LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Workshop on General Logic

Edinburgh, February 1987

Editors: A. Avron
R. Harper
F. Honsell
I. Mason
G. Plotkin

Workshop on General Logic

ECS-LFCS-88-52
(also published as CSR-264-88)

LFCS Report Series

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

May 1988

CONTENTS

Forward

Workshop Timetable

Papers	Author	No.
A Framework for Defining Logics	R Harper	1
Hoare's Logic in the LF	I Mason	4
The Logical Theory of Constructions	P Aczel	5
Delta Lambda Calculus	N de Bruijn	6
Consequence Relations	A Avron	7
Using a Type Checker as a Proof Checker	T Coquand	8
Terminating General Recursion	B Nördstrom	10
The Metatheory of the LF Type System	F Honsell	11
A Search Space for LF	G Plotkin	12
Implementation of Substitution in the AUTOMATH Programme	L Jutting	14
Partial Inductive Definitions	L Hallnas	15
Representing Modal Logic in the LF	A Avron	16
The B Pre-logic	J R Abrial	17
On the Meaning and Construction of Martin-Löf's Theory of Types	R Backhouse	18
The independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes	J Smith	19
Inductively Defined Sets in Martin-Löf's Set Theory	P Dybjer	20

Foreword

This is a collection of papers and notes from the Workshop on General Logic held in Edinburgh in February 1987. The workshop was organised by the Laboratory for the Foundations of Computer Science of the University of Edinburgh. The Laboratory was founded in 1986 with significant financial support from the UK Alvey Directorate, the UK Science and Engineering Council and Industry. Its principal aim is to develop the foundational theory of Computer Science. This work is reflected in the development of systems which are rooted in sound theory. Through the Laboratory's Programme of Industrial Interaction these theories and systems are used and developed in an industrial environment. This gives Industry early access to leading edge technology in Software Engineering and the researchers in LFCS benefit from feedback on the usefulness and applicability of their work in practical environments.

The workshop started on Monday morning and lasted five days. It covered frameworks for defining varieties of logics, with an aim to developing a very general environment for proof checking, editing and discovery. Research along these lines by various people has been proceeding particularly strongly in the last year or so, and it was felt - as seems to have turned out the case - to be a particularly appropriate time to organise a workshop bringing together logicians and computer scientists. The workshop was organised with formal lectures in the morning, a brief lecture in the afternoon and then informal discussions and demonstrations of various computer systems. The systems demonstrated were :

1. The Edinburgh Logical Framework, demonstrated by T. Griffin of Cornell.

2. The System B, demonstrated by J.R. Abrial. Project sponsored by BP; joint project with PRG, Oxford. Carroll Morgan, Paul Gardiner, Mike Spivey.

3. The Centaur System, demonstrated by Laurent Hascoet of INRIA.

4. The I.P.E., demonstrated by Claire Jones of Edinburgh University.

We wish to thank George Cleland, Joan Ratcliff and Eleanor Kerse for their help with the organisation and their performance of the arduous task of ensuring the smooth day-to-day running of the workshop. We also particularly thank Robin Milner for hosting the Workshop party. Finally we thank the participants who were so kind and enthusiastic as to travel at a not particularly convenient time of year- sometimes at short notice - and contribute enthusiastically to the success and usefulness of the workshop. We also wish to thank the Science and Engineering Research Council who provided generous financial support.

Workshop on General Logic

23–27 February 1987

23 February			
09.15	R Harper	1	A Framework for Defining Logics
10.30	P Martin-Löf	2	The Logics of Judgements
11.30	T Griffin	3	An Implementation of an Environment for the Edinburgh LF
14.00	I Mason	4	Hoare's Logic in the LF
24 February			
09.00	P Aczel	5	The Logical Theory of Constructions
10.30	N de Bruijn	6	Delta Lambda Calculus
11.30	A Avron	7	Simple Consequence Relations
14.00	T Coquand	8	Using a Type Checker as a Proof Checker
25 February			
09.00	P Schroeder-Heister	9	Quantified Higher-level Rules and their Application to Logic Programming
10.30	B Nördstrom	10	Terminating General Recursion
11.30	F Honsell	11	The Metatheory of the LF Type System
14.00	G Plotkin	12	A Search Space for LF
26 February			
09.00	E Robinson	13	Categorical Models for the LF Type System
10.30	L Jutting	14	Implementation of Substitution in the AUTOMATH Programme
11.30	L Hallnas	15	Partial Inductive Definitions
14.00	A Avron	16	Representing Modal Logic in the LF
14.30	J R Abrial	17	The B pre-logic Two case studies using B: - a set theory - a generalised substitution calculus
27 February			
09.00	R Backhouse	18	On the Meaning and Construction of Martin-Löf's Theory of Types
10.30	J Smith	19	The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes
11.30	P Dybjer	20	Inductively Defined Sets in Martin-Löf's Set Theory

A Framework for Defining Logics*

Robert Harper Furio Honsell Gordon Plotkin
Laboratory for Foundations of Computer Science
University of Edinburgh

1

Abstract

The Logical Framework (LF) is a system for defining a wide class of logics. It is based on a general treatment of syntax, rules, and proofs in terms of a typed λ–calculus with dependent types. Syntax is treated in a style similar to, but more general than, Martin-Löf's system of arities. The treatment of rules and proofs focuses on the notion of a *judgement*. Logics are encoded in the LF via a new principle, the *judgements as types* principle, whereby each judgement is identified with the type of its proofs. This allows for a smooth treatment of discharge and variable occurrence conditions and leads to a uniform treatment of rules and proofs whereby rules are viewed as proofs of higher–order judgements and proof checking is reduced to type checking. An important benefit of our treatment of formal systems is that logic–independent tools such as proof editors and proof checkers can be constructed.

1 Introduction

Much work has been devoted to building systems for checking and building formal proofs in various logical systems. The AUTOMATH project of deBruijn [4] considered first proof checking. The problem of interactive proof construction was first considered by Milner, *et. al.* in the LCF system [11]. The LCF system was adapted to type theory by Petersson [19]. The work of Huet and Coquand on the Calculus of Constructions [7,8] extends the AUTOMATH and LCF work to a more powerful logic. Paulson's work on Isabelle [18] is a general approach to proof construction based on higher–order resolution. The NuPRL system of Constable [6] is a display–based interactive proof development environment for type theory that includes facilities for notation extension, library management, and automated proof search.

There are a great many logics of interest (equational, first–order, higher–order, temporal, type and set theories, type assignment systems and operational semantics for

*This is a slightly edited version of a paper delivered at the Second Symposium on Logic in Computer Science, Ithaca, NY, June, 1987. Any citations should refer to the proceedings of that conference.

programming languages). Implementing an interactive proof development environment for any style of presentation of any of these logics is a daunting task. For example, one must implement a parser, term manipulation operations (such as substitution and α–conversion), definitions and notation extension, inference rules, proofs, tactics, and tacticals. Thus it is desirable to find a general theory of inference systems that captures the uniformities of a large class of logics so that much of this effort can be expended once and for all.

The Logical Framework is an attempt to provide such a general theory of logics. It is based on a weak type theory that is closely related to AUT–PI and AUT–QE [9], to Martin-Löf's early type theory [14], to Huet and Coquand's Calculus of Constructions [8], and to Meyer and Reinhold's λ^π [16]. It is able to specify both the language of a logic, its axiom and rule schemes, and its proofs. The language of a logic is defined in a general theory of expressions that exploits the λ–calculus structure to provide binding operators, substitution, capture, α–conversion, and schematic abstraction and instantiation.

The treatment of rules and proofs is based on the notion of a *judgement* [15], the unit of assertion in a logical system. (See also Schroeder–Heister [22] for a related point of view.) Each logic is a system for asserting basic judgements. The set of judgements is then closed under two higher–order judgement forms that are used to specify inference rule schemes and to model discharge and variable occurrence conditions such as arise in Hilbert systems or systems of natural deduction. Rules are viewed as the proofs of (possibly higher–order) judgements that specify them. There is no distinction between primitive and derived rules. The extension to higher–order judgements allows for a natural presentation of many logical systems that avoids side conditions on rules. Judgements, rules, and proofs are represented in the LF type theory by applying what we call the *judgements as types* principle whereby each judgement is identified with the type of its proofs. Each basic judgement is represented by a base type of the LF type theory, and the higher–order judgements are represented in a logic–independent way by functional types. Proofs, and hence rules, are represented as terms of the LF type theory, thereby reducing proof checking to type checking.

In Section 2 we present the type theory of the LF, along with some of its important proof–theoretic properties. In Section 3, we introduce the LF's theory of expressions, and consider predicate calculus and Church's higher–order logic as examples. In Section 4 we consider the treatment of judgements, rules, and proofs in the LF. The method is illustrated for predicate calculus and higher–order logic. In Section 5 we compare our work with other systems for defining logics, and in Section 6 we suggest directions for future research.

We gratefully acknowledge the influence of Per Martin-Löf, particularly the lectures delivered in Edinburgh in the spring of 1986. We are grateful to Tim Griffin of Cornell University for his efforts in building a prototype implementation of the LF. We also thank the other members of the Computer Assisted Formal Reasoning project at Edinburgh, especially Arnon Avron and Ian Mason, for their comments and criticisms. Support for this research was provided by the Science and Engineering Research Council.

2 The LF Type Theory

The type theory of the LF is closely related to the Π–fragment of AUT–PI, a language belonging to the AUTOMATH family. The LF type theory is a language with entities of three levels: (1) objects, (2) types and families of types, and (3) kinds. Objects are classified by types, types and families of types by kinds. The kind Type classifies the types; the other kinds classify functions f which yield a type $f(x_1)\dots(x_n)$ when applied to objects $x_1,\dots,x_n$ of certain types determined by the kind of f. Any function definable in the system has a type as domain, while its range can either be a type, if it is an object, or a kind, if it is a family of types. The LF type theory is therefore predicative.

A number of different presentations of this system can be given. We shall describe a version which trades off conciseness against readability. The theory we shall deal with is a formal system for deriving assertions of one of the following shapes:

$\vdash \Sigma$ sig	Σ is a signature
$\vdash_\Sigma \Gamma$ context	Γ is a context
$\Gamma \vdash_\Sigma K$ kind	K is a kind
$\Gamma \vdash_\Sigma A : K$	A has kind K
$\Gamma \vdash_\Sigma M : A$	M has type A

where the syntax is specified by the following grammar:

$$\begin{array}{llll}
\textit{Kinds} & K & ::= & \mathrm{Type} \mid \Pi x{:}\,A.K \\
\textit{Type Families} & A & ::= & c \mid \Pi x{:}\,A.B \mid \lambda x{:}\,A.B \mid AM \\
\textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}\,A.M \mid MN \\
\textit{Signatures} & \Sigma & ::= & \langle\rangle \mid \Sigma, c{:}\,K \mid \Sigma, c{:}\,A \\
\textit{Contexts} & \Gamma & ::= & \langle\rangle \mid \Gamma, x{:}\,A
\end{array}$$

We let M and N range over expressions for objects, A and B for types and families of types, K for kinds, x and y over variables, and c over constants. We write $A \to B$ for $\Pi x{:}\,A.B$ when x does not occur free in B.

The inference rules of the LF type theory appear in Table 1. A term is said to be *well–typed in a signature and context* if it can be shown to either be a kind, have a kind, or have a type in that signature and context. A term is *well–typed* if it is well–typed in some signature and context. The notion of $\beta\eta$–contraction, written $\to_{\beta\eta}$, can be defined both at the level of objects and at the level of types and families of types in the obvious way. Rules (12) and (17) make use of a relation $=_{\beta\eta}$ between terms which is defined in the following way: $M =_{\beta\eta} N$ iff $M \to^*_{\beta\eta} P$ and $N \to^*_{\beta\eta} P$ for some term P. We conjecture that variants of rules (12) and (17) obtained by taking $=_{\beta\eta}$ to be $\beta\eta$–conversion are admissible rules of the theory.

Since the notion of $\beta\eta$–conversion over $K \cup A \cup M$ is not Church–Rosser, the order of technical priority in which the basic metatheoretical results are proved is essential. The following theorem summarizes these results in a convenient order (here α ranges over the basic assertions of the type theory):

Valid Signature

$$\frac{}{\vdash \langle\rangle\ \mathbf{sig}} \tag{1}$$

$$\frac{\vdash \Sigma\ \mathbf{sig} \quad \vdash_\Sigma K\ \mathbf{kind} \quad c \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, c{:}\,K\ \mathbf{sig}} \tag{2}$$

$$\frac{\vdash \Sigma\ \mathbf{sig} \quad \vdash_\Sigma A : \mathrm{Type} \quad c \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, c{:}\,A\ \mathbf{sig}} \tag{3}$$

Valid Context

$$\frac{\vdash \Sigma\ \mathbf{sig}}{\vdash_\Sigma \langle\rangle\ \mathbf{context}} \tag{4}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathbf{context} \quad \Gamma \vdash_\Sigma A : \mathrm{Type} \quad x \notin \mathrm{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\,A\ \mathbf{context}} \tag{5}$$

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma\ \mathbf{context}}{\Gamma \vdash_\Sigma \mathrm{Type}\ \mathbf{kind}} \tag{6}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathrm{Type} \quad \Gamma, x{:}\,A \vdash_\Sigma K\ \mathbf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}\,A.K\ \mathbf{kind}} \tag{7}$$

Valid Elements of a Kind

$$\frac{\vdash_\Sigma \Gamma\ \mathbf{context} \quad c{:}\,K \in \Sigma}{\Gamma \vdash_\Sigma c : K} \tag{8}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathrm{Type} \quad \Gamma, x{:}\,A \vdash_\Sigma B : \mathrm{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\,A.B : \mathrm{Type}} \tag{9}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathrm{Type} \quad \Gamma, x{:}\,A \vdash_\Sigma B : K}{\Gamma \vdash_\Sigma \lambda x{:}\,A.B : \Pi x{:}\,A.K} \tag{10}$$

$$\frac{\Gamma \vdash_\Sigma B : \Pi x{:}\,A.K \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma BN : [N/x]K} \tag{11}$$

$$\frac{\Gamma \vdash_\Sigma A : K \quad \Gamma \vdash_\Sigma K'\ \mathbf{kind} \quad K =_{\beta\eta} K'}{\Gamma \vdash_\Sigma A : K'} \tag{12}$$

Valid Elements of a Type

$$\frac{\vdash_\Sigma \Gamma\ \mathbf{context} \quad c{:}\,A \in \Sigma}{\Gamma \vdash_\Sigma c : A} \tag{13}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathbf{context} \quad x{:}\,A \in \Gamma}{\Gamma \vdash_\Sigma x : A} \tag{14}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathrm{Type} \quad \Gamma, x{:}\,A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}\,A.M : \Pi x{:}\,A.B} \tag{15}$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}\,A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : [N/x]B} \tag{16}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A' : \mathrm{Type} \quad A =_{\beta\eta} A'}{\Gamma \vdash_\Sigma M : A'} \tag{17}$$

Table 1: The LF Type System

Theorem 2.1

1. *Thinning is an admissible rule: if* $\Gamma \vdash_\Sigma \alpha$ *and* $\vdash_{\Sigma,\Sigma'} \Gamma, \Gamma'$ context, *then* $\Gamma, \Gamma' \vdash_{\Sigma,\Sigma'} \alpha$.
2. *Transitivity is an admissible rule: if* $\Gamma \vdash_\Sigma M : A$ *and* $\Gamma, x{:}A, \Delta \vdash_\Sigma \alpha$, *then* $\Gamma, [M/x]\Delta \vdash_\Sigma [M/x]\alpha$.
3. *Uniqueness of types and kinds: if* $\Gamma \vdash_\Sigma M : A$ *and* $\Gamma \vdash_\Sigma M : A'$, *then* $A =_{\beta\eta} A'$, *and similarly for kinds.*
4. *Subject reduction: if* $\Gamma \vdash_\Sigma M : A$ *and* $M \rightarrow^*_{\beta\eta} M'$, *then* $\Gamma \vdash_\Sigma M' : A$, *and similarly for types.*
5. *All well–typed terms are strongly normalizing.*
6. *All well–typed terms are Church–Rosser.*
7. *Each of the five relations defined by the inference system of Table 1 is decidable, as is the property of being well–typed.*

The proof of this theorem is surprisingly complicated. However, the methods developed by van Daalen in his thesis [9] can be adapted to this type theory. It is important to note that the Church–Rosser property, strong normalization, and the presence of type labels are essential in the proof of decidability of the type theory. In particular, given a signature and a context and any object (type) expression, it is decidable if the object (type) expression is well–typed; if so, a type (kind) can be computed for it.

We shall outline only the proof of strong normalization, since, unlike the systems dealt with by van Daalen, it can be proved independently of the other metatheoretic results. Moreover, it yields a corollary that is useful for characterizing the terms that are definable in the LF type theory:

Corollary 2.2 *If* $\Gamma \vdash_\Sigma M : A$, *then* Erase($M$) *can be typed in Curry's type assignment system, where* Erase(M) *denotes the term obtained from* M *by removing the type labels from the* λ *abstractions.*

We start by defining a translation τ of the LF types and kinds into simple types with a base type α, and a translation $\tilde{}$ of LF objects and types into the untyped λ terms with a constant π. These translations are extended to signatures and contexts in the obvious way.

Definition 2.3

$$
\begin{array}{lcl}
\tau(\text{Type}) & = & \alpha \\
\tau(\Pi x{:}A.K) & = & \tau(A) \to \tau(K) \\
\tau(c) & = & \alpha \\
\tau(\lambda x{:}A.B) & = & \tau(B) \\
\tau(AM) & = & \tau(A) \\
\tau(\Pi x{:}A.B) & = & \tau(A) \to \tau(B) \\
\tilde{c} & = & c \\
\tilde{x} & = & x \\
\widetilde{AM} & = & \tilde{A}\tilde{M} \\
\widetilde{MN} & = & \tilde{M}\tilde{N} \\
\widetilde{\Pi x{:}A.B} & = & \pi\tilde{A}\tilde{B} \\
\widetilde{\lambda x{:}A.M} & = & (\lambda y.\lambda x.\tilde{M})\tilde{A} \quad (y \notin \mathrm{FV}(\lambda x.\tilde{M})) \\
\widetilde{\lambda x{:}A.B} & = & (\lambda y.\lambda x.\tilde{B})\tilde{A} \quad (y \notin \mathrm{FV}(\lambda x.\tilde{B}))
\end{array}
$$

The precise sense in which this definition is consistent is stated in the following lemma:

Lemma 2.4

1. *If* $\Gamma \vdash_\Sigma A : K$*, then* $\tau(\Gamma) \vdash_{\tau(\Sigma)} \tilde{A} : \tau(K)$*;*
2. *If* $\Gamma \vdash_\Sigma M : A$*, then* $\tau(\Gamma) \vdash_{\tau(\Sigma),\pi:\alpha\to\alpha\to\alpha} \tilde{M} : \tau(A)$.

Proof. *By induction on the structure of the proof of* $\Gamma \vdash_\Sigma A : K$ *and* $\Gamma \vdash_\Sigma M : A$*. We have only to notice that any derivation of* $\vdash_\Sigma \Gamma, x : A$ context *contains as a subderivation a derivation of* $\Gamma \vdash_\Sigma A : \text{Type}$. □

The translation has been carried out in such a way that the extra combinatorial complexity in the LF terms due to the presence of type labels is not lost. We then have the following result:

Theorem 2.5

1. *If* $A \to_{\beta\eta} A'$*, then* $\tilde{A} \to^{*}_{\beta\eta} \tilde{A'}$*;*
2. *If* $M \to_{\beta\eta} M'$*, then* $\tilde{M} \to^{*}_{\beta\eta} \tilde{M'}$.

Proof. *By induction on the derivation of* $A \to A'$ *and* $M \to M'$. □

Now since the Curry typable terms are strongly normalizing, so too are the terms of the LF. Moreover, it can be easily seen that $\tilde{A} \to^{*}_{\beta\eta}$ Erase(A), and $\tilde{M} \to^{*}_{\beta\eta}$ Erase(M).

A few remarks about the choice of the type theory are in order. It is essential that the type theory be decidable, at least theoretically, for, as we shall see below, proof checking is reduced to type checking. The use of conversion as the only counterpart of definitional equality is due to the fact that at this stage the LF does not deal with the issue of proof reduction and equality in the sense of Prawitz. We use η–conversion in order to make the encoding of syntax more transparent.

3 Theory of Expressions

The approach to formalizing the syntax of a language is inspired by Church [5] and Martin-Lõf system of arities [17]. Each syntactic category of the logic is represented by a type. The expressions of each category are built up using expression–forming constructors, which are formalized by suitable constants of the LF. Variable binding operators are represented by constants whose domain is a function type, so that binding is handled by the λ–calculus of the LF. The type theory of the LF being richer than simple type theory, our approach to syntax is more general than Martin-Lõf's. Explicit use of this extra generality is made in the formalization of Church's higher–order logic.

To illustrate the formalization of syntax within the LF, we consider two examples: the language of Peano arithmetic (as defined in Schoenfield [21]) and the language of higher–order logic [5]. The presentation of the syntax of Peano arithmetic will form a part of its signature, Σ_{PA}, and similarly for the signature of higher–order logic, Σ_{HOL}.

