Technical Report

UCAM-CL-TR-232
ISSN 1476-2986

Number 232

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Mechanising set theory

Francisco Corella

July 1991

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted June 1989 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Corpus Christi College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Set theory is today the standard foundation of mathematics, but most proof development systems (PDS) are based on type theory rather than set theory. This is due in part to the difficulty of reducing the rich mathematical vocabulary to the economical vocabulary of set theory. It is known how to do this in principle, but traditional explanations of mathematical notations in set theoretic terms do not lend themselves easily to mechanical treatment.

We advocate the representation of mathematical notations in a formal system consisting of the axioms of any version of ordinary set theory, such as ZF, but within the framework of higher-order logic with λ-conversion (H.O.L.) rather than first-order logic (F.O.L.). In this system each notation can be represented by a constant, which has a higher-order type when the notation binds variables. The meaning of the notation is given by an axiom which defines the representing constant, and the correspondence between the ordinary syntax of the notation and its representation in the formal language is specified by a rewrite rule. The collection of rewrite rules comprises a rewriting system of a kind which is computationally well behaved.

The formal system is justified by the fact that set theory within H.O.L. is a conservative extension of set theory within F.O.L. Besides facilitating the representation of notations, the formal system is of interest because it permits the use of mathematical methods which do not seem to be available in set theory within F.O.L.

A PDS, called Watson, has been built to demonstrate this approach to the mechanization of mathematics. Watson embodies a methodology for interactive proof which provides both flexibility of use and a relative guarantee of correctness. Results and proofs can be saved, and can be perused and modified with an ordinary text editor. The user can specify his own notations as rewrite rules and adapt the mix of notations to suit the problem at hand; it is easy to switch from one set of notations to another. As a case study, Watson has been used to prove the correctness of a latch implemented as two cross-coupled nor-gates, with an axiomatization of time as a continuum.

Preface

This report is a revised version of my PhD dissertation. The main purpose of this revision is to correct errors in the material concerning rewriting systems. The proofs of theorems F.5 and F.9 were incorrect, theorem F.5, corollary F.6 and theorem F.9 were stated in overly general terms, and some of the terminology was in conflict with standard terminogy in the field of term rewriting systems. I have completely rewritten the material of appendix F, and incorporated it in chapter 3. I have also rewritten and abbreviated section 5.2.3 on the foundations of category theory in the concluding chapter, and I have incorporated the material of appendix E into chapter 2.

I am grateful to the examiners, Joseph Goguen and Andrew Pitts, for correcting errors in the original version of the dissertation, which was sumbitted in June 1989, and making many useful suggestions. These corrections and suggestions were incorporated in the official version of the dissertation, which was accepted in March 1990 and can be found at library of the University of Cambridge.

I am grateful to Nachum Dershowitz for a very helpful discussion on term rewriting systems, and for pointing out to me an article by Xubo Zhang [53] which contains a counterexample to theorem F.5 and corollary F.6 of the dissertation.

IBM Research
P.O.Box 704, Yorktown Heights NY 10598, USA

July 1991 Francisco Corella

Preface of the dissertation

I am deeply indebted to Mike Gordon for his guidance and his encouragement, which started even before I came to Cambridge as a graduate student. I am grateful for his prompt reading and helpful criticism of drafts of the thesis and preliminary reports. The main inspiration for this work has been Mike Gordon's HOL system and the documents describing it, in particular [20]. The idea that set theoretic variable-binding constructs can be reduced to λ-abstraction and higher-order constants in H.O.L. can be found in [21].

In Cambridge I have benefited from many discussions with the members of the Hardware Verification group, other members of the Computer Laboratory, and visitors, in particular with Avra Cohn, Thierry Coquand, Thomas Forster, John Herbert, Jeff Joyce, Tom Melham, Larry Paulson and Glynn Winskel. The observation that set theory within H.O.L. is a conservative extension of set theory within F.O.L. originated in discussions with Thierry Coquand about the foundations of category theory. Glynn Winskel suggested the use of a description operator rather than the selection operator found both in the HOL system and in Bourbaki's set theory.

Part of this research has been done at Schlumberger Palo Alto Research and at the IBM T. J. Watson Research Center. I am grateful to Schlumberger and IBM for their financial support and for providing a stimulating research environment.

At IBM I have benefited from discussions with various members of the Artifical Intelligence group and other research groups, and in particular with my IBM supervisor, Sanjaya Addanki, and with Ben Grosof and Norman Haas. Norman Haas made useful comments on a draft of a preliminary report. During a visit to IBM, Stuart Russell helped me discover a misconception which I had regarding the pseudo-binding method for the formalization of mathematical notations.

At Schlumberger I benefited from discussions with Harry Barrow, Dan Carnese, Pat Hayes, and others. During a visit to Schlumberger, Gordon Plotkin strongly argued in favor of using λ-abstraction as unique variable-binding mechanism.

Otherwise this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

Corpus Christi College
Cambridge

June 1989 F. C.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

It may be fair to say that set theory is today the standard formal system for the foundation of mathematics. One would therefore expect most attempts at mechanizing mathematics to be based on set theory. This is not the case, however. Proof development systems (PDSs—also known as interactive theorem provers) have so far mostly been based on variants of type theory, rather than set theory.[1] This is the case, among others, of HOL [23], TPS [4, 6], Veritas [26], EKL [33], and Nuprl [12].

Type theory and set theory, which originate respectively in Russell's and Zermelo's systems of 1904, are competing alternatives, both having advantages and disadvantages. The advantages of type theory have been pointed out by some of the authors of PDSs based on type theory [5, Preface], [22]; and the choice of type theory for any particular PDS is not at all surprising. What is surprising is that the favor enjoyed by set theory as a foundational system is lost when it comes to implementing a PDS. This suggests that there may be some feature of set theory which causes no difficulty for ordinary mathematical practice but becomes an obstacle to mechanization.

Actually it is not the mechanization of set theory itself which presents a

[1] The system Ontic is presented as being based on ZFC, but it is not clear how the six syntactic categories and twenty three syntactic constructs of Ontic [39, pages 194-196] reduce to the syntax of ordinary, F.O. set theory. The language of Ontic has in fact types, type generators and λ-abstraction, so it is closer to type theory with λ-conversion than to Zermelo's set theory.

difficulty, but the mechanization of mathematics with a set theoretic foundation. Set theory is just a particular first-order theory, and general purpose PDSs have been successfully put to the task of proving theorems in set theory. For example, *Isabelle* [43, 44], an interactive prover designed to support a large class of logics, has been applied to set theory [42]. Also, it has been observed in the automatic theorem proving community that variants of set theory which distinguish sets from classes (often referred to as Gödel-Bernays set theory, [19, 8, 40]) have finitely many axioms, and so existing automatic provers can be used to search for proofs of theorems in such theories. What is difficult is to go from the realm of set theory per se, i.e. from proving propositions which can easily be expressed using only the set-membership predicate symbol $\in$, into the wider realm of mathematics.

It is not that proofs in "mathematics" are harder than proofs in "set theory," and the difficulty is not due to any limitation in the size of the inference steps that mechanical provers can take. Much progress has been made in automatic theorem proving, and an interactive prover could use the full power of an automatic prover to justify each of the steps of an interactive proof; those steps could then be "larger" than the steps of a proof done by hand.

What is then the problem? As it has already been observed by Andrews [5, Preface], the problem is one of *vocabulary.* There is a considerable gap between the extremely economical vocabulary of set theory (a single binary predicate symbol in many versions of set theory) and the very rich vocabulary of mathematics. In informal mathematics, the gap is bridged by *informal mathematical notations.* Mathematical notations are essential: without them, one cannot take off the ground, one cannot move from the realm of set theory into the realm of mathematics. The problem is how to use these notations in a mechanical theorem prover.

Logicians have proposed several competing ways of explaining mathematical notations in terms of first-order set theory. However, these explanations were not conceived with the idea of mechanizing set theory. As we shall see, trying to base the implementation of a PDS on any of them would present considerable practical difficulties. Instead, we propose a new method of formalizing mathematical notations, within a formal system which is a conservative extension of ordinary set theory. The proposed method lends itself readily to mechanization.

1.2 Overview

In the remainder of this introduction we shall first consider, in section 1.3, whether it would be possible to take mathematical notations at face value, i.e. to formalize them without explaining them in terms of any more primitive notations. In this connection we shall briefly discuss the specification language Z. Then, in sections 1.4 and 1.5, we shall review some of the ways of explaining mathematical notations which have been proposed to date. We shall point out their practical shortcomings from the point of view of the implementation of a PDS, but at the same time we shall gather useful ideas which will lead us to the formalization which we are proposing.

The new formalization requires a formal system consisting of Church's higher-order logic (H.O.L.) [10] together with the axioms of ordinary first-order set theory. Chapter 2 is dedicated to the study of that formal system. First, a natural deduction formulation of Church's H.O.L. is given in section 2.3. Then in section 2.4 we prove the fact that any first-order theory (e.g. Zermelo-Fränkel set theory) developed within H.O.L. is a conservative extension of the same theory developed within first-order logic (F.O.L.) This simple result provides the philosophical justification of our approach to the mechanization of set theory; and as suggested in the conclusion, it may also lead to other developments in the foundation and mechanization of mathematics. In section 2.5 we describe our treatment of mathematical notations in the formal system. We show that notations are eliminable, we give the axioms of ZF and the basic notations associated with them, and we discuss which formulas are acceptable as parameters of the axiom schema of replacement.

In the proposed formalization, mathematical notations are considered as shorthands for expressions of the formal system. A PDS for set theory can then use the shorthands for input and output, while internally it uses the corresponding expressions of the formal system. Chapter 3 shows how the shorthands can be specified by rewrite rules and how the translation between surface form and internal representation can be accomplished by a rewriting system of a kind which is particularly well-behaved.

The proposed approach to the mechanization of set theory has been tested by building a prototype PDS, called Watson, and carrying out with it a case study in hardware verification. This is the topic of chapter 4. Section 4.1 describes the language processor of Watson. Section 4.2 then describes a

methodology for interactive proof which has been especially developed for Watson. The methodology borrows ideas from several other PDSs and it features the possibility of editing proofs under construction and libraries of results using an ordinary text editor such as Emacs. The case study is the proof of correctness of a latch with an axiomatization of time as a continuum, something which does not seem to have been done before in the field of Hardware Verification. It is described in section 4.3.

The concluding chapter, after a recapitulation, points out opportunities provided by the formal system in several areas, in particular regarding the foundations of category theory.

1.3 Taking mathematical notations at face value

The problem of bridging the gap between set theory and ordinary mathematical notations could be avoided by "formalizing" mathematical notations directly, without reducing them to axiomatic set theory. In some sense this is what is done in computer algebra systems, and also in the specification language Z [51].

Z must be mentioned here for two reasons: (i) it is part of the field of "formal methods," and (ii) its appeal is partly due to the fact that it is explicitly based on set theory (although not on *axiomatic* set theory).

However, it is not appropriate to compare Z and Watson. While Watson is a *computer tool* for developing *proofs*, Z is a *language* for developing *specifications*. The goal of Z is to facilitate joint development of a specification by a team of engineers; and this is achieved by imposing a discipline by means of a common notation. In contrast, a feature of Watson is that each user is free to design his own notations—while still being able to share results and even proofs with users who prefer different notations; this is the topic of chapter 3.

Computer algebra systems perform computations which a mathematician may need during the course of a proof, but they do not deal with the totality of a mathematical argument, and have no notion of proof within themselves. Therefore it is not necessary to relate rigorously the notations used in a computer algebra system to axiomatic set theory.

A PDS, on the other hand, deals with proofs, and therefore requires a formal system of logic. If we choose to use axiomatic set theory, we must then explain mathematical notations in terms of axiomatic set theory. To this end, we shall begin by reviewing the most common traditional explanations.

1.4 Axiomatic vs. metalinguistic definitions

There are two ways of introducing mathematical notations: by *axiomatic definition* or by *metalinguistic definition.*

In the first case, a new syntactic construct is added to the object language, and a new axiom is added to the inference system as the definition of the new construct. For example, the unordered pair notation can be introduced as a construct which builds a term "$\{\boldsymbol{A}, \boldsymbol{B}\}$" out of two terms $\boldsymbol{A}$ and $\boldsymbol{B}$, with the defining axiom:

$$\forall x \forall y \forall z (z \in \{x, y\} \equiv z = x \vee z = y)$$

In the second case, the object language is not modified at all. Instead, a convention is introduced in the metalanguage by which a certain (parameterized) metalinguistic expression refers to a certain object language expression. For example, to say that, for every pair of terms $\boldsymbol{A}$, $\boldsymbol{B}$ the expression "$\boldsymbol{A} \subseteq \boldsymbol{B}$" stands for the sentence

$$\forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{A} \supset \boldsymbol{x} \in \boldsymbol{B})$$

(where $\boldsymbol{x}$ is a variable which is not free in $\boldsymbol{A}$ or $\boldsymbol{B}$) would be a metalinguistic definition of the subset notation. ($\boldsymbol{A}$ and $\boldsymbol{B}$ are the *parameters* of the notation.)

When notations are introduced by axiomatic definition, one must ensure that they are *eliminable from proofs.* That is, if a result is proved using the notations and their defining axioms, but the result itself does not contain any of the notations, it should be also provable without making use of the notations. In other words, the system obtained by introduction of the notations should be a *conservative* extension of the original formal system.[2] It is also desirable that notations be *eliminable from formulas*, i.e. that any sentence containing notations be logically equivalent (in the extended formal system)

[2]Suppes calls this the "criterion of non-creativity."[52, §2.1]

to a sentence which is free of them.[3] Why notations should be eliminable from proofs is clear: we do not want them to strengthen the system, and in particular we do not want them to introduce a contradiction. Why they should be eliminable from formulas is less clear. In fact, in formal systems other than set theory (e.g. in type theory), the requirement of eliminability from formulas may be superflous, and undesirable.[4] However, in set theory there is a powerful motivation for this requirement, as we shall see in section 2.5.

A simple way of introducing notations by axiomatic definition is to enlarge the vocabulary of a F.O. theory with additional n-ary function symbols (or constants as the special case where $n = 0$). For example, for the pair-set notation we could use a binary function symbol "enum": "enum $\boldsymbol{A}\,\boldsymbol{B}$" would denote the unordered pair $\boldsymbol{A}$, $\boldsymbol{B}$. It is well known [40, §9] [34, §74] that, if x, $y_1 \ldots y_n$ are the pairwise distinct variables occurring free in a sentence $\boldsymbol{P}$, and

$$\forall y_1 \ldots \forall y_n \exists! x \boldsymbol{P}$$

is derivable in a F.O. theory, then the definition of a new function symbol $\boldsymbol{f}$ by the axiom:

$$\forall y_1 \ldots \forall y_n \boldsymbol{P}^x_{f\, y_1 \ldots y_n}$$

(where we assume that "$\boldsymbol{f}\, y_1 \ldots y_n$" is free for x in $\boldsymbol{P}$, and where $\boldsymbol{P}^x_{f\, y_1 \ldots y_n}$ is the result of substituting "$\boldsymbol{f}\, y_1 \ldots y_n$" for the free occurrences of x in $\boldsymbol{P}$) satisfies both eliminability requirements. In the unordered pair example the axiom would be

$$\forall y \forall z \forall u (u \in (\text{enum}\, y\, z) \equiv u = y \vee u = z)$$

and eliminability would be guaranteed by the fact that

$$\forall y \forall z \exists! x \forall u (u \in x \equiv u = y \vee u = z)$$

is derivable in a theory of sets with the axiom of extensionality and the pair-set axiom: existence follows from the latter, uniqueness from the former.

Even when an additional function symbol, e.g. "enum", is used, the traditional notation, e.g. "$\{\boldsymbol{A}, \boldsymbol{B}\}$", is usually still retained, as standing

[3] Suppes' "criterion of eliminability."[ibid.]

[4] ... undesirable because, as Suppes [ibid.] points out, many definitions are most naturally formulated as *conditional* definitions, and as such are not eliminable from formulas.

for the function-symbol notation. This is again a metalinguistic definition: "$\{\boldsymbol{A}, \boldsymbol{B}\}$" stands for "enum $\boldsymbol{A}$ $\boldsymbol{B}$" in the same sense as "$\boldsymbol{A} \subseteq \boldsymbol{B}$" stands for "$\forall x(x \in \boldsymbol{A} \supset x \in \boldsymbol{B})$". But there is a major difference: while going from "$\{\boldsymbol{A}, \boldsymbol{B}\}$" to "enum $\boldsymbol{A}$ $\boldsymbol{B}$" is just a change of notational style, going from "$\boldsymbol{A} \subseteq \boldsymbol{B}$" to "$\forall x(x \in \boldsymbol{A} \supset x \in \boldsymbol{B})$" involves expanding the definition of the notion of subset. The latter could constitute a step of a proof done by hand, but not the former.

The difference between those two kinds of metalinguistic definitions becomes even clearer if we consider the possibility of introducing predicate symbols in the same manner as function symbols, something not usually done by logicians. Then we would have the choice between considering that "$\boldsymbol{A} \subseteq \boldsymbol{B}$" stands for

$$\forall x(x \in \boldsymbol{A} \supset x \in \boldsymbol{B}) \tag{1.1}$$

and considering that it stands for

$$\text{subset } \boldsymbol{A}\ \boldsymbol{B} \tag{1.2}$$

the predicate symbol "subset" being defined by

$$\forall a \forall b(\text{subset } a\ b \equiv \forall x(x \in a \supset x \in b))$$

Clearly in the second case it is the axiom defining the predicate symbol which lends its substance to the notation "$\boldsymbol{A} \subseteq \boldsymbol{B}$". So in such case we shall still say that the notation is defined axiomatically.

Assume now that we want to implement a PDS according to some textbook of set theory. Assume the author of the textbook uses "$\boldsymbol{A} \subseteq \boldsymbol{B}$" as a shorthand for either (1.1) or (1.2). With respect to the textbook, "$\boldsymbol{A} \subseteq \boldsymbol{B}$" is not part of the object language; it is a metalinguisitic shorthand, i.e. an abbreviation which the author uses to simplify his text, just as he may write "wff" for "well-formed formula." From the point of view of the theorem prover, however, "$\boldsymbol{A} \subseteq \boldsymbol{B}$" (for particular instances of $\boldsymbol{A}$ and $\boldsymbol{B}$) is something that the user types in and that the PDS prints out. It is a shorthand for either (1.1) or (1.2), but it is also part of the language of the PDS as much as (1.1) or (1.2). In other words, from the point of view of the PDS, metalinguistic shorthands become object-language shorthands.

While shorthands should be used for interaction with the user, the inference component of the PDS must manipulate expressions of the formal

system. This means that the PDS must use the expansions of the shorthands internally. We shall refer to the language for interaction with the user as the surface language. The formal language is then the internal representation of the surface language.

When axiomatic definitions are used, the surface language is made out of shorthands which are simple stylistic variations with respect to the formal language, e.g. "$\boldsymbol{A} \subseteq \boldsymbol{B}$" for "subset $\boldsymbol{A}\ \boldsymbol{B}$". It is then easy to translate back and forth between the surface language and the formal language. In chapter (3) we shall see that this can be accomplished by a well-behaved rewriting system. On the other hand, when metalinguistic definitions are used, translation becomes difficult or impossible. A given formal language expression may be obtainable by shorthand expansion from multiple surface language expressions, and it may not be clear which of these to print out. In the other direction, expansion of shorthands is combinatorially explosive; this is because a parameter of the notation may occur multiple times in the definiens; the length of the expansion is exponential in the depth of nesting of definitions where this happens, which is itself unbounded.

From the point of view of the implementation of a PDS, then, axiomatic definitions are preferable to metalinguistic ones. This is the opposite of what logicians generally prefer. Typically logicians do not introduce additional predicate symbols such as "subset", because sentence constructors such as "$\boldsymbol{A} \subseteq \boldsymbol{B}$" can be defined metalinguistically. They introduce function symbols such as "enum" only because there are no term constructors in ordinary set theory, and so a notation such as "$\{\boldsymbol{A}, \boldsymbol{B}\}$" cannot be defined metalinguistically in a straightforward way, since there is nothing that it can stand for.[5] For our purposes, on the contrary, we are quite content with additional function symbols and we shall make use of additional predicate symbols as well. Unfortunately, these expedients do not cover all mathematical notations.

1.5 Variable-binding term constructors

We have seen how some mathematical notations can be dealt with easily in a PDS by introducing additional constants, functions symbols, and predicate

[5]Some authors, e.g. Gödel [19], still manage to avoid an axiomatic definition in such cases by resorting to a contextual definition; an example of contextual definition will be given below.

symbols. All such notations have something in common: they do not bind variables. Unfortunately, there are many notations in mathematics which do bind variables. Some of them are *sentence constructors*, such as

$$(\exists x \in \boldsymbol{A})\boldsymbol{P}$$

and generalized quantifiers such as those discussed in section 5.2.2. They can be defined metalinguistically, but metalinguistic definitions are undesirable from the point of view of the implementation of a PDS, as we have seen.

Worse yet, many mathematical notations are *term constructors which bind variables.* Consider for example:

- "$\{x \in \boldsymbol{A} \mid \boldsymbol{P}\}$"—occurrences of x in the sentence $\boldsymbol{P}$ are bound by the notation.
- "$\bigcup_{x \in A} \boldsymbol{B}$"—occurrences of x in the term $\boldsymbol{B}$ are bound by the notation.
- "$\sum_{i=A}^{B} \boldsymbol{C}$"—occurrences of i in the term $\boldsymbol{C}$ are bound by the notation.

It is not clear how to explain these notations even if we have recourse to metalinguistic definitions, since ordinary F.O. set theory has no term constructors, let alone term constructors which bind variables.

We shall now examine four methods which have been used to explain such notations, and point out the difficulties that they raise for the implementation of a PDS.

1.5.1 The pseudo-binding method

A common method of explanation [40, 32] relies again on the conservative introduction of additional function symbols defined axiomatically. Now, however, a different function symbol $\boldsymbol{f}$ is used for each *instance* of the notation. (By an instance of a notation we mean the result of giving particular values to the parameters of the notation.) The arity of the function symbol $\boldsymbol{f}$ is the number of distinct free variables occurring in the instance of the notation, and if those free variables are $y_1 \ldots y_n$, the instance is supposed to stand for "$\boldsymbol{f}\ y_1 \ldots y_n$".

For example, the term "$\{x \in y \mid z \in x\}$" is an instance of the notation "$\{x \in \boldsymbol{A} \mid \boldsymbol{P}\}$", with parameter values $x =$ "x", $\boldsymbol{A} =$ "y" and $\boldsymbol{P} =$ "$z \in x$". Let us use "f" as the function symbol. There are two free variables in the

term, "y" and "z", so the arity of "f" is 2. The term stands for "f y z". The new function symbol "f" is in this case defined by the axiom:

$$\forall y \forall z \forall x (x \in \mathrm{f}\ y\ z \equiv x \in y \wedge z \in x)$$

Eliminability is ensured, in a set theory with extensionality and separation, by the theorem:

$$\forall y \forall z \exists! s \forall x (x \in s \equiv x \in y \wedge z \in x).$$

Notice the fate of the bound variable "x" in "f y z": it disappears. Thus, although the surface form suggests that a variable is being bound, no variable binding occurs in the corresponding formal expression. We shall refer to this method as the *pseudo-binding approach.*

The drawback of the pseudo-binding approach, from the point of view of mechanization, is that the surface form has a very different logical structure from the formal language expression for which it is supposed to stand. For example $\boldsymbol{P}$ is a subexpression of the surface form "$\{x \in \boldsymbol{A} \mid \boldsymbol{P}\}$", but not of the corresponding formal expression "$\boldsymbol{f}\ \boldsymbol{y}_1 \ldots \boldsymbol{y}_n$". Using the formal expression as internal representation for the surface form would then block a proof step consisting of a rewrite within $\boldsymbol{P}$ in "$\{x \in \boldsymbol{A} \mid \boldsymbol{P}\}$". Appendix A shows how this can force considerable detours in the course of a proof; detours, moreover, which it would be difficult to justify to the user.

1.5.2 Bourbaki's approach

Authors seeking explanations closer to actual mathematical practice must remedy the absence of variable-binding term constructors in ordinary set theory. Generally they introduce one or two such constructs and define other notations from them, either axiomatically or metalinguistically.

Bourbaki [9] uses as basic construct Hilbert's ε-operator.[6] Thus Bourbaki develops set theory in an extension of F.O. logic, the ε-calculus. A problem with Bourbaki's approach is that it builds-in the axiom of choice, even though the ε-calculus is a conservative extension of F.O. logic. We shall have more to say about this in section 2.5.5.

[6] He calls the operator τ; also, he dispenses with bound variables in his formal language, using the symbol $\square$ instead, and lines drawn from each occurrence of $\square$ to the occurrence of τ to which it refers.

The ε-construct, "$\varepsilon x \boldsymbol{P}$" denotes "some x such that $\boldsymbol{P}$ if there is any, or a completely unspecified x otherwise". It is indeed a term constructor, which binds the variable x in $\boldsymbol{P}$. Bourbaki defines other mathematical notations metalinguistically.[7] For example, Bourbaki defines "$\{x \mid \boldsymbol{P}\}$" as

$$\varepsilon y \forall x (x \in y \equiv \boldsymbol{P})$$

where y is a variable other than x and not free in $\boldsymbol{P}$.

Bourbaki's grand project, with its emphasis on carefully showing how actual mathematical practice relates to axiomatic set theory, is a valuable source of inspiration and ideas for an attempt at mechanizing mathematics. Mechanization, however, was not the goal of the project, and Bourbaki's approach has, from that point of view, the same drawbacks as any other approach based on metalinguistic definitions. It is remarkable that Bourbaki was aware of the combinatorial explosion that we mentioned in section 1.4, and curious enough to estimate the size of the expansion of the symbol "1", which stands in his system for "$Card(\{\emptyset\})$": the expansion would be tens of thousands of symbols long. The possibility of mechanizing their system may have been in the mind of some of the mathematicians who signed under the collective pseudonym *Nicolas Bourbaki*.

1.5.3 Quine's approach

In [48], Quine uses the construct "$\{x \mid \boldsymbol{P}\}$" (which he calls a *class abstract*) as basic construct, and, like Bourbaki, defines other mathematical notations metalinguistically, with the corresponding drawbacks for mechanization. It is interesting to note that "$\{x \mid \boldsymbol{P}\}$" is itself defined metalinguistically, by means of a contextual definition. In Quine's system set-membership is the only predicate symbol, and equality is not part of the object language (it is defined metalinguistically, as equiextensionality: "$\boldsymbol{A} = \boldsymbol{B}$" for "$\forall x (x \in \boldsymbol{A} \equiv x \in \boldsymbol{B})$"). So the construct can only appear in the following contexts:

$$y \in \{x \mid \boldsymbol{P}\} \tag{1.3}$$

or

$$\{x \mid \boldsymbol{P}\} \in y \tag{1.4}$$

[7] Including the existential and universal quantifiers.

The former expression is defined as $\boldsymbol{P}_y^x$ and the latter is in effect defined as

$$\exists z(\forall x(x \in z \equiv \boldsymbol{P}) \wedge z \in y)$$

The definition of (1.3) as $\boldsymbol{P}_y^x$ gives a meaning to (1.3) whether or not

$$\exists z \forall x(x \in z \equiv \boldsymbol{P}) \tag{1.5}$$

holds. To avoid this, one would define (1.3) in the same fashion as (1.4), as

$$\exists z(\forall x(x \in z \equiv \boldsymbol{P}) \wedge y \in z)$$

In Quine's system, however, class abstracts which do not necessarily satisfy (1.5) play a useful role in *schematic reasoning.*

1.5.4 Bernays' approach

Bernays [8] does not use metalinguistic definitions at all. As primitive variable-binding term constructors, he uses both "$\{x \mid \boldsymbol{P}\}$" and a description operator. Mathematical notations are incorporated into the language as additional constants, function symbols, predicate symbols, or "operators," i.e. variable-binding constructs, defined axiomatically.

This approach is, of those that we have surveyed, the one which lends itself most readily to mechanization. The only problem with it is its complication. There would have to be a large variety of constructs, not just in the surface language, but also in the formal language used as internal representation and manipulated by the inference component of the PDS. For example, to the surface construct "$\sum_{i=A}^{B} \boldsymbol{C}$" would correspond a term-construct with four parameters, the variable i and the terms $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$; the description of the constructor would have to specify that occurrences of i in $\boldsymbol{C}$ are allowed and bound by the construct, while occurrence of i in $\boldsymbol{A}$ or $\boldsymbol{B}$ are either disallowed or not bound by the construct. Other constructs would bind multiple variables. All this would complicate the task of defining notations, and the design of most of the components of the system, including the inference component.

Fortunately, this complication can be avoided, as we shall now see.

Chapter 2

A formal system for the representation of mathematical notations

2.1 Representing notations with higher-order constants

Consider a model $\mathcal{M}$ of set theory, and the notation "$\{\boldsymbol{A}, \boldsymbol{B}\}$". Assuming extensionality and pairing, for every pair of objects $(\mathcal{A}, \mathcal{B})$ in the model, there exists one element $\mathcal{C}$ of $\mathcal{M}$ which is a SET whose ELEMENTS are $\mathcal{A}$ and $\mathcal{B}$. (By SET and ELEMENT we refer to the denotations in $\mathcal{M}$ of the corresponding formal concepts of set theory.) Let $\mathcal{F}$ be the function from $\mathcal{M}^2$ to $\mathcal{M}$ which maps every $(\mathcal{A}, \mathcal{B})$ to the corresponding $\mathcal{C}$. If the terms $\boldsymbol{A}$ and $\boldsymbol{B}$ denote $\mathcal{A}$ and $\mathcal{B}$, then the notation "$\{\boldsymbol{A}, \boldsymbol{B}\}$" denotes $\mathcal{F}(\mathcal{A}, \mathcal{B})$. We have seen that the notation can be *formalized* by introducing an additional function symbol "enum" and considering that "$\{\boldsymbol{A}, \boldsymbol{B}\}$" is a shorthand for "enum$\boldsymbol{A}\,\boldsymbol{B}$". The function symbol then denotes the function $\mathcal{F}$.

Consider now the notation "$\{x \mid \boldsymbol{P}\}$". This is one of the "difficult" notations, since it binds a variable and constructs a term.

In the same manner as "$\{\boldsymbol{A}, \boldsymbol{B}\}$" is related to the pair-set axiom, the notation "$\{x \mid \boldsymbol{P}\}$" is related to the Cantorian comprehension axiom. In Cantorian set theory, the comprehension axiom stipulated that for every "property" of objects there is a SET (unique by extensionality) whose ELEMENTS

are those objects which satisfy the property. This axiom leads to Russell's paradox and had to be abandoned, but modern set theories have weaker axioms which stipulate that there is such a SET for *some* properties of objects. The notation "$\{x \mid P\}$" denotes the SET corresponding to "the property which P states of x," when this is one of such properties.

This suggests trying to formalize "$\{x \mid P\}$" as some kind of function symbol, that we could call "set", applied to some formula Φ denoting the said "property": "$\{x \mid P\}$" would be short for "set Φ". Model-theoretically, the denotation of Φ, i.e. the "property", would be the subset of $\mathcal{M}$ consisting of those objects $\mathcal{X}$ such that P denotes truth when x denotes $\mathcal{X}$; and "set" would denote a function mapping subsets of $\mathcal{M}$ to elements of $\mathcal{M}$.

The problem is of course that there are no formulas Φ denoting subsets of the model $\mathcal{M}$ in F.O.L. However, in Church's system of *Higher-Order Logic with λ-abstraction* there are such formulas. We can take $\Phi =$ "$\lambda x P$". So in Church's system we can formalize "$\{x \mid P\}$" as a shorthand for "set $(\lambda x P)$", or more precisely as "$\text{set}_{\iota(o\iota)}\ (\lambda x P)$". The type $\iota(o\iota)$ of $\text{set}_{\iota(o\iota)}$ indicates that $\text{set}_{\iota(o\iota)}$ denotes, as desired, a function mapping subsets of $\mathcal{M}$ to elements of $\mathcal{M}$; $\text{set}_{\iota(o\iota)}$ is a *higher-order constant.*[1]

As we saw in section 1.5.4, the drawback of Bernays' approach is the syntactic complication of having to specify and manipulate many different variable-binding constructs. We have just seen how this complication can be avoided in Church's system in the particular case of the notation "$\{x \mid P\}$": by considering it as a shorthand for "$\text{set}_{\iota(o\iota)}\ (\lambda x P)$" it suffices to introduce a constant to represent the notation, "$\text{set}_{\iota(o\iota)}$"; variable-binding is accomplished by the preexisting λ-abstraction construct.

Other variable-binding notations can be handled similarly:

"$\{x \in A \mid P\}$"	for	"$\text{subset}_{\iota(o\iota)\iota}\ A\ (\lambda x P)$"
"$\bigcup_{x \in A} B$"	for	"$\text{union}_{\iota(\iota\iota)\iota}\ A\ (\lambda x B)$"
"$\sum_{i=A}^{B} C$"	for	"$\text{sum}_{\iota(\iota\iota)\iota\iota}\ A\ B\ (\lambda i C)$"
"$\{A\}_{x \in B}$"	for	"$\text{range}_{\iota(\iota\iota)\iota}\ B\ (\lambda x A)$"
"$\{A\}_{x \in B, y \in C}$"	for	"$\text{range}_{\iota(\iota\iota\iota)\iota\iota}\ B\ C\ (\lambda x \lambda y A)$"
"$(\forall x \in A) P$"	for	"$\text{forall}_{o(o\iota)\iota}\ A\ (\lambda x P)$"

[1] As we shall see in section 2.3.1, o denotes the type of the two truth values, ι denotes the type of individuals, and $(\alpha\beta)$ denotes the type of functions from the denotation of the type β to the denotation of the type α.

In contrast with Bernays' approach, it is not necessary to specify that the notation "$\{x \in \boldsymbol{A} \mid \boldsymbol{P}\}$" binds x in $\boldsymbol{P}$, that "$\{\boldsymbol{A}\}_{x \in B, y \in C}$" binds x and y in $\boldsymbol{A}$, and so on, since this is implied by the formal counterparts of the notations, "$\mathrm{subset}_{\iota(o\iota)\iota}\ \boldsymbol{A}\ (\lambda x \boldsymbol{P})$", "$\mathrm{range}_{\iota(\iota\iota\iota)\iota\iota}\ \boldsymbol{B}\ \boldsymbol{C}\ (\lambda x \lambda y \boldsymbol{A})$", and so on.

2.2 The double nature of type theory

The problem of formalizing mathematical notations has led us to consider a formal system consisting of set theory within Church's higher-order logic. In the rest of the chapter we are going to study this formal system and describe more precisely the method of handling mathematical notations that we have just sketched out.

But the formal system is unusual. It seems that it has not been used or studied before, besides a brief mention in Henkin's thesis [27, pages 64–67]. So it may be useful to pause now and try to put it in perspective. This section assumes some familiarity with Church's system, thus it anticipates section 2.3; but it can be skipped without loss of continuity.

We have referred to Church's system [10] as H.O.L. (as most authors refer to it nowadays), but Church himself refers to it as type theory. So, after declaring the intention of breaking away with the tradition of using type theory in PDSs, it seems that we have been thrown back to it. And indeed, Church's system is the formal system used in TPS [6], while the HOL prover [23] uses a formal system which is essentially a polymorphic extension of it. Our approach, however, uses Church's system in an essentially different way, as we shall now show.

When the paradoxes showed that Cantorian set theory was contradictory, two ways of overcoming the difficulty were devised. Russell restricted the set-membership construct, by assigning types to variables and ruling out, syntactically, instances of the construct whose parameters were not of the appropriate types. Thus Russell's type theory, at least as later simplified, is a many-sorted set theory. The addition of Church's λ-abstraction construct to the simplified type theory resulted in the system of [10]. Zermelo, on the other hand, restricted the axiom schema of comprehension; he replaced it with a collection of weaker axioms and axioms schemas.[2] Zermelo thus

[2]Namely empty-set, pair-set, union, power-set and separation—replacement was later added by Fränkel.

preserved the syntactic simplicity and much of the flexibility in constructing sets of Cantorian set theory.

Remarkably, besides being a many-sorted set theory, type theory is also an extension of F.O.L.—hence its other name, "H.O.L.". This has not always been understood,[3] but it is true, not only in a debatable philosophical sense, but in a precise metamathematical sense. Indeed, the theory of types admits two model-theoretic interpretations: one, as a many-sorted set theory; the other, as an extension of F.O.L. The latter corresponds to Henkin's standard models, the former to Henkin's general models.[4]

In our approach, we make use of the fact that type theory is an extension of F.O.L. to introduce a second notion of set-membership in Church's type theory. Both notions then coexist. The first one is exrpessed by:

$$\boldsymbol{S}_{o\alpha}\, \boldsymbol{M}_{\alpha}$$

The member $\boldsymbol{M}_{\alpha}$ is of arbitrary type α, the set $\boldsymbol{S}_{o\alpha}$ is of type $o\alpha$. The second one is expressed by:

$$\mathrm{in}_{o\iota\iota}\, \boldsymbol{M}_{\iota}\, \boldsymbol{S}_{\iota}$$

where $\mathrm{in}_{o\iota\iota}$ is a constant. Set and member are both of type ι. In pre-λ simplified type theory "$\boldsymbol{S}_{o\alpha}\, \boldsymbol{M}_{\alpha}$" would have been written "$\boldsymbol{M}_{\alpha} \in \boldsymbol{S}_{o\alpha}$"; but to avoid confusion we shall reserve the use of the symbol "$\in$" to the second notion of set-membership: "$\boldsymbol{M}_{\iota} \in \boldsymbol{S}_{\iota}$" for "$\mathrm{in}_{o\iota\iota}\, \boldsymbol{M}_{\iota}\, \boldsymbol{S}_{\iota}$".

We use the second notion of set membership to formalize the usual notion of set membership in mathematics, while other PDSs such as HOL or TPS, use the first one. Thus, even though TPS, HOL and Watson all make use of type theory, TPS and HOL are truly based on type theory, in the sense that they use a type-theoretic formalization of mathematics, while Watson is based on set theory, in the sense that it uses a set-theoretic formalization. In TPS and HOL, Church's system plays the role of type theory, while in Watson it plays the role of H.O.L.[5] Watson uses Church's system as a richer

[3] Quine [48, pages 257–258] dismisses the view of type theory as H.O.L. as the result of a confusion between schematic (metalinguistic) variables and genuine (object-language) variables.

[4] More precisely, every model of type theory considered as a many-sorted set theory is isomorphic to a general model. From this it follows that completeness for Henkin's general models follows directly from the completeness of many-sorted first-order logic. For the details, see [13].

[5] So, priority apart, Watson would have a stronger claim to the name HOL!

framework than F.O.L. in which to develop F.O. set theory. The fact that H.O.L. is a conservative extension of F.O.L., as we shall see below, means that there is no reason not to do so.

Practically, using the second notion of set-membership provides the set-forming flexibility of set theory, to be contrasted with the rigidity of a type hierarchy.

2.3 Natural deduction formulation of H.O.L.

In this section we describe Church's system of H.O.L., with standard-model semantics and a natural deduction inference system.

2.3.1 The type hierarchy

We define simultaneously by induction the *type expressions*, more simply called *types*, and their denotations:

- "o" is a type expression. It denotes a set $\{\mathrm{F}, \mathrm{T}\}$ of two elements which we shall use as truth-values, T as truth and F as falsity.
- "ι" is a type expression. It denotes an arbitrary non-empty set M. The elements of M are called *individuals*, and M is called the *domain of individuals*.
- If α is a type expression denoting a set A and β a type expression denoting a set B, then "$(\alpha\beta)$" is a type expression denoting the set of functions from B to A.

The types o and ι are the *atomic* types, while the types $(\alpha\beta)$ are the *functional* types. (As we have already done in the definition, we shall use the Greek letters α, β ..., not including of course o and ι, as metalinguistic variables denoting type expressions.)

The denotation M of "ι" determines the denotation $\mathcal{D}_\alpha$ of each type expression α. We shall refer to the family $\mathcal{D}$ of those type denotations as the *(standard) frame* generated by M. (General models make use of "non-standard frames"; since we shall not deal with general models, we shall say "frame" for "standard frame".)

When writing type expressions, parentheses are suppressed with association to the left.

The *order* of a type is defined inductively as follows:

1. The order of ι is 0.
2. The order of o is 1.
3. The order of $\alpha\beta$ is the maximum of (i) the order of α, and (ii) the order of β plus 1.

Every type can be written in a unique way $\delta\alpha_1 \ldots \alpha_n$, where δ is an atomic type. We then say that n is the *arity* of the type. It is easy to see that the order of $\delta\alpha_1 \ldots \alpha_n$ when $n > 0$ is 1 plus the maximum of the orders of α_1, $\ldots$ α_n.

2.3.2 The typed λ-language

For each type α there is a denumerable set of *proper symbols* of type α. This set is partitioned into two subsets, each denumerable, the *constants* of type α and the *variables* of type α.[6] As constants we shall use the identifers in roman font, with the type indicated as a subscript, e.g.:

$$\mathrm{c}_\iota,\ \mathrm{c1}_\iota,\ \mathrm{and}_{ooo},\ \mathrm{subset}_{o\iota\iota},\ \mathrm{subset}_{\iota(o\iota)}.$$

As variables we shall use the identifiers in italic font, again with the type indicated as a subscript, e.g.:

$$x_\iota,\ x1_\iota,\ p_o,\ abc_{o\iota\iota}.$$

There is also an improper symbol, "λ", and parentheses are used for grouping subexpressions.[7] Unless otherwise specified, by "symbol" we shall mean "proper symbol".

Given a frame $\mathcal{D}$, a symbol of type α denotes an element of $\mathcal{D}_\alpha$. More precisely, an *assignment* into $\mathcal{D}$ is a function ϕ from the set of all symbols into $\bigcup_\alpha \mathcal{D}_\alpha$ such that the image by ϕ of a symbol s of type α is an element of $\mathcal{D}_\alpha$,

[6] In our treatment, the only difference between constants and variables is that constants cannot be bound.

[7] We do not consider parentheses as symbols of the language—see chapter 3.

called the *denotation of* s *in* ϕ. We shall also consider *partial assignments*, which are partial such functions. An *interpretation* is a pair $(\mathcal{D}, \phi)$ consisting of a frame $\mathcal{D}$ together with an assignment ϕ into the frame.

We now define by simultaneous induction the *formulas* of the typed λ-language, their *types*, their *denotations* in an interpretation $(\mathcal{D}, \phi)$ (for a fixed frame $\mathcal{D}$ but variable assignment ϕ), and the *binding* of variables:

- If s is a symbol of type α then "s" (i.e. the expression consisting of the single symbol s) is a formula of type α, which denotes the image by ϕ of s.

- If $\boldsymbol{A}$ is a formula of type $\alpha\beta$ denoting a function f from $\mathcal{D}_\beta$ to $\mathcal{D}_\alpha$, and $\boldsymbol{B}$ is a formula of type β denoting an element u of $\mathcal{D}_\beta$, then "$(\boldsymbol{A}\ \boldsymbol{B})$" is a formula of type α denoting $f(u)$. Such a formula is called an *application*, where $\boldsymbol{A}$ plays the *role of function* and $\boldsymbol{B}$ plays the *role of argument*.

- If $\boldsymbol{A}$ is a formula of type α and $\boldsymbol{x}$ is a variable of type β, then "$(\lambda \boldsymbol{x}\boldsymbol{A})$" is a formula of type $\alpha\beta$ which denotes the function from $\mathcal{D}_\beta$ to $\mathcal{D}_\alpha$ which maps every element u of $\mathcal{D}_\beta$ to the denotation of $\boldsymbol{A}$ in the interpretation $(\mathcal{D}, \phi')$, where ϕ' is the assignment which maps $\boldsymbol{x}$ to u but otherwise coincides with ϕ. Such a formula is called an *abstraction*, of which the subformula $\boldsymbol{A}$ is the *body*. The occurrence of $\boldsymbol{x}$ which immediately follows λ is a *binding* occurrence: its *scope* is the body of the abstraction, $\boldsymbol{A}$; it *binds* the occurrences of $\boldsymbol{x}$ in the body which are not themselves bound within $\boldsymbol{A}$ and which are not binding occurrences. An occurrence of a variable in a formula is *free* iff it is not a bound or binding occurrence.

Parentheses can be suppressed when doing so does not make a formula ambiguous, except parentheses around an application "$(\boldsymbol{B}\ \boldsymbol{C})$" which plays the role of argument in an application "$\boldsymbol{A}\ (\boldsymbol{B}\ \boldsymbol{C})$"; such parentheses are compulsory.[8] Synonimously with *formula* we shall sometimes say *well-formed formula*, especially when *well-formedness*, i.e. membership in the set of formulas of the typed λ-language, is being stressed.

It is clear that the denotation of a formula depends only on the denotations of the symbols which occur free in the formula. That is, given a frame

[8] ...even though the typing would make the formula unambiguous without parentheses.

$\mathcal{D}$, two assignments ϕ and ϕ' into $\mathcal{D}$, and a formula $\boldsymbol{A}$, if ϕ and ϕ' coincide on the constants which occur in $\boldsymbol{A}$, and on the variables which occur free in $\boldsymbol{A}$, then $\boldsymbol{A}$ has the same denotation in the interpretations $(\mathcal{D}, \phi)$ and $(\mathcal{D}, \phi')$.

We shall say that an interpretation *satisfies* $\boldsymbol{A}$ iff $\boldsymbol{A}$ is a formula of type o whose denotation is T.

A *context* of the typed λ-language is a pair $(\boldsymbol{A}, x)$, where $\boldsymbol{A}$ is a formula and x is a variable.[9] The type α of $\boldsymbol{A}$ is the *type of the context*, while the type β of x is its *argument-type*. If $\boldsymbol{B}$ is a formula of type β, then the result of substituting $\boldsymbol{B}$ for the free occurrences of x in $\boldsymbol{A}$ is a well-formed formula $\boldsymbol{C}$, of type α. We shall write $\boldsymbol{A}^x$ for $(\boldsymbol{A}, x)$ and $\boldsymbol{A}^x_B$ for the formula $\boldsymbol{C}$. If we let $\mathcal{C} = \boldsymbol{A}^x$ then we shall also write "$\mathcal{C}[\boldsymbol{B}]$" for the latter. A variable y is *free in* $\boldsymbol{A}^x$ iff it is free in $\boldsymbol{A}$ and it is distinct from x; it is *captured by* $\boldsymbol{A}^x$ iff, within $\boldsymbol{A}$, a free occurrence of x occurs in the scope of a binding occurrence of y; it is *adequate to* $\boldsymbol{A}^x$ iff it is neither free in $\boldsymbol{A}^x$ not captured by $\boldsymbol{A}^x$. In the latter case the contexts $\boldsymbol{A}^x$ and $(\boldsymbol{A}^x_y)^y$ are equivalent, in the sense that $\boldsymbol{A}^x_B$ is the same formula as $(\boldsymbol{A}^x_y)^y_B$ for any formula $\boldsymbol{B}$ of same type as x; moreover, x is adequate to $(\boldsymbol{A}^x_y)^y$ and $(\boldsymbol{A}^x_y)^y_x$ is the formula $\boldsymbol{A}$. The formula $\boldsymbol{B}$ is said to be *free for* x *in* $\boldsymbol{A}$ when no variable free in $\boldsymbol{B}$ is captured by $\boldsymbol{A}^x$.

A *simple context* is a context $\boldsymbol{A}^x$ such that x has exactly one free occurrence in $\boldsymbol{A}$.

2.3.3 Conversion—the typed λ-calculus

The typed λ-calculus consists of the typed λ-language, together with a collection of *conversions*. Conversions are certain binary relations between formulas, which will be useful in formulating the inference rules of the formal system, and for proving some of the results. In addition to the usual α, β and η-conversions, we define a γ-conversion. This is a partial converse of η-conversion; γ-normal form coincides with what is sometimes called η-long form. In this section we give the definitions and state some *normalization results*. The technical machinery can be found in appendix B.

A special vocabulary will be used for conversions. If $\mathcal{R}$ is a conversion, a *step* of $\mathcal{R}$ is a pair of formulas $(\boldsymbol{A}, \boldsymbol{B}) \in \mathcal{R}$. If $(\boldsymbol{A}, \boldsymbol{B}) \in \mathcal{R}$ we say that a

[9] This is a one-argument context; an n-argument context would be an $(n+1)$-tuple $(\boldsymbol{A}, x_1 \ldots x_n)$. In section 3.3 we define a more general notion of context which applies to languages other than the typed λ-language.

step of $\mathcal{R}$ *takes* $\boldsymbol{A}$ *to* $\boldsymbol{B}$ and we write $\boldsymbol{A} \xrightarrow{\mathcal{R}} \boldsymbol{B}$. A *chain* of $\mathcal{R}$ is a finite or infinite sequence of formulas where any two consecutive ones form a step. A finite chain is said to *terminate*. We say that $\mathcal{R}$ *converts* $\boldsymbol{A}$ to $\boldsymbol{B}$ iff there exists a chain whose first element is $\boldsymbol{A}$ and whose last element is $\boldsymbol{B}$, i.e. iff $(\boldsymbol{A}, \boldsymbol{B})$ is in the transitive closure of $\mathcal{R}$. Conversions are traditionally named by Greek letters (there is α-conversion, β-conversion and η-conversion; we add γ-conversion); this use of Greek letters bears no relation to the use of Greek letters as metalinguistic variables denoting types, even when the same letter is used in both ways in the same sentence.

There is an expectation that conversions preserve the denotations of formulas, so we shall say that a conversion $\mathcal{R}$ is *sound* iff whenever $\mathcal{R}$ converts $\boldsymbol{A}$ to $\boldsymbol{B}$ the formulas $\boldsymbol{A}$ and $\boldsymbol{B}$ have the same denotation in every interpretation.

We shall consider the following conversions and associated normalization results:

- α-CONVERSION. A step of α-conversion takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\alpha} \boldsymbol{B}$, iff: $\boldsymbol{A}$ is a formula of the form $\mathcal{C}[\lambda \boldsymbol{x} \boldsymbol{U}]$, where $\boldsymbol{U}$ is a formula of type α, $\boldsymbol{x}$ is a variable of type β and $\mathcal{C}$ is a simple context of argument-type $\alpha\beta$; and $\boldsymbol{B}$ is the formula $\mathcal{C}[\lambda \boldsymbol{y} \boldsymbol{U}^{\boldsymbol{x}}_{\boldsymbol{y}}]$, where $\boldsymbol{y}$ is a variable of type β adequate to $\boldsymbol{U}^{\boldsymbol{x}}$. This conversion is symmetric. If $\boldsymbol{A}$ converts to $\boldsymbol{B}$ we say that $\boldsymbol{A}$ and $\boldsymbol{B}$ are *the same up to renaming of bound variables.*

- β-CONVERSION. In a formula $\boldsymbol{A}$, a *β-redex* is a well-formed part of the form "$(\lambda \boldsymbol{x} \boldsymbol{U})\, \boldsymbol{V}$", where $\boldsymbol{U}$ and $\boldsymbol{V}$ are formulas, and $\boldsymbol{x}$ is a variable (of same type as $\boldsymbol{V}$). A step of β-conversion takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\beta} \boldsymbol{B}$, iff $\boldsymbol{A}$ is a formula with a β-redex "$(\lambda \boldsymbol{x} \boldsymbol{U})\, \boldsymbol{V}$" where $\boldsymbol{V}$ is free for $\boldsymbol{x}$ in $\boldsymbol{U}$, and $\boldsymbol{B}$ is the formula obtained by replacing the redex with "$\boldsymbol{U}^{\boldsymbol{x}}_{\boldsymbol{V}}$". A formula is said to be in *β-normal form (β-nf)* iff it contains no β-redexes. Notice that we say that "$(\lambda \boldsymbol{x} \boldsymbol{U})\, \boldsymbol{V}$" is a β-redex even if $\boldsymbol{V}$ is not free for $\boldsymbol{x}$ in $\boldsymbol{U}$; so a formula is *not* in β-nf if it contains a subformula of the form "$(\lambda \boldsymbol{x} \boldsymbol{U})\, \boldsymbol{V}$", whether $\boldsymbol{V}$ is free for $\boldsymbol{x}$ in $\boldsymbol{U}$ or not.

- $\alpha\beta$-CONVERSION. A step of $\alpha\beta$-conversion takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\alpha\beta} \boldsymbol{B}$, iff $\boldsymbol{A} \xrightarrow{\alpha} \boldsymbol{B}$ or $\boldsymbol{A} \xrightarrow{\beta} \boldsymbol{B}$. We say that a chain of $\alpha\beta$-conversion is *trivial* iff it is infinite but has a finite number of β-conversion steps. We say that a chain of $\alpha\beta$-conversion is *complete* iff it terminates in a formula

in β-nf, or else it is infinite. The last formula of a finite complete chain which starts with a formula $\boldsymbol{A}$ is said to be a β-nf *of* $\boldsymbol{A}$.

We observe that for every formula $\boldsymbol{A}$ there exists a non-trivial complete chain starting with $\boldsymbol{A}$. This is because if a formula has a β-redex, even though no β-conversion step may be applicable (because of variable capture), it is always possible to rename bound variables so that a β-conversion step will apply. So as long as the last formula of a chain is not in β-nf it is possible to find an extension of the chain which has an additional β-conversion step.

The *strong normalization theorem of the typed λ-calculus* [29, Apps.1,2] asserts the following: every non-trivial chain terminates; therefore every formula $\boldsymbol{A}$ has a β-nf (the last formula of a complete non-trivial chain starting with $\boldsymbol{A}$); and the β-nfs of $\boldsymbol{A}$ are all the same up to renaming of bound variables.

- η-CONVERSION. In a formula $\boldsymbol{A}$, an *η-redex* is a well-formed part of the form "$\lambda \boldsymbol{x}(\boldsymbol{U}\ \boldsymbol{x})$", where $\boldsymbol{U}$ is a formula of type $\alpha\beta$ and $\boldsymbol{x}$ is a variable of type β which is not free in $\boldsymbol{U}$. A step of η-conversion takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\eta} \boldsymbol{B}$, iff $\boldsymbol{A}$ is a formula having such a redex "$\lambda \boldsymbol{x}(\boldsymbol{U}\ \boldsymbol{x})$", and $\boldsymbol{B}$ is the result of replacing it with $\boldsymbol{U}$. A formula is said to be in *η-normal form (η-nf)* iff it has no η-redexes.

Although formulas in η-nf are compact, generally they are not the formulas used by mathematicians, and they cannot be written using traditional mathematical notations. Take for example the formula

$$\text{exists}_{o(o\iota)}\,(\lambda x_\iota(\text{equal}_{o\iota\iota}\ y_\iota\ x_\iota)) \tag{2.1}$$

As we shall see it can be written using traditional shorthands as:

$$\exists x(y = x)$$

But it is not in η-nf; η-conversion transforms it into

$$\text{exists}_{o(o\iota)}\,(\text{equal}_{o\iota\iota}\ y_\iota) \tag{2.2}$$

which is in η-nf and is more compact than (2.1), but cannot be expressed using the traditional shorthands for equality and existential quantification.

So it seems that the inverse of η-conversion would be of more interest to us than η-conversion, since it converts (2.2) into (2.1). Unfortunately, after a step of (η-conversion)$^{-1}$ takes (2.2) into (2.1), an additional step can take (2.1) into any of the following formulas:

$$(\lambda p_{o\iota}(\text{exists}_{o(o\iota)}\, p_{o\iota}))\ (\lambda x_\iota(\text{equal}_{o\iota\iota}\, y_\iota\, x_\iota)) \tag{2.3}$$

$$\text{exists}_{o(o\iota)}\ (\lambda z_\iota((\lambda x_\iota(\text{equal}_{o\iota\iota}\, y_\iota\, x_\iota))\, z_\iota) \tag{2.4}$$

$$\text{exists}_{o(o\iota)}\ (\lambda x_\iota((\lambda z_\iota(\text{equal}_{o\iota\iota}\, z_\iota)\, y_\iota\, x_\iota)) \tag{2.5}$$

$$\text{exists}_{o(o\iota)}\ (\lambda x_\iota((\lambda z_\iota(\text{equal}_{o\iota\iota}\, y_\iota)\, z_\iota)\, x_\iota)) \tag{2.6}$$

These transformations, however, can be blocked by specifying that the expansion $\boldsymbol{U} \rightarrow$ "$\lambda \boldsymbol{x}(\boldsymbol{U}\, \boldsymbol{x})$" shall not take place if $\boldsymbol{U}$ plays the role of function in an application, or if $\boldsymbol{U}$ is an abstraction. This leads to the introduction of a conversion which is a subrelation of (η-conversion)$^{-1}$; let us call it *γ-conversion*:

- γ-CONVERSION. In a given formula, a *γ-redex* is a well-formed part whose type is functional, which is not an abstraction and which does not play the role of function in an application. A step of *γ-conversion* takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\gamma} \boldsymbol{B}$, iff $\boldsymbol{A}$ is a formula containing a γ-redex $\boldsymbol{U}$ of type $\alpha\beta$, and $\boldsymbol{B}$ is the result of replacing the redex with "$\lambda \boldsymbol{x}(\boldsymbol{U}\, \boldsymbol{x})$", where $\boldsymbol{x}$ is a variable of type β which does not occur free in $\boldsymbol{U}$. A formula with no γ-redexes is said to be in *γ-normal form.*

  If $\boldsymbol{A}$ γ-converts to $\boldsymbol{B}$ and $\boldsymbol{B}$ is in γ-nf we say that $\boldsymbol{B}$ is a γ-nf *of* $\boldsymbol{A}$. A chain of γ-conversion is *complete* iff it is infinite or ends in a formula which is in γ-nf. A formula is in γ-nf iff no step of γ-conversion applies to it; so for every formula $\boldsymbol{A}$ there exists a complete chain starting with $\boldsymbol{A}$. The following strong normalization result is proved in appendix B (corollary B.9): every chain of γ-conversion terminates, therefore every formula $\boldsymbol{A}$ has a γ-nf (the last formula of a complete chain starting with $\boldsymbol{A}$); and the normal forms of $\boldsymbol{A}$ are all the same up to renaming of bound variables.

We shall say that a formula is in *$\beta\gamma$-nf* iff it is both in β-nf and in γ-nf. Formulas in $\beta\gamma$-nf are important because, as we shall see in chapter 3, they are those which can be expressed using ordinary mathematical notations. In addition, conversion to $\beta\gamma$-nf will be part of the process of elimination

of notations from formulas described in section 2.5. With this in mind we introduce the following conversion:

- $\alpha\beta\gamma$-CONVERSION. A step of $\alpha\beta\gamma$-conversion takes $\boldsymbol{A}$ to $\boldsymbol{B}$, $\boldsymbol{A} \xrightarrow{\alpha\beta\gamma} \boldsymbol{B}$, iff $\boldsymbol{A} \xrightarrow{\alpha} \boldsymbol{B}$ or $\boldsymbol{A} \xrightarrow{\beta} \boldsymbol{B}$ or $\boldsymbol{A} \xrightarrow{\gamma} \boldsymbol{B}$. We say that a chain of $\alpha\beta\gamma$-conversion is *trivial* iff it is infinite but has a finite number of β-conversion and γ-conversion steps. We say that a chain of $\alpha\beta\gamma$-conversion is *complete* iff it terminates in a formula in $\beta\gamma$-nf, or else it is infinite. The last formula of a finite complete chain which starts with a formula $\boldsymbol{A}$ is said to be a $\beta\gamma$-nf *of* $\boldsymbol{A}$. As for $\alpha\beta$-conversion, we observe that for every formula $\boldsymbol{A}$ there is a non-trivial complete chain starting with $\boldsymbol{A}$. Corollary B.11 in appendix B establishes the following strong normalization result for $\alpha\beta\gamma$-conversion: every non-trivial chain terminates, therefore every formula $\boldsymbol{A}$ has a $\beta\gamma$-nf (the last formula of a non-trivial complete chain starting with $\boldsymbol{A}$); and the $\beta\gamma$-nfs of $\boldsymbol{A}$ are all the same up to renaming of bound variables.

It should be noted that the steps of η-conversion which are not part of $(\gamma\text{-conversion})^{-1}$ are redundant in the presence of α- and β-conversion. Indeed such a step must either

1. Transform a formula of the form $\mathcal{C}[(\lambda x(\boldsymbol{U}\ x))\ \boldsymbol{V}]$, where x is not free in $\boldsymbol{U}$, into $\mathcal{C}[\boldsymbol{U}\ \boldsymbol{V}]$, or else

2. Transform a formula of the form $\mathcal{C}[\lambda y((\lambda x\boldsymbol{U})\ y)]$, where y is not free in "$\lambda x\boldsymbol{U}$", into $\mathcal{C}[\lambda x\boldsymbol{U}]$.

In the first case, the η-conversion step is also a β-conversion step. In the second case, the η-conversion step can be accomplished by an α-conversion step

$$\mathcal{C}[\lambda y((\lambda x\boldsymbol{U})\ y)] \xrightarrow{\alpha} \mathcal{C}[\lambda z((\lambda x\boldsymbol{U})\ z)]$$

to rename y, if necessary, to a variable z adequate to $\boldsymbol{U}^x$, followed by a β-conversion step

$$\mathcal{C}[\lambda z((\lambda x\boldsymbol{U})\ z)] \xrightarrow{\beta} \mathcal{C}[\lambda z(\boldsymbol{U}^x_z)]$$

followed by an α-conversion step

$$\mathcal{C}[\lambda z(\boldsymbol{U}^x_z)] \xrightarrow{\alpha} \mathcal{C}[\lambda x(\boldsymbol{U})].$$

Therefore the equivalence closure of $\alpha\beta\gamma$-conversion is the same as that of α-conversion $\cup$ β-conversion $\cup$ η-conversion.

The soundness of α-conversion, β-conversion and η-conversion is easy to establish, and well known. The soundness of the other conversions listed above follows.

2.3.4 Logical constants and models

As described so far, the typed λ-calculus is a formal system concerned with functions, function application and function abstraction. To turn it into a *logistic system* we shall make use of the fact that o denotes the two "truth-values," choose constants to be used as logical constants, and define the intended denotations of these logical constants.

Church's system [10] uses only, as logical constants, negation, disjunction, universal quantification, and description or selection operators. Andrews' system Q_0 [1, 5] uses only equality and description or selection operators. In both systems, other logical constructs are introduced by metalinguistic abbreviation. However, since we have seen that metalinguistic definitions are not easily amenable to mechanization, we shall make use of a full slate of logical constants.

To denote *equality* between objects of type α we shall use the constant "$\text{equal}_{o\alpha\alpha}$", with the shorthand

$$\boldsymbol{A} = \boldsymbol{B}$$

for $\text{equal}_{o\alpha\alpha}\ \boldsymbol{A}\ \boldsymbol{B}$. The denotation of a symbol of type $o\alpha\alpha$ in an assignment ϕ into a frame $\mathcal{D}$ is a function from $\mathcal{D}_\alpha$ to the set of functions from $\mathcal{D}_\alpha$ to $\{\text{F}, \text{T}\}$. We define the *intended denotation* of "$\text{equal}_{o\alpha\alpha}$" in the frame $\mathcal{D}$ as the function which maps every element u of $\mathcal{D}_\alpha$ to the function which maps u to T and every element of $\mathcal{D}_\alpha$ other than u to F. Then for every interpretation $\mathcal{I} = (\mathcal{D}, \phi)$ where ϕ maps $\text{equal}_{o\iota\iota}$ to its intended denotation, and for every pair of formulas $\boldsymbol{A}$, $\boldsymbol{B}$ of type α, the denotation of

$$\text{equal}_{o\alpha\alpha}\ \boldsymbol{A}\ \boldsymbol{B}$$

in $\mathcal{I}$ is T iff the denotations of $\boldsymbol{A}$ and $\boldsymbol{B}$ in $\mathcal{I}$ are the same.

