; \Gamma_2$ *be a legal context such that* $\Delta_1 \subseteq \Delta_2$ *and* $\Gamma_1 \subseteq \Gamma_2$.
Then $\Delta_1; \Gamma_1 \vdash A : B \Rightarrow \Delta_2; \Gamma_2 \vdash A : B$. ⊠

Lemma 5.30 (Generation Lemma)

- *If* $x \in \mathcal{V}$ *and* $\Delta; \Gamma \vdash x{:}C$ *then there is* $s \in \{*, \square\}$ *and* $B =_{\beta\delta} C$ *such that* $\Delta; \Gamma \vdash B : s$ *and* $x{:}B \in \Gamma$;
- *If* $b \in \mathcal{C}$ *and* $\Delta; \Gamma \vdash b{:}C$ *then there is* $s \in \boldsymbol{S}$ *and* $B =_{\beta\delta} C$ *such that* $\Delta; \Gamma \vdash B : s$, *and either* $b{:}B \in \Delta$ *or there is* T *such that* $b{:=}T{:}B \in \Delta$;
- *If* $s \in \boldsymbol{S}$ *and* $\Delta; \Gamma \vdash s{:}C$ *then* $s \equiv *$ *and* $C =_{\beta\delta} \square$;
- *If* $\Delta; \Gamma \vdash MN : C$ *then there are* A, B *such that* $\Delta; \Gamma \vdash M : (\Pi x{:}A.B)$ *or* $\Delta; \Gamma \vdash M : (\P x{:}A.B)$, *and* $\Delta; \Gamma \vdash N{:}A$ *and* $C =_{\beta\delta} B[x{:=}N]$;

- *If* $\Delta;\Gamma\vdash(\lambda x{:}A.b):C$ *then there is* B *such that* $\Delta;\Gamma\vdash(\Pi x{:}A.B):*$, $\Delta;\Gamma,x{:}A\vdash b:B$ *and* $C=_{\beta\delta}\Pi x{:}A.B$;
- *Assume* $\Delta;\Gamma\vdash(\Pi x{:}A.B):C$.
  Then $C=_{\beta\delta}*$, $\Delta;\Gamma\vdash A{:}*$ *and* $\Delta;\Gamma,x{:}A\vdash B{:}*$;
- *If* $\Delta;\Gamma\vdash(\P x{:}A.B):C$ *then* $C=_{\beta\delta}\triangle$, $\Delta;\Gamma\vdash A{:}s_1$ *for some* $s_1\in\{*,\square\}$, *and* $\Delta;\Gamma,x{:}A\vdash B{:}s_2$ *for some* $s_2\in\{*,\square,\triangle\}$.

$\boxtimes$

Lemma 5.31 (Unicity of Types) *If* $\Delta;\Gamma\vdash A:B_1$ *and* $\Delta;\Gamma\vdash A:B_2$ *then* $B_1=_{\beta\delta}B_2$. $\boxtimes$

Lemma 5.32 (Correctness of Types) *If* $\Delta;\Gamma\vdash A:B$ *then there is* $s\in\boldsymbol{S}$ *such that* $B\equiv s$ *or* $\Delta;\Gamma\vdash B:s$. $\boxtimes$

From Correctness of Types and the Generation Lemma we conclude:

Lemma 5.33 *If* $\Delta;\Gamma\vdash A:(\Pi x{:}B_1.B_2)$ *then*

- $\Delta;\Gamma\vdash B_1:*$;
- $\Delta;\Gamma,x{:}B_1\vdash B_2:*$.

$\boxtimes$

Lemma 5.34 *If* $\Delta;\Gamma\vdash A:(\P x{:}B_1.B_2)$ *then*

- $\Delta;\Gamma\vdash B_1:s_1$ *for some* $s_1\in\{*,\square\}$;
- $\Delta;\Gamma,x{:}B_1\vdash B_2{:}s_2$ *for some sort* s_2.

$\boxtimes$

5c2 Reduction and conversion

In this section we show some properties of the reduction relations $\rightarrow_\beta$, $\rightarrow_\delta$ and $\rightarrow_{\beta\delta}$. As δ-reduction also depends on books, we first have to give a translation of AUT-68 books and AUT-contexts to λ68-contexts:

Definition 5.35 Let Γ be a AUT-68-context $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$. Then $\overline{\Gamma} \overset{\text{def}}{=} x_1{:}\overline{\alpha_1}, \ldots, x_n{:}\overline{\alpha_n}$.

Definition 5.36 Let $\mathfrak{B}$ be a book. We define the left part $\overline{\mathfrak{B}}$ of a context in $\lambda 68$:

- $\overline{\varnothing} \overset{\text{def}}{=} \varnothing$;
- $\overline{\mathfrak{B}, (\Gamma; b; \text{PN}; \Omega)} \overset{\text{def}}{=} \overline{\mathfrak{B}}, b{:}\,\P\,\overline{\Gamma}.\overline{\Omega}$;
- $\overline{\mathfrak{B}, (\Gamma; x; -; \Omega)} \overset{\text{def}}{=} \overline{\mathfrak{B}}$;
- $\overline{\mathfrak{B}, (\Gamma; b; \Sigma; \Omega)} \overset{\text{def}}{=} \overline{\mathfrak{B}}, b{:=}\,\S\,\overline{\Gamma}.\overline{\Sigma}{:}\,\P\,\overline{\Gamma}.\overline{\Omega}$.

Example 5.37 The translation of the book of Example 5.9 is given in Figure 9 (because of the habit in computer science to use more than one digit for a variable, we have to write some additional brackets around subterms like `proof` to keep things unambiguous). We see that all variable declarations of the original book have disappeared in the translation. In the original book, they do not add any new knowledge but are only used to construct contexts. In our translation, this happens in the right part of the context, instead of the left part.

Lemma 5.38 *Assume, Σ is a correct expression with respect to a book $\mathfrak{B}$.*

1. $\Sigma \rightarrow_\beta \Sigma'$ *if and only if* $\overline{\Sigma} \rightarrow_\beta \overline{\Sigma'}$;
2. $\mathfrak{B} \vdash \Sigma \rightarrow_\delta \Sigma'$ *if and only if* $\overline{\mathfrak{B}} \vdash \overline{\Sigma} \rightarrow_\delta \overline{\Sigma'}$.

PROOF: An easy induction on the structure of Σ. ⊠

The Church-Rosser property of $\rightarrow_{\beta\delta}$ will be proved by the method of Parallel Reduction, invented by Martin-Löf and Tait (see Section 3.2 of [4]).

Definition 5.39 Let Δ be the left part of a context. We define a reduction relation $\Rightarrow_{\beta\delta}$ ("parallel reduction") on the set of terms $\mathcal{T}$:

- For $x \in \mathcal{V}$, $\Delta \vdash x \Rightarrow_{\beta\delta} x$;

```
   prop   :   *,
    and   :   ¶x:prop.¶y:prop.prop,
  proof   :   ¶x:prop.*,
  and-I   :   ¶x:prop.¶y:prop.¶px:(proof)x.¶py:(proof)y.(proof)((and)xy),
 and-01   :   ¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)x,
 and-02   :   ¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)y,
  and-R  :=   §x:prop.§prx:(proof)x.(and-I)xx(prx)(prx)
          :   ¶x:prop.¶prx:(proof)x.(proof)((and)xx),
  and-S  :=   §x:prop.§y:prop.§pxy:(proof)((and)xy).
              (and-I)yx((and-02)xy(pxy))((and-01)xy(pxy))
          :   ¶x:prop.¶y:prop.¶pxy:(proof)((and)xy).(proof)((and)yx)
```

Figure 9: Translation of Example 5.9

- For $b \in \mathcal{C}$, $\Delta \vdash b \Rightarrow_{\beta\delta} b$;
- For $s \in \boldsymbol{S}$, $\Delta \vdash s \Rightarrow_{\beta\delta} s$;
- If $\Delta \vdash P \Rightarrow_{\beta\delta} P'$ and $\Delta \vdash Q \Rightarrow_{\beta\delta} Q'$, then
  - $\Delta \vdash \lambda x{:}P.Q \Rightarrow_{\beta\delta} \lambda x{:}P'.Q'$;
  - $\Delta \vdash \Pi x{:}P.Q \Rightarrow_{\beta\delta} \Pi x{:}P'.Q'$;
  - $\Delta \vdash \P x{:}P.Q \Rightarrow_{\beta\delta} \P x{:}P'.Q'$;
  - $\Delta \vdash PQ \Rightarrow_{\beta\delta} P'Q'$;
- If $\Delta \vdash Q \Rightarrow_{\beta\delta} Q'$ and $\Delta \vdash R \Rightarrow_{\beta\delta} R'$, then $\Delta \vdash (\lambda x{:}P.Q)R \Rightarrow_{\beta\delta} Q'[x{:=}R']$;
- If $b{:=}(\S_{i=1}^{n} x_i{:}A_i.T){:}(\P_{i=1}^{n} x_i{:}A_i.U) \in \Delta$, the term T is not of the form $\S y{:}T_1.T_2$, $\Delta \vdash T \Rightarrow_{\beta\delta} T'$ and $\Delta \vdash M_i \Rightarrow_{\beta\delta} M_i'$ for $i = 1, \ldots, n$, then $\Delta \vdash bM_1 \cdots M_n \Rightarrow_{\beta\delta} T'[x_1, \ldots, x_n{:=}M_1', \ldots, M_n']$.

Some elementary properties of $\Rightarrow_{\beta\delta}$ are:

Lemma 5.40 (Properties of $\Rightarrow_{\beta\delta}$) *Let Δ be the left part of a context. For all terms M, N:*

1. *$\Delta \vdash M \Rightarrow_{\beta\delta} M$;*
2. *If $\Delta \vdash M \to_{\beta\delta} M'$ then $\Delta \vdash M \Rightarrow_{\beta\delta} M'$;*
3. *If $\Delta \vdash M \Rightarrow_{\beta\delta} M'$ then $\Delta \vdash M \twoheadrightarrow_{\beta\delta} M'$.*

PROOF: All proofs can be given by induction on the structure of M. ⊠

We conclude from this lemma that $\twoheadrightarrow_{\beta\delta}$ (the reflexive and transitive closure of $\to_{\beta\delta}$) in the context Δ is the same relation as the reflexive and transitive closure of $\Rightarrow_{\beta\delta}$ in Δ. Therefore, if we want to prove the Church-Rosser theorem for $\twoheadrightarrow_{\beta\delta}$, it suffices to prove the Diamond Property for $\Rightarrow_{\beta\delta}$. We first make some preliminary definitions and remarks:

Lemma 5.41 (Substitution and $\Rightarrow_{\beta\delta}$) *If $\Delta \vdash M \Rightarrow_{\beta\delta} M'$ and $\Delta \vdash N \Rightarrow_{\beta\delta} N'$ then $\Delta \vdash M[y{:=}N] \Rightarrow_{\beta\delta} M'[y{:=}N']$.*

PROOF: Induction on the structure of M. ⊠

Lemma 5.42 *Assume, Δ and Δ,Δ' are left parts of legal contexts, and* $\textsc{fc}(M) \subseteq \textsc{dom}(\Delta)$. *Then $\Delta \vdash M \Rightarrow_{\beta\delta} N$ if and only if $\Delta,\Delta' \vdash M \Rightarrow_{\beta\delta} N$.*

PROOF: By induction on the length of Δ and by induction on the definition of $\Delta \vdash M \Rightarrow_{\beta\delta} N$. All cases in the definition of $\Delta \vdash M \Rightarrow_{\beta\delta} N$ follow directly from the induction hypothesis for $\Delta \vdash M \Rightarrow_{\beta\delta} N$, except for the case $bM_1 \cdots M_n \Rightarrow_{\beta\delta} T'[x_1,\ldots,x_n{:=}M_1',\ldots,M_n']$.

As $\textsc{fc}(M) \subseteq \textsc{dom}(\Delta)$, we have $b \in \textsc{dom}(\Delta)$.

Write $\Delta \equiv \Delta_1, b{:=}(\S_{i=1}^n\, x_i{:}A_i.T){:}(\P_{i=1}^n\, x_i{:}A_i.U), \Delta_2$.

- Notice that T is typable in $\Delta_1; x_1{:}A_1,\ldots,x_n{:}A_n$ (Definition Lemma). By the Free Variable Lemma: $\textsc{fc}(T) \subseteq \textsc{dom}(\Delta_1)$. By the induction hypothesis on the length of Δ we have $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T'$ iff $\Delta \vdash T \Rightarrow_{\beta\delta} T'$, and $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T'$ iff $\Delta,\Delta' \vdash T \Rightarrow_{\beta\delta} T'$;
- We conclude: $\Delta \vdash T \Rightarrow_{\beta\delta} T'$ iff $\Delta,\Delta' \vdash T \Rightarrow_{\beta\delta} T'$;
- By the induction hypothesis on the definition of $\Delta \vdash M \Rightarrow_{\beta\delta} N$, we have $\Delta \vdash M_i \Rightarrow_{\beta\delta} M_i'$ iff $\Delta,\Delta' \vdash M_i \Rightarrow_{\beta\delta} M_i'$;
- Notice that $b{:=}(\S_{i=1}^n\, x_i{:}A_i.T){:}(\P_{i=1}^n\, x_i{:}A_i.U)$ is an element of both Δ and Δ,Δ'. Moreover, $b \notin \textsc{dom}(\Delta')$ (because Δ,Δ' is the left part of a *legal* context). Therefore we have that $\Delta \vdash bM_1 \cdots M_n \Rightarrow_{\beta\delta} N$ if and only if $\Delta,\Delta' \vdash bM_1 \cdots M_n \Rightarrow_{\beta\delta} N$.

⊠

For left parts Δ of contexts and for $M \in \mathcal{T}$ with $\textsc{fc}(M) \subseteq \textsc{dom}(\Delta)$, we define a term M^Δ. In M^Δ, all β-redexes that exist in M are contracted simultaneously (this is a usual step in a proof of Church-Rosser by Parallel Reduction), but also all δ-redexes are contracted. We will show that $\Delta \vdash N \Rightarrow_{\beta\delta} M^\Delta$ for any N with $\Delta \vdash M \Rightarrow_{\beta\delta} N$; so M^Δ helps us to show the Diamond Property for $\Rightarrow_{\beta\delta}$.

Definition 5.43 We define, for any left part Δ of a context and any $M \in \mathcal{T}$ such that $\textsc{fc}(M) \subseteq \textsc{dom}(\Delta)$, M^Δ. The definition of M^Δ is by induction on the length of Δ. So assume $M^{\Delta'}$ has been defined for contexts Δ' that are shorter than Δ. We use induction on the structure of M:

- $x^\Delta \stackrel{\text{def}}{=} x$ for any $x \in \mathcal{V}$;

- $M \equiv b$. Distinguish:
  - $b^\Delta \overset{\text{def}}{=} b$ for any $b \in \textsc{primcons}(\Delta;)$;
  - $b^\Delta \overset{\text{def}}{=} b$ for any $b \in \textsc{defcons}(\Delta;)$ that is not a δ-redex;
  - If $b \in \textsc{defcons}(\Delta;)$ is a δ-redex, then $\Delta \equiv \Delta_1, b{:=}T{:}U, \Delta_2$, where $T \not\equiv \S y{:}T_1.T_2$. By the Definition Lemma, $\Delta_1; \vdash T : U$, so we can assume that T^{Δ_1} has already been defined. Then $b^\Delta \overset{\text{def}}{=} T^{\Delta_1}$;
- $s^\Delta \overset{\text{def}}{=} s$ for any $s \in \boldsymbol{S}$;
- $(\lambda x{:}P.Q)^\Delta \overset{\text{def}}{=} \lambda x{:}P^\Delta.Q^\Delta$;

  $(\Pi x{:}P.Q)^\Delta \overset{\text{def}}{=} \Pi x{:}P^\Delta.Q^\Delta$;

  $(\P x{:}P.Q)^\Delta \overset{\text{def}}{=} \P x{:}P^\Delta.Q^\Delta$;
- M is an application term. We distinguish three possibilities:
  - $M \equiv PQ$ is not a $\beta\delta$-redex. Then we define $M^\Delta \overset{\text{def}}{=} P^\Delta Q^\Delta$;
  - M is a β-redex $(\lambda x{:}P.Q)R$. We define $M^\Delta \overset{\text{def}}{=} Q^\Delta[x{:=}R^\Delta]$;
  - M is a δ-redex $bM_1 \cdots M_n$, and

    $$\Delta \equiv \Delta_1, b{:=}\left(\underset{i=1}{\overset{n}{\S}} x_i{:}A_i.T\right) : \left(\underset{i=1}{\overset{n}{\P}} x_i{:}A_i.U\right), \Delta_2,$$

    where T is not of the form $\S y{:}T_1.T_2$. In that case

    $$\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n \vdash T : U$$

    (by the Definition Lemma), so we can assume that T^{Δ_1} has already been defined.

    Then $M^\Delta \overset{\text{def}}{=} T^{\Delta_1}[x_1, \ldots, x_n{:=}M_1^\Delta, \ldots, M_n^\Delta]$.

Lemma 5.44 *Let Δ be the left part of a legal context. $\Delta \vdash M \Rightarrow_{\beta\delta} M^\Delta$ for all M with* $\textsc{fc}(M) \subseteq \textsc{dom}(\Delta)$.

PROOF: By induction on the definition of M^Δ. We only treat the case $\Delta \vdash bM_1 \cdots M_n \Rightarrow_{\beta\delta} (bM_1 \cdots M_n)^\Delta$ where $bM_1 \cdots M_n$ is a δ-redex. As in the definition of $(bM_1 \cdots M_n)^\Delta$, write

$$\Delta \equiv \Delta_1, b{:=} \left(\mathop{\S}_{i=1}^{n} x_i{:}A_i.T \right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}A_i.U \right), \Delta_2.$$

By induction, we may assume that $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$ and $\Delta \vdash M_i \Rightarrow_{\beta\delta} M_i^\Delta$. By the Definition Lemma, T is typable in $\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n$, so by the Free Variable Lemma, $\text{FC}(T) \subseteq \text{DOM}(\Delta_1)$. By Lemma 5.42, $\Delta \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$. So $\Delta \vdash bM_1 \cdots M_n \Rightarrow_{\beta\delta} T^{\Delta_1}[x_1, \ldots, x_n{:=}M_1^\Delta, \ldots, M_n^\Delta]$. ⊠

Theorem 5.45 *Let Δ be the left part of a legal context. Assume* $\text{FC}(M) \subseteq \text{DOM}(\Delta)$. *If $\Delta \vdash M \Rightarrow_{\beta\delta} N$ then $\Delta \vdash N \Rightarrow_{\beta\delta} M^\Delta$.*

PROOF: Induction on the the definition of M^Δ.

- $M \equiv x$. Then $N \equiv x$ and $M^\Delta \equiv x$;
- $M \equiv b$. Distinguish:
  - $b \in \text{PRIMCONS}(\Delta;)$. Then $N \equiv b$ and $M^\Delta \equiv b$;
  - $b \in \text{DEFCONS}(\Delta;)$, but b is not a δ-redex. Then $N \equiv b$ and $M^\Delta \equiv b$;
  - $b \in \text{DEFCONS}(\Delta;)$, and $\Delta \equiv \Delta_1, b{:=}T{:}U, \Delta_2$, and $T \not\equiv \S y{:}T_1.T_2$. Then either $N \equiv b$ or $N \equiv T'$ where $T \Rightarrow_{\beta\delta} T'$. If $N \equiv b$ then $M \equiv N$ and we can use Lemma 5.44. If $N \equiv T$ then observe that by the induction hypothesis, $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$, that by Lemma 5.42 $\Delta \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$, and that $M^\Delta \equiv T^{\Delta_1}$;
- $M \equiv s$. Then $N \equiv s$ and $M^\Delta \equiv s$;
- $M \equiv \lambda x{:}P.Q$. Then $N \equiv \lambda x{:}P'.Q'$ for some P', Q' with $\Delta \vdash P \Rightarrow_{\beta\delta} P'$ and $\Delta \vdash Q \Rightarrow_{\beta\delta} Q'$. By the induction hypothesis on P and Q we find $\Delta \vdash P' \Rightarrow_{\beta\delta} P^\Delta$ and $\Delta \vdash Q' \Rightarrow_{\beta\delta} Q^\Delta$. Therefore $\Delta \vdash \lambda x{:}P'.Q' \Rightarrow_{\beta\delta} \lambda x{:}P^\Delta.Q^\Delta$.

  The cases $M \equiv \Pi x{:}P.Q$, $M \equiv \P x{:}P.Q$, and $M \equiv PQ$ where PQ is not a $\beta\delta$-redex, are proved similarly;
- M is an application term (and is either a β or a δ-redex). Distinguish:
  - M is a β-redex, $M \equiv (\lambda x{:}P.Q)R$. Distinguish:

* $N \equiv (\lambda x{:}P'.Q')R'$ for P', Q', R' with $\Delta \vdash P \Rightarrow_{\beta\delta} P'$, $\Delta \vdash Q \Rightarrow_{\beta\delta} Q'$ and $\Delta \vdash R \Rightarrow_{\beta\delta} R'$. By induction, $\Delta \vdash Q' \Rightarrow_{\beta\delta} Q^{\Delta}$ and $\Delta \vdash R' \Rightarrow_{\beta\delta} R^{\Delta}$. Therefore $\Delta \vdash N \Rightarrow_{\beta\delta} Q^{\Delta}[x{:=}R^{\Delta}]$;
* $N \equiv Q'[x{:=}R']$ for Q', R' with $\Delta \vdash Q \Rightarrow_{\beta\delta} Q'$ and $\Delta \vdash R \Rightarrow_{\beta\delta} R'$. By induction, $\Delta \vdash Q' \Rightarrow_{\beta\delta} Q^{\Delta}$ and $\Delta \vdash R' \Rightarrow_{\beta\delta} R^{\Delta}$. By Lemma 5.41, $\Delta \vdash Q'[x{:=}R'] \Rightarrow_{\beta\delta} Q^{\Delta}[x{:=}R^{\Delta}]$;

– M is a δ-redex, $M \equiv bM_1 \cdots M_n$,

$$\Delta \equiv \Delta_1, b{:=}\left(\mathop{\S}_{i=1}^{n} x_i{:}A_i.T\right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}A_i.U\right), \Delta_2,$$

where $T \not\equiv \S y{:}T_1.T_2$. Distinguish:

* $N \equiv bM_1' \cdots M_n'$ for M_i' with $\Delta \vdash M_i \Rightarrow_{\beta\delta} M_i'$. By induction, we have $\Delta \vdash M_i' \Rightarrow_{\beta\delta} M_i^{\Delta}$. By the Definition Lemma, T is typable in a context $\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n$, so by the Free Variable Lemma, $\text{FC}(T) \subseteq \text{DOM}(\Delta_1)$. By Lemma 5.44, $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$. By Lemma 5.42, $\Delta \vdash T \Rightarrow_{\beta\delta} T^{\Delta_1}$. Hence $\Delta \vdash N \Rightarrow_{\beta\delta} T^{\Delta_1}[x_1, \ldots, x_n{:=}M_1^{\Delta}, \ldots, M_n^{\Delta}]$;
* $N \equiv T'[x_1, \ldots, x_n{:=}M_1', \ldots, M_n']$ for a T' with $\Delta \vdash T \Rightarrow_{\beta\delta} T'$ and for M_i' with $\Delta \vdash M_i \Rightarrow_{\beta\delta} M_i'$. By the Definition Lemma, T is typable in $\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n$, so by the Free Variable Lemma, $\text{FC}(T) \subseteq \text{DOM}(\Delta_1)$. By Lemma 5.42, $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T'$. By the induction hypothesis on T, $\Delta_1 \vdash T' \Rightarrow_{\beta\delta} T^{\Delta_1}$. As $\Delta_1 \vdash T \Rightarrow_{\beta\delta} T'$, $\text{FC}(T') \subseteq \text{DOM}(\Delta_1)$, so by Lemma 5.42, $\Delta \vdash T' \Rightarrow_{\beta\delta} T^{\Delta_1}$. By the induction hypothesis, also $\Delta \vdash M_i' \Rightarrow_{\beta\delta} M_i^{\Delta}$. Repeatedly applying Lemma 5.41, we find[8]

$$\Delta \vdash T'[x_1, \ldots, x_n{:=}M_1', \ldots M_n'] \Rightarrow_{\beta\delta} T^{\Delta_1}[x_1, \ldots, x_n{:=}M_1^{\Delta}, \ldots, M_n^{\Delta}].$$

$\boxtimes$

[8]We must remark that

$$T'[x_1, \ldots, x_n{:=}M_1', \ldots, M_n'] \equiv T'[x_1{:=}M_1'] \cdots [x_n{:=}M_n']$$

and

$$T^{\Delta_1}[x_1, \ldots, x_n{:=}M_1^{\Delta}, \ldots, M_n^{\Delta}] \equiv T^{\Delta_1}[x_1{:=}M_1^{\Delta}] \cdots [x_n{:=}M_n^{\Delta}].$$

This is correct as we can assume that the x_i do not occur in the M_j' and M_j^{Δ}.

Corollary 5.46 (Diamond Property for $\Rightarrow_{\beta\delta}$) *Let Δ be the left part of a context in which M is typable.*
Assume $\Delta \vdash M \Rightarrow_{\beta\delta} N_1$ and $\Delta \vdash M \Rightarrow_{\beta\delta} N_2$. Then there is P such that $\Delta \vdash N_1 \Rightarrow_{\beta\delta} P$ and $\Delta \vdash N_2 \Rightarrow_{\beta\delta} P$.

PROOF: Immediately from the theorem above: Take $P \equiv M^{\Delta}$. ⊠

Corollary 5.47 (Church-Rosser property for $\rightarrow_{\beta\delta}$) *Let Δ be the left part of a context in which M is typable.*
If $\Delta \vdash M \twoheadrightarrow_{\beta\delta} N_1$ and $\Delta \vdash M \twoheadrightarrow_{\beta\delta} N_2$ then there is P such that $\Delta \vdash N_1 \twoheadrightarrow_{\beta\delta} P$ and $\Delta \vdash N_2 \twoheadrightarrow_{\beta\delta} P$.

PROOF: Directly from Lemma 5.40.2, Lemma 5.40.3 and Corollary 5.46. ⊠

5c3 Subject reduction

Lemma 5.48 (Subject Reduction)
If $\Delta;\Gamma \vdash A : B$ and $A \rightarrow_\beta A'$ then $\Delta;\Gamma \vdash A' : B$.

PROOF: The proof is as in [5]. ⊠

Subject Reduction also holds for the reduction relation $\rightarrow_\delta$:

Lemma 5.49 (Subject Reduction for $\rightarrow_\delta$)
If $\Delta;\Gamma \vdash A : B$ and $A \rightarrow_\delta A'$ then $\Delta;\Gamma \vdash A' : B$.

PROOF: Following the line of [5], we define $\Delta;\Gamma \rightarrow_\delta \Delta;\Gamma'$ if $\Gamma \equiv \Gamma_1, x{:}A, \Gamma_2$, and $\Gamma' \equiv \Gamma_1, x{:}A', \Gamma_2$, and $\Delta \vdash A \rightarrow_\delta A'$. We define $\Delta;\Gamma \rightarrow_\delta \Delta';\Gamma$ similarly, and we simultaneously prove

$$\begin{array}{rcl}
\Delta;\Gamma \vdash A{:}B \text{ and } \Delta \vdash A \rightarrow_\delta A' & \Rightarrow & \Delta;\Gamma \vdash A'{:}B \\
\Delta;\Gamma \vdash A{:}B \text{ and } \Delta;\Gamma \rightarrow_\delta \Delta';\Gamma & \Rightarrow & \Delta';\Gamma \vdash A{:}B \\
\Delta;\Gamma \vdash A{:}B \text{ and } \Delta;\Gamma \rightarrow_\delta \Delta;\Gamma' & \Rightarrow & \Delta;\Gamma' \vdash A{:}B,
\end{array}$$

using induction on the derivation of $\Delta;\Gamma \vdash A{:}B$. We only treat the case in which the last applied rule is the 2nd application rule, and we only prove the first of the three statements for this case. We write $A[x_i{:=}B_i]_{i=m}^{n}$ as a shorthand for $A[x_m{:=}B_m][x_{m+1}{:=}B_{m+1}]\cdots[x_n{:=}B_n]$. We can assume that

$$\Delta \equiv \Delta_1, b{:=} \left(\mathop{\S}_{i=1}^{n} x_i{:}A_i.T\right) : \left(\mathop{\P}_{i=1}^{n} x_i{:}A_i.B\right), \Delta_2 \tag{1}$$

with $B \not\equiv \P y{:}B_1.B_2$, and that the conclusion of the 2nd application rule is

$$\Delta;\Gamma \vdash bM_1\cdots M_n : K_n \tag{2}$$

for some K_n, and therefore

$$\Delta \vdash bM_1\cdots M_n \to_\delta T[x_i{:=}M_i]_{i=1}^{n}.$$

We must prove: $\Delta;\Gamma \vdash T[x_i{:=}M_i]_{i=1}^{n} : K_n$. We do this in two steps.