In a first–order language there are two syntactic categories: the *individual expressions*, which stand for individuals (objects in the domain of quantification), and the *formulas*, which stand for propositions. These are represented in the LF by the type ι of individuals, and the type o of propositions. Thus Σ_{PA} begins with:

$$\begin{array}{lcl} \iota & : & \mathrm{Type} \\ o & : & \mathrm{Type} \end{array}$$

The individual expression constructors of Peano arithmetic are formalized in Σ_{PA} by:

$$\begin{array}{lcl} 0 & : & \iota \\ \mathrm{succ} & : & \iota \rightarrow \iota \\ + & : & \iota \rightarrow \iota \rightarrow \iota \\ \times & : & \iota \rightarrow \iota \rightarrow \iota \end{array}$$

Terms of type ι in Σ_{PA} represent the individual expressions of Peano arithmetic. There are no declarations for the variables of first–order logic. The variables of the object language are identified with the variables of the LF, so that an open term of type ι in Σ_{PA}, all of whose free variables are of type ι, represents an open individual expression. For example, in a context containing $x{:}\,\iota$, the term $\mathrm{succ}(x)$ has type ι as well. This representation is defined compositionally by: $x^\circ = x$, $0^\circ = 0$, $\mathrm{succ}(t)^\circ = \mathrm{succ}(t^\circ)$, $t + u^\circ = +(t^\circ)(u^\circ)$, and $t \times u^\circ = \times(t^\circ)(u^\circ)$,

Theorem 3.1 (Adequacy for Syntax, I) *The correspondence $^\circ$ is a bijection between the expressions of Peano arithmetic and the normal forms of type ι in Σ_{PA} with all free variables of type ι.*

Proof sketch. *The translation is evidently well–defined and one–one. Surjectivity is proved by induction on the structure of the normal forms.* □

The following are the other constant declarations for the formulas of Peano arithmetic:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
= & : & \iota \to \iota \to o & < & : & \iota \to \iota \to o \\
\neg & : & o \to o & \wedge & : & o \to o \to o \\
\vee & : & o \to o \to o & \supset & : & o \to o \to o \\
\forall & : & (\iota \to o) \to o & \exists & : & (\iota \to o) \to o
\end{array}$$

The formula $\psi = \forall x.\phi[x]$ is represented by the term $\psi^\circ = \forall(\lambda x{:}\,\iota.\phi^\circ)$. This approach allows α–conversion and capture–avoiding substitution to be factored out of the definition of each individual logic, leaving it to be implemented once and for all by the framework. This treatment of binding operators relies on the variables of the first–order language being identified with the variables of the LF type theory. For example, if x is a variable of type ι, then $x = x$ is a term of type o.[1] We can bind x by λ–abstraction, obtaining $\lambda x{:}\,\iota.x = x$, and universally quantify it by applying it to the constant $\forall$, obtaining $\forall(\lambda x{:}\,\iota.x = x)$, which represents the first–order formula $\forall x.x = x$.

In this way, each formula ϕ of Peano arithmetic is represented by a term ϕ° of type o in Σ_{PA}, all of whose free variables are of type ι; sentences are represented by the closed terms of type o. An open term M of type o is an *incomplete formula.* Its λ–abstraction is a *formula scheme.* For example, the formula scheme

$$M = \lambda\phi{:}\,o.\lambda\Phi{:}\,\iota \to o.\phi \supset \exists(\Phi)$$

can be instantiated by application to a formula ϕ and a matrix Φ, so that

$$M(\forall(\lambda x{:}\,\iota.x = x))(\lambda x{:}\,\iota.x = x)$$

represents the first–order formula

$$\forall x.x = x \supset \exists x.x = x.$$

Theorem 3.2 (Adequacy for Syntax, II) *The compositional translation* $^\circ$ *is a bijection between the formulas of Peano arithmetic and the long $\beta\eta$ normal forms of type o in Σ_{PA} with all free variables of type ι.*
Proof sketch. *Similar to Theorem 3.1.* □

The role of η–conversion in the above proof is mainly to ensure that the well–typed terms of type o in Σ_{PA} are exactly the formulas of Peano arithmetic, up to the notion of definitional equality built in to the system. There is no intrinsic difference between $\forall(<(0))$ and $\forall(\lambda x{:}\,\iota. <(0)(x))$.

The formalization of the syntax of higher–order logic makes use of the dependent function type of the LF. Quantification in higher–order logic is over a type drawn from the hierarchy of simple functional types with two base types ι (of individuals) and o (of propositions). In order to avoid confusion with the types of the LF, we call the types of

[1] We freely use infix and postfix application in accordance with custom and readability considerations.

higher–order logic "sorts," and we shall write $\sigma \Rightarrow \tau$ for the sort of functions from sort σ to sort τ. In the formalization of higher–order logic the collection of sorts is represented as a type with members ι and o, closed under $\Rightarrow$. The signature Σ_{HOL} thus begins as follows:

$$\begin{array}{lcl}
\mathsf{sorts} & : & \mathsf{Type} \\
\iota & : & \mathsf{sorts} \\
o & : & \mathsf{sorts} \\
\Rightarrow & : & \mathsf{sorts} \rightarrow \mathsf{sorts} \rightarrow \mathsf{sorts}
\end{array}$$

To each sort is associated the type of objects of that sort. The objects of sort ι are, for the present purposes, the natural numbers. The objects of sort o are the propositions of higher–order logic. The quantifiers range over an arbitrary sort, rather than the fixed sort of individuals as in first–order logic. The objects of functional sort are given by typed λ terms (which we write with a capital λ to avoid confusion), and there is a form for expressing application.

$$\begin{array}{lcl}
\mathsf{obj} & : & \mathsf{sorts} \rightarrow \mathsf{Type} \\
0 & : & \mathsf{obj}(\iota) \\
\mathsf{succ} & : & \mathsf{obj}(\iota \Rightarrow \iota) \\
= & : & \Pi\sigma{:}\,\mathsf{sorts}.\,\mathsf{obj}(\sigma \Rightarrow \sigma \Rightarrow o) \\
\neg & : & \mathsf{obj}(o \Rightarrow o) \\
\wedge & : & \mathsf{obj}(o \Rightarrow o \Rightarrow o) \\
\vee & : & \mathsf{obj}(o \Rightarrow o \Rightarrow o) \\
\supset & : & \mathsf{obj}(o \Rightarrow o \Rightarrow o) \\
\forall & : & \Pi\sigma{:}\,\mathsf{sorts}.\,\mathsf{obj}((\sigma \Rightarrow o) \Rightarrow o) \\
\exists & : & \Pi\sigma{:}\,\mathsf{sorts}.\,\mathsf{obj}((\sigma \Rightarrow o) \Rightarrow o) \\
\mathbf{\Lambda} & : & \Pi\sigma{:}\,\mathsf{sorts}.\Pi\tau{:}\,\mathsf{sorts}.(\mathsf{obj}(\sigma) \rightarrow \mathsf{obj}(\tau)) \rightarrow \mathsf{obj}(\sigma \Rightarrow \tau) \\
\mathsf{ap} & : & \Pi\sigma{:}\,\mathsf{sorts}.\Pi\tau{:}\,\mathsf{sorts}.\,\mathsf{obj}(\sigma \Rightarrow \tau) \rightarrow \mathsf{obj}(\sigma) \rightarrow \mathsf{obj}(\tau)
\end{array}$$

The representation of equality and the quantifiers makes use of the dependent function types of the LF. For each sort σ, the equality relation for objects of sort σ is written $=_\sigma$; it is an object of sort $\sigma \Rightarrow \sigma \Rightarrow o$. Similarly, the quantifiers ranging over sort σ are written $\forall_\sigma$ and $\exists_\sigma$; they are objects of sort $(\sigma \Rightarrow o) \Rightarrow o$, just as in Church's formulation. The $\mathbf{\Lambda}$ and ap forms must similarly be tagged with types, which we write as subscripts. The $\mathbf{\Lambda}$ form must be tagged with both the domain and range types, unlike in Church's definition. The difference is minor, and analogs of Theorems 3.1 and 3.2 can be proved.

4 Theory of Rules and Proofs

The treatment of inference rules and proofs lies at the heart of the LF. The approach is organized around the notion of a *judgement* [15], the unit of inference of a logic. Each logic comes with a set of *basic* judgements. In first–order logic there is only one form of basic judgement, the assertion ϕ true that a formula ϕ is (logically) true (usually written as $\vdash \phi$ or just ϕ). Sequent calculi also have one basic judgement, written $\Gamma \Rightarrow \Delta$, the assertion that some formula in Δ is a logical consequence of all the formulas in Γ. In Martin-Lõf's system of type theory, there are four basic judgements, A type, $A = B$, $a \in A$, and $a = b \in A$.

In traditional logical systems the inference rules determine the set of proofs of basic judgements, and thereby also determine the set of "correct" or "evident" [15] basic judgements, namely those that have proofs. There are several approaches to the definition of a proof in a formal system. [21,20] Proofs are sometimes viewed as sequences of formulas that satisfy the condition that each formula is obtained from previous formulas by application of a rule. Another view is that proofs are trees satisfying certain conditions. In any case the *notion* of a proof is independent of the particular rules of inference.

We extend the notion of proof to include our view of rules as proofs of *higher–order judgements*. There are two forms: the *hypothetical* and the *schematic* (or *general*). The hypothetical judgement $J_1 \vdash J_2$ is the assertion that J_2 is a logical consequence of J_1, according to the rules of the logic. It is proved by a function mapping proofs of J_1 to proofs of J_2. The schematic judgement $\bigwedge_{x:A} J(x)$ represents the idea of generality: the judgement $J(x)$ is evident for any object x of type A. It is proved by a function mapping objects x of type A to proofs of $J(x)$.

Rules and proofs are represented as terms of the LF type theory. The basic rules are presented as constants in the signature of the logic, and the derived rules are complex proofs that are λ–abstracted with respect to their premises. Since rules are functions, complex proofs are built by applying (in the sense of the λ–calculus) rules to the proof(s) of their premise(s). Rule schemes are represented as proofs of schematic judgements. Schematic variables are identified with the variables of the LF, so that schematization is achieved by λ abstraction, and schematic instantiation by application.

If rules and proofs are terms, what are to be their types? Since a proof is viewed as evidence for a judgement, it seems natural to identify judgements with the type of their proofs: a judgement is evident iff it has a proof iff there is a term of that type (in the signature of the logic). We call this the *judgements as types* principle, by analogy with the propositions as types principle of Curry, deBruijn, and Howard. Here we are making no commitment to the semantics of a logic. Instead we are merely formalizing the idea that to make an assertion in a logical system, one must have a proof of it.

The type of proofs of a basic judgement is determined by the inference rules of the logic. The types of proofs of the higher–order judgement forms are defined by the LF. We define $J_1 \vdash J_2$ to be $J_1 \rightarrow J_2$, the type of functions mapping J_1 to J_2. This definition is motivated by the meaning of the hypothetical judgement and the judgements as types principle. Similarly, we define $\bigwedge_{x:A} J(x)$ to be $\Pi x{:}\, A.J(x)$, the type of functions mapping objects x of type A to $J(x)$. We write $J_1, \ldots, J_m \vdash_{x_1:A_1,\ldots,x_n:A_n} J$ for $\bigwedge_{x_1:A_1} \cdots \bigwedge_{x_n:A_n} J_1 \vdash \cdots (J_m \vdash J)$. This incorporates and generalizes Martin-Lõf's hypothetico–general judgements [15].

We take incomplete proofs to be open terms of judgement type J. They can be

$$(\neg\neg\mathrm{E})\quad \frac{\neg\neg\phi}{\phi} \qquad\qquad (\supset\mathrm{I})\quad \frac{\begin{array}{c}(\phi)\\ \psi\end{array}}{\phi \supset \psi}$$

$$(\forall\mathrm{I}^*)\quad \frac{\phi[x]}{\forall x.\phi[x]} \qquad\qquad (\forall\mathrm{E})\quad \frac{\forall x.\phi[x]}{\phi[t/x]}$$

*(*x not free in any assumption on which ϕ depends.)*

$$(\exists\mathrm{E}^*)\quad \frac{\exists x.\phi[x] \quad \begin{array}{c}(\phi)\\ \psi\end{array}}{\psi}$$

*(*x not free in ψ or any assumptions on which ψ depends.)*

$$(\mathrm{IND}^*)\quad \frac{\phi(0) \quad \begin{array}{c}(\phi(x))\\ \phi(\mathrm{succ}(x))\end{array}}{\phi(x)}$$

*(*x not free in any assumption, other than $\phi(x)$, in which $\phi(\mathrm{succ}(x))$ depends.)*

Table 2: Some Rules of Peano Arithmetic

completed by substitution or by λ–abstraction, yielding proofs of schematic judgements. Abstraction on judgement type variables not occurring in J yields proofs of hypothetical judgements.

An important consequence of the judgements as types principle is that we are able to reduce proof checking to type checking. A term M is a proof of a judgement J iff M has type J in the signature of the logic. This reduction is the most important reason for insisting that the type theory of the LF be decidable, for otherwise one could not construct a mechanical proof checker for a logic.

To illustrate these ideas, we formalize an illustrative selection of rules from first–order and higher–order logic formalized as systems of natural deduction. Returning to Σ_{PA}, we represent the single judgement form ϕ true by introducing a family of types indexed by the propositions:

$$\mathrm{true} \;:\; o \rightarrow \mathrm{Type}$$

We write "ϕ true" for "true(ϕ)." For any formula ϕ (*i.e.*, any term of type o in Σ_{PA}), the type ϕ true is the type of proofs of ϕ.

Each rule in Table 2 is represented by a constant whose type is the specification of the rule, a higher–order judgement. For instance, the double negation elimination rule is given by:

$$\neg\neg\text{E} \quad : \quad \neg\neg\phi \text{ true} \vdash_{\phi:o} \phi \text{ true}$$

The judgement is schematic in propositions ϕ and hypothetical in proofs of $\neg\neg\phi$, so if ϕ is a formula and M is a proof of $\neg\neg\phi$ true, then $\neg\neg\text{E}(\phi)(M)$ is a proof of ϕ true.

The implication introduction rule makes use of the hypothetical judgement form to model discharge. The formulation of $\supset$I in Table 2 takes a *hypothetical proof of* ϕ as premise, and discharges the hypothesis. We instead treat $\supset$I as taking a *proof of a hypothetical judgement*. The general intention is that a sufficient condition for establishing the truth of $\phi \supset \psi$ is to establish that ψ is a logical consequence of ϕ. The formalization of $\supset$I, which is schematic in ϕ and ψ, is:

$$\supset\text{I} \quad : \quad (\phi \text{ true} \vdash \psi \text{ true}) \vdash_{\phi:o,\psi:o} \phi \supset \psi \text{ true}$$

So, for example, $\supset\text{I}(\phi)(\phi)(\lambda x{:}\,\phi \text{ true}.x)$ is a proof of $\phi \supset \phi$ true.

Universal elimination is given by:

$$\forall\text{E} \quad : \quad \forall(\Phi) \text{ true} \vdash_{\Phi:\iota\rightarrow o,a:\iota} \Phi(a) \text{ true}$$

The rule is schematic in Φ, the matrix of the universally quantified formula, and a, the instance. Given such and a proof M of $\forall(\Phi)$ true, $\forall\text{E}(\Phi)(a)(M)$ is a proof of $\Phi(a)$ true. Substitution is modelled by applying the matrix to the instance.

Universal introduction is formalized like implication introduction. A condition for the truth of $\forall(\Phi)$ is that $\Phi(x)$ is true for arbitrary x. In Table 2 variable occurrence conditions are used for a *schematic proof of the judgement* $\Phi(x)$. We instead use a *proof of a schematic judgement* $\bigwedge_x{:}\,\iota.\Phi(x)$. The rule is given by:

$$\forall\text{I} \quad : \quad (\bigwedge_{x:\iota} \Phi(x) \text{ true}) \vdash_{\Phi:\iota\rightarrow o} \forall(\Phi) \text{ true}$$

Existential elimination uses both hypothetical and schematic judgements, and makes use of scoping to avoid side conditions:

$$\exists\text{E} \quad : \quad \exists(\Phi) \text{ true}, (\Phi(x) \text{ true} \vdash_{x:\iota} \psi \text{ true}) \vdash_{\Phi:\iota\rightarrow o,\psi:o} \psi \text{ true}$$

Since ψ is bound outermost, there is no possibility that the x of the schematic judgement form occur free in an instance.

Induction makes use of scoping and higher-order judgements:

$$\text{IND} \quad : \quad \Phi(0) \text{ true}, (\Phi(x) \text{ true} \vdash_{x:\iota} \Phi(\text{succ}(x)) \text{ true}) \vdash_{\Phi:\iota\rightarrow o,x:\iota} \Phi(x) \text{ true}$$

The correctness of the formalization is expressed by the following theorem:

Theorem 4.1 (Adequacy for Theorems) *There is a (compositionally-defined) bijection between first-order natural deduction proofs of a formula ϕ of Peano arithmetic from assumptions $\phi_1, \ldots, \phi_n$ and normal forms M of type ϕ°* true *in Σ_{PA}, all of whose free variables are of type ι and ϕ_i°* true *$(1 \leq i \leq n)$.*

Proof. *It is straightforward to prove by induction on the length of derivations, that if* $A_1, \ldots, A_n \vdash_{\mathrm{PA}} A$ *is derivable, then*

$$\Gamma, x_1{:}\, A_1 \text{ true}, \ldots, x_n{:}\, A_n \text{ true} \vdash_{\Sigma_{\mathrm{PA}}} M : A_{n+1} \text{ true}$$

is derivable, where Γ *contains assignments of the form* $x{:}\,\iota$ *for the free object variables* x *occurring in the* A_i*'s and in* M*, and where* M *faithfully encodes instantiation and application of rules. Surjectivity can be proved by induction on the structure of the normal forms of type* ϕ true *(for* $\phi{:}\, o$*), keeping in mind the uniqueness of types and the Church–Rosser property.* □

Note that it is possible for M in the above proof to have free variables of type ι, even when $n = 0$ (*i.e.*, when there are no assumptions) and when ϕ has no free variables. This is true, for example, in a proof of

$$\forall x.\phi \supset \exists x.\phi(x).$$

It is a peculiarity of first–order logic that the assumption that the domain of quantification is non–empty is not made explicit in proofs.

The above proof illustrates the fact that judgements in the LF actually encode consequence relations that satisfy, in view of Theorem 2.1, weak forms of thinning, transitivity, and contraction.

We shall give two examples. In the first we present a proof of $\phi \supset (\psi \supset \phi)$ true as a well–typed term in the signature Σ_{PA}. Let x have type ϕ true, and let y have type ψ true. Abstracting the incomplete proof x with respect to y, we obtain a proof $\lambda y{:}\, \psi \text{ true}.x$ of ψ true $\vdash$ ϕ true. Applying $\supset$I to this proof, we obtain the (incomplete) proof $\supset\mathrm{I}(\psi)(\phi)(\lambda y{:}\, \psi \text{ true}.x)$ of $\psi \supset \phi$ true. Abstracting with respect to x, and applying $\supset$I again, we obtain the complete proof

$$\supset\mathrm{I}(\phi)(\psi \supset \phi)(\lambda x{:}\, \phi \text{ true}.\supset\mathrm{I}(\psi)(\phi)(\lambda y{:}\, \psi \text{ true}.x))$$

of $\phi \supset (\psi \supset \phi)$ true.

In the second example we give evidence for the claim that the traditional notion of a derivable rule has a formal counterpart in the LF. We show that in the signature Σ_{PA} the elimination rule for the universal quantifier in Schroeder–Heister's style can be derived. The Schroeder–Heister rule is specified as follows:

$$\forall\mathrm{E}_{SH} \quad : \quad \forall(\Phi) \text{ true}, ((\textstyle\bigwedge_{x:\iota} \Phi(x) \text{ true}) \vdash \psi \text{ true}) \text{ true} \vdash \psi \text{ true}) \vdash_{\Phi:\iota\to o, \psi:o} \psi \text{ true}.$$

It can be easily verified that the term

$$\lambda\Phi{:}\, \iota \to o.\lambda\psi{:}\, o.\lambda p{:}\, \forall(\Phi) \text{ true}.\lambda q{:}\, ((\bigwedge_{x:\iota} \Phi(x) \text{ true}) \vdash \psi \text{ true}).q(\lambda x{:}\, \iota.\forall\mathrm{E}(\Phi)(x)(p))$$

has the above type.

With regard to derived rules, it is interesting to point out that in view of the fact that thinning is an admissible rule of the LF type theory, judgements are "open" concepts. This precludes an induction principle on proofs. Therefore typical admissible rules for a given logic $\mathcal{L}$, or meta rules such as the deduction theorem for a Hilbert–style presentation of first–order logic, are not directly derivable in certain adequate signatures for $\mathcal{L}$.

Turning to the formalization of higher–order logic, we formalize the inference rules in a manner quite similar to that of first–order logic. There is one judgement form, the assertion that ϕ is true, for ϕ an object of sort o.

$$\mathrm{true} \;:\; \mathrm{obj}(o) \rightarrow \mathrm{Type}$$

The rules of β and η–conversion for the λ–calculus appear as axioms about equality. They are schematic in the domain and range sorts of the functions, and in the terms themselves:

$$\begin{aligned}\beta \;&:\; \textstyle\bigwedge_{\sigma:\mathrm{sorts},\tau:\mathrm{sorts},f:\mathrm{obj}(\sigma)\rightarrow\mathrm{obj}(\tau),a:\mathrm{obj}(\sigma)} \mathrm{ap}_{\sigma,\tau}(\Lambda_{\sigma,\tau}(f),a) =_{\tau} f(a)\ \mathrm{true}\\ \eta \;&:\; \textstyle\bigwedge_{\sigma:\mathrm{sorts},\tau:\mathrm{sorts},f:\mathrm{obj}(\sigma\Rightarrow\tau)} \Lambda_{\sigma,\tau}(\lambda x{:}\,\mathrm{obj}(\sigma).\,\mathrm{ap}_{\sigma,\tau}(f,x)) =_{\sigma\Rightarrow\tau} f\ \mathrm{true}\end{aligned}$$

Strictly speaking, the equations in the above axioms should be written using ap, for the type of $=_{\tau}$ is $\mathrm{obj}(\tau \Rightarrow \tau \Rightarrow o)$.