To denote the conjunction of two truth values we use the constant and_{ooo}, with the shorthand

$$\boldsymbol{A} \wedge \boldsymbol{B}$$

for

$$\text{and}_{ooo}\ \boldsymbol{A}\ \boldsymbol{B}.$$

The denotation of a symbol of type ooo in an assignment (into any frame) is a function from $\{\text{F}, \text{T}\}$ to the set of functions from $\{\text{F}, \text{T}\}$ to $\{\text{F}, \text{T}\}$. The intended denotation of and_{ooo} is the function which maps the truth-value u to the function which maps the truth-value v to T if $u = v = \text{T}$ and to F otherwise. That is, the intended denotation of and_{ooo} is the usual *truth-table* for conjunction, in curried form. Then, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are formulas of type o, and if $\mathcal{I} = (\mathcal{D}, \phi)$ is an interpretation where ϕ maps and_{ooo} to its intended denotation, $\mathcal{I}$ satisfies "$\text{and}_{ooo}\ \boldsymbol{A}\ \boldsymbol{B}$" iff it satisfies $\boldsymbol{A}$ and $\boldsymbol{B}$.

Besides and_{ooo} we shall use, as logical connectives, the constants not_{oo}, or_{ooo} and implies_{ooo}, with the usual truth-tables as intended denotations, and with "$\neg\boldsymbol{A}$", "$\boldsymbol{A} \vee \boldsymbol{B}$" and "$\boldsymbol{A} \supset \boldsymbol{B}$", where $\boldsymbol{A}$, $\boldsymbol{B}$ are formulas of type o, as shorthands for "$\text{not}_{oo}\ \boldsymbol{A}$", "$\text{or}_{ooo}\ \boldsymbol{A}\ \boldsymbol{B}$" and "$\text{implies}_{ooo}\ \boldsymbol{A}\ \boldsymbol{B}$".[10] Then an interpretation in which the logical connectives have their intended denotations satisfies "$\neg\boldsymbol{A}$" iff it does not satisfy $\boldsymbol{A}$, it satisfies "$\boldsymbol{A} \vee \boldsymbol{B}$" iff it satisfies $\boldsymbol{A}$ or $\boldsymbol{B}$, and it satisfies "$\boldsymbol{A} \supset \boldsymbol{B}$" iff either it does not satify $\boldsymbol{A}$ or else it satisfies $\boldsymbol{B}$.

We shall also use the constants true_o and false_o, with shorthands $\top$ and $\bot$, as 0-ary logical connectives, with intended denotations T and F.

As universal quantifiers we shall use the constants $\text{forall}_{o(o\alpha)}$, for every type α. If $\boldsymbol{x}$ is a variable of type α and $\boldsymbol{A}$ a formula of type o, we write

$$\forall \boldsymbol{x}\boldsymbol{A}$$

for

$$\text{forall}_{o(o\alpha)}\ \lambda\boldsymbol{x}\boldsymbol{A}.$$

The denotation of a symbol of type $o(o\alpha)$ in an assignment into the frame $\mathcal{D}$ is a function from the set of functions from $\mathcal{D}_\alpha$ to $\{\text{F}, \text{T}\}$ to the set $\{\text{F}, \text{T}\}$. The intended denotation of $\text{forall}_{o(o\alpha)}$ is the function which maps the function with constant value T to T, and every other function to F. For every formula $\boldsymbol{A}$ of type o, variable $\boldsymbol{x}$ of type α, and interpretation $\mathcal{I} = (\mathcal{D}, \phi)$ where ϕ maps $\text{forall}_{o(o\alpha)}$ to its intended denotation, $\mathcal{I}$ satisfies "$\forall\boldsymbol{x}\boldsymbol{A}$" iff "$\lambda\boldsymbol{x}\boldsymbol{A}$" denotes the function with constant value T, i.e. iff for every $u \in \mathcal{D}_\alpha$, the

[10] Negation will have higher precedence than conjunction or disjunction, and these will have higher precedence than implication.

interpretation $(\mathcal{D}, \phi')$, where ϕ' is the assignment which maps $\boldsymbol{x}$ to u and otherwise coincides with ϕ, satisfies $\boldsymbol{A}$.

Besides "for all" we shall use the quantifiers "there exists", "there exists at most one" and "there exists exactly one". ("Exactly one" is useful in connection with the description operator, and "at most one" is useful in connection with the introduction and elimination rules for "exactly one".) For this purpose we choose the constants $\text{exists}_{o(o\alpha)}$, $\text{atmost}_{o(o\alpha)}$ and $\text{unique}_{o(o\alpha)}$, with the shorthands "$\exists \boldsymbol{x}\boldsymbol{A}$", "$!\boldsymbol{x}\boldsymbol{A}$" and "$\exists!\boldsymbol{x}\boldsymbol{A}$" for "$\text{exists}_{o(o\alpha)}\lambda\boldsymbol{x}\boldsymbol{A}$", "$\text{atmost}_{o(o\alpha)}\lambda\boldsymbol{x}\boldsymbol{A}$" and "$\text{unique}_{o(o\alpha)}\ \lambda\boldsymbol{x}\boldsymbol{A}$".[11] The intended denotations of the constants are the obvious ones: they map to T the functions from $\mathcal{D}_\alpha$ to $\{\text{F}, \text{T}\}$ which take the value T at least once ($\text{exists}_{o(o\alpha)}$), at most once ($\text{atmost}_{o(o\alpha)}$), exactly once ($\text{unique}_{o(o\alpha)}$). Then, if ϕ is an assignment into a frame $\mathcal{D}$ in which these constants have their intended denotations, the interpretation $\mathcal{I} = (\mathcal{D}, \phi)$ satisfies "$\exists \boldsymbol{x}\boldsymbol{A}$" iff *some* interpretation $(\mathcal{D}, \phi')$ where ϕ' coincides with ϕ everywhere except perhaps at $\boldsymbol{x}$ satisfies $\boldsymbol{A}$; $\mathcal{I}$ satisfies "$!\boldsymbol{x}\boldsymbol{A}$" iff *at most* one such interpretation satisfies $\boldsymbol{A}$; and $\mathcal{I}$ satisfies "$\exists!\boldsymbol{x}\boldsymbol{A}$" iff exactly one such interpretation satisfies $\boldsymbol{A}$.

Finally, we use the constants $\text{the}_{\alpha(o\alpha)}$ as *description operators*, i.e. to denote objects specified by definite descriptions. We write

$$\mu\boldsymbol{x}\boldsymbol{A}$$

for

$$\text{the}_{\alpha(o\alpha)}\ \lambda\boldsymbol{x}\boldsymbol{A}.$$

The description operators differ from the other logical constants in that they have a range of admissible denotations, rather than a unique intended denotation. The denotation of a symbol of type $\alpha(o\alpha)$ in an assignment into a frame $\mathcal{D}$ is a function from the set of functions from $\mathcal{D}_\alpha$ to $\{\text{F}, \text{T}\}$ to the set $\mathcal{D}_\alpha$. Such a function is an *admissible denotation* of $\text{the}_{\alpha(o\alpha)}$ iff it maps every function $f : \mathcal{D}_\alpha \mapsto \{\text{F}, \text{T}\}$ which takes the value T for exactly one element u of $\mathcal{D}_\alpha$ to precisely that element u. Given a formula $\boldsymbol{A}$ of type o, a variable $\boldsymbol{x}$ of type α, and an interpretation $\mathcal{I} = (\mathcal{D}, \phi)$ such that ϕ maps $\text{the}_{\alpha(o\alpha)}$ to an admissible denotation, if there exists a unique $u \in \mathcal{D}_\alpha$ such that $\boldsymbol{A}$ denotes T in the interpretation $\mathcal{I} = (\mathcal{D}, \phi')$ where ϕ' maps $\boldsymbol{x}$ to u and otherwise coincides with ϕ, then "$\mu\boldsymbol{x}\boldsymbol{A}$" denotes u in $\mathcal{I}$. If there are zero or

[11] The shorthands for quantifiers will have higher precedence than those for connectives.

more than one such u, then "$\mu x \boldsymbol{A}$" is "undefined" in the sense that the fact that ϕ maps $\text{the}_{\alpha(o\alpha)}$ to an admissible denotation does not tell us anything about the denotation of "$\mu x \boldsymbol{A}$" in $\mathcal{I}$.

The *logical constants* are the equality constants, the logical connectives, the quantifiers, and the description operators. An interpretation $(\mathcal{D}, \phi)$ is a *logical interpretation* iff ϕ maps the description operators to admissible denotations in $\mathcal{D}$, and the other logical constants to their intended denotations in the frame.

A *theory* is a set of formulas of type o. The formulas which are elements of a theory Γ are called the *axioms* of Γ. A *model* of a theory Γ is a logical interpretation which satifies the axioms of Γ.

We shall say that a formula $\boldsymbol{P}$ of type o is a *logical consequence* of a theory Γ, written

$$\Gamma \models \boldsymbol{P},$$

iff every model of Γ satisfies $\boldsymbol{P}$.

2.3.5 Natural deduction proofs

Having defined the relation $\models$ of logical consequence between theories and formulas, we must now define the deducibility relation $\vdash$, i.e. we must provide an inference system.

We are using a full slate of logical constants, rather than just a small number of primitive constants, so it is natural to turn to a deduction paradigm which places all the logical connectives and quantifiers on the same footing; such is *natural deduction*. Another reason for choosing natural deduction is that the introduction and elimination rules of natural deduction systems are indeed "natural," in the sense that they correspond rather closely to steps of proofs done by hand. The primitive inference rules of a natural deduction system are sufficient by themselves (even without the addition of derived rules and decition procedures) for constructing rather complex proofs in a natural way. Thus they can form the basis of the inference tool-kit of a proof developement system. We shall see in section 4.2 that this is the case in the theorem prover Watson.

Natural deduction has two ingredients:

1. The use of *proof trees* or, equivalently as observed in [47, §1.6], of linear proofs where the lines of the proof are asymmetric sequents.

2. The use of introduction and elimination rules for the connectives and quantifiers.

The PDSs HOL and TPS are both described as being based on natural deduction, and this is true in the sense that they feature the first of the two ingredients. However, the formal systems upon which they are based are not standard natural deduction systems, lacking the second ingredient.

In our formal system a *sequent* is a pair $(\Gamma, \boldsymbol{P})$, where Γ is a theory and $\boldsymbol{P}$ a formula of type o; we shall often write $\Gamma \vdash \boldsymbol{P}$ rather than $(\Gamma, \boldsymbol{P})$. The axioms of Γ are the *hypotheses*, or *assumptions*, of the sequent, and $\boldsymbol{P}$ is the *conclusion* of the sequent. An *inference rule* with n premises is a mechanically verifiable relation R of arity $n+1$ between sequents, i.e. a set of tuples of the form $(S_1, \ldots S_n, S')$ where $S_1, \ldots S_n$ and S' are sequents. We shall refer to each element $(S_1, \ldots S_n, S')$ of the inference rule R as *an instance of* R, of which $S_1, \ldots S_n$ are the *premises*, and S' is the *conclusion*; and we shall say that S' *follows* from $S_1, \ldots S_n$ by R. We shall write inference rules in the usual schematic way: the premises are written above a horizontal bar, and the conclusion below it, with any conditions written as a comment to the right of the bar; the bar is omitted when there are no premises. A *proof* is a sequence of sequents, called the *lines* of the proof, such that each line follows from zero of more preceding lines by an inference rule. A proof *of* a sequent $(\Gamma, \boldsymbol{P})$ is a proof whose last line is $(\Gamma, \boldsymbol{P})$. We shall say that $\boldsymbol{P}$ *follows* from Γ, written $\Gamma \vdash \boldsymbol{P}$, iff there exists a proof of $(\Gamma, \boldsymbol{P})$. We shall refer to this relation between theories and formulas as the *deducibility* relation. If $\Gamma \vdash \boldsymbol{P}$ we shall say that $\boldsymbol{P}$ is a theorem of Γ, or that the sequent $(\Gamma, \boldsymbol{P})$ is a theorem. When Γ is empty we shall write $\vdash \boldsymbol{P}$ and we shall say that $\boldsymbol{P}$ is a theorem of H.O.L. A *derived inference rule* is a rule which could be added to the *primitive* rules of the system without modifying the deducibility relation.

Observe that $\Gamma \vdash \boldsymbol{P}$ has two meanings: it can refer to a sequent, or it can assert that $\boldsymbol{P}$ follows from Γ. Correspondingly, the inference rule notation (with the premises above a fraction bar and the conclusion below it) also has two meanings. We have explained it above as a schema, i.e. as a description of a generic instance of the rule, the symbol $\vdash$ being interpreted as a sequent constructor. But if the symbol $\vdash$ is interpreted as denoting the deducibilty relation, the inference rule notation can be read as an implication: the statements above the fraction bar together with the side condition imply

the statement below the bar.[12] This second reading is particularly useful for *derived rules*: a rule described by the fraction bar notation is a derived rule iff the implication holds.

We shall say that a sequent $(\Gamma, \boldsymbol{P})$ is *true* iff $\Gamma \models \boldsymbol{P}$. An inference rule is *sound* iff whenever the premises of an instance of the rule are true, the conclusion is also true. We shall now give the inference rules of a system of natural deduction for H.O.L. and show that they are sound.

2.3.6 Inference rules

The following conditions have been omitted from the rules for the sake of clarity:

1. Γ, Γ', Γ'' are theories.
2. $\boldsymbol{A}$, $\boldsymbol{B}$ are formulas of type α.
3. $\mathcal{C}$ is a simple context of type o and argument-type α.
4. $\boldsymbol{P}$, $\boldsymbol{Q}$,.$\boldsymbol{R}$are formulas of type o.
5. x, $y$are variables of type α.

Each rule implicitly includes all applicable conditions among the above ones. If two or more applicable conditions mention "α", they scope of "α" is the entire rule. For example, when the meta-variables "$\boldsymbol{x}$" and "$\boldsymbol{A}$" occur together in the rule $\forall$-*elimination*, the implicit conditions assert that "$\boldsymbol{x}$ and $\boldsymbol{A}$ have the *same* type α".

We write the hypotheses of a sequent in the traditional way, as a theory name followed by a comma and additional formulas separated by commas. Both the theory and the additional formulas are optional. For example,

$$\Gamma, \neg \boldsymbol{P}$$

refers to the set of hypotheses

$$\Gamma \cup \{\text{“}\neg \boldsymbol{P}\text{”}\},$$

[12] The deducibility relation could alternatively be defined as the smallest relation which satisfies the set of those implications, one for each primitive inference rule of the system.

and in

$$\boldsymbol{P} \vdash \boldsymbol{P}$$

is the set of hypotheses is $\{\boldsymbol{P}\}$.

Intrinsic rules

The first two inference rules are required by the fact that proof lines are sequents rather than formulas; they are trivially sound.

1. *Reflexivity of* $\vdash$.

$$\boldsymbol{P} \vdash \boldsymbol{P}$$

2. *Monotonicity of* $\vdash$.

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P}}$$

Substitutivity of equality

3. *Substitutivity of equality.*

$$\frac{\Gamma \vdash \mathcal{C}[\boldsymbol{A}] \qquad \Gamma' \vdash \boldsymbol{A} = \boldsymbol{B}}{\Gamma \cup \Gamma' \vdash \mathcal{C}[\boldsymbol{B}]} \qquad \begin{array}{l}\mathcal{C} \text{ does not capture any} \\ \text{variable free in } \Gamma'\end{array}$$

PROOF OF SOUNDNESS. Let Γ, Γ' be two theories, $\boldsymbol{A}$ and $\boldsymbol{B}$ two formulas of type α, and $\mathcal{C}$ a simple context of type o and argument-type α. Assume that $\Gamma \models \mathcal{C}(\boldsymbol{A})$ and $\Gamma' \models \boldsymbol{A} = \boldsymbol{B}$.

We shall first show that, in every model of Γ', $\mathcal{C}[\boldsymbol{A}]$ and $\mathcal{C}[\boldsymbol{B}]$ have the same denotation. Let $\mathcal{C}$ by the pair $(\boldsymbol{P}, \boldsymbol{x})$, where $\boldsymbol{x}$ is a variable of type α, and $\boldsymbol{P}$ is a formula of type o having a single[13] free occurrence of $\boldsymbol{x}$ (and, for simplicity, no bound or binding occurrences of $\boldsymbol{x}$). Consider a parse tree for $\boldsymbol{P}$ and let $n_0 \ldots n_j$ $(j \geq 0)$ be the ascending path going from the root of the subformula "$\boldsymbol{x}$" to the root of the entire tree. A parse tree for $\mathcal{C}[\boldsymbol{A}]$ can be obtained by grafting[14] a parse tree for $\boldsymbol{A}$ onto node n_0, without disturbing the

[13] Recall that the implicit condition on $\mathcal{C}$ asserts that $\mathcal{C}$ is a *simple* context. Hence if $\mathcal{C} = (\boldsymbol{P}, \boldsymbol{x})$ there is exactly one occurrence of $\boldsymbol{x}$ in $\boldsymbol{P}$. Of course the rule of *Substitivity of equality* for arbitrary contexts $\mathcal{C}$ holds trivially as a *derived* rule of inference in the formal system.

[14] What grafting means should be clear without further explanations, but a formal definition can be found in appendix B, page 180.

original parse tree above n_0. Let $\boldsymbol{A}_0 = \boldsymbol{A}$ and let $\boldsymbol{A}_1 \ldots \boldsymbol{A}_j$ be the formulas corresponding to the subtrees rooted at $n_1 \ldots n_j$ in the tree resulting from the graft. In the same way, a parse tree for $\mathcal{C}[\boldsymbol{B}]$ is obtained by grafting a parse tree for $\boldsymbol{B}$ onto n_0. Let $\boldsymbol{B}_0 = \boldsymbol{B}$ and let $\boldsymbol{B}_1 \ldots \boldsymbol{B}_j$ be the formulas corresponding to the subtrees rooted at $n_1 \ldots n_j$ in the tree resulting from the graft. We show by induction that every model of Γ' assigns the same denotation to $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$, for every i, $0 \leq i \leq j$, and so, in particular, to $\boldsymbol{A}_j = \mathcal{C}[\boldsymbol{A}]$ and $\boldsymbol{B}_j = \mathcal{C}[\boldsymbol{B}]$. This is the case for $i = 0$. Assume that the assertion holds for i, $0 \leq i < j$. Then it is obvious that it holds for $i+1$ in the case where n_{i+1} is an application node. If n_{i+1} is an abstraction node, then $\boldsymbol{A}_{i+1}$ is of the form "$\lambda z \boldsymbol{A}_i$", where z is a variable, and $\boldsymbol{B}_{i+1}$ is "$\lambda z \boldsymbol{B}_i$". Let $(\mathcal{D}, \phi)$ be a model of Γ'. Since the variable z is captured by $\mathcal{C}$ it is not free in Γ', so every interpretation $(\mathcal{D}, \phi')$ where ϕ' differs from ϕ only in the denotation of z is also a model of Γ', hence, by induction hypothesis, assigns the same denotations to $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$. Therefore $(\mathcal{D}, \phi)$ itself assigns the same denotations to "$\lambda z \boldsymbol{A}_i$" and "$\lambda z \boldsymbol{B}_i$", i.e. to $\boldsymbol{A}_{i+1}$ and $\boldsymbol{B}_{i+1}$.

Now, every model of $\Gamma \cup \Gamma'$ is a model of Γ, so it satisfies $\mathcal{C}[\boldsymbol{A}]$, and a model of Γ', so it assigns the denotation T to $\mathcal{C}[\boldsymbol{B}]$ as well as to $\mathcal{C}[\boldsymbol{A}]$. □

Conversion rules

Their soundness follows immediately from the soundness of conversion (section 2.3.3), given the intended denotation of the constants $\text{equal}_{o\alpha\alpha}$.

4. *α-Conversion*

$$\frac{\boldsymbol{A} \xrightarrow{\alpha} \boldsymbol{B}}{\vdash \boldsymbol{A} = \boldsymbol{B}}$$

5. *β-Conversion*

$$\frac{\boldsymbol{A} \xrightarrow{\beta} \boldsymbol{B}}{\vdash \boldsymbol{A} = \boldsymbol{B}}$$

6. *γ-Conversion*

$$\frac{\boldsymbol{A} \xrightarrow{\gamma} \boldsymbol{B}}{\vdash \boldsymbol{A} = \boldsymbol{B}}$$

Rules concerning the logical connectives

We include here the introduction and elimination rules of intuitionistic logic for the logical connectives, and the contradiction rule of classical logic.

7. $\top$-*Introduction*

$$\vdash \top$$

8. $\neg$-*Introduction*

$$\frac{\Gamma, \boldsymbol{P} \vdash \bot}{\Gamma \vdash \neg \boldsymbol{P}}$$

9. $\neg$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \qquad \Gamma' \vdash \neg \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \bot}$$

10. $\wedge$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P} \qquad \Gamma' \vdash \boldsymbol{Q}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}$$

11. $\wedge$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{P}} \qquad \frac{\Gamma \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{Q}}$$

12. $\vee$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q}} \qquad \frac{\Gamma \vdash \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q}}$$

13. $\vee$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q} \qquad \Gamma', \boldsymbol{P} \vdash \boldsymbol{R} \qquad \Gamma'', \boldsymbol{Q} \vdash \boldsymbol{R}}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash \boldsymbol{R}}$$

14. $\supset$-*Introduction*

$$\frac{\Gamma, \boldsymbol{P} \vdash \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q}}$$

15. $\supset$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q} \qquad \Gamma' \vdash \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{Q}}$$

16. *Contradiction*

$$\frac{\Gamma, \neg \boldsymbol{P} \vdash \bot}{\Gamma \vdash \boldsymbol{P}}$$

PROOF OF SOUNDNESS. The soundness of each of these rules is obvious from the definition of the intended denotations of the logical connectives, and from the observations made in section 2.3.4. As a sample, we prove the soundness of $\wedge$-introduction.

Assume $\Gamma \models \boldsymbol{P}$ and $\Gamma' \models \boldsymbol{Q}$. Let $\mathcal{I} = (\mathcal{D}, \phi)$ be an arbitrary model of $\Gamma \cup \Gamma'$. $\mathcal{I}$ is a logical interpretation, so ϕ maps and_{ooo} to its intended denotation. Then, as we observed in section 2.3.4, $\mathcal{I}$ satisfies $\boldsymbol{P} \wedge \boldsymbol{Q}$ iff it satisfies $\boldsymbol{P}$ and $\boldsymbol{Q}$. But $\mathcal{I}$ is a model of Γ, so it satifies $\boldsymbol{P}$, and a model of Γ', so it satisfies $\boldsymbol{Q}$. Therefore $\mathcal{I}$ satisfies $\boldsymbol{P} \wedge \boldsymbol{Q}$. Thus $\Gamma \cup \Gamma' \models \boldsymbol{P} \wedge \boldsymbol{Q}$. □

Equivalence and equality

The inference rules that we have introduced so far are compatible with a wider class of interpretations than the ones which we have specified, namely with interpretations in which the type o denotes a set of "propositions" with arbitrary cardinality, partitioned into two classes: the "true propositions" and the "false propositions". Then the logical equivalence: "$\boldsymbol{P} \supset \boldsymbol{Q}$" together with "$\boldsymbol{Q} \supset \boldsymbol{P}$", would mean that $\boldsymbol{P}$ and $\boldsymbol{Q}$ denote propositions in the same partition, but not necessarily identical. We rule out such interpretations by the following rule:

$$\frac{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q} \qquad \Gamma' \vdash \boldsymbol{Q} \supset \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P} = \boldsymbol{Q}} \tag{2.7}$$

This rule corresponds to Church's axiom [10, p. 61] "$p \equiv q \supset p = q$" (which, he says, means that there are "only two propositions"). It is obviously sound for our class of logical interpretations.

When o does denote the set $S = \{\text{F}, \text{T}\}$, equality in S is logical equivalence; that is, the intended denotation of equal_{ooo} is the curried truth-table corresponding to logical equivalence; so there is no reason for having a separate connective for logical equivalence, besides equal_{ooo}. We shall then use the shorthand

$$\boldsymbol{P} \equiv \boldsymbol{Q}$$

as standing for

$$\text{equal}_{ooo}\, \boldsymbol{P}\, \boldsymbol{Q}$$

(in addition to the shorthand "$\boldsymbol{P} \equiv \boldsymbol{Q}$" which remains in effect). The inference rule (2.7) can then be written:

17. $\equiv$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q} \qquad \Gamma' \vdash \boldsymbol{Q} \supset \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P} \equiv \boldsymbol{Q}}$$

Rules concerning the quantifiers

18. $\forall$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \vdash \forall \boldsymbol{x} \boldsymbol{P}} \quad \boldsymbol{x} \text{ not free in } \Gamma$$

19. $\forall$-*Elimination*

$$\frac{\Gamma \vdash \forall \boldsymbol{x} \boldsymbol{P}}{\Gamma \vdash \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{A}}} \quad \boldsymbol{A} \text{ free for } \boldsymbol{x} \text{ in } \boldsymbol{P}$$

20. $\exists$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{A}}}{\Gamma \vdash \exists \boldsymbol{x} \boldsymbol{P}} \quad \boldsymbol{A} \text{ free for } \boldsymbol{x} \text{ in } \boldsymbol{P}$$

21. $\exists$-*Elimination*

$$\frac{\Gamma \vdash \exists \boldsymbol{x} \boldsymbol{P} \qquad \Gamma', \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{y}} \vdash \boldsymbol{Q}}{\Gamma \cup \Gamma' \vdash \boldsymbol{Q}} \quad \begin{array}{l} \boldsymbol{y} \text{ adequate to } \boldsymbol{P}^{\boldsymbol{x}}; \\ \boldsymbol{y} \text{ not free in } \Gamma' \text{ or } \boldsymbol{Q} \end{array}$$

22. !-*Introduction*

$$\frac{\Gamma \vdash \forall \boldsymbol{x} \forall \boldsymbol{y} (\boldsymbol{P} \wedge \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{y}} \supset \boldsymbol{x} = \boldsymbol{y})}{\Gamma \vdash ! \boldsymbol{x} \boldsymbol{P}} \quad \boldsymbol{y} \text{ adequate to } \boldsymbol{P}^{\boldsymbol{x}}$$

23. !-*Elimination*

$$\frac{\Gamma \vdash ! \boldsymbol{x} \boldsymbol{P}}{\Gamma \vdash \forall \boldsymbol{x} \forall \boldsymbol{y} (\boldsymbol{P} \wedge \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{y}} \supset \boldsymbol{x} = \boldsymbol{y})} \quad \boldsymbol{y} \text{ adequate to } \boldsymbol{P}^{\boldsymbol{x}}$$

24. $\exists$!-*Introduction*

$$\frac{\Gamma \vdash \exists \boldsymbol{x} \boldsymbol{P} \qquad \Gamma \vdash ! \boldsymbol{x} \boldsymbol{P}}{\Gamma \vdash \exists ! \boldsymbol{x} \boldsymbol{P}}$$

25. $\exists!$-*Elimination*

$$\frac{\Gamma \vdash \exists! x \boldsymbol{P}}{\Gamma \vdash \exists x \boldsymbol{P}} \qquad \frac{\Gamma \vdash \exists! x \boldsymbol{P}}{\Gamma \vdash !x \boldsymbol{P}}$$

PROOF OF SOUNDNESS. The proofs of soundness of these rules are left to the reader. They present no difficulty given the observations made in section 2.3.4 about logical interpretations and the following two observations:

1. If y is adequate to $\boldsymbol{P}^x$, and if ϕ and ϕ' are assignments into a frame $\mathcal{D}$ which coincide on variables other than x and y, and such that $\phi(x) = \phi'(y)$, then the denotation of $\boldsymbol{P}$ in the interpretation $(\mathcal{D}, \phi)$ coincides with the denotation of $\boldsymbol{P}^x_y$ in the interpretation $(\mathcal{D}, \phi')$.

2. If $\boldsymbol{A}$ is free for $\boldsymbol{P}^x$ and denotes u in an interpretation $(\mathcal{D}, \phi)$, and if ϕ' is the assignment into $\mathcal{D}$ which maps x to u and otherwise coincides with ϕ, then the denotation of $\boldsymbol{P}^x_{\boldsymbol{A}}$ in the interpretation $(\mathcal{D}, \phi)$ coincides with the denotation of $\boldsymbol{P}$ in the interpretation $(\mathcal{D}, \phi')$.

Rule concerning the description operators

26. μ-*Introduction*

$$\frac{\Gamma \vdash \exists! x \boldsymbol{P}}{\Gamma \vdash \boldsymbol{P}^x_{\mu x P}} \qquad \text{“}\mu x \boldsymbol{P}\text{” free for } x \text{ in } \boldsymbol{P}$$

PROOF OF SOUNDNESS. Assume $\Gamma \models \exists! x \boldsymbol{P}$. Let $\mathcal{I} = (\mathcal{D}, \phi)$ be an arbitrary model of Γ. $\mathcal{I}$ satisfies “$\exists! x \boldsymbol{P}$”. Hence, since it is a logical interpretation, there exists exactly one $u \in \mathcal{D}_\alpha$ (recall that α is the type of x) such that the interpretation $\mathcal{I}' = (\mathcal{D}, \phi')$, where ϕ' is the assignment which maps x to u and otherwise coincides with ϕ, satisfies $\boldsymbol{P}$. But then the denotation of “$\mu x \boldsymbol{P}$” in $\mathcal{I}$ is precisely u. Since “$\mu x \boldsymbol{P}$” is free for x in $\boldsymbol{P}$, by observation 2 above, the denotation of $\boldsymbol{P}^x_{\mu x P}$ in $\mathcal{I}$ coincides with the denotation of $\boldsymbol{P}$ in $\mathcal{I}'$; i.e. $\mathcal{I}$ satisfies $\boldsymbol{P}^x_{\mu x P}$. So $\Gamma \models \boldsymbol{P}^x_{\mu x P}$. □

2.3.7 Soundness and incompleteness

Since each inference rule is sound the inference system itself is sound, i.e. $\Gamma \vdash \boldsymbol{P}$ implies $\Gamma \models \boldsymbol{P}$. It is not complete, however. It follows from Gödel's

incompleteness theorem that no inference system for H.O.L. can be sound and complete for standard models. Our formal system is, however, equivalent to the system of Church [10]. (In appendix C the equivalence is stated precisely, and a sketch of the proof is given.) It follows from this equivalence that the formal system is sound and complete for general models.

2.3.8 Derived rules for equality

Reflexivity of equality has not been included among the primitive rules of inference because it is redundant: for a formula $\boldsymbol{A}$ with at least one abstraction, it is a special case of the rule of α-conversion; for a formula without abstraction, it can be derived using $\vdash (\lambda x_\alpha x_\alpha)\boldsymbol{A} = \boldsymbol{A}$. *Symmetry* and *Transitivity of equality* follow immediately from reflexivity and substitutivity. So we have the following three derived rules of inference for equality:

$$\vdash \boldsymbol{A} = \boldsymbol{A} \qquad \frac{\Gamma \vdash \boldsymbol{A} = \boldsymbol{B}}{\Gamma \vdash \boldsymbol{B} = \boldsymbol{A}} \qquad \frac{\Gamma \vdash \boldsymbol{A} = \boldsymbol{B} \quad \Gamma' \vdash \boldsymbol{B} = \boldsymbol{C}}{\Gamma \cup \Gamma' \vdash \boldsymbol{A} = \boldsymbol{C}}$$

where Γ, Γ', Γ'' are arbitrary theories, and $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are formulas of the same type.

2.4 H.O.L. as a conservative extension of F.O.L.

2.4.1 First-order formulas

In this section we identify the terms and sentences of traditional F.O. logic with certain formulas of the typed λ-language.

To the constants and variables of F.O.L. correspond the constants and variables of type ι, which we shall call the *individual* constants and variables (because they denote individuals). From now on we shall often omit the subscript ι from individual variables, to give them a more traditional appearance. (This is what the PDS Watson does when writing out formulas. The ι subscript is not omitted from individual constants because unsubscripted roman identifiers are used as keywords. But there will be mathematical symbols or

keywords to stand for most individual constants that we shall use, e.g. $\emptyset$ for emptyset$_\iota$.)

An n-ary *predicate symbol* ($n \geq 1$) is a symbol (variable or constant) of type $o\alpha_1 \ldots \alpha_n$, $\alpha_1 = \ldots = \alpha_n = \iota$, other than the logical constant equal$_{o\iota\iota}$. An n-ary *function symbol* ($n \geq 1$) is a symbol (variable or constant) of type $\iota\alpha_1 \ldots \alpha_n$, $\alpha_1 = \ldots = \alpha_n = \iota$.

A *F.O. term* is defined by induction as follows:

1. If s is an individual constant or variable, then the formula "s" is a F.O. term.

2. If f is an n-ary function symbol, $n \geq 1$, and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ are n F.O. terms, then "$f\ \boldsymbol{T}_1\ \ldots\ \boldsymbol{T}_n$" is a F.O. term.

(More generally, we shall call *terms* the formulas of type ι, even if they are not F.O. terms.) A *F.O. sentence* is defined by induction as follows:

1. If $\boldsymbol{T}$ and $\boldsymbol{T}'$ are F.O. terms, then "$\boldsymbol{T} = \boldsymbol{T}'$" (i.e. "equal$_{o\iota\iota}\ \boldsymbol{T}\ \boldsymbol{T}'$") is a F.O. sentence.

2. If p is an n-ary predicate symbol, $n \geq 1$, and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ are n terms, then "$p\ \boldsymbol{T}_1\ \ldots\ \boldsymbol{T}_n$" is a F.O. sentence.

3. "$\bot$" (i.e. "false$_o$") is a F.O. sentence.

4. If $\boldsymbol{S}$ is a F.O. sentence, then "$\neg\boldsymbol{S}$" (i.e. "not$_{oo}\ \boldsymbol{S}$") is a F.O. sentence.

5. If $\boldsymbol{S}$ and $\boldsymbol{S}'$ are F.O. sentences, then "$\boldsymbol{S} \wedge \boldsymbol{S}'$", "$\boldsymbol{S} \vee \boldsymbol{S}'$" and "$\boldsymbol{S} \supset \boldsymbol{S}'$" are F.O. sentences.

6. If $\boldsymbol{S}$ is a F.O. sentence and x is an individual variable then "$\forall x \boldsymbol{S}$" and "$\exists x \boldsymbol{S}$" are F.O. sentences.

(More generally, we shall call *sentences* the formulas of type o, even if they are not F.O. sentences.) A *F.O. theory* is a set of F.O. sentences. A *F.O. formula* is either a F.O. term of a F.O. sentence. A context $\boldsymbol{S}^x$ is called a *F.O. context* when $\boldsymbol{S}$ is a F.O. sentence and x is an individual variable.

A *F.O. vocabulary* is a set of predicate symbols and function symbols. In the rest of this section, $\mathcal{V}$ will denote an arbitrary F.O. vocabulary.

We define a *$\mathcal{V}$-F.O. term* (resp. *sentence, formula*) as a F.O. term (resp. sentence, formula) whose predicate symbols and function symbols are part of the vocabulary $\mathcal{V}$. A *$\mathcal{V}$-F.O. theory* is a set of $\mathcal{V}$-F.O.formulas, and $\boldsymbol{S}^{x}$ is a $\mathcal{V}$-F.O. context iff $\boldsymbol{S}$ is a $\mathcal{V}$-F.O. sentence and x an individual variable.

The *F.O. logical constants* are the logical constants which appear in F.O. sentences, i.e. false_{o}, not_{oo}, and_{ooo}, or_{ooo}, implies_{ooo}, $\text{forall}_{\iota(o\iota)}$ and $\text{exists}_{\iota(o\iota)}$. The notion of $\beta\gamma$-nf provides the following characterization of $\mathcal{V}$-F.O. formulas in terms of the free symbols that they contain:

Theorem 2.1 *The $\mathcal{V}$-F.O. formulas are the formulas of atomic type in $\beta\gamma$-nf which have no free symbols other than:*

1. *Predicate symbols and function symbols of $\mathcal{V}$.*
2. *F.O. logical constants.*
3. *Individual constants and variables.*

PROOF. Let S be the set of symbols consisting of: the symbols of $\mathcal{V}$; the F.O. logical constants; the individual constants and variables. Then, by definition B.5, the $\mathcal{V}$-F.O. formulas are the standard formulas of atomic type generated by S. Therefore, by theorem B.16, they are the formulas of atomic type in $\beta\gamma$-nf whose free symbols are elements of S. □

2.4.2 First-order inference

Having identified the formulas of F.O. logic with certain formulas of H.O.L., the standard natural deduction formulation of classical F.O.L. with equality, for a given F.O. vocabulary $\mathcal{V}$, consists of the inference rules listed below. The metavariables occurring in each rule carry with them the following implicit conditions:

1. Γ, Γ', Γ'' are $\mathcal{V}$-F.O. theories.
2. $\boldsymbol{A}$, $\boldsymbol{B}$ are $\mathcal{V}$-F.O. terms.
3. $\mathcal{C}$ is a simple $\mathcal{V}$-F.O. context.
4. $\boldsymbol{P}$, $\boldsymbol{Q}$, $\boldsymbol{R}$ are $\mathcal{V}$-F.O. sentences.

5. x, y are F.O. variables.

Inference rules:

A. *Reflexivity of* $\vdash$.

$$\boldsymbol{P} \vdash \boldsymbol{P}$$

B. *Monotonicity of* $\vdash$.

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P}}$$

C. *Substitutivity of equality.*

$$\frac{\Gamma \vdash \mathcal{C}[\boldsymbol{A}] \qquad \Gamma' \vdash \boldsymbol{A} = \boldsymbol{B}}{\Gamma \cup \Gamma' \vdash \mathcal{C}[\boldsymbol{B}]}$$

$\mathcal{C}$ does not capture any variable free in Γ'

D. *Reflexivity of equality.*

$$\vdash \boldsymbol{A} = \boldsymbol{A}$$

E. $\neg$-*Introduction*

$$\frac{\Gamma, \boldsymbol{P} \vdash \bot}{\Gamma \vdash \neg \boldsymbol{P}}$$

F. $\neg$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \qquad \Gamma' \vdash \neg \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \bot}$$

G. $\wedge$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P} \qquad \Gamma' \vdash \boldsymbol{Q}}{\Gamma \cup \Gamma' \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}$$

H. $\wedge$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{Q}} \qquad \frac{\Gamma \vdash \boldsymbol{P} \wedge \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{Q}}$$

I. $\vee$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q}} \qquad \frac{\Gamma \vdash \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q}}$$

J. $\vee$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \vee \boldsymbol{Q} \qquad \Gamma', \boldsymbol{P} \vdash \boldsymbol{R} \qquad \Gamma'', \boldsymbol{Q} \vdash \boldsymbol{R}}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash \boldsymbol{R}}$$

K. $\supset$-*Introduction*

$$\frac{\Gamma, \boldsymbol{P} \vdash \boldsymbol{Q}}{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q}}$$

L. $\supset$-*Elimination*

$$\frac{\Gamma \vdash \boldsymbol{P} \supset \boldsymbol{Q} \qquad \Gamma' \vdash \boldsymbol{P}}{\Gamma \cup \Gamma' \vdash \boldsymbol{Q}}$$

M. *Contradiction*

$$\frac{\Gamma, \neg \boldsymbol{P} \vdash \bot}{\Gamma \vdash \boldsymbol{P}}$$

N. $\forall$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}}{\Gamma \vdash \forall \boldsymbol{x} \boldsymbol{P}} \quad \boldsymbol{x} \text{ not free in } \Gamma$$

P. $\forall$-*Elimination*

$$\frac{\Gamma \vdash \forall \boldsymbol{x} \boldsymbol{P}}{\Gamma \vdash \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{A}}} \quad \boldsymbol{A} \text{ free for } \boldsymbol{x} \text{ in } \boldsymbol{P}$$

Q. $\exists$-*Introduction*

$$\frac{\Gamma \vdash \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{A}}}{\Gamma \vdash \exists \boldsymbol{x} \boldsymbol{P}} \quad \boldsymbol{A} \text{ free for } \boldsymbol{x} \text{ in } \boldsymbol{P}$$

R. $\exists$-*Elimination*

$$\frac{\Gamma \vdash \exists \boldsymbol{x} \boldsymbol{P} \qquad \Gamma', \boldsymbol{P}^{\boldsymbol{x}}_{\boldsymbol{y}} \vdash \boldsymbol{Q}}{\Gamma \cup \Gamma' \vdash \boldsymbol{Q}} \quad \begin{array}{l} \boldsymbol{y} \text{ adequate to } \boldsymbol{P}^{\boldsymbol{x}}; \\ \boldsymbol{y} \text{ not free in } \Gamma' \text{ or } \boldsymbol{Q} \end{array}$$

We write $\Gamma \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$ to indicate that a $\mathcal{V}$-F.O. formula $\boldsymbol{P}$ follows from a $\mathcal{V}$-F.O. theory Γ by the above inference rules.

All these rules can be found among the inference rules of H.O.L. (To be precise, when considered as relations among sequents, they are *restrictions* of H.O.L. rules. Indeed they are all among the primitive rules of H.O.L. listed in section 2.3.6, except reflexivity of equality which is a derived rule as noted in section 2.3.8. Hence H.O.L. is an *extension* of F.O.L. That is, Γ being a $\mathcal{V}$-F.O. theory and $\boldsymbol{P}$ a $\mathcal{V}$-F.O. sentence, if $\Gamma \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$ then $\Gamma \vdash \boldsymbol{P}$. The converse is also true:

Theorem 2.2 ((conservative extension)) *H.O.L. is a conservative extension of F.O.L. That is,* Γ *being a* $\mathcal{V}$*-F.O. theory and* $\boldsymbol{P}$ *a* $\mathcal{V}$*-F.O. sentence,* $\Gamma \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$ *iff* $\Gamma \vdash \boldsymbol{P}$.

PROOF. See appendix D. □

2.5 Set theory within H.O.L.

2.5.1 The power of set theory within H.O.L.

The reason for stating theorem 2.2 and giving a careful proof of it in appendix D is that it justifies the development of set theory within H.O.L. rather than F.O.L. Indeed theorem 2.2 can be applied to any version of set theory formulated in the framework of F.O.L., Γ being the set of axioms and $\mathcal{V}$ the vocabulary of the theory. For example, Γ could be the set of axioms of Zermelo-Fränkel set theory, which we shall call ZF; $\mathcal{V}$ would then consist of the single binary predicate symbol $\text{in}_{o\iota\iota}$ (with "$\boldsymbol{A} \in \boldsymbol{B}$" as a shorthand for "$\text{in}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$"). From now on, to make things definite, we shall focus on ZF, and we shall let $\mathcal{V} = \{\text{"in}_{o\iota\iota}\text{"}\}$. However, most other versions of set theory could be used instead of ZF. In section 2.5.6 we discuss the issue of finite vs infinite axiomatizations.

The combination of ZF and H.O.L. appears to be, at first glance, an awesome logistic system. Theorem 2.2 tells us, however, that it is not any more powerful than ordinary ZF within F.O.L.: any set theoretic result derivable from ZF by higher-order means (i.e. any $\mathcal{V}$-F.O. sentence $\boldsymbol{P}$ such that $\text{ZF} \vdash \boldsymbol{P}$) is also derivable from ZF within F.O.L. (i.e. is such that $\text{ZF} \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$). The F.O. proof is of course likely to be more complicated than the H.O. proof, and in fact, since the proof of theorem 2.2 is non-constructive, no means of finding the F.O. proof from the H.O. one are provided. (The problem of finding a constructive proof is left open.)

A corollary of theorem 2.2 is that ZF (or any other version of set theory) within H.O.L. is *relatively consistent* with respect to ZF within F.O.L. If ZF within H.O.L. were inconsistent then $\bot$ would be provable:

$$\text{ZF} \vdash \bot$$

But $\bot$ would then be derivable from ZF in F.O.L.:

$$\text{ZF} \vdash_{\mathcal{V}\text{-F.O.}} \bot$$

So ZF within F.O.L. would itself be contradictory. Therefore developing ZF within H.O.L. is *safe*: no new contradiction can thus be introduced.

In the case study (section 4.3) we shall see how higher-order reasoning can be used to simplify a proof; and in the conclusion we shall point out some exciting possibilities which are opened by the availablity of higher-order means.

But our motivation for developing set theory within H.O.L. was the representation of mathematical notations. We already know how to represent the *logical* notations (equality, the connectives, the quantifiers and the description operators) and the notation of set-membership. We shall refer to these as the *primitive* notations. We shall now describe our method for representing additional, non-primitive notations.

2.5.2 Method for representing notations

We begin by showing how the method works on four examples, one for each class of notations obtained by distinguishing term constructors vs. sentence constructors, and variable-binding vs. non-variable-binding notations.

1. The notation

$$\{x \mid \boldsymbol{P}\}, \tag{2.8}$$

where x is a F.O. variable and $\boldsymbol{P}$ is a sentence, is a variable-binding term constructor. As anticipated in section 2.1, we consider it as a shorthand for

$$\mathrm{set}_{\iota(o\iota)}\ \lambda x \boldsymbol{P}. \tag{2.9}$$

The transformation from (2.8) to (2.9) allows us to represent the notation in the formal language, but it does not explain its meaning. The meaning is expressed by an axiom involving the constant $\mathrm{set}_{\iota(o\iota)}$. H.O.L. allows us to use an axiom of the form "$\mathrm{set}_{\iota(o\iota)} = \ldots$", namely:

$$\mathrm{set}_{\iota(o\iota)} = \lambda p_{o\iota} \mu s \forall x (x \in s \equiv p_{o\iota}\ x) \tag{2.10}$$

Using this axiom we can "expand" the notation (2.8) as follows. Let Σ be a theory consisting of (2.10) and other such object-language definitions. Then:

$$\Sigma \vdash \mathrm{set}_{\iota(o\iota)} = \lambda p_{o\iota} \mu s \forall x (x \in s \equiv p_{o\iota}\ x).$$

By reflexivity and substitutivity of equality:

$$\Sigma \vdash \{\boldsymbol{x} \mid \boldsymbol{P}\} = (\lambda p_{o\iota} \mu s \forall x (x \in s \equiv p_{o\iota}\, x))\ \lambda \boldsymbol{x} \boldsymbol{P}.$$

We rename x (the actual letter "x" used as a variable in the axiom) to $\boldsymbol{x}$ (whatever variable is used in the role of the parameter $\boldsymbol{x}$ of the notation—observe the boldface), and s (the letter "s" used as a variable in the axiom) to some variable $\boldsymbol{s}$ (observe the boldface) distinct from $\boldsymbol{x}$ and not free in $\boldsymbol{P}$:

$$\Sigma \vdash \{\boldsymbol{x} \mid \boldsymbol{P}\} = (\lambda p_{o\iota} \mu \boldsymbol{s} \forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{s} \equiv p_{o\iota}\, \boldsymbol{x}))\ \lambda \boldsymbol{x} \boldsymbol{P}.$$

By the rule of β-conversion (and substitutivity of equality):

$$\Sigma \vdash \{\boldsymbol{x} \mid \boldsymbol{P}\} = \mu \boldsymbol{s} \forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{s} \equiv (\lambda \boldsymbol{x} \boldsymbol{P})\, \boldsymbol{x}).$$

By β-conversion again:

$$\Sigma \vdash \{\boldsymbol{x} \mid \boldsymbol{P}\} = \mu \boldsymbol{s} \forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P}). \tag{2.11}$$

In section 2.5.4 we shall use the axiom of extensionality to derive:

$$\mathrm{ZF} \cup \Sigma \vdash \exists \boldsymbol{s} \forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P}) \supset \forall \boldsymbol{x} (\boldsymbol{x} \in \{\boldsymbol{x} \mid \boldsymbol{P}\} \equiv \boldsymbol{P}) \tag{2.12}$$

from (2.11).

2. The notation "$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$", where $\boldsymbol{x}$ is a F.O. variable, $\boldsymbol{E}$ a term and $\boldsymbol{P}$ a sentence, is a variable-binding sentence constructor; it binds $\boldsymbol{x}$ in $\boldsymbol{P}$. We represent it by the formula:

$$\mathrm{forall}_{o(o\iota)\iota}\, \boldsymbol{E}\ \lambda \boldsymbol{x} \boldsymbol{P}$$

(Notice that $\mathrm{forall}_{o(o\iota)}$ and $\mathrm{forall}_{o(o\iota)\iota}$ are two different constants.) The constant $\mathrm{forall}_{o(o\iota)\iota}$ is defined by the axiom:

$$\mathrm{forall}_{o(o\iota)\iota} = \lambda e \lambda p_{o\iota} \forall x (x \in e \supset p_{o\iota}\, x) \tag{2.13}$$

If Σ contains (2.13), by reflexivity and substitutivity of equality:

$$\Sigma \vdash (\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P} \equiv (\lambda e \lambda p_{o\iota} \forall x (x \in e \supset p_{o\iota}\, x))\ \boldsymbol{E}\ \lambda \boldsymbol{x} \boldsymbol{P}$$

We rename $p_{o\iota}$ (if necessary) to a variable $\boldsymbol{P}$ of type $o\iota$ which does not occur free in $\boldsymbol{E}$, and we rename e (if necessary) to an individual variable $\boldsymbol{e}$ other than $\boldsymbol{x}$ and x. Then we rename x to $\boldsymbol{x}$:

$$\Sigma \vdash (\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P} \equiv (\lambda \boldsymbol{e} \lambda \boldsymbol{p} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{e} \supset \boldsymbol{p}\, \boldsymbol{x}))\, \boldsymbol{E}\, \lambda \boldsymbol{x} \boldsymbol{P}$$

Now we need to assume that $\boldsymbol{x}$ *does not occur free in* $\boldsymbol{E}$. (If $\boldsymbol{x}$ is free in $\boldsymbol{E}$ the notation "$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$" is still well defined, but the derivation is blocked. To proceed with the derivation in that case we would first rename the bound variable $\boldsymbol{x}$ in "$\lambda \boldsymbol{x} \boldsymbol{P}$".) Then, by two steps of β-conversion:

$$\Sigma \vdash (\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P} \equiv \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{E} \supset (\lambda \boldsymbol{x} \boldsymbol{P})\, \boldsymbol{x})$$

And by another step of β-conversion:

$$\Sigma \vdash (\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P} \equiv \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{E} \supset \boldsymbol{P}) \tag{2.14}$$

3. The notation "$\{\boldsymbol{A}, \boldsymbol{B}\}$" constructs a term but does not bind any variable. We consider "$\{\boldsymbol{A}, \boldsymbol{B}\}$" as a shorthand for:

$$\mathrm{enum}_{\iota\iota\iota}\, \boldsymbol{A}\, \boldsymbol{B}.$$

In this we do not depart from the traditional method of introducing an additional function symbol, since $\mathrm{enum}_{\iota\iota\iota}$ is indeed a binary function symbol. H.O.L., however, allows us again to provide the axiom defining the notation in the form "$\mathrm{enum}_{\iota\iota\iota} = \ldots$":

$$\mathrm{enum}_{\iota\iota\iota} = \lambda x \lambda y \mu s \forall z(z \in s \equiv z = x \vee z = y) \tag{2.15}$$

If Σ contains (2.15), by reflexivity and substitutivity of equality:

$$\Sigma \vdash \{\boldsymbol{A}, \boldsymbol{B}\} = (\lambda x \lambda y \mu s \forall z(z \in s \equiv z = x \vee z = y))\, \boldsymbol{A}\, \boldsymbol{B}$$

We rename s and z to two distinct variables $\boldsymbol{s}$ and $\boldsymbol{z}$ which do not occur free in $\boldsymbol{A}$ or $\boldsymbol{B}$ and are distinct from x and y:

$$\Sigma \vdash \{\boldsymbol{A}, \boldsymbol{B}\} = (\lambda x \lambda y \mu \boldsymbol{s} \forall \boldsymbol{z}(\boldsymbol{z} \in \boldsymbol{s} \equiv \boldsymbol{z} = x \vee \boldsymbol{z} = y))\, \boldsymbol{A}\, \boldsymbol{B}$$

Then, by two steps of β-conversion (after renaming y if necessary):

$$\Sigma \vdash \{\boldsymbol{A}, \boldsymbol{B}\} = \mu s \forall z (z \in s \equiv z = \boldsymbol{A} \vee z = \boldsymbol{B}) \qquad (2.16)$$

In section 2.5.7 we shall use the axiom of extensionality and the pair-set axiom to derive:

$$\forall z (z \in \{\boldsymbol{A}, \boldsymbol{B}\} \equiv z = \boldsymbol{A} \vee z = \boldsymbol{B})$$

from (2.16).

4. Our fourth example is "$\boldsymbol{A} \subseteq \boldsymbol{B}$". This notation is a sentence constructor, but it does not bind any variables. We represent if by the formula:

$$\text{subset}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$$

$\text{subset}_{o\iota\iota}$ is a binary predicate symbol. Again we can define it by an equation "$\text{subset}_{o\iota\iota} = \ldots$":

$$\text{subset}_{o\iota\iota} = \lambda x \lambda y \forall z (z \in x \supset z \in y) \qquad (2.17)$$

If Σ contains (2.17):

$$\Sigma \vdash \text{subset}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B} \equiv (\lambda x \lambda y \forall z (z \in x \supset z \in y))\ \boldsymbol{A}\ \boldsymbol{B}$$

If z is a variable not free in $\boldsymbol{A}$ or $\boldsymbol{B}$, by α-conversion and β-conversion:

$$\Sigma \vdash \text{subset}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B} \equiv \forall z (z \in \boldsymbol{A} \supset z \in \boldsymbol{B})$$

In general, representing a non-primitive notation consists of two steps:

1. Choosing a formula to represent the notation in the formal system.
2. Providing an axiom which captures the meaning of the notation.

Here is a simple recipe for the first step.

The representation of a notation will always be of the form:

$$\boldsymbol{c}\ \boldsymbol{ARG}_1\ \ldots\ \boldsymbol{ARG}_n$$

($n \geq 0$) where $\boldsymbol{c}$ is a constant specifically chosen to represent the notation, the *representing constant*. When $n > 0$, $\boldsymbol{ARG}_1, \ldots \boldsymbol{ARG}_n$ are the *arguments* of the representation.

The notation has syntactic *parameters*, such as $\boldsymbol{x}$, $\boldsymbol{E}$, $\boldsymbol{P}$ in "$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$". Some of these parameters, such as $\boldsymbol{E}$ and $\boldsymbol{P}$ in the example, are *formula parameters*, while some, such as $\boldsymbol{x}$ in the example, are *variable parameters*. The notation binds each variable parameter (if any) in a formula parameter; in the example, the notation binds $\boldsymbol{x}$ in $\boldsymbol{P}$.

The collection of arguments $\boldsymbol{ARG}_1, \ldots \boldsymbol{ARG}_n$ is derived as follows from the parameters of the notation. First if there are no parameters then $n = 0$ and the representation is reduced to the representing constant $\boldsymbol{c}$. The type of $\boldsymbol{c}$ is chosen according to the syntactice role of notation; usually the notation plays the role of a term; then the type of $\boldsymbol{c}$ is ι. If there are parameters, then there is one argument $\boldsymbol{ARG}_i$ for each formula parameter. It is constructed as follows. If the notation binds no variable parameters in the formula parameter, then $\boldsymbol{ARG}_i$ is the formula parameter itself. If the notation binds one or more variable parameters in the formula parameter, then $\boldsymbol{ARG}_i$ is obtained by abstracting with respect to each of those variables (in arbitrary order). In the notation "$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$" no variable parameter is bound in the formula parameter $\boldsymbol{E}$, while the variable parameter $\boldsymbol{x}$ is bound in the formula parameter $\boldsymbol{P}$; so the arguments are "$\boldsymbol{E}$" and "$\lambda \boldsymbol{x}\boldsymbol{P}$". Hence the representation:

$$\mathrm{forall}_{o(o\iota)\iota}\, \boldsymbol{E}\ \lambda \boldsymbol{x}\boldsymbol{P}$$

which we saw above. For a more complicated example consider the notation "$\{\boldsymbol{A}\}_{x \in B, y \in C}$" (one of those mentioned in section 2.1): $\boldsymbol{B}$ and $\boldsymbol{C}$ do not bind any variables, while $\boldsymbol{A}$ binds both $\boldsymbol{x}$ and $\boldsymbol{y}$; so the arguments are "$\boldsymbol{B}$", "$\boldsymbol{C}$" and "$\lambda \boldsymbol{x} \lambda \boldsymbol{y} \boldsymbol{A}$". Choosing $\mathrm{range}_{\iota(\iota\iota\iota)\iota\iota}$ as the representing constant we obtain the representation:

$$\mathrm{range}_{\iota(\iota\iota\iota)\iota\iota}\, \boldsymbol{B}\ \boldsymbol{C}\ (\lambda \boldsymbol{x} \lambda \boldsymbol{y} \boldsymbol{A})$$

which we proposed in section 2.1.

The type of the representing constant is determined once we have chosen the ordering of the bound variables in each argument, and the ordering of the arguments themselves. Indeed the type of $\boldsymbol{c}$ is:

$$\delta \alpha_1 \ldots \alpha_n$$

where $\alpha_1, \ldots \alpha_n$ are the types of $\boldsymbol{ARG}_1, \ldots \boldsymbol{ARG}_n$, and δ is normally ι or o (according to whether the notation plays the role of a term or a sentence respectively). The type α_i of each argument $\boldsymbol{ARG}_i$ is in turn:

$$\theta\beta_1 \ldots \beta_m$$

($m \geq 0$) where $\beta_1, \ldots \beta_m$ are the types of the variable parameters bound by the notation in the formula parameter from which $\boldsymbol{ARG}_i$ is constructed (usually $\beta_1 = \ldots = \beta_m = \iota$), and θ is the type of the formula parameter ($\theta = \iota$ if the formula parameter, $\theta = o$ if the formula parameter is a term). If the notation binds at least one variable, the type of at least one argument will be at least 1, so the type of the representing constant will be at least 2, and thus the notation will be represented by a higher-order constant. In the usual case where all the bound variables are of type ι, the order of the type of the representing constant will be exactly 2.[15]

Notice that the recipe works correctly even for the primitive notations: it can account for the internal representation of equality, the logical connectives, the quantifiers (including the quantifiers for higher-order types), the description operators (δ being a higher-order type for a higher-order description operator), and the set-membership notation.

The second step is more difficult. All that can be said in general is that, if $\boldsymbol{c}$ is the constant chosen to represent the notation, the axiom defining the notation is of the form "$\boldsymbol{c} = \boldsymbol{A}$". There is of course no recipe for composing the right-hand side $\boldsymbol{A}$. However, if help is needed, we can turn to Bourbaki [9]. Indeed, *expanding* a notation as we have done in the four examples above (by replacing the constant representing the notation with its definition, then converting to β-nf) results in the Bourbaki-style expansion of the notation, except that Bourbaki uses a selection operator rather than our description operator μ. In most definitions where Bourbaki uses his selection operator, the operator is in fact applied to a definite description, and so a description

[15]Notice that the notation may require a higher-order constant even when it does not bind any variable. This the case when one of the arguments is of type o, given that we have somewhat arbitrarily assigned the order 1 to the atomic type o. For example the term constructor: "if P then A else B" requires a higher-order representing constant, say $\text{if}_{\iota o \iota\iota}$:

"if P then A else B" for "$\text{if}_{\iota o \iota\iota}\, PAB$"

operator would do as well. (A notable exception is Bourbaki's definition of cardinality.) In those cases, it is a simple matter undo the conversion to β-nf and obtain, from Bourbaki's definition, a definition "$\boldsymbol{c} = \boldsymbol{A}$" of the constant which represents the notation in our formal system.

2.5.3 Eliminability of notations from proofs

We have extended our formal system from ZF within F.O.L. to ZF within H.O.L. to be able to accomodate notations. We have justified the move by showing that the extension is conservative. Introducing notations does not require any further extension of the *language* of the formal system, since the constants used to represent the notations, as identifiers in roman font subscripted by types, are already constants of H.O.L. However the *axioms* defining those constants do extend the formal system, and we need to show that the extension is conservative. That is, we need to show that notations are *eliminable from proofs.*

An *(object language) definition* of a constant $\boldsymbol{c}$ is a sentence of the form "$\boldsymbol{c} = \boldsymbol{A}$", where $\boldsymbol{A}$ is a formula which contains no free occurrences of variables, and no occurrences of $\boldsymbol{c}$. Given a set of definitions, the *definition graph* is the graph of the relation "is defined in terms of". That is, a pair of constants $(\boldsymbol{c}, \boldsymbol{c}')$ is an element of the definition graph iff there is a definition "$\boldsymbol{c} = \boldsymbol{A}$" where $\boldsymbol{c}'$ occurs in $\boldsymbol{A}$. A set of definitions Σ is an *abbreviation system* iff:

1. No constant has two definitions in Σ.

2. The definition graph of Σ is a noetherian relation, i.e. it has no infinite chains (see definition B.3 in appendix B, page 185).

3. No logical constant has a definition in Σ.

Notice that it is not sufficient to say that there are no cycles in the graph. For example, consider the infinite family of notations "$\{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\}$", $(n \geq 1)$ and the constants which represent them, say enum_{ι}, $\mathrm{enum}_{\iota\iota}$, $\mathrm{enum}_{\iota\iota\iota}$, etc. Obviously, each constant, of arity n, could be easily defined in terms of the constant of arity $n + 1$. Although such definitions would have an acyclic graph, they would not constitute an abbreviation system, because the graph would have an infinite chain.

Given an abbreviation system Σ, the Σ-*expansion* of a formula $\boldsymbol{A}$ is the result of starting with $\boldsymbol{A}$ and repeatedly replacing occurrences of constants defined by Σ with their definientia until there are no such occurrences left. If $\boldsymbol{A}'$ is the Σ-expansion of $\boldsymbol{A}$:

$$\Sigma \vdash \boldsymbol{A} = \boldsymbol{A}' \tag{2.18}$$

Notice that this is not quite the same notion of "expansion of a notation" that we have seen informally above. The former notion involved a conversion into β-nf. We shall say that $\boldsymbol{A}'$ is a $\Sigma\beta$-*expansion* of $\boldsymbol{A}$ iff it is a β-nf of the Σ-expansion of $\boldsymbol{A}$. We shall say that $\boldsymbol{A}'$ is a $\Sigma\beta\gamma$-*expansion* of $\boldsymbol{A}$ iff it is a $\beta\gamma$-nf of the Σ-expansion of $\boldsymbol{A}$. Clearly (2.18) still holds in these cases.