1. We analyse the structure of K_n, and derive that

$$\Delta \vdash K_n =_{\beta\delta} B[x_i{:=}M_i]_{i=1}^{n};$$

2. We show that $\Delta;\Gamma \vdash T[x_i{:=}M_i]_{i=1}^{n} : B[x_i{:=}M_i]_{i=1}^{n}$.

Ad 1.
We repeatedly apply the Generation Lemma, starting with (2), thus obtaining $K_n, K_{n-1}, \ldots, K_1$, $K'_n, K'_{n-1}, \ldots, K'_1$, $L_n, L_{n-1}, \ldots, L_1$ such that

$$\Delta;\Gamma \vdash bM_1\cdots M_{i-1} : (\P x_i{:}L_i.K'_i); \tag{3}$$

$$\Delta;\Gamma \vdash M_i : L_i; \tag{4}$$

$$\Delta \vdash K_i =_{\beta\delta} K'_i[x_i{:=}M_i]; \tag{5}$$

$$\Delta \vdash K_{i-1} =_{\beta\delta} \P x_i{:}L_i.K'_i. \tag{6}$$

We end with $\Delta;\Gamma \vdash b : (\P x_1{:}L_1.K'_1)$. By (1) and the Generation Lemma:

$$\Delta \vdash \P x_1{:}L_1.K'_1 =_{\beta\delta} \mathop{\P}_{j=1}^{n} x_j{:}A_j.B.$$

By the Church-Rosser Theorem we have $L_1 =_{\beta\delta} A_1$ and

$$\Delta \vdash K_1' =_{\beta\delta} \prod_{j=2}^{n} x_j{:}A_j.B. \tag{7}$$

Hence

$$\begin{aligned}\Delta \vdash \P x_2{:}L_2.K_2' &\overset{(6)}{=}_{\beta\delta} K_1\\ &\overset{(5,7)}{=}_{\beta\delta} \left(\prod_{j=2}^{n} x_j{:}A_j.B\right)[x_1{:=}M_1]\\ &\equiv \prod_{i=2}^{n} x_i{:}A_i[x_1{:=}M_1].B[x_1{:=}M_1],\end{aligned}$$

so by the Church-Rosser Theorem $L_2 =_{\beta\delta} A_2[x_1{:=}M_1]$. Proceeding in this way, we obtain for $i = 1, \ldots, n$:

$$\begin{aligned}\Delta \vdash L_i &=_{\beta\delta} A_i[x_j{:=}M_j]_{j=1}^{i-1}; && (8)\\ \Delta \vdash K_i' &=_{\beta\delta} \prod_{j=i+1}^{n} x_j{:}A_j[x_k{:=}M_k]_{k=1}^{i-1}.B[x_k{:=}M_k]_{k=1}^{i-1};\\ \Delta \vdash K_i &=_{\beta\delta} \prod_{j=i+1}^{n} x_j{:}A_j[x_k{:=}M_k]_{k=1}^{i}.B[x_k{:=}M_k]_{k=1}^{i}.\end{aligned}$$

In particular,

$$\Delta \vdash K_n =_{\beta\delta} B[x_i{:=}M_i]_{i=1}^{n}. \tag{9}$$

Ad 2.
Now we calculate the type of $T[x_i{:=}M_i]_{i=1}^{n}$. By the Definition Lemma on (1) we also have

$$\Delta_1; x_1{:}A_1, \ldots, x_n{:}A_n \vdash T : B, \tag{10}$$

so by the Start Lemma: $\Delta_1; x_1{:}A_1, \ldots, x_{i-1}{:}A_{i-1} \vdash A_i{:}s_i$ for sorts $s_i \in \boldsymbol{S}$. This yields:

$$\begin{aligned}\Delta;\Gamma \vdash A_1 : s_1 &\qquad \text{(Thinning Lemma);}\\ \Delta;\Gamma, x_1{:}A_1 \text{ is legal} &\qquad \text{(Start Rule);}\\ \Delta;\Gamma, x_1{:}A_1 \vdash A_2 : s_2 &\qquad \text{(Thinning Lemma);}\\ \Delta;\Gamma, x_1{:}A_1, x_2{:}A_2 \text{ is legal} &\qquad \text{(Start Rule);}\\ &\vdots\\ \Delta;\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n \text{ is legal.} &\qquad \text{(Start Rule).}\end{aligned}$$

Therefore, we can apply the Thinning Lemma to (10), and we find:

$$\Delta;\Gamma,x_1{:}A_1,\ldots,x_n{:}A_n\vdash T:B.$$

As $\Delta;\Gamma\vdash M_1:L_1$ (4) and $\Delta;\Gamma\vdash A_1:s_1$, we have $\Delta;\Gamma\vdash M_1:A_1$ by the Conversion rule and (8), so by the Substitution Lemma:

$$\begin{array}{rcl}\Delta;\Gamma,x_2{:}A_2[x_1{:=}M_1],\ldots,x_n{:}A_n[x_1{:=}M_1] & \vdash & T[x_1{:=}M_1]:B[x_1{:=}M_1];\\ \Delta;\Gamma & \vdash & A_2[x_1{:=}M_1]:s_2.\end{array}$$

As $\Delta;\Gamma\vdash M_2:L_2$ (4) and $\Delta\vdash A_2[x_1{:=}M_1]=_{\beta\delta}L_2$ (8) we have by conversion $\Delta;\Gamma\vdash M_2:A_2[x_1{:=}M_1]$, and again by the Substitution Lemma:

$$\begin{array}{rcl}\Delta;\Gamma,x_3{:}A_3[x_i{:=}M_i]_{i=1}^2,\ldots,x_n{:}A_n[x_i{:=}M_i]_{i=1}^2 & & \\ & \vdash & T[x_i{:=}M_i]_{i=1}^2:B[x_i{:=}M_i]_{i=1}^2;\\ \Delta;\Gamma & \vdash & A_3[x_1{:=}M_1][x_2{:=}M_2]:s_3.\end{array}$$

Proceeding in this way we eventually find

$$\Delta;\Gamma\vdash T[x_i{:=}M_i]_{i=1}^n:B[x_i{:=}M_i]_{i=1}^n. \qquad (11)$$

Applying Lemma 5.32 to (9) we have $\Delta;\Gamma\vdash K_n:s$. Now use the Conversion Rule, (11), and the fact that $\Delta\vdash K_n=_{\beta\delta}B[x_i{:=}M_i]_{i=1}^n$. ⊠

Corollary 5.50 (Subject Reduction for $\to_{\beta\delta}$) *If $\Delta;\Gamma\vdash A:B$ and $A\twoheadrightarrow_{\beta\delta}A'$ then $\Delta;\Gamma\vdash A':B$.* ⊠

The Subject Reduction Theorem for $\to_\delta$ is used to prove:

Lemma 5.51 *Assume $s\in\boldsymbol{S}$ and M legal.*
Then $(\Delta\vdash M=_{\beta\delta}s)\Rightarrow M\equiv s$.

PROOF: First assume $s\in\{\Box,\triangle\}$. If $\Delta;\Gamma\vdash M:N$ for some Γ and N, and $\Delta\vdash M=_{\beta\delta}s$ then by Church-Rosser $\Delta\vdash M\twoheadrightarrow_{\beta\delta}s$, so by Subject Reduction $\Delta;\Gamma\vdash s:N$, contradicting the Generation Lemma. If $\Delta;\Gamma\vdash N:M$ and $\Delta\vdash M=_{\beta\delta}s$ and $M\not\equiv s$ then we have by Lemma 5.32 that $\Delta;\Gamma\vdash M:P$ for some P, so again $\Delta;\Gamma\vdash s:P$, in contradiction with the Generation Lemma.

Now assume $s\equiv *$, $\Delta;\Gamma\vdash M:N$, and $\Delta\vdash M=_{\beta\delta}s$. Again by Church-Rosser, $\Delta\vdash M\twoheadrightarrow_{\beta\delta}*$, say $\Delta\vdash M\to_{\beta\delta}\ldots\to_{\beta\delta}M'\to_{\beta\delta}*$. By Subject Reduction, $\Delta;\Gamma\vdash M':N$ and $\Delta;\Gamma\vdash *:N$. By the Generation Lemma $\Delta\vdash N=_{\beta\delta}\Box$, so $N\equiv\Box$. Distinguish:

- $M' \equiv (\lambda x{:}A.B)C$ and $* \equiv B[x{:=}C]$. By the Generation Lemma there is B' such that $\Delta \vdash B'[x{:=}C] =_{\beta\delta} \Box$ (hence $B'[x{:=}C] \equiv \Box$), $\Delta;\Gamma \vdash (\lambda x{:}A.B) : (\Pi x{:}A.B')$ and $\Delta;\Gamma \vdash C : A$. $C \equiv \Box$ contradicts $\Delta;\Gamma \vdash C : A$, so $B' \equiv \Box$. By Lemma 5.33 $\Delta;\Gamma \vdash (\Pi x{:}A.\Box) : *$, so by the Generation Lemma $\Delta;\Gamma, x{:}A \vdash \Box : *$, contradiction;
- $M' \equiv bM_1 \cdots M_n$ and $\Delta \vdash bM_1 \cdots M_n \to_\delta T[x_i{:=}M_i]_{i=1}^n \equiv *$. The argument is similar as in the case $M' \equiv (\lambda x{:}A.B)C$.

If $s \equiv *$, $\Delta;\Gamma \vdash N : M$, and $\Delta \vdash M =_{\beta\delta} s$ then by Lemma 5.32 $M \equiv s$ (and we are done) or $\Delta;\Gamma \vdash M : s'$ (which implies $M \equiv s$ by the above argument). ⊠

5c4 Strong normalisation

We prove Strong Normalisation for $\beta\delta$-reduction in $\lambda 68$ by mapping a typable term M (in a context $\Delta;\Gamma$) of $\lambda 68$ to a term $|M|_\Delta$ that is typable in a strongly normalising PTS. The mapping is constructed in such a way that if $M \to_\beta N$, $|M|_\Delta \twoheadrightarrow_\beta^+ |N|_\Delta$, and that if $\Delta \vdash M \to_\delta N$, $|M|_\Delta \twoheadrightarrow_\beta |N|_\Delta$.

Definition 5.52 Let Δ be the left part of a legal context and let $M \in \mathcal{T}$. We define $|M|_\Delta$ by induction on the length of Δ and the structure of M.

- $|x|_\Delta \stackrel{\text{def}}{=} x$ for $x \in \mathcal{V}$;
- $|b|_\Delta \stackrel{\text{def}}{=} b$ for all $b \in \mathcal{C} \setminus \text{DEFCONS}(\Delta;)$;
- $|b|_\Delta \stackrel{\text{def}}{=} \lambda_{i=1}^n x_i{:}\,|A_i|_{\Delta_1} \cdot |T|_{\Delta_1}$

  if $\Delta \equiv \Delta_1, b{:=}(\S_{i=1}^n\, x_i{:}A_i.T){:}(\P_{i=1}^n\, x_i{:}A_i.U), \Delta_2$;
- $|s|_\Delta \stackrel{\text{def}}{=} s$ for $s \in \mathcal{S}$;
- $|\lambda x{:}P.Q|_\Delta \stackrel{\text{def}}{=} \lambda x{:}\,|P|_\Delta \cdot |Q|_\Delta$;
- $|\Pi x{:}P.Q|_\Delta \stackrel{\text{def}}{=} \Pi x{:}\,|P|_\Delta \cdot |Q|_\Delta$;
- $|\P x{:}P.Q|_\Delta \stackrel{\text{def}}{=} \Pi x{:}\,|P|_\Delta \cdot |Q|_\Delta$;
- $|PQ|_\Delta \stackrel{\text{def}}{=} |P|_\Delta\, |Q|_\Delta$.

The following lemmas are useful:

Lemma 5.53 *Let Δ be the left part of a legal context and $M \in \mathcal{T}$. Then* $\text{FV}(|M|_\Delta) = \text{FV}(M)$.

PROOF: The proof is by induction on the definition of $|M|_\Delta$ and is trivial for all cases except the case $M \equiv b$ and $\Delta \equiv \Delta_1, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.U), \Delta_2$ $(T \not\equiv \S\, y{:}T_1.T_2)$.
By the Definition Lemma, T is typable in $\Delta_1;\Gamma$; therefore $\text{FV}(T) \subseteq \text{DOM}\,(\Gamma)$ (Free Variable Lemma). By the induction hypothesis, $\text{FV}\big(|T|_{\Delta_1}\big) \subseteq \text{DOM}\,(\Gamma)$ and therefore $\text{FV}(|b|_\Delta) = \varnothing$. ⊠

Lemma 5.54 *If Δ_1 and Δ_2 are left parts of legal contexts and $\Delta_2 \equiv \Delta_1, \Delta'$ then $|M|_{\Delta_2} \equiv |M|_{\Delta_1}$ for all $M \in \mathcal{T}$ with* $\text{FC}(M) \subseteq \text{DOM}\,(\Delta_1)$.

PROOF: An easy induction on the definition of $|M|_{\Delta_1}$. ⊠

Lemma 5.55 *Let Δ be the left part of a legal context. For all M, N:*

$$|M[x{:=}N]|_\Delta \equiv |M|_\Delta\,[x{:=}\,|N|_\Delta].$$

PROOF: By induction on the definition of $|M|_\Delta$. In the case $M \equiv b$ and $b{:=}T{:}U \in \Delta$, use the fact that $\text{FV}(|M|_\Delta) = \text{FV}(M) = \varnothing$ (Lemma 5.53) and therefore $|M|_\Delta\,[x{:=}\,|N|_\Delta] \equiv |M|_\Delta \equiv |M[x{:=}N]|_\Delta$. ⊠

The purpose of the definition of $|M|_\Delta$ is explained in the following two lemmas:

Lemma 5.56 *If $M \to_\beta N$ then $|M|_\Delta \twoheadrightarrow^+_\beta |N|_\Delta$.*

PROOF: Induction on the structure of M. We only treat the case $M \equiv (\lambda x{:}P.Q)R$ and $N \equiv Q[x{:=}R]$.

$$\begin{array}{rcl} |M|_\Delta & \equiv & (\lambda x{:}\,|P|_\Delta\,.\,|Q|_\Delta)\,|R|_\Delta \\ & \to_\beta & |Q|_\Delta\,[x{:=}\,|R|_\Delta] \\ & \overset{5.55}{\equiv} & |Q[x{:=}R]|_\Delta\,. \end{array}$$

⊠

Lemma 5.57 *If $\Delta \vdash M \to_\delta N$, then $|M|_\Delta \twoheadrightarrow_\beta |N|_\Delta$.*

PROOF: Induction on the structure of M. We only treat the case in which

$$\begin{aligned}
&M \equiv bM_1 \cdots M_n;\\
&\Delta \equiv \Delta_1, b{:=}\left(\overset{n}{\underset{i=1}{\S}}\, x_i{:}A_i.T\right) : \left(\overset{n}{\underset{i=1}{\P}}\, x_i{:}A_i.U\right), \Delta_2;\\
&N \equiv T[x_1, \ldots, x_n{:=}M_1, \ldots, M_n].
\end{aligned}$$

Notice that

$$\begin{array}{rcl}
|M|_\Delta & \equiv & \left(\overset{n}{\underset{i=1}{\lambda}} x_i{:}\, |A_i|_{\Delta_1} . |T|_{\Delta_1}\right) |M_1|_\Delta \cdots |M_n|_\Delta \\
& \twoheadrightarrow_\beta & |T|_{\Delta_1} [x_i{:=}\, |M_i|_\Delta]_{i=1}^n \\
& \overset{5.54}{\equiv} & |T|_\Delta [x_i{:=}\, |M_i|_\Delta]_{i=1}^n \\
& \overset{5.55}{\equiv} & |T[x_i{:=}M_i]_{i=1}^n|_\Delta \\
& \equiv & |T[x_1, \ldots, x_n{:=}M_1, \ldots, M_n]|_\Delta .
\end{array}$$

At the last equivalence, we must make a remark similar to footnote 8 on page 197. ⊠

Let λSN be the PTS over λ-terms with variables from $\mathcal{V} \cup \mathcal{C}$ and sorts from $\boldsymbol{S}$, and the following rules (we choose the name λSN because this system will help us in showing that λ68 is SN):

$$\begin{array}{cc}
\multicolumn{2}{c}{(*, *, *);} \\
(*, *, \Delta); & (\Box, *, \Delta); \\
(*, \Box, \Delta); & (\Box, \Box, \Delta); \\
(*, \Delta, \Delta); & (\Box, \Delta, \Delta).
\end{array}$$

This is in fact the pure type system that is based on the Π-formation rules that were proposed in Section 5b1. λSN is contained in the system **ECC** (see [85]). As **ECC** is β-strongly normalising, also λSN is β-strongly normalising.

We present a translation of λ68-contexts to λSN-contexts:

Definition 5.58 Let $\Delta; \Gamma$ be a legal λ68-context.

- We define $|\Delta|$ by induction on the length of Δ:
  - $|\varnothing| \stackrel{\text{def}}{=} \varnothing$;
  - $|\Delta, b{:}U| \stackrel{\text{def}}{=} |\Delta|, b{:}\, |U|_{\Delta}$;
  - $|\Delta, b{:=}T{:}U| \stackrel{\text{def}}{=} |\Delta|$;
- If $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ then $|\Delta;\Gamma| \stackrel{\text{def}}{=} |\Delta|, x_1{:}\, |A_1|_{\Delta}, \ldots, x_n{:}\, |A_n|_{\Delta}$.

We see that definitions $b{:=}T{:}U$ in Δ are not translated into $|\Delta|$. This corresponds to the fact that all these definitions are unfolded (replaced by their definiendum) in $|b|_{\Delta}$.

Now we are able to prove the most important lemma of this subsection:

Lemma 5.59 *If* $\Delta;\Gamma \vdash_{\lambda 68} M : N$ *then* $|\Delta;\Gamma| \vdash_{\lambda \mathrm{SN}} |M|_{\Delta} : |N|_{\Delta}$.

PROOF: The proof is by induction on the derivation of $\Delta;\Gamma \vdash M : N$. We treat a few cases:

(Start: Primitive Constants)

$$\frac{\Delta;\Gamma \vdash_{\lambda 68} B : s_1 \qquad \Delta;\vdash_{\lambda 68} \P\,\Gamma.B : s_2}{\Delta, b{:}\,\P\,\Gamma.B;\vdash_{\lambda 68} b : \P\,\Gamma.B}(s_1 = *, \Box).$$

By the induction hypothesis, $|\Delta| \vdash_{\lambda \mathrm{SN}} |\P\,\Gamma.B|_{\Delta} : s_2$, so by the Start rule:

$$|\Delta|, b{:}\, |\P\,\Gamma.B|_{\Delta} \vdash b{:}\, |\P\,\Gamma.B|_{\Delta}.$$

Observe that $|\Delta, b{:}\,\P\,\Gamma.B| \equiv |\Delta|, b{:}\, |\P\,\Gamma.B|_{\Delta}$, that $|b|_{\Delta,b:\,\P\,\Gamma.B} \equiv b$ and that (by Lemma 5.54) $|\P\,\Gamma.B|_{\Delta} \equiv |\P\,\Gamma.B|_{\Delta,b:\,\P\,\Gamma.B}$;

(Start: Defined Constants)

$$\frac{\Delta;\Gamma \vdash_{\lambda 68} T : B : s_1 \qquad \Delta;\vdash_{\lambda 68} \P\,\Gamma.B : s_2}{\Delta, b{:=}(\Gamma.T){:}(\P\,\Gamma.B);\vdash_{\lambda 68} b : \P\,\Gamma.B}(s_1 = *, \Box).$$

By induction we have

$$|\Delta;| \vdash_{\lambda \mathrm{SN}} |\P\,\Gamma.B|_{\Delta} : s_2,$$

so (write $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$):

$$|\Delta;| \vdash_{\lambda\mathrm{SN}} \prod_{i=1}^{n} x_i{:}\,|A_i|_\Delta \,.\, |B|_\Delta : s_2. \tag{12}$$

By induction, we also have $|\Delta;\Gamma| \vdash_{\lambda\mathrm{SN}} |T|_\Delta : |B|_\Delta$, so:

$$|\Delta|\,, x_1{:}\,|A_1|_\Delta\,, \ldots, x_n{:}\,|A_n|_\Delta \vdash_{\lambda\mathrm{SN}} |T|_\Delta : |B|_\Delta\,, \tag{13}$$

and by repeatedly applying the λ-rule on (13) and using the fact that, by the Induction Hypothesis, the types $\prod_{j=i}^{n} x_j{:}\,|A_j|_\Delta \,.\, |B|_\Delta$ are all typable, we find:

$$|\Delta;| \vdash_{\lambda\mathrm{SN}} \left(\mathop{\lambda}_{i=1}^{n} x_i{:}\,|A_i|_\Delta \,.\, |T|_\Delta\right) : \left(\prod_{i=1}^{n} x_i{:}\,|A_i|_\Delta \,.\, |B|_\Delta\right); \tag{14}$$

(Application 1) (the Application 2-case is similar)

$$\frac{\Delta;\Gamma \vdash_{\lambda 68} M : (\Pi x{:}A.B) \qquad \Delta;\Gamma \vdash_{\lambda 68} N : A}{\Delta;\Gamma \vdash_{\lambda 68} MN : B[x{:=}N]}$$

By the induction hypothesis, we have

$$|\Delta;\Gamma| \vdash_{\lambda\mathrm{SN}} |M|_\Delta : (\Pi x{:}\,|A|_\Delta \,.\, |B|_\Delta),$$

and $|\Delta;\Gamma| \vdash_{\lambda\mathrm{SN}} |N|_\Delta : |A|_\Delta$. The application rule gives

$$|\Delta;\Gamma| \vdash_{\lambda\mathrm{SN}} |M|_\Delta\, |N|_\Delta : |B|_\Delta\, [x{:=}\, |A|_\Delta].$$

Use the definition of $|MN|_\Delta$ and Lemma 5.55 to obtain

$$|\Delta;\Gamma| \vdash_{\lambda\mathrm{SN}} |MN|_\Delta : |B[x{:=}A]|_\Delta\,.$$

⊠

Corollary 5.60 (Strong Normalisation) *$\lambda 68$ is $\beta\delta$-strongly normalising.*

PROOF: Assume, we have an infinite $\beta\delta$-reduction path in $\lambda 68$:

$$M_1 \to_{\beta\delta} M_2 \to_{\beta\delta} M_3 \to_{\beta\delta} \cdots \tag{15}$$

As δ-reduction is strongly normalising (5.17 and 5.38.2), there must be infinitely many β-reductions in this reduction path, so we have a path

$$N_1 \to_\beta N_1' \twoheadrightarrow_\delta N_2 \to_\beta N_2' \twoheadrightarrow_\delta N_3 \to_\beta N_3' \twoheadrightarrow_\delta \cdots$$

By Lemmas 5.56 and 5.57, this gives us a reduction path

$$|N_1|_\Delta \twoheadrightarrow_\beta^+ |N_1'|_\Delta \twoheadrightarrow_\beta |N_2|_\Delta \twoheadrightarrow_\beta^+ |N_2'|_\Delta \twoheadrightarrow_\beta |N_3|_\Delta \twoheadrightarrow_\beta^+ |N_3'|_\Delta \twoheadrightarrow_\beta \cdots$$

which is an infinite β-reduction path in λSN. By Lemma 5.59, $|N_1|_\Delta$ is a legal term in λSN. But as λSN is strongly normalising, the above infinite β-reduction path cannot exist. Hence, the infinite $\beta\delta$-reduction path (15) does not exist, either. ⊠

5c5 The formal relation between AUT-68 and λ68

Theorem 5.61 *Let $\mathfrak{B}$ be an* AUTOMATH *book and* Γ *an* AUTOMATH *context.*

- *If* $\mathfrak{B};\Gamma \vdash_{AUT-68}$ OK *then* $\overline{\mathfrak{B}};\overline{\Gamma}$ *is legal;*
- *If* $\mathfrak{B};\Gamma \vdash_{AUT-68} \Sigma : \Omega$ *then* $\overline{\mathfrak{B}};\overline{\Gamma} \vdash \overline{\Sigma} : \overline{\Omega}$.

PROOF: We prove both statements simultaneously, using induction on the derivation of $\mathfrak{B};\Gamma \vdash_{\mathrm{AUT-68}}$ OK and $\mathfrak{B};\Gamma \vdash \Sigma : \Omega$ of Definition 5.10 and Definition 5.11. We only treat one case; the other cases are similar or trivial. Assume, the last step of the derivation has been an application of the book extension rule def2:

$$\frac{\mathfrak{B};\Gamma \vdash_{\mathrm{AUT-68}} \Sigma_2{:}\mathtt{type} \quad \mathfrak{B};\Gamma \vdash_{\mathrm{AUT-68}} \Sigma_1{:}\Sigma_2' \quad \mathfrak{B};\Gamma \vdash_{\mathrm{AUT-68}} \Sigma_2 =_{\beta\mathrm{d}} \Sigma_2'}{\mathfrak{B},(\Gamma;k;\Sigma_1;\Sigma_2);\varnothing \vdash_{\mathrm{AUT-68}} \mathrm{OK}}.$$

By the induction hypothesis, we have

$$\overline{\mathfrak{B}};\overline{\Gamma} \vdash_{\lambda 68} \overline{\Sigma_2} : * \tag{16}$$

and

$$\overline{\mathfrak{B}};\overline{\Gamma}\vdash_{\lambda 68}\overline{\Sigma_1}:\overline{\Sigma_2'}. \qquad (17)$$

By Lemma 5.38, we have

$$\overline{\mathfrak{B}}\vdash_{\lambda 68}\overline{\Sigma_2}=_{\beta\delta}\overline{\Sigma_2'}. \qquad (18)$$

Applying the conversion rule of $\lambda 68$ to (16), (17) and (18) yields

$$\overline{\mathfrak{B}};\overline{\Gamma}\vdash_{\lambda 68}\overline{\Sigma_1}:\overline{\Sigma_2}. \qquad (19)$$

Notice that $\overline{\mathfrak{B}};\overline{\Gamma}$ is legal, so for each $x{:}\alpha\in\overline{\Gamma}$ (say: $\overline{\Gamma}\equiv\Gamma_1,x{:}\alpha,\Gamma_2$) we have $\overline{\mathfrak{B}};\Gamma_1\vdash\alpha:s$ for an $s\in\{*,\Box\}$, by the Free Variable Lemma 5.22. Thus we can repeatedly apply the ¶-formation rule (starting with (16)) to obtain:

$$\overline{\mathfrak{B}};\vdash_{\lambda 68}\P\,\overline{\Gamma}.\overline{\Sigma_2}:\Delta \qquad (20)$$

(If $\Gamma\equiv\varnothing$ then we apply the ¶-formation rule zero times, and the type of $\P\,\overline{\Gamma}.\overline{\Sigma_2}$ is $*$ instead of Δ). Now we can apply the (Start: dc) rule on (19), (16) and (20) to obtain:

$$\overline{\mathfrak{B}};k{:=}(\S\,\overline{\Gamma}.\overline{\Sigma_1}){:}(\P\,\overline{\Gamma}.\overline{\Sigma_2});\vdash_{\lambda 68}k:\P\,\overline{\Gamma}.\overline{\Sigma_2},$$

so $\overline{\mathfrak{B},(\Gamma;k;\Sigma_1;\Sigma_2);}\equiv\overline{\mathfrak{B}},k{:=}(\S\,\overline{\Gamma}.\overline{\Sigma_1}){:}(\P\,\overline{\Gamma}.\overline{\Sigma_2});$ is legal. ⊠

It is possible to prove a conservativity theorem (in the style: If $\overline{\mathfrak{B}};\overline{\Gamma}\vdash_{\lambda 68}\overline{\Sigma}:\overline{\Omega}$, then $\mathfrak{B};\Gamma\vdash_{\mathrm{AUT-68}}\Sigma:\Omega$), but we want to prove that all the typable terms of $\lambda 68$ have some interpretation in AUT-68, and not only the terms that have an equivalent in AUT-68. We have to distinguish six different cases, and the interpretation of these six cases is given after the proof of the next theorem.

Theorem 5.62 *Assume $\Delta;\Gamma\vdash_{\lambda 68}M:N$. Then there is an* AUTOMATH *book $\mathfrak{B}$ and an* AUTOMATH *context Γ' such that $\mathfrak{B};\Gamma'\vdash_{AUT-68}$ OK, and $\overline{\mathfrak{B}},\overline{\Gamma'}\equiv\Delta;\Gamma$. Moreover,*

1. *If $N\equiv\Box$ then $M\equiv *$;*
2. *If $\Delta;\Gamma\vdash_{\lambda 68}N:\Box$ then $N\equiv *$ and there is $\Omega\in\mathcal{E}$ such that $\overline{\Omega}\equiv M$ and $\mathfrak{B};\Gamma'\vdash_{AUT-68}\Omega:\mathtt{type}$;*

3. *If $N \equiv \Delta$ then there is $\Gamma'' \equiv x_1{:}\Sigma_1, \dots, x_n{:}\Sigma_n$ and $\Omega \in \mathcal{E}^+$ such that*
   - *Γ', Γ'' is correct with respect to $\mathfrak{B}$;*
   - *$M \equiv \P\,\overline{\Gamma''}.\overline{\Omega}$;*
   - *$\Omega \equiv \mathtt{type}$ or $\mathfrak{B}; \Gamma' \vdash_{AUT-68} \Omega : \mathtt{type}$;*

4. *If $\Delta; \Gamma \vdash_{\lambda 68} N : \Delta$ then there are $b \in \mathcal{C}$ and $\Sigma_1, \dots, \Sigma_n \in \mathcal{E}$ such that $M \equiv b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$. Moreover, $\mathfrak{B}$ contains a line*

$$(x_1{:}\Omega_1, \dots, x_m{:}\Omega_m; b; \Xi_1; \Xi_2)$$

   such that
   - *$m > n$;*
   - *$\mathfrak{B}; \Gamma' \vdash_{AUT-68} \Sigma_i{:}\Omega_i[x_1, \dots, x_{i-1}{:=}\Sigma_1, \dots, \Sigma_{i-1}]$ $(1 \le i \le n)$;*
   - *$N \equiv \left(\P_{i=n+1}^{m}\, x_i{:}\overline{\Omega_i}.\overline{\Xi_2}\right)[x_1, \dots, x_n{:=}\overline{\Sigma_1}, \dots, \overline{\Sigma_n}]$;*

5. *If $N \equiv *$ then there is $\Omega \in \mathcal{E}$ such that $\overline{\Omega} \equiv M$ and $\mathfrak{B}; \Gamma' \vdash_{AUT-68} \Omega : \mathtt{type}$;*

6. *If $\Delta; \Gamma \vdash_{\lambda 68} N : *$ then there are $\Sigma, \Omega \in \mathcal{E}$ such that $\overline{\Sigma} \equiv M$ and $\overline{\Omega} \equiv N$, and $\mathfrak{B}; \Gamma' \vdash_{AUT-68} \Sigma : \Omega$, and $\mathfrak{B}; \Gamma' \vdash_{AUT-68} \Omega : \mathtt{type}$.*

PROOF: We use induction on the derivation of $\Delta; \Gamma \vdash_{\lambda 68} M : N$. We only treat a few cases:

Weakening: definitions The last step in the derivation has been

$$\frac{\Delta; \vdash_{\lambda 68} M : N \qquad \Delta; \Gamma \vdash_{\lambda 68} T : B : s_1 \qquad \Delta; \vdash_{\lambda 68} \P\,\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B); \vdash_{\lambda 68} M : N}$$

where $s_1 \equiv *$ or $s_1 \equiv \Box$. Use the induction hypothesis and determine $\mathfrak{B}$, Γ', Σ_1, Σ_2, Ω_1, and Ω_2 such that $\overline{\mathfrak{B}} \equiv \Delta$, $\overline{\Gamma'} \equiv \Gamma$, $\overline{\Sigma_1} \equiv T$, $\overline{\Sigma_2} \equiv B$, $\overline{\Omega_1} \equiv M$ and $\overline{\Omega_2} \equiv N$. We know by induction that $\mathfrak{B}; \Gamma' \vdash_{\mathrm{AUT-68}} \Sigma_2 : \mathtt{type}$ (if $s_1 \equiv *$) or $\Sigma_2 \equiv *$ (if $s_2 \equiv \Box$). Also, $\mathfrak{B}; \Gamma' \vdash_{\mathrm{AUT-68}} \Sigma_1 : \Sigma_2$. This makes it possible to extend $\mathfrak{B}$ with a new line, thus obtaining a legal book $\mathfrak{B}, (\Gamma'; b; \Sigma_1; \Sigma_2)$. Using Weakening for AUT-68 (Lemma 5.19) and the induction hypothesis on $\Delta; \vdash_{\lambda 68} M : N$, it is not hard to verify the cases 1–6 for $\Delta, b{:=}(\S\,\Gamma.T){:}(\P\,\Gamma.B); \vdash_{\lambda 68} M : N$;

Application 2 The last step in the derivation has been

$$\frac{\Delta;\Gamma \vdash_{\lambda 68} M_1 : (\P x{:}A.B) \qquad \Delta;\Gamma \vdash_{\lambda 68} M_2 : A}{\Delta;\Gamma \vdash_{\lambda 68} M_1 M_2 : B[x{:=}M_2]}.$$

Determine $\mathfrak{B}$, Γ' such that $\overline{\mathfrak{B}} \equiv \Delta$ and $\overline{\Gamma'} \equiv \Gamma$. By Correctness of Types 5.32 and the Generation Lemma 5.30, we have $\Delta;\Gamma \vdash_{\lambda 68} (\P x{:}A.B) : \Delta$, so by the induction hypothesis (case 4), there are $b, \Sigma_1, \ldots, \Sigma_n$ such that $M_1 \equiv b\overline{\Sigma_1} \cdots \overline{\Sigma_n}$, and there is a line

$$(x_1{:}\Omega_1, \ldots, x_m{:}\Omega_m; b; \Xi_1; \Xi_2)$$

in $\mathfrak{B}$ such that $m > n$, $\mathfrak{B};\Gamma' \vdash_{\mathrm{AUT-68}} \Sigma_i{:}\Omega_i[x_j{:=}\Sigma_j]_{j=1}^{i-1}$ for $i = 1, \ldots, n$, and

$$\P x{:}A.B \equiv \left(\P_{i=n+1}^{m} x_i{:}\overline{\Omega_i}.\overline{\Xi_2}\right)[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n}.$$

Observe: $A \equiv \overline{\Omega_{n+1}}[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n}$. As $\mathfrak{B};\Gamma' \vdash_{\mathrm{AUT-68}} \Omega_{n+1} : \mathtt{type}$ or $\Omega_{n+1} \equiv \mathtt{type}$, we have $\Delta;\Gamma \vdash_{\lambda 68} \overline{\Omega_{n+1}} : s$ for an $s \in \{*, \Box\}$, and by Substitution Lemma and Transitivity Lemma we have $\Delta;\Gamma \vdash_{\lambda 68} \overline{\Omega_{n+1}}[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n} : s$, hence $\Delta;\Gamma \vdash_{\lambda 68} A : s$.
With the induction hypothesis we determine $\Sigma \in \mathcal{E}$ such that

$$\mathfrak{B};\Gamma' \vdash_{\mathrm{AUT-68}} \Sigma : \Omega_{n+1}[x_j{:=}\Sigma_j]_{j=1}^{n},$$

and $M_2 \equiv \overline{\Sigma}$. We now treat the most important ones of the cases 1–6:

4. The only thing that does not directly follow from the results above is $m > n+1$. Assume, for the sake of the argument, $m = n+1$. Then $B[x{:=}M_2] \equiv \overline{\Xi_2}[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n+1}$. As $\Delta;\Gamma \vdash_{\lambda 68} B[x{:=}M_2] : \Delta$, $\overline{\Xi_2}[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n+1}$ is of the form $\P x{:}P.Q$, which is impossible;
6. Notice: $B[x{:=}M_2] \equiv \left(\P_{j=n+2}^{m} x_i{:}\overline{\Omega_i}.\overline{\Xi_2}\right)[x_j{:=}\overline{\Sigma_j}]_{j=1}^{n+1}$. We have $\Delta;\Gamma \vdash_{\lambda 68} B[x{:=}M_2] : *$. Therefore $B[x{:=}M_2]$ cannot be of the form $\P y{:}P.Q$, and therefore $m = n+1$. Therefore, $\mathfrak{B};\Gamma' \vdash_{\mathrm{AUT-68}} b(\Sigma_1, \ldots, \Sigma_{n+1}) : \Xi_2[x_i{:=}\Sigma_i]_{i=1}^{n+1}$.