The formalization of the logical rules is similar to that of first–order logic. The universal introduction and elimination rules are formalized as follows:

$$\begin{aligned}\forall\mathrm{I} \;&:\; (\textstyle\bigwedge_{x:\mathrm{obj}(\sigma)} \mathrm{ap}_{\sigma,o}(f,x)\ \mathrm{true}) \vdash_{\sigma:\mathrm{sorts},f:\mathrm{obj}(\sigma\Rightarrow o)} \forall_{\sigma}(f)\ \mathrm{true}\\ \forall\mathrm{E} \;&:\; \forall_{\sigma}(f)\ \mathrm{true} \vdash_{\sigma:\mathrm{sorts},f:\mathrm{obj}(\sigma\Rightarrow o),a:\mathrm{obj}(\sigma)} \mathrm{ap}_{\sigma,o}(f,a)\ \mathrm{true}\end{aligned}$$

The adequacy of this representation of higher–order logic can be established by means similar to that for Peano arithmetic.

5 Comparison with Related Work

Work in the area of proof checking began with the AUTOMATH project [4]. They sought to build a framework for expressing arbitrary mathematical texts in a formal way, and developed many examples, notably the formalization of Landau's textbook on Analysis by Jutting [12]. In contrast to the LF approach they work directly within the type theory, using the propositions as types principle. They do not seem to have isolated any general principles about the formalization of logic. Our work can be seen as a development of the AUTOMATH ideas by providing a framework that keeps the meta– and object level clearly separated. We are also concerned with supporting interactive proof development, particularly automated proof assistance, an area that was never considered by the AUTOMATH project.

Paulson's Isabelle system, as presented in [18], is a generalization of LCF to an arbitrary logic. He is primarily concerned with theorem proving, rather than proof checking and proof editing. Consequently his approach is quite different from ours, particularly in the treatment of rules and proofs. Isabelle avoids the construction of proof trees by viewing proof search as a process of building derived rules of inference. His representation of rules is based on a direct encoding of rules and their side conditions, using a clever algorithm to enforce variable occurrence conditions.

6 Directions for Future Research

The LF system is a first step towards developing a general theory of interactive proof checking and proof construction. Much more work remains to be done. At present we do not treat definitions and abbreviations for an object logic. There appear to be at least two ways in which the LF type theory might be extended to include an account of definitional equality. One way is to parameterize the system by axioms for δ reductions [23,4]. We have not yet conducted a thorough analysis of such an extension. Another approach is to formalize LF type theory as an equational theory, with a set of equations representing definitions being included as part of the signature of the logic. These equations may be directed to obtain a reduction relation suitable for use by the type checker, but in general this relation will not be Church–Rosser or normalizing, and so decidability of type checking is lost.

It would be interesting to develop a characterization of the class of logics that can be formalized within the LF. It is clear from recent results of Avron and Mason [1,2] that one can exploit multiple judgement forms to encode logics that ordinarily have complex side conditions on their rules. While it appears that almost any formal system can be represented in the LF, some representations seem more natural than others. Is there a precise characterization of naturality in this sense? If so, what logics admit natural representations?

In a natural representation of a logic, the variables of the object language are represented by variables of the metalanguage. This means that, for the case of first–order logic, that the type ι can be viewed as the domain of quantification in a given model. Thus a satisfactory account of our treatment of variables seems to involve a notion of model for the LF. We have defined a class of models for which the type theory is complete. It is interesting to consider the possibility of connections between the LF and Burstall and Goguen's institutions [10] and Barwise's abstract model theory [3].

A general treatment of tactics is clearly desirable. The terms representing proofs in an LF encoding of a logic can be viewed as validations (in the sense of LCF [11].) Since the proof terms are total functions, a tactic that is validated by a proof term has the property that any proofs of the subgoals are guaranteed to lead to a proof of the goal (such tactics are called *valid* by Milner). Griffin's implementation of the LF demonstrates that this property can be checked automatically for a small class of tactics called *refinement rules*. The work of Constable and Knoblock [13] carries this idea even further by considering the possibility of proving the validity of tactics for type theory in the type theory itself. It would be interesting to adapt these ideas to the more general setting of the LF. In another direction we have defined a logic–independent search space that generalizes Paulson's higher–order resolution [18].

References

[1] Arnon Avron. *Simple Consequence Relations*. Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. In preparation.

[2] Arnon Avron and Ian Mason. *Case Studies in the Edinburgh Logical Framework*. Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. In preparation.

[3] J. Barwise and S. Feferman, editors. *Model–Theoretic Logics. Perspectives in Mathematical Logic*, Springer-Verlag, 1985.

[4] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

[5] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] Robert L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[7] Thierry Coquand. *Une théorie des constructions*. Thèse de Troisième Cycle, Université Paris VII, January 1985.

[8] Thierry Coquand and Gérard Huet. Constructions: a higher–order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184, Springer-Verlag, 1985.

[9] Diedrik T. van Daalen. *The Language Theory of AUTOMATH*. PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.

[10] Joseph Goguen and Rod Burstall. Introducing Institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs*, pages 221–256, Springer-Verlag, 1984.

[11] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1979.

[12] L. S. Jutting. *Checking Landau's Grundlagen in the AUTOMATH System.* PhD thesis, Eindhoven University, The Netherlands, 1977.

[13] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proceedings of the Symposium on Logic in Computer Science*, pages 237–248, 1986.

[14] Per Martin-Lõf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, North-Holland, Amsterdam, 1973.

[15] Per Martin-Lõf. *On the Meanings of the Logical Constants and the Justifications of the Logical Laws.* Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[16] Albert Meyer and Mark Reinhold. 'Type' is not a type: preliminary report. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[17] Bengt Nordström, Kent Petersson, and Jan Smith. *An Introduction to Martin-Lõf's Type Theory.* University of Gõteborg, Gõteborg, Sweden, 1986. Preprint.

[18] Lawrence Paulson. Natural deduction proof as higher–order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[19] Kent Petersson. *A Programming System for Type Theory.* Technical Report 21, Programming Methodology Group, University of Gõteborg/Chalmers Institute of Technology, March 1982.

[20] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almquist & Wiksell, Stockholm, 1965.

[21] Joseph R. Schoenfield. *Mathematical Logic.* Addison–Wesley, Reading, Massachusetts, 1967.

[22] Peter Schroeder–Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), December 1984.

[23] Sõren Stenlund. *Combinators, λ-terms, and Proof Theory.* D. Reidel, Dordrecht, Holland, 1972.

Hoare's Logic in the LF, a case study

Ian Mason
Laboratory for Foundations of Computer Science
Department of Computer Science, Edinburgh University, EH9 3JZ

4

Contents

1. Introduction 2
2. The Subject Matter 2
3. Version One 3
   3.1. The Basic Types 3
   3.2. The While Language 5
   3.3. Hoare's Triples 5
   3.4. The Judgements 5
   3.5. The Rules Concerning α 5
   3.6. The Hoare Rules and Axioms 6
   3.7. The Adequacy Theorems 7
4. Version Two 11
   4.1. Non-Interference Axioms and Rules 11
   4.2. The Axioms and Rules of Hoare Revisited 12
   4.3. The Adequacy Theorems Repaired 12
5. Version Three 13
   5.1. The Rules of the Judgement QF 13
   5.2. The Rules and Axioms of Third Version of Hoare's Logic . . 14
   5.3. The Adequacy Theorems Revisited 14
6. Conclusions 15
7. Acknowledgements 16
8. References 16

1. Introduction

In this working paper we describe several attempts at defining a simple version of Hoare's logic in the LF. This is used as a framework for discussing certain issues that are raised. We give three different attempts at formulating Hoare's logic in the LF, only two of these are successful. Both successful versions suggest certain desiderata concerning the nature of tacticals in the LF and how they can be used in faithfully presenting logics.

2. The Subject Matter

We begin by describing the logic, in the traditional fashion, that we will be studying. Our description here is based closely on that in [Apt, 1981]. Let τ denote a first order language with equality, the meta-variables x, y, z denote or range over the variables of τ, the meta-variables s, t denote or range over the terms or *expressions* of τ, the meta-variable e is used to denote a quantifier-free formula or *boolean expression* of τ, and, finally, p, q, r denote or range over the formulas or *assertions* of τ.

Let W denote the least class of programs such that

1. for every variable x and expression t, $x\colon = t \in W$; and
2. if $S_1, S_2, S_3 \in W$ then $S_1 \,; S_2 \in W$, and for every boolean expression e of τ, we have that $\mathtt{if}(e, S_1, S_2) \in W$ and $\mathtt{while}(e, S_1) \in W$.

The elements of W are called *while programs*, and we take their interpretation to be understood. The basic formulas of Hoare's logic are objects of the form $\{p\}S\{q\}$ where p, q are assertions and S is a while program. The intuitive meaning of an *asserted program*,

$$\{p\}S\{q\},$$

is as follows: whenever p holds before execution of S and S terminates, then q holds after execution of S. Hoare's logic is a system of formal reasoning about these asserted programs. Its axioms and proof rules are the following.

Axiom 1: Assignment Axiom

$$\{p[t/x]\}x\colon = t\{p\}.$$

Rule 2: Composition Rule

$$\frac{\{p\}S_1\{r\}, \quad \{r\}S_2\{q\}}{\{p\}S_1 \,; S_2\{q\}}$$

Rule 3: If Rule

$$\frac{\{p \wedge e\}S_1\{q\}, \quad \{p \wedge \neg e\}S_2\{q\}}{\{p\}\mathtt{if}(e, S_1, S_2)\{q\}}$$

Rule 4: While Rule

$$\frac{\{p \wedge e\}S\{p\}}{\{p\}\texttt{while}(e,S)\{p \wedge \neg e\}}$$

The final rule involves some notion of a consequence relation for the assertion language. The usual approach is to assume a background first order theory, T, for example Peano arithmetic, and a proof system for the assertion language, for example the usual natural deduction system. The rule in question is the following.

Rule 5: Consequence Rule

$$\frac{p \Rightarrow p_1, \quad \{p_1\}S\{q_1\}, \quad q_1 \Rightarrow q}{\{p\}S\{q\}}$$

Here $p \Rightarrow p_1$ and $q_1 \Rightarrow q$ are assumed to follow from the background first order theory using the proof system for the assertion language. As usual, $p[t/x]$ stands for the result of substituting t for the free occurences of x in p.

3. Version One

3.1. The Basic Types

We begin the first version by fixing a finite first order language, τ. Relative to τ we have the following basic types, l_τ is the type of identifiers or memory locations of the while language, i_τ is the type of first order terms in the language τ, b_τ is the type of boolean expression or equivalently quantifier free first order formulas in the language τ, o_τ is the type of first order formulas in the language τ, w_τ is the type of while programs over τ and finally h_τ is the type of hoare triples over τ. We omit explicitly mentioning the dependence on τ whenever possible, thus we write l rather than l_τ.

- l : TYPE
- i : TYPE
- o : TYPE
- b : TYPE
- w : TYPE
- h : TYPE

It is important to notice that we are, by necessity, distiguishing between the variables of the first order logic, which are simply the variables of the LF, and the variables of the while language. This is because the latter are not substitutive. The variables of i and o are the variables of the LF. Also note that since the LF does not have subtypes we must distinguish between the boolean expressions and the first order formulas. The force of this distinction is somewhat diminished by an identification α,

- α : $b \rightarrow o$,

it is supposed to be the obvious identification of boolean expressions and quantifier free formulas. Later we shall state some axioms which attempt to force this identification to be the obvious one. The next operation is the bang which takes an identifier to its contents.

- $! \ : \ l \rightarrow i$

In other words if $x{:}l$ then $x!$ (which we will write instead of the awkward $!(x)$) is the contents of x. It can thus be thought of as an evaluation mechanism. The logical constants on terms are as follows:

- $c_i \ : \ i,$ for each constant symbol $c \in \tau$.
- $f_i \ : \ \underbrace{i \rightarrow i \rightarrow \ldots \rightarrow i}_{n+1 \ i\text{'s}},$ for each n-ary function symbol $f \in \tau$.

Now for the logical constants on formulas:

- $R_o \ : \ \underbrace{i \rightarrow i \rightarrow \ldots \rightarrow i \rightarrow o}_{n \ i\text{'s}}$ for each n-ary relation symbol $R \in \tau$.
- $=_o \ : \ i \rightarrow (i \rightarrow o)$
- $\Rightarrow_o \ : \ o \rightarrow (o \rightarrow o)$
- $\neg_o \ : \ o \rightarrow o$
- $\forall \ : \ (i \rightarrow o) \rightarrow o$

However to make things more readable we include all the usual logical constants, either as new constants (as we do here) or as defined objects, which we do when it comes to proofs.

- $\wedge_o \ : \ o \rightarrow (o \rightarrow o)$
- $\vee_o \ : \ o \rightarrow (o \rightarrow o)$
- $\exists \ : \ (i \rightarrow o) \rightarrow o$

As our notation suggests we will use infix notation whenever appropriate. Thus rather than write $\vee(x_1)(x_2)$ we will use the more standard notation of $x_1 \vee x_2$. These operations, of course, have analagous versions in the case of boolean expressions. These tedious duplications are forced by our inability, on the face of it, to have subtypes in the LF.

- $R_b \ : \ \underbrace{i \rightarrow i \rightarrow \ldots \rightarrow i \rightarrow b}_{n \ i\text{'s}}$ for each n-ary relation symbol of τ.
- $=_b \ : \ i \rightarrow (i \rightarrow b)$
- $\Rightarrow_b \ : \ b \rightarrow (b \rightarrow b)$
- $\neg_b \ : \ b \rightarrow b$

Again to make things more readable we include all the usual logical constants, either as new constants or as defined objects, whichever is most convenient at the time.

- $\wedge_b \ : \ b \rightarrow (b \rightarrow b)$
- $\vee_b \ : \ b \rightarrow (b \rightarrow b)$

The o and b versions of the same operation can be identified at the level of definitions in the current implementation of the LF. By this we mean that we can use the same symbol for both variations, commonly called *overloading*. Although in the current implementation it is the user that does the overloading not the editor itself. In what follows we will also do this, thus we will write $x \wedge y$ in either of the cases when the operands are of type o or b. Context will always prevent confusion.

3.2. The While Language

We now add constants that correspond to the constructs of our while language, they are simply the curried versions of the constructs described in the informal introduction.

- **ass** : $l \rightarrow (i \rightarrow w)$, denoting $\mathtt{ass}(x)(t)$ by $x{:}= t$.
- **seq** : $w \rightarrow (w \rightarrow w)$, denoting $\mathtt{seq}(w_1)(w_2)$ by $w_1 \,;\, w_2$.
- **if** : $b \rightarrow (w \rightarrow (w \rightarrow w))$, denoting $\mathtt{if}(b_1)(w_1)(w_2)$ by $\mathtt{if}(b_1, w_1, w_2)$.
- **while** : $b \rightarrow (w \rightarrow w)$, denoting $\mathtt{while}(b_1)(w_1)$ by $\mathtt{while}(b_1, w_1)$.

3.3. Hoare's Triples

The syntax of the Hoare triples is easily taken care of by a single constant.

- **triple** : $o \rightarrow (w \rightarrow (o \rightarrow h))$, denoting $\mathtt{triple}(o_1)(w_1)(o_2)$ by $\{o_1\}w_1\{o_2\}$.

3.4. The Judgements

The following are the judgements of our first version, they correspond to judgement TRUE or more appropriately provable sentences in the respective classes of expressions.

- $\vdash_h \ : \ h \rightarrow \mathrm{TYPE}$
- $\vdash_b \ : \ b \rightarrow \mathrm{TYPE}$
- $\vdash_o \ : \ o \rightarrow \mathrm{TYPE}$

Again we can use the definition mechanism to hide the distinction between these three judgements. Thus we shall omit the subscript leaving context to do its job.

3.5. The Rules Concerning α

The rules concerning the identification α are straight forward and so we shall not make any undue fuss.

- $\alpha_1 \ : \ \Pi_{x:b} \ \ (\vdash x) \rightarrow (\vdash \alpha(x))$
- $\alpha_2 \ : \ \Pi_{x:b} \ \ (\vdash \alpha(x)) \rightarrow (\vdash x)$
- $\alpha_{3R} \ : \ \Pi_{x_1:i} \dots \Pi_{x_n:i} \ \ (\vdash \alpha(R_b(x_1, \dots, x_n))) \rightarrow (\vdash R_o(x_1, \dots, x_n))$ for $R \in \tau \cup \{=\}$.

- α_{4R} : $\Pi_{x_1:i} \ldots \Pi_{x_n:i}$ $(\vdash R_o(x_1, \ldots, x_n)) \to (\vdash \alpha(R_b(x_1, \ldots, x_n)))$ for $R \in \tau \cup \{=\}$.

For ease of use it is probably also useful to include, as either new constants or derived terms, the following:

- α_5 : $\Pi_{x_1:i}\Pi_{x_2:i}$ $\vdash (\alpha(x_1 = x_2) \Leftrightarrow x_1 = x_2)$
- α_6 : $\Pi_{x:b}$ $\vdash (\alpha(\neg x) \Leftrightarrow \neg\alpha(x))$
- $\alpha_{7\chi}$: $\Pi_{b_1:b}\Pi_{b_2:b}$ $\vdash (\alpha(b_1 \chi b_2) \Leftrightarrow (\alpha(b_1)\chi\alpha(b_2)))$ for $\chi \in \{\wedge, \vee, \Rightarrow\}$.

Clearly, given the usual interpretation of the logical constants, α_1 and α_2 imply $\alpha_5, \ldots, \alpha_7$ but the converse, even with the addition of α_3 and α_4 is false due to our inability to force b to be inductively generated by the relevant constants. In this sense it could hardly be said that we are formalizing the syntax.

3.6. The Hoare Rules and Axioms

We omit the rules of first order logic, the interested reader is refered to [Harper, Honsell and Plotkin, 1986]. We continue with the rules of Hoare, they are the obvious analogs of the informal rules given the preceeding setting.

- **`Ass`** : $\Pi_{x:l}\Pi_{t:i}\Pi_{p:i\to o}$

$$\vdash \{p(t)\}x:=t\{p(x!)\}$$

- **`Comp`** : $\Pi_{a:o}\Pi_{b:o}\Pi_{c:o}\Pi_{w_1:w}\Pi_{w_2:w}$

$$\vdash \{a\}w_1\{b\} \to \ \vdash \{b\}w_2\{c\} \to \ \vdash \{a\}w_1\,;w_2\{c\}$$

- **`If`** : $\Pi_{e:b}\Pi_{e_1:o}\Pi_{e_2:o}\Pi_{w_1:w}\Pi_{w_2:w}$

$$\vdash \{e_1 \wedge \alpha(e)\}w_1\{e_2\} \to \ \vdash \{e_1 \wedge \neg\alpha(e)\}w_2\{e_2\} \to \ \vdash \{e_1\}\mathtt{if}(e, w_1, w_2)\{e_2\}$$

- **`While`** : $\Pi_{e:b}\Pi_{f:o}\Pi_{w_1:w}$

$$\vdash \{f \wedge \alpha(e)\}w_1\{f\} \to \ \vdash \{f\}\mathtt{while}(e, w_1)\{f \wedge \neg\alpha(e)\}$$

- **`Con`** : $\Pi_{a:o}\Pi_{a_1:o}\Pi_{b:o}\Pi_{b_1:o}\Pi_{w_1:w}$

$$\vdash a_1 \Rightarrow a \to \ \vdash b_1 \Rightarrow b \to \ \vdash \{a_1\}w_1\{b_1\} \to \ \vdash \{a\}w_1\{b\}$$

3.7. The Adequacy Theorems

We now state the adequacy theorems for syntax and semantics that should hold in any successful internalization of Hoare's logic in the LF. Beginning with the adequacy for syntax, which is stated for each separate syntactic category. We begin with some notation. Let Γ^m_n be the following context, for $m, n \in \mathbb{N}$:

$$\Gamma^m_n = \{y_0{:}i, \ldots, y_m{:}i, x_0{:}l, \ldots, x_n{:}l\}.$$

Theorem (Adequacy for Syntax): In the above LF signature and in the context Γ^m_n we have the following facts concerning syntax:

- *l*: All well formed long $\beta\eta$ normal forms of type *l* are LF variables of type *l*, and hence are among the $x_0, \ldots, x_n$.
- *i*: There is a compositional bijection, τ_i, between well formed long $\beta\eta$ normal forms of type *i* and terms of the assertion language built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$.
- *b*: There is a compositional bijection, τ_b, between well formed long $\beta\eta$ normal forms of type *b* and quantifier free formulas of the assertion language built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$.
- *o*: There is a compositional bijection, τ_o, between well formed long $\beta\eta$ normal forms of type *o* and formulas of the assertion language built up from the set of identifiers $\bar{x}$ (which if they occur must occur free) and the logical variables $\bar{y}$.
- *w*: There is a compositional bijection, τ_w, between well formed long $\beta\eta$ normal forms of type *w* and the while programs of τ built up from the set of identifiers $\bar{x}$ and the logical variables *y* (which do not occur in the left hand side of any assignment statement).
- *h*: There is a compositional bijection, τ_h, between well formed long $\beta\eta$ normal forms of type *h* and asserted programs (i.e. Hoare triples) built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$. Where again no variable from $\bar{y}$ can occur in the left hand side of any assignment statement.