Theorem 2.3 *If Σ is an abbreviation system which does not define any constant occurring in Γ, and if $\boldsymbol{P}$ is a sentence which contains no occurrences of constants defined in Σ, then $\Gamma \cup \Sigma \vdash \boldsymbol{P}$ iff $\Gamma \vdash \boldsymbol{P}$.*

PROOF. Let $\Pi = ((\Delta_i, \boldsymbol{Q}_i))_{1 \leq i \leq n}$ be a proof of $\Gamma \cup \Sigma \vdash \boldsymbol{P}$. For every i, $1 \leq i \leq n$, let Δ_i' be the set of Σ-expansions of the axioms of $\Delta_i - \Sigma$; and let $\boldsymbol{Q}_i'$ be the Σ-expansion of $\boldsymbol{Q}_i$. Let $\Pi' = ((\Delta_i', \boldsymbol{Q}_i'))_{1 \leq i \leq n}$; we shall refer to the pairs $(\Delta_i', \boldsymbol{Q}_i')$, $1 \leq i \leq n$, as the lines of Π', even though Π' may not be a proof.

Suppose that line i of Π follows from previous lines by a rule of inference other than 1, 8, 13, 14, 16 or 21. (Rule 1 is reflexivity of $\vdash$ (i.e. $\boldsymbol{P} \vdash \boldsymbol{P}$), and rules 8, 13, 14, 16, 21 are those that discharge assumptions.) Then line i of Π' follows from the corresponding lines of Π' by the same rule.

Suppose that line i of Π follows by 8, 13, 14, 16 or 21 from previous lines, one of which is line j from which a formula $\boldsymbol{R}$ is being discharged. If $\boldsymbol{R}$ happens to be an axiom of Σ, then its Σ-expansion may not be part of the hypotheses of line j of Π'; but then a new line can be derived from line j of Π' by adding the missing hypotheses, with rule 2 as justification; the new line can be used instead of line j when justifying line i of Π'.

Suppose that line i of Π is $(\{\boldsymbol{Q}_i\}, \boldsymbol{Q}_i)$ (i.e. $\boldsymbol{Q}_i \vdash \boldsymbol{Q}_i$). If $\boldsymbol{Q}_i$ is not an axiom of Σ, then line i of Π' is $(\{\boldsymbol{Q}_i'\}, \boldsymbol{Q}_i')$, which is justified by rule 1. If $\boldsymbol{Q}_i$ is an axiom of Σ, then line i of Π' is $(\emptyset, \boldsymbol{Q}_i')$. But then $\boldsymbol{Q}_i$ is "$\boldsymbol{c} = \boldsymbol{A}$", and $\boldsymbol{c}$ and $\boldsymbol{A}$ have the same Σ-expansion $\boldsymbol{A}'$. Therefore $\boldsymbol{Q}_i'$ is "$\boldsymbol{A}' = \boldsymbol{A}'$". Reflexivity of equality is not a primitive rule of inference in the system, but

it can be derived, as we saw at the end of section 2.4.2; so line i of Π' can be proved.

Therefore by inserting additional lines and proof fragments, Π' can be made into a proof. The last line of Π is $(\Gamma \cup \Sigma, \boldsymbol{P})$, where neither $\boldsymbol{P}$ nor the axioms of Γ contain any occurrences of constants defined by Σ. Therefore the last line of Π' is $(\Gamma, \boldsymbol{P})$; hence:

$$\Gamma \vdash \boldsymbol{P}.$$

□

In particular, let $\Gamma = \mathrm{ZF}$, let Σ be an abbreviation system which does not define $\mathrm{in}_{o\iota\iota}$, and let $\boldsymbol{P}$ be a F.O. sentence containing no non-logical constants other than $\mathrm{in}_{o\iota\iota}$. If $\mathrm{ZF} \cup \Sigma \vdash \boldsymbol{P}$ then $\mathrm{ZF} \vdash \boldsymbol{P}$, and by theorem 2.2, $\mathrm{ZF} \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$.

2.5.4 Some axioms and notations of ZF

Extensionality

Most axioms of ZF assert the existence of certain sets, the uniqueness of which follows from the axiom of extensionality. The basic notations of ZF denote those sets. We are now going to review the axioms of ZF and show how each basic notation can be represented in our formal system by a constant and an axiom defining it. We begin with Extensionality, Empty-set, and Replacement.

Extensionality.

$$\forall x \forall y (\forall z (z \in x \equiv z \in y) \supset x = y) \tag{2.19}$$

The use of (2.19) in connection with notations defined in terms of the description operator can be illustrated with "$\{x \mid \boldsymbol{P}\}$". This notation was introduced in section 2.5.2, page 43, where we proved (2.11) and promised to prove (2.12) using the axiom of extensionality.

The existence of a set s such that "$\forall x(x \in s \equiv \boldsymbol{P})$" for an arbitrary sentence $\boldsymbol{P}$ does not follow from the axioms of ZF if ZF is consistent; so it is an explicit condition in (2.12). The proof of (2.12) is as follows. Let $\boldsymbol{R}$ be the formula "$\forall x(x \in s \equiv \boldsymbol{P})$" and let t be a variable distinct from s and x and not free in $\boldsymbol{P}$. Then $\boldsymbol{R}^s_t$ is "$\forall x(x \in t \equiv \boldsymbol{P})$" and we have:

$$\boldsymbol{R} \wedge \boldsymbol{R}^s_t \ \vdash \ \forall x(x \in s \equiv \boldsymbol{P}) \wedge \forall x(x \in t \equiv \boldsymbol{P})$$

$$\begin{aligned} \boldsymbol{R} \wedge \boldsymbol{R}_t^s &\vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P}) \\ \boldsymbol{R} \wedge \boldsymbol{R}_t^s &\vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{t} \equiv \boldsymbol{P}) \\ \boldsymbol{R} \wedge \boldsymbol{R}_t^s &\vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{x} \in \boldsymbol{t}) \end{aligned}$$

Hence by the axiom of extensionality:

$$\begin{aligned} \mathrm{ZF}, \boldsymbol{R} \wedge \boldsymbol{R}_t^s &\vdash \boldsymbol{s} = \boldsymbol{t} \\ \mathrm{ZF} &\vdash \boldsymbol{R} \wedge \boldsymbol{R}_t^s \supset \boldsymbol{s} = \boldsymbol{t} \\ \mathrm{ZF} &\vdash \forall \boldsymbol{s} \forall \boldsymbol{t}(\boldsymbol{R} \wedge \boldsymbol{R}_t^s \supset \boldsymbol{s} = \boldsymbol{t}) \end{aligned}$$

By rule 22 (!-introduction):

$$\mathrm{ZF} \vdash !\boldsymbol{s}\boldsymbol{R}$$

i.e.

$$\mathrm{ZF} \vdash !\boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P})$$

Let $\boldsymbol{Q}$ be the existence assumption "$\exists \boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P})$".

$$\boldsymbol{Q} \vdash \exists \boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P})$$

By rule 24 ($\exists!$-Introduction):

$$\mathrm{ZF}, \boldsymbol{Q} \vdash \exists! \boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P})$$

By rule 26 (μ-Introduction):

$$\mathrm{ZF}, \boldsymbol{Q} \vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \mu \boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P}) \equiv \boldsymbol{P})$$

Then by (2.11) (which asserts that "$\{\boldsymbol{x} \mid \boldsymbol{P}\}$" is equal to its $\Sigma\beta$-expansion "$\mu \boldsymbol{s} \forall \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{P})$"):

$$\mathrm{ZF} \cup \Sigma, \boldsymbol{Q} \vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \{\boldsymbol{x} \mid \boldsymbol{P}\} \equiv \boldsymbol{P})$$

From this we obtain (2.12) by discharging $\boldsymbol{Q}$.

Empty-set

The Empty-set axiom asserts the existence of a set with no elements:

$$\exists s \forall x \neg(x \in s)$$

On the other hand, by Extensionality, we know there is at most one such set:

$$\mathrm{ZF} \vdash !s \forall x \neg(x \in s)$$

Hence

$$\mathrm{ZF} \vdash \exists! s \forall x \neg(x \in s).$$

By μ-Introduction:

$$\mathrm{ZF} \vdash \forall x \neg(x \in \mu s \forall x \neg(x \in s)) \tag{2.20}$$

We introduce the notation

$$\text{“}\emptyset\text{”} \text{ for “emptyset}_\iota\text{”}$$

with the definition:

$$\text{emptyset}_\iota = \mu s \forall x \neg(x \in s). \tag{2.21}$$

If Σ contains (2.21), from (2.20) and (2.21):

$$\mathrm{ZF} \cup \Sigma \vdash \forall x \neg(x \in \emptyset).$$

Replacement

A formula is an instance of the axiom schema of replacement iff it is of the form

$$\begin{aligned} \forall z_1 \ldots \forall z_n (\ & \forall x \forall y \forall u (\boldsymbol{P} \wedge \boldsymbol{P}^y_u \supset y = u) \supset \\ & \forall e \exists s \forall y (y \in s \equiv \exists x (x \in e \wedge \boldsymbol{P}))) \end{aligned} \tag{2.22}$$

where:

1. x and y are distinct individual variables,
2. $\boldsymbol{P}$ is a $\mathcal{V}$-F.O. sentence,
3. $z_1 \ldots z_n$ are the individual variables other than x and y which occur free in $\boldsymbol{P}$,

4. u is an individual variable other x and y which is adequate to $\boldsymbol{P}^y$, and

5. e and s are distinct F.O. variables, other than x and y, which do not occur free in $\boldsymbol{P}$.

We shall refer to $\boldsymbol{P}$ as the *parameter* of the axiom schema.

Replacement has two important special cases. The first one is obtained by taking $\boldsymbol{P}$ of the form "$y = x \wedge \boldsymbol{Q}$", where $\boldsymbol{Q}$ is a $\mathcal{V}$-F.O. sentence with no occurrences of y (of any kind). After simplification we get:

$$\mathrm{ZF} \vdash \forall z_1 \ldots \forall z_n (\forall e \exists s \forall y (y \in s \equiv y \in e \wedge \boldsymbol{Q}^x_y))$$

Nothing prevents us now from calling x what we have been calling y, and $\boldsymbol{P}$ what we have been calling $\boldsymbol{Q}^x_y$. We recognize then the *theorem schema of separation*:

$$\mathrm{ZF} \vdash \forall z_1 \ldots \forall z_n \forall e \exists s \forall x (x \in s \equiv x \in e \wedge \boldsymbol{P}) \tag{2.23}$$

where the conditions are as follows:

1. x is an individual variable,

2. $\boldsymbol{P}$ is a $\mathcal{V}$-F.O. sentence,

3. $z_1 \ldots z_n$ are the individual variables other than x which occur free in $\boldsymbol{P}$,

4. e and s are distinct F.O. variables, other than x, which do not occur free in $\boldsymbol{P}$.

We shall refer to $\boldsymbol{P}$ as the *parameter* of the theorem schema.

Separation allows us to introduce the notation "$\{x \in \boldsymbol{E} \mid \boldsymbol{P}\}$", where x is an individual variable, $\boldsymbol{E}$ a $\mathcal{V}$-F.O. term, and $\boldsymbol{P}$ a $\mathcal{V}$-F.O. sentence. We represent it as:

$$\mathrm{subset}_{\iota(o\iota)\iota}\, \boldsymbol{E}\, \lambda x \boldsymbol{P}.$$

(Notice that $\mathrm{subset}_{\iota(o\iota)\iota}$ is a different constant from the constant $\mathrm{subset}_{o\iota\iota}$ which we used to represent the notation "$\boldsymbol{A} \subseteq \boldsymbol{B}$".) We define $\mathrm{subset}_{\iota(o\iota)\iota}$ with the axiom:

$$\mathrm{subset}_{\iota(o\iota)\iota} = \lambda e \lambda p_{o\iota} \mu s \forall x (x \in s \equiv x \in e \wedge p_{o\iota}\, x) \tag{2.24}$$

Let Σ be an abbreviation system containing this definition. Assume x is not free in $\boldsymbol{E}$, and let s be a F.O. variable other than x which does not occur free in $\boldsymbol{E}$ or $\boldsymbol{P}$. Then by substitutivity of equality and conversion:

$$\Sigma \vdash \{x \in \boldsymbol{E} \mid \boldsymbol{P}\} = \mu s \forall x (x \in s \equiv x \in \boldsymbol{E} \wedge \boldsymbol{P}). \tag{2.25}$$

By rule 14 ($\forall$-Elimination) applied to Separation (2.23):

$$\mathrm{ZF} \vdash \exists s \forall x (x \in s \equiv x \in \boldsymbol{E} \wedge \boldsymbol{P})$$

and by Extensionality:

$$\mathrm{ZF} \vdash !s \forall x (x \in s \equiv x \in \boldsymbol{E} \wedge \boldsymbol{P})$$

Therefore by rule 24 ($\exists!$-Introduction):

$$\mathrm{ZF} \vdash \exists! s \forall x (x \in s \equiv x \in \boldsymbol{E} \wedge \boldsymbol{P})$$

And by rule 26 (μ-Introduction), followed by substitution of the left-hand side of (2.25) for the right-hand side:

$$\mathrm{ZF} \cup \Sigma \vdash \forall x (x \in \{x \in \boldsymbol{E} \mid \boldsymbol{P}\} \equiv x \in \boldsymbol{E} \wedge \boldsymbol{P}). \tag{2.26}$$

The second special case of Replacement is obtained when $\boldsymbol{P}$ is of the form "$y = \boldsymbol{A}$", where $\boldsymbol{A}$ is a $\mathcal{V}$-F.O. term having no free occurrences of y. After simplification we obtain the following theorem schema:

$$\mathrm{ZF} \vdash \forall z_1 \ldots \forall z_n \forall e \exists s \forall y (y \in s \equiv \exists x (x \in e \wedge y = \boldsymbol{A})) \tag{2.27}$$

where the conditions are as follows:

1. x and y are distinct individual variables,
2. $\boldsymbol{A}$ is a $\mathcal{V}$-F.O. term having no free occurrences of y,
3. $z_1 \ldots z_n$ are the individual variables other than x which occur free in $\boldsymbol{A}$,
4. e and s are distinct indidvidual variables, other than x and y, which do not occur free in $\boldsymbol{A}$.

We shall refer to $\boldsymbol{A}$ as the *parameter* of the theorem schema.

This theorem schema allows us to introduce the notation "$\{\boldsymbol{A}\}_{x\in E}$" ("the set consisting of everything of the form $\boldsymbol{A}$ for $x \in \boldsymbol{E}$"), where $\boldsymbol{A}$ and $\boldsymbol{E}$ are $\mathcal{V}$-F.O. terms, and x is an individual variable. We represent it by:

$$\text{range}_{\iota(\iota\iota)\iota}\ \boldsymbol{E}\ \lambda x \boldsymbol{A}.$$

with $\text{range}_{\iota(\iota\iota)\iota}$ defined by:

$$\text{range}_{\iota(\iota\iota)\iota} = \lambda e \lambda a_{\iota\iota} \mu s \forall y (y \in s \equiv \exists x (x \in e \wedge y = a_{\iota\iota}\, x) \tag{2.28}$$

Assume that x is not free in $\boldsymbol{E}$, and let s, y be distinct individual variables, other than x, which do not occur free in $\boldsymbol{E}$ or $\boldsymbol{A}$. If Σ contains definition (2.28), by substitutivity of equality and conversion:

$$\Sigma \vdash \{\boldsymbol{A}\}_{x\in E} = \mu s \forall y (y \in s \equiv \exists x (x \in \boldsymbol{e} \wedge y = \boldsymbol{A})) \tag{2.29}$$

Then by extensionality and theorem schema (2.27):

$$\text{ZF} \cup \Sigma \vdash \forall y (y \in \{\boldsymbol{A}\}_{x\in E} \equiv \exists x (x \in \boldsymbol{e} \wedge y = \boldsymbol{A})). \tag{2.30}$$

2.5.5 The axiom schema problem

In the statement of the axiom schema of replacement we were careful to specify that the parameter $\boldsymbol{P}$ must be a $\mathcal{V}$-F.O. formula. The reason for this restriction is clear: otherwise we would be allowing instances of the schema which are not $\mathcal{V}$-F.O. sentences and therefore cannot be axioms of ZF. In moving from ZF within F.O.L. to ZF within H.O.L. we have enriched the logical framework, but we do not wish to change the set of axioms of the theory ZF, since the resulting formal system might not then be a conservative extension of the original one.

The restriction is not an unimportant one. Formulas containing mathematical notations (other than the notations for set-membership, F.O. quantification, and the logical connectives) are not $\mathcal{V}$-F.O. formulas; therefore, in principle, they cannot be used as parameters of Replacement (2.22), Separation (2.23) and theorem schema (2.27), and they cannot be used in the role of $\boldsymbol{P}$ in (2.26) or in the role of $\boldsymbol{A}$ in (2.30). But Replacement and Separation are essential in set theoretic arguments; if we were to actually rule out the use

of mathematical notations in such contexts we would render the notations practically useless, and we would be going against mathematical practice.

Fortunately, if a sentence $\boldsymbol{P}'$ is logically equivalent to a sentence $\boldsymbol{P}$ (under hypotheses having no free variables), then by substitutivity of equality (equivalence, in this case), $\boldsymbol{P}'$ can be used instead of $\boldsymbol{P}$. And if a term $\boldsymbol{A}'$ is equal to $\boldsymbol{A}$, it can be used instead of $\boldsymbol{A}$. We thus have to show that formulas (terms or sentences) containing mathematical notations are equivalent to $\mathcal{V}$-F.O. formulas; in other words, we have to show that notations are *eliminable from formulas.*

It should be noted that notations would *not* be eliminable if we defined them in terms of a selection operator instead of the description operator. A selection operator could be easily added to our system, as a family of constants $\text{select}_{\alpha(o\alpha)}$ with the notation "$\varepsilon \boldsymbol{x}\boldsymbol{P}$" for "$\text{select}_{\alpha(o\alpha)}\, \lambda \boldsymbol{x}\boldsymbol{P}$" and the rule of inference:

27. *ε-Introduction*

$$\frac{\Gamma \vdash \exists \boldsymbol{x}\boldsymbol{P}}{\Gamma \vdash \boldsymbol{P}^{\boldsymbol{x}}_{\varepsilon \boldsymbol{x}\boldsymbol{P}}} \qquad \text{"}\varepsilon \boldsymbol{x}\boldsymbol{P}\text{" free for } \boldsymbol{x} \text{ in } \boldsymbol{P}$$

The resulting system, ZF within H.O.L.+ε, would still be a conservative extension of ZF within H.O.L. But if we allowed occurrences of the selection operator in the parameter of Replacement, either directly or indirectly by allowing notations defined in terms of it, we would strengthen the system. In particular the axiom of choice would become provable.[16] (It follows from Replacement by letting the parameter $\boldsymbol{P}$ be the sentence

$$y = (x, \varepsilon z(z \in x))$$

with x and y in the roles of $\boldsymbol{x}$ and $\boldsymbol{y}$.) This is precisely what happens in Bourbaki's system (recall section 1.5.2).

Notice that no harm is done by adding the selection operator to the system, as explained, as long as it is not used in the parameter of Replacement. The presence of the selection operator is equivalent to the *type theoretic* axiom of choice [2]. It is interesting to note that in our formal system there is room for both the set theoretic axiom of choice, and its type theoretic counterpart. We have the option of using none, both, or either one.

[16] It is not provable in ZF if we assume that ZF is consistent.

Versions of set theory which distinguish between classes and sets are finitely axiomatizable. The axiom schema problem does not go away in those systems. Although there is no *axiom schema* of replacement, there is a *theorem schema* of class existence [40, proposition 4.4] which gives rise to the same difficulty.

2.5.6 Eliminability of notations from formulas

In fact, notations defined in terms of the description operator are *not* eliminable from formulas in the system that we have so far.[17] But they are eliminable in the slightly stronger system obtained by adding the following additional introduction rule for μ, which covers the case when unique existence does not hold, by stipulating that $\mu x \boldsymbol{P}$ is then the empty set. (Notice that this rule is only for the description operator $\text{the}_{\iota(o\iota)}$; we need no additional introduction rule for $\text{the}_{\alpha(o\alpha)}$ when α is other than ι.)

26a. *μ-Introduction-bis*

$$\frac{\Gamma \vdash \neg\exists! x \boldsymbol{P}}{\Gamma \vdash \forall x \neg(x \in \mu x \boldsymbol{P})} \qquad x \text{ individual variable only}$$

From now on we shall write $\vdash$ for the deducibility relation in this stronger system.

The following theorem states that the stronger system is a conservative extension of F.O.L. in the presence of ZF.

Theorem 2.4 *Let Γ be a $\mathcal{V}$-F.O. theory and $\boldsymbol{P}$ a $\mathcal{V}$-F.O. sentence. If*

$$\Gamma \cup \mathrm{ZF} \vdash \boldsymbol{P}$$

then

$$\Gamma \cup \mathrm{ZF} \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$$

[17] ...since our description operator is even more inderminate than the selection operator. That is, if the description operator was eliminable, it would also be eliminable in the stronger system obtained by turning it into a selection operator (instead of adding a separate selection operator). But then the selection operator could be used in the paramater of Replacement without strengthening the system, which is not the case, assuming the consistency of ZF.

PROOF. An *interpretation with empty set* is an interpretation $(\mathcal{D}, \phi)$ (in the H.O. sense of section 2.3.4) with domain of individuals $\mathcal{D}_\iota = M$ such that $\phi(\text{in}_{o\iota})$ is a function $f \in \mathcal{D}_{o\iota\iota}$ for which there exists a unique $z \in M$ such that, for every $x \in M$,

$$f(x)(z) = \text{F}.$$

The unique element z is called the *empty set in the domain of individuals* of the interpretation. A *logical interpretation with empty-set default* is a logical interpretation with empty set where the denotation of $\text{the}_{\iota(o\iota)}$ is a function $g \in \mathcal{D}_{\iota(o\iota)}$ which maps to the empty set z every function $h \in \mathcal{D}_{o\iota}$ which takes the value T on zero or more than one individuals of M.

Clearly, the system obtained by adding rule 26a is sound for logical interpretations with empty-set default. That is, if $\Gamma \vdash \boldsymbol{P}$ with the new sense of $\vdash$, and $\mathcal{I}$ is a model of Γ with empty-set default, then $\mathcal{I}$ satisfies $\boldsymbol{P}$. It is also clear that any $\mathcal{V}$-F.O. interpretation which is a model of ZF in the F.O. sense has a H.O. extension with empty-set default.

The proof then proceeds as the proof of theorem 2.2 given in appendix D. Assume $\Gamma \cup \text{ZF} \vdash \boldsymbol{P}$ where Γ is a $\mathcal{V}$-F.O. theory and $\boldsymbol{P}$ is a $\mathcal{V}$-F.O. sentence. Let $\mathcal{I}$ be a $\mathcal{V}$-F.O. interpretation which is a model of $\Gamma \cup \text{ZF}$. Since $\mathcal{I}$ is a model of ZF it has a H.O. extension $\mathcal{J}$ which is a logical interpretation with empty-set default. By lemma D.3, $\mathcal{J}$ is a H.O. model of $\Gamma \cup \text{ZF}$. By the soundness of the new inference system for logical interpretations with empty-set default, $\mathcal{J}$ satisfies $\boldsymbol{P}$, in the H.O. sense. By lemma D.3 again, $\mathcal{I}$ satisfies $\boldsymbol{P}$ in the F.O. sense. Thus every $\mathcal{V}$-F.O. interpretation which is a model of $\Gamma \cup \text{ZF}$ satisfies $\boldsymbol{P}$, i.e. $\Gamma \cup \text{ZF} \models_{\text{F.O.}} \boldsymbol{P}$. By the completeness of F.O.L. (theorem D.1), $\Gamma \cup \text{ZF} \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$. □

The following theorem states that notations are also eliminable from proofs in the stronger system.

Theorem 2.5 *If Σ is an abbreviation system which does not define any constant occurring in a theory Γ, and which does not define $\text{in}_{o\iota\iota}$, and if $\boldsymbol{P}$ is a sentence which contains no occurrences of constants defined in Σ, then $\Gamma \cup \Sigma \vdash \boldsymbol{P}$ iff $\Gamma \vdash \boldsymbol{P}$.*

PROOF. Assume $\Gamma \cup \Sigma \vdash \boldsymbol{P}$, and let Π be a proof of the sequent $(\Gamma \cup \Sigma, \boldsymbol{P})$. We construct Π' as in the proof of theorem 2.3 given above. If line i of Π follows from previous lines by rule 26a, of by a rule of inference other than 1, 8, 13, 14, 16 or 21, then line i of Π' follows from the corresponding lines of

Π' by the same rule. This is because $\text{in}_{o\iota\iota}$ is not one of the constants defined by Σ. The proof then proceeds as the proof of theorem 2.3. □

Now we show that notations are eliminable from formulas in the stronger system.

The *residual vocabulary* of an abbreviation system Σ is the set of constants occurring in Σ other than those defined by Σ. A *set-theoretic abbreviation system* is an abbreviation system whose residual vocabulary consists only of F.O. logical constants, $\text{in}_{o\iota\iota}$, $\text{the}_{\iota(o\iota)}$ and individual constants. A μ-sentence is a sentence with no occurrences of constants other than the F.O. logical constants, $\text{in}_{o\iota\iota}$, $\text{the}_{\iota(o\iota)}$ and individual constants, and with no free occurrences of variables other than individual variables. Given a set-theoretic abbreviation system Σ, a Σ-*sentence* (resp. a Σ-*term*) is a sentence (resp. a term) with no occurrences of constants other than constants defined by Σ, F.O. logical constants, $\text{in}_{o\iota\iota}$, $\text{the}_{\iota(o\iota)}$ and individual constants, and with no free occurrences of variables other than individual variables.

Lemma 2.6 *Given a set-theoretic abbreviation system Σ, for every Σ-sentence $\boldsymbol{P}$ there exists a μ-sentence $\boldsymbol{P}'$ such that*

$$\Sigma \vdash \boldsymbol{P} \equiv \boldsymbol{P}'$$

PROOF. The Σ-expansion of a formula does not change the variables which occur free in the formula, it removes any constants defined by Σ which occur in the formula, and it adds only constants which are part of the residual vocabulary of Σ. Therefore the Σ-expansion of a Σ-formula is a μ-formula. So for $\boldsymbol{P}'$ we can simply take the Σ-expansion of $\boldsymbol{P}$. □

Lemma 2.7 *For every μ-sentence $\boldsymbol{P}$ in $\beta\gamma$-nf there exists a $\mathcal{V}$-F.O. sentence $\boldsymbol{P}'$ such that*

$$\text{ZF} \vdash \boldsymbol{P} \equiv \boldsymbol{P}'$$

PROOF. Let $\boldsymbol{P}$ be a μ-sentence in $\beta\gamma$-nf having an occurrence of $\text{the}_{\iota(o\iota)}$. By lemma B.13, page 198, $\boldsymbol{P}$ has a subformula of the form "$\mu\boldsymbol{x}\boldsymbol{Q}$", where $\boldsymbol{x}$ is an individual variable and $\boldsymbol{Q}$ is a sentence. Furthermore, if we choose "$\mu\boldsymbol{x}\boldsymbol{Q}$" to be an innermost such subformula, then $\boldsymbol{Q}$ has no occurrences of $\text{the}_{\iota(o\iota)}$ (since $\boldsymbol{Q}$ itself is a μ-sentence in $\beta\dot{\gamma}$-nf).

Since "$\mu\boldsymbol{x}\boldsymbol{Q}$" has an atomic type (namely, ι), by lemma B.14, $\boldsymbol{P}$ has a subformula of the form "$\boldsymbol{s}\ \boldsymbol{A}_1\ \ldots\ \boldsymbol{A}_n$" where $\boldsymbol{s}$ is a symbol of n-ary type

$\delta\alpha_n \ldots \alpha_1$ and for some i, $1 \leq i \leq n$, $\boldsymbol{A}_i$ is of the form "$\lambda \boldsymbol{y}_1 \ldots \lambda \boldsymbol{y}_m \mu \boldsymbol{x} \boldsymbol{Q}$". But then α_i is of the form $\iota\beta_m \ldots \beta_1$. By lemma B.15 the bound variables of $\boldsymbol{P}$ are of type ι, so $\boldsymbol{s}$ must be a free symbol of $\boldsymbol{P}$; therefore $\boldsymbol{s}$ must be: a F.O. logical constant, the symbol $_{o\iota}$, the symbol $\mathrm{the}_{\iota(o\iota)}$, or an individual constant or variable. The only such symbols that have a type of the form

$$\delta \ldots (\iota \ldots) \ldots$$

are $\mathrm{equal}_{o\iota}$ and $\mathrm{in}_{o\iota}$. Therefore $\boldsymbol{P}$ must have a subformula $\boldsymbol{R}$ of one of the following forms ($\boldsymbol{A}$ being a term):

$$\begin{gathered} \boldsymbol{A} = \mu \boldsymbol{x} \boldsymbol{Q} \\ \mu \boldsymbol{x} \boldsymbol{Q} = \boldsymbol{A} \\ \boldsymbol{A} \in \mu \boldsymbol{x} \boldsymbol{Q} \\ \mu \boldsymbol{x} \boldsymbol{Q} \in \boldsymbol{A} \end{gathered}$$

In other words, $\boldsymbol{R}$ is a formula $\mathcal{C}[\mu \boldsymbol{x} \boldsymbol{Q}]$, where $\mathcal{C}$ is a context of one of the forms ("$\boldsymbol{A} = u$", u), ("$u = \boldsymbol{A}$", u), ("$\boldsymbol{A} \in u$", u), or ("$u \in \boldsymbol{A}$", u), where u is a variable which does not occur free in $\boldsymbol{A}$.

Let then $\boldsymbol{R}'$ be the formula:

$$\exists y((\exists! \boldsymbol{x} \boldsymbol{Q} \supset \boldsymbol{Q}^x_y) \wedge (\neg \exists! \boldsymbol{x} \boldsymbol{Q} \supset \forall z(\neg z \in y)) \wedge \mathcal{C}[y])$$

where y is an individual variable not free in $\boldsymbol{A}$ and adequate to $\boldsymbol{Q}^x$, and z is an individual variable distinct from y. Clearly, using rules 26 and 26a:

$$\mathrm{ZF} \vdash \boldsymbol{R} \supset \boldsymbol{R}'.$$

Using again rules 26 and 26a, together with the axiom of Extensionality of ZF:

$$\mathrm{ZF} \vdash \boldsymbol{R}' \supset \boldsymbol{R}.$$

Hence:

$$\mathrm{ZF} \vdash \boldsymbol{R} \equiv \boldsymbol{R}'.$$

And if $\boldsymbol{P}_1$ is obtained from $\boldsymbol{P}$ by substituting $\boldsymbol{R}'$ for $\boldsymbol{R}$, then:

$$\mathrm{ZF} \vdash \boldsymbol{P} \equiv \boldsymbol{P}_1 .$$

$\boldsymbol{P}_1$ is a μ-sentence in $\beta\gamma$-nf having one fewer occurrence of the$_{\iota(o\iota)}$ than $\boldsymbol{P}$. Therefore there exists a formula $\boldsymbol{P}'$ which has no occurrences of the$_{\iota(o\iota)}$, is a μ-sentence in $\beta\gamma$-nf, and is such that

$$\mathrm{ZF} \vdash \boldsymbol{P} \equiv \boldsymbol{P}'$$

But a μ-sentence in $\beta\gamma$-nf having no occurrences of the$_{\iota(o\iota)}$ is, by theorem 2.1, page 39, a $\mathcal{V}$-F.O. sentence. □

Theorem 2.8 (Eliminability of notations from formulas) *If Σ is a set-theoretic abbreviation system, for every Σ-sentence $\boldsymbol{P}$ there exists a $\mathcal{V}$-F.O. sentence $\boldsymbol{P}'$ such that*

$$\mathrm{ZF} \cup \Sigma \vdash \boldsymbol{P} \equiv \boldsymbol{P}'$$

PROOF. By lemma 2.6 there exists a μ-sentence $\boldsymbol{Q}$ such that $\vdash \boldsymbol{P} \equiv \boldsymbol{Q}$. $\boldsymbol{Q}$ has a $\beta\gamma$-nf $\boldsymbol{Q}'$ ($\alpha\beta\gamma$-CONVERSION, page 24). Every symbol which occurs free in $\boldsymbol{Q}'$ also occurs free in $\boldsymbol{Q}$; therefore $\boldsymbol{Q}'$ is also a μ-sentence. Then by lemma 2.7 there exists a $\mathcal{V}$-F.O. sentence $\boldsymbol{P}'$ such that $\vdash \boldsymbol{Q}' \equiv \boldsymbol{P}'$. By transitivity of equality (equivalence, in this case), $\Sigma \vdash \boldsymbol{P} \equiv \boldsymbol{P}'$. □

Now, given a set-theoretic abbreviation system Σ, if $\boldsymbol{R}$ is a formula which would be an instance of Replacement, except for the fact that the parameter $\boldsymbol{P}$ is a Σ-formula rather than a $\mathcal{V}$-F.O. formula, it follows from theorem 2.8 by substitutivity of equality (equivalence in this case) that:

$$\mathrm{ZF} \cup \Sigma \vdash \boldsymbol{R}.$$

So we can use the constants defined by Σ, and the notations that they represent, in the parameter of Replacement. And therefore we can also use them in the theorem schema of Separation (2.23), in the second special case of Replacement (2.27), and in the characterizations of the notations "$\{\boldsymbol{x} \in \boldsymbol{E} \mid \boldsymbol{P}\}$" (2.26) and "$\{\boldsymbol{A}\}_{\boldsymbol{x}\in\boldsymbol{E}}$" (2.30). More precisely, we have:

$$\mathrm{ZF} \cup \Sigma \vdash \forall \boldsymbol{z}_1 \ldots \forall \boldsymbol{z}_n \forall \boldsymbol{e} \exists \boldsymbol{s} \forall \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{s} \equiv \boldsymbol{x} \in \boldsymbol{e} \wedge \boldsymbol{P})$$

where the conditions are as in (2.23) except that $\boldsymbol{P}$ is now any Σ-sentence, Σ being a set-theoretic abbreviation system;

$$\mathrm{ZF} \cup \Sigma \vdash \forall \boldsymbol{z}_1 \ldots \forall \boldsymbol{z}_n \forall \boldsymbol{e} \exists \boldsymbol{s} \forall \boldsymbol{y} (\boldsymbol{y} \in \boldsymbol{s} \equiv \exists \boldsymbol{x} (\boldsymbol{x} \in \boldsymbol{e} \wedge \boldsymbol{y} = \boldsymbol{A}))$$

where the conditions are as in (2.27) except that $\boldsymbol{A}$ is now any Σ-term, Σ being a set-theoretic abbreviation system;

$$\mathrm{ZF} \cup \Sigma \vdash \forall \boldsymbol{x}(\boldsymbol{x} \in \{\boldsymbol{x} \in \boldsymbol{E} \mid \boldsymbol{P}\} \equiv \boldsymbol{x} \in \boldsymbol{E} \wedge \boldsymbol{P})$$

as in (2.26), except that $\boldsymbol{P}$ is now any Σ-sentence, Σ being a set-theoretic abbreviation system including equation (2.24), the definition of $\mathrm{subset}_{\iota(o\iota)\iota}$; and

$$\mathrm{ZF} \cup \Sigma \vdash \forall \boldsymbol{y}(\boldsymbol{y} \in \{\boldsymbol{A}\}_{x \in E} \equiv \exists \boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{E} \wedge \boldsymbol{y} = \boldsymbol{A}))$$

as in (2.30), except that $\boldsymbol{A}$ is now any Σ-term, Σ being a set-theoretic abbreviation system including equation (2.28), the definition of $\mathrm{range}_{\iota(\iota\iota)\iota}$.

2.5.7 Other axioms and notations of ZF

We now give the remaining axioms of ZF, and show how we formalize a few basic notations associated with them. In this section Σ is a set theoretic abbreviation system including all the definitions introduced in the section.

Pair-set axiom.

$$\forall x \forall y \exists s \forall z(z \in s \equiv z = x \vee z = y)$$

This axiom gives rise to the notation

"$\{\boldsymbol{A}, \boldsymbol{B}\}$" ($\boldsymbol{A}$,$\boldsymbol{B}$: terms) for "$\mathrm{enum}_{\iota\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$"

with the definition

$$\mathrm{enum}_{\iota\iota\iota} = \lambda x \lambda y \mu s \forall z(z \in s \equiv z = x \vee z = y)$$

The expansion of the notation gives:

$$\Sigma \vdash \{\boldsymbol{A}, \boldsymbol{B}\} = \mu \boldsymbol{s} \forall \boldsymbol{z}(\boldsymbol{z} \in \boldsymbol{s} \equiv \boldsymbol{z} = \boldsymbol{A} \vee \boldsymbol{z} = \boldsymbol{B})$$

where $\boldsymbol{s}$, $\boldsymbol{z}$ are distinct individual variables which do not occur free in $\boldsymbol{A}$ or $\boldsymbol{B}$. Then, by the Pair-set axiom and Extensionality:

$$\mathrm{ZF} \cup \Sigma \vdash \forall \boldsymbol{z}(\boldsymbol{z} \in \{\boldsymbol{A}, \boldsymbol{B}\} \equiv \boldsymbol{z} = \boldsymbol{A} \vee \boldsymbol{z} = \boldsymbol{B}).$$

There is also:

"$\{\boldsymbol{A}\}$" ($\boldsymbol{A}$: term) for "$\mathrm{enum}_{\iota\iota}\ \boldsymbol{A}$"

with the definition

$$\text{enum}_{\iota\iota} = \lambda x(\{x, x\})$$

from which:

$$\Sigma \vdash \{\boldsymbol{A}\} = \{\boldsymbol{A}, \boldsymbol{A}\}$$

Union axiom.

$$\forall x \exists s \forall z(z \in s \equiv \exists y(z \in y \wedge y \in x))$$

It gives rise to the notation:

"$\bigcup \boldsymbol{A}$" ($\boldsymbol{A}$: term) for "$\text{union}_{\iota\iota}\ \boldsymbol{A}$"

with the definition:

$$\text{union}_{\iota\iota} = \lambda x \mu s \forall z(z \in s \equiv \exists y(z \in y \wedge y \in x))$$

Expanding the notation we get:

$$\Sigma \vdash \bigcup \boldsymbol{A} = \mu s \forall z(z \in s \equiv \exists y(z \in y \wedge y \in \boldsymbol{A}))$$

where s, z and y are pairwise distinct individual variables which do not occur free in $\boldsymbol{A}$. By Extensionality and the Union axiom:

$$\text{ZF} \cup \Sigma \vdash \forall z(z \in \bigcup \boldsymbol{A} \equiv \exists y(z \in y \wedge y \in \boldsymbol{A}))$$

We have also two related notations:

"$\boldsymbol{A} \cup \boldsymbol{B}$" ($\boldsymbol{A}$,$\boldsymbol{B}$: terms) for "$\text{union}_{\iota\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$"

with the definition:

$$\text{union}_{\iota\iota\iota} = \lambda x \lambda y(\bigcup\{x, y\})$$

from which:

$$\Sigma \vdash \boldsymbol{A} \cup \boldsymbol{B} = \bigcup\{\boldsymbol{A}, \boldsymbol{B}\}$$

and:

"$\bigcup_{x \in E} \boldsymbol{A}$" ($x$: variable; $\boldsymbol{E}$,$\boldsymbol{A}$: terms) for "$\text{union}_{\iota(\iota\iota)\iota}\ \boldsymbol{E}\ \lambda x \boldsymbol{A}$"

with the definition:

$$\text{union}_{\iota(\iota\iota)\iota} = \lambda e \lambda a_{\iota\iota}(\bigcup\{a_{\iota\iota}\ x\}_{x \in e})$$

which can also be written, in fully formal notation:

$$\text{union}_{\iota(\iota\iota)\iota} = \lambda e \lambda a_{\iota\iota}(\text{union}_{\iota\iota}\ (\text{range}_{\iota(\iota\iota)\iota}\ e\ \lambda x(a_{\iota\iota}\ x)))$$

From this definition we get:

$$\Sigma \vdash \bigcup_{x \in E} \boldsymbol{A} = \bigcup(\{\boldsymbol{A}\}_{x \in E})$$

Power-set axiom.

$$\forall x \exists s \forall y (y \in s \equiv \forall z (z \in y \supset z \in x)) \tag{2.31}$$

The axiom becomes more readable if we introduce the $\subseteq$ notation, as explained in section 2.5.2:

$$\forall x \exists s \forall y (y \in s \equiv y \subseteq x) \tag{2.32}$$

Formula (2.32) is not a $\mathcal{V}$-F.O. sentence, so it cannot be an axiom of ZF; but it is a theorem of the theory ZF $\cup\, \Sigma$:

$$\text{ZF} \cup \Sigma \vdash \forall x \exists s \forall y (y \in s \equiv y \subseteq x).$$

The power-set axiom gives rise to the notation:

"$\mathcal{P}(\boldsymbol{A})$" ($\boldsymbol{A}$: term) for "$\text{powerset}_{\iota\iota}\ \boldsymbol{A}$"

with the definition:

$$\text{powerset}_{\iota\iota} = \lambda x \mu s \forall y (y \in s \equiv y \subseteq x).$$

Expanding the notation we get:

$$\Sigma \vdash \mathcal{P}(\boldsymbol{A}) = \mu s \forall y (y \in s \equiv y \subseteq \boldsymbol{A})$$

where s and y are distinct individual variables not free in $\boldsymbol{A}$. And by Extensionality and Power-set:

$$\text{ZF} \cup \Sigma \vdash \forall y (y \in \mathcal{P}(\boldsymbol{A}) \equiv y \subseteq \boldsymbol{A})$$

Axiom of Infinity. It is traditionally given as:

$$\exists s (\emptyset \in s \wedge \forall x (x \in s \supset x \cup \{x\} \in s)). \tag{2.33}$$

But of course this is not a $\mathcal{V}$-F.O. formula, so it cannot be an axiom of ZF. The real axiom is the formula obtained by

1. Computing the Σ-expansion of (2.33).
2. Converting to $\beta\gamma$-nf.
3. Eliminating the occurrences of $\mathrm{the}_{\iota(o\iota)}$ as explained in the proof of lemma 2.7.

(2.33) is then a theorem of $\mathbf{ZF} \cup \Sigma$:

$$\mathbf{ZF} \cup \Sigma \vdash \exists s(\emptyset \in s \wedge \forall x(x \in s \supset x \cup \{x\} \in s)).$$

Axiom of Foundation.

$$\forall x(\neg(x = \emptyset) \supset \exists y(y \in x \wedge y \cap x = \emptyset)) \tag{2.34}$$

Again, the real axiom is obtained from (2.34) by eliminating the abbreviations.

Chapter 3

A rewriting system for the translation of notations

3.1 A customizable surface language

The distinction between surface language and internal representation opens up the possibility of allowing the user to customize the surface language. This is important because mathematicians like to choose or invent their own notations, and it is desirable that a PDS allow them to do so. Even more importantly, what set of notations is "good" depends on the domain or even the problem at hand. The favorite mathematical notations, e.g. "$A + B$" or "$A \cdot B$" or "$A \rightarrow B$" are reused with different meanings in different contexts. Watson has been conceived as a *general purpose* PDS for mathematicians and engineers. Since it is not known what problems it will be applied to, it is not possible to design an optimal set of notations; so it is best to allow the user to specify his/her own notations.

As we shall see, Watson is very flexible in this regard. It is easy to switch from one set of notations to another. Definitions of theories, statements of theorems and lemmas, and proofs of results can be kept in a library in external format and internal format (or only in one of the formats, the other format being easily produced). Different surface languages can be used within the same library, while the common internal representation makes it possible to use results independently of the surface language in which they are stated.

But if the surface language is to be customizable, there must be an easy,

declarative way of specifying mathematical notations. And the front-end of the PDS cannot simply consist of an ad-hoc parser and pretty-printer; it must be able to deal with an entire class of surface languages. In this chapter we are going to present a simple syntactic theory of one-dimensional mathematical languages, within which notations can be specified as *rewrite rules* and translation can be accomplished by rewriting. (We say "one-dimensional" languages because we shall not handle two-dimensional aspects of notations, such as subscripts, superscripts, fraction bars, etc. An alternative linear syntax will have to be specified instead.)

3.2 Labeled expressions and patterns

The fact that expansion of shorthands is some form of rewriting is informally obvious. To translate "$\{x \mid \boldsymbol{P}\}$" into "$\mathrm{set}_{\iota(o\iota)}\,\lambda x \boldsymbol{P}$" one looks, within a given expression, for a subexpression of the form "$\{x \mid \boldsymbol{P}\}$"; one notes which variable plays the role of x, and which sentence plays the role of $\boldsymbol{P}$; then one constructs the expression "$\mathrm{set}_{\iota(o\iota)}\,\lambda x \boldsymbol{P}$"; finally one substitutes the constructed expression for the original subexpression.

The problem in making this more precise is that the theory of rewriting systems has been developed for *algebraic languages*, i.e. for languages whose expressions are *F.O. terms* (hence the phrase "term rewriting system"). We need a theory of rewriting for much richer languages. The language for internal representation is the language of H.O.L., of which F.O.L. is a "small" subset, and F.O. terms are only a "small" subset of F.O.L. The surface language is, in some sense, even richer, since it involves all kinds of variable binding constructs.

In extending the notion of rewriting to these richer languages, one has to be careful. For example, "$x \in y$" normally rewrites to "$\mathrm{in}_{o\iota\iota}\, x\, y$"; but it would be wrong to perform the rewrite within

$$\{x \in y \mid \boldsymbol{P}\} \tag{3.1}$$

This is because, in (3.1), "$x \in y$" is not a sentence, it is only part of a larger construct. This of course can only be seen after parsing the entire expression. It seems therefore that the tranlation process should consist of two steps: first, a syntactic analysis, then a series of rewrites.

Since there are two steps, there must be an intermediate form: the result of the analysis, to which the rewrites are applied. The result of the analysis must specify the boundaries of the subexpressions, and their syntactic categories. We are going to restrict our attention to the style of mathematical notation in which grouping is accomplished exclusively by parentheses. Then the result of the analysis can be shown by adding parentheses around every subexpression which does not have them yet, and annotating each pair of parentheses with a *phrase marker*, or *label*, indicating the syntactic category of the subexpression. Conversely, given such a *labeled expression*, the corresponding *unlabeled expressions* are obtained by erasing the markers, and then suppressing zero, some or all the pairs of parentheses.

We shall place each marker immediately to the right of the opening parenthesis enclosing the corresponding subexpression. The markers are additional symbols to be introduced besides the ordinary symbols of the language. We shall refer to them as *non-terminal symbols* and to the ordinary symbols of the language as *terminal symbols*. This is by an analogy with the theory of context-free languages and grammars which will be made precise in section 3.5. It should be noted that parentheses are neither terminal nor non-terminal symbols. They form a category of symbols by themselves, the *delimiters*.

In the typed λ-calculus, we distinguished in section 2.3.2 between the proper symbols (the constants and the variables) and the single improper symbol λ. They are all terminal symbols. As non-terminal symbols we shall use FML_α and VAR_α for every type α: FML_α will be used as a marker for the formulas of type α; VAR_α will be used for binding occurrences of variables, that is, in "$\lambda \boldsymbol{x} \boldsymbol{A}$", where $\boldsymbol{x}$ is a variable of type α, we shall consider "$\boldsymbol{x}$" as a subexpression of syntactic category "VAR_α". (The reason for the distinction between FML_α and VAR_α is that an occurrence of λ can be followed by a variable only, rather than by a subformula.) So, for example, the result of parsing the expression

$$\lambda x_\iota (p_{o\iota}\ x_\iota)$$

is:

$$(\text{FML}_{o\iota}\ \lambda\ (\text{VAR}_\iota\ x_\iota)\ (\text{FML}_o\ (\text{FML}_{o\iota}\ p_{o\iota})\ (\text{FML}_\iota\ x_\iota))).$$

For the surface language we shall use the same terminal and non-terminal symbols as for the typed λ-language, plus some additional ones. (It should be noted that the surface language consists of shorthands for the formal

language, but these shorthands are optional: the user can use the formal expressions instead. Also, there are expressions in the formal language which cannot be expressed by shorthands—more on this in section 3.4.4. Therefore the formal language must be considered part of the surface language.) As additional terminal symbols we shall use:

1. Miscellaneous mathematical symbols such as "$\forall$", "$\in$", "$\cup$" etc.

2. Unsubscripted roman identifiers used as keywords. For example, in the case study, section 4.3, we shall write "$\boldsymbol{V}$ low" (where $\boldsymbol{V}$ is a term denoting a voltage level) for "$\text{low}_{o\iota}\,\boldsymbol{V}$", and "$\text{nor}\,\boldsymbol{D}\,\boldsymbol{D}'\,\boldsymbol{S}\,\boldsymbol{S}'\,\boldsymbol{S}''$" (where $\boldsymbol{D}$ and $\boldsymbol{D}'$ are terms denoting delays, and $\boldsymbol{S}$, $\boldsymbol{S}'$ and $\boldsymbol{S}''$ are terms denoting signals) for "$\text{nor}_{o\iota\iota\iota\iota\iota}\,\boldsymbol{D}\,\boldsymbol{D}'\,\boldsymbol{S}\,\boldsymbol{S}'\,\boldsymbol{S}''$"; "low" and "nor" are keywords, while $\text{low}_{o\iota}$ and $\text{nor}_{o\iota\iota\iota\iota}$ are constants. Keywords will also be used instead of symbols which are not available (and cannot be imitated by a concatenation of other symbols) in the computing environment of Watson.

3. Unsubscripted italic identifiers used as surface versions of variables of type ι.

We shall use additional non-terminal symbols for the surface language only rarely. The non-terminal symbols of the typed λ-language are sufficient for most purposes; for example, the surface language expression

$$\{x_\iota \in y_\iota \mid z_\iota \in x_\iota\}$$

shall be analysed as:

$$(\text{FML}_\iota\ \{\ (\text{VAR}_\iota\ x) \in (\text{FML}_\iota\ y) \mid (\text{FML}_o\ (\text{FML}_\iota\ z) \in (\text{FML}_\iota\ x))\ \}). \qquad (3.2)$$

When we do use additional non-terminal symbols, they shall be identifiers in small-caps font, some of them subscripted by type expressions.

The reason why we write the markers immediately to the right of the opening parenthesis is that the resulting *labeled expressions* are strikingly similar to algebraic terms, the markers playing the role of algebraic function symbols, and the terminal symbols playing the role of algebraic constants. These algebraic terms are *many-sorted*, the sort of a term being simply the syntactic category of the expression, i.e. the marker following the opening

parenthesis, i.e. the top level algebraic function symbol. The analogy with sorted algebraic terms is not perfect, since function symbols and sorts are identified, but it is good enough to allow the theory of algebraic term rewriting to carry over.

In this analogy between labeled expressions and algebraic terms there is nothing yet corresponding to the algebraic variables, so labeled expressions are the equivalent of *ground terms*. Let us then introduce *pattern-matching variables*, materialized as identifiers in italics subscripted by non-terminal symbols, to play the role of algebraic variables. The subscript of a pattern-matching variable is its sort: if a pattern-matching variable is subscripted by a non-terminal $\boldsymbol{N}$, it can only "match" labeled expressions of sort $\boldsymbol{N}$, i.e. labeled expressions whose top-level label is $\boldsymbol{N}$. The pattern-matching variables should not be confused, of course, with the variables of the typed λ-language, which are terminal symbols. When there is no risk of confusion between the two kinds of variables, we shall refer to pattern-matching variables simply as variables.

We shall call *patterns* the generalization of labeled expressions obtained by allowing pattern-matching variables as "arguments" of the labels/funtion-symbols. For example,

$$(\text{FML}_\iota \{ x_{\text{VAR}_\iota} \in (\text{FML}_\iota\ y) \mid P_{\text{FML}_o} \}).$$

is a pattern which "matches" the labeled expression (3.2) displayed above, with x_{VAR_ι} matching

$$(\text{VAR}_\iota\ x)$$

and P_{FML_o} matching

$$(\text{FML}_o\ (\text{FML}_\iota\ z) \in (\text{FML}_\iota\ x)).$$

The labeled expressions are then the *ground patterns*, i.e. the patterns with no occurrences of pattern-matching variables.

Formally, a *pattern* is defined inductively as a string, or sequence, of symbols consisting of: (i) a single pattern-matching variable; or (ii) a left-parenthesis, a non-terminal symbol, an arbitrary number (zero or more) of terminal symbols or (sub)patterns, and a right-parenthesis. The sort of the pattern is: in case (i), the sort of the pattern-matching variable; and in case (ii), the non-terminal symbol which follows the opening left-parenthesis. A *proper subpattern* of a pattern $\boldsymbol{A}$ is a subpattern of $\boldsymbol{A}$ other than $\boldsymbol{A}$ itself.

We shall refer to a string consisting of a single pattern-matching variable as a *variable pattern*, and to any other pattern as a *non-variable pattern*.

Observe that a string consisting of a single terminal symbol is not a pattern, and a that a terminal symbol does not have a sort. This is another (also unessential) difference with an algebraic language, since non-terminal symbols play the role of algebraic constants in the analogy, but algebraic constants do have sorts in a many-sorted algebraic language.

Alternatively, algebraic constants could be considered to be algebraic function symbols of arity 0. They would then correspond, in the analogy, to non-terminal symbols followed by zero terminal symbols or patterns; i.e. they would correspond to patterns consisting of a left-parenthesis, followed by a non-terminal symbol, followed by a right-parenthesis. Then the terminal symbols would have no counterpart in the correspondance.

A *substitution (for pattern-matching variables)* is a function whose domain is a set of pattern-matching variables, such that the image of each variable (its *substitution value*) is a pattern of same sort as the variable. To *apply a substitution* θ to a pattern $\boldsymbol{P}$ is to replace the occurrences in $\boldsymbol{P}$ of variables in the domain of θ with the *substitution values* of the variables; the result, written $\boldsymbol{P}\theta$, is a *substitution instance* of the pattern $\boldsymbol{P}$. Note that given a pattern $\boldsymbol{P}$ and a substitution instance $\boldsymbol{A}$ of $\boldsymbol{P}$ there exists a unique substitution θ which yields $\boldsymbol{A}$ when applied to $\boldsymbol{P}$ and whose domain is the set of pattern-matching variables occurring in $\boldsymbol{P}$.

In the next section we shall use patterns to specify rewrite rules. But patterns can also be used to *define languages*. Indeed, recall that a labeled expression is nothing but a ground pattern. A set S of patterns is *stable by substitution* iff for every pattern $\boldsymbol{P}$ in S and every substitution θ which maps every pattern-matching variable in its domain to either a variable pattern or an element of S, the pattern $\boldsymbol{P}\theta$ is also in S. Let the *substitution closure* of a set of non-variable patterns Π be the smallest set which includes Π and is stable by substitution. Then the *language* $\mathcal{L}$ *generated by* Π is the set of labeled expressions (ground patterns) in the substitution closure of Π.

For example the typed λ-language can be redefined as the language generated by the following patterns, which we shall call *the basic patterns of the typed λ-language*:

1. For every pair of types α, β, the pattern "$(\mathrm{FML}_\alpha\ A_{\mathrm{FML}_{\alpha\beta}}\ B_{\mathrm{FML}_\beta})$".

2. For every pair of types α, β, the pattern "$(\text{FML}_{\alpha\beta}\ \lambda\ x_{\text{VAR}_\beta}\ A_{\text{FML}_\alpha})$".

3. For every type α and every proper symbol of type α of the typed λ-language, the pattern "$(\text{FML}_\alpha\ s)$". Recall that, in the typed λ-language, the proper symbols are the variables and the constants, all of which are terminal symbols in the extended framework.

4. For every type α and every variable $\boldsymbol{x}$ of type α, the pattern "$(\text{VAR}_\alpha\ \boldsymbol{x})$".

This new definition is equivalent to the one given in section 2.3.2 in the following sense: for every labeled expression in the language generated by the above patterns whose sort is of the form FML_α (this excludes expressions of sort VAR_α) the result of erasing the labels is a formula of the typed λ-language, of type α; conversely, for every formula $\boldsymbol{A}$ of type α of the typed λ-language, there exists exactly one labeled expression of sort FML_α in the language generated by the above set of patterns from which $\boldsymbol{A}$ can be derived by erasing the labels.

Consistently with our previous terminology, we shall refer to labeled expressions of sort FML_ι as *terms*, and to labeled expressions of sort FML_o as *sentences*.

3.3 Rewrite rules

A *rewrite rule* is a pair of patterns of same sort $(\boldsymbol{P}, \boldsymbol{P}')$ which satisfy the following conditions: (i) $\boldsymbol{P}$ is a non-variable pattern, and (ii) every pattern-matching variable which occurs in $\boldsymbol{P}'$ also occurs in $\boldsymbol{P}$. We shall write $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ for $(\boldsymbol{P}, \boldsymbol{P}')$; $\boldsymbol{P}$ is the *left-hand side* of the rule, and $\boldsymbol{P}'$ the *right-hand side*. For example,

$$(\text{FML}_o\ A_{\text{FML}_\iota} \in B_{\text{FML}_\iota}) \longrightarrow (\text{FML}_o\ (\text{FML}_{o\iota}\ (\text{FML}_{o\iota\iota}\ \text{in}_{o\iota\iota})\ A_{\text{FML}_\iota})\ B_{\text{FML}_\iota}) \quad (3.3)$$

is a rewrite rule.

A pattern $\boldsymbol{A}$ *rewrites* to a pattern $\boldsymbol{B}$ by an application of a rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ iff:

1. There exists a subpattern $\boldsymbol{A}'$ of $\boldsymbol{A}$ which is the result of applying a substitution θ to the pattern-matching variables of $\boldsymbol{P}$;

2. $\boldsymbol{B}'$ is the result of applying the same substitution θ to the right-hand side $\boldsymbol{P}'$; and

3. $\boldsymbol{B}$ is the result of replacing an occurrence of $\boldsymbol{A}'$ in $\boldsymbol{A}$ with $\boldsymbol{B}'$.

We then say that $\boldsymbol{B}$ is *a result of applying the rule* to $\boldsymbol{A}$ or, more precisely, to the occurrence of $\boldsymbol{A}'$ which is replaced.

In section 2.3.2 we defined a *context* of the typed λ-language as a pair $\mathcal{C} = (\boldsymbol{A}, x)$ and the notation $\mathcal{C}[\boldsymbol{B}]$ as $\boldsymbol{A}^x_{\boldsymbol{B}}$. Now we can define a more general notion of context by using a pattern-matching variable instead of a variable of the typed λ-language as place holder. We define a context again as a pair $\mathcal{C} = (\boldsymbol{A}, x)$, where now $\boldsymbol{A}$ is a pattern and x a pattern-matching variable. And we define the notation $\mathcal{C}[\boldsymbol{B}]$, where $\boldsymbol{B}$ is a pattern of same sort as x, as the result of substituting $\boldsymbol{B}$ for x in $\boldsymbol{A}$. (When we say that a pattern is of the form $\mathcal{C}[\boldsymbol{B}]$ we shall be implicitly asserting that $\boldsymbol{B}$ is a pattern of same sort as x.) Observe that if $\boldsymbol{A}$ has no occurrences of pattern-matching variables other than x, and $\boldsymbol{B}$ is a labeled expression (i.e. a ground pattern), then $\mathcal{C}[\boldsymbol{B}]$ is a also a labeled expression.[1]

A *simple context* is a context $(\boldsymbol{A}, x)$ such that x has exactly one occurrence in $\boldsymbol{A}$. A *trivial context* is a context $(\boldsymbol{A}, x)$ such that $\boldsymbol{A}$ is the variable pattern "x".

With these definitions we can more succinctly say that a pattern $\boldsymbol{A}$ rewrites to a pattern $\boldsymbol{A}'$ by an application of the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ iff $\boldsymbol{A}$ is of the form:

$$\mathcal{C}[\boldsymbol{P}\theta]$$

where $\mathcal{C}$ is a simple context and θ a substitution, and $\boldsymbol{A}'$ is the expression

$$\mathcal{C}[\boldsymbol{P}'\theta].$$

Since the rule introduces no variables (no variable occurs in $\boldsymbol{P}'$ without occurring in $\boldsymbol{P}$), if $\boldsymbol{A}$ is a labeled expression, then $\boldsymbol{A}'$ is also a lebeled expression.

[1]The new definition is indeed more general than the one of section 2.3.2: given an old-style context (A, x) where x is a variable of type α, a corresponding new-style context can be obtained by (i) labeling A with the appropriate non-terminal symbols to obtain a labeled expression B; (ii) choosing a pattern-matching variable v of sort FML_α, and (iii) replacing the occurrences of "$(FML_\alpha\ x)$" in B which have been derived from free occurrences of x in A with $\boldsymbol{v}$ to obtain a pattern P. The new-style context is (P,v).

The new definition can be used to provide a notion of context even for languages which do not have variables among their terminal symbols.

It is mostly the rewriting of labeled expressions which will be of interest to us, for the task of language translation.

As an example, let us try to apply rule (3.3) to the labeled expression (3.2), page 70. We observe that the subexpression

$$(\text{FML}_o\,(\text{FML}_\iota\,z) \in (\text{FML}_\iota\,x)) \tag{3.4}$$

is the result of applying the substitution

$$\begin{cases} A_{\text{FML}_\iota} \longmapsto \text{“}(\text{FML}_\iota\,z)\text{”} \\ B_{\text{FML}_\iota} \longmapsto \text{“}(\text{FML}_\iota\,x)\text{”} \end{cases}$$

to the left-hand side of the rule. Applying the same substitution to the right-hand side gives

$$(\text{FML}_o\,(\text{FML}_{o\iota}\,(\text{FML}_{o\iota\iota}\,\text{in}_{o\iota\iota})\,(\text{FML}_\iota\,z))\,(\text{FML}_\iota\,x)) \tag{3.5}$$

Then, replacing (3.4) with (3.5) in (3.2) we get:

$$(\text{FML}_\iota\,\{\,(\text{VAR}_\iota\,x) \in (\text{FML}_\iota\,y) \mid (\text{FML}_o\,(\text{FML}_{o\iota}\,(\text{FML}_{o\iota\iota}\,\text{in}_{o\iota\iota})\,(\text{FML}_\iota\,z))\,(\text{FML}_\iota\,x))\,\}).$$

Notice how the rule does not apply to the substring “$x \in y$” of “$\{x \in y \mid z \in x\}$”, for two reasons: “x” is not analysed as a term, and “$x \in y$” is not analysed as a sentence. Generally, the rule applies to “$\boldsymbol{A} \in \boldsymbol{B}$” when $\boldsymbol{A}$ and $\boldsymbol{B}$ are analysed as terms and “$\boldsymbol{A} \in \boldsymbol{B}$” is analyzed as a sentence; it then replaces (the labeled expression resulting from the analysis of) “$\boldsymbol{A} \in \boldsymbol{B}$” with (the labeled expression corresponding to) “$\text{in}_{o\iota\iota}\,\boldsymbol{A}\,\boldsymbol{B}$”. Thus applying the rewrite rule is expanding the shorthand “$\boldsymbol{A} \in \boldsymbol{B}$”.

All shorthands seen in chapter (2) can similarly be expressed as rewrite rules. To obtain a rewrite rule from a metalinguistic description of a shorthand is a matter of:

1. Labeling the schematic formula which gives the representation of the shorthand;

2. Labeling the schematic formula which gives the shortand itself; and

3. Replacing the metalinguistic variables by pattern-matching variables of appropriate sorts.

Thus rule (3.3) above can easily be derived from the shorthand description:

$$\text{“}\boldsymbol{A} \in \boldsymbol{B}\text{”}\ (\boldsymbol{A},\boldsymbol{B}\text{: terms}) \text{ for “in}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}\text{”}$$

by labeling “$\text{in}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$” as

$$(\text{FML}_o\ (\text{FML}_{o\iota}\ (\text{FML}_{o\iota\iota}\ \text{in}_{o\iota\iota})\ \boldsymbol{A})\ \boldsymbol{B}),$$

then labeling “$\boldsymbol{A} \in \boldsymbol{B}$” as

$$(\text{FML}_o\ \boldsymbol{A} \in \boldsymbol{B}),$$

and finally replacing “$\boldsymbol{A}$” with “A_{FML_ι}” and “$\boldsymbol{B}$” with “B_{FML_ι}”.