⊠

Remark 5.63 We give some explanation to the different cases mentioned in the formulation of Theorem 5.62.

- The cases $N \equiv \Box$ and $\Delta;\Gamma \vdash N : \Box$ imply that there are no other terms in $\lambda 68$ than $*$ itself at the same level as $*$. This corresponds to the fact that type is the only "top-expression" in AUT-68;
- The cases $N \equiv *$ and $\Delta;\Gamma \vdash N : *$ give a precise correspondence between expressions of AUT-68 and terms of $\lambda 68$: If $M : N$ in $\lambda 68$ then there are expressions Σ, Ω in AUT-68 such that $\Sigma : \Omega$ in AUT-68 and $\overline{\Sigma} \equiv M$ and $\overline{\Omega} \equiv N$;
- The cases $N \equiv \Delta$ and $\Delta;\Gamma \vdash N : \Delta$ cover terms that do not have an equivalent in AUT-68 but are necessary in $\lambda 68$ to form terms that have equivalents in AUT-68. More specific, this concerns terms of the form $\P_{i=1}^{n} x_i{:}A_i.B$ (which are needed to introduce constants) and terms of the form $bM_1 \cdots M_n$, where b is a constant of type $\P_{i=1}^{m} x_i{:}A_i.B$ for certain $m > n$ (which are needed to construct $\lambda 68$-equivalents of expressions of the form $b(\Sigma_1, \ldots, \Sigma_m)$).

We conclude that $\lambda 68$ and AUT-68 coincide as much as possible, and that the terms in $\lambda 68$ that do not have an equivalent in AUT-68 can be traced easily (these are the terms of type Δ and the terms of a type $N : \Delta$, and the sorts $\Box$ and Δ, which are needed to give a type to $*$ and to the $\P$-types).

Notice that the alternative definition of δ-reduction in $\lambda 68$, discussed at the end of Subsection 5a3, would introduce more terms in $\lambda 68$ without an equivalent in AUT-68, namely terms of the form $\lambda_{i=1}^{n} x_i{:}A_i.B$.

5d Related work

The system AUT-68 is one of several AUTOMATH-systems that have been proposed. Another frequently used system is AUT-QE. In Section 5d1 we compare AUT-68 to AUT-QE and describe how we can easily adapt $\lambda 68$ to a system λQE.

Recently, various type systems with definitions in PTS-style have been proposed by, amongst others, Bloo, Kamareddine and Nederpelt ([16, 17])

and by Severi and Poll ([114]). The presentation of AUT-68 in the PTS-like system $\lambda 68$ makes a good comparison between these systems and the definition system in AUT-68 possible. This will be done in Sections 5d2 and 5d3.

5d1 AUT-QE

The system AUT-QE has many similarities with AUT-68. There are a few extensions:

1. We can also form abstraction expression $[x{:}\Sigma]\texttt{type}$ (thus extending Definition 5.1);

2. Inhabitants of types of the form $[x{:}\Sigma]\texttt{type}$ are introduced by extending the abstraction rules 1 and 2 of Definition 5.11 with the following rule for AUT-QE:

$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}\texttt{type}\qquad\mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Sigma_2{:}\texttt{type}}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Sigma_2:[x{:}\Sigma_1]\texttt{type}}.$$

   Notice that the expression $[x{:}\Sigma_1]\texttt{type}$ is not typable, just as $\texttt{type}$ is not typable. In a translation to a PTS, these expressions should get type $\Box$;

3. There is a new reduction relation on expressions, which is specific for AUT-QE and therefore will be called $\rightarrow_{\text{QE}}$ in the sequel. The relation is described by the rule

$$[x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n][y{:}\Omega]\texttt{type}\rightarrow_{\text{QE}}[x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\texttt{type}$$

   (for $n\geq 0$).

The first two rules are rather straightforward. They correspond to an extension of $\lambda\!\rightarrow$ to λP in Pure Type Systems. It is also easy to extend $\lambda 68$ with similar rules: We just add the Π-formation rule $(*,\Box,\Box)$:

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

In AUT-68 PAT is implemented in De Bruijn-style (see Section 4a4 and Example 5.9). An implementation of predicate logic in Howard-style is not possible in AUT-68, but due to the extension with types of the form $[x{:}\Sigma]\mathtt{type}$, such an implementation becomes possible in AUT-QE. See [39].

The third rule deserves some extra attention, as it is very unusual. It is needed in AUT-QE because that system does not distinguish between λs and Πs. In AUT-68 this did not matter, as from the context it could always be derived whether an expression $[x{:}\Sigma]\Omega$ should be interpreted as $\lambda x{:}\Sigma.\Omega$ or as $\Pi x{:}\Sigma.\Omega$. The latter should have type $\mathtt{type}$, and the first should not have type $\mathtt{type}$.

In AUT-QE the situation is more complicated. A expression $[x{:}\Sigma]\Omega$ may have more than one type:

Example 5.64 Let $\mathfrak{B}$ consist of two lines:

$$(\varnothing, \alpha, —, \mathtt{type}),$$
$$(\alpha{:}\mathtt{type}, x, —, \alpha).$$

Notice that, using rule (abstr.1) of Definition 5.11, we can derive that

$$\mathfrak{B}; \alpha{:}\mathtt{type} \vdash_{\mathrm{QE}} [x{:}\alpha]\alpha : \mathtt{type}. \tag{21}$$

But using the new abstraction rule of AUT-QE we can also derive

$$\mathfrak{B}; \alpha{:}\mathtt{type} \vdash_{\mathrm{QE}} [x{:}\alpha]\alpha : [x{:}\alpha]\mathtt{type}. \tag{22}$$

More generally, we can prove that the two statements below are equivalent (that is: if either of them is derivable then they are both derivable) in AUT-QE:

$$\mathfrak{B};\Gamma \quad \vdash_{\mathrm{QE}} \quad [x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\Omega : [x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\mathtt{type}; \tag{23}$$
$$\mathfrak{B};\Gamma \quad \vdash_{\mathrm{QE}} \quad [x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\Omega : [x_1{:}\Sigma_1]\cdots[x_m{:}\Sigma_m]\mathtt{type} \tag{24}$$

(for $m < n$). In (23), the expression $[x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\Omega$ should be read as $\lambda_{i=1}^{n} x_i{:}\Sigma_i.\Omega$; in (24) it should be read as $\lambda_{i=1}^{m} x_i{:}\Sigma_i. \prod_{j=m+1}^{n} x_j{:}\Sigma_j.\Omega$.

But this equivalence holds only for expressions of the form

$$[x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\Omega$$

and not for general expressions Σ (take, for instance, Σ a variable). In order that the equivalence holds for general expressions Σ, De Bruijn introduced a rule for type inclusion:

$$\frac{\mathfrak{B};\Gamma \vdash_{\mathrm{QE}} \Sigma : [x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]\mathtt{type}}{\mathfrak{B};\Gamma \vdash_{\mathrm{QE}} \Sigma : [x_1{:}\Sigma_1]\cdots[x_{n-1}{:}\Sigma_{n-1}]\mathtt{type}}.$$

Lists of abstractions $[x_1{:}\Sigma_1]\cdots[x_n{:}\Sigma_n]$ were also called *telescopes* by de Bruijn. In the rule for type inclusion, we see that one part of the telescope "collapses".

5d2 Comparison with the DPTSs of Severi and Poll

In [114], Severi and Poll present an extension of PTSs with definitions, thus obtaining Pure Type Systems with Definitions (DPTSs). They extend the usual PTS-rules with the following D-rules:

$$\textbf{(D-start)}\qquad \frac{\Gamma \vdash a : A}{\Gamma, x{=}a{:}A \vdash x : A}$$

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

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

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

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

where D-reduction is defined by the following rules:

$$\Gamma_1, x{=}a{:}A, \Gamma_2 \vdash x \to_{\mathrm{D}} a;$$

$$\Gamma \vdash (x{=}a{:}A \text{ in } b) \to_{\mathrm{D}} b \qquad\qquad (x \notin \mathrm{FV}(b));$$

$$\frac{\Gamma, x{=}a{:}A \vdash b \to_{\mathrm{D}} b'}{\Gamma \vdash (x{=}a{:}A \text{ in } b) \to_{\mathrm{D}} (x{=}a{:}A \text{ in } b')}$$

and the usual compatibility rules. As we see, there is an extra class of terms in DPTSs, namely those of the form $(x{=}a{:}A \text{ in } b)$.

When regarding both systems we find that:

- In DPTSs, definitions do not only occur in a context, but may also occur in terms. Moreover, definitions may disappear from contexts when they are introduced in terms (e.g. the D-form and the D-intro rules, and the last of the three D-reduction rules), and definitions may disappear from terms when the definiendum does not occur in that term (the middle D-reduction rule).

This gives definitions a more temporary character: We can use them as long as needed, and when we do not need them any more, we can remove them from the context.

Definitions can also play a more local role: A definition that is needed in only one term can be imported into that term while it is not necessary to carry it around in the (global) context, as well.

This temporary and local behaviour of definitions is not present in AUTOMATH;

- Due to the fact that definitions can also play a local role, D-reduction can also unfold definitions which are not present in the (global) context, but which are given within the term. For example, we have $\alpha{:}* \vdash (id{=}\lambda x{:}\alpha.x \text{ in } id) \twoheadrightarrow_{\mathrm{D}} \lambda x{:}\alpha.x$, though there is no definition of id in the context $\alpha{:}*$.

  Again, this is not possible in AUTOMATH;

- The start rule for definitions in DPTSs,
$$\frac{\Gamma \vdash T : B}{\Gamma, x{=}T{:}B \vdash x : B}$$
  does not require $\Gamma \vdash B : s$ for a sort s. In $\lambda 68$ we have the rule (Start: dc):
$$\frac{\Delta; \Gamma \vdash T : B : s_1 \qquad \Delta; \vdash \P\,\Gamma.B : s_2}{\Delta, x{:=}\,\S\,\Gamma.T{:}\,\P\,\Gamma.B; \vdash x : \P\,\Gamma.B}(s_1 = *, \Box)$$
  where we see that both B and $\P\,\Gamma.B$ need to be of a certain sort (and B must be of sort $*$ or $\Box$);

- The start rules for definitions in DPTSs and in $\lambda 68$ also differ in another respect, namely the type of definiens and definiendum. In DPTSs they have the same type (in the notation of the previous paragraph: B), while in $\lambda 68$ the definiens T has type B and the definiendum x has type $\P\,\Gamma.B$. This topic has already been discussed when we introduced the definition mechanism of $\lambda 68$ in Section 5b3;

- D-reduction differs from δ-reduction, also when only global definitions are taken into account. For instance, δ-reduction is *substitutive*, i.e. if $\Delta \vdash A \rightarrow_\delta A'$ then $\Delta \vdash A[x{:=}b] \rightarrow_\delta A'[x{:=}b]$ (proof: Induction on the structure of A). D-reduction is not substitutive: take $\Gamma \equiv \alpha{:}*, y{=}\alpha{:}*$. Then $\Gamma \vdash y \rightarrow_{\mathrm{D}} \alpha$, but $\Gamma \not\vdash y[\alpha{:=}M] \rightarrow_{\mathrm{D}} \alpha[\alpha{:=}M]$ for arbitrary M.

In $\lambda 68$, this example would look as follows. Take $\Delta \equiv y{:=}\alpha{:}\P\alpha{:} * .*$. Then $\Delta \vdash y\alpha \rightarrow_{\delta} \alpha$ and $\Delta \vdash y\alpha[\alpha{:=}M] \rightarrow_{\delta} \alpha[\alpha{:=}M]$.

Substitutivity for $\rightarrow_{\mathrm{D}}$ is lost, because unfolding a definition by D-reduction may introduce new free variables in the term. In AUTOMATH, all free variables in the definiens must be added as parameters to the definiendum. In $\lambda 68$ this is visible in the Start and Weakening rules for defined constants: The right part Γ of the context $\Delta;\Gamma$ that is used to type the definiens T in these rules, serves as list of parameters in the definiendum. When an AUTOMATH-definition is unfolded, the free variables occurring in the definiens are replaced by the parameters;

- We see that the definition of y in $\lambda 68$ in the example above is more general than in the corresponding DPTS situation. In the DPTS-example, y D-reduces to one, fixed term α. In the $\lambda 68$ version, yM is defined for any (typable) term M. To do something similar in DPTSs, one needs to define y as $\lambda\alpha{:}*.\alpha$. In particular, one needs to type the term $\lambda\alpha{:}*.\alpha$, which involves the use of Π-formation rule $(\Box,\Box)$, so the use of a higher type system. One could say that AUTOMATH and $\lambda 68$ use an implicit λ-abstraction where DPTSs need an explicit λ-abstraction. On this point, AUTOMATH and $\lambda 68$ are more flexible than DPTSs. This is due to the parameter mechanism of AUTOMATH. It is possible to extend DPTSs with a parameter mechanism as well. This will be the main topic of Chapter 6.

We summarise the differences between DPTSs and AUTOMATH:

- DPTSs have 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. In AUTOMATH, B has to be typable;
- The D-reduction of DPTSs is not substitutive; δ-reduction of AUTOMATH is substitutive;

- AUTOMATH has a parameter mechanism, DPTSs do not have such a mechanism.

5d3 Comparison with systems of Bloo, Kamareddine and Nederpelt

In [17], Bloo, Kamareddine and Nederpelt extend the usual PTSs with both Π-conversion and definitions. [17] starts with PTSs extended with Π-reduction, but *without* definitions (see [73]). This system (which we will call $\lambda\beta\Pi$ for the moment) does not have the Subject Reduction property. For instance, one can derive

$$\alpha{:}{*}, x{:}\alpha \vdash (\lambda y{:}\alpha.y)x : (\Pi y{:}\alpha.\alpha)x,$$

but it is not possible to derive

$$\alpha{:}{*}, x{:}\alpha \vdash x : (\Pi y{:}\alpha.\alpha)x.$$

Adding a definition mechanism results in a system that we will call $\lambda\beta\Pi\delta$ and is the main point of interest in [17]. As a sort of "side effect" of adding this definition mechanism, $\lambda\beta\Pi\delta$ has Subject Reduction.

It will be clear that it is useful to take Π-conversion into consideration when comparing AUTOMATH with $\lambda\beta\Pi$. Though our system $\lambda 68$ does not have Π-conversion, it is very easy to extend it to a system $\lambda\Pi 68$ by:

- Changing rule (App_1) into

$$\frac{\Delta;\Gamma \vdash M : \Pi x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : (\Pi x{:}A.B)N}$$

  (Rule (App_2) remains unchanged — see also the discussion in Section 5b1);

- Adding a new reduction rule $\rightarrow_\Pi$ by

$$(\Pi x{:}A.B)N \rightarrow_\Pi B[x{:=}N].$$

The system $\lambda\Pi 68$ is actually much closer to AUT-68 than $\lambda 68$ as AUT-68 has Π-conversion as well.

In $\lambda\Pi68$ we do not have Subject Reduction, either: It is not hard to derive

$$;\alpha{:}*, x{:}\alpha \vdash (\lambda y{:}\alpha.y)x : (\Pi y{:}\alpha.\alpha)x$$

in $\lambda\Pi68$. Nevertheless, we can not derive

$$;\alpha{:}*, x{:}\alpha \vdash x : (\Pi y{:}\alpha.\alpha)x$$

(In such a derivation, no definitions can occur: Definitions, once they have been introduced, cannot be removed from the left part of the context any more; when we are not allowed to use any definition rules, $\lambda\Pi68$ has not more rules than the system $\lambda\beta\Pi$ of Bloo, Kamareddine and Nederpelt).

The "restoration" of Subject Reduction in $\lambda\beta\Pi\mathrm{d}$ is only because of the special way in which definitions are introduced and removed from the context. We do not go into details on this; the interested reader can consult [17].

Another main difference between $\lambda\Pi68$ and $\lambda\beta\Pi\mathrm{d}$ has already appeared in Section 5d2: In $\lambda\Pi68$ there is a different correspondence between the types of definiendum and definiens than in $\lambda\beta\Pi\mathrm{d}$.

Conclusions

In this chapter we described the most basic AUTOMATH-system, AUT-68, in a PTS style. Though such descriptions have been given before in, for example, [5] and [54], we feel that our description is more accurate than the two ones cited above. Moreover, our description pays attention to the definition system, which is a crucial item in AUTOMATH. The descriptions mentioned above do not.

$\lambda68$, the main topic of this chapter, does not include Π-conversion (while AUTOMATH does). However, it is very easy to adapt $\lambda68$ to include Π-conversion (this was done in Section 5d3 to compare our system to the system in [17]).

The adaption of $\lambda68$ to a system $\lambda\mathrm{QE}$, representing the AUTOMATH-system AUT-QE is not hard, either: It requires adaption of the Π-formation rule to include not only the rule $(*,*,*)$ but also $(*,\Box,\Box)$ and introduction of the additional reduction rule of type inclusion.

Of course, the properties of $\lambda68$ presented in Section 5c have to be reviewed for these new systems.

When comparing $\lambda 68$ to other type systems with definitions, we find an important difference. In $\lambda 68$, the correspondence between types of definiendum and definiens differs from the similar correspondence in the systems in [114] and [17].

The reason why $\lambda 68$ differs from other theories in this respect has been discussed in Section 5b3: The definition system in AUTOMATH allows *parameters* to occur in the definiens, and there is no parameter mechanism in PTSs. In Chapter 6, we extend PTSs with a parameter mechanism. This extension has AUT-68 as a subsystem. Moreover, we show that a parameter mechanism has also other advantages.

Chapter 6

Pure Type Systems with Parameters

This chapter is devoted to the description of pure type systems with parameters. One reason to study this extension of PTSs is to give a better description of AUTOMATH than in the previous Chapter, where we had to work with the sort Δ to store terms and types that did not have a counterpart in AUTOMATH (cf. Subsection 5b1). Such terms and types were needed for the description of the system because no parameters were used. But there are many more arguments why type systems with parameters deserve to be studied:

Definitions The various AUTOMATH systems had mechanisms to incorporate parameters and definitions in the formal language (as we saw in the previous chapter). There are also modern systems in which definitions are part of the formal machinery of the system (see [114], [16]). We will show that the (now widely accepted) system of Severi and Poll [114] can be easily extended with a parameter mechanism;

Programming languages Parameters and parametric definitions are not only used in implementations of type systems. They also occur in many other parts of computer science. For example, look at the following Pascal fragment P with the function `double`:

```
function double(z : integer) : integer;
```

```
begin
   double := z + z
end;
```

In a PTS with definitions like the one in [114], P could be represented by the context declaration

$$\texttt{double} = (\lambda \texttt{z}{:}\texttt{Int}.(\texttt{z}+\texttt{z})) : (\texttt{Int} \rightarrow \texttt{Int}).$$

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

- The declaration has as subterm the type `Int` $\rightarrow$ `Int`. This subterm does not occur in P itself. More general, Pascal does not have a mechanism to construct types of the form $A \rightarrow B$;
- Moreover, due to the way in which `double` is defined, `double` is a separate subterm in a PTS. But `double` itself is not a separate expression in Pascal: you 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. The extension of the system of [114] with a parameter mechanism, to be presented in this chapter, allows us to translate P by the parametric context declaration

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

This declaration does not have the disadvantages described above:

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

First-order logic Implementations of first-order logic in a PTS in PAT-style usually use a PTS that is related to λP. λP has sorts $*, \Box$, axiom $*{:}\Box$, and two Π-formation rules, $(*, *, *)$ and $(*, \Box, \Box)$. In this

PTS it is possible to construct types (that is: terms of type $*$ or $\Box$) that are not in β-normal form. Hence, a derivation in λP can have non-trivial applications of the conversion rule

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

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 PTS in which applications of the conversion rule are only possible when $B_1 \equiv B_2$. This is the case if all types in such a PTS are in β-normal form. As all types in $\lambda{\rightarrow}$ (that is: λP without Π-formation rule $(*, \Box, \Box)$) 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 $\lambda{\rightarrow}$. The introduction of the relation symbols in a first order language involves the Π-formation rule $(*, \Box, \Box)$.

But in a first-order language, a relation symbol R always has a fixed arity $\mathfrak{a}(R)$. This means that R itself is not a proposition. It can only be used to *construct* a proposition: if $t_1, \ldots, t_{\mathfrak{a}(R)}$ are terms, then $R(t_1, \ldots, t_{\mathfrak{a}(R)})$ is a proposition. With the use of parameters in PTSs, it is possible to introduce the relation symbols without Π-formation rule $(*, \Box, \Box)$. This results in a system in which the conversion rule is superfluous, and therefore easier to handle in implementations. See Section 6f;

Philosophical arguments 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.

Assuming a variable $\mathbf{N}$ (the type of natural numbers) of type $*$, a variable 0 (representing the natural number zero) of type $\mathbf{N}$ 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 $\mathbf{N} \to \mathbf{N} \to *$, the induction axiom can be described by the following PTS-type (let's call it: $\mathtt{Ind}$):

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

in a PTS with sorts $*, \square$, axiom $* : \square$ and Π-formation rules $(*, *, *)$, $(*, \square, \square)$, $(\square, *, *)$. With this type $\mathtt{Ind}$ one can introduce a variable $\mathtt{ind}$ of type $\mathtt{Ind}$ that may serve as a proof term for any application of the induction axiom. This is the logician's approach.

For a mathematician, who only *applies* the induction axiom and 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 $\mathtt{ind}$ only in combination with terms $P : \mathbf{N}{\to}*$, $Q : P0$ and $R : \Pi n{:}\mathbf{N}.\Pi m{:}\mathbf{N}.Pn{\to}Snm{\to}Pm$ to form a term $\mathtt{ind}PQR$ of type $\Pi n{:}\mathbf{N}.Pn$. 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):

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

If now $P : \mathbf{N}{\to}*$, $Q{:}P0$ and $R : \Pi n{:}\mathbf{N}.\Pi m{:}\mathbf{N}.Pn{\to}Snm{\to}Pm$, then one can form the term $\mathtt{ind}(P, Q, R)$ of type $\Pi n{:}\mathbf{N}.Pn$. The types that occur in this scheme can all be constructed using sorts $*, \square$, axiom $* : \square$ and rules $(*, *, *)$, $(*, \square, \square)$, hence the rule $(\square, *, *)$ 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). In this light it is interesting to note that AUTOMATH, which has a parameter mechanism, was developed from the viewpoint of mathematicians (see [23]);

A different form of abstraction and application In λ-calculus without parameters there is one mechanism for abstraction and application. For abstraction, we use λ-abstraction, and application is implemented via function application. Abstraction and application form the basis for a type system. A parameter mechanism is a different abstraction-and-application mechanism. In the philosophical argument above, the parametric scheme for induction could only be used when parameters were supplied. In other words: abstraction is allowed, but has to be followed immediately by application. In the perspective of our study of the various ways in which application and abstraction are present in type theory, we conclude that this mechanism for combined abstraction and application, being different from the λ-calculus mechanism, deserves our attention.

We conclude that there is ample motivation to extend PTSs with parameters.

There are several ways in which such an extension can be made. For instance, when working in the systems of the Barendregt Cube, we may want to add only parametric terms $t(p_1, \ldots, p_n)$ for which the parameters $p_1, \ldots, p_n$ have types $A_1, \ldots, A_n$ that are of sort $*$. But we could also decide to add parametric terms $t(p_1, \ldots, p_n)$ without this restriction to the types of the $p_1, \ldots, p_n$.

There is a method to classify these various parametric extensions that corresponds to the classification of type systems that is used in the framework of Pure Type Systems.

In the Barendregt Cube, there are two sorts $*$ and $\Box$, and the various PTSs 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 $(*, *, *)$, $(*, \square, \square)$, $(\square, *, *)$ and $(\square, \square, \square)$. If we do not allow all Π-type constructions, we get one of the subsystems of the Calculus of Constructions in the Barendregt Cube.

Something similar can be done with the parameter mechanism. One option is to provide one, general way of parametric abstraction and parametric application. We then allow all kinds of parameters. On the other hand, there are several ways in which a parameter mechanism may be restricted. We mention two ways:

- Assume, we are working in one of the systems of the Barendregt Cube, extended with parameters, and we have that $t(p_1, \ldots, p_m)$ has type A. By Correctness of Types, A has either type $*$ or type $\square$. One can imagine that we only allow $t(p_1, \ldots, p_m)$ if it has type A of type $*$ (so we only allow parametric *terms*);

- Still working in one of the systems of the Barendregt Cube extended with parameters, we will show that the parameters $p_1, \ldots, p_m$ in a term $t(p_1, \ldots, p_m)$ are typable themselves. Again, a parameter p_i can have a type P_i of type $*$ (so p_i is at *term level*), or a type P'_i of type $\square$ (so p_i is at *type level*), and there are systems in which one would only allow parameters p_i that have a type P_i of type $*$ (or of type $\square$).

These two possibilities for restriction are orthogonal in the sense that they can be combined. In many Pascal versions, for instance, parametric *terms* can only have parameters at *term level*. It is, for instance, not possible in Pascal to write a function `CartProd` that takes two types A and B as parameters, and returns a type that represents the Cartesian product $A \times B$ of A and B.

It is possible to incorporate such restrictions in our system in a similar way as the restrictions on the formation of Π-types in PTSs. We then obtain rules for parameter constructions. These rules have the form (s_1, s_2). The sort s_1 indicates that the parameters $p_1, \ldots, p_m$ have to have types $P_1, \ldots, P_m$ of sort s_1. The sort s_2 indicates that the resulting parametric term must have a type P of sort s_2. The combination of the rules for parameter constructions with the well-known rules for the construction of Π-types in the Barendregt Cube leads to a division of the Barendregt Cube into eight sub-cubes (we illustrate this in Figure 11 on page 278). 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 can 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 6.79).

This chapter is organised as follows. In Section 6a, we give definitions of PTSs extended with parametric constants and definitions. This definition includes an extension of the δ-reduction described in [114] (which unfolds definitions) to parametric definitions. In Section 6b 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 6c, we show some elementary properties of the system introduced in Section 6a, like a Generation Lemma, and the Subject Reduction theorem for $\beta\delta$-reduction. We also prove that $\beta\delta$-reduction is strongly normalising if a slightly stronger PTS is β-strongly normalising.

Section 6d is devoted to the various ways in which parameters can be added to a PTS in a more restricted way, with the refined Barendregt Cube of Figure 11 as a result.

In Section 6e, we compare our system with some other type systems, like AUTOMATH. We place various AUTOMATH systems in the refined Barendregt Cube of Figure 11.

In Section 6f we see that the use of parameters can sometimes result in simpler and more realistic implementations of type systems.

6a Parametric constants and definitions

In [114], PTSs extended to include definitions are abbreviated as DPTSs. In this section we extend PTSs with parametric constants and definitions. This extension will also contain the DPTSs (definitions in DPTSs can be interpreted as parametric definitions with zero parameters). In Section 6e, we show that AUT-68 can be seen as a (on some points somewhat restricted version of a) PTS with parameters and definitions.

Definition 6.1 The set $\mathcal{T}_P$ of *parametric terms* is defined together with the set $\mathcal{L}_V$ of *lists of variables* and the set $\mathcal{L}_T$ of *lists of terms*:

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

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 $\texttt{double}(\texttt{x}{:}\mathbb{N}){=}(\texttt{x}{+}\texttt{x}){:}\mathbb{N}$ 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 stand in contrast to global definitions. Global definitions are given in a context Γ, and refer to any term that is considered within Γ (see the forthcoming Definition 6.8). The definition system in AUTOMATH can be compared to the system of global definitions in this Chapter. However, there are no local definitions in AUTOMATH.

Definition 6.2

- We extend the definition of $\textsc{fv}(A)$, the set of *free variables* of a term A, to parametric terms:

$$\begin{aligned}
\textsc{fv}(c(a_1,\ldots,a_n)) &= \textstyle\bigcup_{i=1}^n \textsc{fv}(a_i);\\
\textsc{fv}\Big(c(\vec{x}{:}\vec{A}){=}A{:}B \text{ IN } C\Big) &= \textstyle\bigcup_{i=1}^n (\textsc{fv}(A_i) \setminus \{x_1,\ldots,x_{i-1}\})\\
&\quad \cup (\textsc{fv}(A) \cup \textsc{fv}(B)) \setminus \{x_1,\ldots,x_n\}\\
&\quad \cup \textsc{fv}(C)
\end{aligned}$$

  where $\vec{x}{:}\vec{A}$ denotes $x_1{:}A_1,\ldots,x_n{:}A_n$;

- We similarly define $\textsc{cons}\,(A)$, the set of *constants and global definitions* of A:

$$\begin{aligned}
\textsc{cons}\,(s) = \textsc{cons}\,(x) &= \emptyset;\\
\textsc{cons}\,(c(a_1,\ldots,a_n)) &= \{c\} \cup \textstyle\bigcup_{i=1}^n \textsc{cons}\,(a_i)\,;\\
\textsc{cons}\,(AB) &= \textsc{cons}\,(A) \cup \textsc{cons}\,(B)\,;\\
\textsc{cons}\,(\lambda x{:}A.B) &= \textsc{cons}\,(A) \cup \textsc{cons}\,(B)\,;\\
\textsc{cons}\,(\Pi x{:}A.B) &= \textsc{cons}\,(A) \cup \textsc{cons}\,(B)\,;\\
\textsc{cons}\,\Big(c(\vec{x}{:}\vec{A}){=}A{:}B \text{ IN } C\Big) &= \textstyle\bigcup_{i=1}^n \textsc{cons}\,(A_i)\\
&\quad \cup \textsc{cons}\,(A) \cup \textsc{cons}\,(B)\\
&\quad \cup (\textsc{cons}\,(C) \setminus \{c\}).
\end{aligned}$$

$\textsc{fv}(A) \cup \textsc{cons}\,(A)$ forms the *domain* $\textsc{dom}\,(A)$ of A.