Remarks:

• Actually there is a minor problem in the statement of the result for *w* and hence for *h*. This arises because in the informal description of the while language there is no distinction between $S_0\,;(S_1\,;S_2)$ and $(S_0\,;S_1)\,;S_2$ where as these are quite distinct from the point of view of the LF internalization. Perhaps the simplest solution to this minor technicality is to be somewhat more precise in the informal description, incorporating this distinction there. Thus the second clause in the informal description should be restated as

2. if $S_1, S_2, S_3 \in W$ then $(S_1\,;S_2) \in W$, and for every boolean expression *e* of τ, we have that $\mathtt{if}(e, S_1, S_2) \in W$ and $\mathtt{while}(e, S_1) \in W$.

• Long $\beta\eta$ normal forms (see [Jensen and Pietrzykowski, 1976] for a definition) have been chosen, rather than $\beta\eta$ normal forms, so as to overcome the difficulty of deciding which bound variables may occur in the set of first order formulas built over $\bar{x}$ and $\bar{y}$. Since both λ-expressions and formulas are only considered upto α-equivalence and also since $\forall(M)$ occurs in long $\beta\eta$ normal form iff $M \equiv \lambda x.N$ where N is of type o we can define $r_o(\forall(\lambda x.N))$ to be $\forall x r_o(N)$.

Outline of Proof: We begin by defining, recursively, the collections of long $\beta\eta$ normal forms of, in the context Γ_n^m, the syntactic categories in question. In what follows N_x^n denotes these forms for the category x.

- $N_l^m = x_0 \mid \ldots \mid x_n$.
- $N_i^m = y_0 \mid \ldots \mid y_m \mid c_i \mid N_l^m! \mid f_i \underbrace{N_i^m \ldots N_i^m}_{s\ \mathbf{times}}$ for all constants c and s-ary functions symbols in τ.
- $N_b^m = R_b \underbrace{N_i^m \ldots N_i^m}_{s\ \mathbf{times}} \mid N_i^m =_b N_i^m \mid \neg_b N_b^m \mid N_b^m \Rightarrow_b N_b^n$ for any s-ary relation in τ.
- $N_o^m = R_o \underbrace{N_i^m \ldots N_i^m}_{s\ \mathbf{times}} \mid N_i^m =_o N_i^m \mid \neg_o N_o^m \mid N_o^m \Rightarrow_o N_o^n \mid \forall(\lambda y_{m+1}{:}i.N_o^{m+1})$ for any s-ary relation in τ.
- $N_w^m = N_l^m := N_i^m \mid (N_w^m \,; N_w^m) \mid \mathtt{if}(N_b^m, N_w^m, N_w^m) \mid \mathtt{while}(N_b^m, N_w^m)$.
- $N_h^m = \{N_o^m\}N_w^m\{N_o^m\}$.

The compositional bijections, for each category, are then defined inductively on the above categories in the obvious fashion. □

Now we proceed to the adequacy for semantics. In the case for b and o there is very little difference between the results here and those stated in [Harper, Honsell and Plotkin, 1986]. The only remark needed to be made is that the identifiers are taken to behave like constants. Thus we shall concentrate on the novel case of h.

Theorem (Adequacy for Semantics): There is a compositional bijection between proofs of a Hoare triple $\{p\}S\{q\}$ from assumptions $r_0, \ldots, r_s$ (in the assertion language) and assumptions $\{p_0\}S_0\{q_0\}, \ldots, \{p_t\}S_t\{q_t\}$ (concerning asserted programs) and well formed $\beta\eta$ normal forms of type

$$\vdash_h \{p\}S\{q\}$$

in the above signature and in the context Γ where

$$\Gamma = \Gamma_n^m \cup \{w_j{:} \vdash_o r_j, v_i{:} \vdash_h \{p_i\}S_i\{q_i\}\}_{0\leq j\leq s, 0\leq i\leq t},$$

and Γ_n^m is adequate for the syntax of objects involved.

Unfortunately this theorem is false under this internalization, the rule Ass is in fact erroneous. We give two examples of this failure, one at the level of locations and the other at the level of LF terms. They are essentially the same example, viewed at different levels of abstraction, and provide us with two different (though essentially equivalent) motivations for the solution we shall present. Take τ to be the language of arithmetic.

Example 1. In the context

$$\Gamma = \{x{:}l,\ y{:}l\}$$

we have the following instance of the assignment axiom Ass,

$$\{\neg(x! = 1)\}y{:} = 1\{\neg(x! = y!)\}.$$

Now x and y are simply LF variables that are declared to be of type l. No other assumption about them has been made. Consequently it is reasonable to assume that they both denote the same physical location or identifier l_0. In this case the axiom states that

$$\{\neg(l_0! = 1)\}l_0{:} = 1\{\neg(l_0! = l_0!)\},$$

and since it is reasonable to assume that $\neg(l_0! = 1)$ is a definite possiblity we arrive at a somewhat unfortunate state of affairs.

Example 2. Suppose $\Gamma = \{y{:}l\}$ is the current context and take the following instantiation of the assignment axiom,

$$\mathtt{ASS}(y)(1)(\lambda u.\neg(y! = u)).$$

This term inhabits the following type

$$\vdash \{\neg(y! = 1)\}y{:} = 1\{\neg(y! = y!)\}.$$

Which one would have hope was uninhabitable.

The problem in the first example, intuitively, is that

$$\{P(t)\}x{:} = t\{P(x!)\}$$

can be false because the assignment $x{:} = t$ can alter the meaning of the predicate $\lambda zP(z)$. Thus the simplest solution to this problem is to axiomatize the relevant notion of non-interference. This solution is some sense however hides the source of the problem, since a casual glance reveals no crucial differences between our interpretation here and the informal description we began with. There is a crucial difference however because our version is inconsistent unlike the informal one.

One point that can be made here is that the LF cannot handle meta-variables in the way they are commonly used in describing logics. The reason that the inconsistency does not arise in the informal version is that, in essence, the operator $p[t/x]$ is call by value; it replaces all occurences of the *value* of the meta-variable x (which is a variable of τ) by the *value* of the meta-variable t (which is a term of τ). In contrast to this, substitution in the LF,

$$(\lambda zP(z))(t)$$

takes place at the level of the meta-variables (i.e. the variables of the LF) and not at the level of their values. In this sense LF substitution could be called call by name.

Perhaps a clearer explaination can be given by examining when the two notions of substitution, 1. $p[t/x]$ and 2. $(\lambda z.P(z))(t)$, coincide and when they differ. To make the following discussion more readable we omit the bang operator, ! , since neither its presence nor its absence has any bearing on the phenomenon we are considering. The first and most obvious difference between these two operators is that they apply to different sorts of objects. The first form has as its arguments — a formula, a variable and a term. The second form has as arguments (in the notation of the LF) a function from terms to formulas and a term. To make this explicit we shall write $Sub1(p, x, t) = p[t/x]$, and $Sub2(P, t) = P(t)$.

One way of unifying the picture is to decompose the first form of substitution, $Sub1(p, x, t)$, into two separate operations. Given a formula p we first form the function, $\lambda x.p$, from terms to formulas, we then apply this function to the given term t. Thus we conclude that

$$Sub1(p, x, t) = p[t/x] = (\lambda x.p)(t) = Sub2(\lambda x.p, t),$$

hence the first form of substitution can be considered as a special case of the second.

Similarly we can express the second form of substitution, $Sub2(P, t)$, in terms of the first, but in this case we need a side condition. $Sub2(P, t)$ can be also be decomposed into two operations. Given P we first obtain a formula by applying P to a *new* variable x, we then replace all occurences of this new variable by the supplied term t. By a *new* variable we mean a variable that does not occur *free* in P. This is the same, in the LF, as saying that the variable does not occur free in $\forall P$. In this case we can conclude that

$$Sub2(P, t) = P(t) = (P(x))[t \setminus x] = Sub1(P(x), x, t) \quad \text{when} \quad x \notin FV(\forall P).$$

It is imporatant to notice that in the Hoare triple

$$\{P(t)\}x:=t\{P(x!)\}$$

free occurences of x in $P(x!)$ are bound by the assignment operator $x:=t$, this is not true of those occurences in $P(t)$. Thus in example 2. we have a clear case of a variable being captured, in the right hand side, during the process of substitution. Thus in one sense α-conversion should take place. This however would not be in the spirit of Hoare's logic, since we want to reason about the identifier x not some α-conversion of it. Thus the assignment axiom can be correctly stated, informally, as

- **Ass** : $\Pi_{x:\iota}\Pi_{t:i}\Pi_{p:i\to o}$

$$x \notin FV(\forall P) \to \ \vdash \{p(t)\}x:=t\{p(x!)\}.$$

We now put this new found knowledge into practice.

4. Version Two

The easiest solution to this problem of formalizing the corrected version of the assignment axiom is to incorporate syntactic notions explicitly into the theory. We do this by adding three new judgements concerning non-interference along the lines of [Reynolds, 1978].

- $\neq \;:\; l \rightarrow (l \rightarrow \mathrm{TYPE})$
- $\sharp_i \;:\; l \rightarrow (i \rightarrow \mathrm{TYPE})$
- $\sharp_o \;:\; l \rightarrow (o \rightarrow \mathrm{TYPE})$

As per usual we will identify a judgement with the type of its evidence, which in this case consists of its proofs. The intuitive meaning of the judgements can be explained, again using infix notation, as follows:

- $x \neq y$ is interpreted as meaning that x and y denote distinct identifiers or locations. As we have already mentioned, because we have no constructors or constants of type l, terms of type l must $\beta\eta$ reduce to LF variables.
- $x\sharp_i t$ is interpreted as meaning that no assignment to the location denoted by x effects the value of the term denoted by t. This of course is equivalent to saying that the location or identifier denoted by x does not occur in the term denoted by t.
- $x\sharp_o e$ is interpreted as meaning that no assignment to the location denoted by x effects the value or meaning of the formula denoted by e. Again this is equivalent to saying that the location or identifier denoted by x does not occur freely in the formula denoted by e (Note that it cannot occur bound).

Again we will omit the subscripts in favor of context.

4.1. Non-Interference Axioms and Rules

It is a simple task to axiomatize the above notions, and we present one such here.

- $\sharp_{0c} \;:\; \Pi_{x:l} \quad x\sharp c_i$, for each constant c in τ.
- $\sharp_1 \;:\; \Pi_{x:l}\Pi_{y:l} \quad x \neq y \rightarrow x\sharp y!$
- $\sharp_{2f} \;:\; \Pi_{t_1:i}\Pi_{t_2:i}\ldots\Pi_{t_n:i}\Pi_{x:l} \quad x\sharp t_1 \rightarrow x\sharp t_2 \rightarrow \ldots \rightarrow x\sharp t_n \rightarrow x\sharp f(t_1, t_2, \ldots, t_n)$, for each n-ary operation in τ.
- $\sharp_{3R} \;:\; \Pi_{t_1:i}\Pi_{t_2:i}\ldots\Pi_{t_n:i}\Pi_{x:l} \quad x\sharp t_1 \rightarrow x\sharp t_2 \rightarrow \ldots \rightarrow x\sharp t_n \rightarrow x\sharp R(t_1, t_2, \ldots, t_n)$, for each n-ary relation in $\tau \cup \{=\}$.
- $\sharp_4 \;:\; \Pi_{e:o}\Pi_{x:l} \quad x\sharp e \rightarrow x\sharp\neg e$
- $\sharp_{5\chi} \;:\; \Pi_{e_1:o}\Pi_{e_2:o}\Pi_{x:l} \quad x\sharp e_1 \rightarrow x\sharp e_2 \rightarrow x\sharp e_1 \chi e_2$ for $\chi \in \{\wedge, \vee, \Rightarrow\}$.
- $\sharp_6 \;:\; \Pi_{f:i\rightarrow o}\Pi_{x:l}\Pi_{y:i} \quad (x\sharp y \rightarrow x\sharp f(y)) \rightarrow x\sharp\forall f$

That we have captured the correct notion is expressed in the following proposition. Its proof is an easy induction.

Proposition: Suppose that $\Gamma = \{x_0{:}l, x_i{:}l, z_i{:}x_0 \neq x_i\}_{i\in I}$ and $\Gamma \vdash (\phi \ : \ o)$. Then the following are equivalent

1. $\Gamma \vdash x_0 \sharp \phi$
2. $FV(\phi) \subseteq \{x_i\}_{i\in I}$.

We should point out that to correctly formalize more complex versions of Hoare's logic, for example one in which recursive procedure calls were allowed, it would be necessary to incorporate the notion on non-interference anyway. Thus in the long run we have not payed such a high price. The notion of non-interference is a syntactic one, and to axiomatize it as we have done above relies heavily on the existance of the type l. Thus we are provided with another, perhaps more compelling, reason why the Hoare's logic variables cannot be identified with the variables of the LF. The above proof system for non-interference has a very special property that we shall discuss in detail later. Put crudely if a non-interference judgement can be proved, then there is, in a strong sense, a unique such proof. But let us not digress from the immediate problem at hand, namely repairing the inconsistency in our first version.

4.2. The Axioms and Rules of Hoare Revisited

Using the non interference judgement we can formulate the correct version of the assignment axiom as follows:

- ASS : $\Pi_{x:l}\Pi_{t:i}\Pi_{p:i\to o} \quad x\sharp\forall p \to \quad \vdash \{p(t)\}x{:} = t\{p(x!)\}$

The remaining rules are the same as in the inadequate version and so we do not waste space repeating them here.

4.3. The Adequacy Theorems Repaired

In this version the adequacy theorem for syntax remains the same, however we modify the definition of Γ^m_n so that it includes the assumption that all distinct LF variables of type l denote distinct locations or identifiers. Explicitly define Γ^m_n as follows

$$\Gamma^m_n = \{y_0{:}i, \ldots, y_m{:}i, x_0{:}l, \ldots, x_n{:}l, z_{i,j}{:}x_i \neq x_j, z^*_{i,j}{:}x_j \neq x_i\}_{0\leq i\leq j\leq n}.$$

The adequacy theorem for semantics is then identical to the false one in the preceeding section. The only difference now is that it is true.

Theorem (Adequacy for Semantics): There is a compositional bijection between proofs of a Hoare triple $\{p\}S\{q\}$ from assumptions $r_0, \ldots, r_s$ (in the assertion language) and assumptions $\{p_0\}S_0\{q_0\}, \ldots, \{p_t\}S_t\{q_t\}$ (concerning asserted programs) and well formed $\beta\eta$ normal forms of type

$$\vdash_h \{p\}S\{q\}$$

in the above signature and in the context Γ where

$$\Gamma = \Gamma^m_n \cup \{w_j{:} \vdash_o r_j, v_i{:} \vdash_h \{p_i\}S_i\{q_i\}\}_{0\leq j\leq s, 0\leq i\leq t},$$

and

$$\Gamma^m_n = \{y_0{:}i, \ldots, y_m{:}i, x_0{:}l, \ldots, x_n{:}l, z_{i,j}{:}x_i \neq x_j, z^*_{i,j}{:}x_j \neq x_i\}_{0 \leq i \leq j \leq n}.$$

is adequate for the syntax of objects involved.

The fact that there is a compositional bijection depends heavily on the following fact concerning the non-interference proof system.

Proposition (Uniqueness): Suppose there are well formed LF terms x, e, P_0 and P_1 such that

0. $\Gamma^m_n \vdash x \ : \ l$,
1. $\Gamma^m_n \vdash e \ : \ o$,
2. $\Gamma^m_n \vdash P_0 \ : \ x \sharp e$,
3. $\Gamma^m_n \vdash P_1 \ : \ x \sharp e$.

Then P_0 and P_1 have the same $\beta\eta$ normal forms.

This is important since if there were distinct proofs of a non-interference judgement then the extra parameter to the Ass axiom would force the mapping to identify different $\beta\eta$ normal forms.

5. Version Three

The third method of representing Hoare's logic is to utilize the method of using judgements to interpret or implement subtypes. In the case of Hoare's logic there is only one problematic subtype, that of the boolean expressions b. In this example we introduce a new judgement QF over o, whose interpretation is that of a formula being *quantifier-free.*

- QF : $o \rightarrow$ TYPE

The two program constructions `if` and `while` now take a extra argument, namely a proof, equivalently an element of the QF judgement, that their o argument is quantifier free. We shall discuss the advantages and disadvantages of this approach after we have described it fully.

- IF : $\Pi_{e:o}\mathtt{QF}(e) \rightarrow w \rightarrow w \rightarrow w$, denoting $\mathtt{IF}(e)(p)(w_1)(w_2)$ by $\mathtt{if}(e, w_1, w_2)_p$.
- WHILE : $\Pi_{e:o}\mathtt{QF}(e) \rightarrow w \rightarrow w$, denoting $\mathtt{WHILE}(e)_p(w_1)$ by $\mathtt{while}(e, w_1)_p$.

5.1. The Rules of the Judgement QF

Axiomatizing the syntactic notion of being quantifier free, as in the case of non-interference, presents no problems.

- $\mathtt{QF}_{1R}$: $\Pi_{t_1:i}\Pi_{t_2:i}\ldots\Pi_{t_n:i} \quad \mathtt{QF}(R(t_1, t_2, \ldots, t_n))$, for each n-ary relation in $\tau \cup \{=\}$.
- $\mathtt{QF}_2$: $\Pi_{e_1:o} \quad \mathtt{QF}(e_1) \rightarrow \mathtt{QF}(\neg e_1)$
- $\mathtt{QF}_{3\chi}$: $\Pi_{e_1:o}\Pi_{e_2:o} \quad \mathtt{QF}(e_1) \rightarrow \mathtt{QF}(e_2) \rightarrow \mathtt{QF}(e_1 \chi e_2)$ for $\chi \in \{\wedge, \vee, \Rightarrow\}$.

Just as in the case of the proof system for non-interference, this proof system has the property (which we will discuss in more detail when we come to the adequacy theorem for syntax) that proofs are in a sense unique.

Proposition (Uniqueness:) Suppose there are well formed LF terms e, P_0 and P_1 such that

1. $\Gamma^m_n \vdash e \;:\; o$,
2. $\Gamma^m_n \vdash P_0 \;:\; \mathtt{QF}(e)$,
3. $\Gamma^m_n \vdash P_1 \;:\; \mathtt{QF}(e)$.

Then P_0 and P_1 have the same $\beta\eta$ normal forms.

5.2. The Rules and Axioms of Third Version of Hoare's Logic

In this final version the rules need only be modified so as to take into account the extra argument to the `if` and `while` constructs. We state them here in there entirety for reason of emphasis.

- $\mathrm{H}_1 \;:\; \Pi_{x:l}\Pi_{t:i}\Pi_{p:i\to o}$
$$x\sharp\forall p \to \;\vdash \{p(t)\}x{:}=t\{p(x!)\}$$
- $\mathrm{H}_2 \;:\; \Pi_{e_1:o}\Pi_{e_2:o}\Pi_{e_3:o}\Pi_{w_1:w}\Pi_{w_2:w}$
$$\vdash \{e_1\}w_1\{e_2\} \to \;\vdash \{e_2\}w_2\{e_3\} \to \;\vdash \{e_1\}w_1\,;w_2\{e_3\}$$
- $\mathrm{H}_3 \;:\; \Pi_{e:o}\Pi_{e_1:o}\Pi_{e_2:o}\Pi_{w_1:w}\Pi_{w_2:w}\Pi_{p:\mathtt{QF}(e)}$
$$\vdash \{e_1 \wedge e\}w_1\{e_2\} \to \;\vdash \{e_1 \wedge \neg e\}w_2\{e_2\} \to \;\vdash \{e_1\}\mathtt{if}(e, w_1, w_2)_p\{e_2\}$$
- $\mathrm{H}_4 \;:\; \Pi_{e:o}\Pi_{e_1:o}\Pi_{w_1:w}\Pi_{p:\mathtt{QF}(e)}$
$$\vdash \{e_1 \wedge e\}w_1\{e_1\} \to \;\vdash \{e_1\}\mathtt{while}(e, w_1)_p\{\neg e_1\}$$
- $\mathrm{H}_5 \;:\; \Pi_{e_1:o}\Pi_{e_1':o}\Pi_{e_2:o}\Pi_{e_2':o}\Pi_{w_1:w}$
$$\vdash e_1 \Rightarrow e_1' \to \;\vdash e_2' \Rightarrow e_2 \to \;\vdash \{e_1'\}w_1\{e_2'\} \to \;\vdash \{e_1\}w_1\{e_2\}$$

5.3. The Adequacy Theorems Revisited

In this third version the only syntactic categories which have changed (other than the elimination of b) are w and h. Consequently we need only state the adequacy theorem for syntax for these two categories, since the previous adequacy theorem for syntax is still applicable to the remaining categories. As before define

$$\Gamma^m_n = \{y_0{:}i, \ldots, y_m{:}i, x_0{:}l, \ldots, x_n{:}l, z_{i,j}{:}x_i \neq x_j, z^*_{i,j}{:}x_j \neq x_i\}_{0\leq i\leq j\leq n}.$$

Theorem (Adequacy for Syntax): In the above LF signature and in the context Γ_n^m we have the following facts concerning syntax:

w : There is a compositional bijection between well formed $\beta\eta$ normal forms of type w and the while programs of τ built up from the set of identifiers $\bar{x}$ and the logical variables y (which do not occur in the left hand side of any assignment statement).

h : There is a compositional bijection between well formed $\beta\eta$ normal forms of type h and asserted programs (i.e. Hoare triples) built up from the set of identifiers $\bar{x}$ and the logical variables $\bar{y}$. Where again no variable from $\bar{y}$ can occur in the left hand side of any assignment statement.