How to label the representation of the shorthand is determined by the definition given above of the typed λ-language as a language generated by a set of patterns. The labeling of the shorthand itself is not fully determined, since we have not defined the surface language yet as a set of labeled expressions. In fact, it is the labeling of the shorthands which will determine the surface language as the language generated by the following set of patterns:

1. The patterns which generate the typed λ-language, given above; and
2. The left-hand sides of all the shorthands in use.

The labeling of shorthands is however partly determined by the fact that both sides of a rule must be of the same sort. So, for example, the top-level label of “$\boldsymbol{A} \in \boldsymbol{B}$” must be the same as the top-level label of “$\text{in}_{o\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$”, viz. FML_o; simply stated, “$\boldsymbol{A} \in \boldsymbol{B}$” must be a sentence. Most often, shorthands can be analysed as not having any “internal structure”: then only the top-level label has to be added, and so the labeling of the shorthand is entirely determined; this is the case for “$\boldsymbol{A} \in \boldsymbol{B}$” which is labeled “$(\text{FML}_o\ \boldsymbol{A} \in \boldsymbol{B})$”.

Sometimes, though, shorthands have internal structure. This is the case, for example, of the notation “$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$”, which we saw in section 2.5.2:

$$\text{“}(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}\text{”}\ (\boldsymbol{x}\text{: variable; } \boldsymbol{E}\text{: term; } \boldsymbol{P}\text{: sentence}) \text{ for “forall}_{o(o\iota)\iota}\ \boldsymbol{E}\ \lambda\boldsymbol{x}\boldsymbol{P}\text{”}$$

Since we are not treating parentheses as terminal symbols, but rather as delimiters, “$(\forall \boldsymbol{x} \in \boldsymbol{E})$” must be a subexpression. We must introduce an auxiliary non-terminal symbol, say RQ (for “Restricted Quantifier”) to serve as its label. Since the representation of the shorthand is a formula of type o,

the top-level label of the shorthand is FML_o. So the labeling of the shorthand is:

$$(\text{FML}_o\,(\text{RQ}\,\forall\,\boldsymbol{x} \in \boldsymbol{E})\,\boldsymbol{P})$$

Using pattern-matching variables x_{VAR_ι}, E_{FML_ι} and P_{FML_o}, the left-hand side of the rule is

$$(\text{FML}_o\,(\text{RQ}\,\forall\,x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\,P_{\text{FML}_o}),$$

and the entire rule:

$$(\text{FML}_o\,(\text{RQ}\,\forall\,x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\,P_{\text{FML}_o}) \longrightarrow$$
$$(\text{FML}_o\,(\text{FML}_{o(o\iota)}\,(\text{FML}_{o(o\iota)\iota}\,\text{forall}_{o(o\iota)\iota})\,E_{\text{FML}_\iota})\,(\text{FML}_{o\iota}\,\lambda\,x_{\text{VAR}_\iota}\,P_{\text{FML}_o}))$$

Even when there are no explicit parentheses in the shorthand it may be reasonable to introduce internal structure; for example, we could emphasize that the notations:

$\{\boldsymbol{x} \in \boldsymbol{E} \mid \boldsymbol{P}\}$,
$(\forall \boldsymbol{x} \in \boldsymbol{E})\boldsymbol{P}$,
$\bigcup_{x \in E} \boldsymbol{A}$ (linearized as "$\bigcup\ \boldsymbol{x} \in \boldsymbol{E}\ ;\ \boldsymbol{A}$"),
etc.

have "$\boldsymbol{x} \in \boldsymbol{E}$" in common by making it a subexpression; such a subexpression would *not* be a sentence, so we would have to introduce again an auxiliary non-terminal symbol, say RANGE, to be used as its label. The left-hand sides of the rules would then be:

$(\text{FML}_\iota\,\{\,(\text{RANGE}\,x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota}) \mid P_{\text{FML}_o}\,\})$,
$(\text{FML}_o\,(\text{RQ}\,\forall\,(\text{RANGE}\,x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota}))\,P_{\text{FML}_o})$
$(\text{FML}_\iota\,\bigcup(\text{RANGE}\,x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\,;\,A_{\text{FML}_\iota})$
etc.

Certain shorthands are parameterized, and give rise to a family of rewrite rules. This is the case of "$\boldsymbol{A} = \boldsymbol{B}$", where $\boldsymbol{A}$ and $\boldsymbol{B}$ are formulas of arbitrary type α; the type α is a parameter of the notation. For every α we have a rewrite rule:

$$(\text{FML}_o\,A_{\text{FML}_\alpha} = B_{\text{FML}_\alpha}) \longrightarrow \text{equal}_{o\alpha\alpha}\,A_{\text{FML}_\alpha}\,B_{\text{FML}_\alpha}$$

(where the labeling of the right-hand side has been erased for readability, since it can easily be restored). The quantifiers also require one rewrite

rule for every type. For example, this is a rewrite rule schema for universal quantification:

$$(\text{FML}_o \, \forall \, x_{\text{VAR}_\alpha} \, P_{\text{FML}_o}) \longrightarrow \text{forall}_{o(o\iota)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$$

We could also introduce internal structure in this notation, which is in fact often written "$(\forall \boldsymbol{x})\boldsymbol{P}$". We would then introduce a family of auxiliary non-terminal symbols QUANT_α; the rule schema would become:

$$(\text{FML}_o \, (\text{QUANT}_\alpha \, \forall \, x_{\text{VAR}_\alpha}) \, P_{\text{FML}_o}) \longrightarrow \text{forall}_{o(o\iota)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$$

The auxiliary non-terminal symbols QUANT_α could be shared by the four quantifiers "for all", "there exists", "there exists at most one", and "there exists exactly one".

The translation between the surface and internal form of individual variables can also be accomplished by rewrite rules; the surface form of an individual variable is an italic identifier, while the internal form is the same identifier subscripted by the type ι. This time we have two rewrite rule schemas, both with the identifier $\boldsymbol{Id}$ as parameter:

$$(\text{FML}_\iota \, \boldsymbol{Id}) \longrightarrow (\text{FML}_\iota \, \boldsymbol{Id}_\iota)$$
$$(\text{VAR}_\iota \, \boldsymbol{Id}) \longrightarrow (\text{VAR}_\iota \, \boldsymbol{Id}_\iota)$$

Table 3.1 gives rewrite rules for all the notations discussed in chapter 2. The auxiliary non-terminal RQ is used, but not RANGE or QUANT_α. The labeling of the right-hand sides has been suppressed for readability. Two-dimensional notations have been linearized as follows:

$$\bigcup \, \boldsymbol{x} \in \boldsymbol{E} \, ; \, \boldsymbol{A} \quad \text{instead of} \quad \bigcup_{x \in E} \boldsymbol{A}$$
$$\{\boldsymbol{A} \, ; \, \boldsymbol{x} \in \boldsymbol{E}\} \quad \text{instead of} \quad \{\boldsymbol{A}\}_{x \in E}$$

3.4 Rewriting

3.4.1 Example

A set of rewrite rules constitutes a *rewriting system*. Given a rewriting system $\mathcal{R}$, we shall say that a pattern $\boldsymbol{A}$ *rewrites to* $\boldsymbol{A}'$ *in* $\mathcal{R}$ iff there exists a chain

1.	$(\text{FML}_o\ A_{\text{FML}_\alpha} = B_{\text{FML}_\alpha})$	$\longrightarrow$	$\text{equal}_{o\alpha\alpha}\ A_{\text{FML}_\alpha}\ B_{\text{FML}_\alpha}$
2.	$(\text{FML}_o\ P_{\text{FML}_o} \equiv Q_{\text{FML}_o})$	$\longrightarrow$	$\text{equal}_{ooo}\ P_{\text{FML}_o}\ Q_{\text{FML}_o}$
3.	$(\text{FML}_o\ \bot)$	$\longrightarrow$	false_o
4.	$(\text{FML}_o\ \top)$	$\longrightarrow$	true_o
5.	$(\text{FML}_o\ \neg\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{not}_{oo}\ P_{\text{FML}_o}$
6.	$(\text{FML}_o\ P_{\text{FML}_o} \wedge Q_{\text{FML}_o})$	$\longrightarrow$	$\text{and}_{ooo}\ \text{P}_{\text{FML}_o}\ \text{Q}_{\text{FML}_o}$
7.	$(\text{FML}_o\ P_{\text{FML}_o} \vee Q_{\text{FML}_o})$	$\longrightarrow$	$\text{or}_{ooo}\ P_{\text{FML}_o}\ Q_{\text{FML}_o}$
8.	$(\text{FML}_o\ P_{\text{FML}_o} \supset Q_{\text{FML}_o})$	$\longrightarrow$	$\text{implies}_{ooo}\ P_{\text{FML}_o}\ Q_{\text{FML}_o}$
9.	$(\text{FML}_o\ \forall\ x_{\text{VAR}_\alpha}\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{forall}_{o(o\alpha)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$
10.	$(\text{FML}_o\ \exists\ x_{\text{VAR}_\alpha}\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{exists}_{o(o\alpha)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$
11.	$(\text{FML}_o\ !\ x_{\text{VAR}_\alpha}\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{atmost}_{o(o\alpha)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$
12.	$(\text{FML}_o\ \exists!\ x_{\text{VAR}_\alpha}\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{unique}_{o(o\alpha)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$
13.	$(\text{FML}_\alpha\ \mu\ x_{\text{VAR}_\alpha}\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{the}_{\alpha(o\alpha)}(\lambda x_{\text{VAR}_\alpha} P_{\text{FML}_o})$
14.	$(\text{FML}_\iota\ \boldsymbol{Id})$	$\longrightarrow$	$(\text{FML}_\iota\ \boldsymbol{Id}_\iota)$
15.	$(\text{VAR}_\iota\ \boldsymbol{Id})$	$\longrightarrow$	$(\text{VAR}_\iota\ \boldsymbol{Id}_\iota)$
16.	$(\text{FML}_o\ A_{\text{FML}_\iota} \in B_{\text{FML}_\iota})$	$\longrightarrow$	$\text{in}_{o\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$
17.	$(\text{FML}_o\ (\text{RQ}\ \forall\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{forall}_{o(o\iota)\iota}\ E_{\text{FML}_\iota}\ (\lambda x_{\text{VAR}_\iota} P_{\text{FML}_o})$
18.	$(\text{FML}_o\ (\text{RQ}\ \exists\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\ P_{\text{FML}_o})$	$\longrightarrow$	$\text{exists}_{o(o\iota)\iota}\ E_{\text{FML}_\iota}\ (\lambda x_{\text{VAR}_\iota} P_{\text{FML}_o})$
19.	$(\text{FML}_o\ A_{\text{FML}_\iota} \subseteq B_{\text{FML}_\iota})$	$\longrightarrow$	$\text{subset}_{o\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$
20.	$(\text{FML}_\iota\ \{\ x_{\text{VAR}_\iota} \mid P_{\text{FML}_o}\ \})$	$\longrightarrow$	$\text{set}_{\iota(o\iota)}\ (\lambda x_{\text{VAR}_\iota} P_{\text{FML}_o})$
21.	$(\text{FML}_\iota\ \{\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota} \mid P_{\text{FML}_o}\ \})$	$\longrightarrow$	$\text{subset}_{\iota(o\iota)\iota}\ E_{\text{FML}_\iota}\ (\lambda x_{\text{VAR}_\iota} P_{\text{FML}_o})$
22.	$(\text{FML}_\iota\ \{\ A_{\text{FML}_\iota}\ ;\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota}\})$	$\longrightarrow$	$\text{range}_{\iota(\iota\iota)\iota}\ E_{\text{FML}_\iota}\ (\lambda x_{\text{VAR}_\iota} A_{\text{FML}_\iota})$
23.	$(\text{FML}_\iota\ \emptyset)$	$\longrightarrow$	emptyset_ι
24.	$(\text{FML}_\iota\ \{A_{\text{FML}_\iota}, B_{\text{FML}_\iota}\})$	$\longrightarrow$	$\text{enum}_{\iota\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$
25.	$(\text{FML}_\iota\ \{A_{\text{FML}_\iota}\})$	$\longrightarrow$	$\text{enum}_{\iota\iota}\ A_{\text{FML}_\iota}$
26.	$(\text{FML}_\iota\ \bigcup A_{\text{FML}_\iota})$	$\longrightarrow$	$\text{union}_{\iota\iota}\ A_{\text{FML}_\iota}$
27.	$(\text{FML}_\iota\ A_{\text{FML}_\iota} \cup B_{\text{FML}_\iota})$	$\longrightarrow$	$\text{union}_{\iota\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$
28.	$(\text{FML}_\iota\ \bigcup\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota}\ ;\ A_{\text{FML}_\iota})$	$\longrightarrow$	$\text{union}_{\iota(\iota\iota)\iota}\ E_{\text{FML}_\iota}\ (\lambda x_{\text{VAR}_\iota} A_{\text{FML}_\iota})$
29.	$(\text{FML}_\iota\ \mathcal{P}\ A_{\text{FML}_\iota})$	$\longrightarrow$	$\text{powerset}_{\iota\iota}\ A_{\text{FML}_\iota}$

Table 3.1: Rewrite rules for the translation of shorthands.

of one-step rewrites by rules of the system which takes $\boldsymbol{A}$ into $\boldsymbol{A}'$, i.e. iff there exists a sequence of patterns $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_n$, $n \geq 1$, such that $\boldsymbol{B}_1 = \boldsymbol{A}$, $\boldsymbol{B}_n = \boldsymbol{A}'$, and each $\boldsymbol{B}_i$ for $1 \leq i < n$ rewrites to $\boldsymbol{B}_{i+1}$ by an application of a rule of $\mathcal{R}$. A *normal form* for $\mathcal{R}$ is a pattern to which no rewrite rule of $\mathcal{R}$ is applicable. $\boldsymbol{A}'$ is a *normal form of* $\boldsymbol{A}$ iff it is a normal form for $\mathcal{R}$ to which $\boldsymbol{A}$ rewrites.

We shall consider a certain collection of shorthands defining a particular surface language, and we shall refer to the set of rules derived from those shorthands as the *rewriting system for shorthand elimintation*, $\mathcal{R}_{\text{ELIM}}$. The translation of a labeled expression from the surface language into the typed λ-language is accomplished by rewriting all the shorthands in the formula; more precisely, by successively applying rewrite rules until no more rules apply, i.e. by computing a normal form of the labeled expression for $\mathcal{R}_{\text{ELIM}}$. As an example, let us compute the representation of

$$\forall x \neg (x \in \emptyset). \tag{3.6}$$

It involves the following rewrite rules from table 3.1:

Rule 9	(Universal quantification)
Rule 5	(Negation)
Rule 16	(Set membership)
Rule 23	(Empty set)
Rule 14	(Variable conversion)
Rule 15	(Variable conversion)

The first step is to parse (3.6), i.e. to add all missing pairs of parentheses, and to add the phrase markers. Parsing is the subject of section 3.5, but it is obvious what the result should be:

$$(\text{FML}_o \, \forall \, (\text{VAR}_\iota \, x)(\text{FML}_o \, \neg \, (\text{FML}_o \, (\text{FML}_\iota \, x) \in (\text{FML}_\iota \, \emptyset)))) \tag{3.7}$$

Let us begin by translating the shorthand for universal quantification. Rule 9 matches the entire formula, with x_{VAR_ι} matching

$$(\text{VAR}_\iota \, x)$$

and P_{FML_o} matching

$$(\text{FML}_o \, \neg \, (\text{FML}_o \, (\text{FML}_\iota \, x) \, \in \, (\text{FML}_\iota \, \emptyset))).$$

We substitute these expressions for $x_{\mathbf{VAR}_\iota}$ and $P_{\mathbf{FML}_o}$ in the right-hand side of rule 9 (and we restore the labels of the right-hand side, which have been suppressed in table 3.1 for readability):

$$\begin{array}{l}(\mathrm{FML}_o \\ \quad (\mathrm{FML}_{o(o\iota)}\ \mathrm{forall}_{o(o\iota)}) \\ \quad (\mathrm{FML}_{o\iota}\ \lambda\ (\mathrm{VAR}_\iota\ x) \\ \qquad (\mathrm{FML}_o\ \neg\ (\mathrm{FML}_o\ (\mathrm{FML}_\iota\ x)\ \in\ (\mathrm{FML}_\iota\ \emptyset)))))\end{array} \tag{3.8}$$

Let us now translate the shorthand for negation. Rule 5 applies to the subformula

$$(\mathrm{FML}_o\ \neg\ (\mathrm{FML}_o\ (\mathrm{FML}_\iota\ x)\ \in\ (\mathrm{FML}_\iota\ \emptyset)))$$

which it transforms into

$$\begin{array}{l}(\mathrm{FML}_o \\ \quad (\mathrm{FML}_{oo}\ \mathrm{not}_{oo}) \\ \quad (\mathrm{FML}_o\ (\mathrm{FML}_\iota\ x)\ \in\ (\mathrm{FML}_\iota\ \emptyset)))\end{array}$$

By substitution in (3.8) we get:

$$\begin{array}{l}(\mathrm{FML}_o \\ \quad (\mathrm{FML}_{o(o\iota)}\ \mathrm{forall}_{o(o\iota)}) \\ \quad (\mathrm{FML}_{o\iota}\ \lambda\ (\mathrm{VAR}_\iota; x) \\ \qquad (\mathrm{FML}_o \\ \qquad\quad (\mathrm{FML}_{oo}\ \mathrm{not}_{oo}) \\ \qquad\quad (\mathrm{FML}_o\ (\mathrm{FML}_\iota\ x)\ \in\ (\mathrm{FML}_\iota\ \emptyset)))))\end{array}$$

Translation of the shorthand for set-membership (rule 16) gives:

$$\begin{array}{l}(\mathrm{FML}_o \\ \quad (\mathrm{FML}_{o(o\iota)}\ \mathrm{forall}_{o(o\iota)}) \\ \quad (\mathrm{FML}_{o\iota}\ \lambda\ (\mathrm{VAR}_\iota; x) \\ \qquad (\mathrm{FML}_o \\ \qquad\quad (\mathrm{FML}_{oo}\ \mathrm{not}_{oo}) \\ \qquad\quad (\mathrm{FML}_o\ (\mathrm{FML}_{o\iota}\ (\mathrm{FML}_{o\iota\iota}\ \mathrm{in}_{o\iota\iota})\ (\mathrm{FML}_\iota\ x))(\mathrm{FML}_\iota\ \emptyset))))).\end{array}$$

Finally we apply rules 23, 14 and 15, which rewrite the following subexpressions:

$$\begin{array}{lll}(\mathrm{FML}_\iota\ \emptyset) & \text{into} & (\mathrm{FML}_\iota\ \mathrm{emptyset}_\iota) \\ (\mathrm{FML}_\iota\ x) & \text{into} & (\mathrm{FML}_\iota\ x_\iota) \\ (\mathrm{VAR}_\iota\ x) & \text{into} & (\mathrm{VAR}_\iota\ x_\iota)\end{array}$$

The result is the representation of (3.6) in the formal system:

$$
\begin{array}{l}
(\mathrm{FML}_{o} \\
\quad (\mathrm{FML}_{o(o\iota)}\ \mathrm{forall}_{o(o\iota)}) \\
\quad (\mathrm{FML}_{o\iota}\ \lambda\ (\mathrm{VAR}_{\iota}\ x_{\iota}) \\
\qquad (\mathrm{FML}_{o} \\
\qquad\quad (\mathrm{FML}_{oo}\ \mathrm{not}_{oo}) \\
\qquad\quad (\mathrm{FML}_{o}\ (\mathrm{FML}_{o\iota}\ (\mathrm{FML}_{o\iota\iota}\ \mathrm{in}_{o\iota\iota})\ (\mathrm{FML}_{\iota}\ x_{\iota}))(\mathrm{FML}_{\iota}\ \mathrm{emptyset}_{\iota})))))
\end{array}
$$

which becomes more readable after erasing phrase markers and unnecessary parentheses:

$$\mathrm{forall}_{o(o\iota)}\ \lambda x_{\iota}(\mathrm{not}_{oo}\,(\mathrm{in}_{o\iota\iota}\ x_{\iota}\ \mathrm{emptyset}_{\iota})). \tag{3.9}$$

Conversely, given an expression of the typed λ-language such as (3.9), the corresponding surface language expression (3.6) is obtained by applying the same rewrite rules *in reverse*. Indeed, it is clear that if $(\boldsymbol{P}, \boldsymbol{P}')$ is a rewrite rule for shorthand translation, then the same pattern matching variables should appear in both patterns $\boldsymbol{P}$ and $\boldsymbol{P}'$, and $\boldsymbol{P}$ should be a non-variable pattern. Thus the opposite pair $(\boldsymbol{P}', \boldsymbol{P})$ should also be a rewrite rule; and if $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$ by $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, then $\boldsymbol{A}'$ rewrites to $\boldsymbol{A}$ by $\boldsymbol{P}' \rightarrow \boldsymbol{P}$. The opposites of the rules of $\mathcal{R}_{\mathrm{ELIM}}$ comprise what we shall call the *rewriting system for shorthand introduction*, $\mathcal{R}_{\mathrm{INTRO}}$. In general, we shall say that a rewriting system is *reversible* iff every rule contains the same pattern-matching variables in its right-hand side as in its left-hand side, and all patterns, left-hand sides as well as right-hand sides, are non-variable patterns. The opposites of the rewrite rules of a reversible system $\mathcal{R}$ form a rewriting system which we shall call the *reverse* of $\mathcal{R}$. Thus $\mathcal{R}_{\mathrm{ELIM}}$ and $\mathcal{R}_{\mathrm{INTRO}}$ are the reverse of each other.

This method of translation raises the following questions:

1. Is it certain that the chain of rewrites will terminate, both in the direct and in the reverse direction?

2. Is the resulting expression independent of the order in which the rewrites are applied, in both directions?

3. Does the direct rewriting process produce a formula of the typed λ-language?

4. Does the reverse rewriting process produce a labeled expression which is part of the surface language?

5. Is the effect of rewriting in one direction, and then the other, to produce the original expression?

6. Is the rewriting process fast enough for its intended use as part of the front-end of an interactive system?

We answer these questions in section 3.4.4 after establishing the results needed for that purpose in sections 3.4.2 and 3.4.3.

3.4.2 Some simple rewriting theory

To establish the needed results we need to develop a modest amount of rewriting theory.

Up to now we have stayed within a very concrete framework. We have introduced a vocabulary consisting of a variety of symbols: (i) The constants and variables of the typed λ-language, and the symbol "λ"; miscellaneous mathematical symbols, and unsubscripted roman and italic identifiers. We have referred to all these symbols as *terminal symbols.* (ii) Small-caps identifiers, subscripted by type expressions of the typed λ-language, or unsubscripted; we have referred to these as *non-terminal symbols*, or *sorts.* (iii) Italic identifiers subscripted by sorts, which we have called *pattern-matching variables.* (iv) The left and right parentheses, called the *delimiters.*

However, everything in the next two sections holds in a more abstract setting, consisting of: (i) An arbitrary set of *terminal symbols.* (ii) An arbitrary set of *non-terminal symbols*, also called *sorts*, disjoint from the set of terminal symbols. (iii) A set of *pattern-matching variables*, disjoint from the two previous sets of symbols, together with a function which assigns a sort to each pattern-matching variable. (iv) An ordered pair of *delimiters* distinct from the symbols in the three previous sets. The reader shall easily verify that all the definitions which have already been given in the concrete setting and are used in the next two sections can be transposed to the abstract setting.

As we have already pointed out, the framework in which we make use of rewriting is slightly different from the usual one, but the differences are unessential and the results of the theory of algebraic term rewriting carry

over to our framework. Rewriting theory is by now well developed, and most, if not all, of the results that we need follow from more general known results, as we shall point out in each case. For surveys of rewriting theory see [15, 16, 31, 35]. But the kind of rewriting system that we need for language translation is a very specific one, and it seems awkward to rely on overly general theorems which require elaborate proofs, when simple direct proofs can be given. So we shall give the simple proofs in addition to indicating the connection with the more general results.

Well-formed trees

We shall make use of *parse trees* of labeled expressions, and more generally of trees associated with patterns. We consider known the definition and basic facts and terminology related to *labeled trees*, when a labeled tree is regarded[2] as a triple (N, S, L), where N is the set of nodes, S is the function mapping each internal node to the ordered sequence of its successors,[3] and L is the function mapping each node to its label.[4] We shall say that a node n' is *above* a node n iff n' is on the path from the root to n, without being equal to n; *below* n iff n is on the path from the root to n', without being equal to n'; *beside* n iff n and n' are on divergent rooted paths.

A *well-formed tree* is a labeled tree whose whose internal nodes are labeled by non-terminal symbols, whose leaf nodes are labeled by terminal symbols or pattern matching variables, and whose root is an internal node or a leaf node labeled by a pattern-matching variable, but not a leaf node labeled by a terminal symbol. We shall refer to a leaf node labeled by a pattern-matching variable as a *variable node*, and to all other nodes as *non-variable* nodes. We shall refer to a node which is neither the root nor a variable node as an *inner*

[2] An alternative approach, where a tree is identified with the set of path coordinates (called *positions*) of its nodes, has been used in the context of term rewriting [49]. The approach which we are following is more convenient for our specific purposes as we shall show below.

[3] A node may be without successors in two different ways: an *internal* node n may have a sequence of successors $S(n)$ which is empty; a *leaf* node has no image by S.

[4] This is a different use of the word *label* from its use in the context of *labeled expressions*. We have been referring as a *label* or *marker* to the non-terminal following a left parenthesis. From now on we shall use only the word marker for this purpose, but we shall continue using the phrase *labeled expression*. Markers will be tree-labels of the parse trees of labeled expressions, but terminal symbols will be tree-labels of parse trees as well.

node; the inner nodes should not be confused with the internal nodes, the latter being simply the non-leaf nodes; the inner nodes are the internal nodes other than the root, if any, and the leaf nodes labeled by terminal symbols, if any.

With every well-formed tree we associate the string (sequence) of symbols defined by induction as follows: if the root of the tree is a leaf node, labeled by a pattern-matching variable v, then the associated string is the one-symbol sequence "v"; if the root is an internal node, then the associated string is the concatenation of: a left parenthesis; the non-terminal symbol which labels the root; the strings associated with the subtrees whose roots are the children of the root (if any); and a right parenthesis.

Clearly, the string associated with a well-formed tree is a pattern. Conversely, given a pattern $\boldsymbol{A}$, there exists a well-formed tree, determined up to isomorphism, whose associated string is $\boldsymbol{A}$. If $\boldsymbol{A}$ is a labeled expression, then a well-formed tree associated with $\boldsymbol{A}$ is nothing but a traditional parse tree of $\boldsymbol{A}$, determined up to isomorphism.

The *depth* of a tree is the number of nodes in the longest path from the root to a leaf node. The depth of a labeled expression or pattern $\boldsymbol{A}$ is the depth of any tree associated with $\boldsymbol{A}$.

Substitution by grafting

Let $\mathcal{T} = (N, S, L)$ and $\mathcal{T}' = (N', S', L')$ be two well-formed trees, let $n \in N$ be a node of $\mathcal{T}$, and let $\mathcal{T}'' = (N'', S'', L'')$ be the subtree of $\mathcal{T}$ rooted at n. Consider the following conditions: (i) the root of $\mathcal{T}'$ is n; (ii) the set of nodes N' of $\mathcal{T}'$ is disjoint from $N - N''$. If both conditions are met, then the triple:

$$\mathcal{T}''' = ((N - N'') \cup N', (S - S'') \cup S', (L - L'') \cup L')$$

is a well-formed tree. We shall refer to this construction as the *graft* of $\mathcal{T}'$ onto node n of $\mathcal{T}$. When $\mathcal{T}'$ meets the conditions we shall say that it is *adequate* for the graft.

Consider now a pattern $\boldsymbol{P}$ and a substitution θ. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{P}$, and let $n_1 \ldots n_k$ be the leaf nodes of $\mathcal{T}$ which are labeled by pattern matching variables in the domain of θ. For $1 \leq i \leq k$ let v_i be the pattern-matching variable which labels n_i, let $\boldsymbol{A}_i$ be the pattern $\theta(v_i)$, and let $\mathcal{T}'_i$ be a well-formed tree associated with $\boldsymbol{A}_i$. If the trees $\mathcal{T}'_i$ have pairwise disjoint sets of nodes, and each $\mathcal{T}'_i$ is adequate for grafting at node

n_i of $\mathcal{T}$, then the result $\mathcal{T}''$ of the k grafts is a well-formed tree associated with the pattern $\boldsymbol{P}\theta$. Observe that the result of the graft can be described as the componentwise union of the $\mathcal{T}'_i$ and the triple (N, S, L') where L' is the restriction of L to $N - \{n_1, \ldots, n_k\}$. We shall refer to a well-formed tree $\mathcal{T}''$ obtained as described from $\mathcal{T}$ and θ as being a *θ-graft* of $\mathcal{T}$.

Conversely, let $\boldsymbol{A}$ be a pattern of the form $\boldsymbol{P}\theta$, where $\boldsymbol{P}$ is a pattern and θ a substitution. If $\mathcal{T}$ is a well-formed tree associated with $\boldsymbol{A}$, then there exists a unique well-formed tree $\mathcal{T}'$ associated with $\boldsymbol{P}$ such that $\mathcal{T}$ is a θ-graft of $\mathcal{T}'$.

Occurrences and overlapping

Let $\mathcal{T}$ be a well-formed tree, let $\boldsymbol{A}$ be a pattern, and let $\mathcal{T}'$ be a well-formed tree associated $\boldsymbol{A}$. We shall say that $\mathcal{T}'$ is an *occurrence* of $\boldsymbol{A}$ in $\mathcal{T}$ iff there is a subtree $\mathcal{T}''$ of $\mathcal{T}$ which is a θ-graft of $\mathcal{T}'$ for some substitution θ. When this is the case every node n of $\mathcal{T}'$ is a node of $\mathcal{T}$, and if n is a non-variable node of $\mathcal{T}'$ then the labels of n in $\mathcal{T}$ and $\mathcal{T}'$ coincide, and the sequences of successors of n in $\mathcal{T}$ and $\mathcal{T}'$ coincide.

Transitivity. We shall often make implicit use of the following observation: if $\mathcal{T}_A$ is an occurrence of a pattern $\boldsymbol{A}$ in a well-formed tree $\mathcal{T}$, and $\mathcal{T}_B$ is an occurrence of a pattern $\boldsymbol{B}$ in $\mathcal{T}_A$, then $\mathcal{T}_B$ is also an occurrence of $\boldsymbol{B}$ in $\mathcal{T}$ itself.

Let $\boldsymbol{A}$ be a non-variable pattern, and $\mathcal{T}'$ an occurrence of $\boldsymbol{A}$ in a well-formed tree $\mathcal{T}$, with root r. Since $\boldsymbol{A}$ is a non-variable pattern the label of r in $\mathcal{T}'$ is a non-terminal symbol $\boldsymbol{f}$. Since the label of r if $\mathcal{T}'$ is not a pattern-matching variable, $\boldsymbol{f}$ is also the label of r in $\mathcal{T}$. Hence r cannot be a leaf node of $\mathcal{T}$. Thus the root of an occurrence of a non-variable pattern in a well-formed tree $\mathcal{T}$ is an internal node of $\mathcal{T}$.

We shall say that two patterns $\boldsymbol{A}$ and $\boldsymbol{B}$ *overlap* iff one of them, say $\boldsymbol{A}$, has a non-variable subpattern $\boldsymbol{A}'$ which has a common instance with the other pattern, $\boldsymbol{B}$; i.e. iff one of them, $\boldsymbol{A}$, is of the form $\mathcal{C}[\boldsymbol{A}']$, where $\mathcal{C}$ is a simple context and $\boldsymbol{A}'$ is a non-variable pattern, and there exist substitutions θ and θ' such that $\boldsymbol{A}'\theta = \boldsymbol{B}\theta'$.

Any non-variable pattern trivially overlaps itself in this sense. However, we shall say that a pattern $\boldsymbol{A}$ *self-overlaps* iff it has a common instance with a *proper* subpattern $\boldsymbol{A}'$ of itself; i.e. iff $\boldsymbol{A}$ is of the form $\mathcal{C}[\boldsymbol{A}']$ where $\mathcal{C}$ is a simple but non-trivial context and $\boldsymbol{A}'$ is a non-variable pattern, and there

exist substitutions θ and θ' such that $\boldsymbol{A}'\theta = \boldsymbol{A}\theta'$.

Let now $\mathcal{T}$ be a well-formed tree, and let $\mathcal{T}'$ and $\mathcal{T}''$ be occurrences in $\mathcal{T}$ of two patterns $\boldsymbol{A}$ and $\boldsymbol{B}$ respectively. We shall say that $\mathcal{T}'$ and $\mathcal{T}''$ *overlap* iff the root of one of them is an internal node of the other. We shall say that they overlap *at the root* iff they overlap and their roots coincide.[5] If $\mathcal{T}'$ and $\mathcal{T}''$ overlap, then so do the patterns $\boldsymbol{A}$ and $\boldsymbol{B}$. Indeed, assume that the root node n of $\mathcal{T}''$ is an internal node of $\mathcal{T}'$. Let $\boldsymbol{A}'$ be the pattern associated with the subtree of $\mathcal{T}'$ rooted at n. Clearly, $\boldsymbol{A}'$ is a non-variable subpattern of $\boldsymbol{A}$, and the pattern associated with the subtree of $\mathcal{T}$ rooted at n is a common instance of $\boldsymbol{A}'$ and $\boldsymbol{B}$.

Let $\mathcal{T}$ be again a well-formed tree, and let now $\mathcal{T}'$ and $\mathcal{T}''$ be occurrences of *the same* pattern $\boldsymbol{A}$. We shall say that $\mathcal{T}'$ and $\mathcal{T}''$ have a *proper overlap* iff they overlap without overlapping at the root, i.e. iff the root of one of them is an internal node of the other *other than the root.* Clearly, if $\mathcal{T}'$ and $\mathcal{T}''$ have a proper overlap then $\boldsymbol{A}$ self-overlaps. If, on the other hand, $\mathcal{T}'$ and $\mathcal{T}''$ overlap at the root, then they coincide.

Lemma 3.1 *Let $\mathcal{T}$ be a well-formed tree and let $\mathcal{T}'$ and $\mathcal{T}''$ be occurrences in $\mathcal{T}$ of two* non-variable *patterns. Then $\mathcal{T}'$ and $\mathcal{T}''$ overlap iff they have a non-variable node in common.*

PROOF. If $\mathcal{T}'$ and $\mathcal{T}''$ overlap, then the root of one of them, say $\mathcal{T}'$, is an internal node of the other. But since $\mathcal{T}'$ is associated with a non-variable pattern, its root is an internal node of itself. Hence $\mathcal{T}'$ and $\mathcal{T}''$ have an internal node (and a fortiori a non-variable node) in common.

Conversely, assume that $\mathcal{T}'$ and $\mathcal{T}''$ have a non-variable node n in common. By going up both trees, it is clear that the root r of one of them, say $\mathcal{T}'$, is a non-variable node of the other. Then r has the same label in $\mathcal{T}$, $\mathcal{T}'$ and $\mathcal{T}''$, and that label is a non-terminal symbol. Hence the root r of $\mathcal{T}'$ is an internal node of $\mathcal{T}''$, and thus $\mathcal{T}'$ and $\mathcal{T}''$ coincide. □

Reduction of a redex

Let $\mathcal{T}$ be a well-formed tree, $\boldsymbol{P} \to \boldsymbol{P}'$ a rewrite rule, and $\mathcal{T}_P$ an occurrence of the left-hand side $\boldsymbol{P}$ in $\mathcal{T}$. Let n be the root of $\mathcal{T}_P$, let $\boldsymbol{A}$ be the pattern

[5] It is possible for the roots to coincide without there being an overlap: such is the case when and only when $\boldsymbol{A}$ and $\boldsymbol{B}$ are variable patterns.

associated with the subtree of $\mathcal{T}$ rooted at n, and let θ be the substitution which yields $\boldsymbol{A}$ when applied to $\boldsymbol{P}$ and whose domain is the set of pattern-matching variables which occur in $\boldsymbol{P}$. We shall say that a tree $\mathcal{T}'$ results from $\mathcal{T}$ by *rewriting* $\mathcal{T}_P$ *with the rule* $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ iff $\mathcal{T}'$ is the result of grafting an adequate tree associated with $\boldsymbol{P}'\theta$ at node n of $\mathcal{T}$. If this is the case, $\mathcal{T}'$ is a well-formed tree, and it has an occurrence $\mathcal{T}_{P'}$ of $\boldsymbol{P}'$ rooted at n. We shall refer to $\mathcal{T}_{P'}$ as the occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$ resulting from the rewrite. All trees resulting from rewrites of a given occurrence of a pattern in a given well-formed tree are isomorphic.

Clearly, if $\mathcal{T}'$ results from $\mathcal{T}$ by rewriting an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$ in $\mathcal{T}$ with the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, then the pattern associated with $\mathcal{T}$ rewrites to the pattern associated with $\mathcal{T}'$ by an application of the rule. Conversely, if a pattern $\boldsymbol{Q}$ rewrites to $\boldsymbol{Q}'$ by an application of the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, and $\mathcal{T}$ is a well-formed tree associated with $\boldsymbol{Q}$, then there exists an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$ in $\mathcal{T}$ such that, if $\mathcal{T}'$ results from $\mathcal{T}$ by rewriting $\mathcal{T}_P$ with the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, then $\mathcal{T}'$ is a well-formed tree associated with the pattern $\boldsymbol{Q}'$.

When a rewriting system $\mathcal{R}$ is given, we shall refer to a pair $(\mathcal{T}_P, \boldsymbol{P} \rightarrow \boldsymbol{P}')$ where $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ is a rewrite rule of $\mathcal{R}$ and $\mathcal{T}_P$ is an occurrence of the left-hand side $\boldsymbol{P}$ in a well-formed tree $\mathcal{T}$, as a *redex* of $\mathcal{R}$ in $\mathcal{T}$. And we shall refer to a rewrite of $\mathcal{T}_P$ by the rule as a *reduction* of the redex. We shall sometimes refer to $\mathcal{T}_P$ itself as the redex, if it is clear from the context which rewrite rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ we have in mind, which is the case in particular when distinct rules have distinct left-hand sides.

A pattern is said to be *linear* iff no pattern-matching variable occurs more than once in it. A rewriting system is *left-linear* iff the left-hand sides of the rules are linear, *right-linear* iff the right-hand sides of the rules are linear.

If both sides of a rewrite rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ are linear and, in addition, $\boldsymbol{P}$ and $\boldsymbol{P}'$ contain exactly the same pattern-matching variables, then every pattern-matching variable used in the rule occurs exactly once in each side. We shall refer to such a rule as a *permuting* rule. In a rewriting system which is left-linear, right-linear and reversible, every rule is a permuting rule.

It turns out that, when $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ is a permuting rule, there is a particularly convenient way in which an occurrence of $\boldsymbol{P}$ in a well-formed tree can be rewritten. Let V be the set of pattern-matching variables which occur in the rule. As before, let $\mathcal{T}_P$ be an occurrence of $\boldsymbol{P}$ in a well-formed tree $\mathcal{T}$, let n be the root of $\mathcal{T}_P$, let $\boldsymbol{A}$ be the pattern associated with the subtree of $\mathcal{T}$ rooted at n, and let θ be the substitution with domain V which yields $\boldsymbol{A}$

when applied to $\boldsymbol{P}$. For each $v \in V$ let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted at the unique leaf note of $\mathcal{T}_P$ labeled by the pattern-matching variable v. There clearly exists a well-formed tree $\mathcal{T}_{P'}$ which is associated with the pattern $\boldsymbol{P}'$ and which satisfies the following conditions: (i) the root is n; (ii) for every $v \in V$, the unique leaf node l_v labeled by v coincides with the root of $\mathcal{T}_v$; and (iii) the inner nodes are not nodes of $\mathcal{T}$. The well-formed trees $\mathcal{T}_v$, $v \in V$ have disjoint sets of nodes and are adequate for grafting at the leaf nodes l_v, $v \in S$ of $\mathcal{T}_{P'}$. The result of the graft is a well-formed tree $\mathcal{T}_{A'}$ whose associated pattern is $\boldsymbol{A}' = \boldsymbol{P}'\theta$. And $\mathcal{T}_{A'}$ is adequate for grafting at node n of $\mathcal{T}$; the result of the graft is a well-formed tree $\mathcal{T}'$. $\mathcal{T}'$ is one of the trees which result from $\mathcal{T}$ by rewriting $\mathcal{T}_P$ with the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$. We shall say that $\mathcal{T}'$ is *the tree which results from $\mathcal{T}$ by rewriting $\mathcal{T}_P$ to $\mathcal{T}_{P'}$ in situ.*

The convenience of this particular way of rewriting an occurrence of a pattern comes from the fact that the original well-formed tree $\mathcal{T}$ is minimally disturbed. In particular, the resulting tree $\mathcal{T}'$ shares with $\mathcal{T}$ the subtrees $\mathcal{T}_v$, $v \in S$, in addition to sharing the root n of $\mathcal{T}_P$ and $\mathcal{T}_{P'}$ and all the nodes above and beside n. We shall refer to this kind of rewriting as *rewriting in situ.*

Symmetry of in-situ rewriting.[6] If $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ is a permuting rule and if $\boldsymbol{P}'$ is a non-variable pattern (so that $\boldsymbol{P}' \rightarrow \boldsymbol{P}$ qualifies as a rewrite rule) then $\boldsymbol{P}' \rightarrow \boldsymbol{P}$ is also a permuting rule. In that case, if $\mathcal{T}'$ results from $\mathcal{T}$ by rewriting in situ an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$ in $\mathcal{T}$ with $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, and if $\mathcal{T}_{P'}$ is the resulting occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$, then $\mathcal{T}$ results from $\mathcal{T}'$ by rewriting in situ the occurrence $\mathcal{T}_{P'}$ of $\boldsymbol{P}'$ in $\mathcal{T}'$ with $\boldsymbol{P}' \rightarrow \boldsymbol{P}$, and $\mathcal{T}_P$ is the resulting occurrence of $\boldsymbol{P}$ in $\mathcal{T}$.

The following lemma exploits the concept of in-situ rewriting.[7]

Lemma 3.2 *Let $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ be a permuting rewrite rule, let $\mathcal{T}$ be a well-formed tree having an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$, let $\mathcal{T}'$ be a well-formed tree obtained from*

[6] A non-symmetric version of in-situ rewriting could also be defining. It could be use for a rule $P \rightarrow P'$ with the only requirement that P' be right-linear. Theorem 3.3 below could then be stated without the conditions of left-linearity and reversibility, while keeping the same proof.

[7] In-situ rewriting and the lemma that follows are not easily available when trees are defined as sets of path coordinates ("positions") as in [49], since coordinates below $\mathcal{T}_P$ in $\mathcal{T}$ depend on the shape of $\mathcal{T}_P$, and coordinates below $\mathcal{T}_{P'}$ in $\mathcal{T}'$ depend on the shape of $\mathcal{T}_{P'}$. This is one reason why we have defined a tree as a triple (N, S, L) instead of following Rosen's approach.

$\mathcal{T}$ by rewriting $\mathcal{T}_P$ by the rule in situ, and let $\mathcal{T}_{P'}$ be the occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$ resulting from the rewrite. Let $\boldsymbol{Q}$ be a linear pattern. Then: (i) any occurrence of $\boldsymbol{Q}$ in $\mathcal{T}$ which does not overlap $\mathcal{T}_P$ is also an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$; and (ii) any occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$ which does not overlap $\mathcal{T}_{P'}$ is also an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$.

PROOF. By the symmetry of in-situ rewriting, (ii) follows from (i). We prove (i). Let $\mathcal{T}_Q$ be an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}$ which does not overlap $\mathcal{T}_P$. Let n be the root of $\mathcal{T}_P$ and m the root of $\mathcal{T}_Q$. In $\mathcal{T}$, the node m can be above, below, or beside n If m is beside n, then the subtree of $\mathcal{T}$ rooted at m is clearly also a subtree of $\mathcal{T}'$, and hence $\mathcal{T}_Q$ is also an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$. If m is below n then, since it cannot be an internal node of $\mathcal{T}_P$, it must belong to a subtree $\mathcal{T}_v$ of $\mathcal{T}$ rooted at a leaf node p of $\mathcal{T}_P$ labeled, in $\mathcal{T}_P$, by a pattern-matching variable v. But $\mathcal{T}_v$ is among the trees which, in the process of rewriting $\mathcal{T}_p$ in situ, are grafted onto $\mathcal{T}_{P'}$, the result of the graft being then grafted onto $\mathcal{T}$ at n. Therefore $\mathcal{T}_v$ is also a subtree of $\mathcal{T}'$, and hence $\mathcal{T}_Q$ is also an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$. If m is above n then, since n cannot be an internal node of $\mathcal{T}_Q$, it must belong to a subtree $\mathcal{T}_w$ of $\mathcal{T}$ rooted at a leaf node q of $\mathcal{T}_Q$ labeled, in $\mathcal{T}_Q$, by a pattern-matching variable w. The subtree of $\mathcal{T}$ rooted at m is modified in the process of rewriting $\mathcal{T}_P$ in situ to $\mathcal{T}_{P'}$. However, the modification takes place only in the subtree rooted at q. Let $\boldsymbol{A}$ be the subtree of $\mathcal{T}$ rooted at m. Let θ be the substitution defined on the pattern-matching variables of $\boldsymbol{Q}$ such that $\boldsymbol{A} = \boldsymbol{Q}\theta$. Since $\boldsymbol{Q}$ is a linear pattern, it has only one occurrence of w. Let then θ' be the subtitution which coincides with θ except that it maps w to the pattern associated with the subtree of $\mathcal{T}'$ rooted at q. Since the subtree of $\mathcal{T}$ rooted at m is a θ-graft of $\mathcal{T}_Q$, the subtree of $\mathcal{T}'$ rooted at m is a θ'-graft of $\mathcal{T}_Q$, and hence $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$. □

Termination

A rewriting system is *terminating* iff there are no infinite chains of rewrites, i.e. iff there is no infinite sequence of patterns $(\boldsymbol{A}_i)_{i\in\omega}$ where each $\boldsymbol{A}_i$ rewrites to $\boldsymbol{A}_{i+1}$ by an application of one of the rules of the system. To answer question 1 of section 3.4.1 we shall establish in section 3.4.4 that $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are both terminating. For that purpose we shall make use of the following definition and theorem.

Definition 3.1 *A rewriting system is* left-to-right non-overlapping *iff no left-hand side overlaps any right-hand side.*

(In connection with the preceding definition, recall that the notion of pattern overlap is *symmetric.* Some people refer to this same notion as *symmetric left-to-right non-overlap.*)

Theorem 3.3 *If a rewriting system is left-linear, right-linear, reversible and left-to-right non-overlapping, then it is terminating.*[8]

This theorem follows from a result of Dershowitz [15, Theorem 33, page 109] which asserts that a right-linear rewrite system is terminating iff it has no infinite "forward closures". Indeed a left-to-right non-overlapping system has only trivial forward closures, the rules themselves. However there is a direct proof based on lemma 3.2 which does not require the use of the of the forward closure construction. Here is the direct proof.

PROOF. Let $\mathcal{R}$ be a rewriting system satisfying the conditions of the theorem. Every rule of $\mathcal{R}$ is then a permuting rule.

Let $\boldsymbol{A}$ be a pattern which rewrites to a pattern $\boldsymbol{A}'$ by an application of a rule $\boldsymbol{P} \to \boldsymbol{P}'$ of $\mathcal{R}$. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{A}$. There exists an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$ in $\mathcal{T}$ such that any well-formed tree obtained by rewriting $\mathcal{T}_P$ with the rule $\boldsymbol{P} \to \boldsymbol{P}'$ is associated with the pattern $\boldsymbol{A}'$. Since $\boldsymbol{P} \to \boldsymbol{P}'$ is a permuting rule, $\mathcal{T}_P$ can be rewritten in situ; let $\mathcal{T}'$ be the resulting well-formed tree associated with $\boldsymbol{A}'$, and $\mathcal{T}_{P'}$ the resulting occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$.

Let now $(\mathcal{T}_Q, \boldsymbol{Q} \to \boldsymbol{Q}')$ be a redex of $\mathcal{R}$ in $\mathcal{T}'$. Since $\mathcal{R}$ is left-to-right non-overlapping, the patterns $\boldsymbol{P}'$ and $\boldsymbol{Q}$ do not overlap. As a consequence, their occurrences $\mathcal{T}_{P'}$ and $\mathcal{T}_Q$ do not overlap. Let n be the common root of $\mathcal{T}_P$ and $\mathcal{T}_{P'}$, and m the root of $\mathcal{T}_Q$. Since $\boldsymbol{Q}$ is a non-variable pattern, m is an internal node of $\mathcal{T}_Q$, and therefore $n \neq m$.

Since $\mathcal{T}_Q$ does not overlap $\mathcal{T}_{P'}$, by lemma 3.2(ii), $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}$ also, and hence $(\mathcal{T}_Q, \boldsymbol{Q} \to \boldsymbol{Q}')$ is a redex of $\mathcal{R}$ in $\mathcal{T}$; and since $n \neq m$, it does not coincide with the redex $(\mathcal{T}_P, \boldsymbol{P} \to \boldsymbol{P}')$.

We have shown that if $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$ in one step by application of a rule of $\mathcal{R}$, then every redex of $\mathcal{R}$ in $\boldsymbol{A}'$ is a redex in $\boldsymbol{A}$ other than the one being reduced. Hence there are strictly fewer redexes in $\boldsymbol{A}'$ than in $\boldsymbol{A}$. Therefore there can be no infinite chain of rewrites by rules of $\mathcal{R}$. □

[8] See footnote 6 page 89.

Confluence

A rewriting system is *confluent* iff whenever two chains of rewrites lead from a pattern $\boldsymbol{A}$ to two patterns $\boldsymbol{B}$ and $\boldsymbol{C}$, then two chains of rewrites lead from $\boldsymbol{B}$ and $\boldsymbol{C}$ to the same pattern $\boldsymbol{D}$. A rewriting system is *convergent* (or *complete*) iff it is terminating and confluent. In a convergent system, every pattern has a unique normal form.

To answer question 2 of section 3.4.1 we shall establish in section 3.4.4 that $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are both convergent. For that purpose we shall make use of the following definition and theorem.

Definition 3.2 *A rewriting system is* non-overlapping *iff left-hand sides of distinct rules do not overlap, and no left-hand side is self-overlapping.*

Theorem 3.4 *If a rewriting system is left-linear, right-linear, reversible, non-overlapping and left-to-right non-overlapping, then it is convergent.*

A rewriting system which is left-linear and non-overlapping is said to be *orthogonal* [35]. Theorem 3.4 is a special case of a theorem which asserts that every orthogonal rewriting system is confluent. A proof of this more general theorem can be found in [49]. But again a simpler proof based on lemma 3.2 can be given.

PROOF. Let $\mathcal{R}$ be a rewriting system which satisfies the conditions of the theorem. By theorem 3.3 we already know that $\mathcal{R}$ is terminating. To show that it is confluent it suffices to show that whenever a pattern $\boldsymbol{A}$ rewrites to distinct patterns $\boldsymbol{B}$ and $\boldsymbol{C}$ *in one step*, then $\boldsymbol{B}$ and $\boldsymbol{C}$ both rewrite to the same pattern $\boldsymbol{D}$ *in one step*.[9]

Assume then that $\boldsymbol{A}$ rewrites to $\boldsymbol{B}$ by one application of a rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, and to a pattern $\boldsymbol{C}$ distinct from $\boldsymbol{B}$ by one application of a rule $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{A}$. There is an occurrence $\mathcal{T}_P$ of $\boldsymbol{P}$ in $\mathcal{T}$ such that any tree resulting from $\mathcal{T}$ by rewriting $\mathcal{T}_P$ with the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ is a well-formed tree associated with the pattern $\boldsymbol{B}$. And there is an occurrence $\mathcal{T}_Q$ of $\boldsymbol{Q}$ in $\mathcal{T}$ such that any tree resulting from $\mathcal{T}$ by rewriting

[9] We are *not* using Newman's theorem here (theorem B.1, page 186). If we knew only that B and C rewrite to D in some number of steps (local confluence), then, having termination, we would use Newman's theorem to establish confluence. But if B and C both rewrite to D in a single step, confluence is obvious, independently of termination: fill-in the lattice.

$\mathcal{T}_Q$ with the rule $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ is a well-formed tree associated with the pattern $\boldsymbol{C}$.

Because $\mathcal{R}$ is non-overlapping, $\mathcal{T}_P$ and $\mathcal{T}_Q$ do not overlap. For if $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ and $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ are distinct rules of $\mathcal{R}$, then the patterns $\boldsymbol{P}$ and $\boldsymbol{Q}$ do not overlap. And if the rules coincide, then $\mathcal{T}_P$ and $\mathcal{T}_Q$ are occurrences in $\mathcal{T}$ of the same non-self-overlapping pattern; this means that they cannot have a proper overlap; and they cannot overlap at the root, because then they would be identical, and $\boldsymbol{B}$ and $\boldsymbol{C}$ would be the same pattern, contrary to the hypothesis.

Since $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ and $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ are permuting rules, $\mathcal{T}_P$ and $\mathcal{T}_Q$ can be rewritten in situ. Let $\mathcal{T}_{P'}$ be a well-formed tree associated with $\boldsymbol{P}'$ which has the same root as $\mathcal{T}_P$, which for every pattern-matching variable v occurring in $\boldsymbol{P}'$ has the same leaf node labeled v as $\mathcal{T}_P$, and whose inner nodes are not nodes of $\mathcal{T}$. Let $\mathcal{T}_{Q'}$ be a well-formed tree associated with $\boldsymbol{Q}'$ which has the same root as $\mathcal{T}_Q$, which for every pattern-matching variable v occurring in $\boldsymbol{Q}'$ has the same leaf node labeled v as $\mathcal{T}_Q$, and whose inner nodes are not nodes of $\mathcal{T}$ *nor of* $\mathcal{T}_{Q'}$. Let $\mathcal{T}_B$ be the tree obtained from $\mathcal{T}$ by rewriting in situ $\mathcal{T}_P$ to $\mathcal{T}_{P'}$, and $\mathcal{T}_C$ the tree obtained from $\mathcal{T}$ by rewriting in situ $\mathcal{T}_Q$ to $\mathcal{T}_{Q'}$. $\mathcal{T}_B$ is associated with the pattern $\boldsymbol{B}$, and $\mathcal{T}_C$ with $\boldsymbol{C}$ respectively. By lemma 3.2(i), $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}_B$. $\mathcal{T}_{Q'}$ has no inner nodes which are nodes of $\mathcal{T}_B$, and thus within $\mathcal{T}_B$ it is possible to rewrite $\mathcal{T}_Q$ in situ to $\mathcal{T}_{Q'}$: let $\mathcal{T}_{B'}$ be the resulting well-formed tree, and $\boldsymbol{B}'$ the pattern associated with it. Symmetrically, by lemma 3.2(i), $\mathcal{T}_P$ is an occurrence of $\boldsymbol{P}$ in $\mathcal{T}_C$, and since $\mathcal{T}_{P'}$ has no inner nodes which are nodes of $\mathcal{T}_C$, $\mathcal{T}_P$ can be rewritten in situ, within $\mathcal{T}_C$, to $\mathcal{T}_{P'}$: let $\mathcal{T}_{C'}$ be the resulting well-formed tree, and $\boldsymbol{C}'$ the pattern associated with it. It is easy to verify that the trees $\mathcal{T}_{B'}$ and $\mathcal{T}_{C'}$ are idendical. Hence the patterns $\boldsymbol{B}'$ and $\boldsymbol{C}'$ are identical, and $\boldsymbol{B}$ and $\boldsymbol{C}$ do rewrite in one step to the same pattern. □

Rewriting algorithm

When a rewriting system $\mathcal{R}$ satisfies the conditions of theorem 3.4, it is particularly easy to compute the normal form of any given pattern. Because $\mathcal{R}$ is left-linear and non-overlapping (orthogonal), it is possible to use the *full substitution*, or *Gross-Knuth* rewriting strategy [35]. Informally speaking, this strategy calls for "reducing at once all the redexes present in the pattern" at each step. Moreover, because the system is right-linear and left-to-right

non-overlapping, no new redexes are introduced by one full substitution step. Hence only one step is necessary!

We make this more precise by describing an algorithm which performs one step of full substitution, and proving that it finds the normal form of the pattern given as input to it when the rewriting system satisfies the conditions of theorem 3.4.

Algorithm 3.1 *Given: a rewriting system $\mathcal{R}$ satisfying the conditions of theorem 3.4. Input: a pattern $\boldsymbol{A}$. Ouput: a pattern $\boldsymbol{A}'$, which purports to be the normal form of $\boldsymbol{A}$.*

1. *If $\boldsymbol{A}$ is of the form $\boldsymbol{P}\theta$ where $\boldsymbol{P}$ is the left-hand side of a rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ of $\mathcal{R}$ and θ is a substitution whose domain V is the set of pattern-matching variables occurring in the rule, return $\boldsymbol{A}' = \boldsymbol{P}'\theta'$, where θ' is the substitution which maps every $v \in V$ to the result $\boldsymbol{B}'_v$ of applying the algorithm recursively to the value $\boldsymbol{B}_v = \theta(v)$.*

Otherwise:

2. *If $\boldsymbol{A}$ is not an instance of the left-hand side of any rule of $\mathcal{R}$, but it is of the form "$(\boldsymbol{f}\ \boldsymbol{B}_1 \ldots \boldsymbol{B}_n)$", where $\boldsymbol{f}$ is a non-terminal symbol and each $\boldsymbol{B}_i$, $1 \leq i \leq n$, is either a terminal symbol or a pattern, then return $\boldsymbol{A}' =$ "$(\boldsymbol{f}\ \boldsymbol{B}'_1 \ldots \boldsymbol{B}'_n)$" where $\boldsymbol{B}'_i = \boldsymbol{B}_i$ if $\boldsymbol{B}_i$ is a terminal symbol, or $\boldsymbol{B}'_i$ is the result of applying the algorithm recursively to $\boldsymbol{B}_i$ if $\boldsymbol{B}_i$ is a pattern.*

3. *If $\boldsymbol{A}$ is a variable pattern, return $\boldsymbol{A}'$=$\boldsymbol{A}$.*

Observe that the algorithm is deterministic. Indeed, since $\mathcal{R}$ is non-overlapping, $\boldsymbol{A}$ can be the left-hand side of at most one rewrite rule. And the algorithm terminates: every recursive call takes as input a proper subpattern of $\boldsymbol{A}$.

Theorem 3.5 *Algorithm 3.1 computes the normal form of its input.*

PROOF. Reasoning by induction, assume that the algorithm is correct on any input of depth less than $k \geq 1$, and let $\boldsymbol{A}$ be a pattern of depth k. Let us show that the pattern $\boldsymbol{A}'$ returned by the algorithm when applied to $\boldsymbol{A}$ is indeed the normal form of $\boldsymbol{A}$. We distinguish the same cases as in the description of the algorithm.

1. Assume that $\boldsymbol{A}$ is of the form $\boldsymbol{P}\theta$ for some rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ of $\mathcal{R}$. Let us use the same notations as in the description of the algorithm. By induction hypothesis, each $\boldsymbol{B}'_v$ is the normal form of $\boldsymbol{B}_v$. Hence each $\boldsymbol{B}_v$ rewrites to $\boldsymbol{B}'_v$, and therefore $\boldsymbol{A} = \boldsymbol{P}\theta$ rewrites to $\boldsymbol{P}\theta'$. Since $\boldsymbol{P}\theta'$ rewrites to $\boldsymbol{P}'\theta' = \boldsymbol{A}'$ be an application of the rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$, if follows that $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$.

   To show that $\boldsymbol{A}'$ is a normal form of $\mathcal{R}$, we reason by contradiction. Assume that a rule $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ applies to $\boldsymbol{A}'$. Then, if $\mathcal{T}$ is a well-formed tree associated with $\boldsymbol{A}'$, there is an occurrence $\mathcal{T}_Q$ of $\boldsymbol{Q}$ in $\mathcal{T}$. There is also an occurrence of $\mathcal{T}_{P'}$ in $\mathcal{T}$ whose root coincides with the root of $\mathcal{T}$. Each leaf node of $\mathcal{T}_{P'}$ is labeled by a pattern-matching variable $v \in V$, and is the root of a subtree $\mathcal{T}_v$ of $\mathcal{T}$ associated with the pattern $\boldsymbol{B}'_v$. Since $\mathcal{R}$ is left-to-right non-overlapping $\boldsymbol{Q}$ and $\boldsymbol{P}'$ do not overlap. Hence the root n of $\mathcal{T}_Q$ cannot be an internal node of $\mathcal{T}_{P'}$. Then n must be a node of $\mathcal{T}_v$ for some $v \in V$. But this means that the rule $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ applies to $\boldsymbol{B}'_v$, which contradicts the fact that $\boldsymbol{B}'_v$ is in normal form.

2. Assume now that $\boldsymbol{A}$ is of the form "$(\boldsymbol{f}\ \boldsymbol{B}_1\ \ldots\ \boldsymbol{B}_n)$" without being an instance of a left-hand side of a rule of $\mathcal{R}$, and let us use again the same notations as in the description of the algorithm. By induction hypothesis, for every i, $1 \leq i \leq n$, if $\boldsymbol{B}_i$ is a pattern (rather than a terminal symbol), then $\boldsymbol{B}'_i$ is the normal form of $\boldsymbol{B}_i$.

   Since each $\boldsymbol{B}_i$ rewrites to $\boldsymbol{B}'_i$, it is clear that $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$. Let $\boldsymbol{C}_1, \ldots, \boldsymbol{C}_m$, $m \leq 1$, be a chain of one step rewrites leading from $\boldsymbol{A} = \boldsymbol{C}_1$ to $\boldsymbol{C}_m = \boldsymbol{A}'$. To show that $\boldsymbol{A}'$ is a normal form we reason by contradiction. Assume that a rule $\boldsymbol{Q} \rightarrow \boldsymbol{Q}'$ of $\mathcal{R}$ applies to $\boldsymbol{A}'$. Let $\mathcal{T}_{A'}$ be a well-formed tree associated with $\boldsymbol{A}'$. There is an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}_{A'}$, and since every $\boldsymbol{B}'_i$ is in normal form, the root of the occurrence must coincide with the root of $\mathcal{T}_{A'}$. This means that $\boldsymbol{A}'$ is an instance of $\boldsymbol{Q}$.