Remark 6.3 The definition of

$$\textsc{fv}\Big(c(\vec{x}{:}\vec{A}){=}A{:}B \text{ IN } C\Big)$$

and

$$\textsc{cons}\,\Big(c(\vec{x}{:}\vec{A}){=}A{:}B \text{ IN } C\Big)$$

make clear what the binding structure in a term $c(\vec{x}{:}\vec{A}){=}A{:}B$ 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_j, for $j \geq i$. That is: the type of a parameter x_j 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 appears to be $B[x_j{:=}a_j]_{j=1}^{n}$);

- Moreover, we see that c is not included in the constants of

$$c(\vec{x}{:}\vec{A}){=}A{:}B \text{ IN } C.$$

  This is because c is a *local* definition, and acts as a binder for the occurrences of c in C.

Remark 6.4 There are several reasons for including the type B in a local definition $c(\vec{x}{:}\vec{A}){=}A{:}B$ IN C:

- 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 checking algorithm. Of course, this also depends on whether type checking 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 make difference between 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 6.5 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 $(\lambda \mathsf{x}{:}A.\mathsf{x})\mathsf{x}$ but $(\lambda \mathsf{y}{:}A.\mathsf{y})\mathsf{x}$. Similarly, we write $\mathsf{c}(\mathsf{x}'{:}A){=}\mathsf{x}'{:}A$ IN $\mathsf{c}(\mathsf{x})$ instead of $\mathsf{c}(\mathsf{x}{:}A){=}\mathsf{x}{:}A$ IN $\mathsf{c}(\mathsf{x})$.

Definition 6.6 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:

$$\begin{array}{rcl} c(b_1,\ldots,b_n)[x{:=}a] & \equiv & c(b_1[x{:=}a],\ldots,b_n[x{:=}a]); \\ (c(\vec{x}{:}\vec{A}) = A{:}B \text{ IN } C)[x{:=}a] & \equiv & c(x_1{:}A_1[x{:=}a],\ldots,x_n{:}A_n[x{:=}a]){=} \\ & & \quad A[x{:=}a]{:}B[x{:=}a] \text{ IN } C[x{:=}a]. \end{array}$$

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

Definition 6.7 The set of *contexts* 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. We denote contexts by $\Gamma, \Gamma', \ldots$.

Definition 6.8 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. See Remark 6.4.

In the rest of this chapter, 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 6.9 Let $\Gamma \in \mathcal{C}_P$, $M \in \mathcal{T}_P$. We define $\Gamma[x{:=}M]$ as follows:

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

For a term A we defined $\text{FV}(A)$ and $\text{CONS}(A)$. For a context Γ we do not form one set $\text{CONS}(\Gamma)$, but we split this set into a set $\text{PRIMCONS}(\Gamma)$, containing the primitive constants of Γ, and a set $\text{DEFCONS}(\Gamma)$, containing the defined constants of Γ.

Definition 6.10 Let Γ be a context. We define the free variables, constants and definitions of Γ:

Γ	$\text{FV}(\Gamma)$	$\text{PRIMCONS}(\Gamma)$	$\text{DEFCONS}(\Gamma)$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\Gamma, x{:}A$	$\text{FV}(\Gamma) \cup \{x\}$	$\text{PRIMCONS}(\Gamma)$	$\text{DEFCONS}(\Gamma)$
$\Gamma, c(\Delta){:}A$	$\text{FV}(\Gamma)$	$\text{PRIMCONS}(\Gamma) \cup \{c\}$	$\text{DEFCONS}(\Gamma)$
$\Gamma, c(\Delta){=}a{:}A$	$\text{FV}(\Gamma)$	$\text{PRIMCONS}(\Gamma)$	$\text{DEFCONS}(\Gamma) \cup \{c\}$

Finally we define the *domain* of Γ, $\text{DOM}(\Gamma)$, by

$$\text{FV}(\Gamma) \cup \text{PRIMCONS}(\Gamma) \cup \text{DEFCONS}(\Gamma).$$

In ordinary Pure Type Systems we have that, for a legal term A in a legal context Γ, $\text{FV}(A) \subseteq \text{FV}(\Gamma)$. 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: $\text{FV}(A) \subseteq \text{FV}(\Gamma)$ and $\text{CONS}(A) \subseteq \text{PRIMCONS}(\Gamma) \cup \text{DEFCONS}(\Gamma)$. 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 6.11 For a context Γ and a term A with $\text{DOM}(A) \subseteq \text{DOM}(\Gamma)$ we define

$$\begin{array}{rcl}
\text{DEFCONS}_\Gamma(A) & = & \text{CONS}(A) \cap \text{DEFCONS}(\Gamma); \\
\text{PRIMCONS}_\Gamma(A) & = & \text{CONS}(A) \cap \text{PRIMCONS}(\Gamma).
\end{array}$$

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 $(c(\Delta){=}b{:}B \text{ IN } a)$ of A, then c is a *locally defined constant*;
- If $c \in \text{DEFCONS}_\Gamma(A)$, then c is a *globally defined constant*;
- If $c \in \text{PRIMCONS}_\Gamma(A)$ (or $c \notin \text{DOM}(\Gamma)$), then c is a *primitive constant.*

Example 6.12 It is possible that $c \in \text{CONS}(A)$ is a globally defined constant with respect to a context Γ, but a primitive constant with respect to a context Γ'. Take for example $A \equiv \texttt{id}$, $\Gamma \equiv \alpha{:}{*}, \texttt{id}(){=}(\lambda \texttt{x}{:}\alpha.\texttt{x}){:}(\alpha \to \alpha)$, and $\Gamma' \equiv \alpha{:}{*}, \texttt{id}(){:}(\alpha \to \alpha)$.

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 6.13 $\Gamma \in \mathcal{C}_P$ is *sound* if $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$ implies

$$\text{DOM}(a) \cup \text{DOM}(A) \subseteq \text{DOM}(\Gamma_1) \cup \text{DOM}(\Delta)$$

and

$$\text{DOM}(B_i) \subseteq \text{DOM}(\Gamma_1, \Delta_i)\,.$$

The contexts occurring in the type systems proposed in this chapter are all sound (see Lemma 6.23). This fact will be useful when proving properties of these systems.

We will consider some extensions of Pure Type Systems (PTSs). The definition of PTSs can be found in the appendix (Definition A.20), and has already been discussed in Section 4b1.

- An extension that includes globally and locally defined constants is described and studied in [114]: "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.[1] One could agree that a parameter-free primitive constant is a special sort 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 ($\vec{\text{C}}$-PTSs). The $\vec{\text{C}}$-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: $\vec{\text{D}}$-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 $\vec{\text{C}}$D-PTSs.

Combining the various extensions, we obtain a hierarchy that can be depicted as in Figure 10.

Example 6.14 We give some examples of the possibilities of parameters and definitions.

- We illustrate the difference between PTSs, C-PTSs and $\vec{\text{C}}$-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;
  - Though the representation of objects like the type of natural numbers and the natural number zero as a variable works fine in

[1]There are, however, extensions of PTSs in which constants play an essential role. See for instance the Modal PTSs in the thesis of Borghuis [18], p. 28–29

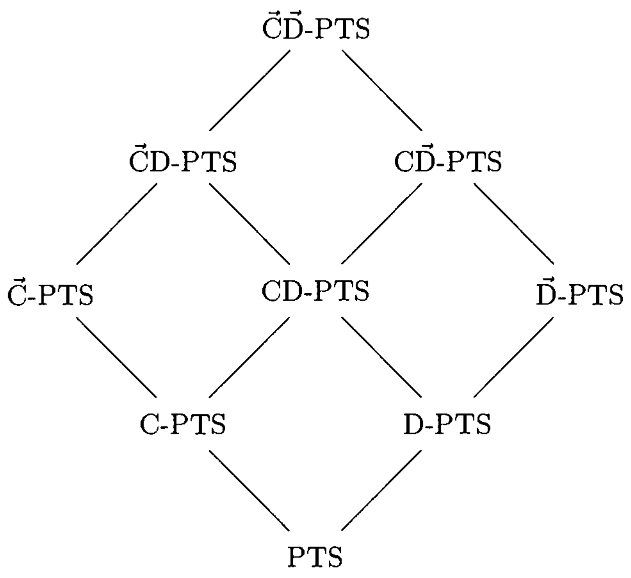


Figure 10: The hierarchy of parameters and definitions

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 $N{:}{*} \vdash \lambda o{:}N.t : N \to N$. In this judgement, o acts as a variable, while it was initially introduced as a constant.

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 Example 5.9, we introduced for each proposition Σ the type `proof`(Σ) 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 `prop` $\to$ `type` and this type cannot be constructed in $\lambda{\to}$ (notice that `type` $\equiv *$, so the construction of `prop` $\to$ `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 $\vec{\mathrm{C}}$-PTSs it is possible to introduce a parametric version of proof by the following context declaration:

$$\texttt{proof(p:prop)} : \texttt{type}.$$

This does not involve the construction of a type $\texttt{prop} \rightarrow \texttt{type}$. Nevertheless it is possible to construct the term $\texttt{prop}(P)$ for any term $P : \texttt{prop}$. We obtain a form of polymorphism without using the polymorphism of λ-calculus.

A disadvantage may be that we cannot speak about the term proof "as it is". When using proof in the syntax, it must always be applied to a parameter $T : \texttt{prop}$.

However, an advantage is that we can restrict ourselves to a much more simple type system. In the situation above we remain within the types of the system $\lambda{\rightarrow}$. We do not need to use types of the system λP. This may have advantages in implementations of type systems. For instance, the system $\lambda{\rightarrow}$ 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 6f;

- We illustrate the difference between PTSs, D-PTSs and $\vec{\mathrm{D}}$-PTSs.
  - In a simple PTS like $\lambda{\rightarrow}$ one can derive the following statement for an identity function:

$$\alpha{:}{*} \vdash (\lambda \mathrm{x}{:}\alpha.\mathrm{x}) : \alpha \rightarrow \alpha;$$

- The same derivation can be made in the corresponding D-PTS, but in that D-PTS we have the possibility of abbreviating the term $\lambda x{:}\alpha.x$. We can do this in two ways. First of all, we can introduce this definition in the context:

  $$\alpha{:}{*}, \mathtt{id}{=}(\lambda x{:}\alpha.x){:}(\alpha{\to}\alpha) \vdash \mathtt{id} : \alpha{\to}\alpha.$$

  But we can also decide to make a local definition:

  $$\alpha{:}{*} \vdash (\mathtt{id}{=}(\lambda x{:}\alpha.x){:}(\alpha{\to}\alpha) \text{ IN } \mathtt{id}) : \\ \mathtt{id}{=}(\lambda x{:}\alpha.x){:}(\alpha{\to}\alpha) \text{ IN } \alpha{\to}\alpha.$$

  We see that the definition of id appears both in the term and in the type of the term, but not in the context.

  The advantages of definitions are:

  - We can abbreviate long expressions. This makes terms more *surveyable*: id is shorter than $\lambda x{:}\alpha.x$;
  - We can give names to important expressions. This makes terms more *understandable*: id expresses that we have to do with the identity function, whilst $\lambda x{:}\alpha.x$ does not express this fact;

- In a $\vec{\mathrm{D}}$-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 $\vec{\mathrm{D}}$-PTSs we can only work with parametric definitions). We obtain (in the case of the global definition):

    $$\alpha{:}{*}, \mathtt{id}(){=}(\lambda x{:}\alpha.x){:}(\alpha{\to}\alpha) \vdash \mathtt{id}() : \alpha{\to}\alpha;$$

  - But we could also decide to use one or more parameters in the definition of id. For instance, we could parametrise the variable α. This results in the declaration

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

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 $\mathtt{x}$, 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 $\mathtt{id}$) but type α (which would also be the type of $\mathtt{id}t$ if we had used the identity $\mathtt{id}{=}\lambda\mathtt{x}{:}\alpha.\mathtt{x}$ from λ-calculus);

- Finally, one could parametrise both α and $\mathtt{x}$. 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

$$\mathtt{id}(\alpha{:}*, \mathtt{x}{:}\alpha){=}\mathtt{x}{:}\alpha \vdash \mathtt{id}(t) : T.$$

The global definitions given in the $\vec{\mathrm{D}}$-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 $\rightarrow_\beta$, $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\beta^+$, 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 ($\delta 1$) in the definition below. But now, a δ-reduction step can also unfold *local* definitions. Therefore, two new reduction steps are introduced. Rule ($\delta 2$) 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 ($\delta 3$) 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 6.15 We define the following three reduction rules:

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

$$\Gamma \vdash c(\Delta){=}a{:}A \text{ IN } b \rightarrow_\delta b \quad \text{if } c \notin \text{CONS}(b) \qquad (\delta 2)$$

$$\frac{\Gamma, c(\Delta){=}a{:}A \vdash b \rightarrow_\delta b'}{\Gamma \vdash c(\Delta){=}a{:}A \text{ IN } b \rightarrow_\delta c(\Delta){=}a{:}A \text{ IN } b'} \qquad (\delta 3)$$

Furthermore, we have some compatibility rules. These rules are not very complicated, there are only quite a lot of them.

Definition 6.16 We define the following compatibility rules:

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

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

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

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

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

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

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

Remark 6.17 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 6.26).

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

Definition 6.18 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 ($\delta1$), ($\delta2$) and ($\delta3$) of Definition 6.15 and under the compatibility rules of Definition 6.16.

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

$$\frac{\Gamma_1 \vdash A \to_\delta A'}{\Gamma_1, x{:}A, \Gamma_2 \to_\delta \Gamma_1, x{:}A', \Gamma_2}$$

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

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

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

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

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

We now describe the extensions to PTSs that are needed to obtain $\vec{\mathrm{C}}$-PTSs and $\vec{\mathrm{D}}$-PTSs. We don't discuss D-PTSs and $\vec{\mathrm{C}}$D-PTSs: D-PTSs are introduced in [114] and $\vec{\mathrm{C}}$D-PTSs can be constructed by extending D-PTSs with the additional rules for $\vec{\mathrm{C}}$-PTSs.

Definition 6.20 ($\vec{\mathrm{C}}$-PTS: Pure type systems with parametric constants) The *typing relation* $\vdash^{\vec{C}}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition A.20 and the following ones (we still write $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$):

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

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

where $s \in \boldsymbol{S}$ and the c that is introduced in the $\vec{\mathrm{C}}$-weakening rule is assumed to be Γ-fresh.

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

The extra condition $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash^{\vec{C}} A : s$ in the $\vec{\mathrm{C}}$-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.

Adapting these rules for $\vdash^{\vec{C}}$ and the rules for definitions of [114] results in rules for *parametric definitions*:

Definition 6.21 ($\vec{\mathrm{D}}$-PTS: Pure type systems with parametric definitions) The *typing relation* $\vdash^{\vec{D}}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition A.20 and the following ones:

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

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

($\vec{\mathrm{D}}$-form)
$$\frac{\Gamma, c(\Delta){=}a{:}A \vdash^{\vec{D}} B : s}{\Gamma \vdash^{\vec{D}} c(\Delta){=}a{:}A \text{ IN } B : s}$$

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

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

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

$\vdash^{\vec{D}}$ includes the definition system of [114]: The $\vec{\mathrm{D}}$-application rule for $n = 0$ can be seen as the δ-start rule of D-PTSs.

Definition 6.22 (Pure Type Systems with (parametric) constants and (parametric) definitions)
Let $\mathfrak{S}$ be a specification (see A.17).

- A *pure type system with (parametric) constants* $\vec{\mathrm{C}}$-PTS is denoted as $\lambda^{\vec{C}}(\mathfrak{S})$ 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 $\vdash^{\vec{C}}$;
- A *pure type system with (parametric) definitions* $\vec{\mathrm{D}}$-PTS is denoted as $\lambda^{\vec{D}}(\mathfrak{S})$ 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 $\vdash^{\vec{D}}$;
- A *pure type system with (parametric) constants and (parametric) definitions* $\vec{\mathrm{C}}\vec{\mathrm{D}}$-PTS is denoted as $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ 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 $\vdash^{\vec{C}\vec{D}}$, which is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ that is closed under the rules of Definition A.20 and the rules of $\vdash^{\vec{C}}$ and $\vdash^{\vec{D}}$.

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 $\vec{\mathrm{C}}\vec{\mathrm{D}}$-PTSs are sound (see Definition 6.13). As $\vec{\mathrm{C}}\vec{\mathrm{D}}$-PTSs are clearly extensions of PTSs, $\vec{\mathrm{C}}$-PTSs and $\vec{\mathrm{D}}$-PTSs, this implies that all contexts occurring in PTSs, $\vec{\mathrm{C}}$-PTSs and $\vec{\mathrm{D}}$-PTSs are sound. We need this fact in many proofs in the next sections. The proof of the lemma below is by induction on the derivation of $\Gamma \vdash^{\vec{C}\vec{D}} a : A$.

Lemma 6.23 *Assume* $\Gamma \vdash^{\vec{C}\vec{D}} b : B$.

1. $\mathrm{DOM}(b), \mathrm{DOM}(B) \subseteq \mathrm{DOM}(\Gamma)$*;*
2. Γ *is sound.*

PROOF: We prove the statements (1) and (2) simultaneously by induction on the derivation of $\Gamma \vdash^{\vec{C}\vec{D}} b : B$. We treat the two most important cases:

- ($\vec{\mathrm{D}}$-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);
- ($\vec{\mathrm{D}}$-formation) $\Gamma \vdash (c(\Delta){=}a{:}A \text{ IN } B) : s$ because $\Gamma, c(\Delta){=}a{:}A \vdash B : s$. (1) follows from the induction hypothesis for (2); (2) is trivial.

$\boxtimes$

6b Properties of terms

In this section, we prove properties of terms without wondering whether these terms are legal or not. In Section 6b1 we discuss some basic properties, such as a Substitution Lemma, and substitutivity. Section 6b2 is devoted to the Church-Rosser property for $\beta\delta$-reduction, and in Section 6b3 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.

6b1 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 $\rightarrow_\delta$.

Lemma 6.24 **($\rightarrow_\delta$-weakening)** *Let* $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle \in \mathcal{C}_P$ *be such that*

$$\Gamma_1, \Gamma_3 \vdash b \rightarrow_\delta b'.$$

Then

$$\Gamma_1, \Gamma_2, \Gamma_3 \vdash b \rightarrow_\delta b'.$$

⊠

The implications from left to right of the following lemma are a particular case of Lemma 6.24.

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 $\rightarrow_{\beta\delta}$. The last part states that declarations of the form $c(\Delta){=}a{:}A$ in a context do not have any influence on the $\rightarrow_{\beta\delta}$ reduction behaviour of terms $b \in \mathcal{T}_P$ with $c \notin \text{CONS}(b)$. This allows to remove definition declarations, as rule (δ2) of the definition of δ-reduction does for local definitions.

The lemma is proved by induction on the definition of $\rightarrow_\beta$ and $\rightarrow_\delta$.

Lemma 6.25

1. *Let* $\langle\Gamma_1, x{:}A, \Gamma_2\rangle \in \mathcal{C}_P$ *and* $b \in \mathcal{T}_P$.
   $\Gamma_1, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$ *if and only if* $\Gamma_1, x{:}A, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$;

2. *Let* $\langle\Gamma_1, c(\Delta){:}A, \Gamma_2\rangle \in \mathcal{C}_P$ *and* $b \in \mathcal{T}_P$.
   $\Gamma_1, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$ *if and only if* $\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$;

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* $c \notin \text{CONS}(b)$.
   $\Gamma_1, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$ *if and only if* $\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash b \rightarrow_{\beta\delta} b'$.

⊠

Now we show that the compatibility rule for $c(\Delta){=}a{:}A$ IN b when we reduce inside b is a derived rule (and therefore not included in the list of compatibility rules in Definition 6.16).

Lemma 6.26 *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 6.24, $\Gamma, c(\Delta){=}a{:}A \vdash b \to_\delta b'$. 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 6.27 (Substitution Lemma) *Suppose $x \not\equiv y$ and $x \notin \text{FV}(d)$. 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 6.28 (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. For example, let

$$\Gamma \equiv \mathsf{x}{:}\alpha, \mathsf{x}'{:}\alpha, \mathsf{c}(){=}\mathsf{x}{:}\alpha.$$

We have

$$\Gamma \vdash^{\vec{D}} \mathsf{c}() : \alpha$$

and

$$\Gamma \vdash \mathsf{c}() \to_\delta \mathsf{x},$$

but not

$$\Gamma \vdash \mathsf{c}()[\mathsf{x}{:=}\mathsf{x}'] \to_\delta \mathsf{x}[\mathsf{x}{:=}\mathsf{x}'].$$

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 following version of substitutivity. It is adapted so that the substitution now occurs in the context as well. The proof is by induction on the derivation of $\Gamma \vdash a \to_\delta a'$.

Lemma 6.29 (Weak 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 $\Gamma \vdash a \to_\delta a'$. We only consider the two most interesting cases:

- $\Gamma_1, c(\Delta){=}d{:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n) \to_\delta d[x_i{:=}b_i]_{i=1}^n$. ($\delta 1$)
  Now
  $$\langle \Gamma_1, c(\Delta){=}d{:}A, \Gamma_2 \rangle [x{:=}b] \equiv \\ \langle \Gamma_1[x{:=}b], c(\Delta[x{:=}b]){=}d[x{:=}b]{:}A[x{:=}b], \Gamma_2[x{:=}b] \rangle$$
  so
  $$\langle \Gamma_1, c(\Delta){=}d{:}A, \Gamma_2 \rangle [x{:=}b] \vdash \\ c(b_1[x{:=}b], \ldots, b_n[x{:=}b]) \to_\delta d[x{:=}b][x_i{:=}b_i[x{:=}b]]_{i=1}^n$$
  and as the x_i are bound in $\langle \Gamma_1, c(\Delta){=}d{:}A, \Gamma_2 \rangle$, $x_i \notin \mathrm{FV}(b)$ by the variable convention, so by the Substitution Lemma
  $$d[x{:=}b][x_i{:=}b_i[x{:=}b]]_{i=1}^n \equiv \left(d[x_i{:=}b_i]_{i=1}^n\right)[x{:=}b];$$
- $c \notin \mathrm{CONS}(a)$ and $\Gamma \vdash c(\Delta){=}d{:}A$ IN $a \to_\delta a$ ($\delta 2$). We have that c is bound in $c(\Delta){=}d{:}A$ IN a, so by the variable convention $c \notin \mathrm{CONS}(b)$, so $c \notin \mathrm{CONS}(a[x{:=}b])$. Hence
  $$\Gamma[x{:=}b] \vdash (c(\Delta){=}d{:}A \text{ IN } a)[x{:=}b] \to_\delta a[x{:=}b].$$

⊠

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

6b2 Church-Rosser for $\to_{\beta\delta}$

In this section we prove the Church-Rosser theorem for $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\delta$ and $\twoheadrightarrow_{\beta\delta}$. As for ordinary λ-terms, we have:

Theorem 6.31 (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 6.32 For $a \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$ we define a term $|a|_\Gamma \in \mathcal{T}_P$ as follows:

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

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

Lemma 6.33

- $|b|_{\Gamma_1,\Gamma_2} \equiv |b|_{\Gamma_1,x:A,\Gamma_2}$;

- $|b|_{\Gamma_1,\Gamma_2} \equiv |b|_{\Gamma_1,c(\Delta):A,\Gamma_2}$.

⊠

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

Lemma 6.34 *For all $b \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$, $|b|_\Gamma$ has no subterms of the form $(c(\Delta){=}a{:}A \text{ IN } d)$.* ⊠

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, $\Gamma \equiv \langle \mathsf{c}(){=}\mathsf{c}'(){:}*, \mathsf{c}'(){=}\mathsf{c}''(){:}*\rangle$. Then $|\mathsf{c}()|_\Gamma \equiv |\mathsf{c}'()|_\varnothing \equiv \mathsf{c}'()$, but $\mathsf{c}'()$ is not in δ-normal form with respect to Γ. This is due to the fact that Γ is not a sound context (see 6.13).

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 $\vec{\text{C}}\vec{\text{D}}$-PTSs are sound (Lemma 6.23).

Lemma 6.35 *Let Γ be a sound context such that* $\text{DOM}(b) \subseteq \text{DOM}(\Gamma)$. *Then* $\text{DOM}(|b|_\Gamma) \subseteq \text{DOM}(\Gamma) \setminus \text{DEFCONS}(\Gamma)$.

PROOF: 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 $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle$.

$$\begin{array}{rcl}
\text{DOM}(|b|_\Gamma) & = & \text{DOM}\left(|a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n\right) \\
& \subseteq & \left(\text{DOM}\left(|a|_{\Gamma_1,\Delta}\right) \setminus \{x_1,\ldots,x_n\}\right) \cup \bigcup_{i=1}^n \text{DOM}(|b_i|_\Gamma) \\
& \overset{(\text{IH})}{\subseteq} & (\text{DOM}(\Gamma_1,\Delta) \setminus \text{DEFCONS}(\Gamma_1)) \setminus \{x_1,\ldots,x_n\} \\
& & \qquad \cup\, (\text{DOM}(\Gamma) \setminus \text{DEFCONS}(\Gamma)) \\
& = & \text{DOM}(\Gamma) \setminus \text{DEFCONS}(\Gamma)\,.
\end{array}$$

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

- $b \equiv c(\Delta){=}a{:}A$ IN b'.

$$\begin{aligned}\text{DOM}\left(|b|_\Gamma\right) &= \text{DOM}\left(|b'|_{\Gamma,c(\Delta)=a:A}\right)\\ &\overset{\text{(IH)}}{\subseteq} \text{DOM}\left(\Gamma, c(\Delta){=}a{:}A\right) \setminus \text{DEFCONS}\left(\Gamma, c(\Delta){=}a{:}A\right)\\ &= \text{DOM}(\Gamma) \setminus \text{DEFCONS}(\Gamma)\,.\end{aligned}$$

⊠

With the above we can show:

Lemma 6.36 *If* Γ *is sound and* DOM$(d) \subseteq$ DOM(Γ), *then*

$$\Gamma \vdash d \twoheadrightarrow_\delta |d|_\Gamma.$$

Proof: Induction on the total number of symbols occurring in Γ and d. We treat the two most important cases:

- $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2\rangle$ and $d \equiv c(b_1, \dots, b_n)$.
Notice that $\Gamma \vdash d \rightarrow_\delta a[x_i{:=}b_i]_{i=1}^n$.
By induction, $\Gamma_1, \Delta \vdash a \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}$, so $\Gamma \vdash a \twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}$ (Lemma 6.24), so by Lemma 6.29,

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

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

$$\begin{aligned}\Gamma \vdash a[x_i{:=}b_i]_{i=1}^n &\twoheadrightarrow_\delta |a|_{\Gamma_1,\Delta}[x_i{:=}b_i]_{i=1}^n\\ &\overset{\text{(IH, 6.30)}}{\twoheadrightarrow_\delta} |a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n\\ &\equiv |c(b_1, \dots, b_n)|_\Gamma;\end{aligned}$$

- $d \equiv c(\Delta){=}a{:}A$ IN b. By the induction hypothesis,

$$\Gamma, c(\Delta){=}a{:}A \vdash b \twoheadrightarrow_\delta |b|_{\Gamma,c(\Delta)=a:A}$$

hence

$$\Gamma \vdash c(\Delta){=}a{:}A \text{ IN } b \twoheadrightarrow_\delta c(\Delta){=}a{:}A \text{ IN } |b|_{\Gamma,c(\Delta)=a:A}.$$

Now $\text{DOM}(d) \subseteq \text{DOM}(\Gamma)$, so $\text{DOM}(b) \subseteq \text{DOM}(\Gamma, c(\Delta){=}a{:}A)$, so by Lemma 6.35,

$$\text{DOM}\left(|b|_{\Gamma,c(\Delta)=a:A}\right) \subseteq \text{DOM}(\Gamma, c(\Delta){=}a{:}A) \setminus \text{DEFCONS}(\Gamma, c(\Delta){=}a{:}A),$$

so $c \notin \text{CONS}\left(|b|_{\Gamma,c(\Delta)=a:A}\right)$, so

$$\Gamma \vdash d \twoheadrightarrow_\delta c(\Delta){=}a{:}A \text{ IN } |b|_{\Gamma,c(\Delta)=a:A} \to_\delta |b|_{\Gamma,c(\Delta)=a:A}.$$

⊠

Corollary 6.37 *In any $\vec{C}\vec{D}$-PTS, the relation $\to_\delta$ is weakly normalising, i.e. each legal term has a β-normal form.*

PROOF: By Lemma 6.34 and Lemma 6.35, $|b|_\Gamma$ is in δ-normal form; by Lemma 6.36, $|b|_\Gamma$ is a δ-normal form of b. ⊠

The mapping $|-|_{\_}$ also helps us to show that $\to_{\beta\delta}$ is confluent (Theorem 6.42). For the proof we use some lemmas:

Lemma 6.38 *Assume $\langle\Gamma_1, \Gamma_3\rangle$ is sound and* $\text{DOM}(b) \subseteq \text{DOM}(\Gamma_1, \Gamma_3)$. *Then $|b|_{\Gamma_1,\Gamma_2,\Gamma_3} \equiv |b|_{\Gamma_1,\Gamma_3}$.*

PROOF: 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 $\Gamma_3 \equiv \langle\Gamma_{31}, c(\Delta){=}a{:}A, \Gamma_{32}\rangle$.

$$\begin{array}{rcl} |c(b_1,\ldots,b_n)|_{\Gamma_1,\Gamma_3} & \equiv & |a|_{\Gamma_1,\Gamma_{31},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_3}] \\ & \overset{\text{(IH,6.30)}}{\equiv} & |a|_{\Gamma_1,\Gamma_2,\Gamma_{31},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_2,\Gamma_3}] \\ & \equiv & |c(b_1,\ldots,b_n)|_{\Gamma_1,\Gamma_2,\Gamma_3}; \end{array}$$

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

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

⊠

Lemma 6.39 *Assume* $\langle\Gamma_1,\Gamma_2\rangle$ *is sound, and* $\text{DOM}(a) \subseteq \text{DOM}(\Gamma_1,\Gamma_2)$, $\text{DOM}(b) \subseteq \text{DOM}(\Gamma_1)$ *and* $x \notin \text{DOM}(\Gamma_1)$. *Then*

$$|a|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}] \equiv |a[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}.$$

PROOF: 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 $\Gamma_2 \equiv \Gamma_{21}, c(\Delta){=}c'{:}C, \Gamma_{22}$.

$$\begin{aligned}&|a|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}]\\ &\equiv |c'|_{\Gamma_1,\Gamma_{21},\Delta}[x_i{:=}|b_i|_{\Gamma_1,\Gamma_2}]_{i=1}^{n}[x{:=}|b|_{\Gamma_1}]\\ &\overset{(6.27)}{\equiv} |c'|_{\Gamma_1,\Gamma_{21},\Delta}[x{:=}|b|_{\Gamma_1}][x_i{:=}|b_i|_{\Gamma_1,\Gamma_2}[x{:=}|b|_{\Gamma_1}]]_{i=1}^{n}\\ &\overset{\text{(IH)}}{\equiv} |c'[x{:=}b]|_{\Gamma_1,\Gamma_{21}[x:=b],\Delta[x:=b]}[x_i{:=}|b_i[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]}]_{i=1}^{n}\\ &\equiv |c(b_1,\ldots,b_n)[x{:=}b]|_{\Gamma_1,\Gamma_2[x:=b]};\end{aligned}$$

- $a \equiv c(\Delta){=}c'{:}C \text{ IN } d$.

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

⊠

Lemma 6.40 *If* $\Gamma \vdash d \to_\delta d'$, Γ *is sound, and* $\text{DOM}(d) \subseteq \text{DOM}(\Gamma)$, *then* $|d|_\Gamma \equiv |d'|_\Gamma$ *and* $\text{DOM}(d') \subseteq \text{DOM}(\Gamma)$.

PROOF: We prove the following two statements simultaneously by induction on the definition of $|d|_\Gamma$:

- If $\Gamma \vdash d \to_\delta d'$ then $|d|_\Gamma \equiv |d'|_\Gamma$;
- If $\Gamma \to_\delta \Gamma'$ then $|d|_\Gamma \equiv |d|_{\Gamma'}$.