That there is a compositional bijection relies heavily on the uniqueness theorem for the QF judgement, since if there were distinct proofs then the extra parameter to the if and while operations would force the mapping to identify different $\beta\eta$ normal forms.

Theorem (Adequacy for Semantics): There is a compositional bijection between proofs of a Hoare triple $\{p\}S\{q\}$ from assumptions $r_0, \dots, r_s$ (in the assertion language) and assumptions $\{p_0\}S_0\{q_0\}, \dots, \{p_t\}S_t\{q_t\}$ (concerning asserted programs) and well formed $\beta\eta$ normal forms of type

$$\vdash_h \{p\}S\{q\}$$

in the above signature and in the context Γ where

$$\Gamma = \Gamma_n^m \cup \{w_j{:} \vdash_o r_j, v_i{:} \vdash_h \{p_i\}S_i\{q_i\}\}_{0 \le j \le s, 0 \le i \le t},$$

and

$$\Gamma_n^m = \{y_0{:}i, \dots, y_m{:}i, x_0{:}l, \dots, x_n{:}l, z_{i,j}{:}x_i \ne x_j, z_{i,j}^*{:}x_j \ne x_i\}_{0 \le i \le j \le n}$$

is adequate for the syntax of objects involved.

6. Conclusions

Thus we have presented two distinct versions of Hoare's logic, both somewhat more complex than the informal description. The question then arises as to which one is better or more faithful. In this conclusion we shall try to argue that the third version has the potential to be the most faithful. If we translate the uniqueness properties for the non-interference and quantifier-free judgements into properties concerning the search space, we notice that they assert that these spaces are linear. Consequently it would be desirable for the LF to provide tacticals which automatically construct the proofs. Note that such a tactical is no more complex in nature that the already implemented **Pi-Intro***, [Griffin, 1987]. It would however be more complex than those found in [Schmidt, 1983]. If this were the case then both judgements, QF and $\sharp$, could be hidden from the user. In the case of the third version this would result in a system almost identical to the informal description. The only difference being the presence of the type l and the associated function !. The third version also exemplifies a useful technique for implementing syntactic subtypes as judgements rather than distinct types. If the proof system for these judgements have the uniqueness property, then they have a distinct methodological advantage over the *proliferation of types* method.

7. Acknowledgements

This paper, such as it is, could not have been written without the numerous helpful comments and suggestions of Furio Honsell. Gordon Plotkin also helped "show the fly the way out of the fly-bottle", as did Arnon Avron, Tim Griffin and Bob Harper.

8. References

- Apt, K.R. Ten Years of Hoare's Logic: A Survey— Part 1. A.C.M. Transactions on Programming Languages and Systems. Vol. 3, No. 4, October 1981, pp 431-483.
- Griffin, T. So It Doesn't Whistle. To appear.
- Harper, R., Honsell, F., and Plotkin, G. A Framework for Defining Logics. Proceedings of the Second Annual Conference on Logic in Computer Science. Cornell, 1987.
- Jensen, D.C., and Pietrzykowski, T. Mechanizing ω-order Type Theory through Unification. Theoretical Computer Science. Vol 3. 1976. pp123 171.
- Reynolds, J.C. Syntactic Control of Interference. Conference Record of the Fifth Annual Symposium on Principles of Programming Languages, Tucson, 1978.
- Schmidt, D. A Programming Notation for Tactical Reasoning. Department of Computer Science Internal Report No. CSR-141-83, Edinburgh University, 1983.

Peter Aczel

Long Term Program

To implement in a practically useful computer environment a formal language for mathematics which $\left\{\begin{array}{l}\text{mathematicians}\\ \text{software engineers}\end{array}\right\}$ will want to use.

? formal language
computer environment

Automath
Nuprl

? The formal language should be "sensible".

Need to experiment!

Need a framework in which a formal language can be easily implemented and the implementation easily modified
e.g. Edinburgh LF

Formal Language = Interpreted $\boxed{\text{formal system}}$

The Developement of the notion "Formal System"

1. Informal Axiom System
   - e.g. Euclid ← categorical
   - or axioms for a vector space ← many interpretations
   - Primitive notions treated formally
   - Logical Inference left informal

   Made rigorous by Hilbert

2. Formalisation of Logic

   Frege

   Hilbert & Ackerman

   First order logic

   2nd order logic

   ⋮

3. Formal Language for Mathematics

   e.g. Principia Mathematica

   sensible?

4. Hilbert's Metamathematics

   Formal system for Maths.

   = Formal Language for Maths without it's interpretation

   Finitism and Hilbert's program

5. Tarski's mathematical semantics

6. Formal Systems in general

   Post, Curry

Modern Mathematical Logic

Axiomatic Set Theory
Formal Systems
Mathematical Semantics

Formal Languages ?

Refs for formal systems (previous treatments)

Foundations of Mathematical Logic
H.B. Curry 1963
Theory of Formal Systems
R.M. Smullyan 1961

{computation
{deduction?

{ algorithm
{ system of generation rules

{partial recursive functions
{recursively enumerable relations

{universal machine
{universal formal system

But no computational analogue of formal language (= interpreted formal system)

Deduction involves understanding

Formal Systems à la Curry

- A formal system is a deductive theory i.e. there is a collection of statements and axioms and rules of inference for obtaining theorems
- Statements have the form

$$P(a_1, \ldots, a_n)$$

predicate   expressions

e.g. $\vdash \varphi$ $a = b \in A$ A type etc...

- Expressions are either
  - strings of symbols (Syntactic systems)
  - terms in the free algebra (Ob systems) relative to a signature

e.g. syntactic   Post Productions
Smullyan formal systems
Formal grammars

Ob   Abstract syntax
Prolog

The Edinburgh LF notion of formal system has also

- Many sorts
- Dependent sorts
- Higher order sorts
- Variable binding operations

(v)

ARITIES $(k_1 \cdots k_n)$

for $n \geq 0,\ k_1, \ldots, k_n \geq 0$

$[n] = (\overbrace{0 \cdots 0}^{n})$ is the arity of n-place functions

$[0] = ()$ is the arity of objects

$(k_1 \cdots k_n)$ is the arity of n-place functionals having arguments of arities $[k_1], \ldots, [k_n]$ and having objects as values

$[0]$ has level 0

$[n]$ (for $n > 0$) has level 1

$(k_1 \cdots k_n)$ (with some $k_i > 0$) has level 2

SIGNATURE

= list of constant symbols each of some arity.

Variables

$X^n, Y^n, \ldots$ of arity [n]

$x, y, \ldots$ of arity [0]

Expressions of each arity

object expression
= expression of arity [0]

I. (i) Every constant or variable of arity [0] is an object expression

(ii) If f is a constant or variable of arity [n] and $a_1, \ldots, a_n$ are object expressions then

$$f(a_1, \ldots, a_n)$$

is an object expression

(iii) If F is a constant of arity $(k_1 \cdots k_n)$ and $f_1, \ldots, f_n$ are expressions of arities $k_1, \ldots, k_n$ respectively then

$$F(f_1, \ldots, f_n)$$

is an object expression

II. If a is an object expression and $X_1,\ldots,X_n$ is a non-repeating list of variables of arities $[k_1],\ldots,[k_n]$ respectively then

$$(X_1,\ldots,X_n)\,a$$

is an expression of arity $(k_1\cdots k_n)$.

For $i=1,\ldots,n$ free occurrences of X_i in a become bound in $(X_1,\ldots,X_n)a$.

Substitution

If $F=(X_1,\ldots,X_n)a$ of arity $(k_1\cdots k_n)$
f_i of arity $[k_i]$ for $i=1,\ldots,n$

then $$F(f_1,\ldots,f_n)$$

is the result of simultaneously substituting f_i for X_i in a for $i=1,\ldots,n$. (changing bound variables when necessary) Then $X_i(b_1,\ldots,b_{k_i})$ become $f_i(b_1,\ldots,b_{k_i})$ which also involves a substitution if $k_i>0$

Conventions

(1) Identify α-convertible expressions

(2) Use standard conventions for leaving out brackets infix notation and variable binding

e.g. $a \,\&\, b$ for $\&(a, b)$

$(\forall x)a$ for $\forall((x)a)$

$(a \underset{x}{\supset} b)$ for $\sqsupset((x)a, (x)b)$

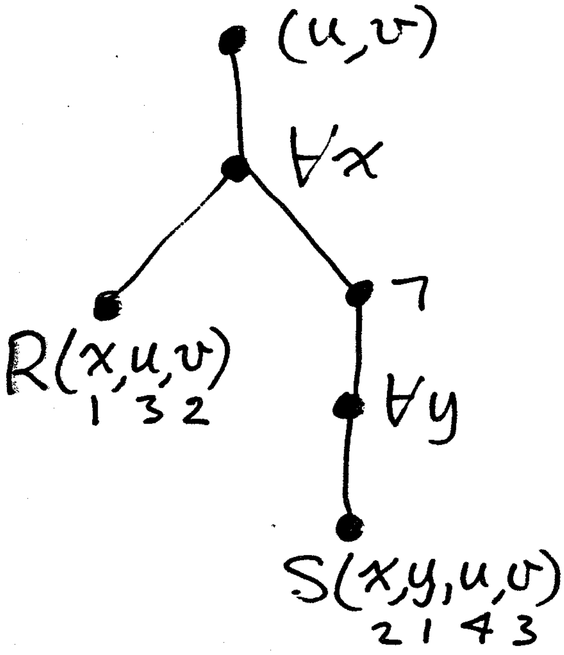
or $(x)a \sqsupset (x)b$

Here $\&, \forall, \sqsupset$ have arities

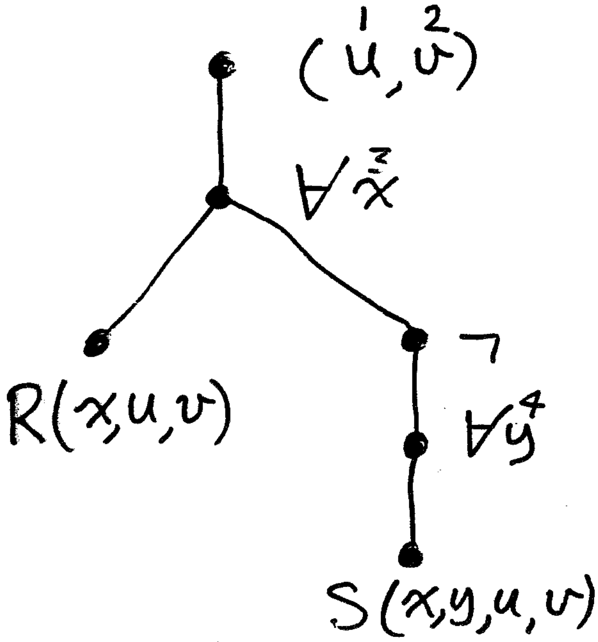
$[2]\ (=(0\,0)), (1), (1\,1)$

A variant of de Bruijn numbers

e.g. $(u,v)\,\forall x\,(R(x,u,v) \supset \neg \forall y\, S(x,y,u,v))$



$(\ ,\ )\forall(R(1,3,2) \supset \neg\forall S(2,1,4,3$



$2\forall(R(3,1,2) \supset \neg\forall S(3,4,1,2))$

Each expansion Ω' of a signature Ω determines a many-sorted algebra

$$A = \left((A_n)_{n\geq 0},\ (C^m_n)_{m,n\geq 0},\ (U^m_i)_{1\leq i\leq m}\ (F^A_k)_{F\in\Omega,\ k\geq 0} \right)$$

where

- A_n is the set of closed expressions of arity $[n]$ for Ω' (up to α-convertibility)
- $C^m_n : A_m \times \overbrace{A_n \times \dots \times A_n}^{m} \longrightarrow A_n$

  with $C^m_n(f, g_1, \dots, g_m) = f\circ(g_1,\dots,g_m)$

  $= (y_1,\dots,y_n)f(g_1(y_1,\dots,y_n),\dots,g_m(y_1,\dots,y_n$
- $U^m_i = (x_1,\dots,x_m)\,x_i \in A_m$
- $F^A_k : A_{k+k_1} \times \dots \times A_{k+k_n} \rightarrow A_k$ $\boxed{F\in\Omega \text{ of arity } (k_1 \dots k_n)}$

  with $F^A_k(f_1,\dots,f_n) =$

  $(x_1,\dots,x_k)F((y_1,\dots,y_{k_1})\,f_1(x_1,\dots,x_k,y_1,\dots,y_{k_1}),$

  $\dots\ (y_1,\dots,y_{k_n})f_n(x_1,\dots,x_k,y_1,\dots,y_{k_n}))$

e.g. $\forall_k(f) = (x_1,\dots,x_k)(\forall x)\,f(x_1,\dots,x_k,x$

Forms of Judgement

assuming $a_1,\dots,a_n$ true

$$a_1,\dots,a_n : a \qquad \text{a prop}$$

$$a_1,\dots,a_n \vdash a \qquad \text{a true}$$

$a_1,\dots,a_n, a$ are object expressions

Axioms and rules of Inference

will be written $\dfrac{\begin{array}{c}J_1\\ \vdots\\ J_k\end{array}}{J}$

and parametric assumptions will be left out in the usual natural deduction style

e.g. $$\frac{\begin{array}{c}:a\\ a\vdash b\end{array}}{\vdash a\supset b}$$

rather than $$\frac{\begin{array}{c}a_1,\dots,a_n : a\\ a_1,\dots,a_n, a\vdash b\end{array}}{a_1,\dots,a_n \vdash a\supset b}$$

Substitution rule

$$\frac{J}{J[f_1,\dots,f_n / X_1^{k_1},\dots,X_n^{k_n}]}$$

If $f_1,\dots,f_n$ have arities $k_1,\dots,k_n$ respectively

Assumption rules

$$\frac{:a}{a \vdash a} \qquad \frac{\begin{array}{c}:a\\ :b\end{array}}{a : b} \qquad \frac{\begin{array}{c}:a\\ \vdash b\end{array}}{a \vdash b}$$

$\sqsupset$- rules

($\sqsupset$ constant of arity (11))
f, g expressions of arity $[1]$

$$\frac{\begin{array}{c}: f(x)\\ f(x) : g(x)\end{array}}{: f \sqsupset g} \qquad \frac{\begin{array}{c}: f(x)\\ f(x) \vdash g(x)\end{array}}{\vdash f \sqsupset g}$$

x not free in f, g or any parametric assumption.

$$\frac{\begin{array}{c}\vdash f \sqsupset g\\ \vdash f(a)\end{array}}{\vdash g(a)}$$

$a \underset{x}{\supset} b \underset{def}{\equiv} (x)a \sqsupset (x)b$

$a \supset b \underset{def}{\equiv} a \underset{x}{\supset} b$ where x is not free in a, b

$(\forall x)a \underset{def}{\equiv} T \underset{x}{\supset} a$

i.e. $\forall \underset{def}{\equiv} (X^1)((x)T \sqsupset X^1)$

T- rules

$: T \qquad \vdash T$

Alternative to $\sqsupset$

$\supset$-rules

$$\frac{\begin{matrix}:a\\ a:b\end{matrix}}{:a\supset b} \qquad \frac{\begin{matrix}:a\\ a\vdash b\end{matrix}}{\vdash(a\supset b)} \qquad \frac{\begin{matrix}\vdash a\supset b\\ \vdash a\end{matrix}}{\vdash b}$$

$\forall$-rules

$$\frac{:f(x)}{:\forall(f)} \qquad \frac{\vdash f(x)}{\vdash\forall(f)}$$

f of arity [1]
x not free in f or in any parametric assumption.

$$\frac{\vdash\forall(f)}{\vdash f(a)}$$

$$\boxed{f\sqsupset g \underset{\mathrm{def}}{=} \forall x(f(x)\supset g(x))}$$

Could also have $\bot$, &, $\vee$, $\exists$, $=$.

Can also have an evaluation relation $\Rightarrow$

$a\Rightarrow b$ means a has canonical value b

$:x\Rightarrow y \qquad \vdash\lambda(f)\Rightarrow\lambda(f)$

ap const of arity [2]
λ constant of arity (1)

$$\frac{\begin{matrix}\vdash e\Rightarrow\lambda(f)\\ \vdash f(a)\Rightarrow c\end{matrix}}{\vdash ap(e,a)\Rightarrow c}$$ etc...

f of arity [1]

Abreviations

f type	$: f(x)$
$a \in f$	$\vdash f(a)$
$g \in f_1 \to \dots \to f_n \to f$	$\vdash f_1(x_1) \supset f_2(x_2) \supset \dots \supset f_n(x_n) \supset f(g(x_1, \dots, x_n))$

Representation of formal arithmetic

Signature

Symbols	arity
0	$[0]$
$\mathcal{J}, \mathcal{F}, Tr, \neg$	$[1]$
$+, \cdot, =, \&, \vee, \supset$	$[2]$
$\forall, \exists$	(1)

$\mathcal{J}, \mathcal{F}, Tr$ types

$\mathcal{J}$ type of terms
$\mathcal{F}$ type of formulae
Tr type of theorems

<u>Term Formation</u>

$0 \in \mathcal{J}$
$S \in \mathcal{J} \to \mathcal{J}$
$+, \cdot \in \mathcal{J} \to \mathcal{J} \to \mathcal{J}$

<u>Formulae Formation</u>

$= \in \mathcal{J} \to \mathcal{J} \to \mathcal{F}$
$\neg \in \mathcal{F} \to \mathcal{F}$
$\&, \vee, \supset \in \mathcal{F} \to \mathcal{F} \to \mathcal{F}$

$(\forall, \exists$
$\equiv$

$\vdash (\mathcal{J}(x) \underset{x}{\supset} \mathcal{F}(X(x)) \supset \mathcal{F}(\underset{\exists}{\forall} x X(x))$

Logic

$\vdash Tr(x) \supset Tr(y) \supset Tr(x \,\&\, y)$

& $\vdash Tr(x \,\&\, y) \supset Tr(x)$

$\vdash Tr(x \,\&\, y) \supset Tr(y)$

$\vdash \mathcal{F}(x) \supset \mathcal{F}(y) \supset (Tr(x) \supset Tr(y)) \supset Tr(x \supset y)$

$\supset$ $\vdash Tr(x \supset y) \supset Tr(x) \supset Tr(y)$

$\vdash Tr(x) \supset \mathcal{F}(y) \supset Tr(x \vee y)$

$\vdash \mathcal{F}(x) \supset Tr(y) \supset Tr(x \vee y)$

$\vdash Tr(x \vee y) \supset \mathcal{F}(z) \supset (Tr(x) \supset Tr(z))$

$\vee$ $\quad \supset (Tr(y) \supset Tr(z)) \supset Tr(z)$

$\neg$ omitted

$\vdash (\mathcal{J}(x) \underset{x}{\supset} Tr(X(x)) \supset Tr(\forall x X(x))$

$\forall$ $\vdash Tr(\forall x X(x)) \supset (\mathcal{J}(x) \underset{x}{\supset} Tr(X(x)))$

$\vdash (\mathcal{J}(x) \underset{x}{\supset} \mathcal{F}(X(x))) \supset (\mathcal{J}(x) \underset{x}{\supset} Tr(X(x)) \supset Tr(\exists x X(x)$

$\exists$ $\vdash Tr(\exists x X(x)) \supset \mathcal{F}(y) \supset (Tr(X(x)) \underset{x}{\supset} Tr(y)) \supset Tr(y$

$\vdash Tr(\forall x (x = x))$

$=$ $\vdash (\mathcal{J}(x) \underset{x}{\supset} \mathcal{F}(X(x))) \supset Tr(\forall x \forall y (x = y \supset (X(x) \supset$
$\quad X(y))$

Non-logical

$$\vdash Tr(\forall x \forall y\, (S(x)=S(y) \supset x=y))$$

$$\vdash Tr(\forall x \neg\, S(x)=0)$$

$$\vdash Tr(\forall x \;\; x+0=x)$$

$$\vdash Tr(\forall x \forall y\; (x+S(y) = S(x+y)))$$

similarily for $\cdot$

$$\vdash (J(x) \underset{x}{\supset} \mathcal{F}(X(x))) \supset Tr(X(0))$$

$$\supset (J(x) \underset{x}{\supset} (Tr(X(x)) \supset Tr(X(S(x)))))$$

$$\supset Tr(\forall x\, X(x))$$

De Bruijn

AUTOMATH (1967)