   We have shown that $\boldsymbol{A}' = \boldsymbol{C}_m$ is an instance of the left-hand side $\boldsymbol{Q}$ of a rule of $\mathcal{R}$, and we know that the same is not the case for $\boldsymbol{A} = \boldsymbol{C}_1$. Therefore there must be some j, $1 \leq j < m$, such that $\boldsymbol{C}_{j+1}$ is an instance of $\boldsymbol{Q}$ but $\boldsymbol{C}_j$ is not. $\boldsymbol{C}_j$ rewrites to $\boldsymbol{C}_{j+1}$ by an application of a rule $\boldsymbol{R} \rightarrow \boldsymbol{R}'$ of $\mathcal{R}$. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{C}_j$. There exists an occurrence $\mathcal{T}_R$ of $\boldsymbol{R}$ in $\mathcal{T}$ which, when rewritten,

yields a tree associated with $\boldsymbol{R}'$. Since $\boldsymbol{R} \rightarrow \boldsymbol{R}'$ is a permuting rule, $\mathcal{T}_R$ can be rewritten in situ; let $\mathcal{T}'$ be a tree for $\boldsymbol{C}_{j+1}$ obtained by an in-situ rewrite, and $cal T_{R'}$ the occurrence of $\boldsymbol{R}'$ in $\mathcal{T}'$ to which $\mathcal{T}_R$ rewrites. Since $\boldsymbol{C}_{j+1}$ is an instance of $\boldsymbol{Q}$, there exists an occurrence $\mathcal{T}_Q$ of $\boldsymbol{Q}$ in $\mathcal{T}'$ whose root is the root of $\mathcal{T}'$. Since $\mathcal{R}$ is left-to-right non-overlapping, $\boldsymbol{Q}$ and $\boldsymbol{R}'$ do not overlap, and hence their respecitve occurrences $\mathcal{T}_Q$ and $\mathcal{T}_{R'}$ do not overlap. Since $\boldsymbol{Q}$ is a linear pattern, lemma 3.2(ii) applies, and asserts that $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ also in the tree $\mathcal{T}$. It is clear that the root of $\mathcal{T}$ is the same as the root of $\mathcal{T}'$. Thus $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ at the root of $\mathcal{T}$, which means that $\boldsymbol{C}_j$ is an instance of $\boldsymbol{Q}$, a contradiction.

3. Finally, assume that $\boldsymbol{A}$ is a variable pattern "v". The output of the algorithm is $\boldsymbol{A}$ itself. Since non-variable patterns are not allowed as left-hand sides of rules, $\boldsymbol{A}$ is trivially a normal form, and the algorithm is also correct in this case.

□

Observe that the algorithm performs at most as many rewrites as there are internal nodes in $\boldsymbol{A}$. Deciding whether a given pattern has an occurrence rooted at a given node of a well-formed tree can done in constant time if the pattern is linear. Rewriting an occurrence can also be done in constant time by pointer manipulations. Hence, for a given rewriting system $\mathcal{R}$, the algorithm runs in linear time with respect to the size of $\boldsymbol{A}$.

3.4.3 Language translation by rewriting

So far, in section 3.4.2, we have studied rewriting as an operation on patterns, without reference to any particular *language*. Recall that in section 3.2 we defined a language as the set of labeled expressions (ground patterns) in the substitution closure of some set of patterns, which are said to generate the language. To answer questions 3, 4 and 5 of section 3.4.1, we need a few results concerning language generation, and the interaction between language generation and rewriting.

This section remains in the abstract setting introduced at the beginning of section 3.4.2.

We begin by characterizing languages in terms of parse trees of labeled expressions.

Let $\mathcal{T}$ be a well-formed tree whose set of nodes is N, and whose set of non-variable nodes is $N' \subseteq N$. Let Σ be a set of occurrences of non-variable patterns in $\mathcal{T}$. We shall say that Σ is a *tessellation* of $\mathcal{T}$ iff the sets of non-variable nodes of the elements of Σ form a partition of N'. In the special case where $\mathcal{T}$ is associated with a ground pattern (i.e. when it is the parse tree of a labeled expression), Σ is a tessellation of $\mathcal{T}$ iff the sets of non-variable nodes of the elements of Σ form a partition of N.

Let now Π be a set of non-variable patterns and $\mathcal{T}$ a well-formed tree. We shall say that Π *tessellates* $\mathcal{T}$ iff there exists a tessellation Σ of $\mathcal{T}$ whose elements are occurrences of patterns which are elements of Π. We shall say that Π tessellates a pattern $\boldsymbol{A}$ iff it tessellates at least one of the trees associated with $\boldsymbol{A}$; in which case it tessellates them all, since they are all isomorphic.

Observe that a well-formed tree consisting of a single leaf node labeled by a pattern-matching variable has an empty set of non-variable nodes, and hence is trivially tessellated by any set of non-variable patterns. Thus a variable pattern is trivially tessellated by any set of non-variable patterns.

Lemma 3.6 *A non-variable pattern $\boldsymbol{A}$ belongs to the subsitution closure of a set of non-variable patterns Π iff it is tessellated by Π. (As a special case, a labeled expression $\boldsymbol{A}$ belongs to the language generated by a set of non-variable patterns Π iff it is tessellated by Π.)*

PROOF. The set of non-variable patterns tessellated by Π is clearly stable by substitution, and includes Π. Hence it is a superset of the substitution closure of Π.

We prove the converse by induction on the depth of $\boldsymbol{A}$. Let $k \geq 1$ and assume that every non-variable pattern of depth less than k which is tessellated by Π is in the substitution closure of Π. Let $\boldsymbol{A}$ be a non-variable pattern of depth k tessellated by Π. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{A}$, and Σ a tessellation of $\mathcal{T}$ by occurrences of patterns which are elements of Π. One of these occurences, $\mathcal{T}_0$, must be rooted at the root n of $\mathcal{T}$. Let V be the set (possibly empty) of pattern-matching variables which label the leaf nodes of $\mathcal{T}_0$. Let $\boldsymbol{P}$ be the pattern of Π associated with $\mathcal{T}_0$. Since $\mathcal{T}_0$ is an occurrence of $\boldsymbol{P}$ rooted at n, $\mathcal{T}$ is a θ-graft of $\mathcal{T}_0$, θ being a

substitution with domain V. Hence $\boldsymbol{A} = \boldsymbol{P}\theta$. Let v be an arbitrary element of V (if V is not empty), and let m be a leaf node of $\mathcal{T}_0$ labeled by v in $\mathcal{T}_0$.

Assume that the image of v by θ is a non-variable pattern $\boldsymbol{A}'$, and let $\mathcal{T}'$ be the occurrence of $\boldsymbol{A}'$ in $\mathcal{T}$ rooted at m. Let $\mathcal{T}_1 \in \Sigma$ be an occurrence of a pattern $\boldsymbol{Q} \in \Pi$ which has a non-variable node in common with $\mathcal{T}'$. $\mathcal{T}_1$ has no non-variable node in common with $\mathcal{T}_0$, since they are both in the tessellation Σ of $\mathcal{T}$. But then the root of $\mathcal{T}_1$ (which is a non-variable node of $\mathcal{T}_1$ since $\boldsymbol{Q}$ is a non-variable pattern) is in $\mathcal{T}'$, and $\mathcal{T}_1$ is also an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$. This means that the elements of Σ which have non-variable nodes in common with $\mathcal{T}'$ form a tessellation Σ' of $\mathcal{T}'$, by patterns which are elements of Π. Since the depth of $\mathcal{T}'$ is less than k, by induction hypothesis $\boldsymbol{A}'$ is in the substitution closure of Π.

Thus every non-variable pattern in the range of θ is in the substitution closure of Π. Therefore $\boldsymbol{A} = \boldsymbol{P}\theta$ belongs to the substitution closure of Π. □

We shall say that a set of non-variable patterns Π is *non-overlapping* iff distinct patterns of Π do not overlap and no pattern of Π is self-overlapping. The relationship with the notion of a non-overlapping rewriting system is as follows: if a rewriting system $\mathcal{R}$ is non-overlapping, then the set of its left-hand side patterns is non-overlapping; conversely, if the set of left-hand side patterns of $\mathcal{R}$ is non-overlapping and, in addition, distinct rules have distinct left-hand sides, then $\mathcal{R}$ is non-overlapping.

Lemma 3.7 *Let Π be a non-overlapping set of non-variable patterns and $\mathcal{T}$ a well-formed tree having a tessellation Σ by occurrences of patterns which are elements of Π. Then every occurrence of a pattern $\boldsymbol{P} \in \Pi$ in $\mathcal{T}$ is an element of Σ.*

PROOF. Indeed let $\mathcal{T}_P$ be an occurrence of $\boldsymbol{P}$ in $\mathcal{T}$. Since $\boldsymbol{P}$ is a non-variable pattern, the set of non-variable nodes of $\mathcal{T}_P$ is non-empty and, being included in the set of non-variable nodes of $\mathcal{T}$, it intersects the set of non-variable nodes of an occurrence $\mathcal{T}_Q \in \Sigma$ of some pattern $\boldsymbol{Q} \in \Pi$. By lemma 3.1, $\mathcal{T}_P$ and $\mathcal{T}_Q$ overlap. Hence $\boldsymbol{P}$ and $\boldsymbol{Q}$ overlap. This means, since they are both elements of Π, that they coincide. But then $\mathcal{T}_P$ and $\mathcal{T}_Q$ must overlap at the root, otherwise $\boldsymbol{P}$ would be self-overlapping. Hence the occurrences $\mathcal{T}_P$ and $\mathcal{T}_Q$ coincide. □

A consequence of this lemma is that a non-overlapping set of non-variable patterns Π determines a unique tessellation by occurrences of patterns of Π

of every well-formed tree associated with every pattern in the substitution closure of Π (and in particular, of every parse tree of every labeled expression in the language generated by Π). We shall write

$$\text{tess}_\Pi(\mathcal{T})$$

for the tessellation of such a tree $\mathcal{T}$ determined by Π.

We come now to the interaction of language generation and rewriting.

Lemma 3.8 *Let Π be a non-overlapping set of linear non-variable patterns and let $\mathcal{R}$ be a reversible, left-linear, right-linear rewriting system such that for every rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ of $\mathcal{R}$ both $\boldsymbol{P}$ and $\boldsymbol{P}'$ are in the substitution closure of Π. Let $\mathcal{T}$ be a well-formed tree associated with some pattern in the substitution closure of Π, which has an occurrence $\mathcal{T}_P$ of the left-hand side $\boldsymbol{P}$ of a rule $\boldsymbol{P} \rightarrow \boldsymbol{P}'$ of $\mathcal{R}$. Let $\mathcal{T}'$ be a tree obtained from $\mathcal{T}$ by rewriting $\mathcal{T}_P$ in situ, and let $\mathcal{T}_{P'}$ be the occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$ resulting from the rewrite. Then*

$$(\text{tess}_\Pi(\mathcal{T}) - \text{tess}_\Pi(\mathcal{T}_P)) \cup \text{tess}_\Pi(\mathcal{T}_{P'})$$

is a tessellation of $\mathcal{T}'$ by occurrences of patterns of Π.

PROOF. First observe that every rule of $\mathcal{R}$ is a permuting rule, and that hence the rewrite of $\mathcal{T}_P$ can indeed be done in situ.

Let N be the set of non-variable nodes of $\mathcal{T}$, N_1 the set of non-variable nodes of $\mathcal{T}_P$, and $N_0 = N - N_1$. Then N_0 and N_1 partition N. Let N' be the set of non-variable nodes of $\mathcal{T}'$ and N_1' the set of non-variable nodes of $\mathcal{T}_{P'}$. From the definition of an in-situ rewrite, it follows that $N_0 = N - N_1 = N' - N_1'$.

Let $\Sigma = \text{tess}_\Pi(\mathcal{T})$ and $\Sigma_1 = \text{tess}_\Pi(\mathcal{T}_P)$. An occurrence of a pattern in $\mathcal{T}_P$ is an occurrence of the same pattern in $\mathcal{T}$, and thus $\Sigma_1 \subseteq \Sigma$. Let $\Sigma_0 = \Sigma - \Sigma_1$. Since the sets of non-variable nodes of elements of Σ_1 partition N_1, and the sets of non-variable nodes of elements of Σ partition N, the sets of non-variable nodes of elements of Σ_0 partition N_0. Let $\Sigma_1' = \text{tess}_\Pi(\mathcal{T}_{P'})$ and

$$\Sigma' = \Sigma_0 \cup \Sigma_1' = (\text{tess}_\Pi(\mathcal{T}) - \text{tess}_\Pi(\mathcal{T}_P)) \cup \text{tess}_\Pi(\mathcal{T}_{P'})$$

Since the sets of non-variable nodes of elements of Σ_0 partition N_0, and the sets of non-variable nodes of elements of Σ_1' partition N_1', the sets of non-variable nodes of elements of Σ' partition N'.

It remains to show that every element of Σ' is an occurrence in $\mathcal{T}'$ of a pattern which is an element of Π. This is clear for the elements of Σ'_1. Let now $\mathcal{T}_Q \in \Sigma_0$ be an occurrence of a pattern $\boldsymbol{Q} \in \Pi$ in $\mathcal{T}$. The non-variable nodes of $\mathcal{T}_Q$ are not elements of N_1. Hence, by lemma 3.1, $\mathcal{T}_Q$ does not overlap $\mathcal{T}_P$. But then, by lemma 3.2(i), $\mathcal{T}_Q$ is an occurrence of $\boldsymbol{Q}$ in $\mathcal{T}'$. (Observe that this makes use of the fact that $\boldsymbol{Q}$ is a linear pattern.) □

Theorem 3.9 *Let Π be a non-overlapping set of non-variable patterns and let $\mathcal{R}$ be a reversible, left-linear, right-linear rewriting system such that for every rule $\boldsymbol{P} \to \boldsymbol{P}'$ of $\mathcal{R}$ both $\boldsymbol{P}$ and $\boldsymbol{P}'$ are in the substitution closure of Π. If $\boldsymbol{A}$ is a pattern in the substitution closure of Π, and $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$ in $\mathcal{R}$, then $\boldsymbol{A}'$ is also in the substitution closure of Π. (As a special case, if $\boldsymbol{A}$ is a labeled expression in the language generated by Π, and $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$ in $\mathcal{R}$, then $\boldsymbol{A}'$ is also in the language generated by Π'.)*

PROOF. It suffices to prove this when $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$ in one step, by application of a rewrite rule $\boldsymbol{P} \to \boldsymbol{P}'$ of $\mathcal{R}$. Let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{A}$. There exists a redex $(\mathcal{T}_P, \boldsymbol{P} \to \boldsymbol{P}')$ of $\mathcal{R}$ in $\mathcal{T}$ which, when reduced, yields a well-formed tree associated with $\boldsymbol{A}'$; and since $\mathcal{R}$ consists of permuting rules, the reduction can be done in-situ. Let then $\mathcal{T}'$ be a well-formed tree associated with $\boldsymbol{A}'$ resulting from an in-situ rewrite of $\mathcal{T}_P$, and let $\mathcal{T}_{P'}$ be the occurrence of $\boldsymbol{P}'$ in $\mathcal{T}'$ resulting from the rewrite. By lemma 3.8, the set:

$$(\mathrm{tess}_\Pi(\mathcal{T}) - \mathrm{tess}_\Pi(\mathcal{T}_P)) \cup \mathrm{tess}_\Pi(\mathcal{T}_{P'})$$

is a tessellation of $\mathcal{T}'$ by occurrences of patterns of Π. Hence by lemma 3.6 $\boldsymbol{A}'$ is in the substitution closure of Π. □

3.4.4 Adequacy of the translation system

We return now to the concrete setting of the typed λ-language and the surface language which extends it, and we are ready to answer the questions that were raised at the end of section 3.4.1.

Let Π_{TLL} be the set of basic patterns of the typed λ-language described in section 3.2. Let Π_{SHORT} be the set of patterns which define the shorthands, i.e. the patterns used as left-hand sides of $\mathcal{R}_{\mathrm{ELIM}}$ and right-hand sides of

$\mathcal{R}_{\text{INTRO}}$; and let Π_{REPR} be the set of patterns used as representation of the shorthands, i.e. the patterns used as right-hand sides of $\mathcal{R}_{\text{ELIM}}$ and left-hand sides of $\mathcal{R}_{\text{INTRO}}$. The typed λ-language is the language generated by Π_{TLL} and the surface language is the language generated by $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$.

Since the rewrite rules are customized by the user, $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are not known in advnace. But we make the following stipulations:

1. The rules of $\mathcal{R}_{\text{ELIM}}$ consist, as is the case in the sample rewriting system of table 3.1, of rules for the primitive notations, rules for non-primitive notations, and rules that allow the omission of the subscript ι from variables in the surface language (rule schemas 14 and 15 in table 3.1).

2. Except for the subscript omission rules, each rule of $\mathcal{R}_{\text{ELIM}}$ has been constructed following the recipe given in section 2.5.2. This means in particular that:

   (a) The same pattern-matching variables occur in the left-hand side and the right-hand side. (This has allowed us to assert that $\mathcal{R}_{\text{ELIM}}$ is reversible, and to define $\mathcal{R}_{\text{INTRO}}$.)

   (b) Each pattern-matching variable used in the rule has a sort of the form FML_α or VAR_α.

   (c) The left-hand side and the right-hand side have sorts of the form FML_α. (The type α is o for a notation playing the syntactic role of a sentence, or ι for a notation playing the syntactic role of a term; it can also be some other type, as is the case for the shorthands used for higher-order description operators.)

   (d) The right-hand side is a pattern in the substitution closure of Π_{TLL}, of the form:

   $$c\ \boldsymbol{ARG}_1\ \ldots\ \boldsymbol{ARG}_n$$

   i.e., with explicit parentheses and labeling by non-terminal symbols:

   $$(\text{FML}_o \ldots (\text{FML}_{o\alpha_n \ldots \alpha_2} (\text{FML}_{o\alpha_n \ldots \alpha_1}\ c)\ \boldsymbol{ARG}_1) \ldots \boldsymbol{ARG}_n)$$

   where c is the representing constant of the notation.[10] Distinct rules have distinct representing constants.

[10] $ARG_1 \ldots ARG_n$ are as described in section 2.5.2, but with pattern-matching variables instead of syntactic parameters.

(e) Each pattern-matching variable used in a rule of $\mathcal{R}_{\text{ELIM}}$ occurs exactly once in the right-hand side.

(f) If the left-hand side has internal structure, i.e. if it has proper non-variable subpatterns, then the non-terminals other than the topmost one are *not* of sort FML_α or VAR_α.

3. Each pattern-matching variable used in the rule occurs exactly once in the left-hand side. (Otherwise a syntactic parameter would be repeated twice in the notation, which is unnecessary, and rare in ordinary notations.)

4. Patterns describing distinct notations are not the same up to renaming of variables. Patterns describing notations are those of Π_{TLL}, which describe the basic notations of the typed λ-language, and those of Π_{SHORT}, which describe the additional notations introduced by the rewrite rules. This stipulation means that Π_{TLL} and Π_{SHORT} are disjoint, that elements of Π_{SHORT} which are left-hand sides of distinct rules of $\mathcal{R}_{\text{ELIM}}$ are distinct, and that distinct patterns which are elements of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ cannot be identified by renaming variables.

These stipulations imply that Π_{SHORT} and Π_{REPR} are sets of linear non-variable patterns. Observe that Π_{TLL} is also a set of linear non-variable patterns.

Because distinct notations have distinct representing constants, elements of Π_{REPR} which are right-hand sides of distinct rules of $\mathcal{R}_{\text{ELIM}}$, i.e. left-hand sides of distinct rules of $\mathcal{R}_{\text{INTRO}}$, are distinct. Moreover, it is clear from the general form of those patterns of Π_{REPR} constructed according to the recipe of section 2.5.2, and from the form of the subscript omission rules, that distinct patterns which are elements of Π_{REPR} do not overlap, and that no element of Π_{REPR} is self-overlapping. Thus Π_{REPR} is non-overlapping.

Consider now the set of patterns describing notations of the surface language, $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$. The patterns of Π_{TLL} have no proper subpatterns; and if a pattern $\boldsymbol{P}$ of Π_{SHORT} has a proper subpattern $\boldsymbol{P}'$, then the sort of $\boldsymbol{P}'$ is not of the form FML_α or VAR_α. Every pattern of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$, on the other hand, has a sort of the form FML_α or VAR_α. Hence no pattern of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ can have a common instance with a proper subpattern of a pattern of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$.

This means, first, that no pattern of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ is self-overlapping. It also means that distinct patterns of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ do not overlap. Indeed, assume that two patterns $\boldsymbol{P}, \boldsymbol{P}' \in \Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ overlap, and let us prove that they are identical.

Since neither of $\boldsymbol{P}$ and $\boldsymbol{P}'$ can have a common instance with a proper subpattern of the other, they themselves must have a common instance $\boldsymbol{A}$:

$$\boldsymbol{A} = \boldsymbol{P}\theta = \boldsymbol{P}'\theta'$$

where θ and θ' are substitutions. Consider a parse tree $\mathcal{T}$ associated with $\boldsymbol{A}$, let r be the root of $\mathcal{T}$, and let $\mathcal{T}_P$ and $\mathcal{T}_{P'}$ be the occurrences of $\boldsymbol{P}$ and $\boldsymbol{P}'$ rooted at r. A node n of $\mathcal{T}$ cannot be both a variable node of $\mathcal{T}_P$ and an internal node of $\mathcal{T}_{P'}$, or viceversa, since the variable nodes of $\mathcal{T}_P$ and $\mathcal{T}_{P'}$ are labeled by pattern-matching variables with sorts of the form FML_α or VAR_α, while their internal nodes other than the root are labeled by non-terminal symbols which are not of the form FML_α or VAR_α. Hence $\mathcal{T}_P$ and $\mathcal{T}_{P'}$ have the same set N of varaible nodes. But then, since $\boldsymbol{P}$ and $\boldsymbol{P}'$ do not have repeated occurrences of variables, the set of pairs (v, v') where v and v' are the labels of n in $\mathcal{T}_P$ and $\mathcal{T}_{P'}$ for some $n \in N$ is a bijection θ from the set of pattern-matching variables of $\boldsymbol{P}$ onto the set of pattern-matching variables of $\boldsymbol{P}'$. Thus $\boldsymbol{P}$ and $\boldsymbol{P}'$ are the same up to renaming of variables. By the above stipulations, they must then coincide.

We have shown that no pattern of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ is self-overlapping, and that distinct patterns of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ do not overlap. Thus the set of non-variable patterns $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ is non-overlapping. Also, since the patterns of Π_{REPR} are in the substitution closure of Π_{TLL}, it is easy to see that no pattern of Π_{REPR} overlaps any pattern of Π_{SHORT}.

Since Π_{SHORT} and Π_{REPR} are sets of linear patterns, the reversible systems $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are left and right-linear. Since no pattern of Π_{REPR} overlaps any pattern of Π_{SHORT}, they are left-to-right non-overlapping. Hence, by theorem 3.3, they are both terminating. Moreover, since Π_{SHORT}, as a subset of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$, is non-overlapping, and distinct rules of $\mathcal{R}_{\text{ELIM}}$ have distinct left-hand sides, $\mathcal{R}_{\text{ELIM}}$ is non-overlapping. And since Π_{REPR} is non-overlapping, and distinct rules of $\mathcal{R}_{\text{INTRO}}$ have distinct left-hand sides, $\mathcal{R}_{\text{INTRO}}$ is also non-overlapping. Therefore, by theorem 3.4, both $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are convergent.

We have thus answered questions 1 and 2 of section 3.4.1. Every chain of rewrites, in either direction, terminates. And every pattern has a unique

normal form both for $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$. Hence if a labeled expression is rewritten, in either direction, until no more rules apply, then the result will be independent of the choices made during the rewriting process regarding which rewrite to perform next.

The fact that the rewriting systems $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ satisfy the conditions of theorem 3.4 also means that algorithm 3.1 is applicable in both cases. Since the running time of the algorithm is linear in the size of the expression being rewritten, this provides at least a theoretical answer to question 6. In fact, experience shows that the speed of the rewriting algorithm is indeed perfectly adequate for interactive use. The rewriting time is negligible compared with the parsing time or the pretty-printing time.

The answer to questions 3 and 4 is provided by theorem 3.9. It is not the case, of course, that an *arbitrary* labeled expression is rewritten by $\mathcal{R}_{\text{ELIM}}$ to an expression of the typed λ-language, or by $\mathcal{R}_{\text{INTRO}}$ to an expression of the surface language. But consider an expression *of the surface language.* The set of non-variable patterns $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$, which generates the surface language, is non-overlapping; the patterns used as left-hand sides and right-hand sides of $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$, the elements of Π_{SHORT} and Π_{REPR}, are in the substitution closure of $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$; and the rewriting systems $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ are reversible, left-linear and right-linear. Hence they both rewrite an expression of the surface language to an expression of the surface language. Moreover, consider an expression $\boldsymbol{A}$ of the surface language which is in normal form for $\mathcal{R}_{\text{ELIM}}$. By lemma 3.6, $\boldsymbol{A}$ is tessellated by $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$; but since $\boldsymbol{A}$ is in normal form, a parse tree for $\boldsymbol{A}$ has no occurrences of patterns which are elements of Π_{SHORT}. Hence $\boldsymbol{A}$ is tessellated by Π_{TLL}, and thus it is an expression of the typed λ-language. Therefore the direct rewriting process takes an expression of the surface language to an expression of the typed λ-language.

It remains to answer question 5.

An expression of the surface language which is in normal form for $\mathcal{R}_{\text{ELIM}}$ is an expression of the typed λ-language. Conversely, an expression $\boldsymbol{A}$ of the typed λ-language is in normal form for $\mathcal{R}_{\text{ELIM}}$. Indeed, let $\mathcal{T}$ be a parse tree of $\boldsymbol{A}$. By lemma 3.6, there is a tessellation Σ of $\mathcal{T}$ by occurrences of patterns of Π_{TLL}. Since the set of non-variable patterns $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ is non-overlapping, by lemma 3.7, any occurrence of an element of Π_{SHORT} in

$\mathcal{T}$ must belong to Σ. Given that Π_{TLL} and Π_{SHORT} are disjoint, this means that there are no occurrences of patterns of Π_{SHORT} in $\mathcal{T}$. Thus no rule of $\mathcal{R}_{\text{ELIM}}$ applies to $\boldsymbol{A}$.

Consider then the process of rewriting an expression $\boldsymbol{A}$ of the typed λ-language with $\mathcal{R}_{\text{INTRO}}$ until a normal form $\boldsymbol{A}'$ for $\mathcal{R}_{\text{INTRO}}$ is found, and then rewriting $\boldsymbol{A}'$ back with $\mathcal{R}_{\text{ELIM}}$ until a normal form $\boldsymbol{A}''$ for $\mathcal{R}_{\text{ELIM}}$ is found. Since $\mathcal{R}_{\text{INTRO}}$ rewrites $\boldsymbol{A}$ to $\boldsymbol{A}'$, $\mathcal{R}_{\text{ELIM}}$ rewrites $\boldsymbol{A}'$ to $\boldsymbol{A}$. And since $\boldsymbol{A}$ is in normal form, $\boldsymbol{A}'' = \boldsymbol{A}$.

Consider the opposite process: an expression $\boldsymbol{A}$ of the surface language if rewritten with $\mathcal{R}_{\text{ELIM}}$ until a normal form $\boldsymbol{A}'$ for $\mathcal{R}_{\text{ELIM}}$ is found. We have seen that $\boldsymbol{A}'$ is an expression of the typed λ-language. Then $\boldsymbol{A}'$ is rewritten with $\mathcal{R}_{\text{INTRO}}$ until a normal form $\boldsymbol{A}''$ for $\mathcal{R}_{\text{INTRO}}$ is found. Informally speaking, if $\boldsymbol{A}$ contains subexpressions which could have been written using shorthands but are written instead in internal representation, then $\boldsymbol{A}''$ will differ from $\boldsymbol{A}$, since in $\boldsymbol{A}''$ the shorthands will have been used instead. But if this is not the case, i.e., formally, if $\boldsymbol{A}$ is in normal form for $\mathcal{R}_{\text{INTRO}}$, then by the same argument as above, $\boldsymbol{A}'' = \boldsymbol{A}$.

Thus $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ define a bijective correspondance between the typed λ-language and the set of expressions of the surface language which are in normal form for $\mathcal{R}_{\text{INTRO}}$.

Let now $\boldsymbol{A}$ be a labeled expression consisting only of shorthands, i.e. an expression of the language generated by Π_{SHORT}. By lemma 3.6, a parse tree $\mathcal{T}$ of $\boldsymbol{A}$ has a tessellation Σ by occurrences of elements of Π_{SHORT}. Since a pattern which is an element of Π_{REPR} does not overlap any pattern which is an element Π_{SHORT}, no pattern in Π_{REPR} can have an occurrence in $\mathcal{T}$. Hence $\boldsymbol{A}$ is in normal form for $\mathcal{R}_{\text{INTRO}}$. By a similar argument, any expression of the language generated by Π_{REPR} is in normal form for $\mathcal{R}_{\text{ELIM}}$.

Conversely, let $\boldsymbol{A}$ be an expression of the language generated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$ which is in normal form for $\mathcal{R}_{\text{ELIM}}$. A parse tree of $\boldsymbol{A}$ is tessellated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$, but can have no occurrences of patterns which are elements of Π_{SHORT}, hence it is tessellated by Π_{REPR}. Thus $\boldsymbol{A}$ is an expression of the language generated by Π_{REPR}. Similarly, an expression of the language generated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$ which is in normal form for $\mathcal{R}_{\text{INTRO}}$ is in the language generated by Π_{SHORT}.

The set $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$ consists of linear non-variable patterns, and is nonoverlapping. Hence, by theorem 3.9, if an expression $\boldsymbol{A}$ of the language

generated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$ rewrites to $\boldsymbol{A}'$ in either $\mathcal{R}_{\text{ELIM}}$ or $\mathcal{R}_{\text{INTRO}}$, then $\boldsymbol{A}'$ belongs also to the language generated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$.

Let then $\boldsymbol{A}$ be an expression of the language generated by Π_{SHORT}, let $\boldsymbol{A}'$ be its normal form for $\mathcal{R}_{\text{ELIM}}$, and let $\boldsymbol{A}''$ be the normal form of $\boldsymbol{A}'$ for $\mathcal{R}_{\text{INTRO}}$. Since $\boldsymbol{A}$ rewrites to $\boldsymbol{A}'$, $\boldsymbol{A}'$ is in the language generated by $\Pi_{\text{SHORT}} \cup \Pi_{\text{REPR}}$, and hence, since $\boldsymbol{A}'$ is in normal form for $\mathcal{R}_{\text{ELIM}}$, it is in the language generated by Π_{REPR}. Furthermore, $\boldsymbol{A}$ is in normal form for $\mathcal{R}_{\text{INTRO}}$, and $\boldsymbol{A}'$ rewrites to $\boldsymbol{A}$ in $\mathcal{R}_{\text{INTRO}}$. Hence $\boldsymbol{A}'' = \boldsymbol{A}$. Similarly, a complete rewrite by $\mathcal{R}_{\text{INTRO}}$ (i.e. a rewrite to normal form) takes an expression $\boldsymbol{B}$ of the language generated by Π_{REPR} to an expression $\boldsymbol{B}'$ of the language generated by Π_{SHORT}, and then a complete rewrite by $\mathcal{R}_{\text{ELIM}}$ takes $\boldsymbol{B}'$ back to $\boldsymbol{B}$. Thus $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ define a bijective correspondance between the language generated by Π_{SHORT} (the set of labeled expressions consisting only of shorhtands) and the language generate by Π_{REPR}.

An important subset of the surface language is the set of expressions which consist only of "ordinary", or "first-order" mathematical notations. To make this notion precise we define $\mathcal{R}_{\text{FO}}$ as the subset of $\mathcal{R}_{\text{ELIM}}$ consisting of the following rules:

1. The rules that allow the omission of the subscript ι from variables in the surface language (rule schemas 14 and 15 in table 3.1).

2. Those other rules of $\mathcal{R}_{\text{ELIM}}$ whose sides are of sort FML_o or FML_ι, and which contain no pattern-matching variables or sort other than VAR_ι, FML_ι, or FML_o. We shall refer to the notations encoded by these rules as "ordinary notations". An ordinary notation is thus one which behaves syntactically as a term or a sentence, and whose syntactic parameters are individual variables, terms, or sentences. Observe that these notations have representing constants with types of order 0, 1 or 2.

Let $\Pi_{\text{SHORT-FO}}$ be the set of left-hand sides of rules of $\mathcal{R}_{\text{FO}}$, and $\Pi_{\text{REPR-FO}}$ the set of right-hand sides. We can then define the "language of ordinary notations" as the language generated by $\Pi_{\text{SHORT-FO}}$. This language is important because it excludes notations such are higher-order quantification which may be unfamiliar to the user a PDS for set theory. It may be desirable to make it possible for the user to work entirely within this language, by designing the inference toolkit appropriately.

By the same kind of argument as above, it is clear that $\mathcal{R}_{\text{ELIM}}$ and $\mathcal{R}_{\text{INTRO}}$ define a bijective correspondance between the language generated by $\Pi_{\text{SHORT-FO}}$ and the language generated by $\Pi_{\text{REPR-FO}}$.

Using the concepts and results of appendix B, it is possible to give a simple characterization of the language generated by $\Pi_{\text{REPR-FO}}$. This language consists, on one hand, of the labeled expressions of the form "(VAR$_\iota$ ***Id***)", and on the other hand, of the result of labeling with phrase markers certains formulas of the typed λ-language. It can be seen that these formulas coincide with the standard formulas (definition B.5, page 201) of atomic type generated by the set of representing constants of the ordinary notations. But these formulas in turn coincide with the formulas of atomic type in $\beta\gamma$-nf which contain no free symbols other than variables of type ι or constants representing ordinary notations (theorem B.16, page 201).

It follows that any term or sentence of the typed λ-language which contains no free symbols other than individual variables or constants representing ordinary notations is provably equivalent to a formula which can be written using only shorthands.

3.5 Parsing

3.5.1 The parsing problem

In sections 3.3 and 3.4 we have seen how translation from the surface language into the typed λ-language, and from the typed λ-language into the surface language, can be accomplished by rewriting. But rewriting requires labeled expressions, as discussed in section 3.2. In this section we examine the problem of adding parentheses and markers (non-terminal symbols) to an unlabeled expression, in order to turn it into a labeled expression. This is, in our context, the *parsing problem*. By *unlabeled expression* we mean an arbitrary sequence of terminal symbols and delimiters (left or right parentheses). By *labeled expression* we mean as before a pattern (as defined in section 3.2) having no occurrences of pattern-matching variables.

The parsing problem is of course relative to a given language. It can be formulated more precisely as follows. Let us say that a labeled expression $\boldsymbol{A}'$ is a *marking* of an unlabeled expression $\boldsymbol{A}$ iff $\boldsymbol{A}$ can be obtained from $\boldsymbol{A}'$ by erasing all the markers and zero, some or all pairs of matching parentheses.

Let $\mathcal{L}$ be the language generated by a set of non-variable patterns Π. To parse an unlabeled expression $\boldsymbol{A}$ with respect to $\mathcal{L}$ is to find all the markings of $\boldsymbol{A}$ which are elements of the set $\mathcal{L}$. If there are such markings $\boldsymbol{A}$ is *well-formed* with respect to $\mathcal{L}$, otherwise it is *ill-formed.* When $\boldsymbol{A}$ is well-formed, it is *ambiguous* if there are more than one such markings, *unambiguous* if there is only one.

Our parsing problem is related to the more traditional problem of parsing an expression with respect to a context-free grammar as follows. We shall refer to a non-variable pattern which has no proper non-variable subpatterns (i.e. which has no internal structure) as a *one-level pattern.* Recall that a *linear* pattern is a pattern which does not have more than one occurrence of any variable. To each linear one-level pattern

$$(f\ \boldsymbol{A}_1\ \ldots\ \boldsymbol{A}_n),$$

(where f is a non-terminal symbol and each $\boldsymbol{A}_i$ is either a terminal symbol or a pattern-matching variable) we associate the two grammar productions:

$$f \longmapsto \boldsymbol{A}'_1\ \ldots\ \boldsymbol{A}'_n$$
$$f \longmapsto (\ \boldsymbol{A}'_1\ \ldots\ \boldsymbol{A}'_n\ )$$

where $\boldsymbol{A}'_i$ is the same as $\boldsymbol{A}_i$ if $\boldsymbol{A}_i$ is a terminal symbol, and $\boldsymbol{A}'_i$ is the sort of $\boldsymbol{A}_i$ if $\boldsymbol{A}_i$ is a pattern-matching variable. If Π is a set of linear one-level patterns, $\mathcal{L}$ is the language generated by Π, Π' is the set of productions associated with the patterns of Π, and $\mathcal{L}'$ is the context-free language generated by the grammar Π', then it is easy to see that $\mathcal{L}'$ coincides with the set of expressions which have markings in $\mathcal{L}$. So to decide whether $\boldsymbol{A}$ has a marking in $\mathcal{L}$ is the same as to decide whether it is an expression of the context-free language $\mathcal{L}'$. Moreover a parse tree of $\boldsymbol{A}$ with respect to the context-free grammar determines a marking, and conversely a marking determines a parse tree (up to isomorphism). Thus the problem of parsing with respect to the language generated by a set of linear one-level patterns reduces to the problem of parsing with respect to a context-free grammar.

The languages that we are interested in are the surface language and the typed λ-language, which is contained in the surface language. Recall that the typed λ-language is generated by the set of non-variable patterns Π_{TLL}, and the surface language by $\Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$. Let $\Pi_{\text{SURF}} = \Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$. Two potential difficulties arise regarding parsing with respect to these languages:

1. While the elements of Π_{TLL} are linear one-level patterns, the same in not necessarily the case for Π_{SHORT}.

2. The patterns of Π_{TLL} are described (page 1) by pattern schemas, parameterized by schematic type variables. Notations for higher-order quantifiers and descriptors also require type parameters. Thus Π_{SURF} is infinite. If it consisted only of linear one-level patterns an equivalent grammar could be derived from it, but it would be an infinite one.

The second difficulty is not in fact a practical one. Type variables can be used in a straightforward way by the parser, so that there is not much practical difference between a pattern schema with type parameters and an ordinary pattern.

The first difficulty has been avoided in the first version of Watson, described in chapter 4, by ruling out patterns with internal structure as left-hand sides of rules of $\mathcal{R}_{\text{ELIM}}$. It is theoretically possible, however, to reduce the problem of parsing with respect to Π_{SURF} to the context-free parsing problem, even when internal strucutre is allowed in the elements of Π_{SURF}. In the remainder of this section we show how this can be done.

A *projection* of a set of non-variable patterns Π is a set of linear, one-level patterns obtained as follows: take each pattern $\boldsymbol{P}$ which is an element of Π or a subpattern of an element of Π, and replace each (variable or non-variable) subpattern $\boldsymbol{P}'$ of $\boldsymbol{P}$ with a pattern-matching variable of same sort as $\boldsymbol{P}'$, using a different pattern-matching variable each time; then from the resulting set remove any patterns which are the same as others up to renaming of variables.

For example, if Π consists of the two patterns:

$$\begin{array}{l}(\text{FML}_o\ (\text{RQ}\ \forall\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\ P_{\text{FML}_o}) \\ (\text{FML}_o\ (\text{RQ}\ \exists\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\ P_{\text{FML}_o})\end{array} \tag{3.10}$$

then the following patterns:

$$\begin{array}{l}(\text{FML}_o\ Q_{\text{RQ}}\ P_{\text{FML}_o}) \\ (\text{RQ}\ \forall\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota}) \\ (\text{RQ}\ \exists\ x_{\text{VAR}_\iota} \in E_{\text{FML}_\iota})\end{array} \tag{3.11}$$

are the elements of a projection Π' of Π.

Observe that different projections of a given set of non-variable patterns differ only in the names of the pattern-matching variables used in the patterns.

If Π' is a projection of Π, every pattern of Π is in the substitution closure of Π'. Hence the language $\mathcal{L}$ generated by Π is a subset of the language $\mathcal{L}'$ generated by Π'. Therefore to parse an unlabeled expression $\boldsymbol{A}$ with respect to $\mathcal{L}$ one can begin by parsing it with respect to $\mathcal{L}'$. Since Π' is a set of linear one-level patterns, parsing with respect to $\mathcal{L}'$ is the same as parsing with respect to the context-free grammar derived from Π' as described above. The result is the set of markings of $\boldsymbol{A}$ which are elements of the set $\mathcal{L}'$. It can then be determined which of these markings are in the substitution closure of Π, i.e. which of them are also elements of $\mathcal{L}$. In this sense, the problem of parsing with respect to the language generated by an arbitrary set of non-variable patterns reduces to the problem of parsing with respect to a context-free grammar.

In the case of interest to us, if Π_{SURF} contains patterns other than one-level patterns, we can parse an unlabeled expression $\boldsymbol{A}$ with respect to a projection Π_{PROJ} of Π_{SURF}, then check the resulting markings for membership in the surface language. This check may be as simple as inspecting the sort of each marking, and rejecting those whose sorts are auxiliary non-terminal symbols. (Recall that the auxiliary non-terminal symbols are symbols which are not of the form FML_α or VAR_α, used as sorts of any proper non-variable subpatterns of elements of Π_{SHORT}.) Indeed, if the rewrite rules are properly designed, the surface language should coincide with the set of labeled expressions of sort FML_α or VAR_α in the language generated by Π_{PROJ}. We shall not try to give design guidelines that ensure that this is indeed the case. But we shall describe a method for verifying this a posteriori.

Let F be a set of non-terminal symbols. We shall say that a pattern $\boldsymbol{P}$ is *peripherally F-free* iff its sort it not an element of F and it contains no occurrences of pattern-matching variables whose sort is an element of F. For example, let X be the set of symbols which can be used as auxiliary non-terminals, i.e. the set of non-terminal symbols which are not of the form FML_α or VAR_α. Then the set $\Pi_{\text{SURF}} = \Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$ is peripherally X-free.

A set S of patterns is *F-stable* iff for every pattern $\boldsymbol{P}$ in S and every substitution θ whose range consists only of pattern-matching variables and elements of S, and whose domain contains only pattern-matching variables whose sort is in F, the pattern $\boldsymbol{P}\theta$ is also in S. The *F-closure* of a set Π

of non-variable patterns is the smallest F-stable set which includes Π. The *F-contraction* of Π is the set of patterns in the F-closure of Π which are peripherally F-free.

For example, Π' being the set of patterns (3.11), and X being as before the set of auxiliary non-terminal symbols, the X-closure of Π' consists of the following patterns:

$$\begin{array}{l}
(\mathrm{FML}_o\ Q_{\mathrm{RQ}}\ P_{\mathrm{FML}_o}) \\
(\mathrm{RQ}\ \forall\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota}) \\
(\mathrm{RQ}\ \exists\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota}) \\
(\mathrm{FML}_o\ (\mathrm{RQ}\ \forall\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota})\ P_{\mathrm{FML}_o}) \\
(\mathrm{FML}_o\ (\mathrm{RQ}\ \exists\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota})\ P_{\mathrm{FML}_o})
\end{array}$$

and the F-contraction of Π' is the set of two patterns:

$$\begin{array}{l}
(\mathrm{FML}_o\ (\mathrm{RQ}\ \forall\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota})\ P_{\mathrm{FML}_o}) \\
(\mathrm{FML}_o\ (\mathrm{RQ}\ \exists\ x_{\mathrm{VAR}_\iota} \in E_{\mathrm{FML}_\iota})\ P_{\mathrm{FML}_o})
\end{array} \tag{3.12}$$

Theorem 3.10 *Let F be a set of non-terminal symbols, and Π a set of linear non-variable patterns. Then the set of peripherally F-free patterns in the substitution closure of Π coincides with the substitution closure of the F-contraction of Π. Hence the set of labeled expressions of the language generated by Π whose sort is not an element of F coincides with the language generated by the F-contraction of Π.*

In fact the result holds without the requirement that the patterns of Π be linear, but the proof is not as simple, and we do not need the stronger version.

PROOF. It is clear that the F-closure of Π is a subset of the full substitution closure of Π. Hence the F-contraction of Π, and therefore its substitution closure, are also subsets of the substitution closure of Π. Moreover, since every pattern in the F-contraction of Π is peripherally F-free, and the set of peripherally F-free patterns is stable by substitution, every pattern in the substitution closure of the F-contraction of Π is peripherally F-free. Therefore the substitution closure of the F-contraction of Π is a subset of the set of periferally F-free patterns in the substitution closure of Π.

Conversely, let $\boldsymbol{A}$ be an element of the substitution closure of Π which is peripherally F-free, and let $\mathcal{T}$ be a well-formed tree associated with $\boldsymbol{A}$. By

lemma 3.6, $\mathcal{T}$ has a tessellation Σ by occurrences of elements of Π. Recall that the elements of Σ are occurrences in $\mathcal{T}$ of patterns which are elements of Π. We shall say that two such occurrences are *F-connected* iff the root of one of them is labeled by an element of F and coincides with a variable-node of the other. We shall say that a subset Σ' of Σ is *F-connected* iff between any two elements of Σ' there exists a chain of elemets of Σ' were consecutive elements are F-connected. An *F-connected component* of Σ is a maximal F-connected subset of Σ.

Clearly, every F-connected subset of Σ has an element which is closest to the root. Making use of this observation, it is easy to prove by induction that, for every F-connected subset Σ' of Σ, there exists an occurrence $\mathcal{T}'$ in $\mathcal{T}$ of an element $\boldsymbol{A}'$ of the F-closure of Π, such that the set of non-variable nodes of $\mathcal{T}'$ is the union of the sets of non-variable nodes of the elements of Σ'. Moreover, if Σ' is an F-connected component of Σ, then, given that $\boldsymbol{A}$ is periferally F-free, the root of $\mathcal{T}'$ cannot labeled by an element of F, and no variable node of $\mathcal{T}'$ (if any) can be labeled by a pattern-matching variable whose sort is an element of F; thus $\boldsymbol{A}'$ is periferally F-free, and hence it belongs to the F-contraction of Π. Since the F-connected components of Σ partition Σ, the sets of non-variable nodes of the corresponding occurrences of elements of the F-contraction of Π partition the set of non-variable nodes of $\mathcal{T}$, and hence those occurrences themselves constitute a tessellation of $\mathcal{T}$. Therefore, by lemma 3.6, $\boldsymbol{A}$ belongs to the substitution closure of the F-contraction of Π.

The corollary follows immediately from definitions. □

Theorem 3.10 provides the announced method for verifying that the surface language coincides with the set of labeled expressions of the language generated by Π_{PROJ} whose sorts are of the form FML_α or VAR_α. Indeed the latter is also the language generated by the X-contraction of Π_{PROJ}, while the former is the language generated by Π_{SURF}. It suffices then to compute the X-contraction of Π_{PROJ}, which we shall call Π_{CONT}, and verify that the substitution closures of Π_{CONT} and Π_{SURF} coincide.

The closures coincide iff each element of Π_{CONT} is in the closure of Π_{SURF} and viceversa. But recall that $\Pi_{\text{SURF}} = \Pi_{\text{TLL}} \cup \Pi_{\text{SHORT}}$, that the patterns in Π_{TLL} have no proper non-variable subpatterns, and that those in Π_{SHORT} have no proper non-variable subpatterns whose sort is of the form FML_α or VAR_α. On the other hand the patterns in Π_{CONT} are all of sort FML_α or VAR_α. Hence, using lemma 3.6, it is clear that if a pattern $\boldsymbol{A} \in \Pi_{\text{SURF}}$ is in

the closure of Π_{CONT}, then it must be of the form $\boldsymbol{P}\theta$, where $\boldsymbol{P} \in \Pi_{\text{CONT}}$ and θ is a substitution whose range consists entirely of variable patterns. Since $\boldsymbol{A}$ is linear, this means that $\boldsymbol{A}$ is the same as $\boldsymbol{P}$ up to variable renaming. The same argument can be used with the roles of Π_{SURF} and Π_{CONT} reversed. Indeed the sorts of proper non-variable subpatterns of elements of Π_{CONT} are in X by construction, while all the sorts of the elements of Π_{SURF} are of the form FML_α or VAR_α. Hence if an element of Π_{CONT} is in the closure of Π_{SURF}, it must coincide with an element of Π_{SURF} up to variable renaming. Thus the closures of Π_{CONT} and Π_{SURF} coincide iff Π_{CONT} and Π_{SURF} themselves contain the same patterns up to renaming of variables.

When applying this verification method it is sufficient to consider only those patterns of Π_{SURF} which do have internal structure. That is, Π'_{SURF} being the set of such patterns, Π'_{PROJ} the projection of Π'_{SURF} and Π'_{CONT} the X-contraction of Π'_{PROJ}, it is sufficient to verify that Π'_{SURF} and Π'_{CONT} contain the same patterns up the renaming of variables. For, let $\Pi_0 = \Pi_{\text{SURF}} - \Pi'_{\text{SURF}}$ be the set of one-level patterns in Π_{SURF}. Then:

$$\begin{aligned} \Pi_{\text{SURF}} &= \Pi'_{\text{SURF}} \cup \Pi_0 \\ \Pi_{\text{PROJ}} &= \Pi'_{\text{PROJ}} \cup \Pi_0 \\ \Pi_{\text{CONT}} &= \Pi'_{\text{CONT}} \cup \Pi_0 \end{aligned}$$

And $\Pi'_{\text{SURF}} \cup \Pi_0$ coincides with $\Pi'_{\text{CONT}} \cup \Pi_0$ up to renaming of variables iff the same is the case for Π'_{SURF} and Π'_{CONT}.

For example, if the two patterns (3.10) are the only ones in Π_{SURF} having internal structure, it is sufficent to compute the projection (3.11) of (3.10), and the X-contraction (3.12) of (3.11), and check whether (3.10) and (3.12) are the same up to renaming of variables. In this case they are identical.

3.5.2 Bracketing conventions

In this section we further consider the problem of parsing a labeled expression with respect to a language generated by a set Π of linear one-level patterns.

A problem with the parsing paradigm presented so far is that explicit parentheses are required too often. For example, if Π were the surface language generated by the left-hand sides of the rewrite rules given in table 3.1,

page 79, the expressions

$$\boldsymbol{A} \wedge \boldsymbol{B} \wedge \boldsymbol{C}$$
$$\boldsymbol{A} \wedge \boldsymbol{B} \supset \boldsymbol{C}$$
$$\boldsymbol{A} \wedge \boldsymbol{B} \vee \boldsymbol{C}$$
$$\boldsymbol{A} \vee \boldsymbol{B} \wedge \boldsymbol{C}$$

would be ambiguous without parentheses. Generally, if $\boldsymbol{P}_1, \boldsymbol{P}_2 \in \Pi$ are patterns of same sort $\boldsymbol{f}$, if $\boldsymbol{P}_1$ is right-recursive (it ends with a pattern-matching variable of sort $\boldsymbol{f}$), if $\boldsymbol{P}_2$ is left-recursive (it begins with a pattern-matching variable of sort $\boldsymbol{f}$), and if $\boldsymbol{P}$ is obtained by substituting $\boldsymbol{P}_1$ for the first pattern-matching variable of $\boldsymbol{P}_2$, or $\boldsymbol{P}_2$ for the last pattern-matching variable of $\boldsymbol{P}_1$, then any unlabeled expression derived from a labeled expression which is a substitution instance of $\boldsymbol{P}$ will be ambiguous without parentheses.

The usual solution to this is to specify the precedence and associativity of operators. For example, by saying that $\wedge$ is left associative we would imply that the first of the above expressions stands for

$$(\boldsymbol{A} \wedge \boldsymbol{B}) \wedge \boldsymbol{C}.$$

By saying that $\wedge$ has higher precedence than $\supset$, we would imply that the second expression stands for

$$(\boldsymbol{A} \wedge \boldsymbol{B}) \supset \boldsymbol{C}.$$

And by saying that $\wedge$ and $\vee$ are mutually left-associative we would imply that the third and fourth expressions stand for

$$(\boldsymbol{A} \wedge \boldsymbol{B}) \vee \boldsymbol{C}$$
$$(\boldsymbol{A} \vee \boldsymbol{B}) \wedge \boldsymbol{C}$$

The parser of Watson uses a simple convention based on the idea of a *bracketing rule.* A bracketing rule is a triple consisting of two patterns $\Pi_1, \Pi_2 \in \Pi$ and an integer n specifying a position in the second pattern. (Positions start at 1, and there is one for each pattern-matching variable and each terminal symbol in the pattern.) The implied meaning is that parentheses are *required* around an instance of the first pattern when substituted for the pattern-matching variable v which occupies the n-th position in the second pattern.[11] To parse $\boldsymbol{A}$ with respect to a language and a set of bracketing

[11] We could also define a bracketing rule as a pair $(P_1, \mathcal{C})$, where $\mathcal{C}$ is a context (P_2, v).

rules is to compute the markings of $\boldsymbol{A}$ which are part of the language and are compatible with the rules. We shall refer to a bracketing rule $(\boldsymbol{P}_1, \boldsymbol{P}_2, n)$ by saying that parentheses are required around $\boldsymbol{P}_1$ in the n-th position of $\boldsymbol{P}_2$.

This convention is more general than the usual conventions of precedence and associativity. For example, consider parsing with respect to the surface language corresponding to the rewriting system of table 3.1, page 79. To achieve the effect of $\wedge$ being left-associative, we specify that parentheses are required around the left-hand side of rule 6 of table 3.1 (rule for conjunction) in the third position of the same pattern. Then "$\boldsymbol{A} \wedge \boldsymbol{B} \wedge \boldsymbol{C}$" cannot stand for "$\boldsymbol{A} \wedge (\boldsymbol{B} \wedge \boldsymbol{C})$", so it can only stand for "$(\boldsymbol{A} \wedge \boldsymbol{B}) \wedge \boldsymbol{C}$". Similarly, to achieve the effect of $\wedge$ having higher precedence than $\supset$ we specify that parentheses are required around the left-hand side of rule 8 (implication) in the first or third position of the left-hand side of rule 6 (conjunction). To achieve the effect of $\wedge$ and $\vee$ being mutually left-associative, we specify that parentheses are required around each of the left-hand sides of rules 6 (conjunction) and 7 (disjunction) in the first position of the other. Right-associativity and mutual right-associativity are achieved similarly.

An advantage of bracketing rules is that they cover other conventions relative to the use of parentheses as well. For example a bracketing rule can be used to specify that the parentheses around the argument of a set-theoretic function application "$\boldsymbol{F}(\boldsymbol{A})$" are compulsory.

In Watson, which is implemented in Prolog, the set of bracketing rules can be specified as a user-supplied Prolog predicate. But it can also be specified indirectly, by declaring the associativity and precedence of some of the patterns that make up the surface language. (Associativity and precedence can be assigned to any right-recursive or left-recursive pattern, not just to operators; in particular, precedence can be assigned to quantifiers.) Then Watson derives the set of bracketing rules by assuming that parentheses are required, by default, around every right-recursive pattern in the first position of a left-recursive pattern of same sort, and around every left-recursive pattern in the last position of a right-recursive pattern of same sort; and then using the declarations of associativity and precedence to specify exceptions. Compulsory parentheses in $\boldsymbol{F}(\boldsymbol{A})$ and similar conventions can be specified by additional, explicit bracketing rules.

Chapter 4

The proof development system Watson

The proof development system Watson has been implemented to show the feasiblity of using the formal system described in chapter 2, and to gain insights into the task of mechanizing mathematics with a set theoretic foundation. At this point Watson is only a prototype, with a limited linguistic coverage and a bare-bones inference tool-kit. But it has been used to do a proof which has eluded previous efforts in the field of hardware verification.

4.1 The language processor of Watson

4.1.1 Coverage

The user of Watson can easily design his own notations, using mathematical symbols and keywords. There are restrictions, however, in the kind of notations which can be specified. Some of these have been imposed only for the sake of simplicity, and are likely to be lifted in future versions of Watson, while others will be more difficult to overcome. We list them in order of decreasing difficulty:

1. Fraction bars, superscripts, subscripts and other two-dimensional notations are not handled; equivalent linear notations must be used instead.

2. The convention of writing

$$\boldsymbol{A}\ \mathcal{R}\ \boldsymbol{B}\ \mathcal{R}'\ \boldsymbol{C}$$

for

$$(\boldsymbol{A}\ \mathcal{R}\ \boldsymbol{B}) \wedge (\boldsymbol{B}\ \mathcal{R}'\ \boldsymbol{C})$$

where $\mathcal{R}$ and $\mathcal{R}'$ are relation symbols, such as $=$, $<$ or $\leq$, cannot be easily handled by a rewriting system of the kind described in section 3.4.

3. As mentioned before, the linguistic theory of chapter 3 applies to languages where parentheses are the only means of delimiting subexpressions. This excludes the use of several kinds of delimiters such as ( ), [ ] and { } for grouping; the use of a dot whose scope extends as far to the right as possible as in "$\forall x.\boldsymbol{P}$"; the use of groups of dots as in [19] or [48]; etc. To cover these other styles, the theory would have to be considerably generalized. However, it is expected that a rewriting system of the kind described in section 3.4 could still be used for translation of shorthands after syntactic analysis has taken place.

4. Watson does not handle repetitive notations, such as

$$\{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\}. \tag{4.1}$$

Theoretically, such notations present no difficulty. For example (4.1) could be considered as a shorthand for

$$\mathrm{enum}_{\iota\alpha_1\ldots\alpha_n}\ \boldsymbol{A}_1\ \ldots\ \boldsymbol{A}_n \tag{4.2}$$

with $\alpha_1 = \ldots = \alpha_n = \iota$. Each constant $\mathrm{enum}_{\iota\alpha_1\ldots\alpha_n}$ would then be defined by an axiom

$$\mathrm{enum}_{\iota\alpha_1\ldots\alpha_n} = \lambda x_1 \ldots \lambda x_n \mu s \forall y (y \in s \equiv y = x_1 \vee \ldots \vee y = x_n) \tag{4.3}$$

where s, y, x_1 ... x_n are pairwise distinct variables of type ι. A practical difficulty, though, is to find a simple way for the user to specify the notation schema (4.1) $\rightarrow$ (4.2) and the axiom schema (4.3).

5. Watson only accepts one-level linear patterns as left-hand sides of rewrite rules for elimination of shorthands. This means that it does

not handle notations having internal structure, or notations with multiple occurrences of a parameter. The restriction to one-level patterns could be lifted easily as explained at the end of section 3.5.1, but doing so would complicate the specification of bracketing conventions. The linearity restriction does not seem to be of practical importance.

4.1.2 The components of the language processor

When Watson reads a mathematical expression $\boldsymbol{E}$ it first parses it with respect to the surface language; then it rewrites the resulting surface language expression into an expression of the typed λ-language. When writing a mathematical expression, Watson follows the opposite steps: it takes the internal representation of the expression, which is a formula of the typed λ-language, and rewrites it using the reverse rewriting system. This in effect introduces as many surface notations as possible. Then it takes the resulting labeled expression and removes the phrase markers and a maximal set of parentheses which can be removed without rendering the expression ambiguous. So at run-time the language processor of Watson has three components: a parser, a pretty-printer, and a rewriter. In addition there is a compiler, which takes as input a set of notation specifications, and a set of bracketing specifications (declarations of associativity, mutual associativity and precedence, and additional bracketing rules), and produces the data that drive the parser, pretty-printer and rewriter.

Compiler

Each notation specification is a rewrite rule. The left-hand side of the rule is a labeled expression. The right-hand side, on the other hand, is an ordinary expression of the typed λ-language, without phrase markers, and which does not have to be fully parenthesized. This is possible because the syntax of the typed λ-language is built-in, and the compiler can invoke the parser on the right-hand side, with that restricted syntax.

The compiler computes data for use by the parser, the pretty printer, and the rewriter. The data for the parser and the pretty-printer consist essentially of a *definite clause grammar* [45, 46] derived from the set of left-hand sides (which are linear one-level patterns) as explained in section 3.5.1. In addition, there is an *operator table* telling which sub-expressions of the form "$\boldsymbol{A}\, s_2\, \boldsymbol{A}'$"

cannot immediately follow s_1 without intervening parentheses, s_1 and s_2 being mathematical symbols (such as + or ×) or keywords. The operator table is computed automatically, without the user having to declare symbols as operators explicitly.

The user has some control over the spacing produced by the pretty-printer: a compulsory blank can be specified by leaving two or more blanks at the desired position in the left-hand side of the rewrite rule.

Once the compiler has been run on a particular rewriting system and set of bracketing rules, the data it produces can be written out as a *grammar file* in binary format. Any number of such grammar files, which define as many surface languages, can be saved. It is possible, and it takes practically no time, to load any of these files at any time, even in the middle of a proof, thus switching to a different syntax.

Parser

As mentioned above, the parser uses a definite clause grammar, as proposed in [46]. However, instead of the parsing method described in [46], the parser uses a bottom-up method inspired by [38].[1] The bottom-up method has been enhanced by two improvements which exploit special characteristics of mathematical notations:

1. Different mathematical notations tend to use different sets of mathematical symbols and keywords. A notation cannot be part of a given mathematical expression if not all the symbols and keywords used in the notation appear in the expression. Thus it is possible to rule out many notations by a simple scan of the input string. Typically, only a handful of notations are left, and the parser operates with a very small grammar.

2. Long mathematical expressions tend to be polynomials, by which we simply mean here expressions of the form

$$A_1\, s_1\, A_2\, s_2\, \ldots\, A_n\, s_n\, A_{n+1}$$

[1] An initial implementation based on [46] was too slow; the second implementation based on [38] was several orders of magnitude faster. The first implementation may have been naïve, however.

constructed from notations "$A\ s_i\ A'$". As mentioned above, the compiler produces an operator table. If, say, the notations "$A\ s_i\ A'$" are mutually left-associative, the operator table will tell us that "$A_2 s_2 A_3$", though a well-formed substring, cannot be a well-formed subexpression of the entire expression. The same is true for "$A_i\ s_i\ A_{i+1}$" for every $i > 1$. Thus the combinatorial explosion of well-formed substrings that would arise in the bottom-up parse of a long polynomial is prevented.

Pretty-printer

The main task of the pretty-printer is to determine a minimal set of parentheses which makes the expression to be printed unambiguous. In addition, the pretty-printer deals with spacing and indentation.

In Watson, parenthesization of the output requires no additional information besides the left-hand sides of the rewrite rules, and the bracketing conventions (associativity, precedence, and so on) available to the parser. It is computed bottom up following the parse tree of the expression. For each subtree, heavier and heavier parenthesizations are tried out until one is found unambiguous by parsing it. When parsing a candidate parenthesization, well-formed substrings of the subexpressions are reused; thus, although pretty-printing is slower than parsing, the performance of the algorithm is still adequate to interactive use.

Rewriter

In both directions the rewriter uses algorithm 3.1 which, as explained in section 3.4.2, runs in time linear in the size of the expression being rewritten, for a given set of rewrite rules. As mentioned in section 3.4.4, the rewriting time is negligible compared with the parsing time or the pretty-printing time.

4.2 A methodology for interactive proof

4.2.1 Motivation

It would have been possible to try out the formal system and the theory of mathematical languages that we have described so far by modifying an existing theorem prover. For example, a language processor could have been

added to the HOL theorem prover. However, the new formal system and new linguistic theory invited the design of a new theorem prover according to the same guiding principle, viz. the desire to allow mathematicians and engineers to carry out proofs in a way not much different from what they are accustomed to. So a new methodology for interactive proof as been developed for Watson. The new methodology does not ignore previous work in interactive theorem proving; rather, it borrows and combines features from several different sources, notably HOL and TPS.