We prove the two non-trivial cases:

- $\Gamma \equiv \langle \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \rangle$, $d \equiv c(b_1, \ldots, b_n)$, and $d' \equiv a[x_i{:=}b_i]_{i=1}^n$.

$$\begin{array}{rcl} |d|_\Gamma & \equiv & |a|_{\Gamma_1,\Delta}[x_i{:=}|b_i|_\Gamma]_{i=1}^n \\ & \overset{(6.33)}{\equiv} & |a|_{\Gamma_1}[x_i{:=}|b_i|_\Gamma]_{i=1}^n \\ & \overset{(6.38)}{\equiv} & |a|_\Gamma[x_i{:=}|b_i|_\Gamma]_{i=1}^n \\ & \overset{(6.39)}{\equiv} & |a[x_i{:=}b_i]_{i=1}^n|_\Gamma \\ & \equiv & |d'|_\Gamma. \end{array}$$

  The several cases that have to be distinguished for $\Gamma \to_\delta \Gamma'$ are all easy to prove;

- $d \equiv c(\Delta){=}a{:}A$ IN b. Use induction on $\Gamma \vdash d \to_\delta d'$. We treat two cases:

  - $d' \equiv b$ and $c \notin$ CONS(b). Then $|d|_\Gamma \equiv |b|_{\Gamma,c(\Delta)=a:A} \overset{(6.38)}{\equiv} |b|_\Gamma \equiv |d'|_\Gamma$;
  - $d' \equiv c(\Delta){=}a{:}A$ IN b' and $\Gamma, c(\Delta){=}a{:}A \vdash b \to_\delta b'$. Then $|d|_\Gamma \equiv |b|_{\Gamma,c(\Delta)=a:A} \overset{(\mathrm{IH})}{\equiv} |b'|_{\Gamma,c(\Delta)=a:A} \equiv |d'|_\Gamma$.

  If $\Gamma \to_\delta \Gamma'$ then $|d|_\Gamma \equiv |b|_{\Gamma,c(\Delta)=a:A} \overset{(\mathrm{IH})}{\equiv} |b|_{\Gamma',c(\Delta)=a:A} \equiv |d|_{\Gamma'}$.

⊠

Lemma 6.41 *If Γ is sound,* DOM$(d) \subseteq$ DOM(Γ) *and $d \to_\beta d'$, then* $|d|_\Gamma \twoheadrightarrow_\beta |d'|_\Gamma$. ⊠

The proof is similar to the proof of Lemma 6.40.

Theorem 6.42 (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.

$$\begin{tikzcd}
& b_2 \arrow[r, two heads, "6.36"', "\delta"'] & {|b_2|_\Gamma} \arrow[dr, two heads, "6.31", "\beta"'] & \\
a \arrow[ur, two heads, "\beta\delta"] \arrow[r, two heads, "6.36", "\delta"'] \arrow[dr, two heads, "\beta\delta"'] & {|a|_\Gamma} \arrow[ur, two heads, "6.41\ 6.40"', "\beta"] \arrow[dr, two heads, "6.40\ 6.41", "\beta"'] & & d \\
& b_1 \arrow[r, two heads, "6.36"', "\delta"'] & {|b_1|_\Gamma} \arrow[ur, two heads, "6.31", "\beta"'] &
\end{tikzcd}$$

⊠

6b3 Strong normalisation for $\to_\delta$

In [40], 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 [119]. A similar technique to the one of de Vrijer is also used in [114] to prove strong normalisation for δ-reduction in $\vec{\mathrm{D}}$-PTSs. We extend these techniques to prove strong normalisation for δ-reduction in $\vec{\mathrm{C}}\vec{\mathrm{D}}$-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 6.43 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.

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

Following the line of [119] one can prove the following lemma (using induction on the definitions of $\mathtt{M}_{\_}(-,-)$):

Lemma 6.44

1. *If* Γ *is sound,* $\text{DOM}(a) \subseteq \text{DOM}(\Gamma)$ *and* $x \notin \text{FV}(a) \cup \text{FV}(\Gamma)$, *then* $\mathtt{M}_x(\Gamma, a) = 0$*;*

2. *If* $\langle \Gamma_1, \Gamma_3 \rangle$ *is sound and* $\text{DOM}(a) \subseteq \text{DOM}(\langle \Gamma_1, \Gamma_3 \rangle)$, *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,* $\text{DOM}(b) \subseteq \text{DOM}(\langle \Gamma_1, \Gamma_2 \rangle)$, $\text{DOM}(a) \subseteq \text{DOM}(\Gamma_1)$, $x \not\equiv z$ *and* $x \notin \text{FV}(\Gamma_1)$, *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 [119], as contexts are involved in our situation.

Lemma 6.45 *Let Γ be sound,* $\text{DOM}(a) \subseteq \text{DOM}(\Gamma)$. *If $\Gamma \vdash a \to_\delta b$, 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 $\Gamma \vdash a \to_\delta b$, then $\mathtt{M}_x(\Gamma, a) \geq \mathtt{M}_x(\Gamma, b)$;
2. If $\Gamma \to_\delta \Gamma'$, 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 [119] (used to prove the finite developments theorem), in [40] and in [114] (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 6.46 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:

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

Similar properties as in Lemma 6.44 and Lemma 6.45 hold for $\mathtt{L}_{-}(-)$:

Lemma 6.47

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

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

Lemma 6.48 *If* Γ *is sound,* $\text{DOM}(a) \subseteq \text{DOM}(\Gamma)$ *and* $\Gamma \vdash a \to_\delta b$, *then* $\text{L}_\Gamma(a) > \text{L}_\Gamma(b)$.

PROOF: Similar to the proof of Lemma 6.45. ⊠

Theorem 6.49 (Strong Normalisation for δ) *The reduction δ (when restricted to sound contexts Γ and terms a with* $\text{DOM}(a) \subseteq \text{DOM}(\Gamma)$*) is strongly normalising, i.e. there are no infinite δ-reduction paths.*

PROOF: This follows from lemma 6.48. ⊠

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

6c 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 $\Gamma \vdash^{\vec{C}\vec{D}} a : A$ or $\Gamma \vdash^{\vec{C}\vec{D}} A : a$. The main property we prove is that strong normalisation is preserved by certain extensions.

Many of the standard properties of PTSs in [5], [54] hold for $\vec{\text{C}}\vec{\text{D}}$-PTSs as well. In the same way as in [5], [54] we can prove the Substitution Lemma, Correctness of Types, Subject Reduction (for $\beta\delta$-reduction) and Uniqueness of Types (for singly sorted $\vec{\text{C}}\vec{\text{D}}$-PTSs):

Theorem 6.50 *Let $\mathfrak{S}$ be a specification. The type system $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ has the following properties:*

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

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

The Generation Lemma is extended with two extra cases:

Lemma 6.51 (Generation Lemma, extension)

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

⊠

In case 1(b) we do not necessarily have $\Gamma_1, \Delta \vdash^{\vec{C}\vec{D}} A : s$. For instance, in the $\vec{C}\vec{D}$-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 $\Gamma \vdash^{\vec{C}\vec{D}} b : B$.

Lemma 6.52 (Correctness of Contexts)

1. *If $\Gamma, x{:}A, \Gamma'$ is legal then there exists a sort s such that $\Gamma \vdash^{\vec{C}\vec{D}} A : s$;*

2. If $\Gamma, c(\Delta){:}A, \Gamma'$ *is legal then* $\Gamma, \Delta \vdash^{\vec{C}\vec{D}} A : s$;

3. If $\Gamma, c(\Delta){=}a{:}A, \Gamma'$ *is legal then* $\Gamma, \Delta \vdash^{\vec{C}\vec{D}} a : A$.

⊠

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

Now we prove that $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ is $\beta\delta$-strongly normalising if a slightly larger PTS $\lambda(\mathfrak{S}')$ is β-strongly normalising. The proof follows the same ideas of [114] 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 $\vec{C}\vec{D}$-PTS $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$, then $\|a\|_\Gamma$ will be typable in a PTS $\lambda(\mathfrak{S}')$, where $\mathfrak{S}'$ is a so-called *completion* (see Definition 6.60) of the specification $\mathfrak{S}$. 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 6.49), this guarantees that $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ is $\beta\delta$-strongly normalising whenever $\lambda(\mathfrak{S}')$ 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 $\lambda(\mathfrak{S}')$.

Δ still denotes a list of variables $x_1{:}B_1, \ldots, x_n{:}B_n$ 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 6.53 For $a \in \mathcal{T}_P$ and $\Gamma \in \mathcal{C}_P$ we define $\|a\|_\Gamma$ as follows:

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

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

The mapping $\|\text{-}\|_{-}$ is extended to contexts.

Definition 6.54 For a context $\Gamma \in \mathcal{C}_P$ we define $\|\Gamma\|$ as follows:

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

We have similar properties for $\|\text{-}\|_{-}$ as for $|\text{-}|_{-}$:

Lemma 6.55 *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 6.38.

Lemma 6.56 *Assume* $\langle\Gamma_1,\Gamma_2\rangle$ *is sound, and* DOM$(a) \subseteq$ DOM$(\langle\Gamma_1,\Gamma_2\rangle)$, DOM$(b) \subseteq$ DOM(Γ_1), *and* $x \notin$ DOM(Γ_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 6.39.

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

Lemma 6.57 *Let* Γ *be sound, and assume* DOM$(a) \subseteq$ DOM(Γ). *If* $a \to_{\delta} b$ *then* $\|a\|_{\Gamma} \twoheadrightarrow_{\beta} \|b\|_{\Gamma}$.

PROOF: Using induction on the definition of $\Gamma \vdash a \to_{\delta} b$, we simultaneously prove:

1. If $a \to_\delta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta \|b\|_\Gamma$;
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:

$$\begin{array}{rcl}
\|c(b_1,\ldots,b_n)\|_\Gamma & \equiv & \|\lambda\,\Delta.a\|_{\Gamma_1}\,\|b_1\|_\Gamma\cdots\|b_n\|_\Gamma \\
& \equiv & \left(\overset{n}{\underset{i=1}{\lambda}}\, x_i{:}\,\|B_i\|_{\Gamma_1,\Delta_i}\,.\,\|a\|_{\Gamma,\Delta}\right)\|b_1\|_\Gamma\cdots\|b_n\|_\Gamma \\
& \twoheadrightarrow_\beta^n & \|a\|_{\Gamma_1,\Delta}\,[x_i{:=}\,\|b_i\|_\Gamma]_{i=1}^n \\
& \overset{(6.55)}{\equiv} & \|a\|_{\Gamma_1}\,[x_i{:=}\,\|b_i\|_\Gamma]_{i=1}^n \\
& \overset{(6.55)}{\equiv} & \|a\|_{\Gamma}\,[x_i{:=}\,\|b_i\|_\Gamma]_{i=1}^n \\
& \overset{(6.56)}{\equiv} & \|a[x_i{:=}b_i]_{i=1}^n\|_\Gamma\,;
\end{array}$$

- $\Gamma \vdash c(\Delta)=a{:}A$ IN $b \to_\delta b$ because $c \notin$ CONS(b). Then $c \notin$ FV$(\|b\|_\Gamma)$. Hence

$$\begin{array}{rcl}
\|c(\Delta)=a{:}A \text{ IN } b\|_\Gamma & \equiv & \left(\lambda c : \|\prod\Delta.A\|_\Gamma\,.\,\|b\|_{\Gamma,c(\Delta)=a:A}\right)\|\lambda\,\Delta.a\|_\Gamma \\
& \to_\beta & \|b\|_{\Gamma,c(\Delta)=a:A}\,[c{:=}\,\|\lambda\,\Delta.a\|_\Gamma] \\
& \equiv & \|b\|_\Gamma\,.
\end{array}$$

⊠

Lemma 6.58 *Let* Γ *be sound, and assume* DOM$(a) \subseteq$ DOM(Γ). *If* $a \to_\beta b$ *then* $\|a\|_\Gamma \twoheadrightarrow_\beta^+ \|b\|_\Gamma$.

PROOF: The following two statements are proved simultaneously by induction on the structure of a.

1. If $a \to_\beta b$ then $\|a\|_\Gamma \twoheadrightarrow_\beta^+ \|b\|_\Gamma$;
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:

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

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

$$\begin{aligned}
\|(\lambda x{:}p.q)r\|_\Gamma &\equiv \left(\lambda x{:}\,\|p\|_\Gamma\,.\,\|q\|_{\Gamma,x:p}\right)\|r\|_\Gamma \\
&\to_\beta \|q\|_{\Gamma,x:p}\,[x{:=}\,\|r\|_\Gamma] \\
&\overset{(6.56)}{\equiv} \|q[x{:=}r]\|_\Gamma;
\end{aligned}$$

- $c(\Delta){=}a{:}A$ IN $b \to_\beta c(\Delta'){=}a{:}A$ IN b, where
  - $\Delta' \equiv x_1{:}B_1,\ldots,x_j{:}B'_j,\ldots,x_n : B_n$;
  - $B_j \to_\beta B'_j$.

  Write $C_i \equiv B_i$ if $i \neq j$ and $C_j \equiv B'_j$. Furthermore, let $\Delta'_i \equiv x_1{:}C_1,\ldots,x_{i-1}{:}C_{i-1}$. Observe that

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

⊠

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 6.59 The specification $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ 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 6.60 A specification $\mathfrak{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ is a *completion* of $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ if

1. $\boldsymbol{S} \subseteq \boldsymbol{S}'$, $\boldsymbol{A} \subseteq \boldsymbol{A}'$, and $\boldsymbol{R} \subseteq \boldsymbol{R}'$;
2. 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 6.61 *Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $\mathfrak{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that $\mathfrak{S}'$ is a completion of $\mathfrak{S}$. If $\Gamma \vdash^{\vec{C}\vec{D}}_{\mathfrak{S}} a : A$ then*

$$\|\Gamma\| \vdash_{\mathfrak{S}'} \|a\|_{\Gamma} : \|A\|_{\Gamma}.$$

PROOF: Induction on the derivation of $\Gamma \vdash^{\vec{C}\vec{D}}_{\mathfrak{S}} a : A$. 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:

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

  Write $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$. If $n = 0$ then we know by induction that

$$\|\Gamma\| \vdash_{\mathfrak{S}'} \|a\|_{\Gamma} : \|A\|_{\Gamma}$$

  and we are done because $\|c(b_1, \ldots, b_n)\|_{\Gamma} \equiv \|a\|_{\Gamma_1} \overset{(6.55)}{\equiv} \|a\|_{\Gamma}$.
  Now assume $n > 0$. As we have a derivation of $\Gamma_1, c(\Delta){=}a{:}A, \Gamma_2 \vdash^{\vec{C}\vec{D}} b_1{:}B_1$, we can use Correctness of Contexts to find a (shorter) derivation of $\Gamma_1, \Delta \vdash^{\vec{C}\vec{D}} a{:}A$. By the induction hypothesis, we have

$$\|\Gamma_1, \Delta\| \vdash_{\mathfrak{S}'} \|a\|_{\Gamma_1,\Delta} : \|A\|_{\Gamma_1,\Delta'}. \tag{1}$$

Moreover, we can use the induction hypothesis to find

$$\|\Gamma_1\|, c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1}, \|\Gamma_2\| \vdash_{\mathfrak{S}'} \|b_i\|_\Gamma : \left\|B_i[x_j{:=}b_j]_{j=1}^{i-1}\right\|_\Gamma. \quad (2)$$

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

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \|\textstyle\prod \Delta.A\|_{\Gamma_1} : s. \quad (3)$$

Using rule (λ), (1) and (3) result in $\|\Gamma_1\| \vdash_{\mathfrak{S}'} \|\lambda\,\Delta.a\|_{\Gamma_1} : \|\prod \Delta.A\|_{\Gamma_1}$. By definition of $\|\prod \Delta.A\|_{\Gamma_1}$, this means

$$\|\Gamma\| \vdash_{\mathfrak{S}'} \|\lambda\,\Delta.a\|_{\Gamma_1} : \textstyle\prod_{i=1}^n x_i{:}\, \|B_i\|_{\Gamma_1,\Delta} \cdot \|A\|_{\Gamma_1,\Delta}. \quad (4)$$

By Lemma 6.56, $\left\|B_i[x_j{:=}b_j]_{j=1}^{i-1}\right\|_\Gamma \equiv \|B_i\|_{\Gamma,\Delta_i}[x_j{:=}\, \|b_j\|_\Gamma]_{j=1}^{i-1}$. Using (2) and the application rule, we can derive from (4) that:

$$\|\Gamma\| \vdash_{\mathfrak{S}'} (\|\lambda\,\Delta.a\|_{\Gamma_1} \|b_1\|_\Gamma \cdots \|b_n\|_\Gamma) : \left(\|A\|_{\Gamma_1,\Delta}[x_j{:=}\, \|b_j\|_\Gamma]_{j=1}^n\right).$$

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

$$\begin{aligned} \|A\|_{\Gamma_1,\Delta}[x_j{:=}\, \|b_j\|_\Gamma]_{j=1}^n &\overset{(6.55)}{\equiv} \|A\|_{\Gamma,\Delta}[x_j{:=}\, \|b_j\|_\Gamma]_{j=1}^n \\ &\overset{(6.56)}{\equiv} \left\|A[x_j{:=}b_j]_{j=1}^n\right\|_\Gamma; \end{aligned}$$

- δ-weakening:

$$\frac{\Gamma \vdash^{\vec{C}\vec{D}} b : B \qquad \Gamma, \Delta \vdash^{\vec{C}\vec{D}} a : A}{\Gamma, c(\Delta){=}a{:}A \vdash^{\vec{C}\vec{D}} b : B}.$$

By induction, $\|\Gamma, \Delta\| \vdash_{\mathfrak{S}'} \|a\|_{\Gamma,\Delta} : \|A\|_{\Gamma,\Delta}$, so

$$\|\Gamma\|, x_1{:}\, \|B_1\|_{\Gamma,\Delta_1}, \ldots, x_n{:}\, \|B_n\|_{\Gamma,\Delta_n} \vdash_{\mathfrak{S}'} \|a\|_{\Gamma,\Delta} : \|A\|_{\Gamma,\Delta}. \quad (5)$$

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

$$\|\Gamma\|, x_1{:}\, \|B_1\|_{\Gamma,\Delta_1}, \ldots, x_{i-1}{:}\, \|B_{i-1}\|_{\Gamma,\Delta_{i-1}} \vdash_{\mathfrak{S}'} \|B_i\|_{\Gamma,\Delta_i} : s_i. \quad (6)$$

By Correctness of Types for $\lambda_{\mathfrak{S}}^{\vec{C}\vec{D}}$, there are two possibilities:

- There is $s \in \boldsymbol{S}$ such that $A \equiv s$. As $\mathfrak{S}'$ is a completion of $\mathfrak{S}$, there is $s' \in \boldsymbol{S}'$ such that $\|\Gamma, \Delta\| \vdash_{\mathfrak{S}'} s : s'$.
- There is $s' \in \boldsymbol{S}$ such that $\Gamma, \Delta \vdash^{\vec{C}\vec{D}} A : s'$. Then by induction, $\|\Gamma, \Delta\| \vdash \|A\|_{\Gamma,\Delta} : s'$.

In any case: we can determine $s'_0 \in \boldsymbol{S}'$ such that

$$\|\Gamma\|, x_1{:}\, \|B_1\|_{\Gamma,\Delta_1}, \ldots, x_n{:}\, \|B_n\|_{\Gamma,\Delta_n} \vdash_{\mathfrak{S}'} \|A\|_{\Gamma,\Delta} : s'_0. \tag{7}$$

As $\mathfrak{S}'$ 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 (6) and (8), and as conclusion:

$$\|\Gamma\| \vdash_{\mathfrak{S}'} \textstyle\prod_{i=1}^{n} x_i{:}\, \|B_i\|_{\Gamma,\Delta_i} . \|A\|_{\Gamma,\Delta} : s'_n.$$

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 $\|\Gamma\| \vdash_{\mathfrak{S}'} \|b\|_{\Gamma} : \|B\|_{\Gamma}$, we can use the weakening rule of $\lambda\mathfrak{S}'$ to obtain

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

We are done because $\|b\|_{\Gamma} \equiv \|b\|_{\Gamma,c(\Delta)=a:A}$ and $\|B\|_{\Gamma} \equiv \|B\|_{\Gamma,c(\Delta)=a:A}$ (Lemma 6.55);

- δ-formation:

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

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

$$\|\Gamma_1\|, c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} \vdash_{\mathfrak{S}'} \|B\|_{\Gamma} : s. \tag{8}$$

By Correctness of Contexts on (8) there is $s_1 \in \boldsymbol{S}'$ such that

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \|\textstyle\prod \Delta.A\|_{\Gamma_1} : s_1. \tag{9}$$

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

$$\|\Gamma_1\|, c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} \vdash_{\mathfrak{S}'} s : s_2. \tag{10}$$

As $\mathfrak{S}'$ 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:

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \Pi c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} .s : s_3. \tag{11}$$

We can now apply λ-formation on (8) and (11):

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \left(\lambda c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) : \left(\Pi c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} .s\right). \tag{12}$$

As we have a derivation of $\Gamma_1, c(\Delta){=}a{:}A \vdash^{\vec{C}\vec{D}} B : s$, we can apply Correctness of Contexts to find a (shorter) derivation of $\Gamma_1, \Delta \vdash^{\vec{C}\vec{D}} a{:}A$, so by induction:

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

Using (9), we can repeatedly apply λ-abstraction and obtain

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \|\lambda\, \Delta.a\|_{\Gamma_1} : \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \tag{13}$$

Using (12) and the application rule, we find:

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \left(\lambda c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : s;$$

- δ-introduction:

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

In a similar way as in the previous case, we can find derivations of (13) and

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \left(\lambda c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|b\|_\Gamma\right) : \left(\Pi c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right). \tag{14}$$

Using (13), (14) and the application rule, we find

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \left(\lambda c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|b\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : \|B\|_\Gamma \left[c{:=}\, \|\lambda\, \Delta.a\|_{\Gamma_1}\right].$$

By the induction hypothesis,

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \left(\lambda c{:}\, \|\textstyle\prod \Delta.A\|_{\Gamma_1} . \|B\|_\Gamma\right) \|\lambda\, \Delta.a\|_{\Gamma_1} : s,$$

so we can apply the conversion rule to find

$$\|\Gamma_1\| \vdash_{\mathfrak{S}'} \|c(\Delta){=}a{:}A \text{ in } b\|_{\Gamma_1} : \|c(\Delta){=}a{:}A \text{ in } B\|_{\Gamma_1} ;$$

- δ-conversion:

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

  By induction, $\|\Gamma\| \vdash_{\mathfrak{S}'} \|b\|_\Gamma : \|B\|_\Gamma$ and $\|\Gamma\| \vdash_{\mathfrak{S}'} \|B'\|_\Gamma : s$. By Lemma 6.57, $\|B\|_\Gamma =_\beta \|B'\|_\Gamma$. By Conversion, $\|\Gamma\| \vdash_{\mathfrak{S}'} \|b\|_\Gamma : \|B'\|_\Gamma$.

⊠

Theorem 6.62 *Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ and $\mathfrak{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}')$ be such that $\mathfrak{S}'$ is a completion of $\mathfrak{S}$. If the PTS $\lambda(\mathfrak{S}')$ is β-strongly normalising, then the $\vec{C}\vec{D}$-PTS $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ is $\beta\delta$-strongly normalising.*

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

Observe that the number of β-reductions in this sequence is infinite. Otherwise there would be $n \in \mathbf{N}$ such that $\Gamma \vdash a_n \rightarrow_\delta a_{n+1} \rightarrow_\delta a_{n+2} \cdots$, which contradicts the fact that δ-reduction is strongly normalising (Theorem 6.49).

We conclude that the reduction sequence is of the form

$$\Gamma \vdash a \twoheadrightarrow_\delta a_{n_1} \rightarrow_\beta a_{n_2} \twoheadrightarrow_\delta a_{n_3} \rightarrow_\beta a_{n_4} \twoheadrightarrow_\delta \cdots$$

By lemmas 6.57 and 6.58 there is an infinite β-reduction sequence starting at $\|a\|_\Gamma$:

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

and by Theorem 6.61, $\|\Gamma\| \vdash_{\mathfrak{S}'} \|a\|_\Gamma : \|A\|_\Gamma$, which contradicts the assumption that $\lambda(\mathfrak{S}')$ is β-strongly normalising. ⊠

6d Restrictive use of parameters

In the extension of PTSs to $\vec{C}\vec{D}$-PTSs presented in Sections 6a-6c, we did not put any serious restrictions on the use of parameters:

1. If $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is a specification, then the introduction of a parametric constant c in $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ 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 the system λC, which is on top of the Barendregt Cube. 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 $\vec{C}\vec{D}$-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 is 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 6.63 Consider the Pascal function `double` that was presented in the Introduction to this Chapter.

- 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 6e.

6d1 $\vec{\text{C}}\vec{\text{D}}$-PTSs with restricted parameters

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

Definition 6.64 (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 A.17), 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 $\vec{\text{C}}$-weakening rule of Definition 6.20. We call this rule *restricted* $\vec{\text{C}}$-weakening ($\hat{\text{C}}$-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 holds for all $i \in \{1, \dots, n\}$. However, it is not necessary that all the s_i are equal: in one application of rule (Ĉ-weak) it is possible to rely on more than one element of $\boldsymbol{P}$.

Definition 6.65 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 A.20, ($\vec{\text{C}}$-app) (see Definition 6.20), and (Ĉ-weak).

Similarly, we present a restricted version of the δ-weakening rule of Definition 6.21. We call this rule *restricted* $\vec{\text{D}}$-weakening ($\hat{\text{D}}$-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 $i \in \{1, \dots, n\}$, 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 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 $i \in \{1, \dots, n\}$, $(s_i, \text{TOP}) \in \boldsymbol{P}$ must hold, but the s_i may, again, be different.

Definition 6.66 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 A.20, ($\vec{\text{D}}$-app), both versions of ($\hat{\text{D}}$-weak), ($\vec{\text{D}}$-form), ($\vec{\text{D}}$-intro), and ($\vec{\text{D}}$-conv) (see Definition 6.21).

Definition 6.67 The *typing relation* $\vdash^{\hat{C}\hat{D}}$ is obtained from the relation $\vdash^{\vec{C}\vec{D}}$ by replacing rule ($\vec{\text{C}}$-weak) by rule (Ĉ-weak) and rule ($\vec{\text{D}}$-weak) by rules ($\hat{\text{D}}$-weak).

Definition 6.68 (Pure Type Systems with Restricted Parameters and Restricted Parametric Definitions) Let $\mathfrak{S}$ be a parametric specification. The pure type system with restricted parameters and restricted parametric definitions (ĈD̂-PTS) and parametric specification $\mathfrak{S}$ is denoted $\lambda^{\hat{C}\hat{D}}(\mathfrak{S})$. 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 $\vec{C}\vec{D}$-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. We only give a stronger formulation of the extension of the Generation Lemma 6.51:

Lemma 6.69 (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* $\Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A and \Gamma_2 \rangle$, $\Gamma_1, \Delta \vdash^{\hat{C}\hat{D}} a : A : s$, *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* $\Gamma = \langle \Gamma_1, c(\Delta){=}a{:}A and \Gamma_2 \rangle$, $\Gamma_1, \Delta \vdash^{\hat{C}\hat{D}} a : A : \mathrm{TOP}$, *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 6.70 Our systems with restricted parameters cover the PTSs with Definitions (D-PTSs) that were introduced by Severi and Poll in [114]. Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ a specification, and observe the parametric specification $\mathfrak{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. 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 $\hat{D}$-weak for $n = 0$. In that case, we obtain only definitions without parameters, and the rules of the parametric system reduces precisely to the rules of a D-PTS with specification $\mathfrak{S}$.[2]

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

In the introduction to this Chapter, we argued that a parameter mechanism can be seen as a system for abstraction and application that is weaker

[2]The parametric system with specification $\mathfrak{S}'$ has a $\hat{C}$-weakening rule while the systems of Severi and Poll do not. But the $\hat{C}$-weakening rule can only be used for $n = 0$, and in that case $\hat{C}$-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.

than the λ-calculus mechanism. We will make this precise by proving (in Theorem 6.79) that a D-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ is as powerful as any $\hat{C}\hat{D}$-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 $\hat{C}\hat{D}$-PTS *parametrically conservative*:

Definition 6.71 Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. $\mathfrak{S}$ 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 $\hat{C}\hat{D}$-PTS can be extended to a parametrically conservative one by taking its *parametric closure*:

Definition 6.72 Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. We define $\mathrm{CL}(\mathfrak{S})$, the *parametric closure* of $\mathfrak{S}$, 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 6.73 *Let $\mathfrak{S}$ be a parametric specification.*

1. $\mathrm{CL}(\mathfrak{S})$ *is parametrically conservative;*
2. $\mathrm{CL}(\mathrm{CL}(\mathfrak{S})) = \mathrm{CL}(\mathfrak{S})$.

⊠

6d2 Imitating parameters by λ-abstractions

Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. If $\mathfrak{S}$ is parametrically conservative, then each parametric rule (s_1, s_2) of $\mathfrak{S}$ has an equivalent 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 $\mathfrak{S}$ has the same "power" (see Theorem 6.79) as $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. With Remark 6.70 in mind, this even means that $\mathfrak{S}$ has the same power as the D-PTS with specification $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

In order to compare $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ with $\mathfrak{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$, we need to remove the parameters from the syntax of $\lambda^{\hat{C}\hat{D}}(\mathfrak{S})$. This is easy:

- The parametric application in a term $c(b_1, \dots, b_n)$ 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 6.74 We define the parameter-free translation $\{t\}$ of a term $t \in \mathcal{T}_P$ as follows:

$$\begin{array}{rcl}
\{x\} & \equiv & x; \\
\{s\} & \equiv & s; \\
\{c(b_1,\dots,b_n)\} & \equiv & c\,\{b_1\}\cdots\{b_n\}\,; \\
\{ab\} & \equiv & \{a\}\,\{b\}\,; \\
\{\lambda x{:}A.b\} & \equiv & \lambda x{:}\,\{A\}\,.\,\{b\}\,; \\
\{\Pi x{:}A.B\} & \equiv & \Pi x{:}\,\{A\}\,.\,\{B\}\,; \\
\{c(\Delta){=}a{:}A \text{ IN } b\} & \equiv & c(){=}\,\{\lambda\,\Delta.a\}:\{\prod\Delta.A\} \text{ IN } \{b\}\,.
\end{array}$$

Definition 6.75 We extend the definition of $\{\_\}$ to contexts:

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

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 6.76 *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

$$c(b_1, \dots, b_n) \rightarrow_\delta a[x_i{:=}b_i]_{i=1}^n.$$

Lemma 6.77

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 6.76. (3) follows from (1) and (2). We only show (2), using induction on the definition of $\Gamma \vdash a \to_\delta a'$, and treating only the most important case.

Assume $\Gamma \equiv \Gamma_1, c(\Delta){=}a{:}A, \Gamma_2$ and $\Gamma \vdash c(b_1, \ldots, b_n) \to_\delta a[x_i{:=}b_i]^n_{i=1}$. Observe that $\{\Gamma\} \equiv \{\Gamma_1\}, c(){=}\{\lambda\,\Delta.a\} : \{\prod \Delta.A\}, \{\Gamma_2\}$, so

$$\begin{array}{rcl}\{\Gamma\} \vdash c()\,\{b_1\} \cdots \{b_n\} & \to_\delta & \{\lambda\,\Delta.a\}\,\{b_1\} \cdots \{b_n\} \\ & \twoheadrightarrow_\beta & \{a\}\,[x_i{:=}\,\{b_i\}]^n_{i=1} \\ & \overset{(6.76)}{\equiv} & \{a[x_i{:=}b_i]^n_{i=1}\}\,. \end{array}$$

⊠

Remark 6.78 In 6.77.1, we cannot replace $\twoheadrightarrow^+_\beta$ by $\to_\beta$. This has to do with the definition of $\{c(\Delta){=}a{:}A \text{ IN } b\}$. One β-reduction in Δ gives rise to (at least) two β-reductions in $c(){=}\{\lambda\,\Delta.a\} : \{\prod \Delta.A\}$ IN $\{b\}$.