LANGUAGE FOR MATHEMATICS

WRITING }
MACHINE CHECKING } FEASIBLE

6

UNLIKE STANDARD LOGIC

AT THAT TIME, THE AUTOMATH PROJECT TOOK ABBREVIATIONS SERIOUSLY

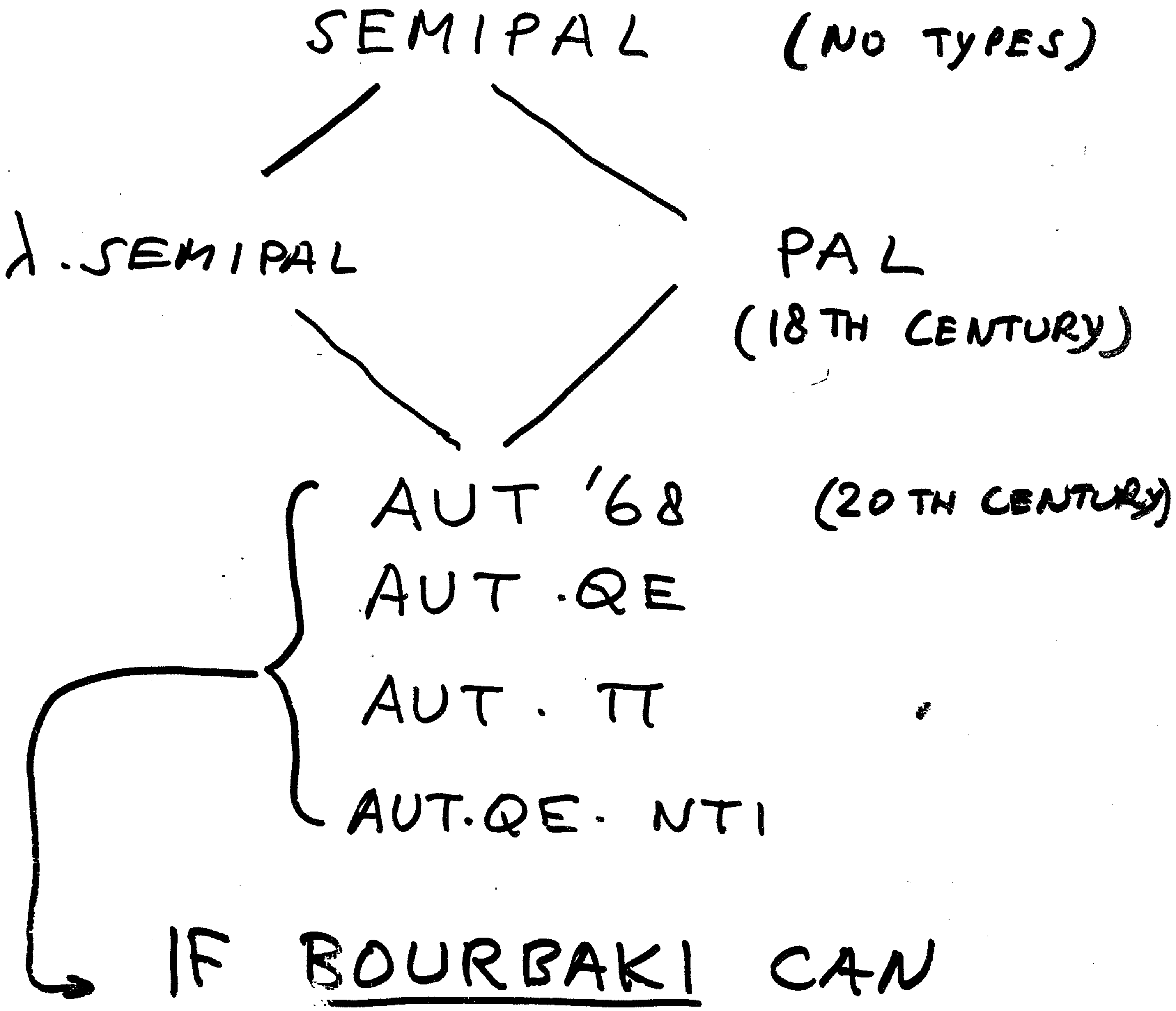
THESE BASIC REQUIREMENTS

FORCED US INTO

INVENTING THINGS LIKE:

- PROPOSITIONS AS TYPES.
- NATURAL DEDUCTION (EXTENDED WITH OBJECT VARIABLES)
- DEPTH REFERENCE SYSTEM
- λ-TYPED λ-CALCULUS

THE AUT FAMILY



SEMIPAL (NO TYPES)

λ.SEMIPAL

PAL (18TH CENTURY)

AUT '68 (20TH CENTURY)

AUT.QE

AUT.π

AUT.QE.NTI

IF *BOURBAKI* CAN WRITE IT, THEN *WE* CAN WRITE IT AND OUR CHECKER CAN CHECK IT

NO ATTEMPT TO MAKE PROOF FINDING AUTOMATIC

APART FROM TYPE CHECKS, THAT IS A KIND OF PROVING ACTIVITY TOO!

PHILOSOPHY: PROOF FINDING IS A HARD JOB. IT DESERVES SERIOUS PROFESSIONAL EFFORTS.

BUT THE EFFICIENCY REQUIREMENTS OF PROOF FINDING SHOULD NEVER INFLUENCE OUR CHECKING SYSTEM

SOME CHARACTERISTICS

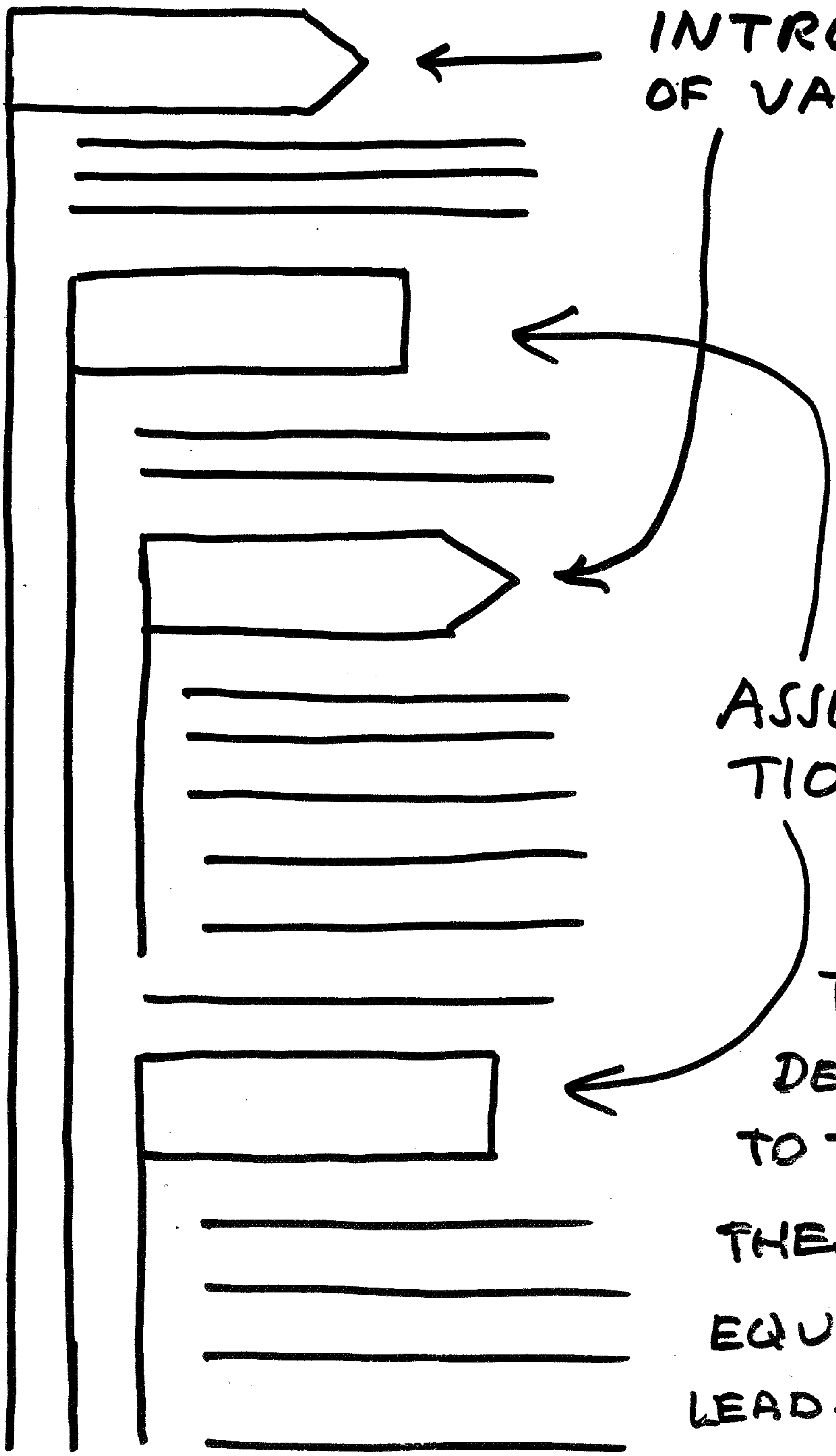
NOT TIED TO STANDARD LOGIC AND SET THEORY.

NEW LOGICAL INFERENCE RULES CAN BE DERIVED AND APPLIED JUST LIKE MATH. THEOREMS

THE AUTOMATH SYSTEM CAN HANDLE THINGS WHICH ARE USUALLY NOT CONSIDERED TO BE MATHEMATICS, BUT WHICH NEVERTHELESS MAKE SENSE

ALL MATH STUDENTS CAN LEARN TO HANDLE THE SYSTEM

LAYOUT OF AUT BOOKS



INTROD. OF VARIABLE

ASSUMP-TIONS

THE DESIRE TO TREAT THESE EQUALLY, LEADS TO

<u>PROPOSITIONS</u> AS TYPES

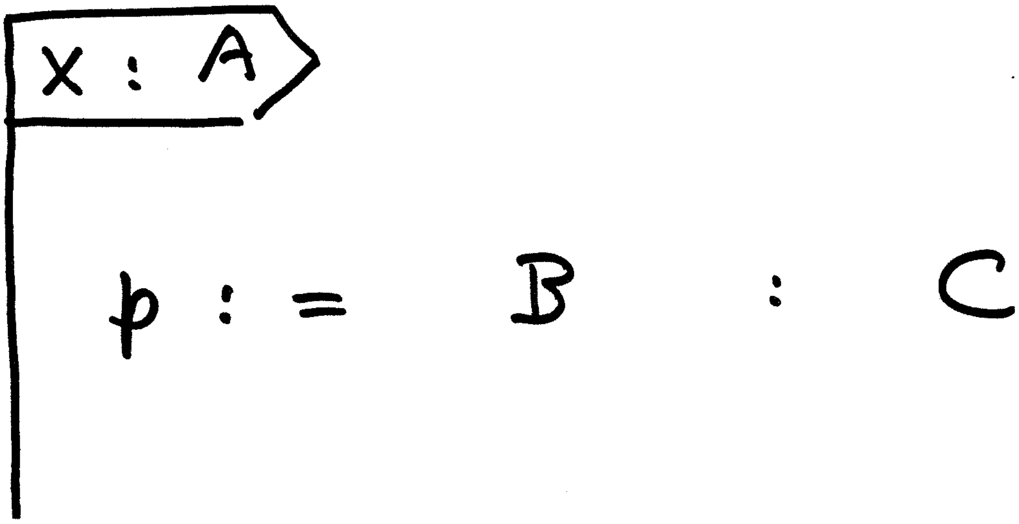
AUT LINES

<CONTEXT> <IDENTIFIER>

<DEFINITION> <TYPE>

BY λ. ABSTRACTION WE CAN (AT LEAST IN SOME OF THE AUT LANGUAGES) REPLACE THE LINE BY AN EQUIVALENT ONE, WITH EMPTY CONTEXT

EXAMPLE



$x : A$

$p := B \quad : \quad C$

EQUIVALENT TO

$$q := [x : A]\, B \quad : \quad [x : A]\, C$$

ANALOGOUSLY WITH 'PRIMITIVE' LINES

NEXT STEP :

ELIMINATE ALL
DEFINITIONAL LINES

LEADS TO **AUT.SL**

SINGLE - LINE AUTOMATH

THIS IS USEFUL FOR
LANGUAGE THEORY
(NEDERPELT 1973).

NOT FOR THE PRACTICE OF
WRITING MATHEMATICS.

AUT BOOK AS SINGLE TREE
λ - TYPED λ - TREE
SYSTEM Λ

$\Delta\Lambda$ (1985)

TREES REPRESENTING AUT. BOOKS IN ALL THEIR DETAILS

DIFFERENCE WITH Λ: GETTING FROM AUT. BOOK TO Λ-TREE, WE FIRST ELIMINATE ALL DEFINITIONAL LINES.

PASSING FROM AUT BOOK TO $\Delta\Lambda$ TREE, WE DO NOT.

DELTA REDUCTION

IN $\ldots\ldots, f(A), \ldots$

WE REPLACE f BY ITS DEFINITION. IF THAT WAS

$x : T$

$f := \ldots\ x\ \ldots\ x\ ..\ :\ \ldots$

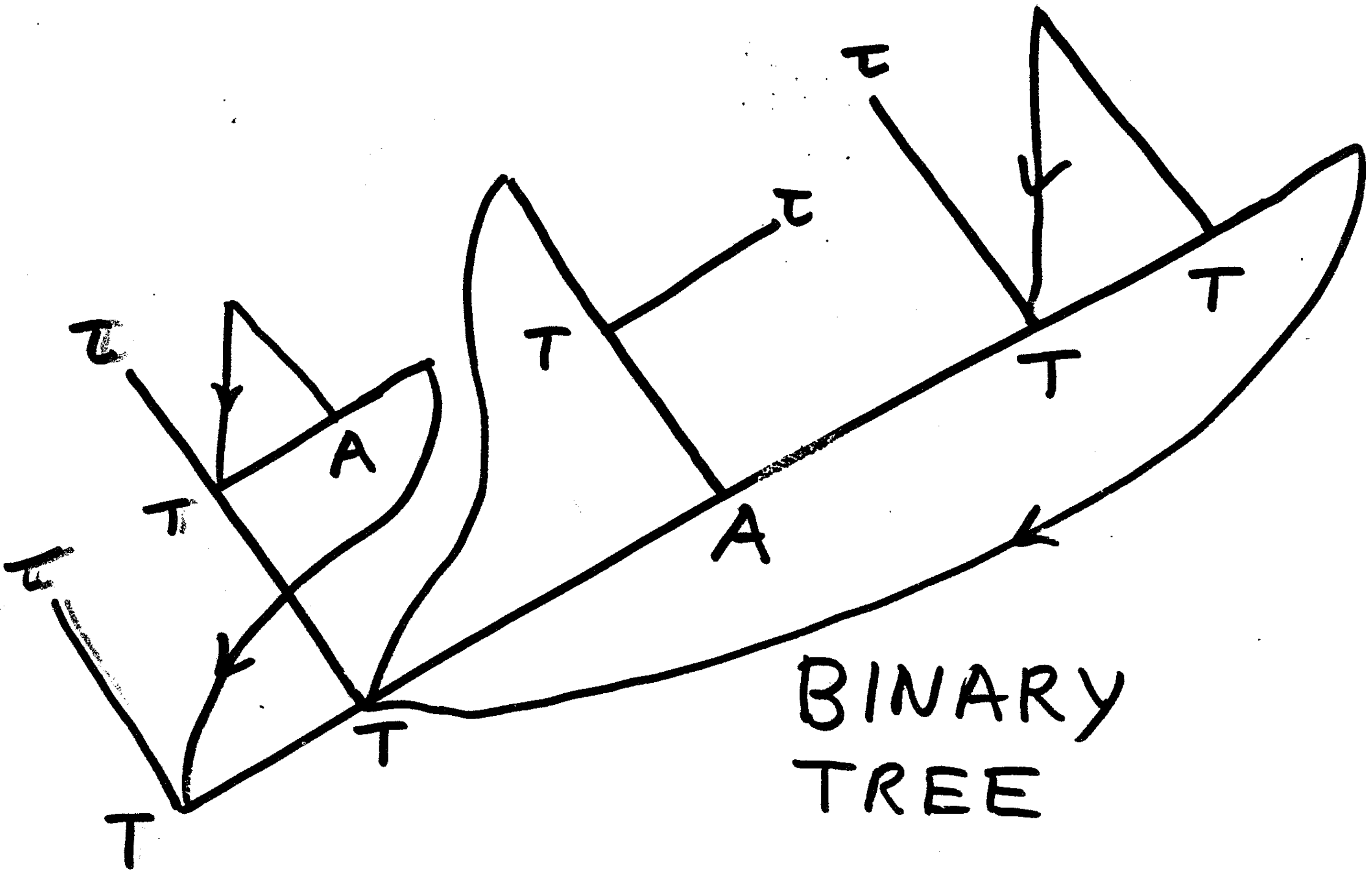
WE REPLACE

$\ldots, f(A), \ldots$

by

$\ldots, \ldots A \ldots A ..$

LAMDA TREE



BINARY
TREE

BINARY NODES

LABELED A OR T

ENDPOINTS

τ OR VARIABLE

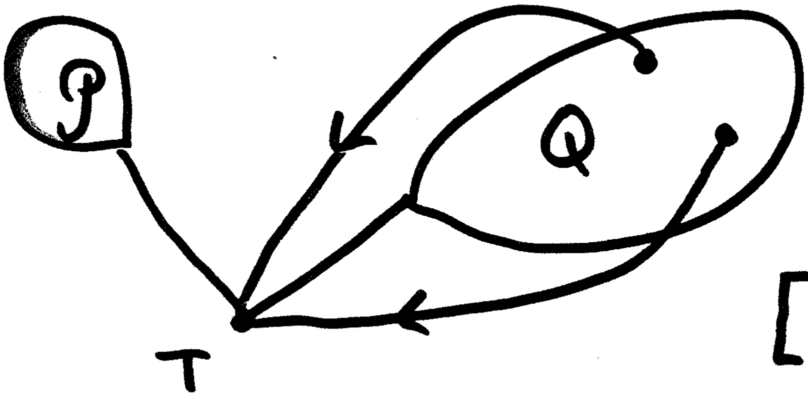
NO FREE VARIABLES
NO CONSTANTS APART
FROM τ

VARIABLES CAN BE INDICATED

1. BY NAME OF ENDPOINT OF ARROW

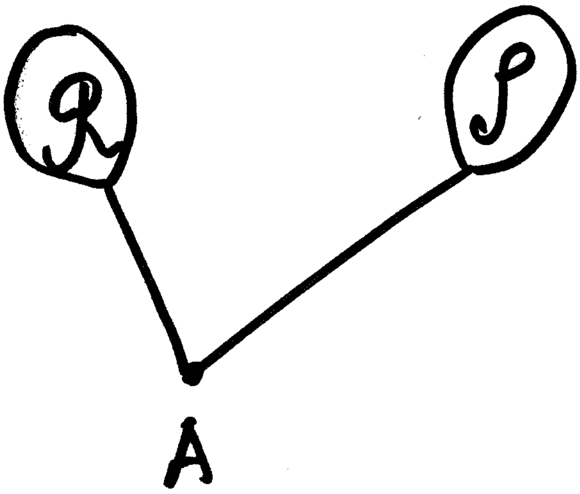
2. BY DEPTH REFERENCE

INTERPRETATION



$[x : \mathcal{P}]\ Q$

(DOTS REPLACED BY x)

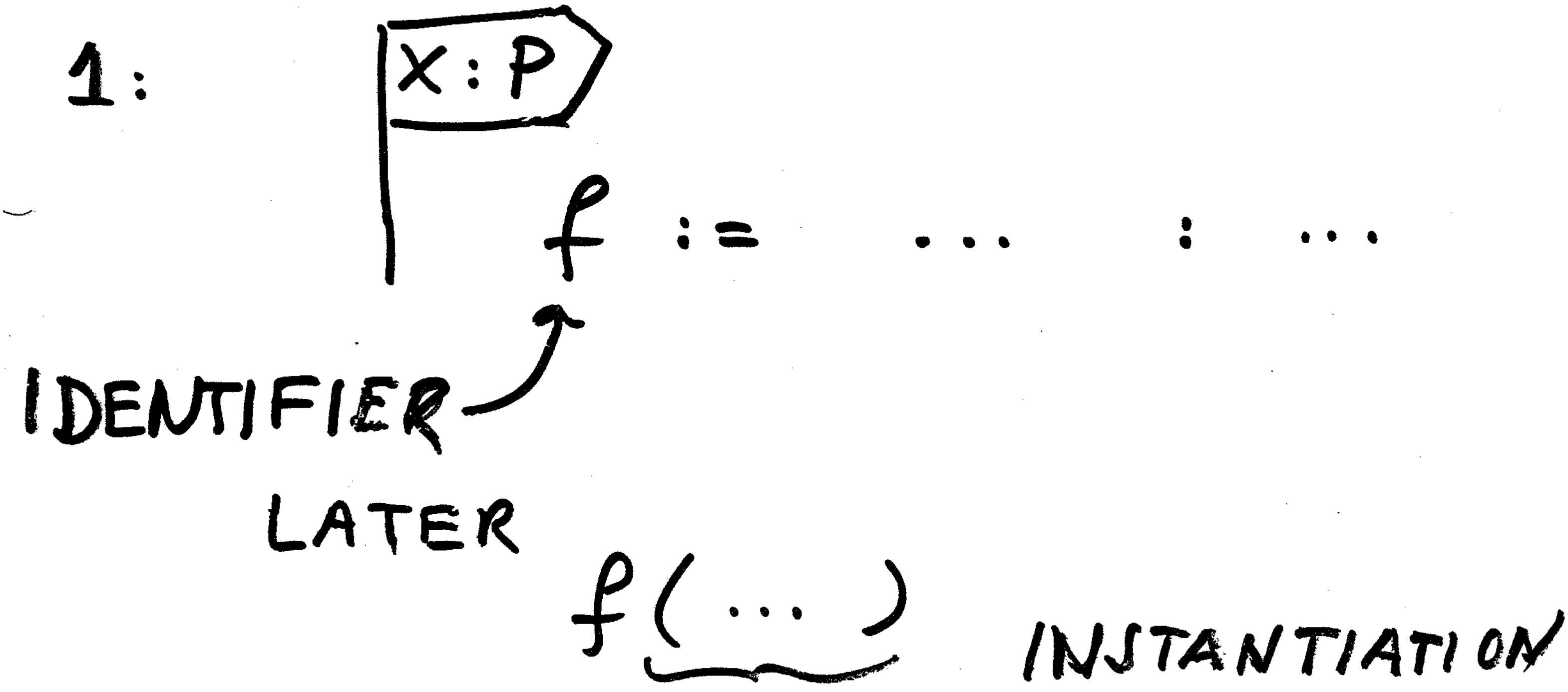


$\langle \mathcal{R} \rangle \mathcal{S}$

AUTOMATH

TWO WAYS TO TREAT FUNCTIONALITY

1:



$x : P$

$f := \dots : \dots$

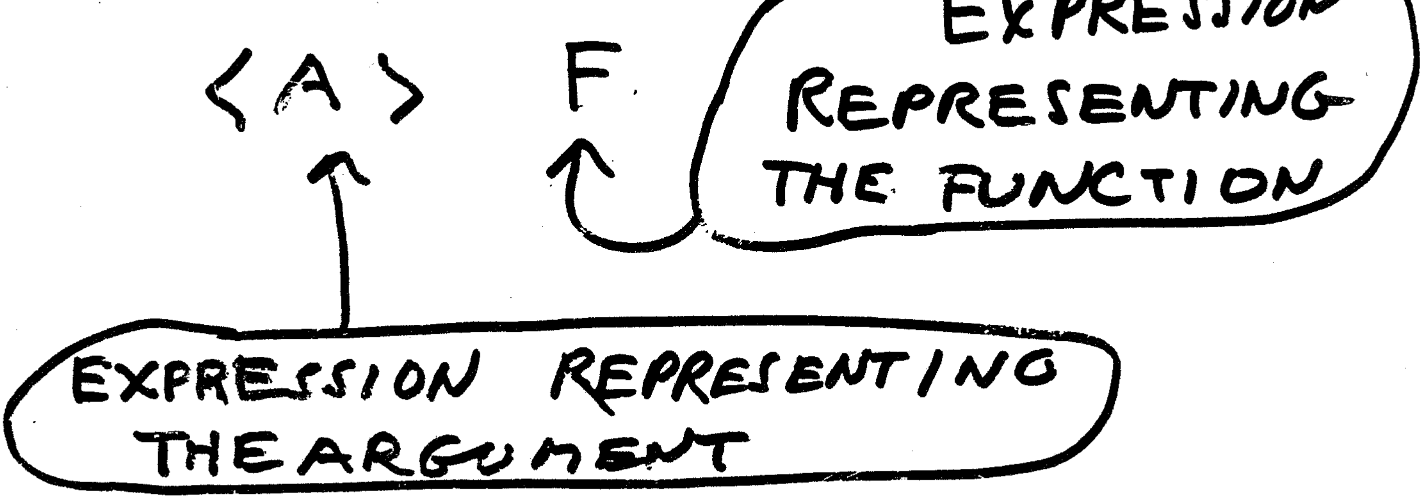
IDENTIFIER

LATER

$f(\underbrace{\dots})$

INSTANTIATION

2: THE 'APPLICATION' OF λ CALCULUS:



$\langle A \rangle \; F$

EXPRESSION REPRESENTING THE FUNCTION

EXPRESSION REPRESENTING THEARGUMENT

N THE TRANSITION

ROM AUT. BOOK TO

ΔΛ THE INSTANTIATIONS

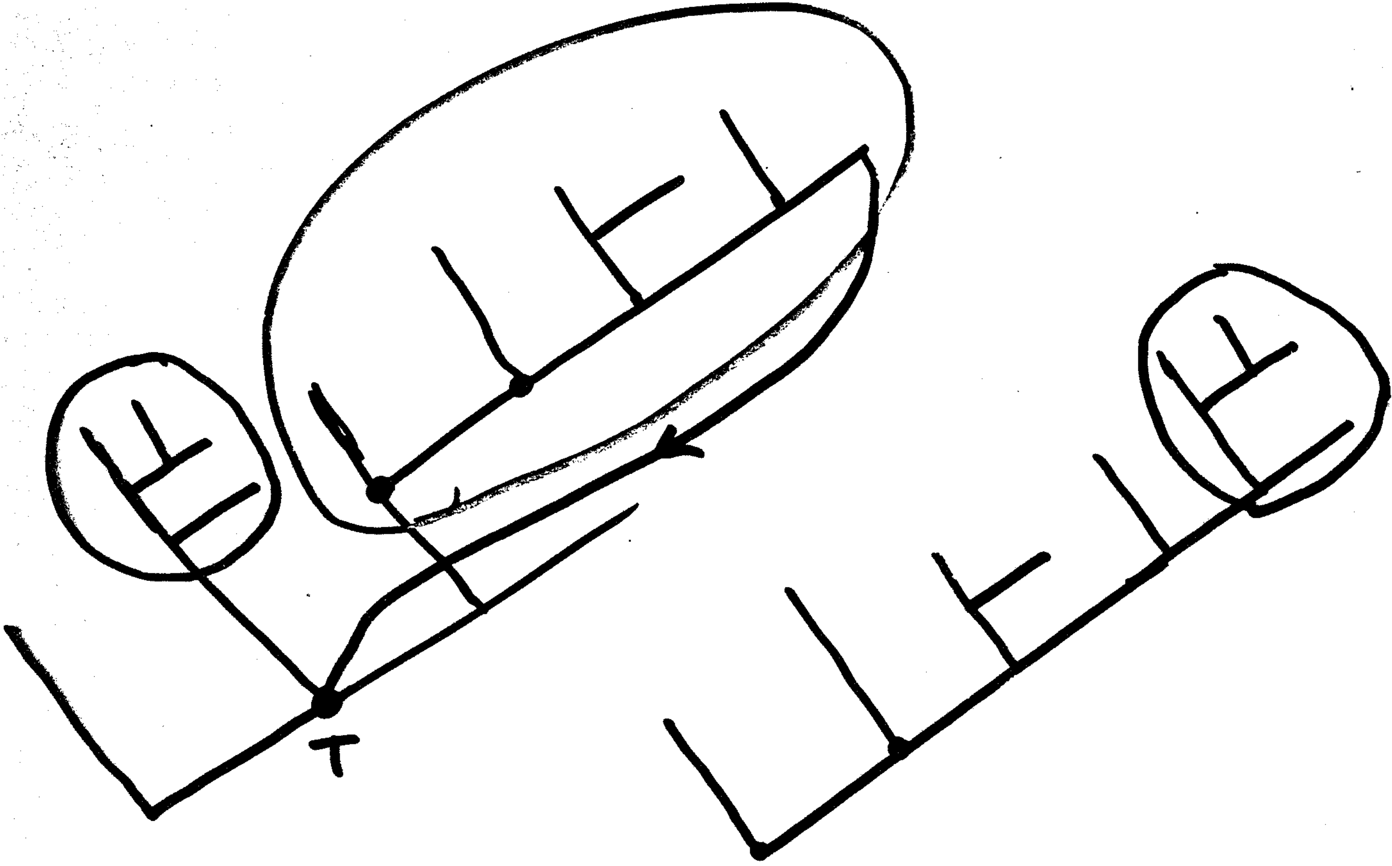
BECOME APPLICATIONS

DELTA REDUCTION

BECOMES

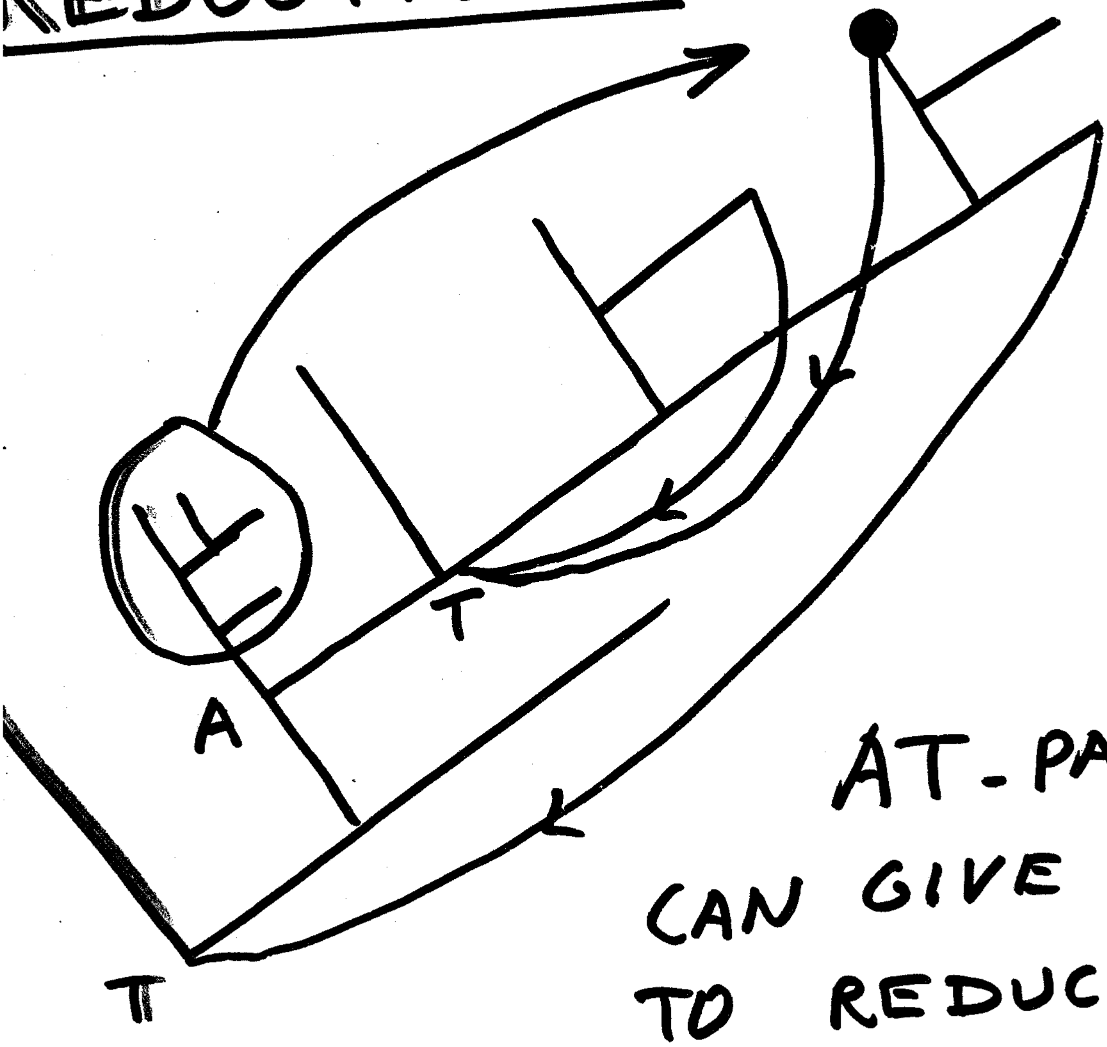
<u>LOCAL</u> <u>BETA</u> <u>REDUCTION</u>

TYPE OF A SUBTREE



REPLACEMENT OF
RIGHTMOST
END POINT
BY ITS TYPE

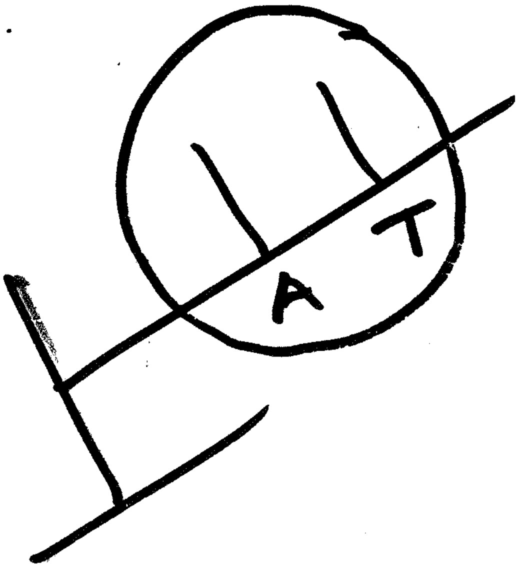
REDUCTIONS



AT-PAIR CAN GIVE RISE TO REDUCTION

(LOCAL BETA REDUCTION)

2)



CUT OUT AT. PAIR WHERE T HAS NO CUSTOMERS

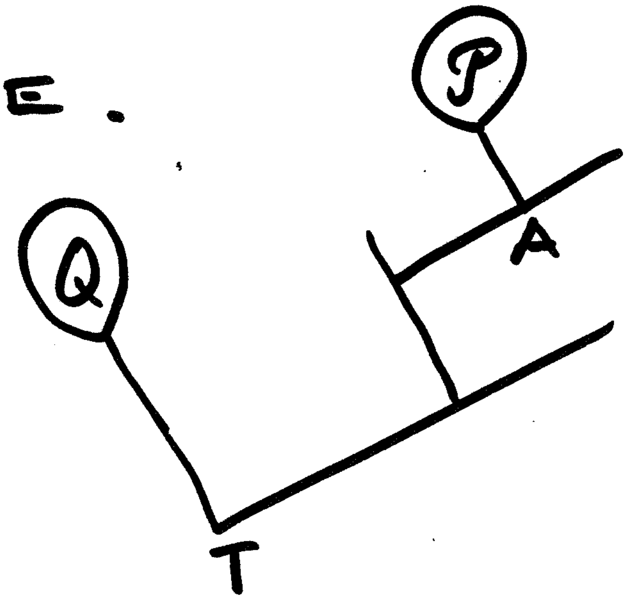
CORRECTNESS OF A LAMBDATREE:

1. SYNTACTIC CORRECTNESS
   A's, T's, τ's, green ARROWS

2. SEMICORRECTNESS

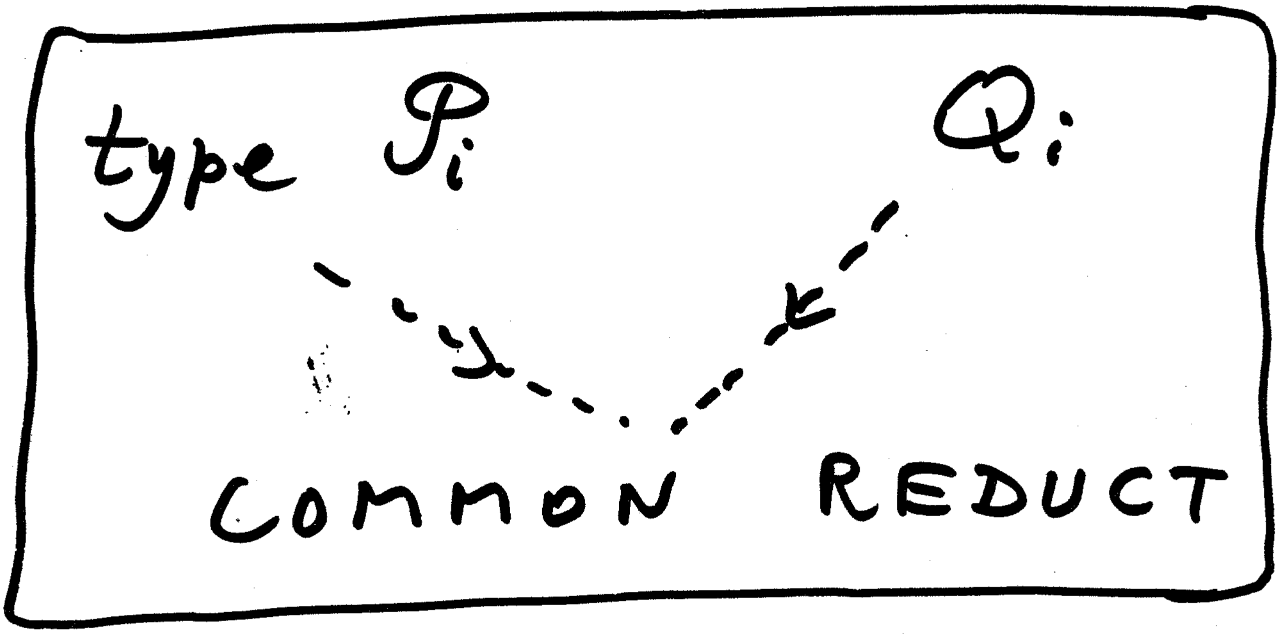
3. CORRECTNESS

IN SEMICORRECT TREE THERE IS TO EVERY A-NODE A MATCHING T-NODE.



TREE IS *CORRECT*

IFF FOR ALL i

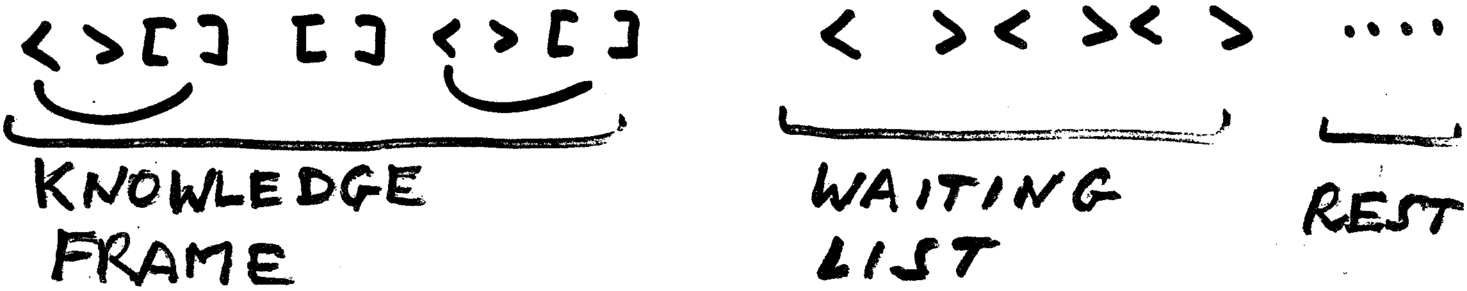


type P_i

Q_i

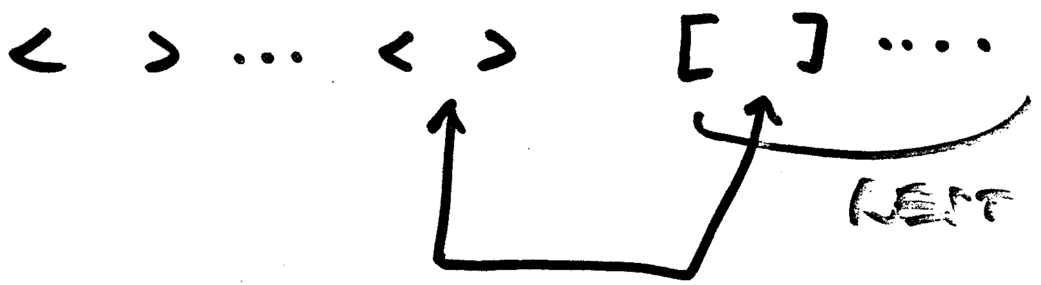
COMMON REDUCT

ROUGH SKETCH.



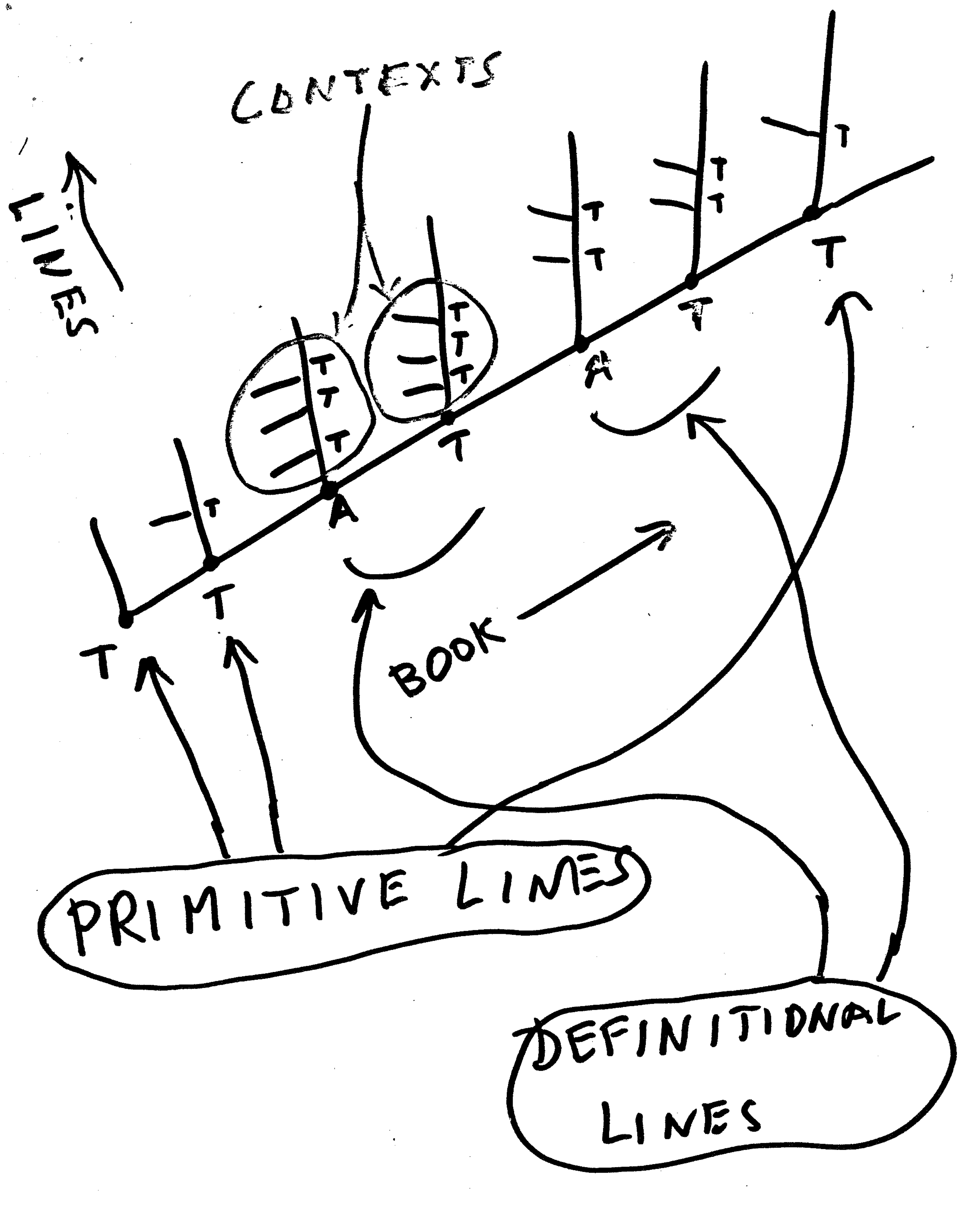
TRY TO GET THE FIRST ITEM OF "REST" TO THE LEFT.