The main feature of the new methodology is that it provides a comfortable environment where the user can:

1. Interrupt his/her work, save it in a hierarchically organized library, and retrieve it and resume it later.

2. Use results before proving them without giving up on the guarantee of correctness.

3. Inspect his/her work and, moreover, correct it, using an ordinary text editor.

4. Switch easily from one surface language to another; use results stated or proved using a different surface language; share results with researchers using a different surface language.

This flexibility is provided at two levels: within a given proof, and across proofs, when developing one or more theories.

We now make this more concrete by providing a brief description of the inferential aspects of Watson.

4.2.2 Named theories

A theory, as we have seen, is a set of sentences. A theory can be given a name by a *theory declaration*. In most cases, a declaration associates a name with a theory of the form

$$\Gamma_1 \cup \ldots \cup \Gamma_n \cup \{\boldsymbol{P}_1, \ldots, \boldsymbol{P}_m\},$$

where $\Gamma_1 \ldots \Gamma_n$, the *immediate subtheories*, are previously declared theories, and $\boldsymbol{P}_1 \ldots \boldsymbol{P}_m$, the *immediate axioms*, are sentences. Such a declaration can

be entered by a command given to Watson; but usually it is easier to write it down in the appropriate file of the library, in the following form:

> Defined theory *Name*;
> subtheories *Name*, ..., *Name*;
> axioms *Name*: "*Formula*", ..., *Name*: "*Formula*".

For example, here is the theory devices used in the case study, exactly as it appears in the library:

```
Defined theory devices;
subtheories time, voltages;
axioms
nor_def: "%nor_oiiiii = \dlh\dhl\x\y\z (
    dlh %in %time /\
    0 < dlh /\
    dhl %in %time /\
    0 < dhl /\
    x %in %time -> %voltages /\
    y %in %time -> %voltages /\
    z %in %time -> %voltages /\
    !t0 !t1 (
        t0 %in %time /\ t1 %in %time /\ t0 + dlh < t1 /\
        !t(t0 < t /\ t < t1 => x(t) %low /\ y(t) %low) =>
        ?t2 (t1 < t2 /\ !t (t0 + dlh < t /\ t < t2 => z(t) %high))) /\
    !t0 !t1 (
        t0 %in %time /\ t1 %in %time /\ t0 + dhl < t1 /\
        !t(t0 < t /\ t < t1 => x(t) %high \/ y(t) %high) =>
        ?t2 (t1 < t2 /\ !t (t0 + dhl < t /\ t < t2 => z(t) %low))))",
latch_def: "%latch_oiiiiii = \dlh\dhl\a\b\q\qbar(
    %nor dlh dhl a qbar q /\ %nor dlh dhl q b qbar)",
latch_spec_def: "%LatchSpec_oiiiii = \d\a\b\q\qbar(
    d %in %time /\
    0 < d /\
    a %in %time -> %voltages /\
    b %in %time -> %voltages /\
    q %in %time -> %voltages /\
    qbar %in %time -> %voltages /\
```

```
!t0 !t1 !t2 (
  t0 %in %time /\
  t1 %in %time /\
  t2 %in %time /\
  t0 + d < t1 /\
  t1 < t2 /\
  !t (t0 < t /\ t < t1 => b(t) %high) /\
  !t (t0 < t /\ t < t2 => a(t) %low)
  =>
  ?t3 (
    t2 < t3 /\
    !t (t0 + d < t /\ t < t3 => q(t) %high) /\
    !t (t0 + d < t /\ t < t3 => qbar(t) %low)))
/\
!t0 !t1 !t2 (
  t0 %in %time /\
  t1 %in %time /\
  t2 %in %time /\
  t0 + d < t1 /\
  t1 < t2 /\
  !t (t0 < t /\ t < t1 => a(t) %high) /\
  !t (t0 < t /\ t < t2 => b(t) %low)
  =>
  ?t3 (
    t2 < t3 /\
    !t (t0 + d < t /\ t < t3 => qbar(t) %high) /\
    !t (t0 + d < t /\ t < t3 => q(t) %low))))".
```

It consists of two immediate subtheories, `time` and `voltages`, and three immediate axioms `nor_def`, `latch_def`, and `latch_spec_def`. The example shows some of the notational compromises that have been made:

1. The identifiers in roman font of the formal system become identifiers preceded by %, while identifiers in italic font use no prefix. Thus

%high	stands for	high
%time	stands for	time
%in	stands for	in
x	stands for	x
dlh	stands for	dlh
qbar	stands for	$qbar$

2. Type subscripts are written on the same line, separated by an underscore from the identifier; o is written o, ι is written i. Thus

%nor_oiiiii	stands for	$\mathrm{nor}_{o\iota\iota\iota\iota\iota}$
%latch_oiiiiii	stands for	$\mathrm{latch}_{o\iota\iota\iota\iota\iota\iota}$
%LatchSpec_oiiiii	stands for	$\mathrm{LatchSpec}_{o\iota\iota\iota\iota\iota}$

3. The symbol ! is used for $\forall$, ? for $\exists$, /\ for $\wedge$, and so on. The keyword %in, stands, as we have seen, for the roman identifier "in" which itself is used instead of the unavailable symbol $\in$.

Sometimes this kind of theory declaration cannot be used, and sometimes it could be used but it is not convenient. In particular, it cannot be used for ZF, which has an infinite number of axioms. We still want to tell Watson the name of the theory, zf, and list some of the axioms, such as empty-set, pair-set, and so on. We do this by an *incomplete declaration* of the form

```
Primitive theory Name;
subtheories Name, ..., Name;
axioms Name: "Formula", ..., Name: "Formula";
free variables: ....
```

This tells Watson that the named theory has certain axioms and subtheories, but that there may be others which are not mentioned.[2] Since Watson needs to know exactly what variables are free in the axioms of the theory, any such variables which do not occur free in the mentioned axioms or subtheories must be listed in the optional part `free variables` of the declaration. Here is a possible declaration of the theory zf:

[2]The terminology `Defined theory` vs. `Primitive theory` is borrowed from concept declarations in KL-ONE.

```
Primitive theory zf;
axioms extens: "!x !y (!z (z %in x <=> z %in y) => x = y)",
empty_set: "?s !x ~ x %in s",
pair_set: "!x !y ?s !z (z %in s <=> z %in x \/ z %in y)",
union: "!x ?s !z (z %in s <=> ?y (z %in y /\ y %in x))",
power_set_prim: "!x ?s !y(y %in x <=> !z(z %in y => z %in x))".
```

The instances of the axiom schema of replacement are missing. This is remedied by an inference rule (in the sense of section 4.2.5) which takes as arguments the parameters e, x, y, u, s and $\boldsymbol{P}$ of the axiom schema and produces a theorem of the form:

$$\mathrm{ZF} \vdash \forall z_1 \ldots \forall z_n(\ \forall x \forall y \forall u(\boldsymbol{P} \wedge \boldsymbol{P}^y_u \supset y = u) \supset \\ \forall e \exists s \forall y(y \in s \equiv \exists x(x \in e \wedge \boldsymbol{P}))).$$

Besides the axiom schema of replacement, the axioms of infinity:

$$\exists s(\emptyset \in s \wedge \forall x(x \in s \supset x \cup \{x\} \in s))$$

and foundation:

$$\forall x(\neg(x = \emptyset) \supset \exists y(y \in x \wedge y \cap x = \emptyset))$$

are also missing from the declaration of `zf`. This is because it is not easy to express them without using the notations "$\emptyset$", "$\{\boldsymbol{A}\}$", "$\boldsymbol{A} \cup \boldsymbol{B}$", and "$\boldsymbol{A} \cap \boldsymbol{B}$". Eliminating the notations by hand would be quite tedious. It would be possible to write a program to eliminate notations systematically by the method described in section 2.5.6. But there would be no point to it. Instead, we simply declare the abbreviated axioms to be theorems in the theory `zfa` consisting of `zf` together with axioms defining the representing constants of a number of mathematical notations:

```
Theorem infinity: zfa |-
"?s (%emptyset %in s /\ !x (x %in s => x %union {x} %in s))".

Theorem foundation: zfa |-
"!x (~ x = %emptyset => ?y (y %in x /\ y %inters x = %emptyset))".
```

Of course no proofs are provided for these theorems.

4.2.3 The library

The library is a tree of Unix directories which contains the following kinds of data:

1. Declarations of theories.

2. Declarations of theorems and lemmas. Each declaration states a given result and associates a name with it; optionally, there is a pointer to a proof of the result, consisting of the name of a proof file, together with the name of a line which proves the result. The difference between the two kinds of results is that lemmas are not visible to Watson outside the directory where they are declared.

3. Proofs.

The hierarchical arrangement of the data in the library has the purpose of facilitating its organization and retrieval. It does not impose any usage restriction, except the above-mentioned one for lemmas. A theory or theorem declared in a given directory may be used in any other directory.

Data within a directory is organized in files, as follows:

1. The text file `thrsrsts.ext` contains declarations of theories, theorems and lemmas, and pointers to proofs, in human-readable and editable format, using some particular surface language.

2. The binary file `thrsthms.bin` contains declarations of theories and thoerems, but as internal data structures immediately usable by the theorem prover. All mathematical expressions involved in the declarations are formulas of the typed λ-language; that is, they make use of no mathematical notations.

3. The binary file `lmmsprfs.bin` contains declarations of lemmas and pointers to proofs, in binary format, again with no use of mathematical notations.

4. Any number of files *ProofName*`.psel` contain proofs, either completed or in progress, in human-readable and editable format (`psel` stands for "proof state in external language").

5. Any number of files *ProofName*.psil contain proofs in binary format (psel stands for "proof state in internal language").

Thus all declarations and proofs can be stored in text files and/or in binary files, either file format being optional. By default, Watson makes use of the binary files, which can be read in and written out almost instantaneously. On demand, however, Watson can produce a text file using whatever surface syntax is currently active. And it can read in a text file and produce the corresponding binary file.

The ability to edit files using an ordinary text editor has turned out to be very useful in practice. For example:

1. The proof of correctness of the latch makes use of about twenty simple theorems of the theory time. The proof was started before any of those theorems had been proved, or even stated. As the need for a theorem arose, the theorem was added to the text file thrsrsts.ext in the time directory. This was done using an ordinary text editor in a different window. Then the corrected text file was read in by a single command to Watson, without interrupting the proof, and it was possible to use the theorem immediately. Once the proof of correctness was completed, all the theorems of time were proved easily.

2. During development of the proof of correctness errors were often made. For example, at some point the subformula q %low was written instead of q(t) %low. The error propagated itself over many lines of the proof before it was discovered. But, even though no log of the commands was being kept, it was not necessary to redo any work. The proof was written out in text format; all occurrences of q %low were changed to q(t) %low with a single query-replace command of the text-editor; the text file was read in (and verified); and the proof was resumed at the point were the error had been discovered.

Because it is easy to correct mistakes, the user can concentrate on the overall argument of the proof, rather than on getting every detail right.

The duality of text and binary files means that every result can be stated and proved in whatever surface syntax is more convenient. It can then be used (via the binary format) when a different syntax is in effect. This also realizes the above-mentioned feature that results and even proofs can be easily shared among researchers using different notations.

4.2.4 Proofs

In section 2.3.5 we defined the notion of proof in the formal system. We now use the word in a different sense, to denote a data structure in Watson which materializes the state of development of a proof. The relationship between the two senses of the word will be made more precise in section 4.2.9. A Watson proof can exist in three different forms: as a text file, as a binary file, or as an in-core data structure which is being modified by Watson in response to user commands.

A Watson *proof* consists of *lines*, justifications of the lines, and a few other pieces of information which we shall describe later. Each line has at most one justification. Each justification consists of an inference rule (in the sense of section 4.2.5), with optional parameters, and a list of *premises*. Each premise is either a line of the proof or a named theorem or lemma; we say that a line having a justification *follows* from the the premises of its justification. The proof can be viewed as a labeled ordered graph, with the lines (and any named results used in the proof) as nodes, the justifications as labels, and the premises of the justification of a line as children of that line. The existence of a data structure that embodies a proof under construction is a departure from HOL, and more generally from the LCF tradiction. In this respect Watson is closer to TPS. A proof graph has also been advocated in [41].

Each line of a proof is a named sequent, i.e. it consists of a name, a set of hypotheses, and a conclusion. The hypotheses being sentences, the set of hypotheses is a theory. It is not necessarily a *named* theory, however: it would be awkward to have to declare and name each of the temporary theories which result from assumption introdution and discharge in a natural deduction proof. Instead, the set of hypotheses is described by a theory name together with a list of lines:

$$Theory, Line_1 \ldots Line_n \quad \text{|- "}Conclusion\text{"}.$$

The intended theory is the union of the theory named *Theory* and the set of conclusions of the lines named $Line_1 \ldots Line_n$. We shall refer to *Theory* as *the theory of the line*, and to $Line_1 \ldots Line_n$ as *the assumptions* of the line. The expectation is that the theory will change rarely, or not at all, in the course of a proof, while the assumptions will change often.

A proof has *proved lines* and *unproved lines*. The set of proved lines is

defined as the smallest set S such that S contains every line which follows only from theorems, lemmas, and lines which are elements of S. Under normal circumstances, if a line has been proved in this sense, then its conclusion indeed follows from its hypotheses in the formal system; this is made more precise in section 4.2.9. Among the unproved lines some do not have a justification at all; these are called *goals*. Usually, when the proof is finished, all the lines have been proved and one of them is the theorem or lemma which was to be proved; such a line cannot have assumptions. The fact that a line proves a named theorem or lemma is recorded as part of the proof; we have seen that this is also recorded with the result in the file `thrsrsts.ext`.

The fact that a line has been proved is implicit in the proof graph; but proved lines are also explicitly marked as such. The marking has two purposes:

1. It makes it easy to tell the user which lines have been proved. When a goal receives a justification which turns it into a proved line, any line of which it is a premise may become proved; this in turn may cause other lines to become proved, and so on. The marking is propagated in the graph, and, as lines are marked, their names are displayed on the screen.

2. The order in which the lines are marked defines the *direct proof* which corresponds most closely, in some sense, to the way in which the user has constructed the proof. When writing out the proof as a text file the user has two options: either to list the lines in the order in which they have been created, or to list them in the order in which they have been marked as proved.

4.2.5 Inference toolkit

In section 2.3.5 we defined an *inference rule* as a relation between sequents. Now we are also going to use the phrase to refer to a Watson procedure related to a primitive or derived inference rule of the formal system. There are three kinds of such procedures:

1. A *forward* rule takes as arguments the premises (existing lines, theorems or lemmas), a name for the conclusion, and perhaps some parameters, computes the conclusion (a sequent which follows from the

premises by the corresponding inference rule of the formal system), and adds it to the proof graph, under the given name, together with a justification.

2. A *backward* rule takes as arguments the conclusion (an existing goal), perhaps some of the premises and some parameters, and names for the remaining premises; it computes the remaining premises, it adds them to the graph uder the given names, and it attaches a justification to the conclusion. Since the conclusion acquires a justification it ceases to be a goal, while the new lines, which have no justifications, become new goals.

3. A *verification* rule takes as arguments the names of the premises and conclusion, and perhaps some parameters; it adds no new lines to the graph; it verifies that the conclusion does follow from the premises by the corresponding inference rule of the formal system, and it adds to the graph a justification of the conclusion.

The inference toolkit of the first version of Watson is minimal, although sufficient to do non-trivial proofs with relative comfort. It has the following components:

1. *Primitive inference rules.* For each of the primtive inference rules of the formal system there is at least a verification version, and usually either a forward version, a backward version, or sometimes both. Most of the rules have been made more comfortable by generalizing them slightly (then they correspond to derived rules of the formal system). For example `forall_elim`, which implements ∀-Elimination, can specialize several variables at once, and `and_intro`, which implements ∧-Introduction, can assemble any number of conjuncts. The substitution rule is generalized as:

$$\frac{\Gamma \vdash \mathcal{C}[\boldsymbol{A}] \qquad \Gamma' \vdash \forall \boldsymbol{x}_1 \ldots \forall \boldsymbol{x}_n (\boldsymbol{A} = \boldsymbol{B})}{\Gamma \cup \Gamma' \vdash \mathcal{C}[\boldsymbol{B}]} \qquad \begin{array}{l} \mathcal{C} \text{ does not capture any} \\ \text{variable free in } \Gamma' \\ \textit{and} \text{ in } \forall \boldsymbol{x}_1 \ldots \forall \boldsymbol{x}_n (\boldsymbol{A} = \\ \boldsymbol{B}) \end{array}$$

which is easily shown to be a derived rule of inference.

2. *Rule of replacement.* This is the rule which makes up for the absence of the axiom schema of replacement in the named theory `zf`. In fact it does more, since it accepts as parameters of the schema Σ-sentences rather than just $\mathcal{V}$-F.O. sentences, based on theorem 2.8.

   More precisely, the rule comes in a verification version and a forward version. The verification version takes as argument an existing goal $\Gamma \vdash \boldsymbol{Q}$. It verifies that the theory of the goal has `zf` as a subtheory (not necessarily an *immediate* subtheory) and it checks whether its conclusion is of the form:

$$\forall z_1 \ldots \forall z_n (\ \forall x \forall y \forall u (\boldsymbol{P} \wedge \boldsymbol{P}^y_u \supset y = u) \supset \\ \forall e \exists s \forall y (y \in s \equiv \exists x (x \in e \wedge \boldsymbol{P})))$$

   where $z_1, \ldots, z_n$, e, x, y, u and s are as in the description of the axiom schema (section 2.5.4, page 53), and where $\boldsymbol{P}$ is a Σ-sentence for some set-theoretic abbreviation system Σ which is included in the theory of the sequent. In practical terms, the rule verifies that every constant occurring in $\boldsymbol{P}$ which is not a F.O. logical constant, $\mathrm{in}_{o\iota\iota}$, $\mathrm{the}_{\iota(o\iota)}$, or a constant of type ι, is ultimately defined in terms of such constants by a set of definitions which are axioms of the theory of the sequent.

   The forward version takes as arguments the name for the line to be created and the parameters e, x, y, u, s and $\boldsymbol{P}$ of the schema, puts together the instance of the schema, constructs a line with the instance of the schema as conclusion, `zf` as theory, and no assumptions, and adds the line to the graph, with a justification having no premises.

3. A few convenience rules, such as: reflexivity and symmetry of equality; a rule of separation, which is the equivalent of the rule of replacement for the axiom schema of separation; a rule of abbreviation expansion, which takes as a parameter a constant, finds a definition for it in the theory of the sequent, replaces the constant with its definition, and converts to β-nf; and a few others.

4. The *Horn rule*, described below.

5. The four classical *methods of proof*, described below.

4.2.6 The Horn rule

The Horn rule comes in a forward version and a verification version. There is no backward version yet, although one would be very useful. We describe the forward version. It takes as arguments the premises, which must be of the form:

$$\Gamma_0 \vdash \forall x_1 \ldots \forall x_n (A_1 \wedge \ldots \wedge A_m \supset C)$$
$$\Gamma_1 \vdash B_1$$
$$\ldots$$
$$\Gamma_m \vdash B_m$$

It tries to find $U_1 \ldots U_n$ such that each B_i is the result of simultaneously substituting $U_1 \ldots U_n$ for $x_1 \ldots x_n$ in A_i without variable capture. If it succeeds, it forms the sentence $C^{x_1 \ldots x_n}_{U_1 \ldots U_n}$. If here again there is no variable capture, the rule succeeds and adds the sequent

$$\Gamma_0 \cup \ldots \cup \Gamma_n \vdash C^{x_1 \ldots x_n}_{U_1 \ldots U_n}$$

to the proof graph with its justification.[3]

The Horn rule does not shorten proofs dramatically: it replaces a call to `forall_elim`, a call to `and_intro` and a call to `impl_elim` (which implements $\supset$-Elimination). However it is convenient because it saves the trouble of specifying the formulas to be substituted for the variables in the universal prefix. Also, it makes proofs clearer; the Horn rule is the natural way of making use of theorems of the form:

$$\forall x_1 \ldots \forall x_n (A_1 \wedge \ldots \wedge A_m \supset C). \tag{4.4}$$

Since such theorems are very common, the rule is used very often.

But the rule is also important in that it shows how techniques developed for automatic theorem proving in F.O.L. could be adapted for use in set theory. In F.O.L, if $A_1, \ldots, A_m, B_1, \ldots, B_m, C$ were atomic formulas, (4.4) would of course be a Horn clause, and the rule would be the equivalent of performing m resolution steps. But here, even when $x_1 \ldots x_n$ are variables of type ι, (4.4) is not a Horn clause, since $A_1 \ldots A_m$ and C are formulas

[3]To compute the union of the hypotheses of a set of lines, Watson takes the union of the sets of assumptions, and the greatest of the theories of the lines. If there is not a greatest theory, the rule fails. As mentioned before, it is expected that all the theories will most often be the same in the course of a proof.

of H.O.L. Generally the sentence (4.4) is not even a substitution instance of a Horn clause, since the variables $x_1 \ldots x_n$ can occur, in the body of an abstraction within a A_i or within C. This means that Andrews' "Rule Q" [5, §5236, page 177] is not sufficent to derive the conclusion of the Horn rule from the premises. But we see that we can proceed as in F.O.L. with the simple addition of a check for variable capture.

It should be observed that the necessity of adapting F.O. techniques is not an unpleasant byproduct of our decision to develop set theory within H.O.L. The problem is inherent to the use of mathematical notations which bind variables. Since the problem does not occur in axiomatic set theory without such notations, this shows again, from a different angle, that mathematical notations do add something practically substantial to axiomatic set theory, which cannot easily be dismissed as "syntactic sugar."

4.2.7 The methods of proof

There are four classical methods of proof, which Bourbaki calls: the method of the auxiliary hypothesis, the method of proof by disjunction of cases, the method of the auxiliary constant, and the method of proof by contradiction. They correspond closely to the rules of the formal system where assumptions are discharged. However, there is slightly more to them than those rules, so in Watson they are provided as separate elements of the toolkit.

Remark. From the example given in [4] it appears that the rules P-INDIRECT and P-CHOOSE of TPS are identical to the method of proof by contradiction and the method of the auxiliary constant in Watson, respectively.

Remark. Each of the methods is invoked as a command with no arguments. (Except that the method of the auxiliary constant takes the constant as an optional argument. The "constant" is in fact a variable, as in Bourbaki.) They act upon the *current line* and the *current goal.* The current line and goal are maintained by Watson automatically, but the choices made by the system can be overriden by the user. The maintenance of the current goal is in some sense a generalization of the mechanism provided by the "subgoal package" of Cambridge LCF.

We describe now each of the four methods:

1. *Method of the auxiliary hypothesis*—`aux_hypo`. It can be invoked when

the conclusion of the current goal G is of the form "$\boldsymbol{P} \supset \boldsymbol{Q}$", and it uses a rule `impl_intro` which implements primitive rule 14 ($\supset$-Introduction). The procedure `aux_hypo` which implements the method takes the following two steps:

(a) It *assumes* $\boldsymbol{P}$, i.e. it creates and proves a line L with conclusion $\boldsymbol{P}$ and with L itself as the only assumption, using an inference rule procedure `assume` which corresponds to primitive rule 1 (Reflexivity of $\vdash$).[4]

(b) It calls the backward version of `impl_intro` on G, with the name of the newly created line L as a parameter. This results in the creation of a new goal G' with conclusion $\boldsymbol{Q}$, with the same theory as G, and with the same assumptions as G plus L; and also in the justification of G by `impl_intro` with G' as premise.

After the call to `aux_hypo` the current goal becomes G'. The user will try to prove this new goal, which means proving $\boldsymbol{Q}$ with the additional hypothesis $\boldsymbol{P}$. If the user succeeds, the marking of G' as proved will trigger the automatic marking of G as proved. Thus when the user proves $\boldsymbol{Q}$ under the hypothesis $\boldsymbol{P}$, he or she will be notified that "$\boldsymbol{P} \supset \boldsymbol{Q}$" as been proved as well.

2. *Disjunction of cases*—`by_cases`. It can be invoked when the current line L has a conclusion of the form $\boldsymbol{P}_1 \vee \ldots \vee \boldsymbol{P}_n$, and it uses a rule `or_elim` which implements a generalization of rule 13 ($\vee$-Elimination) to multiple disjuncts. It takes the following steps:

(a) It assumes $\boldsymbol{P}_1 \ldots \boldsymbol{P}_n$, which results in the creation of n lines $L_1 \ldots L_n$.

(b) It calls the backward version of `or_elim` on the current goal G, with the current line L as disjunctive premise, and the names of the lines $L_1 \ldots L_n$ as parameters. The rule creates n case premises

[4] Every line must have a theory, besides its list of assumptions. The theory given to L by `aux_hypo` is the one of the goal line G. (It might be better to use the empty theory `empty`.) The theory is passed to `assume` as a parameter. Thus the procedure `assume` combines the primitive rules 1 (Reflexivity of $\vdash$) and 2 (Monotonicity of $\vdash$).

$M_1 \ldots M_n$. Each M_i is obtained from G by adding L_i as an assumption, and becomes a new goal. The old goal G is justified by `or_elim` with $L, M_1 \ldots M_n$ as premises.

After invoking `by-cases` the user will try to prove each of the new goals M_i. This means proving the conclusion $\boldsymbol{Q}$ of G n times, each time with a $\boldsymbol{P}_i$ as additional hypothesis. When the last of theses subsidiary goals is proved, the goal G will automatically be marked as proved, and the user will be notified that $\boldsymbol{Q}$ is now proved, without additional hypotheses.

3. *Method of the auxiliary constant*—`aux_const`. It can be invoked when the conclusion of the current line L is of the form "$\exists x \boldsymbol{P}$". It uses the backward version of `exists-elim`, which implements rule 21 ($\exists$-Elimination). It takes a variable y as an optional parameter. If y is given, it checks that it is of same type as x and adequate to $\boldsymbol{P}^x$. If the y parameter is not present, x is used instead. Then `aux_const` takes the following steps:

   (a) It assumes $\boldsymbol{P}^x_y$, which results in the creation of a new proved line L'.

   (b) It calls `exists_elim`, passing it as a parameter the name of the new line L'. This results in the creation of a new goal G', which is the same as the current goal G, but with G' as an additional assumption. A justification of G from L and G' by rule `exists-elim` is added to the proof graph.

   After invoking `aux_const` the user's task is to prove G', i.e. to prove the conclusion $\boldsymbol{Q}$ of G with the additional hypothesis $\boldsymbol{P}^x_y$. Once this is done, the user will be notified that G has been proved as well, i.e. that $\boldsymbol{Q}$ has been proved without the additional hypothesis.

4. *Method of proof by contradiction*—`by_contra`. It relies on two inference rules: `not-intro`, which implements primitive rule 8 ($\neg$-Introduction), and `contra`, which implements primitive rule 16 (Contradiction). The procedure `by_contra` assumes the contrary $\boldsymbol{P}$ of the conclusion $\boldsymbol{Q}$ of the current goal G. (If $\boldsymbol{Q}$ is of the form $\neg\boldsymbol{Q}'$ then $\boldsymbol{P}$ is $\boldsymbol{Q}'$; if the conclusion of G is not a negation, then $\boldsymbol{P}$ is $\neg\boldsymbol{Q}$.) This results in

a newly proved line L having $\boldsymbol{P}$ as conclusion and only assumption. Then it calls the backward version of the appropriate inference rule: `not_intro` if $\boldsymbol{Q}$ is a negation, `contra` otherwise. This results in the creation of a new goal G' with conclusion $\perp$ and the hypotheses of G plus the additional hypothesis L, and in the justification of G with premise G'.

After invoking `by_contra` the task of the user is to prove G', i.e. to prove $\perp$ with the contrary $\boldsymbol{P}$ as additional hypothesis. When this has been achieved G will be marked as proved automatically, and the user will be notified that $\boldsymbol{Q}$ has been proved.

4.2.8 A simple example

To acquire a feel for the way proofs are done in Watson, with the limited inference toolkit available in this initial version, let us examine in detail the proof of a simple theorem, theorem `time8` in the theory `time`:

$$\forall x \forall y (x \in \text{time} \wedge y \in \text{time} \supset x < y \vee y \leq x)$$

We prove this knowing that the ordering of time is total:

$$\forall x \forall y (x \in \text{time} \wedge y \in \text{time} \supset x \leq y \vee y \leq x) \tag{4.5}$$

and $x < y$ being defined from $x \leq y$ as follows:

$$\forall x \forall y (x < y \equiv x \leq y \wedge \neg\, x = y). \tag{4.6}$$

So the proof is very simple, by cases using (4.5). If $y \leq x$ obviously $x < y \vee y \leq x$. If $x \leq y$, then we consider again two cases: $x = y$, $\neg x = y$. In the first case $y \leq x$ by reflexivity of $\leq$, in the second case $x < y$ using (4.6).

Let us now follow the details of the proof in Watson. Here is the statement of the theorem in the file `thrsrsts.ext`:

```
Theorem time8: time |-
    "!x !y (x %in %time /\ y %in %time => x < y \/ y =< x)".
```

We shall see in section 4.3 what the representations of the notations `%time`, `<` and `=<` are in the formal system, but we don't need to know this now. We begin by setting up the goal with a command `mk_goal`:

```
GOAL: Line g1: time |-
"!x !y(x %in %time /\ y %in %time => x<y \/ y=<x)".
```

Then we call the backward version of the rule `forall-intro`, which implements a generalization of rule 18 (∀-Introduction) that introduces a universal prefix with multiple variables. We obtain a new goal:

```
GOAL: Line g2: time |- "x %in %time /\ y %in %time => x<y \/ y=<x".
```

and line g1 acquires a justification, with line g2 as premise. We can ask Watson to print the current state of the proof:

```
Line g1: time |- "!x !y(x %in %time /\ y %in %time => x<y \/ y=<x)"
would follow from g2.
Rule: forall_intro.

GOAL: Line g2: time |- "x %in %time /\ y %in %time => x<y \/ y=<x".

The current goal is line g2.
```

To prove line g2 we prove

$$x < y \;\backslash/\; y =< x$$

under the assumption

$$x \;\%in\; \%time \;/\backslash\; y \;\%in\; \%time.$$

That is, we use the method of the auxiliary hypothesis, by calling `aux_hypo`. This creates and proves line `l1`:

```
Line l1: time, l1 |- "x %in %time /\ y %in %time"
follows by rule: assume.
```

sets up the new goal g3:

```
GOAL: Line g3: time, l1 |- "x<y \/ y=<x".
```

and justifies line g2 by line g3 and the rule `impl_intro`:

```
Line g2: time |- "x %in %time /\ y %in %time => x<y \/ y=<x"
would follow from g3.
Rule: impl_intro.
```

Now we work forward from line 11 towards satisfying goal g3. We start by splitting the conjunction, by a call to the forward version of and_elim, which implements primitive rule 11 (∧-Elimination):

```
Line 12: time, 11 |- "x %in %time"
follows from 11.
Rule: and_elim.

Line 13: time, 11 |- "y %in %time"
follows from 11.
Rule: and_elim.
```

Then we use of the axiom of theory time which asserts that =< is a total ordering:

```
total: "!x!y (x %in %time /\ y %in %time => x =< y \/ y =< x)",
```

We apply the axiom by a call to horn, which creates and proves line 14:

```
Line 14: time, 11 |- "x=<y \/ y=<x"
follows from theorem total, 12, 13.
Rule: horn.
```

In this case horn does not simplify the proof much, since the formulas to be substituted for the variables x and y are just the variables themselves; but horn still makes the proof clearer.

From line 14, as anticipated, we reason by cases. This is done by a call to by_cases, which makes the two assumptions:

```
Line 15: time, 15 |- "y=<x"
follows by rule: assume.

Line 16: time, 16 |- "x=<y"
follows by rule: assume.
```

sets up two goals:

```
GOAL: Line g4: time, 16, l1 |- "x<y \/ y=<x".

GOAL: Line g5: time, 15, l1 |- "x<y \/ y=<x".
```

and justifies the old goal g3 by the new goals and or_elim:

```
Line g3: time, l1 |- "x<y \/ y=<x"
would follow from 14, g4, g5.
Rule: or_elim.
```

Line g5 follows immediately from line 15 by $\vee$-Introduction. So we call the *verification* version of the rule or_intro which implements $\vee$-Introduction. No new lines are created, but a justification for g5 is added to the proof graph:

```
Line g5: time, 15, l1 |- "x<y \/ y=<x"
follows from 15.
Rule: or_intro.
```

Now we have to prove goal g4 from line 16. We want to distinguish two cases, $x = y$ and $\neg\, x = y$, so we need the theorem:

$$\vdash x = y \vee \neg\, x = y.$$

This is an instance of a tautology; however, using a tautology rule would clash with the style of proof by natural deduction. In fact, there is no tautology rule in Watson (although clearly one should be implemented at some point). The appropriate thing to do in a natural deduction system is to use the well-known proof of the principle of the excluded middle. This proof has been done in Watson, giving the theorem

```
Theorem excluded_middle: empty |- "!p_o (p_o \/ ~ p_o)".
```

Notice that this theorem could not be expressed if we were working within F.O.L. rather than H.O.L. We would then only have a *theorem schema*:

$$\vdash \boldsymbol{P} \vee \neg \boldsymbol{P}$$

with a schematic proof. In the object language, we would have a different proof for every instance of $\boldsymbol{P}$. This is an example of how the formal system of Watson makes it possible to bring schematic arguments into the object language, so that they can be carried out using the theorem prover.

So we make use of `excluded_middle`. We apply to it `forall_elim`, specializing p_o to $x = y$:

```
Line 17: empty |- "x = y \/ ~ x = y"
follows from theorem excluded_middle.
Rule: forall_elim.
```

Now we call `by_cases`, which results in making the two assumptions:

```
Line 18: empty, 18 |- "~ x = y"
follows by rule: assume.

Line 19: empty, 19 |- "x = y"
follows by rule: assume.
```

setting up the two goals:

```
GOAL: Line g7: time, 19, 16, 11 |- "x<y \/ y=<x".

GOAL: Line g8: time, 18, 16, 11 |- "x<y \/ y=<x".
```

and justifying the old goal g4:

```
Line g4: time, 16, 11 |- "x<y \/ y=<x"
would follow from 17, g7, g8.
Rule: or_elim.
```

We prove g7 first, from 19. Axiom `refl` of theory `time` asserts that $\leq$ is reflexive:

```
refl: "!x (x %in %time => x =< x)",
```

We use it by calling `horn` on the axiom and line 12:

```
Line l10: time, l1 |- "x=<x"
follows from theorem refl, l2.
Rule: horn.
```

Then by substitutivity of equality from l10 and l9:

```
Line l11: time, l1, l9 |- "y=<x"
follows from l10, l9.
Rule: subst_mp.
```

We close the gap by calling the verification version of or_intro on l11 and g7. Goal g7 is now proved:

```
Line g7: time, l9, l6, l1 |- "x<y \/ y=<x"
follows from l11.
Rule: or_intro.
```

We turn then to goal g8, which we prove from line l8. We form the conjunction of the two case assumptions, lines l6 and l8:

```
Line l12: time, l6, l8 |- "x=<y /\ ^ x = y"
follows from l6, l8.
Rule: and_intro.
```

This means that x is strictly less than y. To get this we use the axiom tlt_i_def of theory time:

```
tlt_i_def: "!x !y (x < y <=> x =< y /\ ^x=y)",
```

by specializing it into:

```
Line l13: time |- "x<y <=> x=<y /\ ^ x = y"
follows from theorem tlt_i_def.
Rule: forall_elim.
```

Rather inelegantly, we have to explicitly swap the two sides of the equivalence:

```
Line l14: time |- "x=<y /\ ^ x = y <=> x<y"
follows from l13.
Rule: sym_eq.
```

Now by substitutivity of equality (equivalence, in this case):

```
Line l15: time, l6, l8 |- "x<y"
follows from l12, l14.
Rule: subst_mp.
```

The last step is to close the gap between line l15 and goal g8 by a call to the verification version of or_intro. A justification is entered for g8, which becomes proved. Then Watson tells us that g4 is proved as well, and that so are g3, g2 and g1. Watson then notices that g1 is a root of the proof graph (i.e. there are no unproved lines having a justification which has g1 has a premise) and says so. Finally, Watson notices that there are no goals left, i.e. that everything has been proved, and lets us know. Here is the resulting proof, with the lines printed in the order in which they have been entered. A direct proof, with the lines printed in the order in which they have been proved, can also be produced, but it is less readable.

```
Line g1: time |- "!x !y(x %in %time /\ y %in %time => x<y \/ y=<x)"
follows from g2.
Rule: forall_intro.

Line g2: time |- "x %in %time /\ y %in %time => x<y \/ y=<x"
follows from g3.
Rule: impl_intro.

Line l1: time, l1 |- "x %in %time /\ y %in %time"
follows by rule: assume.

Line g3: time, l1 |- "x<y \/ y=<x"
follows from l4, g4, g5.
Rule: or_elim.

Line l2: time, l1 |- "x %in %time"
follows from l1.
Rule: and_elim.

Line l3: time, l1 |- "y %in %time"
follows from l1.
```

```
Rule: and_elim.

Line l4: time, l1 |- "x=<y \/ y=<x"
follows from theorem total, l2, l3.
Rule: horn.

Line l5: time, l5 |- "y=<x"
follows by rule: assume.

Line l6: time, l6 |- "x=<y"
follows by rule: assume.

Line g4: time, l6, l1 |- "x<y \/ y=<x"
follows from l7, g7, g8.
Rule: or_elim.

Line g5: time, l5, l1 |- "x<y \/ y=<x"
follows from l5.
Rule: or_intro.

Line l7: empty |- "x = y \/ ^ x = y"
follows from theorem excluded_middle.
Rule: forall_elim.

Line l8: empty, l8 |- "^ x = y"
follows by rule: assume.

Line l9: empty, l9 |- "x = y"
follows by rule: assume.

Line g7: time, l9, l6, l1 |- "x<y \/ y=<x"
follows from l11.
Rule: or_intro.

Line g8: time, l8, l6, l1 |- "x<y \/ y=<x"
follows from l15.
Rule: or_intro.
```

```
Line l10: time, l1 |- "x=<x"
follows from theorem refl, l2.
Rule: horn.

Line l11: time, l1, l9 |- "y=<x"
follows from l10, l9.
Rule: subst_mp.

Line l12: time, l6, l8 |- "x=<y /\ ^ x = y"
follows from l6, l8.
Rule: and_intro.

Line l13: time |- "x<y <=> x=<y /\ ^ x = y"
follows from theorem tlt_i_def.
Rule: forall_elim.

Line l14: time |- "x=<y /\ ^ x = y <=> x<y"
follows from l13.
Rule: sym_eq.

Line l15: time, l6, l8 |- "x<y"
follows from l12, l14.
Rule: subst_mp.
```

4.2.9 The guarantee of correctness

Difficulties

The most important service provided by a PDS is that it guarantees the correctness of the proof. But this guarantee is relative to the correctness of the prover. In order for the guarantee to be absolute, the prover itself would have to be proved correct. This, however, is difficult to do, given the state of the art in program verification. To date, no PDS has been formally verified. And, since a PDS is a complex program, it is almost certain that any unverifed PDS has bugs in it.

In Watson there is an additional obstacle to guaranteeing the correctness

of the proof. We have seen that the user is allowed extreme flexibility in the way he or she conducts a proof and develops a theory. Results can be used before they are proved, and proofs and theories can be freely edited with an ordinary text editor. So the user can easily introduce circularities and other errors in his proofs and theories.

In spite of these difficulties, Watson does provide some guarantee of correctness. The approach to doing so is inspired in part by the one followed in HOL, which itself goes back to LCF.

HOL/LCF: an architectural guarantee of correctness

In HOL as in LCF the prover consists of a collection of routines written in and accessible from a typed functional language called ML. In ML there is a special type `thm`. Only the routines which implement the forward versions of the primitive inference rules produce data structures of type `thm`. Forward versions of derived inference rules produce theorems only by calling primitive rules. Backward versions of rules are called *tactics*. Given a goal, a tactic produces a set of subgoals, together with a *validation routine*. When the subgoals are proved, i.e. when data structures representing sequents which coincide with the subgoals but have type `thm` are obtained, the validation routine can be called on the proved subgoals to prove the goal, i.e. to compute a data structure representing a sequent which coincides with the goal but has type `thm`.

This scheme can be thought of as providing an *architectural* guarantee of correctness: by restricting the pieces of code which are allowd to create theorems, only those pieces of code have to be assumed correct. Thus the LCF/HOL guarantee is a relative one, which can be stated as follows: if the forward versions of the primitive rules of inference are implemented correctly, and the typing mechanism of ML functions correctly, and the user does not circumvent the typing mechanism (say, by using the routine `mk_thm` which can make anything into a `thm`), then anything with type `thm` does indeed represent a theorem of the formal system. It should be noted that this approach has two drawbacks: it relies on a particular kind of programming language, and it produces theorems rather than proofs.[5]

[5]That is, proofs are not materialized by any data structures. So the only way of recording a proof is by logging the commands which are used to construct it. And the only way of communicating the proof of a theorem is to provide an ML program which

Watson: three levels of guarantee

Watson provides three distinct guarantees, based on three differents sets of hypotheses. The first guarantee can be stated as follows:

1. if the rules of inference and graph handling mechanisms are implemented correctly, and
2. the user does not manipulate the graph except through the rules of inference provided by Watson, and
3. the theorems and lemmas used in the proof are correct,

then whenever a line is marked as being proved its conclusion does indeed follow from its hypotheses in the formal system.

This is an architectural guarantee of correctness, like the HOL/LCF guarantee. Buggy programs can be used to constructs proofs; as long as these programs interact with the proof graph via the designated entry points, i.e. the proof rules provided by Watson, the guarantee of correctness is preserved. The difference with HOL/LCF is that the trusted code includes not only the implementation of the primitive rules, but the implementation of derived rules as well; and not only the forward versions of rules, but all versions of each rule. An essential aspect of this guarantee is that it rules out circularities in the proof.

The second guarantee relies on the *single proof verifier*. This is a program which verifies the justifications of the lines of a proof, and the marking of lines as proved. Each justification is in fact the name of a validation routine to be called by the verifier to check that the conclusion does follow from the premises. The verifier calls the validation routine of each line, passing it as parameters the premises and conclusion.[6] If no errors are found, it then

computes the theorem. Understanding the proof by reading the program can be next to impossible. However, Avra Cohn is now working on an enhancement to the tactic mechanism which will produce proof trees in addition to the validation functions.

[6]In some cases, parameters are kept together with the name of the validation routine, as part of the justification of a line. The verifier then passes those parameters as arguments to the validation routine, in addition to the premises and the conclusion of the inference step. Observe that the validation routine is not very different from the verification version of the rule. In fact, to save some code, the validation routine serves a dual purpose in Watson. Besides being used by the verifier, it is also used during the interactive construction of the

checks that the marking is correct according to the state of the graph. The guarantee can be stated as follows:

1. if the validation routines are correctly implemented, and
2. the single proof verifier is correct, and
3. the theorems used in the proof are correct,

then if the verifier finds no errors and a line is marked as proved its conclusion does follow from its hypotheses in the formal system.

This second guarantee is not architectural: it does not rely on the correctness of the code which constructs the proof, but rather on the correctness of a separate program (the verifier and validation routines) which goes over the proof after it has been constructed. Thus this guarantee is preserved even if the user writes programs that access the proof graph directly. Also, this guarantee does away with reliance on the correctness of the backward versions of the rules, thus playing a role similar to the role played by the tactic mechanism in HOL/LCF. But it goes further, since it does not rely on the forward versions either: it relies only on the verification versions. In HOL/LCF it is natural to view the forward versions of rules as the primary ones, because the programming language ML is functional, and thus an inference rule is most naturally embodied in a function which computes the conclusion from the premises. However, independently of any particular programming environment, the verification versions of the rules seem simpler and in some sense more primitive than the forward versions.

The third guarantee relies on a *recursive proof verifier*. The recursive verifier verifies the proof on which it is invoked, plus the proofs of any theorems and lemmas used in the proof which do have proofs stored in the library, plus the proofs of any results used in those proofs, and so on. When done, it lists all the results which have been used and for which no proofs have been found in the library. The recursive verifier also checks the declarations of all the named theories which come up in the proofs (in particular, it verifies that there are no circularities in the declarations, and that the lists of free variables of each theory are correct). This third guarantee can be stated as follows:

---

proof: the verification version of the rule calls the validation routine, and if it gets a green light, it adds the justification record to the proof graph.

1. if the validation routines are correct, and

2. if the recursive proof verifier is correct, and

3. if all the results accepted without proof hold in the formal system

then if the verifier finds no errors, any line marked as proved is a result which holds in the formal system. In particular this guarantee protects against circularities *across proofs* which are possible because the user can use a result before proving it. When the recursive verifier finds no results without proof, Watson solemnly declares that the proof has been done *from first principles.*

All three guarantees are practically useful. The first guarantee allows the user to conduct a proof interactively as he or she wishes, forward or backward, entering and using lines before proving them, without the danger of forgetting to prove a subgoal or doing a circular "proof." The second guarantee is most useful when proofs are edited by hand. When the edited text file is read in, the single proof verifier is called on it and guarantees among other things that the justifications, which may have been entered by hand, do hold. The third guarantee is useful at the end of a long proof which may have required many lemmas and may thus have spawned many subsidiary proofs. The user will then call the resursive proof verifier to make sure, among other things, that he or she has not forgotten to prove any lemmas, and that there are no circularities among the lemmas.

4.3 A case study in hardware verification

4.3.1 Choice of the case study

As a case study we have chosen the proof of correctness of a latch implemented as two cross-coupled nor-gates. In the field of hardware verification this proof is important because the latch is one of the most elementary devices which can retain data, and the basic building block of flip-flops. Thus the latch is the bridge between asynchronous and clocked devices. From the point of view of theorem proving, the proof of correctness of the latch is non-trivial while not being very long, and it illustrates some interesting aspects of the formal system, such as the use of the axiom schema of replacement (via its consequence, the theorem schema of separation).

The proof is also interesting because it seems to have resisted previous efforts of researchers in the field of hardware verification.

John Herbert has proved the correctess of a Master-Slave flip-flop and an Edge-Triggered D flip-flops using HOL [28]. The latch is a basic component of these flip-flops. The correctness of the latch has been proved, and the the proof is reported in detail. But a discrete model of time has been used; moreover, the fact that the latch retains data for as long as the inputs show no new data is proved by induction, which relies precisely on the discreteness of time. In Watson, on the other hand, time has been axiomatized as a continuum, and the proof that the latch retains data uses the characteristic property of a continuous ordering, namely that every non-empty set bounded below has a greatest lower bound.

Hanna and Daeche report in [25] that the proof of correctness of an Edge-Triggered D flip-flop has been carried out using Veritas. The correctness of the latch is mentioned as a lemma, but there is no indication of how the proof has been done, even though it is the most mathematically interesting part of the overall proof, the rest of which is described in detail. The authors do not even say if they have used a discrete or continuous axiomatization of time, which, as we have seen, is an essential distinction regarding the proof of correctness of the latch. Since the authors do not actually say that the lemma has been proved, it may be that it has been accepted without proof.

4.3.2 Axiomatization of continuous time

Time is axiomatized in classical mechanics as an affine space over an oriented one-dimensional vector space over the field of the real numbers. That is, the set of *durations* is an oriented one-dimensional vector space over the reals (the orientation distinguishes positive durations from negative ones), and the set of *instants* is an affine space over that vector space. Recall that U is an affine space over a vector space V when it is equipped with a function which maps every pair of points A, B of U to a vector $\overline{AB}$ and which satisfies the following two conditions, where A, B and C range over U, and x over V:

$$\forall A \forall B \forall C (\overline{AB} + \overline{BC} = \overline{AC})$$
$$\forall A \forall x \exists! B (\overline{AB} = x)$$

The ordering of the reals induces orderings on the durations and on the instants. All these isomorphic orderings are total; *divisible*, or *dense*, i.e. such

that there is a point in between any two points; and moreover *continuous*, or having the *greatest lower bound property*, i.e. such that every non-empty set bounded below has a greatest lower bound.

For the purpose of this case study, however, we shall simplify this structure, without loss of generality.[7] We identify durations and instants, and we structure time as an additive group with a total, dense, continuous ordering (thus omitting scalar multiplication). Let us review the constants, notations and axioms that we use in Watson (all the axioms are in the theory `time` unless otherwise noted):

1. The constant time_ι denotes the set of instants/durations. We use the keyword time as a shorthand for time_ι:

$$\text{time} \longrightarrow \text{time}_\iota$$

2. *Commutative group structure.* The constant tplus_ι denotes the group operation, i.e. addition of durations:

$$\text{tplus}_\iota \in \text{time} \times \text{time} \mapsto \text{time}$$

Furthermore, to represent the notation "$\boldsymbol{A} + \boldsymbol{B}$" we need a constant $\text{tplus}_{\iota\iota\iota}$:

$$(\text{FML}_\iota\ A_{\text{FML}_\iota} + B_{\text{FML}_\iota}) \longrightarrow \text{tplus}_{\iota\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$$

which is defined as

$$\text{tplus}_{\iota\iota\iota} = \text{apply}_{\iota\iota\iota\iota}\ \text{tplus}_\iota$$

The constant $\text{apply}_{\iota\iota\iota\iota}$ is defined in theory `zfa`; it takes a set theoretic operation to a H.O.L. operation:

$$\text{apply}_{\iota\iota\iota\iota} = \lambda f \lambda x \lambda y (f(x, y))$$

In this axiom, "$f(x, y)$" involves two notations: the ordered pair notation "$(\boldsymbol{A}, \boldsymbol{B})$" and the set theoretic function application "$\boldsymbol{A}(\boldsymbol{B})$".

[7] By "without loss of generality" we mean that the simplified structure can be defined in terms of the complete structure, and that the statement of correctness of the latch for the complete structure follows immediately from the statement of correctness for the simplified structure.

The latter uses another constant in the "apply" family; it stands for "$\text{apply}_{\iota\iota\iota}$ $\boldsymbol{A}$ $\boldsymbol{B}$", where $\text{apply}_{\iota\iota\iota}$, also defined in `zfa`, takes set theoretic functions to H.O.L. functions:

$$\text{apply}_{\iota\iota\iota} = \lambda f \lambda x \mu y((x, y) \in f)$$

The constant tzero_ι denotes the neutral element of addition; the symbol 0 is used as a shorthand for tzero_ι:

$$0 \longrightarrow \text{tzero}_\iota$$

The constant tminus_ι denotes the unary minus operation, i.e. the function which maps group elements to their inverses in the group. To represent the notation "$-\boldsymbol{A}$" we use a constant $\text{tminus}_{\iota\iota}$:

$$(\text{FML}_\iota - A_{\text{FML}_\iota}) \longrightarrow \text{tminus}_{\iota\iota}\ A_{\text{FML}_\iota}$$

The function symbol $\text{tminus}_{\iota\iota}$ denotes the H.O.L. function corresponding to the set theoretic function denoted by tminus_ι:

$$\text{tminus}_{\iota\iota} = \text{apply}_{\iota\iota\iota}\ \text{tminus}_\iota$$

With these constants we express the axioms of the commutative group structure as follows:

$\forall x \forall y(x \in \text{time} \wedge y \in \text{time} \supset x + y = y + x)$
$\forall x \forall y \forall z(x \in \text{time} \wedge y \in \text{time} \wedge z \in \text{time} \supset (x + y) + z = x + (y + z))$
$\forall x(x \in \text{time} \supset x + 0 = x)$
$\forall x(x \in \text{time} \supset x + (-x) = 0)$

3. *Ordering.* We are going to use $\leq$ and $<$. We use the constants tle_ι and tlt_ι to denote the corresponding set theoretic relations on time_ι. But we also need two binary predicate symbols to represent the notations; we use $\text{tle}_{o\iota\iota}$ and $\text{tlt}_{o\iota\iota}$:

$$(\text{FML}_\iota\ A_{\text{FML}_\iota} \leq B_{\text{FML}_\iota}) \longrightarrow \text{tle}_{o\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$$
$$(\text{FML}_\iota\ A_{\text{FML}_\iota} < B_{\text{FML}_\iota}) \longrightarrow \text{tlt}_{o\iota\iota}\ A_{\text{FML}_\iota}\ B_{\text{FML}_\iota}$$

and we define them with the axioms:

$$\begin{aligned} \mathrm{tle}_{o\iota\iota} &= \mathrm{apply}_{o\iota\iota\iota}\, \mathrm{tle}_{\iota} \\ \mathrm{tlt}_{o\iota\iota} &= \mathrm{apply}_{o\iota\iota\iota}\, \mathrm{tlt}_{\iota} \end{aligned}$$

where $\mathrm{apply}_{o\iota\iota\iota}$ is yet another constant of the "apply" family, which this time maps set theoretic binary relations to H.O.L. binary relations:

$$\mathrm{apply}_{o\iota\iota\iota} = \lambda r \lambda x \lambda y((x, y) \in r)$$

Using these notations we express the relationship between the two binary relations with the axiom:

$$\forall x \forall y(x < y \equiv x \leq y \wedge \neg\, x = y)$$

Then we state the axioms of a total, dense, continous ordering:

$$\begin{aligned} &\forall x(x \in \mathrm{time} \supset x \leq x) \\ &\forall x \forall y \forall (x \leq y \wedge y \leq z \supset x \leq z) \\ &\forall x \forall y(x \leq y \wedge y \leq x \supset x = y) \\ &\forall x \forall y(x \in \mathrm{time} \wedge y \in \mathrm{time} \supset x \leq y \vee y \leq x) \\ &\forall x \forall y(x < y \supset \exists z(x < z \wedge z < y) \\ &\forall s(\ s \subseteq \mathrm{time} \wedge \neg s = \emptyset \wedge \exists y \forall x(x \in s \supset y \leq x) \supset \\ &\quad \exists glb(\forall x(x \in s \supset glb \leq x) \wedge \forall y(\forall x(x \in s \supset y \leq x) \supset y \leq glb))) \end{aligned}$$

4. Finally we express the *compatibility of the ordering with the group structure*:

$$\forall x \forall y \forall z(x \in \mathrm{time} \wedge y \in \mathrm{time} \wedge z \in \mathrm{time} \wedge x \leq y \supset x + z \leq y + z)$$

4.3.3 Modelling of devices

A gate is sometimes modelled as having a certain delay d and producing at time $t + d$ the output corresponding to the inputs at time t. This is an unrealistic assumption for two reasons:

1. A short enough change in one of the inputs may not be propagated to the output.

2. The delay d given in the specification of a gate is an upper bound; there is no guarantee that the output will not change sooner. Consequently, there is no guarantee that a given value will linger at the output for the full delay d after the inputs have changed.

A more realistic assumption, and one which is sufficient to do the proof, is the following: if certain values are continuously present at the inputs for a time interval $]t_0, t_1[$ of duration greater than d, then the corresponding value will be present at the output over $]t_0 + d, t_2[$, for some $t_2 > t_1$. Notice that not positive lower bound is specified for the duration $t_2 - t_1$ during which the output keeps its value after the input values have changed; it can be arbitrarily small.

In certain technologies one of the transitions from low to high or high to low takes significanly longer than the other; so we use two distinct delays: *dlh* from low to high, and *dhl* from high to low.

We model the signals as functions from time into the set of *voltages*. We do not need any structure on the set of voltages, other than two voltage ranges:

$$\begin{aligned} \text{highvrange}_\iota &\subseteq \text{voltages}_\iota \\ \text{lowvrange}_\iota &\subseteq \text{voltages}_\iota \end{aligned}$$

To say that a voltage is high or low we use the postfix notations "$\boldsymbol{A}$ high" and "$\boldsymbol{A}$ low", which we represent with the constants $\text{high}_{o\iota}$ and $\text{low}_{o\iota}$:

$$\begin{aligned} (\text{FML}_o\ A_{\text{FML}_\iota}\ \text{high}) &\longrightarrow \text{high}_{o\iota}\ A_{\text{FML}_\iota} \\ (\text{FML}_o\ A_{\text{FML}_\iota}\ \text{low}) &\longrightarrow \text{low}_{o\iota}\ A_{\text{FML}_\iota} \end{aligned}$$

The two representing constants are defined as the unary predicates denoting membership in the voltage ranges, by the axioms:

$$\begin{aligned} \text{high}_{o\iota} &= \text{apply}_{o\iota\iota}\ \text{highvrange}_\iota \\ \text{low}_{o\iota} &= \text{apply}_{o\iota\iota}\ \text{lowvrange}_\iota \end{aligned}$$

where $\text{apply}_{o\iota\iota}$ is the constant of the "apply" family which maps a set to a unary predicate:

$$\text{apply}_{o\iota\iota} = \lambda r \lambda x (x \in r)$$

We use the notation

$$\text{nor}\ \boldsymbol{D\ E\ A\ B\ C}$$

to express the fact that C is the output signal of a nor gate with low-to-high delay D and high-to-low delay E when the input signals are A and B. The notation is represented in the formal language by the predicate nor_{oiiiii}:

$$(\text{FML}_o \text{ nor } D_{\text{FML}_i}\ E_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ C_{\text{FML}_i}) \longrightarrow \\ \text{nor}_{oiiiii}\ D_{\text{FML}_i}\ E_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ C_{\text{FML}_i}$$

Here is the definition of the predicate, as it appears in the declaration of the theory devices (the notation %time -> %voltages denotes the set of set-theoretic functions from %time to %voltages):

```
nor_def: "%nor_oiiiii = \dlh\dhl\x\y\z (
    dlh %in %time /\
    0 < dlh /\
    dhl %in %time /\
    0 < dhl /\
    x %in %time -> %voltages /\
    y %in %time -> %voltages /\
    z %in %time -> %voltages /\
    !t0 !t1 (
        t0 %in %time /\ t1 %in %time /\ t0 + dlh < t1 /\
        !t(t0 < t /\ t < t1 => x(t) %low /\ y(t) %low) =>
        ?t2 (t1 < t2 /\ !t (t0 + dlh < t /\ t < t2 => z(t) %high))) /\
    !t0 !t1 (
        t0 %in %time /\ t1 %in %time /\ t0 + dhl < t1 /\
        !t(t0 < t /\ t < t1 => x(t) %high \/ y(t) %high) =>
        ?t2 (t1 < t2 /\ !t (t0 + dhl < t /\ t < t2 => z(t) %low))))"
```

The latch is implemented as two cross-coupled nor-gates (figure 4.1). The circuit has two parameters, the delays D and E of the nor-gates. We use the notation

$$\text{latch } D\ E\ A\ B\ Q\ R$$

to express the fact that Q and R are the output signals of the latch, implemented with nor-gates of low-to-high delay D and high-to-low delay E, when the input signals are A and B. The notation is represented in the formal language by the constant $\text{latch}_{oiiiiii}$ (the implementation predicate of the latch):

$$(\text{FML}_o \text{ latch } D_{\text{FML}_i}\ E_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ Q_{\text{FML}_i}\ R_{\text{FML}_i}) \longrightarrow \\ \text{latch}_{oiiiiii}\ D_{\text{FML}_i}\ E_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ Q_{\text{FML}_i}\ R_{\text{FML}_i}$$

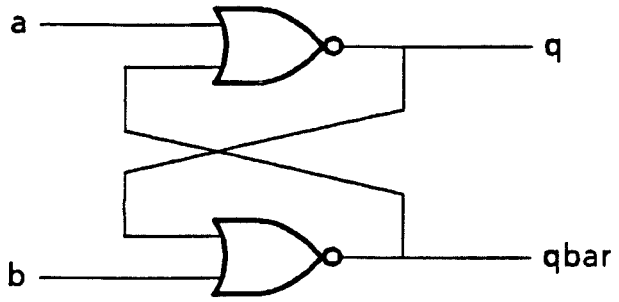

Figure 4.1: Implementation of the latch

And here is the definition of the implementation predicate, as it appears in the theory devices.

```
latch_def: "%latch_oiiiiii = \dlh\dhl\a\b\q\qbar(
    %nor dlh dhl a qbar q /\ %nor dlh dhl q b qbar)".
```

Finally we need a specification for the latch. The inputs a and b of the latch encode one bit of data as follows: if b is high and a low we say that a 1 is present at the inputs; if a is high and b low a 0 is present at the inputs; if both a and b are low then we say that no data is present at the inputs; the case where both a and b are high is meaningless. The outputs encode a 1 when q is high and *qbar* is low, and a zero when q is low and *qbar* is high. The desired behaviour is for the outputs to reflect the input data, with some delay d, when data are present at the imputs, and to remember the data when it goes away from the inputs, for as long as both inputs stay low.

More precisely, if b is high over $]t_0, t_1[$, with $t_1 - t_0 > d$ and a is low over the same interval, and stays low beyond t_1 until t_2, then q should be high and *qbar* should be low over $]t_0 + d, t_3[$, for some $t_3 > t_2$. (Proving that the outputs keep their values *beyond* t_2 should enable the verification of circuits which use the latch as a component.) Similarly, if a is high over $]t_0, t_1[$, $t_1 > t_0 + d$, and b is low over $]t_0, t_2[$, $t_2 > t_1$, *qbar* should be high and q low over $]t_0 + d, t_3[$, for some $t_3 > t_2$. Here is the definition of the specification predicate of the latch, as it appears in the declaration of the theory devices.

```
latch_spec_def: "%LatchSpec_oiiiii = \d\a\b\q\qbar(
    d %in %time /\
    0 < d /\
    a %in %time -> %voltages /\
    b %in %time -> %voltages /\
    q %in %time -> %voltages /\
    qbar %in %time -> %voltages /\
    !t0 !t1 !t2 (
      t0 %in %time /\
      t1 %in %time /\
      t2 %in %time /\
      t0 + d < t1 /\
      t1 < t2 /\
      !t (t0 < t /\ t < t1 => b(t) %high) /\
```

```
    !t (t0 < t /\ t < t2 => a(t) %low)
    =>
    ?t3 (
      t2 < t3 /\
      !t (t0 + d < t /\ t < t3 => q(t) %high) /\
      !t (t0 + d < t /\ t < t3 => qbar(t) %low)))
  /\
  !t0 !t1 !t2 (
    t0 %in %time /\
    t1 %in %time /\
    t2 %in %time /\
    t0 + d < t1 /\
    t1 < t2 /\
    !t (t0 < t /\ t < t1 => a(t) %high) /\
    !t (t0 < t /\ t < t2 => b(t) %low)
    =>
    ?t3 (
      t2 < t3 /\
      !t (t0 + d < t /\ t < t3 => qbar(t) %high) /\
      !t (t0 + d < t /\ t < t3 => q(t) %low))))".
```

As for the other device predicates, we shall use a notation

$$(\text{FML}_o\ \text{LatchSpec}\ D_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ Q_{\text{FML}_i}\ R_{\text{FML}_i}) \longrightarrow \text{LatchSpec}_{oiiiii}\ D_{\text{FML}_i}\ A_{\text{FML}_i}\ B_{\text{FML}_i}\ Q_{\text{FML}_i}\ R_{\text{FML}_i}$$

We are going to prove that the two cross-coupled nor-gates of figure 4.1 implement a latch whose delay is the sum of the low-to-high and high-to-low delays of the gates. Here is the statement of the correctness theorem, as it appears in the `thrsrsts.ext` file of the directory `lib/time/hv`:

```
Theorem latch_correct: devices |- "!dlh !dhl !a !b !q !qbar (
    %latch dlh dhl a b q qbar =>
    %LatchSpec (dlh + dhl) a b q qbar)".
```

4.3.4 Proof of correctness of the latch

Summary of the proof

Here is an informal explanation of the behaviour of the latch, which one would expect to find in an electrical engineering textbook:

> Assume that b is high and a is low. After a delay dhl, $qbar$ is low. Now both a and $qbar$ are low, so after a further delay dlh q is high. Now that q is high it does not matter if b goes low, $qbar$ will stay low. So as long as a stays low, q will stay high.