Similarly, we cannot replace the $\twoheadrightarrow^+_\delta$ in 6.77.2 by $\to_\delta$.

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

Theorem 6.79 *Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. Assume $\mathfrak{S}$ is parametrically conservative. Let $\mathfrak{S}' = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$. Then*

$$\Gamma \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}} a : A \Longrightarrow \{\Gamma\} \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} \{a\} : \{A\}\,.$$

PROOF: Induction on the derivation of $\Gamma \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}} a : A$. With the help of Lemma 6.76 and Lemma 6.77.3, 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

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

By the induction hypothesis, we have:

$$\begin{array}{rcll} \{\Gamma\} & \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} & \{b\} : \{B\}; & (15) \\ \{\Gamma, \Delta_i\} & \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} & \{B_i\} : s_i; & (16) \\ \{\Gamma, \Delta\} & \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} & \{a\} : \{A\}; & (17) \\ \{\Gamma, \Delta\} & \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} & \{A\} : s. & (18) \end{array}$$

$\mathfrak{S}$ 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 (18) and (16), obtaining

$$\{\Gamma\} \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} \textstyle\prod_{i=1}^{n} x_i{:}\, \{B_i\} \,.\, \{A\} : s. \tag{19}$$

Notice: $\prod_{i=1}^{n} x_i{:}\, \{B_i\} \,.\, \{A\} \equiv \{\prod \Delta.A\}$. Repeatedly using λ-formation, using (17) and (19), results in

$$\{\Gamma\} \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} \mathop{\lambda}_{i=1}^{n} x_i{:}\, \{B_i\} \,.\, \{a\} : \{\textstyle\prod \Delta.A\}\,. \tag{20}$$

Similarly, $\lambda_{i=1}^{n} x_i{:}\, \{B_i\} \,.\, \{a\} \equiv \{\lambda\, \Delta.a\}$. Using (D̂-weak) (for the specification $\mathfrak{S}'$) on (15), (16), (19) and (20) results in

$$\{\Gamma\}\,, c(){=}\,\{\lambda\, \Delta.a\} : \{\textstyle\prod \Delta.A\} \vdash^{\hat{C}\hat{D}}_{\mathfrak{S}'} \{b\} : \{B\}\,.$$

⊠

Remark 6.80 The results in Section 6d2 were 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 6.70, 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 $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

6d3 Refined Barendregt Cubes

Theorem 6.79 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 $\hat{\text{C}}\hat{\text{D}}$-PTS with parametric specification $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \varnothing)$ 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 6d2 to a concrete situation: the Barendregt Cube. The Barendregt Cube (Figure 13 on page 300) 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 13 have the following Π-formation rules:

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

This cube can be constructed not only for PTSs, but also for C-PTSs, D-PTSs, $\vec{\text{C}}$-PTSs, $\vec{\text{D}}$-PTSs, and their combinations (see Figure 10 on page 235).

With Theorem 6.79, we can place certain $\hat{\text{C}}\hat{\text{D}}$-PTSs in the cube of D-PTSs (and, with Remark 6.80 in mind, certain $\hat{\text{C}}$-PTSs can be placed in the cube of C-PTSs). Let us, for example, have a look at the following parametric specifications:

$$\begin{array}{l}(\boldsymbol{S},\boldsymbol{A},\{(*,*,*),(*,\Box,\Box)\},\varnothing);\\(\boldsymbol{S},\boldsymbol{A},\{(*,*,*),(*,\Box,\Box)\},\{(*,*)\});\\(\boldsymbol{S},\boldsymbol{A},\{(*,*,*),(*,\Box,\Box)\},\{(*,\Box)\});\\(\boldsymbol{S},\boldsymbol{A},\{(*,*,*),(*,\Box,\Box)\},\{(*,*),(*,\Box)\}).\end{array}$$

where $\boldsymbol{S}=\{*,\Box\}$ and $\boldsymbol{A}=\{(*,\Box)\}$. According to Theorem 6.79, the ĈD̂-PTSs with the above specifications are all equal in power, and according to Remark 6.70, they are all equal in power to the D-PTS with the specification of λP.

Now look at the parametric specification

$$\mathfrak{S}=(\boldsymbol{S},\boldsymbol{A},\{(*,*,*)\},\{(*,*),(*,\Box)\}).$$

The Ĉ-PTS $\lambda^{\hat{C}}(\mathfrak{S})$ is clearly stronger than the PTS $\lambda{\to}$, as in $\lambda^{\hat{C}}(\mathfrak{S})$ it is possible (in a restricted way) to talk about predicates. For instance, we can have the following context:

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

This context introduces an equality predicate `eq` on objects of type α, and axioms `refl`, `symm`, `trans` for the reflexivity, symmetry and transitivity of `eq`. It is not possible to introduce such a predicate `eq` in the PTS $\lambda{\to}$ without any parameter mechanism. On the other hand, $\lambda^{\hat{C}}(\mathfrak{S})$ is weaker than the PTS λP: in λP we can construct the type $\Pi x{:}\alpha.\Pi y{:}\alpha.*$, which allows us to introduce variables `eq` of type $\Pi x{:}\alpha.\Pi y{:}\alpha.*$. 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 Chapter).

Altogether, this puts the Ĉ-PTS $\lambda^{\hat{C}}(\mathfrak{S})$ clearly in between the PTSs $\lambda{\to}$ and λP. Similarly, the ĈD̂-PTS $\lambda^{\hat{C}\hat{D}}(\mathfrak{S})$ is in between the D-PTSs

$\lambda{\to}$ and λP. We can illustrate this in the Barendregt Cube by putting the specification $\mathfrak{S}$ 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 6.79, 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 11.

6e 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 12 on page 283, and motivated in the four subsections below.

6e1 ML

In ML (see for instance [90]) one can define the polymorphic identity as follows (we use the notation of this Chapter. In ML, the types and the parameters are left implicit):

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

But it is not possible to make an explicit λ-abstraction over $\alpha{:}*$: the expression

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

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 $(\Box, *)$.

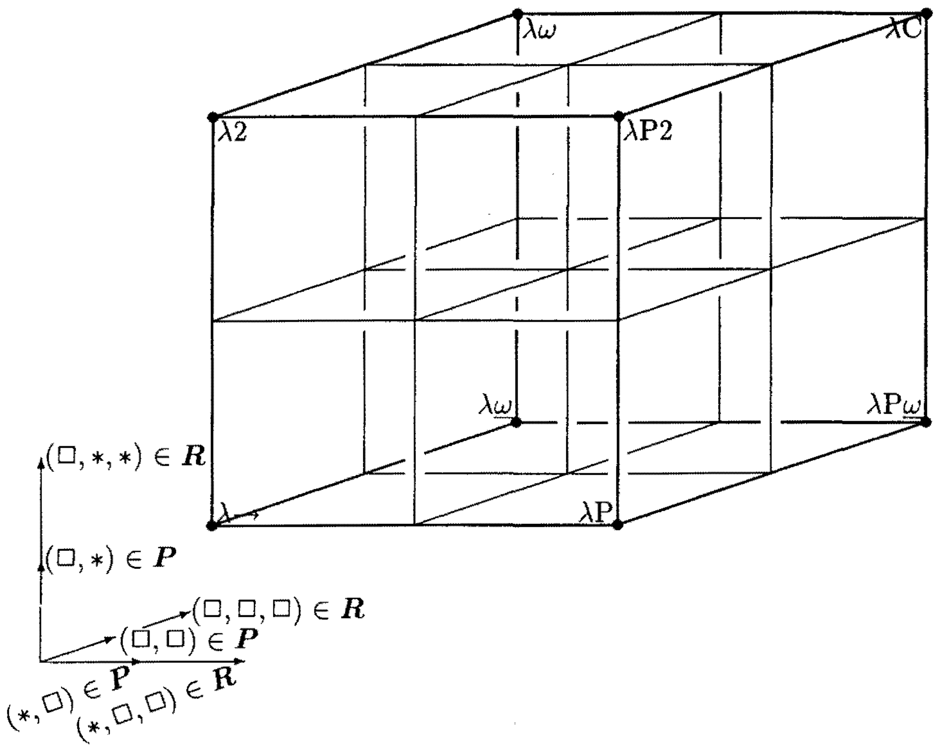


Figure 11: The refined Barendregt Cube

Similarly, one can introduce the type of lists together with some elementary operations in ML as follows:

$$\begin{aligned}\texttt{List}(\alpha{:}*) &: *;\\ \texttt{nil}(\alpha{:}*) &: \texttt{List}(\alpha);\\ \texttt{cons}(\alpha{:}*) &: \alpha \to \texttt{List}(\alpha) \to \texttt{List}(\alpha),\end{aligned}$$

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

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

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 $\lambda\omega$.

6e2 LF

Geuvers [54] initially describes the system LF (see [59]) 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:

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

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

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

$$\begin{aligned}(\lambda_0 x{:}A.M)N &\to_{\beta_0} M[x{:=}N];\\ (\lambda_P x{:}A.M)N &\to_{\beta_P} M[x{:=}N].\end{aligned}$$

Geuvers then shows that

- If $M : *$ or $M : A : *$ in LF, then the β_P-normal form of M contains no λ_P;
- If $\Gamma \vdash_{\mathrm{LF}} M : A$, and Γ, M, A do not contain a λ_P, then $\Gamma \vdash_{\mathrm{LF}^-} M : A$;

- If $\Gamma \vdash M : A(: *)$, all in β_P-normal form, then $\Gamma \vdash_{\mathrm{LF}^-} M : A(: *)$.

This means that the only real need for a type $\Pi x{:}A.B : \Box$ is to be able to declare a variable in it. The only point at which this is really done is where the bool-style implementation of PAT is made (see Section 4a4): the construction of the type of the operator Prf (in an unparameterised form) has to be made as follows:

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

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

6e3 λ68 and AUT-68

Looking back at the system AUT-68 of Section 5a and its λ-calculus variant λ68 that was constructed and discussed in Sections 5b-5c, we remark that AUT-68 has a parameter mechanism and a mechanism for global parametric definitions:

- A line $(\Gamma; k; \textsc{pn}; \texttt{type})$ in a book is nothing more that 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 $k(\Gamma){=}\Sigma_1{:}*$. 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 $A{:}\texttt{type}$ (in PTS-terminology: $A : *$). We conclude that AUT-68 has the parameter rules $(*, \Box)$ and $(\Box, \Box)$;
- Similarly, lines of the form $(\Gamma; k; \textsc{pn}; \Sigma_2)$ and $(\Gamma; k; \Sigma_1; \Sigma_2)$, where $\Sigma_2{:}\texttt{type}$, 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 $\hat{C}\hat{D}$-PTS with specification

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

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 [5], where it cannot be clearly positioned in the Barendregt Cube. Another advantage is that $\lambda^{\hat{C}\hat{D}}(\mathfrak{S}_{68})$ has parameters. Thus, it is closer to the original system AUT-68 than the system $\lambda 68$ that was described in Chapter 5, and in [5] (though in Theorem 5.62 and Remark 5.63, we minutely described the way in which the parameter mechanism appears in $\lambda 68$).

On the other hand, we should not say that AUT-68 is exactly the system $\lambda^{\hat{C}\hat{D}}(\mathfrak{S}_{68})$. There are several differences:

- DPTSs have 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. In AUTOMATH, B has to be typable;
- The D-reduction of DPTSs is not substitutive; δ-reduction of AUTOMATH is substitutive;

These differences can also be found between AUT-68 and the DPTSs of Severi and Poll (see Section 5d2).

6e4 λQE and AUT-QE

In λQE we have a Π-formation rule $(*, \Box, \Box)$ additionally to the rules of $\lambda 68$. This means that the applicational and abstractional behaviour can be described by the $\hat{C}\hat{D}$-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. Again, this is not the exact representation of AUT-QE; there are differences that are similar to those described in Section 5d2. Moreover, AUT-QE has a rule of type inclusion (see the Conclusion of Chapter 5), which is not taken into account in $\hat{C}\hat{D}$-PTSs.

6e5 PAL

The AUTOMATH languages are all based on two concepts: typed λ-calculus and a parameter/definition mechanism. Both concepts can be isolated: it is possible to study λ-calculus without a parameter/definition mechanism (for instance via the format of Pure Type Systems), 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 $\hat{C}\hat{D}$-PTS with the following parametric specification:

$$\begin{array}{rcl} \boldsymbol{S} & = & \{*, \Box\} \\ \boldsymbol{A} & = & \{(*, \Box)\} \\ \boldsymbol{R} & = & \varnothing \\ \boldsymbol{P} & = & \{(*, *), (*, \Box), (\Box, *), (\Box, \Box)\} \end{array}$$

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

6f 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 [5].

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 $*_p, \Box_p$ 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

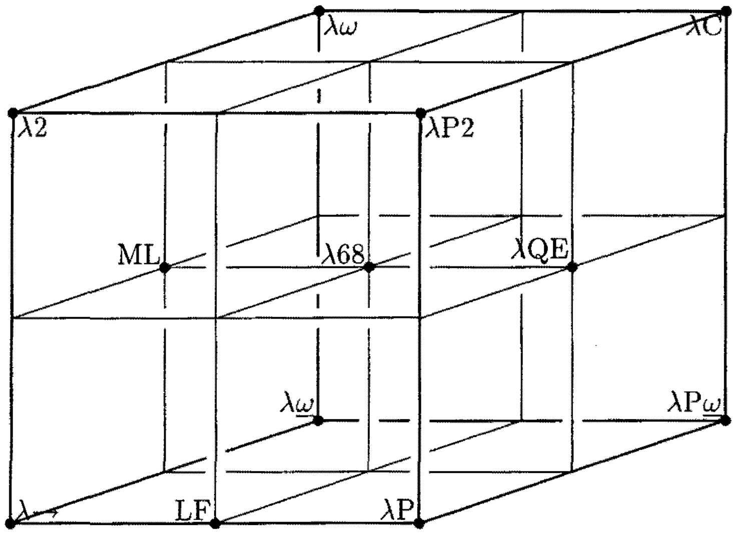


Figure 12: LF, ML, λ68, and λQE in the refined Barendregt Cube

$(*_s, *_f, *_f)$ makes it possible to form functions of several arguments between object types. There is no sort $\square_f$, as free variables for function spaces are not allowed. The construction of relation symbols requires Π-formation rule $(*_s, \square_p, \square_p)$.

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

$$\begin{aligned}
\boldsymbol{S} &= \{*_s, *_p, *_f, \square_s, \square_p\};\\
\boldsymbol{A} &= \{(*_s, \square_s), (*_p, \square_p)\};\\
\boldsymbol{R} &= \{(*_s, *_s, *_f), (*_s, *_f, *_f), (*_s, *_p, *_p), (*_p, *_p, *_p), (*_s, \square_p, \square_p)\}.
\end{aligned}$$

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

Example 6.81 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 $P_1 P_2$. 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_is are of type $*_s$), we apply $\vec{\mathrm{C}}$-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 6.84). It is reasonable to replace 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 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 $\mathfrak{S}' = (\boldsymbol{S}', \boldsymbol{A}', \boldsymbol{R}', \boldsymbol{P}')$, where:

$$\begin{array}{rcl} \boldsymbol{S}' & = & \{*_s, *_p, \Box_s, \Box_p\}; \\ \boldsymbol{A}' & = & \{(*_s, \Box_s), (*_p, \Box_p)\}; \\ \boldsymbol{R}' & = & \{(*_s, *_p, *_p), (*_p, *_p, *_p)\}; \\ \boldsymbol{P}' & = & \{(*_s, *_s), (*_s, \Box_p)\}. \end{array}$$

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 6.82 *If* $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} P : Q : *_s$ *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, 6.69, and determine $B_1, \ldots, B_n, B$ and $s_1, \ldots, s_n, s$ such that $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1}$ and $\Gamma, x_1{:}B_1, \ldots x_{i-1}{:}B_{i-1} \vdash^{\hat{C}}_{\mathfrak{S}'} B_i{:}s_i$, 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 $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} P_1 : \Pi x{:}R_1.R_2$, and $Q =_\beta R_2[x{:=}P_2]$. By Correctness of Types there is $s \in \boldsymbol{S}'$ such that $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} (\Pi x{:}R_1.R_2) : s$. By the Generation Lemma and the definition of $\boldsymbol{R}'$, $\Gamma, x{:}R_1 \vdash^{\hat{C}}_{\mathfrak{S}'} R_2{:}*_p$. By the Substitution Lemma, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} R_2[x{:=}P_2]{:}*_p$. Let Q' be a common β-reduct of Q and $R_2[x{:=}P_2]$. By Subject Reduction, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} Q' : *_s$ and $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} Q' : *_p$, 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, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} \Pi x{:}R_1'.R_2' : *_s$. 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 6.83 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 $c(b_1, \dots, b_n)$ such that there are $B_1, \dots, B_n$ with $b_i : B_i : *_s$. This corresponds exactly to the definition of terms in first-order logic. We conclude that our specification $\mathfrak{S}'$ results in an exact description of the terms of first-order logic.

Theorem 6.84 *Assume* $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} P : s$. *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;
- $P \equiv c(b_1, \dots, b_n)$. By the second extension of the Generation Lemma 6.69, there are sorts $s_1, \dots, s_n$ and terms $B_1, \dots, B_n$ such that $(s_i, s) \in \boldsymbol{P}'$, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$ and $\Gamma, x_1{:}B_1, \dots, x_{i-1}{:}B_{i-1} \vdash^{\hat{C}}_{\mathfrak{S}'} B_i{:}s_i$. By the definition of $\boldsymbol{P}'$, $s_i \equiv *_s$ for all i. By the Substitution Lemma, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s$. By Lemma 6.82, the b_i are in β-normal form. Therefore $c(b_1, \dots, b_n)$ is in β-normal form;

- $P \equiv P_1P_2$. By the Generation Lemma, there are R_1, R_2 such that $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} P_1 : \Pi x{:}R_1.R_2$ and $s =_\beta R_2[x{:=}P_2]$. By Correctness of Types, the Generation Lemma and the definition of $\boldsymbol{R}'$, $\Gamma, x{:}R_1 \vdash^{\hat{C}}_{\mathfrak{S}'} R_2 : *_p$. By the Substitution Lemma, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} R_2[x{:=}P_2] : *_p$. By Subject Reduction, $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} s : *_p$. 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 $\Gamma \vdash^{\hat{C}}_{\mathfrak{S}'} P_1 : s_1$ and $\Gamma, x{:}P_1 \vdash^{\hat{C}}_{\mathfrak{S}'} P_2 : s_2$. 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

$$(*_s, *_s, *_f) \qquad (*_s, *_f, *_f) \qquad (*_s, \Box_p, \Box_p)$$

by parametric rules

$$(*_s, *_s) \qquad (*_s, \Box_p)$$

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).

Conclusions: Yet another extension of PTSs?

Since PTSs have been introduced, many extensions have been proposed (see [6] for a non-exhaustive list). The reader may wonder why yet another extension of PTSs is proposed in this Chapter, and whether it is more interesting than those other extensions or not. In this section we give an answer to these questions.

Practical motivation

We gave already some reasons for the use of parameters at the beginning of this Chapter:

- 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 [114];

- Parameters and parametric definitions occur in many implementations of type systems, and more general, in programming languages. The Pascal-function `double` that was introduced at the beginning of this Chapter 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. The description given in Chapter 5 is precise, but it is not a description that looks natural. The separate abstractors ¶ and § do their job as well as possible in a type system without parameters, but a description of AUTOMATH that includes parameters does more justice to that system. Moreover, it places AUTOMATH in a more general framework, so that it can easily be compared with other type systems (see Figure 12 on page 283);

- Modern type systems, like LF and ML, have already been described as one of the systems of the Barendregt Cube (Figure 13 on page 300). But in Section 6e we showed that a more detailed description can be given in the refined Barendregt Cube of Figure 12;

- As argued in Section 6f, 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 original first-order language and therefore closer to the intuition;

- As argued in the beginning of this Chapter, 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.

But "parameters give a better description of the type theory that is used in practice" is not the only argument in favour of the system of this Chapter. There is more than that.

The heart of type theory

In the Introduction we extensively discussed the notions of functionalisation and instantiation and declared them to be the heart of type theory. After our exploration of type theory throughout the present work, we still think they are, for more than one reason:

- Functionalisation and instantiation stood at the cradle of type theory. The story of type theory began with Frege's abstraction principles (instantiation was not explicitly defined, but definitely present in an implicit form), and the logical paradoxes that arose if one does not use these principles carefully. Type theory made a careful use of Frege's principles possible;

- An important application of modern type theory is logic. This is due to the PAT-principle, which on its turn is based on the interpretation of $\rightarrow$ and $\forall$ as function types. And function types exist because of functionalisation and instantiation.

The parameter mechanism shows us a new, different form of functionalisation and instantiation and therefore makes the theory of functions richer and more interesting:

- It gives us a better idea of the possibilities of the traditional forms of functionalisation and instantiation;

- It places these traditional forms in a broader perspective by showing that these forms are not the only possible forms of functionalisation and instantiation.

In this light, the parameter mechanism is not only an *extension* of Pure Type Systems (as depicted in Figure 10 on page 235), but also (and particularly) a *refinement* of this framework, resulting in refinements of parts of it, like the Barendregt Cube (Figure 11 on page 278).

Future work

There are several things 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 Chapter. 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;

- In the systems proposed in this chapter, it is not possible to have a parametric constant (or definition) that takes a parametric function as a parameter. For example: We want to formulate the property $\mathtt{Ref}(B)$ for binary relations B over type T, indicating that this relation is reflexive. In our current system, B cannot be a parametric function $b(x{:}T, y{:}T)$, because $b(x{:}T, y{:}T)$ is not a term. We must make the full λ-abstraction $\lambda x{:}T.\lambda y{:}T.b(x, y)$ (which is a term) if we want to give it as an argument to `Ref`.

  It may be useful to design a system in which the parametric function $b(x{:}T, y{:}T)$ could be substituted for B without the need of making the λ-abstractions.

- There may be a relation between the parameter mechanism of this chapter 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.

Appendix A

Preliminaries

In this thesis we try to present various important type systems that were proposed during this century in a uniform framework. An important part of this framework is formed by the so-called Pure Type Systems (PTSs). Therefore, a short introduction to typed lambda calculus and PTSs is essential for the understanding of this thesis.

Lambda calculus was introduced by Church [28, 29], as a formalisation of the notion of function. With this formal notation he could formulate his set of postulates for the foundation of logic. Kleene and Rosser [74] showed that Church's set of postulates was inconsistent. The lambda calculus itself, however, appeared to be a very useful tool. In Chapter 2 of this thesis we showed that it is much more clear and accurate than the notion of (propositional) function as introduced by Russell and Whitehead in Principia Mathematica [121].

Nowadays, [4] is the standard work for (untyped) lambda calculus. We present the basic definitions and properties of the λ-calculus in Section Aa.

Being a suitable framework for the formalisation of functions, it is not surprising that lambda calculus appeared to be an excellent tool for formalising the Simple Theory of Types [30]. In Section Ab we give a short description of Church's formalisation. This formalisation is at the basis of most modern type theories and especially at the basis of PTSs. PTSs were introduced by Terlouw [118] and Berardi [13], providing a general framework in which many type systems can be described. Section Ac presents the definition of PTSs and Section Ad discusses the most important meta-properties as described in [55], [5], and [54].

Aa Lambda calculus

We give a description of typed lambda terms. This description follows the line of [5], as it mainly serves as a description of Pure Type Systems (PTSs).

Definition A.1 Let $\mathbb{V}$ be a set of variables and $\mathbb{C}$ a set of constants. The set $\mathcal{T}(\mathbb{V}, \mathbb{C})$ (shorthand: $\mathcal{T}$, if it is clear which sets $\mathbb{V}$ and $\mathbb{C}$ are used) of typed lambda terms with variables from $\mathbb{V}$ and constants from $\mathbb{C}$ is defined by the following abstract syntax:

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

If x does not occur in B then $\Pi x{:}A.B$ is sometimes denoted by $A \to B$.

We assume that $\mathbb{V}$ and $\mathbb{C}$ are countably infinite. We use $\equiv$ to denote syntactical equality between typed lambda terms.

We use x, y, z, α, β as meta-variables over $\mathbb{V}$. In examples, we sometimes want to use some specific elements of $\mathbb{V}$; we use typewriter-style to denote such specific elements. So: x is a specific element of $\mathbb{V}$; while x is a meta-variable *over* $\mathbb{V}$. The variables x, y, z are assumed to be *distinct* elements of $\mathbb{V}$ (so x $\not\equiv$ y etc.), while meta-variables $x, y, z, \ldots$ may refer to variables in the object language that are syntactically equal. We use $A, B, C, \ldots, a, b, \ldots$ as meta-variables over $\mathcal{T}$.

A term $\lambda x{:}A.b$ has as intuitive interpretation the function that assigns $b[x{:=}a]$ (the term b in which each occurrence of x has been replaced by a) to each a that belongs to (is an element of, has type) A. If b has type B, then $\lambda x{:}A.b$ is a function from A to B. $A \to B$ should be interpreted as the type of functions from A to B. This means: $\lambda x{:}A.b$ has type $A \to B$.

Example A.2 λx:A.x is the identity function on A, and has type $A \to A$.

In some situations, we allow that the type B of b *depends on* the variable x. In that case, $b[x{:=}a]$ is of type $B[x{:=}a]$ for a of type A. Then $\lambda x{:}A.b$ is a function with domain A and range $\bigcup_{a:A} B[x{:=}a]$, with the special property that the function value for $a{:}A$, $b[x{:=}a]$, belongs to the subset $B[x{:=}a]$ of $\bigcup_{a:A} B[x{:=}a]$. The type of such functions will be represented by $\Pi x{:}A.B$.

Example A.3 The polymorphic identity λy:$*$.λx:y.x has as type

$$\Pi y{:}*.y \to y.$$

We could have written Πy:$*$.Πx:y.y instead of Πy:$*$.y $\to$ y. Here $*$ can be interpreted as the class of types.

Remark A.4 The term λy:**type**.λx:y.x in Example A.3 is a function of two variables: y and x. The function is constructed by repeated λ-abstraction. The first λ-abstraction (over x) leads to a function of one variable: λx:y.x, and another λ-abstraction (over y) leads to the desired function of two variables. The use of repeated λ-abstraction in order to represent functions of more than one variable is called "currying" after H. B. Curry, though currying was already discovered by Schönfinkel in 1924 [109], before Curry discovered it, and the basic ideas for currying can already be found in the works of Frege, which date from 1879 (see Section 1b1 of this thesis).

The following notational conventions allow us to reduce the number of brackets in terms:

Notation A.5

- We write $\lambda\vec{x}{:}\vec{A}.B$, or $\lambda_{i=1}^{n} x_i{:}A_i.A$, as shorthand for

$$\lambda x_1{:}A_1.(\lambda x_2{:}A_2.(\cdots(\lambda x_n{:}A_n.A)\cdots));$$

- We use $\Pi\vec{x}{:}\vec{A}.B$, or $\Pi_{i=1}^{n} x_i{:}A_i.A$, as shorthand for

$$\Pi x_1{:}A_1.(\Pi x_2{:}A_2.(\cdots(\Pi x_n{:}A_n.A)\cdots));$$

- We write $AB_1 \cdots B_n$ as shorthand for

$$(\cdots((AB_1)B_2)\cdots B_n).$$

Definition A.6 For $A \in \mathcal{T}$ we define $\text{FV}(A)$, the set of *free variables* of A, as follows:

- $\text{FV}(c) = \varnothing$ for $c \in \mathbb{C}$;

- $\mathrm{FV}(x) = \{x\}$ for $x \in \mathbb{V}$;
- $\mathrm{FV}(A_1A_2) = \mathrm{FV}(A_1) \cup \mathrm{FV}(A_2)$;
- $\mathrm{FV}(\lambda x{:}A_1.A_2) = (\mathrm{FV}(A_1) \setminus \{x\}) \cup \mathrm{FV}(A_2)$;
- $\mathrm{FV}(\Pi x{:}A_1.A_2) = (\mathrm{FV}(A_1) \setminus \{x\}) \cup \mathrm{FV}(A_2)$.

Definition A.7 If $\mathrm{FV}(a) = \{x_1, \ldots, x_n\}$ and $A_1, \ldots, A_n$ is a list of terms then $\lambda_{i=1}^{n} x_i{:}A_i.a$ is a *closure* of a.

Notice that there may be many different closures of one and the same term a.

A subset of the set of λ-terms that will be used in this thesis is the set of the so-called λI-terms:

Definition A.8 Let $\mathbb{V}$ be a set of variables and $\mathbb{C}$ a set of constants. The set of λI-terms $\mathcal{T}_I$ over $\mathbb{V}$ and $\mathbb{C}$ is defined as follows:

- If $v \in \mathbb{V}$ then $v \in \mathcal{T}_I$; if $c \in \mathbb{C}$ then $c \in \mathcal{T}_I$;
- If $A, B \in \mathcal{T}_I$ then $AB \in \mathcal{T}_I$;
- If $A, b \in \mathcal{T}_I$ and $x \in \mathrm{FV}(b)$ then $\lambda x{:}A.b \in \mathcal{T}_I$;
- If $A, B \in \mathcal{T}_I$ then $\Pi x{:}A.B \in \mathcal{T}_I$.

So within the set of λI-terms, a λ-abstraction $\lambda x{:}A.b$ can only be made if the term b really *depends* on the variable x. This means that *constant* functions, and functions of more variables that are constant in one or more of their variables, are excluded from the set of λI-terms.

Terms that are equal up to a change of bound variables are considered to be syntactically equal. This allows us to assume the so-called *Barendregt Convention*:

Convention A.9 (Barendregt Convention) Bound variables will be chosen to be different from free variables. For instance, we write $(\lambda \mathrm{y}{:}A.\mathrm{y})\mathrm{x}$ instead of $(\lambda \mathrm{x}{:}A.\mathrm{x})\mathrm{x}$.

Once this variable convention has been assumed we can define substitution in a straightforward manner (whereas the definition in [38] is more complicated, and a formal definition of substitution is completely absent in [28, 29] and [31]):

Definition A.10 (Substitution) We define $A[x{:=}B]$ by induction on the structure of A:

- $y[x{:=}B] \equiv \begin{cases} B & \text{if } y \equiv x; \\ y & \text{if } y \not\equiv x; \end{cases}$
- $(A_1 A_2)[x{:=}B] \equiv A_1[x{:=}B]A_2[x{:=}B]$;
- $(\lambda y{:}A_1.A_2)[x{:=}B] \equiv \lambda y{:}A_1[x{:=}B].A_2[x{:=}B]$;
- $(\Pi y{:}A_1.A_2)[x{:=}B] \equiv \Pi y{:}A_1[x{:=}B].A_2[x{:=}B]$.

We use the abbreviation $A[x_i{:=}B_i]_{i=m}^{n}$ to denote

$$A[x_m{:=}B_m]\cdots[x_n{:=}B_n].$$

If $m > n$ then $A[x_i{:=}B_i]_{i=m}^{n}$ denotes A. We also use the notation $A[\vec{x}{:=}\vec{B}]$ for $A[x_i{:=}B_i]_{i=1}^{n}$.

On lambda terms we have the notion of *β-reduction*.

Definition A.11 (β-reduction) The relation $\to_\beta$ is described by the contraction rule

$$(\lambda x{:}A_1.A_2)B \to_\beta A_2[x{:=}B]$$

and the usual compatibility rules (so: if $A \to_\beta A'$ then $AB \to_\beta A'B$, $BA \to_\beta BA'$, $\lambda x{:}A.B \to_\beta \lambda x{:}A'.B$, $\lambda x{:}B.A \to_\beta \lambda x{:}B.A'$, $\Pi x{:}A.B \to_\beta \Pi x{:}A'.B$ and $\Pi x{:}B.A \to_\beta \Pi x{:}B.A'$).