This may be convincing, but it is not a proof. The proof which has been carried out in Watson can be summarized in English as follows.

PROOF OF CORRECTNESS OF THE LATCH. Assume first that b is high over $]t_0, t_1[$, $t_1 > t_0 + dhl + dlh$, and a low over $]t_0, t_2[$, $t_2 > t_1$. Then, from the specification of the second nor-gate (the one with inputs q, b and output $qbar$), $qbar$ is low over $]t_0 + dhl, t_1[$ (in fact, over $]t_0 + dhl, t_4[$, for some $t_4 > t_1$). Therefore, from the specification of the first nor-gate (the one with inputs a, $qbar$ and output q), q is high over $]t_0 + dhl + dlh, t_1[$ (in fact, over $]t_0 + dhl + dlh, t_4[$, for some $t_4 > t_1$).

Reasoning by contradiction, assume that it is not the case that, for some $t_3 > t_2$, q is high and $qbar$ low over $]t_0 + dhl + dlh, t_3[$. Let then s be the set of time points $t > t_0 + dhl + dlh$ where it is not the case that q is high and $qbar$ is low. The set s cannot be empty, because then for *any* $t_3 > t_2$ q would be high and $qbar$ low over $]t_0 + dhl + dlh, t_3[$. And the set s is bounded below, by $t_0 + dhl + dlh$. So let t_5 be its greatest lower bound.

We have seen that q is high over $]t_0 + dhl + dlh, t_1[$, while $qbar$ is low over the same interval. So there are no elements of s in $]t_0 + dhl + dlh, t_1[$. Therefore t_1 is a lower bound of s, and $t_1 \leq t_5$. Also, t_5 cannot be strictly greater than t_2, because then q would be high and $qbar$ low over $]t_0 + dhl + dlh, t_3[$, with $t_3 = t_5$, contradicting our hypothesis. So $t_1 \leq t_5 \leq t_2$.

Since there are no elements of s in $]t_0 + dhl + dlh, t_5[$, q is high and $qbar$ is low over the interval. In addition, $qbar$ is low over $]t_0 + dhl, t_1[$. Hence, since $t_0 + dhl < t_0 + dhl + dlh < t_1 \leq t_5$, $qbar$ is low over $]t_0 + dhl, t_5[$.

Since q is high over $]t_0 + dhl + dlh, t_5[$ and b is high over $]t_0, t_1[$, for every $t \in]t_0, t_5[$ at least one of them is high at time t. Since $qbar$ is low over $]t_0 + dhl, t_5[$ while a is low over $]t_0, t_2[$, both of them are low over $]t_0 + dhl, t_5[$.

So, since at least one of the inputs of the second nor-gate is high over $]t_0, t_5[$, the output *qbar* is low over $]t_0 + dhl, t_6[$, for some $t_6 > t_5$. And since both inputs of the first nor-gate are low over $]t_0 + dhl, t_5[$, the output q is high over $]t_0 + dhl + dlh, t_7[$, for some $t_7 > t_5$.

Let t_8 be the earliest of t_6 and t_7. Then q is high and *qbar* is low over $]t_0 + dhl + dlh, t_8[$. So there are no elements of s over that interval, and t_8 is a lower bound of s. But $t_8 > t_5$, a contradiction.

We still have to prove that, if a is high over $]t_0, t_1[$, $t_1 > t_0 + dhl + dlh$, and b low over $]t_0, t_2[$, $t_2 > t_1$, then *qbar* is high and q low over $]t_0 + dhl + dlh, t_3[$, for some $t_3 > t_2$. But this follows from the previous case if we observe that a nor-gate is symmetric, and therefore the latch is symmetric too. □

The proof in Watson has about 350 lines. The symmetry of the nor-gate and the symmetry of the latch have actually been proved as separate theorems:

```
Theorem nor_sym: devices |-
"!dlh !dhl !x !y !z(%nor dlh dhl x y z => %nor dlh dhl y x z)".

Proof of theorem nor_sym: line g1 in nor_sym.

Theorem latch_sym: devices |-
"!dlh !dhl !a !b !q !qbar(%latch dlh dhl a b q qbar => %latch dlh
    dhl b a qbar q)".

Proof of theorem latch_sym: line g1 in lat_sym.
```

and they have been used in the proof of correctness of the latch to derive the second case from the first one. Many simple consequences of the axiomatization of time were required during the proof; they have all been proved as separate theorems. However the proof has not been done from first principles; the following theorems have been admitted without proof:

```
Theorem thm2: zfa |-
"!a !b !x !y (x %in a /\ y %in b =>' (x,y) %in a # b)".
```

(where `a # b` denotes the cartesian product of `a` and `b`).

```
Theorem thm3: zfa |-
"!a !b !x !y ((x,y) %in a # b => x %in a /\ y %in b)".

Theorem thm4: zfa |-
"!a !b !f !x (f %in a -> b /\ x %in a => f(x) %in b)".
```

No particular difficulty is expected in proving these theorems with Watson.

Usage of the theorem schema of separation

The proof makes use of the theorem schema of separation. This is implicit in the sentence of the summary of the proof:

> ... Let then s be the set of time points $t > t_0 + dhl + dlh$ where it is not the case that q is high and $qbar$ is low. ...

This sentence corresponds to the following two lines of the proof:

```
Line 187: devices |-
"!dhl !dlh !q !qbar !t0 !e ?s !t(t %in s <=> t %in e /\
     (t0+dhl+dlh<t /\ ^ (q(t) %high /\ qbar(t) %low)))"
follows by rule: sep_inst.

Line 188: devices |-
"?s !t(t %in s <=> t %in %time /\ (t0+dhl+dlh<t /\
     ^ (q(t) %high /\ qbar(t) %low)))"
follows from 187.
Rule: forall_elim.
```

Line 188 is an instance of schema (2.23), page 54, $\boldsymbol{P}$ being:

```
t0+dhl+dlh<t /\ ^ (q(t) %high /\ qbar(t) %low)
```

When represented in the surface language, $\boldsymbol{P}$ is a formula which contains the following constants: $\text{tplus}_{\iota\iota\iota}$, $\text{tlt}_{o\iota\iota}$, and_{ooo}, not_{oo}, $\text{apply}_{\iota\iota\iota}$, $\text{high}_{o\iota}$, $\text{low}_{o\iota}$. Of these, and_{ooo} and not_{oo} are F.O. logical constants. But the presence of the others means that $\boldsymbol{P}$ is not a μ-sentence. So Watson must look for axioms

defining these constants in the theory devices. In this case, the constants are ultimately defined in terms of F.O. logical constants, $\text{in}_{o\iota\iota}$, $\text{the}_{\iota(o\iota)}$, and the following constants of type ι: tplus_ι, tlt_ι, time_ι, highvrange_ι and lowvrange_ι. So by theorem 2.8, $\boldsymbol{P}$ can be used as a parameter. This determination is done by Watson quite fast, using data structures compiled when theories are declared. Notice that this does not require user intervention: in particular the user does not have to flag particular axioms of a theory which are going to be used as definitions of constants representing notations.

Simplification of the proof by H.O. reasoning

In the summary of the proof we have used the time interval notation "$]\boldsymbol{A}, \boldsymbol{B}[$". This notation can be represented internally as

$$\text{OpenIval}_{\iota\iota\iota}\ \boldsymbol{A}\ \boldsymbol{B}$$

with the constant $\text{OpenIval}_{\iota\iota\iota}$ defined by

$$\text{OpenIval}_{\iota\iota\iota} = \lambda a \lambda b \{x \mid a < x \wedge x < b\}.$$

However, although intervals are convenient in the English-language summary of the proof, their use tends to complicate the formal proof, so in fact they have not been used.

Intervals would have been useful in the following two proof fragments: (i) the proof fragment showing that *qbar* is low over $]t_0 + dhl, t_5[$, given that it is low over $]t_0 + dhl, t_1[$ and $]t_0 + dhl + dlh, t_5[$, and (ii) the proof fragment showing that at least one of the inputs to the second nor-gate, $b(t)$ or $q(t)$, is high for every $t \in]t_0, t_5[$, given than b is high over $]t_0, t_1[$ and q is high over $]t_0 + dhl + dlh, t_5[$. With intervals, one can prove as a lemma that, if $a \leq b < c \leq d$ then $]a, d[\subseteq]a, c[\cup]b, d[$ (in fact $]a, d[=]a, c[\cup]b, d[$), and then one can use the lemma twice. Without intervals, each proof fragment can be done easily, but the little argument by cases involved in the proof of the lemma has to be done twice.

But there is an interesting alternative to using intervals by which the common argument can be factored out. This is to prove the following *higher-order theorem:*

```
Theorem time12: time |-
```

```
"!p_oi !a !b !c !d (
    a =< b /\
    b < c /\
    c =< d /\
    !t (a < t /\ t < c => p_oi t) /\
    !t (b < t /\ t < d => p_oi t)
  =>
    !t (a < t /\ t < d => p_oi t))".

Proof of theorem time12: line g1 in time12.
```

and then use it by specializing p_{oi} to the appropriate λ-abstraction,

$$\lambda t(qbar(t)\ \text{low})$$

or

$$\lambda t(b(t)\ \text{high} \vee q(t)\ \text{high}).$$

This is a second example of how schematic reasoning can be formalized in Watson as higher-order reasoning. (The first example was the use of the theorem of the excluded-middle in the proof described in section 4.2.8.)

Chapter 5

Conclusion

5.1 Summary

We have proposed an approach to the mechanization of mathematics with a set theoretic foundation.

The approach is based on a formal system consisting of the axioms of ordinary set theory, but within H.O.L. rather than F.O.L.; more precisely, within a version of H.O.L. with λ-abstraction and with a description operator which maps undefined descriptions to the empty set. We have shown that set theory within H.O.L. is a conservative extension of set theory within F.O.L.; that is, any F.O. set theoretic result proved in the proposed formal system can also be proved, in principle, in set theory within F.O.L.

In such a formal system, mathematical notations can be represented by constants, those which bind variables necessitating higher-order constants. The representing constants can be defined, in the object language, by axioms constituting a set theoretic abbreviation system, i.e. by a set of definitions with a well-founded definition graph, and with a residual vocabulary consisting only of F.O. logical constants, the predicate and function symbols of the version of set theory under consideration, the description operator for individuals, and individual constants.

We have shown that the formal system remains a conservative extension of set theory within F.O.L. when it is augmented with the axioms of a set theoretic abbreviation system and constants defined by the abbreviation system are allowed in instances of the axiom schema of replacement (in theories

such as ZF) or instances of the theorem schema of class existence (in theories which distinguish between sets and classes).

We have shown how a well-behaved rewriting system can be used to translate between the formal language and the surface language, the latter being the result of augmenting the former with the syntax of mathematical notations.

The proof development system Watson has been built to demonstrate this approach to the mechanization of mathematics. Watson features a methodology for interactive proof which reconciles safety with flexiblity of use, by providing a guarantee of correctness at three different levels. Theory declarations, statements of results, and proofs can be saved in a tree structured library, and can be perused and modified using an ordinary text editor. The surface language is customizable: the user can specify his own notations as rewrite rules. It is easy to switch from one surface language to another. Different languages can be used within the same library, and researchers using different surface languages can easily share results and even proofs. Watson has been used to prove the correctness of a latch implemented as two cross-coupled nor-gates, with an axiomatization of time as a continuum.

5.2 Opportunities provided by the formal system

The motivation for the proposed formal system is to accomodate mathematical notations. This requires up to second-order constants with axiomatic definitions, up to third-order equality for the axioms defining the constants, and λ-abstraction; but no more. Higher-order quantification, in particular, is not required. So it seems that our formal system is much richer than required for our stated goal.

There is however no point in restricting the system since we have shown that, as it is, it adds no new results to ordinary set theory within F.O.L. Instead, we should consider the rich inferential apparatus that we are getting "for free" as an exciting new resource to be explored. Indeed, although the formal system does not introduce new results, it does introduce a rich new class of mathematical methods which are not available in ordinary set theory within F.O.L. We conclude by pointing to some possibilities which are thus

opened by the formal system.

5.2.1 Schematic reasoning

A schematic proof is a proof which contains syntactic variables standing for indeterminate sentences, terms or symbols. It represents a family of object language proofs. Schematic proofs can only be done properly when there is a formal distinction between an object language and a metalanguage.

Schematic proofs come up mostly in metamathematics. But schematic reasoning also comes up in ordinary mathematics. For example, when mathematicians talk about "classes" within ZF, as in [32], their arguments can be formalized as schematic proofs, where what they call a class becomes a generic sentence asserting some property of a generic individual. For an extremely elaborate example of schematic reasoning, see Quine [48].

In our formal system some arguments which would require a schematic proof in ZF within F.O.L. can be carried out as ordinary proofs of higher-order theorems. We have seen two examples of this, in sections 4.2.8 and 4.3.4.

5.2.2 Generalized quantifiers

Many generalized quantifiers are definable in the formal system, including cardinality quantifiers, topological quantifiers, and branching quantifiers. Others should be axiomatizable if not definable. We give three examples of definable quantifiers:

1. *More than half, a cardinality quantifier.* We consider the object language sentence:

   More than half of the $\boldsymbol{x}$ in $\boldsymbol{E}$ are such that $\boldsymbol{P}$

   as standing for:

   $$\text{MoreThanHalf}_{o(o\iota)\iota}\, \boldsymbol{E}\ \lambda \boldsymbol{x} \boldsymbol{P}$$

   Then $\text{MoreThanHalf}_{o(o\iota)\iota}$ can be defined as:

   $$\text{MoreThanHalf}_{o(o\iota)\iota} = \lambda e \lambda p_{o\iota}(\ \text{Card}(\{x \in e \mid \neg(p_{o\iota}\, x)\}) < \text{Card}(\{x \in e \mid p_{o\iota}\, x\}))$$

where the cardinality of a set might be defined as the smallest ordinal number which is equinumerous to the set, and then $\boldsymbol{A} < \boldsymbol{B}$ is an alternative notation for $\boldsymbol{A} \in \boldsymbol{B}$.

2. *A topological quantifier.* We consider the object language sentence:

   An open set of $\boldsymbol{x}$ in $\boldsymbol{E}$ are such that $\boldsymbol{P}$

   as standing for:

   $$\text{Open}_{o(o\iota)\iota}\, \boldsymbol{E}\, \lambda \boldsymbol{x} \boldsymbol{P}$$

   The notion of an open set is relative to a particular topology; so let the constant OpenSets_ι denote the set of open sets of the topology. Then the quantifier $\text{Open}_{o(o\iota)\iota}$ is defined as:

   $$\text{Open}_{o(o\iota)\iota} = \lambda e \lambda p_{o\iota}(\{x \in e \mid p_{o\iota}\, x\} \in \text{OpenSets}_\iota).$$

3. *A branching quantifier.* We consider the object language sentence:

   For every x there exists a y and for every u a v (depending only on u) such that $\boldsymbol{P}$

   as standing for:

   $$\text{BQ}_{o(o\iota\iota\iota)}\, \lambda \boldsymbol{x} \lambda \boldsymbol{y} \lambda \boldsymbol{u} \lambda \boldsymbol{v} \boldsymbol{P}$$

   with the branching quantifier $\text{BQ}_{o(o\iota\iota\iota)}$ defined as follows:

   $$\text{BQ}_{o(o\iota\iota\iota)} = \lambda p_{o\iota\iota\iota} \exists f_{\iota\iota} \exists g_{\iota\iota} \forall x_\iota \forall u_\iota (p_{o\iota\iota\iota}\, x_\iota\, (f_{\iota\iota}\, x_\iota)\, u_\iota\, (g_{\iota\iota}\, u_\iota))$$

5.2.3 Foundations of category theory

Category theory seems to be the only branch of mathematics where Zermelo's method for avoiding the paradoxes—by limiting Comprehension—is an obstacle to mathematical practice. This is because category theory considers collections of objects, such as the collection of all sets, all groups, all topological spaces, and so on, which are not co-extensional to sets. On the other hand in Russell's solution to the paradoxes—type theory—Comprehension is not restricted, and the problem does not come up: the collection of all groups of a given type constitutes a category in a higher type.

In ZF/HOL, although those large collections are still not co-extensional to any sets, it is possible to refer to them by formulas of types other than ι, e.g. "$\lambda x.\top$" for the collection of all sets (in a version of ZF without urelements), or "λx(x is a group)" for the collection of all groups. Then, since HOL is also type theory [13], one can develop category theory as one would in type theory. The fact that ZF/HOL is a conservative extension of ZF/FOL means that any first-order set theoretic result provable by means of category theory in this fashion is also provable in ZF/FOL.

This seems to be a valid alternative to the introduction of *universes* [17, 18, 24, 36, 37], which is the traditional way of providing a set theoretic foundation for category theory.

Appendix A

Difficulties with pseudo-binding

Section 1.5 describes the pseudo-binding method, which is probably the most common approach to the explanation of mathematical notations which bind variables and construct terms. In this appendix we illustrate the practical difficulties which would arise if we tried to use the formal language expressions prescribed by the pseudo-binding method as internal representation of the corresponding surface language expressions in a PDS.

Assume that in the course of a proof we have proved the following lines:

$$u = \{x \in y \mid x \cap z = x\} \tag{A.1}$$

$$v = \{x \in y \mid x \subseteq z\} \tag{A.2}$$

and we want to prove "$u = v$" using the theorem

$$\forall x \forall y (x \cap y = x \equiv x \subseteq y) \tag{A.3}$$

In Watson, one can rewrite (A.1) to

$$u = \{x \in y \mid x \subseteq z\} \tag{A.4}$$

in one step, by a generalization of the rule of substitution, using (A.3). Notice that this does not make use of the definition of the notation "$\{\boldsymbol{x} \in \boldsymbol{A} \mid \boldsymbol{P}\}$". It only makes use of the fact that, in Watson's internal representation as well as in the surface language, "$x \cap z = x$" is indeed a subexpression of the

sentence (A.1). Another substitution step then yields "$u = v$" from (A.4) and (A.2). If on the other hand the pseudo-binding method had been followed, then the expression

$$\{x \in y \mid x \cap z = x\} \tag{A.5}$$

would be a shorthand for a term consisting of a function symbol, say "f", applied to the free variables "y" and "z" of (A.5), so line (A.1) would be a shorthand for:

$$u = \mathrm{f}\, y\, z \tag{A.6}$$

The rewrite "$x \cap z = x$" $\longrightarrow$ "$x \subseteq z$" cannot be performed on (A.6), since "$x \cap z = x$" is not a subsexpression of "$u = \mathrm{f}\, y\, z$". We must then use the axiom defining the function symbol f:

$$\forall x(x \in \mathrm{f}\, y\, z \equiv x \in y \wedge x \cap z = z) \tag{A.7}$$

The rewrite can be performed in (A.7):

$$\forall x(x \in \mathrm{f}\, y\, z \equiv x \in y \wedge x \subseteq z) \tag{A.8}$$

Then "u" can be substituted for "$\mathrm{f}\, y\, z$":

$$\forall x(x \in u \equiv x \in y \wedge x \subseteq z) \tag{A.9}$$

The right-hand side of (A.2),

$$\{x \in y \mid x \subseteq z\} \tag{A.10}$$

also stands for a function symbol, say "g", applied to its free variables "y" and "z"; so (A.2) stands for

$$v = \mathrm{g}\, y\, z \tag{A.11}$$

We must use the definition of "g":

$$\forall x(x \in \mathrm{g}\, y\, z \equiv x \in y \wedge x \subseteq z) \tag{A.12}$$

and substitute "v" for "$\mathrm{g}\, y\, z$":

$$\forall x(x \in v \equiv x \in y \wedge x \subseteq z) \tag{A.13}$$

From (A.9) and (A.13), by substitution:

$$\forall x(x \in u \equiv x \in v) \tag{A.14}$$

Finally, from (A.14) and the axiom of extensionality:

$$u = v$$

To summarize, the use of the pseudo-binding method blocks the rewrite "$x \cap z = x$" $\longrightarrow$ "$x \subseteq z$" and forces instead a detour in the proof. The detour includes the use of three axioms: the two axiomatic definitions of "f" and "g", and the set theoretic axiom of extensionality. This is an artificial complication of the proof. Moreover, it would be quite awkward to justify to the user of a PDS why the rewrite is allowed in (A.7) but not in (A.1).

Appendix B

Conversion

In this appendix we prove the strong normalization theorem for $\alpha\beta\gamma$-conversion and we establish the characterization of formulas in $\beta\gamma$-nf as *standard formulas*. For a thorough study of the lambda calculus see [7].

B.1 Preliminaries

B.1.1 α-classes

The relation $\boldsymbol{A} \stackrel{\alpha}{\rightarrow} \boldsymbol{B}$ between formulas $\boldsymbol{A}$ and $\boldsymbol{B}$ is symmetric, so its reflexive-transitive closure "$\boldsymbol{A}$ α-converts to $\boldsymbol{B}$", i.e. "$\boldsymbol{A}$ and $\boldsymbol{B}$ are the same up to renaming of bound variables" is an equivalence relation, whose classes we shall call *α-classes*, or simply *classes* when no confusion is possible. We shall write $\overline{\boldsymbol{A}}$ for the α-class of $\boldsymbol{A}$.

The normalization theorems are more easily proved for α-classes first, because conversion behaves more regularly between α-classes than between formulas.

We say that $(\mathcal{A}, \mathcal{B})$ is a step of β-conversion *between classes* $\mathcal{A}$ and $\mathcal{B}$ iff there exist representatives $\boldsymbol{A}$ of $\mathcal{A}$ and $\boldsymbol{B}$ of $\mathcal{B}$ such that $(\boldsymbol{A}, \boldsymbol{B})$ is a step of β-conversion between the *formulas* $\boldsymbol{A}$ and $\boldsymbol{B}$; γ-conversion and η-conversion between classes are defined identically from γ-conversion and η-conversion between formulas.

A class $\mathcal{A}$ is said to be a *class in β-nf* iff there exists no class $\mathcal{B}$ such that $(\mathcal{A}, \mathcal{B})$ is a step of β-conversion. (Note that a similar definition could

not be used for β-nf between formulas, because no β-conversion step applies to a formula having a single β-redex "$(\lambda \boldsymbol{x}\boldsymbol{U})\ \boldsymbol{V}$" where $\boldsymbol{V}$ is not free for $\boldsymbol{x}$ in $\boldsymbol{U}$.) If $\mathcal{A}$ is in β-nf, then every representative of $\mathcal{A}$ is in β-nf. Conversely, if a representative of $\mathcal{A}$ is in β-nf, the same is true of $\mathcal{A}$ as a class.

A class $\mathcal{A}$ is said to be a *class in γ-nf* iff there exists no class $\mathcal{B}$ such that $(\mathcal{A}, \mathcal{B})$ is a step of γ-conversion. If $\mathcal{A}$ is in γ-nf, then every representative of $\mathcal{A}$ is in γ-nf. Conversely, if a representative of $\mathcal{A}$ is in γ-nf, the same is true of $\mathcal{A}$ as a class.

We say that $(\mathcal{A}, \mathcal{B})$ is a step of $\beta\gamma$-conversion iff it is a step of β-conversion or a step of γ-conversion. We say that a class $\mathcal{A}$ is in $\beta\gamma$-nf iff there is no class $\mathcal{B}$ such that $(\mathcal{A}, \mathcal{B})$ is a step of $\beta\gamma$-conversion, i.e. iff it is both in β-nf and γ-nf.

B.1.2 α-trees

The proofs of the normalization theorems will be considerable simplified by considering tree structures associated with α-classes and derived form the parse trees of their representatives, which we shall call *α-trees*.

The typed λ-language is a context-free language, generated by the grammar consisting of the following production schemas.

$$
\begin{array}{rcl}
\mathrm{FML}_\alpha & \longrightarrow & \mathrm{FML}_{\alpha\beta}\ \mathrm{FML}_\beta \\
\mathrm{FML}_{\alpha\beta} & \longrightarrow & \lambda\ \mathrm{VAR}_\beta\ \mathrm{FML}_\alpha \\
\mathrm{FML}_\alpha & \longrightarrow & \mathbf{Id}_\alpha \\
\mathrm{FML}_\alpha & \longrightarrow & \boldsymbol{Id}_\alpha \\
\mathrm{VAR}_\alpha & \longrightarrow & \boldsymbol{Id}_\alpha
\end{array}
$$

where the non-terminal symbols are the subscripted symbols FML_α and VAR_α, for every type α, and the terminal symbols are the subscripted symbols $\mathbf{Id}_\alpha$, for every roman identifier **Id** and type α, and $\boldsymbol{Id}_\alpha$, for every italic identifier $\boldsymbol{Id}$ and type α.

Any formula of the typed λ-language has a parse tree with respect to this grammar, defined in the usual way as a triple (N, S, L), where N is the set of nodes of the tree, S the function mapping each internal node to the sequence of its children, and L the function mapping each node to its label (leaf nodes are labeled by terminal symbols, internal nodes by non-terminal symbols). (A generalization of the notion of parse trees, applicable to arbitrary "labeled

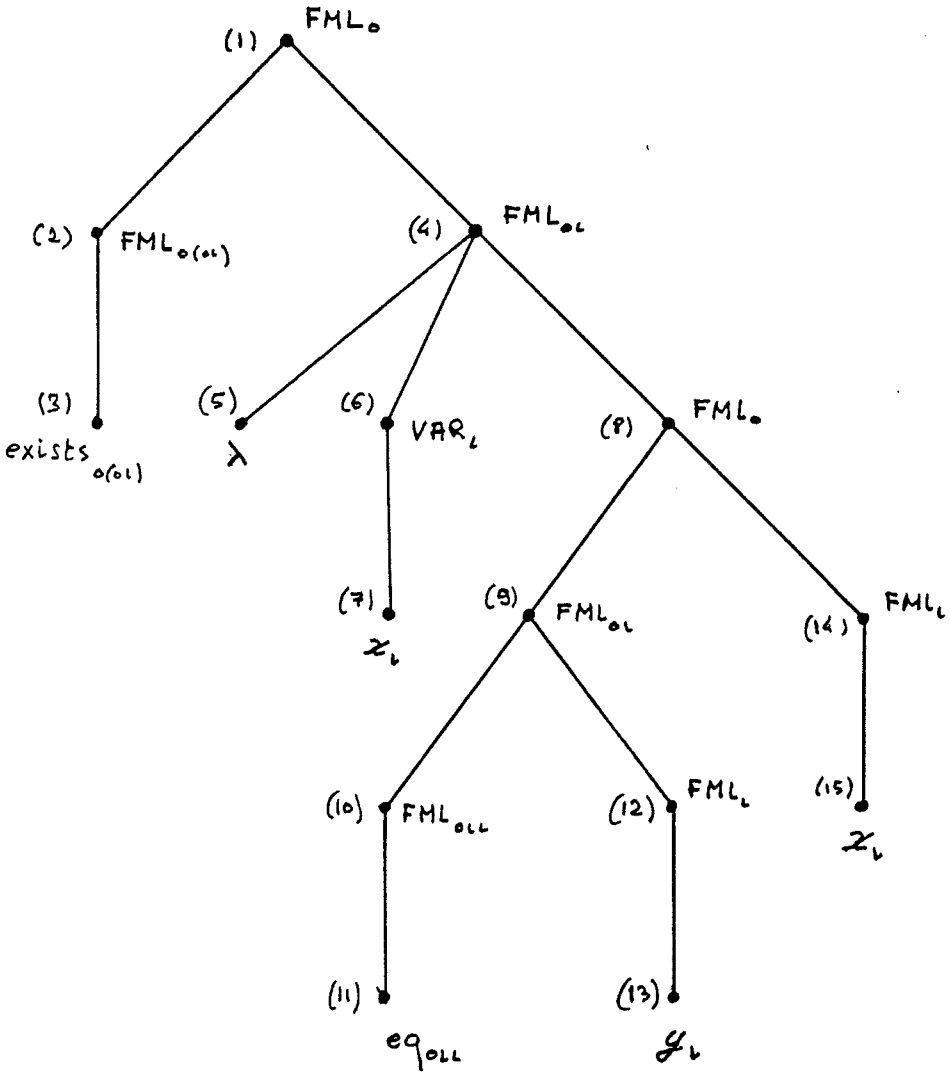

Figure B.1: Parse tree of "exists$_{o(o\iota)}$ λx_ι(equal$_{o\iota\iota}$ y_ι x_ι)"

expressions" will be given in section 3.4.2.) For example, figure B.1 shows a pictorial representation of the parse tree of

$$\text{exists}_{o(o\iota)}\ \lambda x_\iota(\text{equal}_{o\iota\iota}\ y_\iota\ x_\iota)$$

Any set of appopriate cardinality can serve as set of nodes of a parse tree. In figure B.1 we have simply chosen as nodes the consecutive integers $1\ldots15$, assigned in a preorder traversal of the tree. The three components N, S, L are then as follows (square brackets are used to denote sequences);

$$\begin{aligned}
N &= \{1,\ldots,15\}\\
S &= \{(1,[2,4]),(2,[3]),(4,[5,6,8]),(6,[7]),(8,[9,14]),(9,[10,12]),\\
&\quad (10,[11]),(12,[13]),(14,[15])\}\\
L &= \{\ (1,\text{FML}_o),(2,\text{FML}_{o(o\iota)}),(3,\text{exists}_{o(o\iota)}),(4,\text{FML}_{o\iota}),(5,\lambda),\\
&\quad (6,\text{VAR}_\iota),(7,x_\iota),(8,\text{FML}_o),(9,\text{FML}_{o\iota}),(10,\text{FML}_{o\iota\iota}),\\
&\quad (11,\text{equal}_{o\iota\iota}),(12,\text{FML}_\iota),(13,y_\iota),(14,\text{FML}_\iota),(15,x_\iota)\}
\end{aligned}$$

An *α-tree* for an α-class $\mathcal{A}$ is a structure (N', S', L', B) obtained as follows: take a parse tree (N, S, L) of a representative $\boldsymbol{A}$ of $\mathcal{A}$; let $N' = N$ and $S' = S$; let L' be the result of removing from L the node-label pairs corresponding to binding occurrences and bound occurrences of variables; and let B be the set of pairs (p, q) where p is a node corresponding to a bound occurrence of a variable, and q is the node corresponding to the binding occurrence which is referred to by the bound occurrence. For example, to derive an α-tree from the parse tree of figure B.1, we remove the pairs $(7, x_\iota)$ and $(15, x_\iota)$ from L and we let $B = \{(15, 7)\}$; so the four components N', S', L' and B of the α-tree are:

$$\begin{aligned}
N' &= N\\
&= \{1,\ldots,15\}\\
S' &= S\\
&= \{(1,[2,4]),(2,[3]),(4,[5,6,8]),(6,[7]),(8,[9,14]),(9,[10,12]),\\
&\quad (10,[11]),(12,[13]),(14,[15])\}\\
L' &= \{(1,\text{FML}_o),(2,\text{FML}_{o(o\iota)}),(3,\text{exists}_{o(o\iota)}),(4,\text{FML}_{o\iota}),(5,\lambda),\\
&\quad (6,\text{VAR}_\iota),(8,\text{FML}_o),(9,\text{FML}_{o\iota});(10,\text{FML}_{o\iota\iota}),\\
&\quad (11,\text{equal}_{o\iota\iota}),(12,\text{FML}_\iota),(13,y_\iota),(14,\text{FML}_\iota)\}\\
B &= \{(15,7)\}
\end{aligned}$$

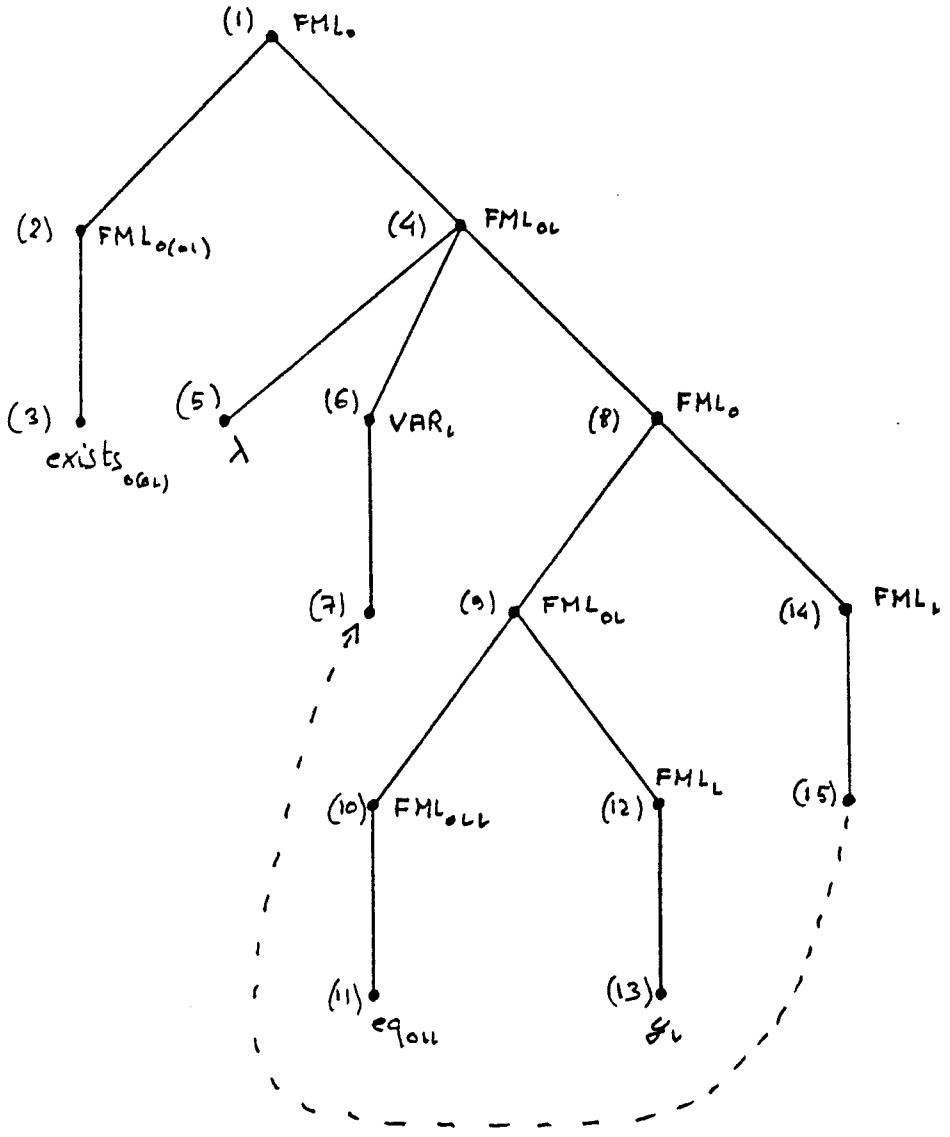

Figure B.2: α-tree of α-class of $\text{exists}_{o(o\iota)}\ \lambda x_\iota(\text{equal}_{o\iota\iota}\ y_\iota\ x_\iota)$

Figure B.2 shows a pictorial representation of the α-tree.

We define an *α-tree* as any structure which can be obtained as the α-tree of some α-class by the above construction. (We leave it to the reader to propose an intrinsic definition.) There is an obvious notion of isomorphism of α-trees, and the construction defines the α-tree of a given α-class up to isomorphism.

Given an α-tree $\mathcal{T} = (N, S, L, B)$ and a node $n \in N$, we define the subtree of $\mathcal{T}$ rooted at n as the structure $\mathcal{T}' = (N', S', L', B')$ where N' is the set of descendents of n, and S', L', B' are the restrictions of the functions S, L, B to N'. In a parse tree, an occurrence of variable in a subtree may of course be bound by a binding occurrence outside of the subtree; this means that a node of $\mathcal{T}'$ may have a dangling binding; that is, B' may have elements (x, y) where $y \in N - N'$. So a subtree of an α-tree is a *quasi-α-tree* rather than an α-tree. We leave it to the reader to come up with an *intrinsic* definition of the notion of quasi-α-tree such that $\mathcal{T}'$ satisfies the definition iff it can be obtained as a subtree of an α-tree $\mathcal{T}$. Notice that an α-tree is a special case of a quasi-α-tree.

The notion of *isomorphism* between quasi-α-trees is straightforward except perhaps for the treatment of dangling bindings. We shall stipulate that an isomorphism must carry a dangling binding to another dangling binding which points to the same outside node. The formal definition is as follows. Let $\mathcal{T} = (N, S, L, B)$ and $\mathcal{T}' = (N', L', S', B')$ be two quasi-α-trees. An isomorphism from $\mathcal{T}$ to $\mathcal{T}'$ is a bijection f from N onto N' such that:

1. S' coincides with the function which, for every n in the domain of S, maps $f(n)$ to the sequence $(f(s_i))_{1 \leq i \leq j}$, where the sequence $(s_i)_{1 \leq i \leq j}$ is the image of n by the funcion S.

2. L' coincides with the function which, for every n in the domain of L, maps $f(n)$ to $L(n)$.

3. B' coincides with the function which, for every n which has an image p by B, maps $f(n)$ to $f(p)$ if $p \in N$, or to p if $p \notin N$.

Isomorphism of α-trees is a special case of isomorphism of quasi-α-trees.

α-Trees are useful because β-conversion and γ-conversion for α-classes can be realized as transformations on α-trees. With a view to the definition of such transformations, we introduce the following operation on quasi-α-trees, defined up to isomorphism: the result of *grafting* a quasi-α-tree

$\mathcal{T} = (N, L, S, B)$ onto a node n of a quasi-α-tree $\mathcal{T}' = (N', L', S', B')$ is the quadruple $(N - N'' \cup N''', S - S'' \cup S''', L - L'' \cup L''', B - B'' \cup B''')$, where (N'', L'', S'', B'') is the subtree of $\mathcal{T}$ rooted at n and (N''', L''', S''', B''') is a quasi-α-tree isomorphic to $\mathcal{T}$, rooted at n, and whose set of nodes N''' is disjoint from $N - N''$.

From now on we shall refer to isomorphic quasi-α-trees as if they were identical, and we shall speak in the singular of *the* α-tree of a given class.

B.1.3 Conversion in terms of α-trees

In an α-tree $\mathcal{T} = (N, S, L, B)$, a β-redex is a subtree rooted at a node n, which is of the following form, as depicted in the upper half of figure B.3: the label of n is FML_α; n has two children (from left to right) p and q; p is labeled $\text{FML}_{\alpha\beta}$ and has three children (from left to right) r, s and t; r is a leaf node labeled by the symbol λ; s is labeled by VAR_β, and has a single child u, which is an unlabeled leaf node; t is labeled FML_α and is the root of a subtree $\mathcal{T}_t$; we call $v_1 \ldots v_j$ the j $(0 \leq j)$ leaf nodes of $\mathcal{T}_t$ which B maps to u, and $w_1 \ldots w_j$ their parents (which are labeled FML_β); finally, q is labeled FML_β and is the root of a subtree $\mathcal{T}_q$. To *reduce* such a redex is to graft $\mathcal{T}_q$ onto $\mathcal{T}_t$ at nodes $w_1 \ldots w_n$ and then to graft the result onto $\mathcal{T}$ at node n. The result is the tree $\mathcal{T}'$ depicted in the lower half of figure B.3.

A pair of α-classes $(\mathcal{A}, \mathcal{A}')$ constitutes a step of β-conversion iff the α-tree of $\mathcal{A}'$ is the result of reducing a β-redex in the α-tree of $\mathcal{A}$.

In an α-tree $\mathcal{T} = (N, S, L, B)$, a γ-redex is a node n labeled $\text{FML}_{\alpha\beta}$ which is a not an abstraction node and which is not the left child of an application node.[1] The *reduction* of such a γ-redex is accomplished by "inserting" the *γ-link* of type $\alpha\beta$ at n; the γ-link of type $\alpha\beta$ is the structure, defined up to isomorphism, depicted in figure B.4. (We leave it to the reader to provide an explicit formal definition for it.) The insertion of the γ-link, as shown in figure B.5, consists of grafting the subtree rooted at n onto the *tip* m of the γ-link, then grafting the resulting quasi-α-tree onto node n of $\mathcal{T}$. It should be noted that a γ-link is not a well-formed α-tree or quasi-α-tree (because of

[1] The notions of *abstraction node* and *application node* come in the obvious way from the corresponding notions for formulas of the typed λ-language. Abstraction nodes are characterized by having three children, while application nodes are characterized by having two.

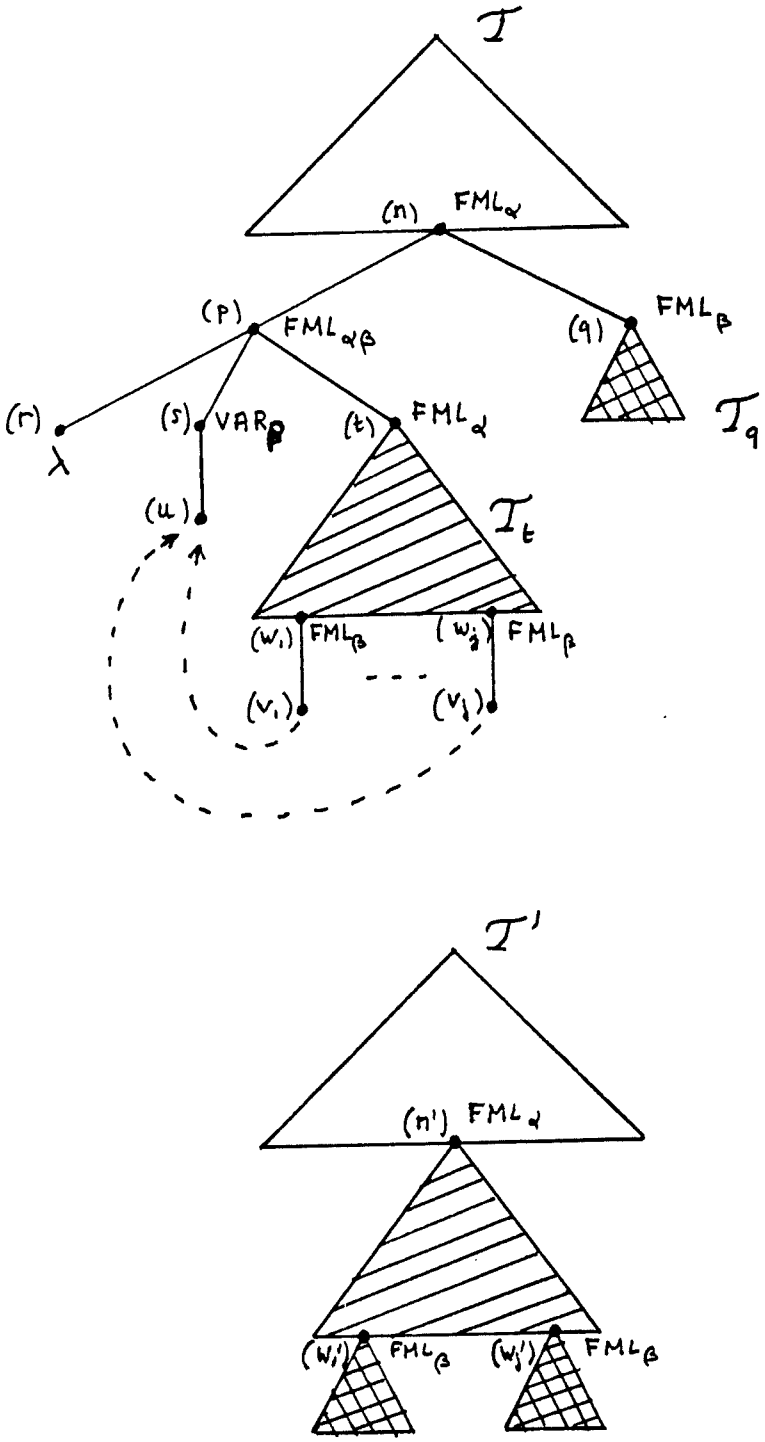

Figure B.3: Reduction of a β-redex

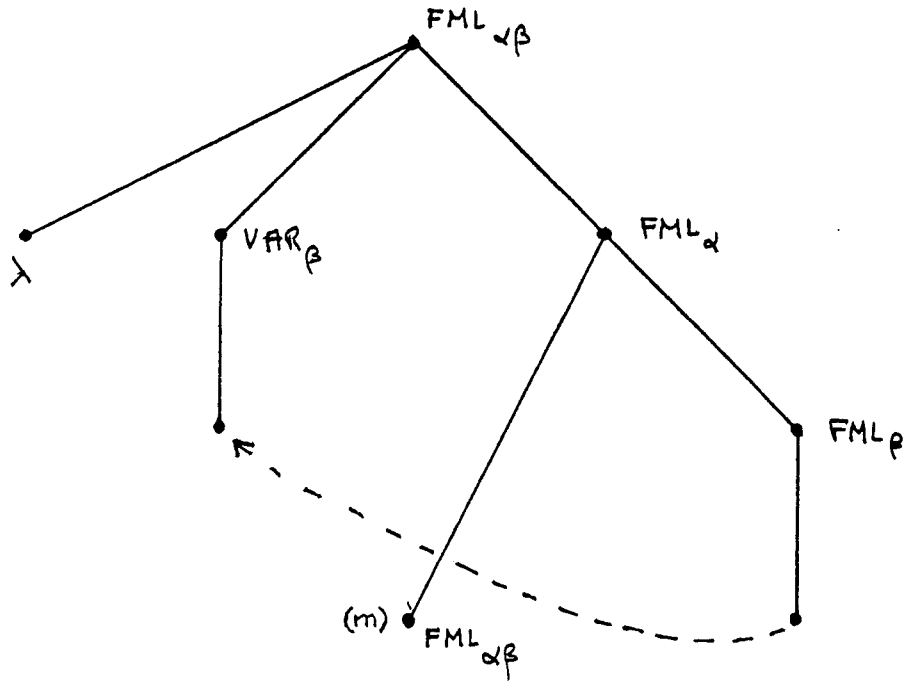

Figure B.4: γ-link of type $\alpha\beta$

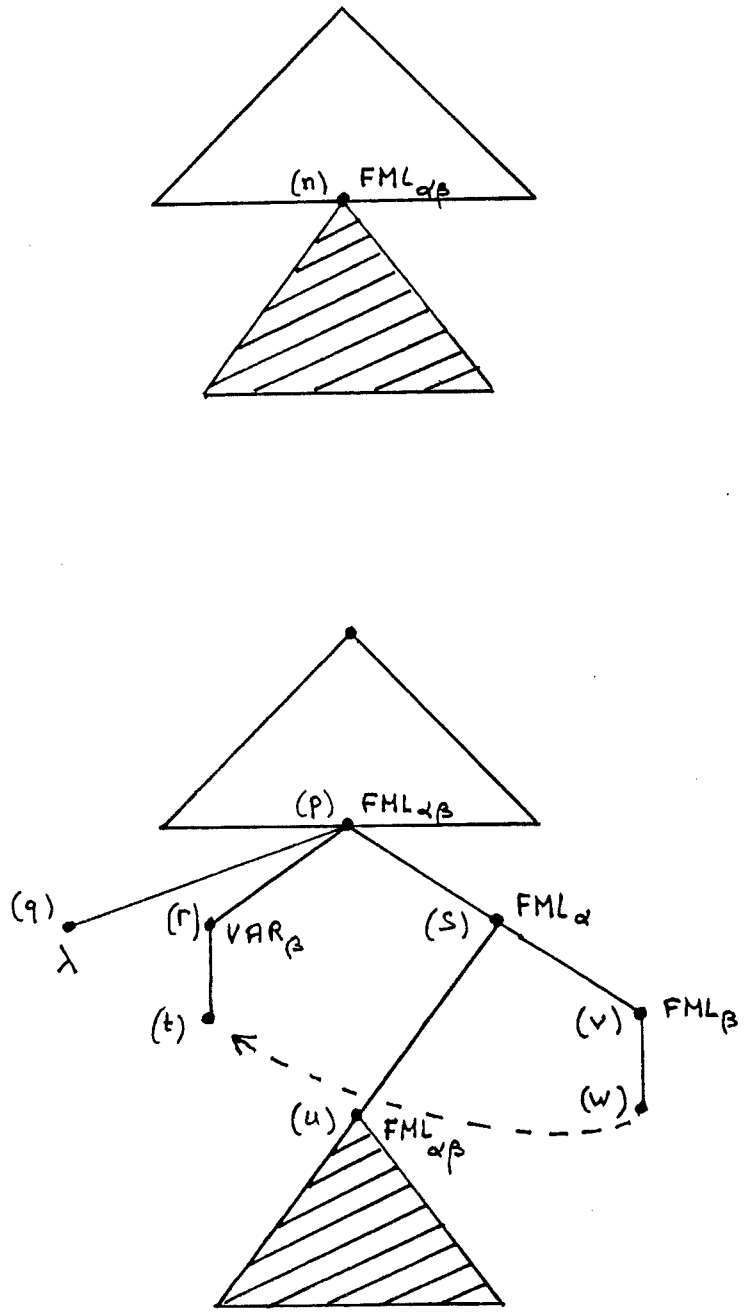

Figure B.5: Reduction of a γ-redex by insertion of a γ-link.

the truncated tip) but the definition of grafting, without modification, can be extended to cover the case of grafting onto a γ-link.

The insertion of a γ-link hardly seems a reduction. A reduction of sorts does take place, however, as we shall see in section B.2.2.

A pair of α-classes $(\mathcal{A}, \mathcal{A}')$ constitutes a step of γ-conversion iff the α-tree of $\mathcal{A}'$ is the result of reducing a γ-redex in the α-tree of $\mathcal{A}$.

Insertion of a γ-link at a node n *without restrictions* on n corresponds to the *opposite* of η-conversion. That is, a pair of α-classes $(\mathcal{A}, \mathcal{A}')$, with associated pair of α-trees $(\mathcal{T}, \mathcal{T}')$, constitutes a step of η-conversion iff $\mathcal{T}$ is the result of inserting a γ-link of type $\alpha\beta$ at a node n labeled $\text{FML}_{\alpha\beta}$ of $\mathcal{T}'$.

In section 2.3.3 we saw, for formula conversion, that steps of η-conversion whose opposite pairs were not steps of γ-conversion were redundant in the presence of α-conversion and β-conversion. For α-classes, a step of η-conversion whose opposite pair is not a step of γ-conversion is a pair $(\mathcal{A}, \mathcal{A}')$, with associated pair of α-trees $(\mathcal{T}, \mathcal{T}')$, such that $\mathcal{T}$ results from $\mathcal{T}'$ by insertion of the γ-link of type $\alpha\beta$ at a node n labeled $\text{FML}_{\alpha\beta}$, in the two cases in which such a node n is not a γ-redex: when n is the left child of an application node in $\mathcal{T}'$, and when n is an abstraction node. In both cases, $(\mathcal{A}, \mathcal{A}')$ is a β-conversion step.

B.1.4 Newman's theorem

The definitions and results in this section are taken from [30].

We shall write $\mathcal{R}^*$ for the transitive-reflexive closure of a binary relation $\mathcal{R}$.

Definition B.1 *A binary relation $\mathcal{R}$ is* confluent *iff whenever $(a, b) \in \mathcal{R}^*$ and $(a, c) \in \mathcal{R}^*$ there exists d such that $(b, d) \in \mathcal{R}^*$ and $(c, d) \in \mathcal{R}^*$.*

Definition B.2 *A binary relation $\mathcal{R}$ is* locally confluent *iff whenever $(a, b) \in \mathcal{R}$ and $(a, c) \in \mathcal{R}$ there exists d such that $(b, d) \in \mathcal{R}^*$ and $(c, d) \in \mathcal{R}^*$.*

Definition B.3 *A binary relation $\mathcal{R}$ is* noetherian *iff it has no infinite chains (i.e. iff there is no infinite sequence where every pair of consecutive items is an element of $\mathcal{R}$).*

Remark. A relation $\mathcal{R}$ is *well-founded* iff every non-empty set S has an element x which is minimal for $\mathcal{R}$ (i.e. which is such that $(y, x) \in \mathcal{R}$ implies

$y \notin S$). If $\mathcal{R}$ is well-founded, then its inverse $\mathcal{R}^{-1}$ is obviously noetherian. Conversely, if $\mathcal{R}^{-1}$ is noetherian, then $\mathcal{R}$ is well-founded (using the axiom of choice, or just dependent choice). We shall use one or the other terminology depending on which relation, $\mathcal{R}$ or $\mathcal{R}^{-1}$, it is more convenient to use.

Theorem B.1 (Newman) *A noetherian relation is confluent iff it is locally confluent.*

PROOF. See [30]. □

Definition B.4 *y is a* normal form *of x for the binary relation $\mathcal{R}$ iff $(x, y) \in \mathcal{R}^*$ and there exists no z such that $(y, z) \in \mathcal{R}$.*

(Notice that this notion of normal form is in agreement with the definitions of β-nf, γ-nf and $\beta\gamma$-nf for α-classes, and with the definition of γ-nf for formulas. But it is not in agreement with the definition of β-nf and $\beta\gamma$-nf for formulas, since a formula may have a β-redex, and hence not be in normal form in the sense of section 2.3.3, while no step of β-conversion applies to it (because of variable capture) and hence the formula is in normal form for β-conversion in the above sense.)

Clearly, if a relation is noetherian every formula has at least one normal form, and if a relation is confluent, every formula has at most one normal form. Thus if a relation is both noetherian and confluent (or, by Newman's theorem, noetherian and locally confluent), then every formula has exactly one normal form.

B.1.5 Well-ordering of finite multisets

In this section we define the ordering of multisets on an ordered set, we provide a useful criterion for establishing that one multiset is less than another, and we prove that multisets on a well-ordered set are well ordered. We must distinguish two kinds of multisets. Dershowitz [14] does not make the distinction, but his arguments cannot be carried out rigorously without making it. Outside of this appendix, by "multiset" we shall refer to a multiset of the first kind. The proof of the main lemma, lemma B.4 is adapted from [14].

Two kinds of multisets

Let A be a set. A *multiset of the first kind on* A is a function with domain A whose range consists of cardinal numbers. A *multiset of the second kind on* A is simply a function whose range is a subset of A. A multiset M of the first kind is *associated* with a multiset f of the second kind iff, for every $x \in A$, $M(x)$ is the cardinality of $f^{-1}(x)$ (the set of inverse images of x by f). The cardinal number $M(x)$ is referred to as the *count* of x in the multiset of the first kind M.

We shall say that a multiset of the second kind is *finite* iff its domain is finite (i.e. iff it is finite as a set of ordered pairs). We shall say that a multiset of the first kind is *finite* (as such a multiset) iff all the counts are finite and only a finite number of them are not zero. We shall only be concerned with finite multisets of either kind.

Remark. Each finite multiset of the second kind determines a finite multiset of the first kind. Conversely, each finite multiset of the first kind is associated with at least one finite multiset of the second kind (proof left to the reader). In fact, a finite multiset of the first kind is associated with "many" finite multisets of the second kind, so many that they do not form a set. So the finite multisets of the second kind on a set A do not form a set. The finite multisets of the first kind on A, however, do form a set, which we shall write $\mathcal{M}_A$.

Ordering of multisets

Let now $<$ be a binary relation on A. We define a binary relation on the set of multisets of the first kind on A as follows: M is less than M', $M < M'$ (we use the same notation, but there should be no risk of confusion), iff

1. For some $x \in A$, $M(x) < M'(x)$, and
2. For every $x \in A$, if $M'(x) < M(x)$ then there exists $y > x$ such that $M(y) < M'(y)$.

Let f and g be multisets of the second kind. We say that a function h from the domain of f into the domain of g is a *bridge function* from f to g (relative to the binary relation $<$ on A) iff every y in the domain of g falls in one of two categories:

1. y has exactly one inverse x by h, and $f(x) = g(y)$ or
2. every inverse x of y by h is such that $f(x) < g(y)$,

and moreover the second category is non-empty. We say that f is *less than* g, $f < g$ (we use the same notation once again, but there should be no risk of confusion), iff there exists a bridge function from f to g. (Of course, there is no set whose elements are the ordered pairs (f, g) such that $f < g$ in this sense.)

Lemma B.2 *Let f and g be finite multisets of the second kind on A, and let M and M' be the corresponding multisets of the first kind. If $<$ is an ordering on A, then $f < g$ iff $M < M'$.*

PROOF. Left to the reader. □

Remark. Relying on lemma B.2, we shall usually establish $M < M'$ by constructing a bridge function from f to g.

Lemma B.3 *If $(A, <)$ is an ordered set, then so is $(\mathcal{M}_A, <)$.*

PROOF. The relation $<$ on finite multisets of the first kind is obviously irreflexive. We show that it is transitive. Let M, M' and M'' be finite multisets of the first kind on A, such that $M < M'$ and $M' < M''$. Let f, f', f'' be finite multisets of the second kind associated with M, M' and M''. By lemma B.2 $f < f'$ and $f' < f''$, so there exist bridge functions h from f to f' and h' from f' to f''. The composition $h' \circ h$ is then a bridge function from f to f''. Therefore $f < f''$ and by lemma B.2 again, $M < M''$. □

Lemma B.4 *There are no infinite descending chains of finite multisets of the second kind on a well-ordered set.*

PROOF(adapted from [14]). Let $(A, <)$ be a well-ordered set. We reason by contradiction. Assume that there exists an infinite descending chain of finite multisets of the second kind $(f_i)_{i \geq 0}$. Let B_i be the domain of each f_i.

We construct a sequence of trees $(t_i)_{i \geq 0}$ as follows. We start with a root node, and construct t_0 by installing as children of the root as many nodes as there are elements in B_0, labeled by the elements of B_0. We refer to these labels as B-labels. Then we add a second kind of labels, called A-labels, the A-label of each of the children of the root being the image by f_0 of its B-label. Now assume that we have constructed the tree t_i, and that the following induction conditions hold in t_i:

1. Every node other than the root has an A-label; if node p is a child of node q and q is not the root, the A-label of p is less than the A-label of q.

2. Only leaf nodes have B-labels; the B-labeling is a bijection between a subset of the leaf nodes and B_i.

3. If a leaf node has a B-label x, then its A-label is $f_i(x)$.

(Observe that t_0 satisfies the induction conditions.) Let h be a bridge function from f_{i+1} to f_i. We construct t_{i+1} as follows. We consider each leaf node n which does have a B-label y.

1. If y has inverse images $x_1 \ldots x_k$ $(k \geq 0)$ by h, and $f_{i+1}(x_j) < f_i(y)$ for every j, $1 \leq j \leq k$, we install k nodes $p_1 \ldots p_k$ as children of n. To each p_j we assign the A-label x_j and the B-label $f_{i+1}(x_j)$. Finally we remove the B-label of n. (Notice that in the case where $k = 0$ this reduces to removing the B-label of n; no children are installed.)

2. If y has one inverse image x, and $h(x) = y$, we change the B-label of n to x.

It is clear that t_{i+1} satisfies the induction conditions.

If at some stage of the construction no new nodes are added, at least one B-label will be removed. So this can only happen a finite number of consecutive times. Therefore the number of nodes of t_i is unbounded.

Having constructed the sequence of doubly-labeled trees $(t_i)_{i\geq 0}$, let $(t'_i)_{i\geq 0}$ be the sequence of singly-labeled trees obtained by dropping the B-labeling component from each tree. Each t'_i is a subtree of t'_{i+1}. The componentwise union of the t'_i is then a tree t'_ω.

Since the size of the t'_i is unbounded, t'_ω is an infinite tree. By Koenig's lemma it has in infinite branch. The A-labels along this branch constitute an infinite descending chain in $(A, <)$, a contradiction. □

Theorem B.5 (Dershowitz) *If $(A, <)$ is a well-ordered set, then so is $(\mathcal{M}_A, <)$.*

PROOF. By contradiction. Let $(M_i)_{0\leq i}$ be an infinite descending chain of finite multisets of the first kind on A. There exists a sequence $(f_i)_{0\leq i}$

where each f_i is a finite multiset of the second kind associated with M_i.[2] By lemma B.2 $(f_i)_{0\leq i}$ is a descending chain, which by lemma B.4 is a contradiction.

B.2 Normalization

B.2.1 Conversion to β-nf

The normalization result regarding β-conversion is the strong normalization theorem for the typed λ-calculus. We state the result without proof for α-classes. Then we derive the result for formulas, as a corollary. The purpose of this is to follow the same pattern that we shall follow for the normalization results regarding conversion to γ-nf and to $\beta\gamma$-nf. Proofs of the result in the literature, though, are usually given for formulas.

For additional definitions and remarks see section 2.3.3.

Theorem B.6 *β-conversion of α-classes is confluent and noetherian; therefore every α-class has a unique β-nf.*

PROOF. Omitted. Confluence can be established as in the untyped λ-calculus. The first proof of confluence is due to Church [11]. More recent proofs can be found in [7, §11.1] and [29, App. 1]. The termination result is due to Türing. A proof can be found in [29, App. 2]. □

Corollary B.7 *Every sequence of formulas which is a non-trivial chain of $\alpha\beta$-conversion steps terminates. Every formula has a β-nf. The β-nfs of a given formula are all the same up to renaming of bound variables.*

PROOF. If $(\boldsymbol{A}, \boldsymbol{B})$ is a formula α-conversion step, then $\overline{\boldsymbol{A}} = \overline{\boldsymbol{B}}$, and if $(\boldsymbol{A}, \boldsymbol{B})$ is a formula β-conversion step, then $(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}})$ is an α-class β-conversion step. Therefore from an infinite non-trivial chain of (formula) $\alpha\beta$-conversion one could derive an infinite chain of class β-conversion. So there are no infinite non-trivial chains of (formula) $\alpha\beta$-conversion. For every formula $\boldsymbol{A}$ there is a complete non-trivial chain starting with $\boldsymbol{A}$ (section 2.3.3). Such a chain must terminate, and the last formula in it is a β-nf of $\boldsymbol{A}$. So every formula has a

[2] AC not needed.

β-nf. If $\boldsymbol{B}$ is a β-nf of $\boldsymbol{A}$ then, by definition (section 2.3.3), $\boldsymbol{A}$ $\alpha\beta$-converts to $\boldsymbol{B}$, and $\boldsymbol{B}$ is in β-nf. So $\overline{\boldsymbol{A}}$ β-converts to $\overline{\boldsymbol{B}}$ and $\overline{\boldsymbol{B}}$ is in β-nf as an α-class; i.e. $\overline{\boldsymbol{B}}$ is *the* β-nf of $\overline{\boldsymbol{A}}$. Thus all the β-nfs of $\boldsymbol{A}$ are representatives of one and the same α-class, and hence they are all the same up to renaming of bound variables. □

B.2.2 Conversion to γ-nf

Theorem B.8 *γ-conversion between α-classes is noetherian and confluent; therefore every α-class has a unique γ-nf.*

PROOF. Consider the reduction of a γ-redex as shown in figure B.5. The nodes q, r, t, w cannot be γ-redexes because they are not labeled FML_γ. Node p cannot be a γ-redex because it is the root of an abstraction. Node u cannot be a γ-redex because it is the left child of an application node. This leaves only s and v as possible γ-redexes among the new nodes introduced by the reduction. On the other hand the reduction eliminates the γ-redex n. The types of the potential new redexes are α and β, while the type of the eliminated redex is $\alpha\beta$. The order of β is strictly less than the order of $\alpha\beta$. The order of α is at most equal to the order of $\alpha\beta$, while the arity of α is one less than the arity of $\alpha\beta$. So if we define an ordering on the types: $\eta < \theta$ iff the order of η is less than the order of θ or the orders are the same but the arity of η is less than the arity of θ, then we have:

$$\begin{cases} \alpha < \alpha\beta \\ \beta < \alpha\beta \end{cases}$$

This means (by lemma B.2) that the multiset of types of γ-redexes (i.e. the function which maps each type α to the number of γ-redexes of type α) is less after than before the reduction. The ordering $<$ on the set of types is clearly a well-founded relation; therefore, by theorem B.5, there cannot be an infinite descending chain of multisets. So there cannot be an infinite descending chain of γ-reductions; i.e. class γ-conversion is noetherian.

By theorem B.1, to prove that class γ-conversion is confluent it suffices to prove that it is locally confluent. But this is obvious: if an α-tree has two redexes n_1 and n_2, the result of inserting a γ-link at n_1, then at n_2 (i.e. at the node which obviously corresponds to n_2 in the α-tree resulting from

the first insertion) is the same α-tree (of course up to isomorphism) which results from first inserting a γ-link at n_2, then at n_1. □

Corollary B.9 *Every sequence of formulas which is a chain of γ-conversion steps terminates. Every formula has a γ-nf. The γ-nfs of a given formula are all the same up to renaming of bound variables.*

PROOF. If $(\boldsymbol{A}, \boldsymbol{B})$ is a step of formula γ-conversion, $(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}})$ is a step of α-class γ-conversion. So if $(\boldsymbol{A}_n)_{n\geq 0}$ were an infinite chain of formula γ-conversion, $(\overline{\boldsymbol{A}_n})_{n\geq 0}$ would be an infinite chain of α-class γ-conversion. Therefore there is no infinite chain of formula γ-conversion. Hence every formula has a γ-nf. If the formula $\boldsymbol{B}$ is a γ-nf of the formula $\boldsymbol{A}$, the α-class $\overline{\boldsymbol{B}}$ is the γ-nf of the α-class $\overline{\boldsymbol{A}}$. Thus all the γ-nfs of a given formula are elements of one α-class, and hence they are all the same up to renaming of bound variables. □

B.2.3 Conversion to $\beta\gamma$-nf

Theorem B.10 *$\beta\gamma$-conversion between α-classes is noetherian and confluent; therefore every α-class has a unique $\beta\gamma$-nf.*

PROOF. We prove first local confluence, then termination; confluence will then follow from theorem B.1.

Local confluence

Let $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{A}''$ be three α-classes such that $(\mathcal{A}, \mathcal{A}')$ and $(\mathcal{A}, \mathcal{A}'')$ are $\beta\gamma$-conversion steps. We want to show that there exists an α-class $\mathcal{B}$ such that $\mathcal{A}'$ and $\mathcal{A}''$ convert to $\mathcal{B}$ (by perhaps multiple steps of $\beta\gamma$-conversion). If in fact $(\mathcal{A}, \mathcal{A}')$ and $(\mathcal{A}, \mathcal{A}'')$ are both β-conversion steps, the existence of $\mathcal{B}$ follows from the confluence of β-conversion. If they are both γ-conversion steps, the existence of $\mathcal{B}$ follows from the confluence of γ-conversion.

It remains to consider the case where one of them, say $(\mathcal{A}, \mathcal{A}')$, is a β-conversion step, while the other, $(\mathcal{A}, \mathcal{A}'')$, is a γ-conversion step. Let $\mathcal{T}$, $\mathcal{T}'$, $\mathcal{T}''$ be the α-trees of $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{A}''$. Since $\mathcal{T}'$ results from $\mathcal{T}$ by reduction of a β-redex, $\mathcal{T}$ and $\mathcal{T}'$ are as figure B.3 shows. $\mathcal{T}''$ results from insertion of the γ-link of type θ at a node m labeled FML_θ, where θ is a functional type. The node m cannot be r, s, u or any of the nodes v_i $(1 \leq i \leq j)$, since these do not have such a label. It cannot be p either, since p is the root of an abstraction. The following possibilities remain for m:

1. m is one of the nodes in the unshaded triangle; i.e. m is a node of $\mathcal{T}$ which is not in the subtree rooted at n, or m coincides with n. There is a corresponding node m' in $\mathcal{T}'$. Insertion of the γ-link of type θ at m' results in an α-tree $\mathcal{T}'''$; and $\mathcal{T}'''$ results also from $\mathcal{T}''$ by a β-conversion step. Figure B.6-I summarizes the relationships between the four α-trees. The α-class associated with $\mathcal{T}'''$ can then play the role of $\mathcal{B}$.

2. m is one of the nodes of singly shaded triangle, other than t; i.e. m is one of the nodes of the subtree rooted at t, $\mathcal{T}_t$, but not t or one of the nodes v_i $(1 \leq i \leq j)$—it may be one of the nodes w_i $(1 \leq i \leq j)$. Same as case 1.

3. m is a node of the doubly shaded triangle other than q; i.e. m is a node of the subtree rooted at q, $\mathcal{T}_q$, other than the root. There are j nodes $m_1 \ldots m_j$ which correspond to m in $\mathcal{T}'$, one in each of the j copies of $\mathcal{T}_q$ rooted at $w'_1 \ldots w'_j$. By insertion of the γ-link of type θ at $m_1 \ldots m_j$, i.e. by j consecutive γ-conversion steps, $\mathcal{T}'$ converts to an α-tree $\mathcal{T}'''$ which results also from $\mathcal{T}''$ be a β-conversion step. The situation is depicted in figure B.6-II. The desired α-class $\mathcal{B}$ is the one associated with $\mathcal{T}'''$.

4. $m = t$ (then $\theta = \alpha$); this can happen only if t is not an abstraction node. The node corresponding to $m = t$ in $\mathcal{T}'$ is n'. As before, the insertion of the γ-link of type θ at n' produces an α-tree $\mathcal{T}'''$ which results also from $\mathcal{T}''$ by reduction of a β-redex. But this time, the insertion of the γ-link does not always constitute the reduction of a γ-redex. We must therefore distinguish two cases:

   (a) n is not the left child of an application node in $\mathcal{T}$. Then n' is not the left child of an application node in $\mathcal{T}'$, hence it is a γ-redex, and insertion of the γ-link does constitute the reduction of a γ-redex. There is then no difference with cases 1 and 2. The relationships between the four trees $\mathcal{T}$, $\mathcal{T}'$, $\mathcal{T}''$ and $\mathcal{T}'''$ is as shown in figure B.6-I. The desired α-class $\mathcal{B}$ is the one associated with $\mathcal{T}'''$.

   (b) n is the left child of an application node in $\mathcal{T}$; hence the same is true of n' in $\mathcal{T}'$. Then n' is *not* a γ-redex, and insertion of

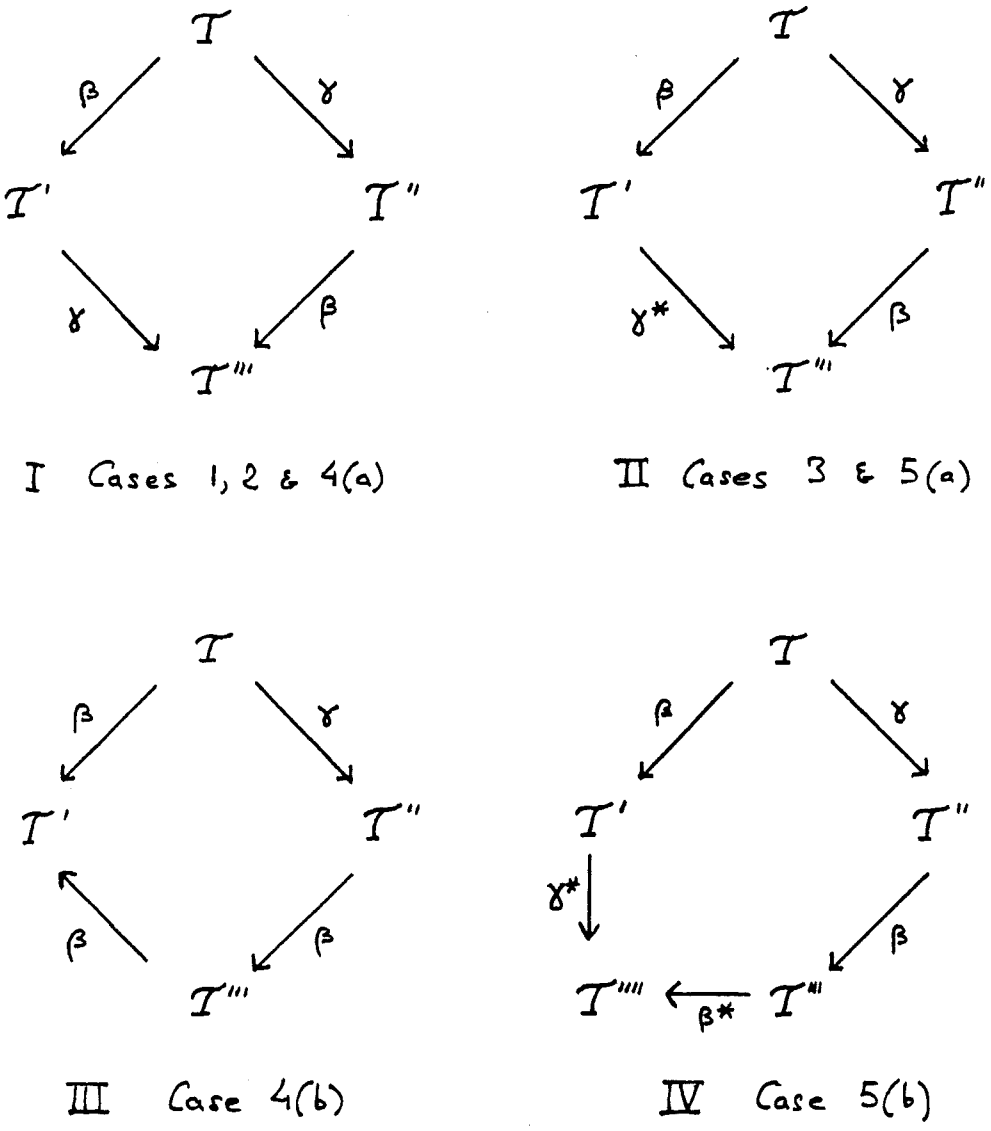

Figure B.6: Confluence of $\beta\gamma$-conversion

the γ-link of type α does *not* correspond to a γ-conversion step. However, as seen in section B.1.3, the opposite transformation, from $\mathcal{T}'''$ to $\mathcal{T}'$, is then a step of β-conversion. Figure B.6-III summarizes the relationships between the four α-trees. As the desired α-class $\mathcal{B}$ to which both $\mathcal{A}'$ and $\mathcal{A}''$ convert we can simply take in this case $\mathcal{A}'$, since two steps of β-conversion take $\mathcal{A}''$ to $\mathcal{A}'$.

5. $m = q$ (then $\theta = \beta$). The nodes corresponding to m in $\mathcal{T}'$ are $w'_1 \ldots w'_j$. By inserting the γ-link of type β at each of those nodes we obtain an α-tree $\mathcal{T}'''$ which results also from $\mathcal{T}''$ by a step of β-conversion. Some of these insertions will constitute γ-conversion steps, some will not, depending on whether the corresponding node w'_i is not or is the left child of an application node in $\mathcal{T}'$. By performing first the insertions which do constitute γ-conversion steps we obtain an intermediate tree $\mathcal{T}''''$. Each of the other insertions is the opposite of a β-conversion step; so $\mathcal{T}'''$ β-converts to $\mathcal{T}''''$. The situation is depicted in figure B.6-IV. As the desired α-class $\mathcal{B}$ we take the one associated with the α-tree $\mathcal{T}''''$.

Termination

Reasoning by contradiction, assume that there exists an infinite chain $(\mathcal{A}_i)_{i\geq 0}$ of $\beta\gamma$-conversion. Since γ-conversion is noetherian, there must be an infinite number of β-conversion steps in the chain. Let $f(i)$ be the integer such that $(\mathcal{A}_{f(i)}, \mathcal{A}_{f(i)+1})$ is the i-th such step, and let $\mathcal{B}_i$ be the γ-nf of $\mathcal{A}_{f(i)}$. Notice that $\mathcal{B}_{i+1}$ is the γ-nf of $\mathcal{A}_{f(i)+1}$ besides being the γ-nf of $\mathcal{A}_{f(i+1)}$, since $\mathcal{A}_{f(i)+1}$ γ-converts to $\mathcal{A}_{f(i+1)}$. We are going to show that there is a chain of β-conversion, having at least one step, which takes $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$. This means that the formulas $\mathcal{B}_i$, $i \geq 0$, are part of an infinite chain of β-conversion, a contradiction since β-conversion is noetherian.

Let $\mathcal{T}$, $\mathcal{T}'$, $\mathcal{T}''$ be the α-trees associated with $\mathcal{A}_{f(i)}$, $\mathcal{A}_{f(i)+1}$ and $\mathcal{B}_i$. $\mathcal{T}'$ results from $\mathcal{T}$ by reduction of a β-redex, so $\mathcal{T}$ and $\mathcal{T}'$ are as depicted in figure B.3. $\mathcal{T}''$ results from $\mathcal{T}$ by insertion of γ-links of appropriate types at all the γ-redexes of $\mathcal{T}$. Thus we have the same situation that was discussed in the proof of local confluence, except that all the γ-redexes of $\mathcal{T}$ are involved rather than just one of them.

As explained in the proof of local confluence, each one of the γ-redexes of $\mathcal{T}$ has zero, one or more images in $\mathcal{T}'$: those in the unshaded and singly

shaded triangles of the upper half of figure B.3 have one image, those in the doubly shaded triangle have one image in each of the copies of the triangle grafted on $w_1 \dots w_j$ ($j \geq 0$). Thus to the set N of γ-redexes in $\mathcal{T}$ corresponds a set of nodes N' in $\mathcal{T}'$. The insertion of γ-links of appropriate types at each one of the nodes in N' results in a tree $\mathcal{T}'''$, which results also from $\mathcal{T}''$ by reduction of one β-redex. Not all the nodes in N' are γ-redexes. For those which are γ-redexes, insertion of the γ-link is reduction of the redex; for those which are not, insertion of the γ-link is the opposite of the reduction of a β-redex. Let $\mathcal{T}''''$ be the tree resulting from $\mathcal{T}'$ by insertion of the γ-links at those which are γ-redexes. Then $\mathcal{T}''''$ also results from $\mathcal{T}'''$ by removal of the other γ-links, i.e. by as many β-conversion steps. We have therefore the situation depicted in figure B.7.

It is clear that every γ-redex of $\mathcal{T}'$ must be one of the images of a γ-redex of $\mathcal{T}$. Therefore all the γ-redexes of $\mathcal{T}'$ are elements of N', and $\mathcal{T}''''$ is the result of reducing *all* the γ-redexes of $\mathcal{T}'$. So the α-class associated with $\mathcal{T}''''$ is the γ-nf of $\mathcal{A}_{f(i)+1}$, i.e. $\mathcal{B}_{i+1}$. But then a step of β-conversion takes $\mathcal{B}_i$ to the α-class associated with $\mathcal{T}'''$, which itself β-converts to $\mathcal{B}_{i+1}$. So we have proved that there is indeed a β-conversion chain, of non-zero length, taking $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$. □

Corollary B.11 *Every sequence of formulas which is a non-trivial chain of $\alpha\beta\gamma$-conversion steps terminates. Every formula has a $\beta\gamma$-nf. The $\beta\gamma$-nfs of a given formula are all the same up to renaming of bound variables.*

PROOF. See the proof of corollary B.7. □

B.2.4 Properties of $\beta\gamma$-nf

Lemma B.12 *If $\boldsymbol{A}$ is a formula of atomic type in $\beta\gamma$-nf, there exist an integer $n \geq 0$, a symbol $\boldsymbol{s}$, and n formulas $\boldsymbol{B}_1 \dots \boldsymbol{B}_n$ such that $\boldsymbol{A}$ is:*

$$\boldsymbol{s}\,\boldsymbol{B}_1\ \dots\ \boldsymbol{B}_n.$$

Moreover, for every i, $1 \leq i \leq n$, there exist an integer $m \geq 0$, m variables $\boldsymbol{x}_1 \dots \boldsymbol{x}_m$ and a formula $\boldsymbol{C}$ of atomic type (and in $\beta\gamma$-nf) such that $\boldsymbol{B}_i$ is the formula:

$$\lambda \boldsymbol{x}_1 \dots \lambda \boldsymbol{x}_m \boldsymbol{C}.$$

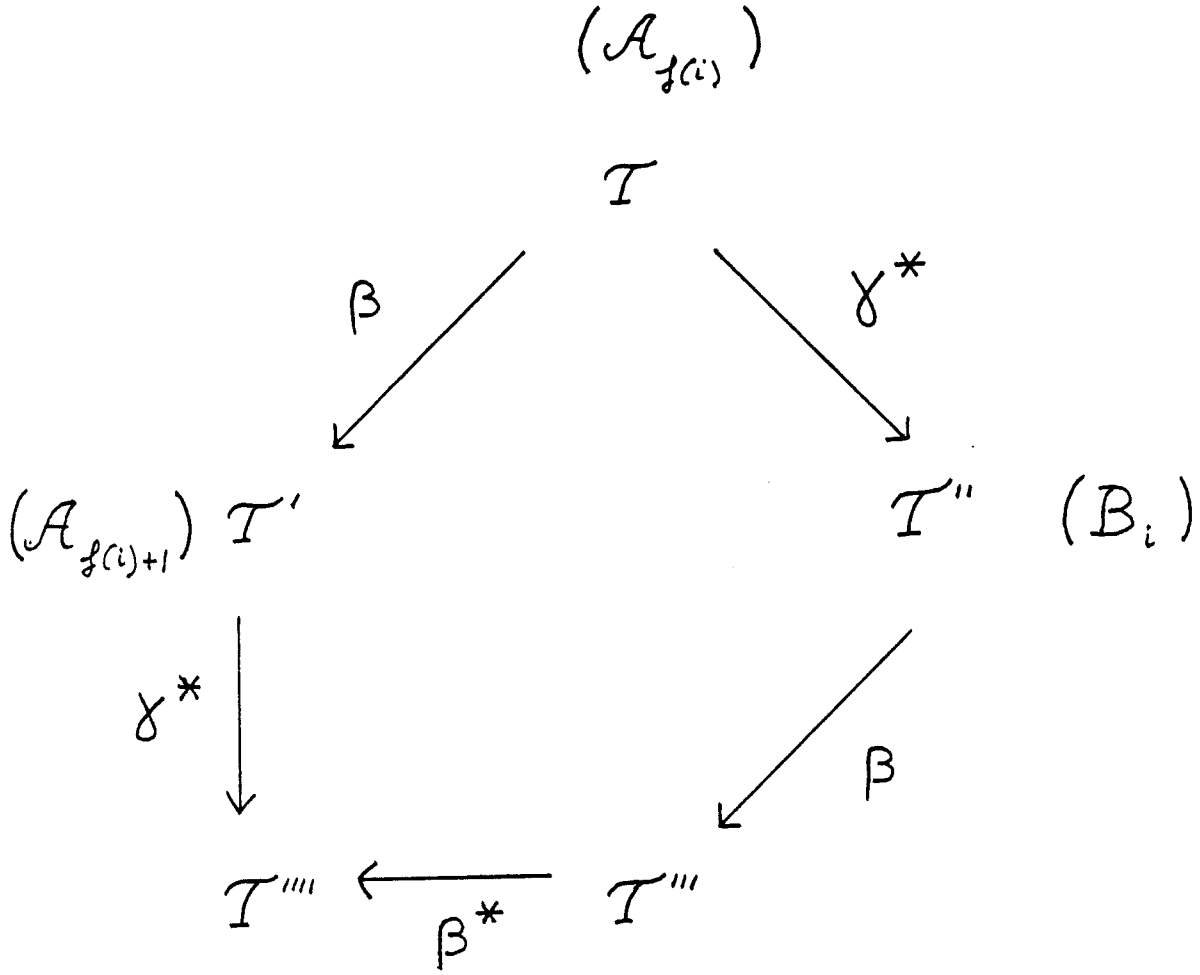

Figure B.7: Termination of $\beta\gamma$-conversion

PROOF. Consider a parse tree for $\boldsymbol{A}$. Let p_0 be the root of the tree. Since the type of $\boldsymbol{A}$ is atomic, p_0 cannot be an abstraction node. If it is a symbol node, let $n = 0$ and let s be the associated symbol. If it is an application node, let p_1 be its left child. If p_1 is itself an application node, let p_2 be its left child, and so on until we reach a node p_n which is not an application node. p_n cannot be an abstraction node, otherwise p_{n-1} would be the root of a β-redex. Therefore p_n must be a symbol node. Let s be the associated symbol. Let $q_1 \ldots q_n$ be the right children of $p_{n-1} \ldots p_0$, and let $\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$ be the formulas associated with the subtrees rooted at $q_1 \ldots q_n$. As desired, $\boldsymbol{A}$ is then the formula:

$$s\,\boldsymbol{B}_1\,\ldots\,\boldsymbol{B}_n.$$

Now consider an arbitrary i, $1 \leq i \leq n$. If q_i is not an abstraction node it must be of atomic type: otherwise it would be a γ-redex. Then let $m = 0$ and let $\boldsymbol{C}$ be $\boldsymbol{B}_i$. If q_i is an abstraction node, let r_1 be the root of the body of the abstraction; if r_1 is itself an abstraction node let r_2 be the root of the body of this second abstraction, and so on until we find that r_m is not an abstraction node. r_m must then be of atomic type, otherwise it would be a γ-redex. Let $\boldsymbol{C}$ be the formula associated with the subtree rooted at r_m, and let $\boldsymbol{x}_1 \ldots \boldsymbol{x}_m$ be the variables bound by the abstractions whose bodies are rooted at $r_1 \ldots r_m$. Then, as desired, $\boldsymbol{B}_i$ is the formula

$$\lambda \boldsymbol{x}_1 \ldots \lambda \boldsymbol{x}_m \boldsymbol{C}.$$

□

Lemma B.13 *If a formula $\boldsymbol{A}$ of atomic type in $\beta\gamma$-nf has an occurrence of a symbol s, which is not a binding occurrence of a variable, then it has a subformula of atomic type which is of the form*

$$s\,\boldsymbol{B}_1\,\ldots\boldsymbol{B}_n$$

($n \geq 0$), and where the occurrence of s in question is the one shown. Moreover, for every i, $1 \leq i \leq n$, there exist an integer $m \geq 0$, m variables $\boldsymbol{x}_1 \ldots \boldsymbol{x}_m$ and a formula $\boldsymbol{C}$ of atomic type (and in $\beta\gamma$-nf) such that $\boldsymbol{B}_i$ is the formula

$$\lambda \boldsymbol{x}_1 \ldots \lambda \boldsymbol{x}_m \boldsymbol{C}.$$

PROOF. If s is of atomic type, let $n = 0$. Otherwise consider a parse tree for $\boldsymbol{A}$. Let p_0 be the symbol node corresponding to the occurrence of s. p_o cannot be the root, since $\boldsymbol{A}$ is of atomic type. Let p_1 be the parent of p_0; p_1 must be an application node, and p_0 must be its left child, otherwise p_0 would be a γ-redex. If p_1 is not of atomic type, it must itself have a parent p_2 of which it is the left child, otherwise p_1 would be a γ-redex. And so on until we find a node p_n which is of atomic type. Let $q_1 \ldots q_n$ be the right children of $p_1 \ldots p_n$, and let $\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$ be the formulas associated with the subtrees rooted at $q_1 \ldots q_n$. Then, as desired, the formula rooted at p_n is

$$s\,\boldsymbol{B}_1 \ldots \boldsymbol{B}_n,$$

and it is a subformula of $\boldsymbol{A}$ having an atomic type. The second part of the proof is as in the proof of lemma B.12 above. □

Lemma B.14 *If $\boldsymbol{A}$ is a formula of atomic type in $\beta\gamma$-nf which has an occurrence, other than $\boldsymbol{A}$ itself, of a formula $\boldsymbol{C}$ of atomic type, then $\boldsymbol{A}$ has a subformula of atomic type of the form*

$$s\,\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$$

($n \geq 0$), where s is a symbol and for some i, $1 \leq i \leq n$, $\boldsymbol{B}_i$ is of the form

$$\lambda \boldsymbol{x}_1 \ldots \lambda \boldsymbol{x}_m \boldsymbol{C},$$

the occurrence of $\boldsymbol{C}$ in question being the one shown.

PROOF. Consider a parse tree for $\boldsymbol{A}$, and let r_0 be the root of the subtree associated with the occurrence of $\boldsymbol{C}$ in question. r_0 is not the root, so it has a parent p. If p is an application node, r_0 must be its right child (since it is of atomic type); then we let $m = 0$ and we call u the node p. If p is an abstraction node, then we let $r_1 = p$. r_1 must have a parent; if it is an abstraction node, then we call it r_2, and so on until we reach a node r_m whose parent is not an abstraction node. We call u the parent of r_m; u is an application node, and r_m cannot be its left child, because u would then the root of a β-redex. So r_m is the right child of u (as in the case $m = 0$ above).

We let v_1 be the left child of u. If v_1 is not a symbol node it must be an application node (otherwise its parent would be the root of a β-redex); then

we let v_2 be its left child, and so on until we reach a node v_i ($i \geq 1$) which is a symbol node. Let s be the symbol associated with v_i.

We let $w_1 = u$. If w_1 has functional type it must be the left child of a node which we call w_2 (otherwise it would be a γ-redex), and so on until we reach a node w_j ($j \geq 1$)) which is of atomic type.

Let $n = i + j - 1$ and let $q_1, \ldots, q_{i-1}, q_i, \ldots, q_n$ be the right children of $v_{i-1}, \ldots, v_1, w_1, \ldots, w_j$ (which reduces to $w_1, \ldots, w_j$ when $i = 1$). Let $\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$ be the formulas associated with the subtrees rooted at $q_1 \ldots q_n$. Let $\boldsymbol{x}_1 \ldots \boldsymbol{x}_m$ be the variables bound by the abstractions rooted at $r_m \ldots r_1$. Then the subformula of $\boldsymbol{A}$ associated with the subtree rooted at w_j is

$$s\,\boldsymbol{B}_1 \ldots \boldsymbol{B}_n.$$

It has an atomic type. And $\boldsymbol{B}_i$ is:

$$\lambda \boldsymbol{x}_1 \ldots \lambda \boldsymbol{x}_m \boldsymbol{C}.$$

□

Lemma B.15 *If $\boldsymbol{A}$ is a formula of atomic type in $\beta\gamma$-nf, and if the free symbols of $\boldsymbol{A}$ have types of order 2 or less, then the bound variables of $\boldsymbol{A}$ are of type ι.*

PROOF. By induction on the depth of $\boldsymbol{A}$. Assume $\boldsymbol{A}$ is of depth k, and assume the lemma holds for all formulas of depth less than k. By lemma B.12, $\boldsymbol{A}$ is of the form

$$s\,\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$$

($n \geq 0$) where s is a symbol. Let us show that the bound variables of each B_i are of type ι. Again by lemma B.12, each $\boldsymbol{B}_i$ is of the form

$$\lambda \boldsymbol{x}_1 \ldots \lambda \boldsymbol{x}_m \boldsymbol{C},$$

($m \geq 0$) where $\boldsymbol{C}$ is of atomic type (and, of course, in $\beta\gamma$-nf). By induction hypothesis, all variables bound within $\boldsymbol{C}$ are of type ι. It remains to show that $\boldsymbol{x}_1 \ldots \boldsymbol{x}_m$ are of type ι. If δ is the type of $\boldsymbol{C}$ and $\alpha_1, \ldots, \alpha_m$ are the types of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$, the type of $\boldsymbol{B}_i$ is $\delta\alpha_m \ldots \alpha_1$. Since the order of the type of s is at most 2 the order of the type of $\boldsymbol{B}_i$ is at most 1, i.e. the order of $\delta\alpha_m \ldots \alpha_1$ is at most 1. Hence the order of each of $\alpha_1, \ldots, \alpha_m$ is 0, i.e. $\alpha_1 = \ldots = \alpha_m = \iota$. □

Definition B.5 *The standard formulas generated by a set of symbols S are defined inductively as follows:*

1. *If $\boldsymbol{s}$ is a symbol of atomic type which is an element of S, or if $\boldsymbol{s}$ is a variable of type ι, "$\boldsymbol{s}$" is a standard formula generated by S.*

2. *If $\boldsymbol{A}$ is a standard formula generated by S and $\boldsymbol{x}$ is a variable, "$\lambda\boldsymbol{x}\boldsymbol{A}$" is a standard formula generated by S.*

3. *If $\boldsymbol{s}$ is an element of S of type $\delta\alpha_1 \ldots \alpha_n$, where δ is an atomic type, and $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n$ are standard formulas generated by S of types $\alpha_1, \ldots, \alpha_n$, then "$\boldsymbol{s}\,\boldsymbol{A}_1 \ldots \boldsymbol{A}_n$" is a standard formula generated by S.*

Theorem B.16 *S being a set of symbols whose types are of order at most 2, a formula $\boldsymbol{A}$ of atomic type is a standard formula generated by S iff it is in $\beta\gamma$-nf and its free symbols are elements of S or variables of type ι.*

PROOF. Let $\boldsymbol{A}$ be a formula of atomic type. If $\boldsymbol{A}$ is a standard formula generated by S it is obvious, by induction according to definition B.5, that it is in $\beta\gamma$-nf and has no free symbols other than elements of S or variables of type ι. Conversely, we reason by induction on the depth of $\boldsymbol{A}$. Assume that every formula of depth less than k which is in $\beta\gamma$-nf and whose free symbols are elements of S or variables of type ι is a standard formula generated by S. Let $\boldsymbol{A}$ be a formula of depth k in $\beta\gamma$-nf which has no free symbols other than elements of S or variables of type ι. By lemma B.12 $\boldsymbol{A}$ is of the form

$$s\,\boldsymbol{B}_1 \ldots \boldsymbol{B}_n$$

where s is a symbol, which must be an element of S, or a variable of type ι. Every $\boldsymbol{B}_i$ is of the form

$$\lambda\boldsymbol{x}_1 \ldots \lambda\boldsymbol{x}_m\boldsymbol{C},$$

$\boldsymbol{C}$ being a formula of atomic type. By lemma B.15, $\boldsymbol{x}_1 \ldots \boldsymbol{x}_m$ are variables of type ι; hence every symbol free in $\boldsymbol{C}$ which is not a variable of type ι is also free in $\boldsymbol{A}$, and therefore is an element of S. Since $\boldsymbol{C}$ is in $\beta\gamma$-nf (as a subformula of $\boldsymbol{A}$) and its type is atomic, by the induction hypothesis it is a standard formula generated by S. Therefore $\boldsymbol{B}_i$ is also a standard formula generated by S. Since this is the case for every $\boldsymbol{B}_i$, $\boldsymbol{A}$ itself is a standard formula generated by S. □

Appendix C

Equivalence to Church's system

The system of [10] differs from ours in that it uses only the logical constants not_{ooo}, or_{ooo}, $\text{forall}_{o(o\alpha)}$ and $\text{the}_{\alpha(o\alpha)}$ (with other names). In our system, the logical constants not used by Church can be "defined" in terms of those used by Church; more precisely we have:

$$\vdash \text{and}_{ooo} = \lambda p_o \lambda q_o(\neg(\neg p_o \vee \neg q_o))$$
$$\vdash \text{implies}_{ooo} = \lambda p_o \lambda q_o(\neg p_o \vee q_o)$$
$$\vdash \text{equal}_{o\alpha\alpha} = \lambda x_\alpha \lambda y_\alpha \forall f_{o\alpha}(f_{o\alpha}\, x_\alpha \supset f_{o\alpha}\, y_\alpha)$$
$$\vdash \text{true}_o = \forall x_\iota(x_\iota = x_\iota)$$
$$\vdash \text{false}_o = \neg\text{true}_o$$
$$\vdash \text{exists}_{o(o\alpha)} = \lambda p_{o\alpha}(\neg\forall x_\alpha \neg(p_{o\alpha}\, x_\alpha))$$
$$\vdash \text{atmost}_{o(o\alpha)} = \lambda p_{o\alpha} \forall x_\alpha \forall y_\alpha(p_{o\alpha}\, x_\alpha \wedge p_{o\alpha}\, y_\alpha \supset x_\alpha = y_\alpha)$$
$$\vdash \text{unique}_{o(o\alpha)} = \lambda p_{o\alpha}(\exists x_{o\alpha}(p_{o\alpha}\, x_\alpha) \wedge !x_{o\alpha}(p_{o\alpha}\, x_\alpha))$$

The above set of theorems of H.O.L. is an *abbreviation system* (see section 2.5). Let then ϕ be the function which maps every formula $\boldsymbol{A}$ to the formula obtained by eliminating the abbreviations from $\boldsymbol{A}$. We have $\vdash \boldsymbol{A} = \phi(\boldsymbol{A})$; when $\boldsymbol{A}$ is of type o this can be written

$$\vdash \boldsymbol{A} \equiv \phi(\boldsymbol{A}). \tag{C.1}$$

Let Φ be the function which maps every theory to the set of images by ϕ of its axioms.

Remark. Let $\boldsymbol{A}$ be a formula, let $\boldsymbol{B} = \phi(\boldsymbol{A})$, let $\boldsymbol{A}'$ be a $\beta\gamma$-nf of $\boldsymbol{A}$ and $\boldsymbol{B}'$ a $\beta\gamma$-nf of $\boldsymbol{B}$. Because $\boldsymbol{A}'$ is in $\beta\gamma$-nf, by lemma B.13, each occurrence of a logical constant in $\boldsymbol{A}'$ is part of a pattern which can be written using the shorthand corresponding to the constant introduced in section 2.3.4. For example, every occurrence of and_{ooo} is part of a pattern "$\text{and}_{ooo}\,\boldsymbol{A}\,\boldsymbol{B}$" which can be written "$\boldsymbol{A} \wedge \boldsymbol{B}$". So $\boldsymbol{A}'$ can be entirely written using the shorthands for the logical constants: there are no occurrences of logical constants not covered by the shorthands (an example of which would be the occurrence of and_{ooo} shown in "$\boldsymbol{B}\ (\text{and}_{ooo}\,\boldsymbol{A})$". Each shorthand has a reading in Church's system, e.g. "$\boldsymbol{A} \wedge \boldsymbol{B}$" is read "$\neg(\neg\boldsymbol{A} \vee \neg\boldsymbol{B})$" in Church's system. $\boldsymbol{B}'$ is then the formula which results from expanding the shorthands according to their reading in Church's system.

Our system is equivalent to Church's in that $\Gamma \vdash \boldsymbol{A}$ iff $\Phi(\Gamma) \vdash_{\text{Church}} \phi(\boldsymbol{A})$. We give now a sketch of the proof of this equivalence.

Assume $\Phi(\Gamma) \vdash_{\text{Church}} \phi(\boldsymbol{A})$. This means that there exists in Church's system a proof of $\phi(\boldsymbol{A})$ from $\Phi(\Gamma)$. Such a proof is a sequence of formulas $\boldsymbol{A}_0 \ldots \boldsymbol{A}_n$ where $\boldsymbol{A}_n$ is $\phi(\boldsymbol{A})$ and for every i $(1 \leq i \leq n)$ one of the following conditions holds:

1. $\boldsymbol{A}_i$ is a formula of $\Phi(\Gamma)$.

2. $\boldsymbol{A}_i$ is an alphabetic variant of one of Church's axioms. We mean of course one of the axioms 1, 2, 3, 4, 5^{α}, 6^{α}, 9^{α}, $10^{\alpha\beta}$, or (Church's reading of) "$p_o \equiv q_o \supset p_o = q_o$"; an alphabetic variant of an axiom is an axiom obtained by optionally renaming variables (free or bound) occurring in the axiom.

3. $\boldsymbol{A}_i$ follows from previous formulas by one of Church's rules of inference (rules I through VI), with the restriction that no variable free in the hypotheses $\Phi(\Gamma)$ can play the role of x_α in rules IV or VI.

There is no difficulty in showing, by induction on i, that

$$\Gamma \vdash \boldsymbol{A}_i$$

for every i, $1 \leq i \leq n$. In particular $\Gamma \vdash \boldsymbol{A}_n$, i.e. $\Gamma \vdash \phi(\boldsymbol{A})$, and by (C.1),

$$\Gamma \vdash \boldsymbol{A}$$

Conversely, assume $\Gamma \vdash \boldsymbol{A}$. A proof of this in our system is a sequence of pairs $(\Gamma_i, \boldsymbol{A}_i)$ $(1 \leq i \leq n)$ where $(\Gamma_n, \boldsymbol{A}_n)$ is $(\Gamma, \boldsymbol{A})$ and each pair follows from zero or more previous ones by application of a natural-deduction rule of inference. This time we show by induction that, for every i $(1 \leq i \leq n)$ $\Phi(\Gamma_i) \vdash_{\text{Church}} \phi(\boldsymbol{A}_i)$. Again there is no particular difficulty in doing so. For example, consider the case where $(\Gamma_i, \boldsymbol{A}_i)$ follows from $(\Gamma_j, \boldsymbol{A}_j)$ and $(\Gamma_k, \boldsymbol{A}_k)$ $(j < i$ and $k < i)$ by $\wedge$-introduction. That is, $\boldsymbol{A}_i$ is $\boldsymbol{A}_j \wedge \boldsymbol{A}_k$ and Γ_i is $\Gamma_j \cup \Gamma_k$. By the induction hypothesis there is a proof P_j in Church's system of $\phi(\boldsymbol{A}_j)$ from $\Phi(\Gamma_j)$ and a proof P_k of $\phi(\boldsymbol{A}_k)$ from $\Phi(\Gamma_k)$. We want a proof of $\phi(\boldsymbol{A}_i)$ from $\Phi(\Gamma_i) = \Phi(\Gamma_j) \cup \Phi(\Gamma_k)$. If $\boldsymbol{A}'_i$, $\boldsymbol{A}'_j$, $\boldsymbol{A}'_k$ are the images by ϕ of $\boldsymbol{A}_i$, $\boldsymbol{A}_j$, $\boldsymbol{A}_k$, we have

$$\boldsymbol{A}'_i = \text{“}(\lambda p_o \lambda q_o (\neg(\neg p_o \vee \neg q_o))\ \boldsymbol{A}'_j\ \boldsymbol{A}'_k\text{”}$$

Let

$$\boldsymbol{A}''_i = \text{“}\neg(\neg \boldsymbol{A}'_j \vee \neg \boldsymbol{A}'_k)\text{”}.$$

In Church's system there is a proof P of $\boldsymbol{A}''_i$ from $\boldsymbol{A}'_j$ and $\boldsymbol{A}'_k$, since the system is complete for the propositional calculus. And there is a proof P' of $\boldsymbol{A}'_i$ from $\boldsymbol{A}''_i$ by conversion. The concatenation of P_j, P_k, P and P' would be the desired proof of $\boldsymbol{A}'_i$ from $\Phi(\Gamma_i)$, except that proof P_k may apply rules IV or VI to a variable which occurs free in the hypotheses of P_j (or vice versa). This problem can be solved by selectively renaming occurrences of the offending variable in P_k (or P_j) before concatenating.

Appendix D

Proof of the conservative extension theorem

In this appendix $\mathcal{V}$ is an arbitrary F.O. vocabulary.

Given a non-empty set M, a *$\mathcal{V}$-F.O. assignment* into M is a function defined over the union of $\mathcal{V}$ and the set of the individual constants and variables which takes the following kinds of values, or *denotations*:

1. An individual constant or variable denotes an element of M.
2. An n-ary function symbol denotes a function from M^n into M.
3. An n-ary predicate symbol denotes an n-ary relation over M, i.e. a subset of M^n.

A $\mathcal{V}$-F.O. interpretation is a pair (M, ϕ) where M is a non-empty set (the *domain* of the interpretation), and ϕ is a $\mathcal{V}$-F.O. assignment into M.

The *denotation* (in the F.O. sense) of a $\mathcal{V}$-F.O. term $\boldsymbol{T}$ in a $\mathcal{V}$-F.O. interpretation (M, ϕ) is defined inductively as follows:

1. If s is an individual variable or constant its denotation is $\phi(s)$.
2. If $\boldsymbol{f}$ is an n-ary function symbol of $\mathcal{V}$ denoting a function f from M^n into M, and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ are n $\mathcal{V}$-F.O. terms with denotations $t_1 \ldots t_n$, the denotation of the term "$\boldsymbol{f}\ \boldsymbol{T}_1\ \ldots\ \boldsymbol{T}_n$" is $f(x_1, \ldots, x_n)$.

Whether or not a $\mathcal{V}$-F.O. interpretation $\mathcal{I} = (M, \phi)$ *satisfies* (in the F.O. sense) a $\mathcal{V}$-F.O. sentence $\boldsymbol{S}$ is defined inductively as follows:

1. $\boldsymbol{T}$ and $\boldsymbol{T}'$ being $\mathcal{V}$-F.O. terms, $\mathcal{I}$ satisfies "$\boldsymbol{T} = \boldsymbol{T}'$" iff $\boldsymbol{T}$ and $\boldsymbol{T}'$ have the same denotation in $\mathcal{I}$.

2. $\boldsymbol{p}$ being an n-ary predicate symbol, mapped by ϕ to the n-ary relation $r \in M^n$, and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ being n terms with denotations $u_1 \ldots u_n$ in $\mathcal{I}$, $\mathcal{I}$ satisfies "$\boldsymbol{p}\,\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$" iff $(u_1, \ldots, u_n) \in r$.

3. No $\mathcal{V}$-F.O. interpretation $\mathcal{I}$ satisfies "$\bot$".

4. $\boldsymbol{S}$ being a $\mathcal{V}$-F.O. sentence, $\mathcal{I}$ satisfies "$\neg\boldsymbol{S}$" iff it does not satisfy $\boldsymbol{S}$.

5. $\boldsymbol{S}$ and $\boldsymbol{S}'$ being $\mathcal{V}$-F.O. sentences, $\mathcal{I}$ satisfies "$\boldsymbol{S} \wedge \boldsymbol{S}'$" iff it satisfies both $\boldsymbol{S}$ and $\boldsymbol{S}'$; it satisfies "$\boldsymbol{S} \vee \boldsymbol{S}'$" iff it satisfies either $\boldsymbol{S}$ or $\boldsymbol{S}'$; it satisfies "$\boldsymbol{S} \supset \boldsymbol{S}'$" iff it does not satisfy $\boldsymbol{S}$ or else it satisfies $\boldsymbol{S}'$.

6. $\boldsymbol{S}$ being a $\mathcal{V}$-F.O. sentence, and $\boldsymbol{x}$ an individual variable, $\mathcal{I}$ satisfies "$\forall\boldsymbol{x}\boldsymbol{S}$" iff, for every $u \in M$, the interpretation $\mathcal{I}' = (M, \phi')$, where ϕ' maps $\boldsymbol{x}$ to u and otherwise coincides with ϕ, satisfies $\boldsymbol{S}$; it satisfies "$\exists\boldsymbol{x}\boldsymbol{S}$" iff, for some $u \in M$, $\mathcal{I}'$ defined as before satisfies $\boldsymbol{S}$.

A $\mathcal{V}$-F.O. interpretation $\mathcal{I}$ is a *model* (in the F.O. sense) of a $\mathcal{V}$-F.O. theory Γ iff it satisfies all the axioms of Γ. Γ being a F.O. theory, $\boldsymbol{P}$ a F.O. sentence, and $\mathcal{V}$ a F.O. vocabulary which includes all the predicate and function symbols occurring in Γ and $\boldsymbol{P}$, we say that $\boldsymbol{P}$ is a logical consequence of Γ with respect to $\mathcal{V}$, iff every $\mathcal{V}$-F.O. interpretation which is a model of Γ satisfies $\boldsymbol{P}$. This, however, does not depend on $\mathcal{V}$, so we shall simply say that $\boldsymbol{P}$ is a *logical consequence of* Γ *in the F.O. sense*, written $\Gamma \models_{\text{F.O.}} \boldsymbol{P}$.

Theorem D.1 (Soundness and completeness of F.O.L.) Γ *being a* $\mathcal{V}$*-F.O. theory, and* $\boldsymbol{P}$ *a* $\mathcal{V}$*-F.O. sentence,* $\Gamma \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$ *iff* $\Gamma \models_{\text{F.O.}} \boldsymbol{P}$.

PROOF. Known [5, 40]. □

Definition D.1 *Let* $\mathcal{I} = (M, \phi)$ *be a* $\mathcal{V}$*-F.O. interpretation and* $\mathcal{J} = (\mathcal{D}, \psi)$ *a H.O. interpretation as defined in section 2.3.4. We say that* $\mathcal{J}$ *is a* H.O. extension *of* $\mathcal{I}$ *iff* $\mathcal{D}$ *is the frame generated by* M, $\mathcal{J}$ *is a logical interpretation, and:*

1. *For every individual constant or variable* $\boldsymbol{s}$, $\psi(\boldsymbol{s}) = \phi(\boldsymbol{s})$.

2. *For every n-ary function symbol* $\boldsymbol{f}$, *if* $\phi(\boldsymbol{f})$ *is a function* $f : M^n \mapsto M$, *then* $\psi(\boldsymbol{f})$ *is the function* $g \in \mathcal{D}_{\iota\alpha_1\ldots\alpha_n}$, $\alpha_1 = \ldots = \alpha_n = \iota$, *obtained by currying* f, *i.e. such that, for every n-tuple* $(x_1 \ldots x_n) \in M^n$,

$$g(x_1)\ldots(x_n) = f(x_1, \ldots, x_n)$$

3. *For every n-ary predicate symbol* $\boldsymbol{p}$, *if* $\phi(\boldsymbol{p})$ *is a relation* $r \subseteq M^n$, *then* $\psi(\boldsymbol{p})$ *is the function* $g \in \mathcal{D}_{o\alpha_1\ldots\alpha_n}$, $\alpha_1 = \ldots = \alpha_n = \iota$, *such that, for every n-tuple* $(x_1 \ldots x_n) \in M^n$,

$$g(x_1)\ldots(x_n) = \begin{cases} \mathrm{T} \text{ if } (x_1 \ldots x_n) \in r \\ \mathrm{F} \text{ if } (x_1 \ldots x_n) \notin r \end{cases}$$

Lemma D.2 *Every* $\mathcal{V}$*-F.O. interpretation has a H.O. extension.*

PROOF. Let (M, ϕ) be a $\mathcal{V}$-F.O. interpretation, and let $\mathcal{D}$ be the frame generated by M. Since M is not empty, it has an element v. We define inductively a map $\alpha \mapsto p_\alpha$, with domain the set of all types, and such that $p_\alpha \in \mathcal{D}_\alpha$ for every type α, as follows:

1. $p_\iota = v$
2. $p_o = \mathrm{F}$
3. $p_{\alpha\beta} : \mathcal{D}_\beta \mapsto \mathcal{D}_\alpha$ is the constant function with value p_α.

Now we define an assignment ψ into $\mathcal{D}$ such that $(\mathcal{D}, \psi)$ is a H.O. extension of (M, ϕ), as follows:

1. ψ maps every symbol in $\mathcal{V}$ and every individual constant or variable to the value required by definition D.1.
2. ψ maps every description operator $\mathrm{the}_{\alpha(o\alpha)}$ to the function $f : \mathcal{D}_{o\alpha} \mapsto \mathcal{D}_\alpha$ which sends every function $g : \mathcal{D}_\alpha \mapsto \{\mathrm{F}, \mathrm{T}\}$ which takes the value T for a single element u of $\mathcal{D}_\alpha$ to $f(g) = u$, and every other function $h : \mathcal{D}_\alpha \mapsto \{\mathrm{F}, \mathrm{T}\}$ to $f(h) = p_\alpha$.
3. ψ maps every other logical constant to its intended denotation in $\mathcal{D}$.
4. ψ maps every other symbol, of type α, to p_α.

□

Lemma D.3 *If $\boldsymbol{S}$ is a $\mathcal{V}$-F.O. sentence, $\mathcal{I}$ a $\mathcal{V}$-F.O. interpretation, and $\mathcal{J}$ a H.O. extension of $\mathcal{I}$, then $\mathcal{I}$ satisfies $\boldsymbol{S}$ iff $\mathcal{J}$ does.*

PROOF. First, given $\mathcal{I} = (M, \phi)$ and $\mathcal{J} = (\mathcal{D}, \psi)$, we prove that every $\mathcal{V}$-F.O. term $\boldsymbol{T}$ has the same denotation in the two interpretations, by induction on $\boldsymbol{T}$.

1. If $\boldsymbol{T}$ is of the form "$\boldsymbol{s}$", where s is an individual constant or variable, then the two denotations of $\boldsymbol{T}$ are $\phi(s)$ and $\psi(s)$, which coincide, given that $\mathcal{J}$ is a H.O. extension of $\mathcal{I}$.

2. Let $\boldsymbol{T}$ be of the form "$\boldsymbol{f}\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$", where $\boldsymbol{f}$ is an n-ary function symbol and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ are $\mathcal{V}$-F.O. terms which, by induction hypothesis, have the same denotations $u_1 \ldots u_n$ under $\mathcal{I}$ and $\mathcal{J}$. Let $\phi(\boldsymbol{f})$ be $f : M^n \mapsto M$. Then, since $\mathcal{J}$ is a H.O. extension of $\mathcal{I}$, $\psi(\boldsymbol{f})$ is the function $g \in \mathcal{D}_{\iota\alpha_1 \ldots \alpha_n}$, $\alpha_1 = \ldots = \alpha_n = \iota$, such that, for every n-tuple $(x_1 \ldots x_n) \in M^n$,
$$g(x_1) \ldots (x_n) = f(x_1, \ldots, x_n).$$
Then, in particular
$$g(u_1) \ldots (u_n) = f(u_1, \ldots, u_n),$$
and $g(u_1) \ldots (u_n)$ is the H.O.-denotation of $\boldsymbol{T}$ in $\mathcal{J}$, while $f(u_1, \ldots, u_n)$ is the F.O.-denotation of $\boldsymbol{T}$ in $\mathcal{I}$.

Now, for every $\mathcal{V}$-F.O. sentence $\boldsymbol{S}$ we prove the following by induction on the definition of $\boldsymbol{S}$ as a $\mathcal{V}$-F.O. sentence: for every $\mathcal{V}$-F.O. interpretation $\mathcal{I}$ and every H.O. extension $\mathcal{J}$ of $\mathcal{I}$, $\mathcal{I}$ satisfies $\boldsymbol{S}$ iff $\mathcal{J}$ does.

1. Let $\boldsymbol{S}$ be of the form "$\boldsymbol{T} = \boldsymbol{T}'$", where $\boldsymbol{T}$ and $\boldsymbol{T}'$ are $\mathcal{V}$-F.O. terms. Let $\mathcal{I}$ be any $\mathcal{V}$-F.O. interpretation, and $\mathcal{J}$ any H.O. extension of $\mathcal{I}$. We know that $\boldsymbol{T}$ and $\boldsymbol{T}'$ have the same denotations u and u' in $\mathcal{I}$ as in $\mathcal{J}$. By definition, $\mathcal{I}$ satisfies $\boldsymbol{S}$ (in the F.O. sense) iff $u = u'$; but, as observed in section 2.3.4, since $\mathcal{J}$ is a logical interpretation, it satisfies $\boldsymbol{S}$ (in the H.O. sense) iff $u = u'$.

2. Let $\boldsymbol{S}$ be of the form $\boldsymbol{p}\,\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$, where $\boldsymbol{p}$ is an n-ary predicate and $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ are $\mathcal{V}$-F.O. terms. Let $\mathcal{I} = (M, \phi)$ be any $\mathcal{V}$-F.O. interpretation, and let $\mathcal{J} = (\mathcal{D}, \psi)$ be any H.O. extension of $\mathcal{I}$. We know that $\boldsymbol{T}_1 \ldots \boldsymbol{T}_n$ have the same denotations $u_1 \ldots u_n$ in $\mathcal{I}$ and $\mathcal{J}$. Let $\phi(\boldsymbol{p})$ be $r \in M^n$ and $\psi(\boldsymbol{p})$ be $g \in \mathcal{D}_{o\alpha_1 \ldots \alpha_n}$, $\alpha_1 = \ldots = \alpha_n = \iota$. $\mathcal{I}$ satisfies $\boldsymbol{S}$ iff $(u_1, \ldots, u_n) \in r$. But, by definition of the notion of extension, this is the case precisely when $g(u_1) \ldots (u_n) = \mathrm{T}$, i.e. when $\mathcal{J}$ satisfies $\boldsymbol{S}$.

3. Let $\boldsymbol{S}$ be of the form "$\boldsymbol{S}' \wedge \boldsymbol{S}''$", where $\boldsymbol{S}'$ and $\boldsymbol{S}''$ are $\mathcal{V}$-F.O. sentences. By induction hypothesis, for any $\mathcal{V}$-F.O. interpretation $\mathcal{I}$, and any H.O. extension $\mathcal{J}$ of $\mathcal{I}$, $\mathcal{I}$ satisfies $\boldsymbol{S}'$ iff $\mathcal{J}$ does, and the same holds for $\boldsymbol{S}''$. Let $\mathcal{I}$ and $\mathcal{J}$ by any such interpretations. By definition, $\mathcal{I}$ satisfies $\boldsymbol{S}$ iff it satisfies both $\boldsymbol{S}'$ and $\boldsymbol{S}''$. But, as observed in section 2.3.4, since $\mathcal{J}$ is a logical interpretation, it satisfies $\boldsymbol{S}$ (in the H.O. sense) iff it satisfies both $\boldsymbol{S}'$ and $\boldsymbol{S}''$. Therefore $\mathcal{I}$ satisfies $\boldsymbol{S}$ iff $\mathcal{J}$ does.

   The cases where $\boldsymbol{S}$ is of the form "$\bot$", "$\neg\boldsymbol{S}'$", "$\boldsymbol{S}' \vee \boldsymbol{S}''$" or "$\boldsymbol{S}' \supset \boldsymbol{S}''$" are treated similarly.

4. Let $\boldsymbol{S}$ be of the form "$\forall \boldsymbol{x}\boldsymbol{S}'$", where $\boldsymbol{x}$ is an individual variable and $\boldsymbol{S}'$ is a $\mathcal{V}$-F.O. sentence. Let $\mathcal{I} = (M, \phi)$ be a $\mathcal{V}$-F.O. interpretation, and let $\mathcal{J} = (\mathcal{D}, \psi)$ be a H.O. extension of $\mathcal{I}$. Let u be an element of M. Let ϕ' be the $\mathcal{V}$-F.O. assignment into M which maps $\boldsymbol{x}$ to u and otherwise coincides with ϕ, and let $\mathcal{I}'$ be the $\mathcal{V}$-F.O. interpretation (M, ϕ'). Let ψ' be the assignment into the frame $\mathcal{D}$ which maps $\boldsymbol{x}$ to u and otherwise coincides with ψ, and let $\mathcal{J}'$ be the H.O. interpretation (M, ψ'). Clearly, $\mathcal{J}'$ is a H.O. extension of $\mathcal{I}'$. Hence, by induction hypothesis, $\mathcal{I}'$ satisfies $\boldsymbol{S}'$ iff $\mathcal{J}'$ does.

   $\mathcal{I}$ satisfies $\boldsymbol{S}$ (in the F.O. sense) iff, for every $u \in M$, the F.O. interpretation $\mathcal{I}'$ constructed from u as described above satisfies $\boldsymbol{S}'$. $\mathcal{J}$satisfies $\boldsymbol{S}$ (in the H.O. sense) iff, for every $u \in M$, the H.O. interpretation $\mathcal{J}'$ constructed from u as described above satisfies $\boldsymbol{S}'$. Therefore $\mathcal{I}$ satisfies $\boldsymbol{S}$ in the F.O. sense iff $\mathcal{J}$ satisfies $\boldsymbol{S}$ in the H.O. sense.

   The case where $\boldsymbol{S}$ is of the form "$\exists \boldsymbol{x}\boldsymbol{S}'$" is treated similarly.

□

PROOF OF THEOREM 2.2. We already now that H.O.L. is an extension of F.O.L. Conversely, assume $\Gamma \vdash \boldsymbol{P}$, where Γ is a $\mathcal{V}$-F.O. theory and $\boldsymbol{P}$ is

a $\mathcal{V}$-F.O. sentence. Let $\mathcal{I}$ be a $\mathcal{V}$-F.O. interpretation which is a model of Γ. By lemma D.2, there exists a H.O. extension $\mathcal{J}$ of $\mathcal{I}$. By lemma D.3, $\mathcal{J}$ is a model of Γ (in the H.O. sense). By the assumption and by the soundness of H.O.L. (section 2.3.7), $\mathcal{J}$ satisfies $\boldsymbol{P}$, in the H.O. sense. By lemma D.3 again, $\mathcal{I}$ satisfies $\boldsymbol{P}$ in the F.O. sense. Thus every $\mathcal{V}$-F.O. interpretation which is a model of Γ satisfies $\boldsymbol{P}$, i.e. $\Gamma \models_{\text{F.O.}} \boldsymbol{P}$. By the completeness of F.O.L. (theorem D.1), $\Gamma \vdash_{\mathcal{V}\text{-F.O.}} \boldsymbol{P}$. □

Bibliography

[1] Peter B. Andrews. *A Transfinite Type Theory with Type Variables.* North-Holland Publishing Company, Amsterdam, 1965.

[2] Peter B. Andrews. General models, descriptions and choice in type theory. *Journal of Symbolic Logic*, 37(2):385–394, June 1972.

[3] Peter B. Andrews. General models and extensionality. *Journal of Symbolic Logic*, 37(2):395–397, June 1972.

[4] P. B. Andrews, D. A. Miller, E. L. Cohen, and F. Pfenning. Automating higher-order logic. *Contemporary Mathematics*, 29:169–192, 1984.

[5] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory.* Academic Press, 1986.

[6] P. B. Andrews, S. Issar, D. Nesmith, and F. Pfenning. The TPS theorem proving system. In *Proceedings of the Ninth International Conference on Automated Deduction, Argonne, Illinois, U.S.A.*, Springer-Verlag, 1988.

[7] H. P. Barendregt. *The Lambda Calculus.* North-Holland, 1984.

[8] Paul Bernays. *Axiomatic Set Theory.* North-Holland, Amsterdam, 1958.

[9] Nicolas Bourbaki. *Eléments de Mathématique, I: Théorie des Ensembles.* Diffusion C.C.L.S., Paris, 1970. English translation published by Addison-Wesley, 2nd ed. 1974.

[10] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[11] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[12] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, U.S.A., 1986.

[13] Francisco Corella. The double nature of type theory. Technical Report RC 15473, IBM Research, January 1990.

[14] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[15] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, chapter 15. North-Holland, 1989.

[17] Solomon Feferman. Set-theoretical foundations of category theory. In *Reports of the Midwest Category Seminar III.* Springer Verlag, 1969. Lecture Notes in Mathematics 106.

[18] Pierre Gabriel. Des catégories abéliennes. *Bulletin de la Société Mathématique de France*, 90:323–448, 1962.

[19] Kurt Gödel. *The consistency of the axiom of choice and the generalized continuum-hypothesis with the axioms of set theory.* Princeton University Press, 1940.

[20] Michael J. C. Gordon. *HOL, A Machine Oriented Formulation of Higher Order Logic.* Technical Report 68, University of Cambridge, Computer Laboratory, Cambridge CB2 3QG, England, 1985.

[21] Michael J. C. Gordon. *Rationale for Set Theory rather than Higher Order Logic.* 1985. Unpublished.

[22] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, North Holland, 1986.

also issued as University of Cambridge Computer Laboratory Technical Report No. 77, 1985.

[23] Michael J. C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[24] A. Grothendieck and J. L. Verdier. Exposé I: Préfaisceaux. In *Théorie des Topos et Cohomologie Etale des Schémas.* Springer-Verlag, 1972. Lecture Notes in Mathematics 269.

[25] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh workshop on VLSI*, pages 179–213, North-Holland, Amsterdam, 1986.

[26] F. K. Hanna and N. Daeche. Purely functional implementation of a logic. In *Proceedings of the Eighth International Conference on Automated Deduction, Oxford, U. K.*, Springer-Verlag, 1986.

[27] Leon Henkin. *The completeness of Formal Systems.* PhD thesis, Princeton University, October 1947.

[28] John M. J. Herbert. *Application of Formal Methods to Digital System Design.* PhD thesis, University of Cambridge, December 1986. Corpus Christi College.

[29] J. R. Hindley and J. P. Seldin. *Introduction to combinators and λ-calculus. London Mathematical Society Studen Texts 1*, Cambridge University Press, 1986.

[30] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. In *18th IEEE Symposium on the Foundations of Computer Science*, 1977.

[31] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[32] Peter T. Johnstone. *Notes on logic and set theory. Cambridge Mathematical Textbooks*, Cambridge University Press, 1987.

[33] Jussi Ketonen and Joseph S. Weening. *EKL—An Interactive Proof Checker, User's Reference Manual.* Technical Report STAN-CS-84-1006, Stanford University, Department of Computer Science, Stanford, CA 94305, 1984.

[34] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, 1952.

[35] Jan Willem Klop. Term rewriting systems, from Church-Rosser to Knuth-Bendix and beyond. In *Automata, Languages and Programming, 17th International Colloquium*, pages 250–269. Springer-Verlag, July 1990. Lecture Notes in Computer Science 443.

[36] Saunders Mac Lane. One universe as a foundation for category theory. In *Reports of the Midwest Category Seminar III.* Springer Verlag, 1969. Lecture Notes in Mathematics 106.

[37] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[38] Rainer Manthey and François Bry. Satchmo: a theorem prover implemented in Prolog. In *9th Conference on Automated Deduction*, LNCS Springer Verlag, 1988.

[39] David A. McAllester. *Ontic: A Knowledge Representation System for Mathematics.* PhD thesis, M.I.T., May 1987.

[40] Elliott Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, New York, 1964.

[41] L. G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4:445–462, 1988.

[42] P. A. J. Noel. *Experimenting with Isabelle in ZF Set Theory.* Technical Report 177, University of Cambridge, Computer Laboratory, Cambridge CB2 3QG, England, September 1989.

[43] Lawrence C. Paulson. *Natural Deduction Proof as Higher-Order Resolution.* Technical Report 82, University of Cambridge, Computer Laboratory, Cambridge CB2 3QG, England, December 1985.

[44] Lawrence C. Paulson. *The Foundation of a Generic Theorem Prover.* Technical Report 130, University of Cambridge, Computer Laboratory, Cambridge CB2 3QG, England, 1988.

[45] F. C. N. Pereira and H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[46] F. C. N. Pereira and H. D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.

[47] Dag Prawitz. Ideas and results in proof theory. In *Second Scandinavian Logic Symposium*, North Holland, Amsterdam, 1971.

[48] W. V. Quine. *Set Theory and Its Logic.* Harvard University Press, Cambridge, Massachusetts, U.S.A., 1967 edition, 1963.

[49] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the Association for Computing Machinery*, 20(1):160–187, January 1973.

[50] John Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[51] John Michael Spivey. *The Z Notation.* Prentice Hall, UK, 1989.

[52] Patrick C. Suppes. *Axiomatic Set Theory.* Van Nostrand, Princeton, N.J., 1960.

[53] Xubo Zhang. Overlap closures do not suffice for termination of term rewriting systems. *Information Processing Letters (Netherlands)*, 37(1), January 1991.