$\twoheadrightarrow_\beta$ is the smallest reflexive and transitive relation that includes $\to_\beta$; $=_\beta$ is the smallest reflexive, symmetric and transitive relation that includes $\to_\beta$. By $A \twoheadrightarrow_\beta^+ B$ we indicate that $A \twoheadrightarrow_\beta B$, but $A \not\equiv B$.

A term that has no subterm of the form $(\lambda x{:}A_1.A_2)B$ is a term in *β-normal form*, or a *normal form* if no confusion arises. We write $A \to_\beta^{\mathrm{nf}} B$ if $A \to_\beta B$ and B is in β-normal form. Similarly, $A \twoheadrightarrow_\beta^{\mathrm{nf}} B$ if $A \twoheadrightarrow_\beta B$ and B is in β-normal form.

The most important property of $\rightarrow_\beta$ is the so-called *Church-Rosser property*:

Theorem A.12 *If $A \twoheadrightarrow_\beta B_1$ and $A \twoheadrightarrow_\beta B_2$ then there is C such that $B_1 \twoheadrightarrow_\beta C$ and $B_2 \twoheadrightarrow_\beta C$.*

There are numerous proofs of this theorem in the literature. The most well-known is via the Strip Lemma (see [4], Chapter 11), another short and elegant proof is given by Tait and Martin-Löf [88], also described in Chapter 3 of [4].

In this thesis we see many variants on the basic lambda terms of Definition A.1, for instance lambda terms with parameters in Chapter 6. It is easy to prove that these variants have the Church-Rosser property for $\rightarrow_\beta$ as well.

Ab Simply typed λ-calculus

We give a definition of the simply typed λ-calculus as introduced by Church [30] in 1940.

Definition A.13 The types of $\lambda{\rightarrow}$ are defined as follows:

- ι and o are types;
- If α and β are types, then so is $\alpha \rightarrow \beta$.

We denote the set of simple types by $\mathbb{T}$.

ι represents the type of individuals; o is the type of propositions. $\alpha \rightarrow \beta$ is the type of functions with domain α and range β. We use $\alpha, \beta, \ldots$ as meta-variables over types. $\rightarrow$ associates to the right: $\alpha \rightarrow \beta \rightarrow \gamma$ denotes $\alpha \rightarrow (\beta \rightarrow \gamma)$.

The terms of the original presentation of $\lambda{\rightarrow}$ are a bit different from the presentation in [5]. We give some explanation after repeating the original definition.

Definition A.14 The terms of $\lambda{\rightarrow}$ are the following:

- $\neg$, $\wedge$, $\forall_\alpha$ for each type α, and $\imath_\alpha$ for each type α, are terms;

- A variable is a term;
- If A, B are terms, then so is AB;
- If A is a term, and x a variable, then $\lambda x{:}\alpha.A$ is a term.

Definition A.15 A *context* in $\lambda\!\!\rightarrow$ is a set $\{x_1{:}\alpha_1, \dots x_n{:}\alpha_n\}$ where the x_i are distinct variables and the α_i are types.

Some terms are typable (legal) in $\lambda\!\!\rightarrow$, according to the following derivation rules:

Definition A.16 The judgement $\Gamma \vdash A : \alpha$ holds if it can be derived using the following rules:

- $\Gamma \vdash \neg : o \rightarrow o$;
  $\Gamma \vdash \wedge : o \rightarrow o \rightarrow o$;
  $\Gamma \vdash \forall_\alpha : (\alpha \rightarrow o) \rightarrow o$;
  $\Gamma \vdash \imath_\alpha : (\alpha \rightarrow o) \rightarrow \alpha$;
- $\Gamma \vdash x : \alpha$ if $x{:}\alpha \in \Gamma$;
- If $\Gamma, x{:}\alpha \vdash A : \beta$ then $\Gamma \vdash (\lambda x{:}\alpha.A) : \alpha \rightarrow \beta$;
- If $\Gamma \vdash A : \alpha \rightarrow \beta$ and $\Gamma \vdash B : \alpha$ then $\Gamma \vdash (AB) : \beta$.

We use $\vdash_{\lambda\rightarrow}$ if we need to distinguish derivability in $\lambda\!\!\rightarrow$ from derivability in other type systems.

The simply typed λ-calculus can be seen as a pure type system, and therefore has the properties of pure type systems, that can be found at the end of the following Section. To adapt the simply typed λ-calculus to a pure type system, some amendments are made:

- The two basic types ι, o are replaced by an infinite set of *type variables*;
- The constants $\neg$, $\wedge$, $\forall_\alpha$ and $\imath_\alpha$ are not introduced in the PTS-presentation.

These adaptions do not seriously affect the system and are only used to make $\lambda\!\!\rightarrow$ fit in the PTS-framework.

Ac Pure type systems

Pure Type Systems (PTSs) were introduced by Berardi [13] and Terlouw [118] 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 ([91], Chapter III, Definition 1.3) and many rules are highly influenced by rules of known type systems like Church's Simple Theory of Types [30] and Automath (see 5.5.4. of [39], and Section 5a).

The description below is based on [5].

Definition A.17 (Specification) A *specification* is a triple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$, such that $\boldsymbol{S} \subseteq C$, $\boldsymbol{A} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ 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 A.18 (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* $\mathrm{DOM}\left(\vec{x}{:}\vec{A}\right)$ of the context. The *empty context* is denoted $\langle\rangle$.

We use Γ, Δ as meta-variables for contexts.

Substitution can be extended to contexts:

Definition A.19 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 A.20 (Pure Type Systems) Let $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$ be a specification. The Pure Type System $\lambda\mathfrak{S}$ describes in which ways judgements

$\Gamma \vdash_{\mathfrak{S}} A : B$ (or $\Gamma \vdash A : B$, if it is clear which $\mathfrak{S}$ is used) can be derived. $\Gamma \vdash A : B$ states that A has type B in context Γ.

(axiom)	$\langle\rangle \vdash s_1 : s_2$	$(s_1, s_2) \in \boldsymbol{A}$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	$x \notin \mathrm{DOM}(\Gamma)$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$	$x \notin \mathrm{DOM}(\Gamma)$
(Π)	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3}$	$(s_1, s_2, s_3) \in \boldsymbol{R}$
(λ)	$\dfrac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash (\Pi x{:}A.B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi x{:}A.B)}$	
(appl)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$	
(conv)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$	

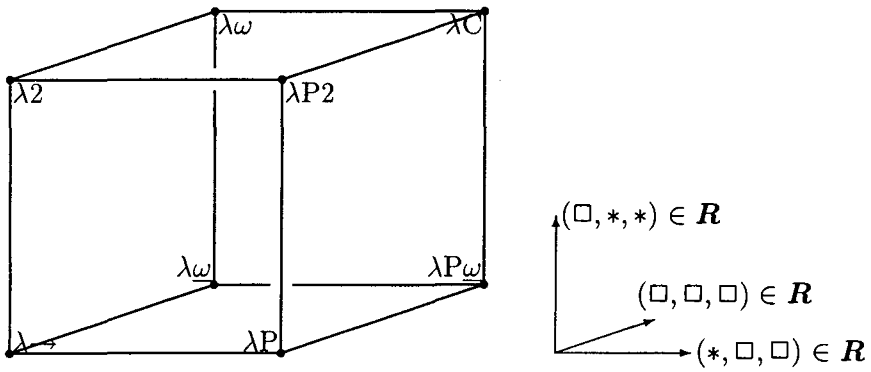
A context Γ is *legal* if there are A, B such that $\Gamma \vdash A : B$. A term A is *legal* if there are Γ, B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.

An important class of examples of PTSs is formed by the eight PTSs of the so-called Barendregt Cube. These systems all have $\{*, \Box\}$ as set of sorts, and $*{:}\Box$ as only axiom, but they differ on the Π-formation rules that are allowed, depending on which triples are in $\boldsymbol{R}$:

$\lambda{\to}$	$(*,*,*)$			
$\lambda 2$	$(*,*,*)$	$(\Box,*,*)$		
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 can be depicted in the Barendregt Cube (see Figure 13).

The systems in the Cube are related to many other type systems. The overview below is taken from [5].



$\lambda\omega$
λC
$\lambda 2$
$\lambda P2$
$\lambda\underline{\omega}$
$\lambda P\underline{\omega}$
$\lambda{\to}$
λP
$(\square,*,*)\in \boldsymbol{R}$
$(\square,\square,\square)\in \boldsymbol{R}$
$(*,\square,\square)\in \boldsymbol{R}$

Figure 13: The Barendregt Cube

System	Related system	Names, references
$\lambda{\to}$	λ^{τ}	simply typed λ-calculus; [30], [4] (Appendix A), [65] (Chapter 14)
$\lambda 2$	F	second order typed λ-calculus; [56], [103]
λP	AUT-QE[1]	[21]
	LF[2]	[59]
$\lambda P2$		[84]
$\lambda\underline{\omega}$	POLYREC	[102]
$\lambda\omega$	Fω	[56]
λC	CC	Calculus of Constructions; [35]

Another PTS that occurs in this thesis is the Extended Calculus of Con-

[1]A more precise study of AUT-QE, respecting the parameter structure of AUT-QE, shows that AUT-QE can be positioned a little bit higher in the Cube: exactly inbetween λP and λC. See Chapter 6, especially Section 6e4. — footnote by the author.

[2]In Chapter 6, Section 6e2, we show that the practical use of LF does not use the full power of λP. In the refinement of the Barendregt Cube presented there, we show that the use of LF in practice corresponds to a system that is inbetween $\lambda{\to}$ and λP. — footnote by the author.

structions ECC (see [86]). This is a PTS with

$$\begin{array}{rcl} \boldsymbol{S} & = & \mathbb{N}; \\ \boldsymbol{A} & = & \{(n, n+1) \mid n \in \mathbb{N}\}; \\ \boldsymbol{R} & = & \{(m, 0, 0) \mid m \in \mathbb{N}\} \cup \{(m, n, r) \mid 0 \leq m, n \leq r\}. \end{array}$$

This is indeed an extension of λC (write $*$ for 0 and $\square$ for 1).

Ad Metaproperties of PTSs

Pure Type Systems have some important meta-properties, which we describe below. The proofs can be found in [55] and [54]. Throughout this section, $\vdash$ denotes derivability in a PTS with a certain specification $\mathfrak{S} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$.

Lemma A.21 (Restricted Weakening) *If $\Gamma \vdash A : B$, we may assume the derivation of $\Gamma \vdash A : B$ to contain only applications of the rule (weak) that are of the form*

$$\frac{\Gamma \vdash u : B \qquad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash u : B}$$

where $u \in \mathbb{V} \cup \mathbb{C}$.

Lemma A.22 (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. $\mathrm{FV}(B), \mathrm{FV}(C) \subseteq \mathrm{DOM}(\Gamma)$*;*
3. $\mathrm{FV}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ *for $1 \leq i \leq n$.*

Lemma A.23 (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 A.24 (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 A.25 (Thinning Lemma) *If Δ is legal, $\Gamma \subseteq \Delta$ and $\Gamma \vdash A : B$, then $\Delta \vdash A : B$.*

Lemma A.26 (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 A.27 (Generation Lemma)

1. *If $\Gamma \vdash c : C$ for a $c \in \mathbb{C}$ then there is $s \in \boldsymbol{S}$ such that $C =_\beta s$ and $(c{:}s) \in \boldsymbol{A}$;*
2. *If $\Gamma \vdash x : C$ 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 A.28 (Correctness of Types) *If $\Gamma \vdash A : B$ then $B \equiv s$ or $\Gamma \vdash B : s$ for some $s \in \boldsymbol{S}$.*

Lemma A.29 (Subterm Lemma) *If A is legal and B is a subterm of A, then B is legal.*

Lemma A.30 (Subject Reduction) *If $\Gamma \vdash A : B$ and $A \to_\beta A'$ then $\Gamma \vdash A' : B$.*

Lemma A.31 (Strengthening Lemma) *If $\Gamma, x{:}A, \Delta \vdash B : C$ and $x \notin \mathrm{FV}(\Delta) \cup \mathrm{FV}(B) \cup \mathrm{FV}(C)$, then $\Gamma, \Delta \vdash B : C$.*

The proof of this lemma is due to Van Benthem Jutting [12].

Lemma A.32 (Unicity of Types) *If $\mathfrak{S}$ is singly sorted, $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$, then $B_1 =_\beta B_2$.*

Lemma A.33 (Strong Permutation Lemma) *If* $\Gamma, x{:}A, y{:}B, \Delta \vdash C : D$ *and* $x \notin \mathrm{FV}(B)$*, then* $\Gamma, y{:}B, x{:}A, \Delta \vdash C : D$.

Definition A.34 (Topsort) A sort s is a *topsort* if there is no $s' \in \boldsymbol{S}$ such that $(s, s') \in \boldsymbol{A}$.

Lemma A.35 (Topsort Lemma) *If* s *is a topsort and* $\Gamma \vdash A : s$ *then* A *is not of the form* A_1A_2 *or* $\lambda x{:}A_1.A_2$.

Theorem A.36 (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.

Appendix B

Type systems in this thesis

Ba The Ramified Theory of Types

Ba1 RTT

Definition B.1 (Propositional functions, 2.3) We define a collection $\mathcal{P}$ of *propositional functions* (pfs), and for each element f of $\mathcal{P}$ we simultaneously define the collection $\text{FV}(f)$ of *free variables* of f:

1. If $i_1, \ldots, i_{\mathfrak{a}(R)} \in \mathcal{A} \cup \mathcal{V}$ then $R(i_1, \ldots, i_{\mathfrak{a}(R)}) \in \mathcal{P}$.
   $\text{FV}\big(R(i_1, \ldots, i_{\mathfrak{a}(R)})\big) \stackrel{\text{def}}{=} \{i_1, \ldots, i_{\mathfrak{a}(R)}\} \cap \mathcal{V}$;

2. If $f, g \in \mathcal{P}$ then $f \vee g \in \mathcal{P}$ and $\neg f \in \mathcal{P}$.
   $\text{FV}(f \vee g) \stackrel{\text{def}}{=} \text{FV}(f) \cup \text{FV}(g)$; $\text{FV}(\neg f) \stackrel{\text{def}}{=} \text{FV}(f)$;

3. If $f \in \mathcal{P}$ and $x \in \text{FV}(f)$ then $\forall x[f] \in \mathcal{P}$.
   $\text{FV}(\forall x[f]) \stackrel{\text{def}}{=} \text{FV}(f) \setminus \{x\}$;

4. If $n \in \mathbb{N}$ and $k_1, \ldots, k_n \in \mathcal{A} \cup \mathcal{V} \cup \mathcal{P}$, then $z(k_1, \ldots, k_n) \in \mathcal{P}$.
   $\text{FV}(z(k_1, \ldots, k_n)) \stackrel{\text{def}}{=} \{z, k_1, \ldots, k_n\} \cap \mathcal{V}$.
   If $n = 0$ then we write $z()$ in order to distinguish the pf $z()$ from the variable z;

5. All pfs can be constructed by using the construction-rules 1, 2, 3 and 4 above.

Definition B.2 (Ramified types, 2.37)

1. 0^0 is a ramified type;
2. If $t_1^{a_1}, \ldots, t_n^{a_n}$ are ramified types, and $a \in \mathbb{N}$, $a > \max(a_1, \ldots, a_n)$, then $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ is a ramified type (if $n = 0$ then take $a \geq 0$);
3. All ramified types can be constructed using the rules 1 and 2.

If t^a is a ramified type, then a is called the *order* of t^a.

Definition B.3 (Predicative types, 2.41)

1. 0^0 is a predicative type;
2. If ${t_1}^{a_1}, \ldots, {t_n}^{a_n}$ are predicative types, and $a = 1 + \max(a_1, \ldots, a_n)$ (take $a = 0$ if $n = 0$), then $(t_1^{a_1}, \ldots, t_n^{a_n})^a$ is a predicative type;
3. All predicative types can be constructed using the rules 1 and 2 above.

Definition B.4 (Contexts, 2.43) Let $x_1, \ldots, x_n \in \mathcal{V}$ be distinct variables, and assume $t_1^{a_1}, \ldots, t_n^{a_n}$ are ramified types. Then $\{x_1{:}t_1^{a_1}, \ldots, x_n{:}t_n^{a_n}\}$ is a *context*. The set $\{x_1, \ldots, x_n\}$ is called the *domain* of the context and is denoted by $\mathrm{dom}(\{x_1{:}t_1^{a_1}, \ldots, x_n{:}t_n^{a_n}\})$.

Definition B.5 (Ramified Theory of Types: RTT, 2.45) The *judgements* $\Gamma \vdash f : t^a$ is inductively defined as follows:

1. **(start)** For all a:
$$\vdash a : 0^0.$$
For all atomic pfs f:
$$\vdash f : ()^0;$$
2. **(connectives)** Assume $\Gamma \vdash f{:}(t_1^{a_1}, \ldots, t_n^{a_n})^a$, $\Delta \vdash g{:}(u_1^{b_1}, \ldots, u_m^{b_m})^b$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$ and $y \in \mathrm{dom}(\Delta)$. Then
$$\Gamma \cup \Delta \vdash f \vee g : \left(t_1^{a_1}, \ldots, t_n^{a_n}, u_1^{b_1}, \ldots, u_m^{b_m}\right)^{\max(a,b)};$$
$$\Gamma \vdash \neg f : (t_1^{a_1}, \ldots, t_n^{a_n})^a;$$

3. **(abstraction from parameters)** If $\Gamma \vdash f : (t_1^{a_1}, \ldots, t_m^{a_m})^a$, $t_{m+1}^{a_{m+1}}$ is a *predicative* type, $g \in \mathcal{A} \cup \mathcal{P}$ is a parameter of f, $\Gamma \vdash g : t_{m+1}^{a_{m+1}}$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$, then
$$\Gamma' \vdash h : (t_1^{a_1}, \ldots, t_{m+1}^{a_{m+1}})^{\max(a, a_{m+1}+1)}.$$
Here, h is a pf obtained by replacing all parameters g' of f which are α_Γ-equal to g by y. Moreover, Γ' is the subset of the context $\Gamma \cup \{y : t_{m+1}^{a_{m+1}}\}$ such that $\mathrm{dom}(\Gamma')$ contains exactly all the variables that occur in h;

4. **(abstraction from pfs)** If $(t_1^{a_1}, \ldots, t_m^{a_m})^a$ is a *predicative* type, $\Gamma \vdash f : (t_1^{a_1}, \ldots, t_m^{a_m})^a$, $x < z$ for all $x \in \mathrm{dom}(\Gamma)$, and $y_1 < \cdots < y_n$ are the free variables of f, then
$$\Gamma' \vdash z(y_1, \ldots, y_n) : (t_1^{a_1}, \ldots, t_m^{a_m}, (t_1^{a_1}, \ldots, t_m^{a_m})^a)^{a+1},$$
where Γ' is the subset of $\Gamma \cup \{z{:}(t_1^{a_1}, \ldots, t_m^{a_m})^a\}$ such that $\mathrm{dom}(\Gamma') = \{y_1, \ldots, y_n, z\}$;

5. **(weakening)** If Γ, Δ are contexts, $\Gamma \subseteq \Delta$, and $\Gamma \vdash f : t^a$, then also $\Delta \vdash f : t^a$;

6. **(substitution)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y : t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $\Gamma \vdash k : t_i^{a_i}$ then
$$\Gamma' \vdash f[y{:=}k] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n})^b.$$
Here, $b = 1 + \max(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c)$, and
$$c = \max\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\}$$
(if $n = 1$ and $\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\} = \varnothing$ then take $b = 0$) and once more, Γ' is the subset of $\Gamma \cup \{y : t_i^{a_i}\}$ such that $\mathrm{dom}(\Gamma')$ contains exactly all the variables that occur in $f[y{:=}k]$;

7. **(permutation)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $x < y'$ for all $x \in \mathrm{dom}(\Gamma)$, then
$$\Gamma' \vdash f[y{:=}y'] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n}, t_i^{a_i})^a.$$

Γ' is the subset of $\Gamma \cup \{y{:}t_i^{a_i}, y'{:}t_i^{a_i}\}$ such that domΓ' contains exactly all the variables that occur in $f[y{:=}y']$;

8. **(quantification)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, then

$$\Gamma \vdash \forall y{:}t_i^{a_i}[f] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n})^a.$$

Ba2 λRTT

Definition B.6 (Terms of λRTT, 4.3) Let $\mathcal{A}$, $\mathcal{V}$ and $\mathcal{R}$ be as in Chapter 2. Define the set $\mathcal{T}$ of terms of λRTT by:

$$\begin{array}{rcl} \mathcal{T} & ::= & *_s \mid *_{\mathbb{N}^+} \mid \Box_{\mathbb{N}^+} \mid \mathcal{A} \mid \mathcal{V} \mid \mathcal{R} \mid \iota \mid \bot \mid \\ & & \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}. \end{array}$$

Definition B.7 (Derivation Rules for λRTT, 4.9) Let s, s_1, s_2 range over $\boldsymbol{S} = \{*_s, \Box_1, \Box_2, \ldots, *_1, *_2, \ldots\}$, and let

$$\begin{array}{rcl} \boldsymbol{R} & = & \{(*_s, \Box_n, \Box_n) \mid n \geq 1\} \cup \\ & & \{(\Box_m, \Box_n, \Box_n) \mid 1 \leq m < n\} \cup \\ & & \{(*_m, *_n, *_{\max(m,n)}) \mid m, n \geq 1\} \cup \\ & & \{(*_s, *_n, *_n) \mid n \geq 1\} \cup \\ & & \{(\Box_m, *_n, *_n) \mid 1 \leq m < n\}. \end{array}$$

The derivation rules for λRTT are as follows:

(Axioms)
$$\vdash *_n : \Box_n \qquad (n \geq 1)$$
$$\vdash \iota : *_s$$
$$\vdash \bot : *_1$$
$$\vdash a : \iota \qquad (a \in \mathcal{A})$$
$$\vdash R : \underbrace{\iota \to \cdots \to \iota}_{\mathfrak{a}(R) \text{ times } \iota} \to *_1 \qquad (R \in \mathcal{R})$$

(Start)
$$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A}$$

(Weak)
$$\frac{\Gamma \vdash M : N \qquad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash M : N}$$

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

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

$$(\Pi\text{-el})\quad \frac{\Gamma\vdash M:\Pi x{:}A.B \qquad \Gamma\vdash N:A}{\Gamma\vdash MN:B[x{:=}N]}$$

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

$$(\text{Incl})\quad \frac{\Gamma\vdash A:*_n}{\Gamma\vdash A:*_{n+1}}$$

Bb AUTOMATH

Bb1 AUT-68

Definition B.8 (Expressions, 5.1) We define the set $\mathcal{E}$ of AUT-*68-expressions* inductively:

(variable) If $x\in\mathcal{V}$ then $x\in\mathcal{E}$;

(parameter) If $a\in\mathcal{C}$, $n\in\mathbb{N}$ ($n=0$ is allowed) and $\Sigma_1,\ldots,\Sigma_n\in\mathcal{E}$ then $a(\Sigma_1,\ldots,\Sigma_n)\in\mathcal{E}$;

(abstraction) If $x\in\mathcal{V}$, $\Sigma\in\mathcal{E}\cup\{\mathsf{type}\}$ and $\Omega\in\mathcal{E}$ then $[x{:}\Sigma]\Omega\in\mathcal{E}$;

(application) If $\Sigma_1,\Sigma_2\in\mathcal{E}$ then $\langle\Sigma_2\rangle\Sigma_1\in\mathcal{E}$.

We define also $\mathcal{E}^+\stackrel{\text{def}}{=}\mathcal{E}\cup\{\mathsf{type}\}$.

Definition B.9 (Books and lines, 5.7) An AUT-68-*book* is a finite list (possibly empty) of (AUT-68)-lines (to be defined next). If $l_1,\ldots,l_n$ are the lines of book $\mathfrak{B}$, we write $\mathfrak{B}\equiv l_1,\ldots,l_n$.

An AUT-68-*line* is a 4-tuple $(\Gamma;k;\Sigma_1;\Sigma_2)$. Here,

- Γ is a context, i.e. a finite (possibly empty) list $x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n$, where the x_is are different elements of $\mathcal{V}$ and the α_is are elements of $\mathcal{E}^+$;
- k is an element of $\mathcal{V}\cup\mathcal{C}$;
- Σ_1 can be (only):

- ∘ The symbol — (if $k \in \mathcal{V}$);
- ∘ The symbol PN (if $k \in \mathcal{C}$);
- ∘ An element of $\mathcal{E}$ (if $k \in \mathcal{C}$);

- Σ_2 is an element of $\mathcal{E}^+$.

Definition B.10 (Correct books and contexts, 5.10) A book $\mathfrak{B}$ and a context Γ are *correct* if $\mathfrak{B}; \Gamma \vdash \text{OK}$ can be derived with the following rules

(axiom) $$\varnothing; \varnothing \vdash \text{OK}$$

(context ext.) $$\frac{\mathfrak{B}_1, (\Gamma; x; —; \alpha), \mathfrak{B}_2; \Gamma \vdash \text{OK}}{\mathfrak{B}_1, (\Gamma; x; —; \alpha), \mathfrak{B}_2; \Gamma, x{:}\alpha \vdash \text{OK}}$$

(book ext.: var1) $$\frac{\mathfrak{B}; \Gamma \vdash \text{OK}}{\mathfrak{B}, (\Gamma; x; —; \texttt{type}); \varnothing \vdash \text{OK}}$$

(book ext.: var2) $$\frac{\mathfrak{B}; \Gamma \vdash \Sigma_2 : \texttt{type}}{\mathfrak{B}, (\Gamma; x; —; \Sigma_2); \varnothing \vdash \text{OK}}$$

(book ext.: pn1) $$\frac{\mathfrak{B}; \Gamma \vdash \text{OK}}{\mathfrak{B}, (\Gamma; k; \text{PN}; \texttt{type}); \varnothing \vdash \text{OK}}$$

(book ext.: pn2) $$\frac{\mathfrak{B}; \Gamma \vdash \Sigma_2 : \texttt{type}}{\mathfrak{B}, (\Gamma; k; \text{PN}; \Sigma_2); \varnothing \vdash \text{OK}}$$

(book ext.: def1) $$\frac{\mathfrak{B}; \Gamma \vdash \Sigma_1 : \texttt{type}}{\mathfrak{B}, (\Gamma; k; \Sigma_1; \texttt{type}); \varnothing \vdash \text{OK}}$$

(book ext.: def2)
$$\frac{\mathfrak{B}; \Gamma \vdash \Sigma_2 : \texttt{type} \qquad \mathfrak{B}; \Gamma \vdash \Sigma_1 : \Sigma_2' \qquad \mathfrak{B}; \Gamma \vdash \Sigma_2 =_{\beta\text{d}} \Sigma_2'}{\mathfrak{B}, (\Gamma; k; \Sigma_1; \Sigma_2); \varnothing \vdash \text{OK}}$$

For the (book ext.) rules, we assume that the introduced identifiers $x \in \mathcal{V}$ and $k \in \mathcal{C}$ do not occur anywhere in $\mathfrak{B}$ and Γ.

Definition B.11 (Correct statements, 5.11) A statement $\mathfrak{B}; \Gamma \vdash \Sigma : \Omega$ is *correct* if it can be derived with the rules below (the start rule uses the notions of correct context and correct book as given in Definition 5.10).

(start)
$$\frac{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2\vdash \text{OK}}{\mathfrak{B};\Gamma_1,x{:}\alpha,\Gamma_2\vdash x{:}\alpha}$$

(parameters)
$$\frac{\begin{array}{c}\mathfrak{B}\equiv\mathfrak{B}_1,(x_1{:}\alpha_1,\ldots,x_n{:}\alpha_n;b;\Omega_1;\Omega_2),\mathfrak{B}_2\\ \mathfrak{B};\Gamma\vdash\Sigma_i{:}\alpha_i[x_1,\ldots,x_{i-1}{:=}\Sigma_1,\ldots,\Sigma_{i-1}](i=1,\ldots,n)\end{array}}{\mathfrak{B};\Gamma\vdash b(\Sigma_1,\ldots,\Sigma_n):\Omega_2[x_1,\ldots,x_n{:=}\Sigma_1,\ldots,\Sigma_n]}$$

(abstr.1)
$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}\mathtt{type}\qquad\mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Omega_1{:}\mathtt{type}}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Omega_1:\mathtt{type}}$$

(abstr.2)
$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1{:}\mathtt{type}\qquad\mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Omega_1{:}\mathtt{type}\qquad\mathfrak{B};\Gamma,x{:}\Sigma_1\vdash\Sigma_2{:}\Omega_1}{\mathfrak{B};\Gamma\vdash[x{:}\Sigma_1]\Sigma_2:[x{:}\Sigma_1]\Omega_1}$$

(application)
$$\frac{\mathfrak{B};\Gamma\vdash\Sigma_1:[x{:}\Omega_1]\Omega_2\qquad\mathfrak{B};\Gamma\vdash\Sigma_2:\Omega_1}{\mathfrak{B};\Gamma\vdash\langle\Sigma_2\rangle\Sigma_1:\Omega_2[x{:=}\Sigma_2]}$$

(conversion)
$$\frac{\mathfrak{B};\Gamma\vdash\Sigma:\Omega_1\qquad\mathfrak{B};\Gamma\vdash\Omega_2{:}\mathtt{type}\qquad\mathfrak{B};\Gamma\vdash\Omega_1=_{\beta\mathrm{d}}\Omega_2}{\mathfrak{B};\Gamma\vdash\Sigma:\Omega_2}$$

When using the parameter rule, we assume that $\mathfrak{B};\Gamma\vdash$ OK, even if $n=0$.

Bb2 $\lambda68$

Definition B.12 (Terms, 5.21.1) The terms of $\lambda68$ form a set $\mathcal{T}$ defined by

$$\mathcal{T} ::= \mathcal{V}\mid\mathcal{C}\mid\boldsymbol{S}\mid\mathcal{TT}\mid\lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T}\mid\S\mathcal{V}{:}\mathcal{T}.\mathcal{T}\mid\Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}\mid\P\mathcal{V}{:}\mathcal{T}.\mathcal{T},$$

where $\boldsymbol{S}$ is the set of sorts $\{*,\square,\triangle\}$.

Definition B.13 (Contexts, 5.21.2) We define the notion of context inductively:

- $\varnothing;\varnothing$ is a context; DOM $(\varnothing;\varnothing)=\varnothing$;
- If $\Delta;\Gamma$ is a context, $x\in\mathcal{V}$, x does not occur in $\Delta;\Gamma$ and $A\in\mathcal{T}$, then $\Delta;\Gamma,x{:}A$ is a context (x is a newly introduced variable); DOM $(\Delta;\Gamma)=$ DOM $(\Delta;\Gamma)\cup\{x\}$;
- If $\Delta;\Gamma$ is a context, $b\in\mathcal{C}$, b does not occur in $\Delta;\Gamma$ and $A\in\mathcal{T}$ then $\Delta,b{:}A;\Gamma$ is a context (in this case b is a *primitive* constant; cf. the primitive notions of AUTOMATH in Section 5a1); DOM $(\Delta,b{:}A;\Gamma)=$ DOM $(\Delta;\Gamma)\cup\{b\}$;

- If $\Delta;\Gamma$ is a context, $b \in \mathcal{C}$, b does not occur in $\Delta;\Gamma$, $A \in \mathcal{T}$, and $T \in \mathcal{T}$, then $\Delta, b{:=}T{:}A;\Gamma$ is a context (in this case b is a *defined* constant; cf. the definitions of AUTOMATH in Section 5a1); $\text{DOM}(\Delta, b{:=}T{:}A;\Gamma) = \text{DOM}(\Delta;\Gamma) \cup \{b\}$.

Definition B.14 (Derivation rules, 5.21.3)

(Axiom) $$;\vdash * : \Box$$

(Start: v) $$\frac{\Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x{:}A \vdash x : A}$$ where $s \equiv *, \Box$

(Start: pc) $$\frac{\Delta;\Gamma \vdash B : s_1 \qquad \Delta;\vdash \P\Gamma.B : s_2}{\Delta, b{:}\,\P\Gamma.B;\vdash b : \P\Gamma.B}$$ where $s_1 \equiv *, \Box$

(Start: dc) $$\frac{\Delta;\Gamma \vdash T : B : s_1 \qquad \Delta;\vdash \P\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\Gamma.B);\vdash b : \P\Gamma.B}$$ where $s_1 \equiv *, \Box$

(Weak: v) $$\frac{\Delta;\Gamma \vdash M : N \qquad \Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x{:}A \vdash M : N}$$ where $s \equiv *, \Box$

(Weak: pc) $$\frac{\Delta;\vdash M : N \qquad \Delta;\Gamma \vdash B : s_1 \qquad \Delta;\vdash \P\Gamma.B : s_2}{\Delta, b{:}\,\P\Gamma.B;\vdash M : N}$$ where $s_1 \equiv *, \Box$

(Weak: dc) $$\frac{\Delta;\vdash M : N \qquad \Delta;\Gamma \vdash T : B : s_1 \qquad \Delta;\vdash \P\Gamma.B : s_2}{\Delta, b{:=}(\S\,\Gamma.T){:}(\P\Gamma.B);\vdash M : N}$$ where $s_1 \equiv *, \Box$

(Π-form) $$\frac{\Delta;\Gamma \vdash A : * \qquad \Delta;\Gamma, x{:}A \vdash B : *}{\Delta;\Gamma \vdash (\Pi x{:}A.B) : *}$$

(¶-form) $$\frac{\Delta;\Gamma \vdash A : s_1 \qquad \Delta;\Gamma, x{:}A \vdash B : s_2}{\Delta;\Gamma \vdash (\P x{:}A.B) : \Delta}$$ where $s_1 \equiv *, \Box$

(λ) $$\frac{\Delta;\Gamma \vdash \Pi x{:}A.B : * \qquad \Delta;\Gamma, x{:}A \vdash F : B}{\Delta;\Gamma \vdash (\lambda x{:}A.F) : (\Pi x{:}A.B)}$$

(App$_1$) $$\frac{\Delta;\Gamma \vdash M : \Pi x{:}A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x{:=}N]}$$

$$(\mathbf{App_2})\qquad \frac{\Delta;\Gamma\vdash M:\P x{:}A.B \qquad \Delta;\Gamma\vdash N:A}{\Delta;\Gamma\vdash MN:B[x{:=}N]}$$

$$(\mathbf{Conv})\qquad \frac{\Delta;\Gamma\vdash M:A \qquad \Delta;\Gamma\vdash B:s \qquad \Delta\vdash A=_{\beta\delta}B}{\Delta;\Gamma\vdash M:B}.$$

The newly introduced variables in the Start-rules and Weakening-rules are assumed to be fresh. Moreover, when introducing a variable x with a "pc"-rule or a "dc"-rule, we assume $x\in\mathcal{C}$, and when introducing x via a "v"-rule, we assume $x\in\mathcal{V}$.

Bc $\vec{\mathbf{C}}\vec{\mathbf{D}}$-PTSs and their subsystems

Bc1 PTSs with parameters and definitions

Definition B.15 (Terms, 6.1) The set $\mathcal{T_P}$ of *parametric terms* is defined together with the set $\mathcal{L_V}$ of *lists of variables* and the set $\mathcal{L_T}$ of *lists of terms*:

$$\begin{array}{rcl}\mathcal{T_P} & ::= & \mathcal{V}\mid\boldsymbol{S}\mid\mathcal{C}(\mathcal{L_T})\mid\mathcal{T_P}\mathcal{T_P}\mid\lambda\mathcal{V}{:}\mathcal{T_P}.\mathcal{T_P}\mid\\ & & \Pi\mathcal{V}{:}\mathcal{T_P}.\mathcal{T_P}\mid\mathcal{C}(\mathcal{L_V}){=}\mathcal{T_P}{:}\mathcal{T_P}\ \text{IN}\ \mathcal{T_P};\\ \mathcal{L_V} & ::= & \varnothing\mid\langle\mathcal{L_V},\mathcal{V}{:}\mathcal{T_P}\rangle;\\ \mathcal{L_T} & ::= & \varnothing\mid\langle\mathcal{L_T},\mathcal{T_P}\rangle.\end{array}$$

where $\mathcal{V}$ is a set of variables, $\mathcal{C}$ is a set of constants, and $\boldsymbol{S}$ is a set of sorts.

Definition B.16 The set of *contexts* is given by

$$\mathcal{C_P}\ ::=\ \varnothing\mid\langle\mathcal{C_P},\mathcal{V}{:}\mathcal{T_P}\rangle\mid\langle\mathcal{C_P},\mathcal{C}(\mathcal{L_V}){=}\mathcal{T_P}{:}\mathcal{T_P}\rangle\mid\langle\mathcal{C_P},\mathcal{C}(\mathcal{L_V}){:}\mathcal{T_P}\rangle.$$

Definition B.17 ($\vec{\mathbf{C}}$: parametric constants, 6.20) The *typing relation* $\vdash^{\vec{C}}$ is the smallest relation on $\mathcal{C_P}\times\mathcal{T_P}\times\mathcal{T_P}$ closed under the rules in Definition A.20 and the following ones (we write $\Delta\equiv x_1{:}B_1,\ldots,x_n{:}B_n$):

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

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

where $s \in \boldsymbol{S}$ and the c that is introduced in the $\vec{\text{C}}$-weakening rule is assumed to be Γ-fresh.

Definition B.18 ($\vec{\mathbf{D}}$: parametric definitions 6.21) The *typing relation* $\vdash^{\vec{D}}$ is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ closed under the rules in Definition A.20 and the following ones:

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

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

$$(\vec{\mathbf{D}}\textbf{-form}) \quad \frac{\Gamma, c(\Delta){=}a{:}A \vdash^{\vec{D}} B : s}{\Gamma \vdash^{\vec{D}} c(\Delta){=}a{:}A \text{ IN } B : s}$$

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

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

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

Definition B.19 (Pure Type Systems with (parametric) constants and (parametric) definitions, 6.22) Let $\mathfrak{S}$ be a specification.

- A *pure type system with (parametric) constants* $\vec{\text{C}}$-PTS is denoted as $\lambda^{\vec{C}}(\mathfrak{S})$ 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 $\vdash^{\vec{C}}$;
- A *pure type system with (parametric) definitions* $\vec{\text{D}}$-PTS is denoted as $\lambda^{\vec{D}}(\mathfrak{S})$ 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 $\vdash^{\vec{D}}$;
- A *pure type system with (parametric) constants and (parametric) definitions* $\vec{\text{C}}\vec{\text{D}}$-PTS is denoted as $\lambda^{\vec{C}\vec{D}}(\mathfrak{S})$ 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 $\vdash^{\vec{C}\vec{D}}$, which is the smallest relation on $\mathcal{C}_P \times \mathcal{T}_P \times \mathcal{T}_P$ that is closed under the rules of Definition A.20 and the rules of $\vdash^{\vec{C}}$ and $\vdash^{\vec{D}}$.

Bc2 PTSs with restricted parameters and definitions

Definition B.20 (Parametric Specification, 6.64) 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 A.17), 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.

Definition B.21 (Ĉ: restricted constants, 6.65) Let $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. The *typing relation* $\vdash^{\hat{C}}$ is obtained from the relation $\vdash^{\vec{C}}$ by replacing rule ($\vec{\text{C}}$-weak) by the following rule (Ĉ-weak):

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

Definition B.22 (D̂: restricted definitions, 6.66) Let $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. The *typing relation* $\vdash^{\hat{D}}$ is obtained from the relation $\vdash^{\vec{D}}$ by replacing rule ($\vec{\text{D}}$-weak) by the following rules (D̂-weak):

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

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

Definition B.23 Let $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. The *typing relation* $\vdash^{\hat{C}\hat{D}}$ is obtained from the relation $\vdash^{\vec{C}\vec{D}}$ by replacing rule ($\vec{\text{C}}$-weak) by rule (Ĉ-weak) and rule ($\vec{\text{D}}$-weak) by rules (D̂-weak).

Definition B.24 (Pure Type Systems with Restricted Parameters and Restricted Parametric Definitions, 6.68) Let $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{P})$ be a parametric specification. The pure type system with restricted parameters and restricted parametric definitions (ĈD̂-PTS) and parametric specification $\mathfrak{S}$ is denoted $\lambda^{\hat{C}\hat{D}}(\mathfrak{S})$. 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 $\vdash^{\hat{C}\hat{D}}$.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] 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.

[3] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, 1993.

[4] 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.

[5] H.P. Barendregt. Lambda calculi with types. In [2], pages 117–309. Oxford University Press, 1992.

[6] 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.

[7] P. Benacerraf and H. Putnam, editors. *Philosophy of Mathematics.* Cambridge University Press, second edition, 1983.

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

[9] L.S. van Benthem Jutting. A Translation of Landau's "Grundlagen" in AUTOMATH. Technical report, Eindhoven University of Technology, 1976.

[10] 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).

[11] L.S. van Benthem Jutting. Description of AUT-68. Technical Report 12, Eindhoven University of Technology, 1981. Also in [95], pp. 251–273.

[12] L.S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.

[13] 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.

[14] E.W. Beth. *The Foundations of Mathematics.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1959.

[15] R. Bloo. *Preservation of Strong Normalisation for Explicit Substitutions.* PhD thesis, Eindhoven University of Technology, 1997.

[16] R. Bloo, F. Kamareddine, and R. Nederpelt. The Barendregt Cube with Definitions and Generalised Reduction. *Information and Computation*, 126(2):123–143, 1996.

[17] R. Bloo, F. Kamareddine, and R. Nederpelt. On λ- and π-conversion in the λ-cube and the need for abbreviations. *Submitted to APAL*, 1997.

[18] 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.

[19] L.E.J. Brouwer. *Over de Grondslagen der Wiskunde.* PhD thesis, Universiteit van Amsterdam, 1907. Dutch; English translation in [63].

[20] N.G. de Bruijn. AUTOMATH, a language for mathematics. Technical Report 68-WSK-05, T.H.-Reports, Eindhoven University of Technology, 1968.

[21] 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 [95], pages 73–100.

[22] N.G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In P. Dybjer et al., editors, *Proceedings of the Workshop on Programming Languages.* Marstrand, Sweden, 1987. Reprinted in [95] in combination with *Formalizing the Mathematical Vernacular* (formerly unpublished, 1982), Examples of an MV Book.

[23] N.G. de Bruijn. Reflections on Automath. Eindhoven University of Technology, 1990. Also in [95], pages 201–228.

[24] C. Burali-Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo Matematico di Palermo*, 11:154–164, 1897. English translation in [61], pages 104–112.

[25] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (Erster Artikel). *Mathematische Annalen*, 46:481–512, 1895.

[26] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (Zweiter Artikel). *Mathematische Annalen*, 49:207–246, 1897.

[27] A.-L. Cauchy. *Cours d'Analyse de l'Ecole Royale Polytechnique.* Debure, Paris, 1821. Also as *Œuvres Complètes* (2), volume III, Gauthier-Villars, Paris, 1897.

[28] A. Church. A set of postulates for the foundation of logic (1). *Annals of Mathematics*, 33:346–366, 1932.

[29] A. Church. A set of postulates for the foundation of logic (2). *Annals of Mathematics*, 34:839–864, 1933.

[30] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[31] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[32] A. Church. Comparison of Russell's resolution of the semantic antinomies with that of Tarski. *The Journal of Symbolic Logic*, 41:747–760, 1976.

[33] A.B. Compagnoni. *Higher-Order Subtyping with Intersection Types.* PhD thesis, Katholieke Universiteit Nijmegen, 1995.

[34] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New Jersey, 1986.

[35] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[36] H.B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Science of the USA*, 20:584–590, 1934.

[37] H.B. Curry. *Foundations of Mathematical Logic.* McGraw-Hill Series in Higher Mathematics. McGraw-Hill Book Company, Inc., 1963.

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

[39] 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 [95], pages 101–126.

[40] D.T. van Daalen. *The Language Theory of Automath.* PhD thesis, Eindhoven University of Technology, 1980.

[41] R. Dedekind. *Stetigkeit und irrationale Zahlen.* Vieweg & Sohn, Braunschweig, 1872.

[42] G. Dowek et al. The Coq Proof Assistant Version 5.6, Users Guide. Technical Report 134, INRIA, Le Chesney, 1991.

[43] Euclid. The Elements. 325 B.C.. English translation in [60].

[44] S. Feferman. Toward useful type-free theories I. *Journal of Symbolic Logic*, 49:75–111, 1984.

[45] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Nebert, Halle, 1879. Also in [61], pages 1–82.

[46] G. Frege. *Grundlagen der Arithmetik, eine logisch-mathematische Untersuchung über den Begriff der Zahl.* , Breslau, 1884.

[47] G. Frege. *Funktion und Begriff, Vortrag gehalten in der Sitzung vom 9. Januar der Jenaischen Gesellschaft für Medicin und Naturwissenschaft.* Hermann Pohle, Jena, 1891. English translation in [89], pages 137–156.

[48] G. Frege. *Grundgesetze der Arithmetik, begriffschriftlich abgeleitet,* volume I. Pohle, Jena, 1892. Reprinted 1962 (Olms, Hildesheim).

[49] G. Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik,* new series, 100:25–50, 1892. English translation in [89], pages 157–177.

[50] G. Frege. Ueber die Begriffschrift des Herrn Peano und meine eigene. *Berichte über die Verhandlungen der Königlich Sächsischen Gesellschaft der Wissenschaften zu Leipzig, Mathematisch-physikalische Klasse 48*, pages 361–378, 1896. English translation in [89], pages 234–248.

[51] G. Frege. Letter to Russell. English translation in [61], pages 127–128, 1902.

[52] G. Frege. *Grundgesetze der Arithmetik, begriffschriftlich abgeleitet,* volume II. Pohle, Jena, 1903. Reprinted 1962 (Olms, Hildesheim).

[53] C.I. Gerhardt, editor. Die philosophischen Schriften von Gottfried Wilhelm Leibniz. Berlin, 1890.

[54] J.H. Geuvers. *Logics and Type Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[55] 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.

[56] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[57] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. German; English translation in [61], pages 592–618.

[58] K. Gödel. Russell's mathematical logic. In P.A. Schlipp, editor, *The Philosophy of Bertrand Russell.* Evanston & Chicago, Northwestern University, 1944. Also in [7], pages 447–469.

[59] 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.

[60] T.L. Heath. *The Thirteen Books of Euclid's Elements.* Dover Publications, Inc., New York, 1956.

[61] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

[62] A. Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer Verlag, Berlin, 1934.

[63] A. Heyting, editor. *Brouwer: Collected Works*, volume 1. North-Holland, Amsterdam, 1975.

[64] D. Hilbert and W. Ackermann. *Grundzüge der Theoretischen Logik.* Die Grundlehren der Mathematischen Wissenschaften in Einzel-

darstellungen, Band XXVII. Springer Verlag, Berlin, first edition, 1928.

[65] 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.

[66] W.A. Howard. The formulas-as-types notion of construction. In [112], pages 479–490, 1980.

[67] P.B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, New York, 1995.

[68] B. Jacobs. Quotients in simple type theory. Draft version of 30 March 1994.

[69] F. Kamareddine and T. Laan. A Correspondence between Nuprl and the Ramified Theory of Types. Technical Report 96-12, TUE Computing Science Notes, 1996. Also as Technical Report TR-1996-18, Department of Computing Science, University of Glasgow, 1996.

[70] F. Kamareddine and T. Laan. A reflection on Russell's ramified types and Kripke's hierarchy of truths. *Journal of the Interest Group in Pure and Applied Logic*, 4(2):195–213, 1996.

[71] F. Kamareddine and R. Nederpelt. On stepwise explicit substitution. *International Journal of Foundations of Computer Science*, 4:197–240, 1993.

[72] F. Kamareddine and R.P. Nederpelt. A unified approach to type theory through a refined λ-calculus. *Theoretical Computer Science*, 136:183–216, 1994.

[73] F. Kamareddine and R.P. Nederpelt. Canonical typing and Π-conversion in the Barendregt Cube. *Journal of Functional Programming*, 6(2):245–267, 1996.

[74] S.C. Kleene and J.B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36:630–636, 1935.

[75] J.W. Klop. Term rewriting systems. In [2], pages 1–116. Oxford University Press, 1992.

[76] G.T. Kneebone. *Mathematical Logic and the Foundations of Mathematics.* D. Van Nostrand Comp., London, New York, Toronto, 1963.

[77] A.N. Kolmogorov. Zur Deutung der Intuitionistischen Logik. *Mathematisches Zeitschrift*, 35:58–65, 1932.

[78] S. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[79] T. Laan. A formalization of the Ramified Type Theory. Technical Report 94-33, TUE Computing Science Reports, Eindhoven University of Technology, 1994.

[80] T. Laan. A Modern View on the Ramified Theory of Types. In J.C. van Vliet, editor, *Proceedings CSN 95*, pages 122–133, Amsterdam, 1995. Stichting Mathematisch Centrum.

[81] T. Laan. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Eindhoven University of Technology, 1997.

[82] T. Laan and R.P. Nederpelt. A modern elaboration of the Ramified Theory of Types. *Studia Logica*, 57(2/3):243–278, 1996.

[83] E. Landau. *Grundlagen der Analysis.* , Leipzig, 1930.

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

[85] Z. Luo. ECC and extended Calculus of Constructions. Department of Computer Science, University of Edinburgh.

[86] 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.

[87] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[88] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, Amsterdam, 1975. North-Holland. Studies in Logic and the Foundations of Mathematics **80**.

[89] B. McGuinness, editor. *Gottlob Frege: Collected Papers on Mathematics, Logic, and Philosophy.* Basil Blackwell, Oxford, 1984.

[90] R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML.* MIT Press, Cambridge (Massachusetts)/London, 1990.

[91] R.P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types.* PhD thesis, Eindhoven University of Technology, 1973. Also in [95], pages 389–468.

[92] R.P. Nederpelt. Presentation of natural deduction. *Recueil des travaux de l'Institut Mathématique, Nouvelle série*, 2(10):115–126, 1977. Symposium: Set Theory. Foundations of Mathematics, Beograd 1977.

[93] R.P. Nederpelt. Type systems — basic ideas and applications. *Proceedings of CSN 1990*, pages 367–383, 1990. Stichting Mathematisch Centrum, Amsterdam.

[94] R.P. Nederpelt. The fine-structure of lambda calculus. Technical Report 92-07, Computing Science Notes, Eindhoven University of Technology, 1992.

[95] 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.

[96] M.J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science.* Springer Verlag, 1977.

[97] G. Peano. *Arithmetices principia, nova methodo exposita.* Bocca, Turin, 1889. English translation in [61], pages 83–97.

[98] G. Peano. *Formulaire de Mathématique.* Bocca, Turin, 1894–1908. 5 successive versions; the final edition issued as *Formulario Mathematico.*

[99] W. Peremans. Ups and downs of type theory. Technical Report 94-14, TUE Computing Science Notes, Eindhoven University of Technology, 1994.

[100] W. Van Orman Quine. *Set Theory and its Logic.* Harvard University Press, Cambridge, Massachusetts, 1963.

[101] F.P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society,* 2nd series, 25:338–384, 1926.

[102] G.R. Renardel de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Information and Computation,* 99:154–177, 1991.

[103] J.C. Reynolds. *Towards a theory of type structure*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

[104] A.C.M. van Rooij. *Analyse voor Beginners.* Epsilon Uitgaven, Utrecht, 1986.

[105] J.B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349, 1984.

[106] B. Russell. Letter to Frege. English translation in [61], pages 124–125, 1902.

[107] B. Russell. *The Principles of Mathematics.* Allen & Unwin, London, 1903.

[108] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Also in [61], pages 150–182.

[109] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. Also in [61], pages 355–366.

[110] K. Schütte. *Beweistheorie.* Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen, Band 103. Springer Verlag, Berlin, 1960.

[111] J.P. Seldin. Personal communication, 1996.

[112] J.P. Seldin and J.R. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[113] P. Severi. *Normalisation in Lambda Calculus and its Relation to Type Inference.* PhD thesis, Eindhoven University of Technology, 1996.

[114] 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.

[115] T. Streicher. *Semantics of Type Theory.* Birkhäuser, 1991.

[116] W.W. Tait. Infinitely long terms of transfinite type. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, Amsterdam, 1965. North-Holland.

[117] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936. German translation by L. Blauwstein from the Polish original (1933) with a postscript added.

[118] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[119] R. de Vrijer. A direct proof of the finite developments theorem. *The Journal of Symbolic Logic*, 50(2):339–343, 1985.

[120] H. Weyl. *Das Kontinuum.* Veit, Leipzig, 1918. German; also in: Das Kontinuum und andere Monographien, Chelsea Pub.Comp., New York, 1960.

[121] 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.

[122] R.L. Wilder. *The Foundations of Mathematics.* Robert E. Krieger Publishing Company, Inc., New York, second edition, 1965.

[123] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. *Math. Annalen*, 65:261–281, 1908.

[124] J. Zucker. Formalization of classical mathematics in Automath. In *Colloque International de Logique,* Clermont-Ferrand, pages 135–145, Paris, CNRS, 1977. Colloques Internationaux du Centre National de la Recherche Scientifique, 249.

Subject Index

Name Index

List of Figures

Summary

In this thesis we provide insight in the evolution of the notion of type in logic and mathematics during the last one hundred and twenty years. We want to stress that we do more than merely giving a historical overview. We not only describe the type systems that have been developed in this period, but also describe them in a modern terminology. This terminology meets comtemporary requirements on formality and accuracy. In this way, the systems can be described within one framework, making a comparison between the systems possible.

We chronologically follow the development of type theory, starting with Frege (1879). Some important, though less-known type systems are studied. They are described in a modern terminology without violating the original philosophy behind them. This results in a modern and historically correct description of the various systems. We also discuss some important developments in type theory and their influence on modern type theory.

The most important basics of current type theory, functional abstraction and function application, can already be found in the theories of Frege. His notion of abstraction is incorporated in Bertrand Russell's Ramified Type Theory (RTT) (1908) which was constructed as a solution for the logical paradoxes that arose at the turn of the century. The thesis provides a formalisation of RTT. It appears that the notion of function application is only introduced at an informal level in the original system. Using techniques of λ-calculus we give an accurate definition of function application as is present in RTT.

RTT has two hierarchies: one of types and one of orders. Ramsey (1926) shows that the logical paradoxes can also be avoided when using a simple type theory, without a hierarchy of orders. However, the thesis shows that orders still play an important role in logic. It describes a close relation

between the orders of RTT and the truth levels in Kripke's Theory of Truth (1975). Kripke's truth levels can be seen as a semantical interpretation of the notion of order in RTT.

After having translated RTT in modern terminology, we describe RTT in so-called "propositions-as-types" style. This gives RTT a position in the framework of Pure Type Systems (PTSs), in which many modern type systems have already been classified.

The type theory at the basis of the proof checker AUTOMATH has already been described before in the PTS framework. However, no attention has been paid to the definition and parameter mechanisms, which play prominent roles in AUTOMATH. The thesis gives a detailed description of AUTOMATH. Then we extend the framework of PTSs with a parameter mechanism. This mechanism is constructed in such a way that it can be combined with the extension of PTSs with definitions as described by Severi and Poll. In the refined framework, PTSs with definitions and parameters, we not only classify various AUTOMATH systems, but also other important type systems, like LF and ML.

Samenvatting

Het proefschrift beoogt inzicht te geven in de ontwikkeling van het begrip type in logica en wiskunde in de afgelopen honderd-twintig jaar. Hierbij wordt nadrukkelijk meer gedaan dan (een vorm van) geschiedschrijving. Type-systemen die in deze periode zijn ontwikkeld worden dan ook niet alleen beschreven, ze worden ook vertaald naar een moderne terminologie, die voldoet aan de eisen die heden ten dage aan formele type-systemen gesteld worden. Daardoor kunnen deze systemen binnen een en hetzelfde kader worden geplaatst, zodat een onderlinge vergelijking tussen deze systemen mogelijk wordt.

Het proefschrift volgt, chronologisch, de ontwikkeling van de type-theorie sinds Frege (1879). Een aantal belangrijke, doch minder bekende type-systemen wordt bestudeerd. Deze systemen worden beschreven in een tegenwoordig gebruikelijke terminologitegenwoordig gebruikelijke terminologie, zonder dat de oorspronkelijke filosofie achter het systeem geweld wordt aangedaan. Hierdoor wordt een moderne, maar historisch verantwoorde beschrijving van de diverse type-systemen gegeven. Ook een aantal belangrijke ontwikkelingen binnen de type-theorie wordt beschreven en hun invloed op de hedendaagse type-theorie wordt besproken.

Het blijkt dat abstractie (een van de belangrijkste pijlers van de moderne type-theorie, functie-abstractie en functie-applicatie) reeds te vinden is in de theorie van Frege. Het abstractie-begrip van Frege wordt overgenomen door Bertrand Russell in diens Vertakte Type-theorie (VTT) (1908), die ontstaat als reactie op de logische paradoxen die rond de eeuwwisseling ontdekt werden. Het proefschrift geeft een formalisering van VTT. Met name het begrip functie-applicatie blijkt in het oorspronkelijke systeem slechts op informeel niveau aanwezig te zijn. Met behulp van technieken uit de moderne λ-calculus kan een accurate formulering gegeven worden

van de functie-applicatie zoals die aanwezig is in VTT.

VTT bestaat uit twee hiërarchieën: één van types en één van ordes. Door Ramsey (1926) wordt aangetoond dat de logische paradoxen ook vermeden kunnen worden met een enkelvoudige type-theorie, waarin geen hiërarchie van ordes zit. Het proefschrift laat echter zien dat het begrip orde nog steeds een belangrijke plaats inneemt in de logica. Het proefschrift legt een nauwkeurig verband tussen de ordes uit VTT en de waarheidsniveaus in Kripkes Theory of Truth (1975). Het blijkt dat Kripkes waarheidsniveaus kunnen worden gezien als een semantische interpretatie van het orde-begrip uit VTT.

Behalve de vertaling van VTT in moderne terminologie beschrijft het proefschrift VTT ook in zogenaamde "propositions-as-types"-stijl. Hierdoor krijgt VTT een plaats binnen het raamwerk van "Pure Type Systems" (PTSs), een framework waarbinnen reeds vele moderne type-systemen zijn geclassificeerd.

De type-theorie die ten grondslag ligt aan de proof checker AUTOMATH is reeds eerder geplaatst in het raamwerk der PTSs, maar daarbij is geen aandacht geschonken aan het definitie-mechanisme en het parameter-mechanisme, die in AUTOMATH prominent aanwezig zijn. In het proefschrift wordt eerst een nauwkeurige beschrijving van AUTOMATH gegeven. Daarna wordt het raamwerk van PTSs uitgebreid met een parameter-mechanisme. Dit raamwerk is zo opgesteld, dat de uitbreiding gecombineerd kan worden met de uitbreiding van PTSs met definities zoals omschreven door Severi en Poll. In het fijnere raamwerk dat zo verkregen wordt, PTSs met definities en parameters, worden niet alleen de verschillende AUTOMATH-systemen geclassificeerd. Ook andere belangrijke type-systemen, zoals de systemen die ten grondslag liggen aan LF en ML, kunnen met dit fijnere raamwerk nauwkeuriger beschreven worden.

Curriculum Vitae

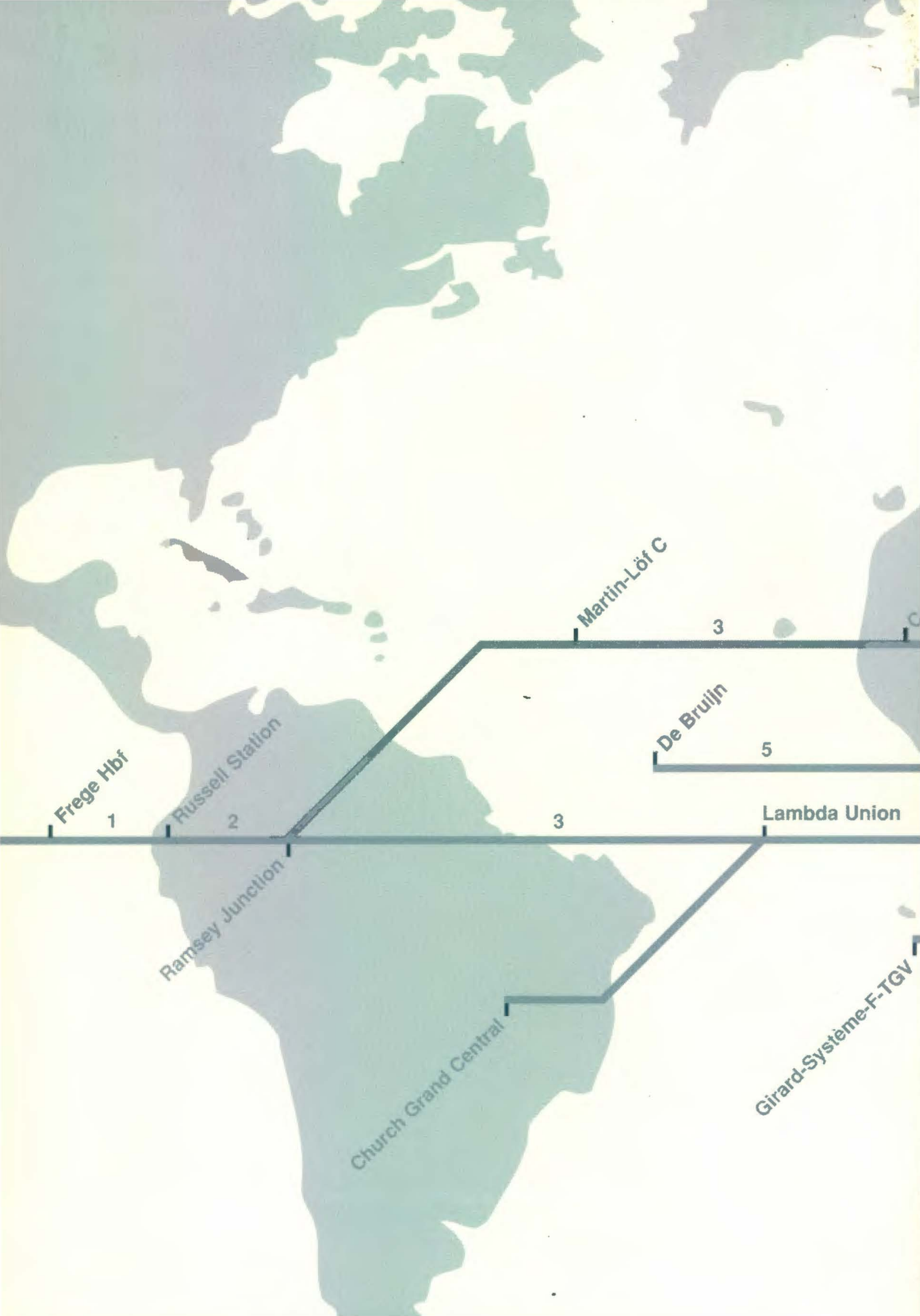
5 February 1970
Born in Etten-Leur

1982–1988
Secondary school: VWO, Macropedius College, Gemert

1988–1993
Master's degree (cum laude) in mathematics,
Catholic University of Nijmegen.

1993–1997
PhD, Eindhoven University of Technology and Tilburg University.
Supervisors: dr. R.P. Nederpelt and prof. dr. H.C.M. de Swart.

Titles in the IPA Dissertation Series


Martin-Löf C
3
De Bruijn
5
Frege Hbf
1
Russell Station
2
3
Lambda Union
Ramsey Junction
Church Grand Central
Girard-Système-F-TGV