1)

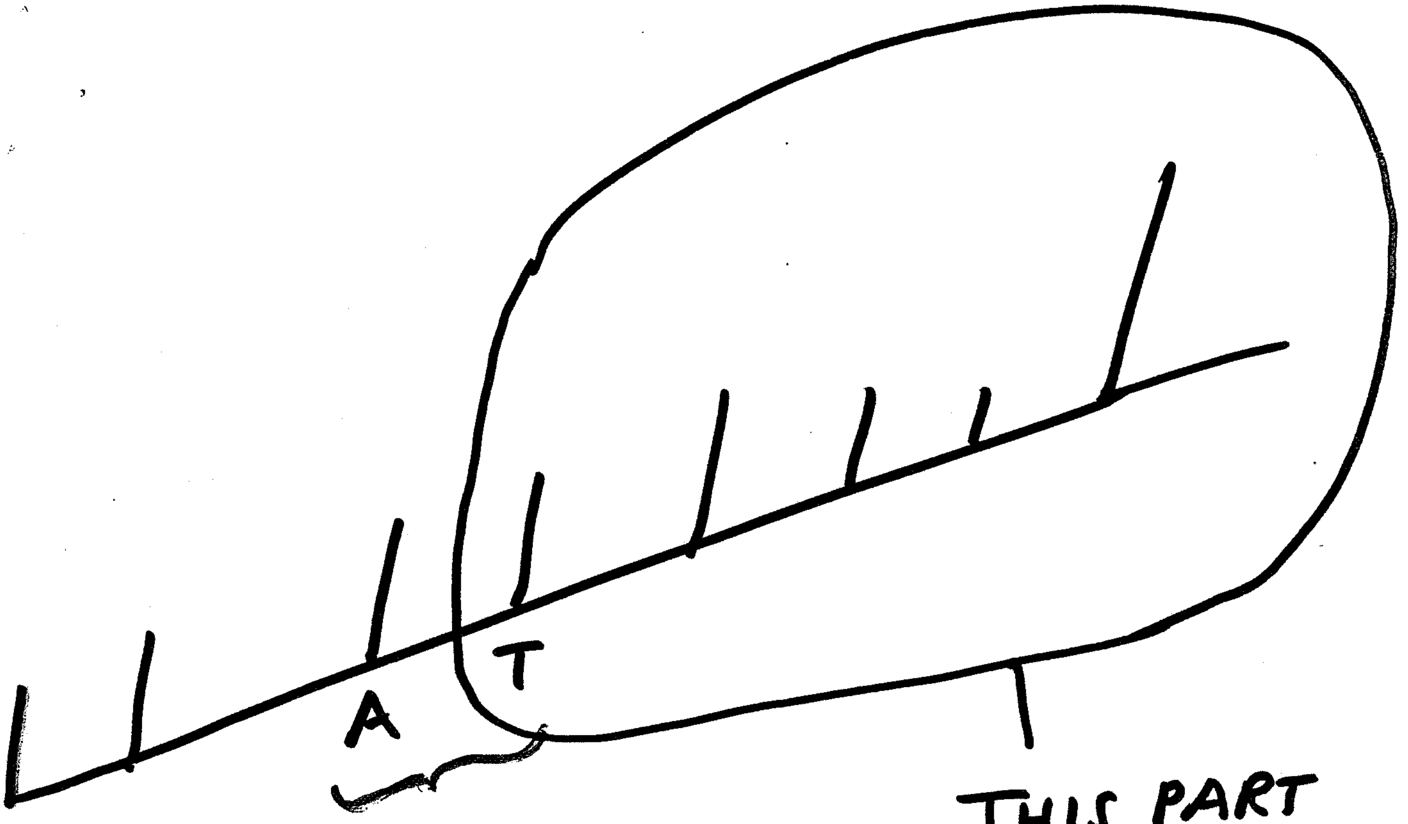


NOW PUT THIS PAIR IN KNOWLEDGE FRAME AFTER CHECKING THE SEMICORRECTNESS OF WHAT IS IN IT.

IF WAITING LIST EMPTY THEN JUST SHIFT [ ].



AUT BOOK AS LAMBDA TREE



DEFINITIONAL LINE

THIS PART NEEDNOT BE CORRECT BUT CAN BECOME CORRECT AFTER DELTA REDUCTION

IT SEEMS TO BE WISE TO ADMIT THIS IN THE SIDE LINES TOO

THAT IS WHAT ΔΛ DOES

2. $\underbrace{\langle\ \rangle \cdots \langle\ \rangle}\quad \underbrace{\langle P \rangle} \cdots$

NOW CHECK P

AND SHIFT $\langle P \rangle$ TO THE LEFT

3. $\underbrace{\langle\ \rangle \cdots \langle\ \rangle}\quad \underbrace{\tau}$

SEMICORRECT $\underline{\text{IFF}}$ WAITING LIST IS EMPTY

4. $\underbrace{\langle\ \rangle \cdots \langle\ \rangle}\quad \underbrace{x}$

CORRECT IFF

$$\underbrace{\langle\ \rangle \cdots \langle\ \rangle}\quad \underbrace{\text{typ}\ x}$$

IS CORRECT.

THERE IS A <u>SIMPLE</u> ALGORITHM WITH THE FOLLOWING EFFECT:

INPUT: ANY SYNTACTIC CORRECT TREE

OUTPUT: **EITHER** SOME A-NODE WITHOUT MATCHING T.

OR FULL LIST OF n ($n =$ # of A's) PAIRS

$(\mathcal{P}_i , Q_i)$

ALGORITHM WORKS IN LINEAR TIME

REFERENCE :

N. G. de BRUIJN. GENERALIZING AUTOMATH BY MEANS OF A LAMBDA-TYPED LAMBDA CALCULUS.

IN:
MATH. LOGIC & THEORETICAL COMPUTER SCIENCE
LECTURE NOTES IN PURE AND APPLIED MATH.
MARCEL DEKKER,
NEW YORK - BASEL 1987

Simple Consequence Relations

Arnon Avron

February 20 1987

7

1 Introduction

This paper has several purposes. From a very general perspective it aims to help clarifying questions like: What is a logic? What is a formal inference system? What are the differences between the usual kinds of formal systems and what is common to them? Are they the only possible useful ones? What is so special about the usual connectives and how does one characterize them? etc. At a first glance these questions seem to have only (?) philosophical interest. Their practical importance is immediately realized, though, when one is trying to design a general framework for implementing systems of logic on a computer. This is exactly what the current Edinburgh LF-system (see [HHP]) is. The present paper resulted from an attempt to solve basic problems which were encountered while developing this system. Our point of view is therefore a completely practical one, and we do not claim to solve the deep problems that exist concerning the foundations of logic and the meanings of the logical constants.

Another purpose paper, intimately connected with the first, is to suggest new methods for representing logics, that might make search for proofs, proof checking and implementation easier. The new notions and representation methods introduced in it, as well as various characterizations of more familiar ones have already contributed to the further development of the LF system and I hope it will be of help for any future effort at the same direction.

Because of the general nature of the ideas discussed below it is very difficult to trace the origin of each of them or to give credit. Nevertheless, I like to mention at least the papers [Sc1], [Sc2] of Scott, in which the notion of a consequence relation (of the type dealt with here) was first introduced (as well as many other important ideas), [Ha], [Be] and above all — ([Gen]). Exactly like Hacking in [Ha], I see my paper just as a collection of footnotes to this brilliant work of Gentzen.

2 The Notion of a Consequence Relation

2.1 Axiomatic Systems

Traditionally a "formal system" is understood to include the following components:

1. A formal language L with several syntactic categories, one of which is the category of "well formed formulae " (wff).
2. An effective set of wffs called "axioms".
3. An effective set of rules (called "inference rules") for deriving theorems from the axioms.

The set of "theorems" is usually taken to be the minimal set of wffs which includes all the axioms and is closed under the rules of inference.

Systems of this sort have many names in the literature. Here we shall call them *Axiomatic systems* (or, sometimes: Hilbert-systems for theoremhood). Undoubtly they constitute the most basic kind of formal system and so their importance cannot be denied. One can argue that in fact all other, more complicated deduction formalisms reduce to systems of this sort. This is true, though, for *every* recursively-defined system. Take for example the wffs in the propositional calculus. One can regard them as the "theorems" of the axiomatic system in which the "wffs" are strings of symbols, the "axioms" are the propositional variables and the "inference rules"- the usual formation rules.[1] The concept of theoremhood in systems of the above sort is not sufficient, therefore, to characterize the notion of a *Logic*. It is too broad a concept. On the other hand the notion of theoremhood of wffs is at the same time also too narrow to characterize what a logical system concerning these wffs is all about.

Let as make our last point clearer by a few very simple examples. Take what is known as Kleene's 3-valued logic. It has 3 "truth-values": 1,0 and -1, of which 1 is taken as the only designated one. The operations corresponding to the usual connectives are: $\neg a = -a, a \vee b = max(a, b), a \wedge b = min(a, b)$. Suppose that L is the language of propositional calculus where the wffs are defined as usual. It is immediate then that *no* wff is a theorem of this logic (i.e. there is no wff that gets a designated value under all assignments). The notion of theoremhood seems to be vacuous for this logic. One might ask therefore in what sense it is a "logic".

On the other hand, consider the case in which we take both 1 and 0 to be designated. It is easy then to see that a wff is a theorem of the new logic iff it is a

[1] In some recent systems of typed constructive mathematics this resemblance is taken rather seriously and both "proposition" and "theorem" are taken as (different) "judgements" so that there is no significant difference between possible proofs of these "judgements"!

classical tautology. From the point of view of theoremhood there is no difference between this logic and the classical, two-valued one. But are they really the same? Obviously not, a major difference is, e.g. , that the "new" 3-valued logic is *paraconsistent*: It is possible for inconsistent theory to be non-trivial in this logic. (it is possible, e.g. , for p and $\neg p$ to be both "true" while q is "false").

2.2 Consequence Relations

Both examples above show that sets of "logical truths" are not enough for characterizing logics. The second example indicates that what is really important is what wffs follow from what theories. Indeed, in modern treatments of logic [2] another concept, that of a *consequence relation* (C.R.) is taken as the most fundamental concept of logic. Logic might be defined, in fact, as the science of consequence relations. Unfortunately, the notion of a C.R. has in the literature several (similar, but not identical) meanings. We shall define first the one which we are going to use here (which is rather general) and then discuss some possible reasonable variations.

Definition A *consequence relation* (C.R.) on a set $\sum$ of formulas is a binary relation $\vdash$ between finite multisets of formulas s.t:

(I) Reflexivity: $A \vdash A$ for every formula A.

(II) Transitivity, or "Cut": if $\Gamma_1 \vdash \Delta_1, A$ and $A, \Gamma_2 \vdash \Delta_2$ then $\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$

2.3 Remarks and variations

1. We use above the notion of a "multiset". By this we mean "sets" in which the number of times each element occurrs is significant, but not the order of the elements. Thus, for example, $[A, A, B] = [A, B, A] \neq [A, B]$. In this example we use $[\cdot]$ to denote a multiset. We shall also use "," for denoting the operation of multisets-union (so $[A, A, B], [A, B] = [A, A, B, A, B]$), and omit the "[ ]" whenever there is no danger of confusion.

   It is more customary to take a C.R. to be a relation between *sets*, rather then multisets. This is undoubtly more intuitive and so preferable wherever possible. We define, accordingly, a C.R. to be *regular* if it can be taken to be between sets. There are, however, logics the full understanding of which requires us to make finer distinctions that only the use of multisets enable us to make. Examples are: Relevance logics, Girard's Linear logic and the finite valued logics of Lukasiewicz (see examples below). It is possible, of course,

[2] See, e.g. [Sc1],[Ur],[Ga],[Ha].

to go one step further and to take C.R.'s to be between *sequences* of formulas (as Gentzen himself did). This, however, will considerably complicate the transitivity condition (II), and the need for it seems to be very rare indeed. We choose, therefore, not to be *so* general (at least in the present paper).

2. We define a C.R. to be a relation between *finite* (multi)sets. This means that we are assuming *compactness* for all the logics we consider. This rules out many "model-theoretic logics". Our excuse for this elimination is that we are primarily interested in formal systems that can (at least in principle) be computerized. Now effective rules with infinite number of premises *are* possible, but the amount of information needed for applying such rules in any particular case should always be finite. Accordingly, we believe that any effective presentation of such rules should be within the scope of our framework. This issue is opened to further investigations, though!

3. In most definitions of a C.R. that one can find in the literature there is a third condition besides the two formulated above. This is the *weakening* condition, according to which if $\Gamma \vdash \Delta$ then also (i) $\Theta, \Gamma \vdash \Delta$ and (ii) $\Gamma \vdash \Delta, \Theta$. (Some times only condition (i) is demanded, especially when one is interested only in a single-conclusioned C.R., i.e. if $\Gamma \vdash \Delta$ then Δ consists of a single formula). Again, this is a very natural restriction but it fails for many systems. In fact it fails for every system for non-monotonic reasoning, as well as some C.R. based on Relevance logics and Linear Logic.

   In the sequel, a C.R. which is regular and is closed under weakening will be called *ordinary*.

4. The above definition of a C.R should more accurately be taken as a definition of a *simple* C.R.. In [Ga] there is another requirement: *uniformity*. This means that $\vdash$ should be "closed under substitutions". This condition involves the inner structure of wffs and so is a little bit vague. Although the meaning of it is quite obvious in particular cases, it is less obvious how to define it in general. One might ask: substitutions of what for what? In order to provide a precise definition (and for other reasons as well) one should define a C.R. to be a *ternary* relation $\Gamma \vdash_{\vec{x}} \Delta$ where Γ and Δ are as before and $\vec{x}$ is a finite set of *variables of the language*. (It is assumed, therefore, that some of the syntactic categories of the language include special subcategories of variables for these categories). For example: $\forall x\phi \vdash_{\phi,y} \phi(y/x)$ intuitively means that for every formula ϕ and any individual term t which is free for x in ϕ, $\phi(t/x)$ follows from $\forall x\phi$. The use of this general notion of a C.R. requires extending the cut condition to some version of resolution (i.e.: unification should be incorporated), and it involves delicate problems

concerning substitutions. We prefer therefore to postpone treating these problems to the second part of this paper, and here we treat only simple C.R's.

3 Some Examples of Abstract Consequence Relations

3.1 Classical Propositional Logic

Truth: [3] $A_1, \ldots, A_n \vdash_t B_1, \ldots, B_m$ iff for every valuation which makes all the A_i's true makes one of the B_j's true as well.

Validity: [4] $A_1, \ldots, A_n \vdash_v B_1, \ldots, B_m$ iff any substitution of sentences for atomic propositional sentences which makes all the A_i's tautologies does the same to one of the B_i's.

Reduction: $A_1, \ldots, A_n \vdash_r B_1, \ldots, B_m$ iff $A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B_1 \vee B_2 \vee \cdots \vee B_m$ is a tautology.

If we limit ourselves above to the case $m = 1$ we get the single-conclusioned counterparts of these three C.R.'s. It is not difficult to see that in this case all three are in fact identical. In the multiple-conclusioned case, however, the "validity" C.R. differs from the other two, and is the *minimal* C.R. which extends the single-conclusioned one.

3.2 First-order logic

Let $A_1, \ldots, A_n$ and B be formulas of some first-order language L (i.e. they may contain free variables).

Truth: $A_1, \ldots, A_n \vdash_t B$ iff B is true in every model relative to any assignment which makes all the A_i's true.

Validity: $A_1, \ldots, A_n \vdash_v B$ iff B is valid in any model (for L) in which all the A_i's are valid (by "valid" we mean: true relative to all assignments).

The above are examples of two important *single-conclusioned* C.R.'s which are frequently associated with first order logic. Unlike the propositional case, they are *not* identical. $\forall x p(x)$ follows, for example, from $p(x)$ according to the second, but not according to the first. On the other hand, the classical deduction theorem

[3] the name "Truth-functional" might, perhaps, be better.

[4] the name "Tautological" is also possible.

holds for the first but not for the second. The two consequence relations *are* identical, though, from the point of view of *theoremhood*: $\vdash_v A$ iff $\vdash_t A$ (and in fact if all formulas of Γ are closed then $\Gamma \vdash_t A$ iff $\Gamma \vdash_v A$). Both C.R.'s defined above can be extended to multiple-conclusioned C.R.'s in more than one interesting way, but we need not go into the details.

3.3 Propositional Modal Logic

Truth: $A_1, \ldots, A_n \vdash_t B$ iff for any frame and for any valuation in this frame, B is *true* in every *world* in this frame in which all the A_i's are true.

Validity: $A_1, \ldots, A_n \vdash_v B$ iff B is valid (i.e. true in all worlds) in every frame relative to any valuation which makes all the A_i's valid.

Again we consider here two important single-conclusioned C.R.'s. The situation concerning them is similar to that in the previous case: $A \vdash_v \Box A$ but $A \not\vdash_t \Box A$. The deduction theorem obtains for $\vdash_t$ but not for $\vdash_v$. Again the two C.R.'s are identical as far as *theorems* are concerned.

3.4 Three-Valued Logic

Assume again a propositional language with the connectives $\neg, \vee, \wedge$. Let corresponding operations on the truth-values {-1,0,1} be defined as in section 2.1. We define now 5 different C.R.'s based on the resulting structure. In these definitions v denotes an assignment of truth values to formulas which respects the operations, $\Gamma = A_1, \ldots, A_m$ and $\Delta = B_1, \ldots, B_n$.

Kl: $\Gamma \vdash_{Kl} \Delta$ iff $v(B_i) = 1$ for some i or $v(A_j) \in \{-1, 0\}$ for some j.

Pac: $\Gamma \vdash_{Pac} \Delta$ iff either $v(B_i) \in \{1, 0\}$ for some i or $v(A_j) = -1$ for some j.

Lt: $\Gamma \vdash_{Lt} \Delta$ iff for some i,j $v(B_i) = 1$ or $v(A_j) = -1$ or $v(B_i) = v(A_j) = 0$.

Sob: $\Gamma \vdash_{Sob} \Delta$ iff either $v(B_i) = 1$ for some i or $v(A_j) = -1$ for some j or $v(A_i) = v(B_j) = 0$ for *all* i, j.

Luk: $\Gamma \vdash_{Luk} \Delta$ iff either $v(B_i) = 1$ for some i or $v(A_j) = -1$ for some j or at least *two* formulas in Γ, Δ get 0 (under v).

Notes:

1. $\vdash_{Kl}$ corresponds to taking 1 as the only designated value, and so it is the obvious C.R. defined dy Kleene's 3-valued logic. As remarked above, it has no theorems.

2. $\vdash_{Pac}$ corresponds to taking both 1 and 0 as designated. As noted above, it has the same set of theorems as classical propositional calculus, but it is *paraconsistent* ($P, \neg P \not\vdash_{Pac} Q$).

3. $A_1, \ldots, A_m \vdash_{Lt} B_1, \ldots, B_n$ iff for every v, $v(A_1 \wedge A_2 \wedge \ldots \wedge A_m) \leq v(B_1 \vee B_2 \vee \ldots \vee B_n)$. This C.R. also has no theorems. In fact if $\Gamma \vdash_{Lt} \Delta$ then both Γ and Δ are none-empty.

4. $A_1, \ldots, A_n \vdash_{Sob} B$ iff $A_1 \to (A_2 \to \ldots \to (A_n \to B) \ldots)$ is valid when $\to$ is defined as in Sobociński 3-valued logic, (See [Sob] or [AB], pp. 148-9). Moreover, $A_1, \ldots A_m \vdash_{Sob} B_1, \ldots B_n$ iff $\vdash_{RM_3} A_1 \to (A_2 \to \ldots \to (A_m \to (B_1 + \ldots + B_n)) \ldots)$ [5] , where RM_3 (the 3-valued extension of RM- see [AB] or [Du]) is the strongest in the family of logics created by the Relevantists school. ***Weakening fails for this C.R. on both sides*** (this is our first example of this sort!).

5. $A_1 \ldots, A_n \vdash_{Luk} B$ iff $A_1 \to (A_2 \to \ldots \to (A_n \to B) \ldots)$ is valid in Lukasiewicz three-valued logic. (using negation, it is easy to give a corresponding interpretation for *every* sequent). Its main property is that *contraction fails* (on both sides). It is therefore completely necessary to work with *multisets* within this C.R. .

6. The classical C.R. (our first example) can also be characterized in the present framework by : $\Gamma \vdash \Delta$ iff for every v either $v(A_i) = -1$ or $v(B_j) = 1$ or at least one formula in $\Gamma \cup \Delta$ gets 0 (the proof of this claim uses known proof-theoretical reductions). Note that this C.R. is *not* the union of $\vdash_{Kl}$ and $\vdash_{Pac}$. For example, if we take $\Gamma = \{C \vee \neg P, P \vee R\}, \Delta = \{C, S, R \wedge \neg S\}$ then classically $\Gamma \vdash \Delta$, but this is not the case if we interpret $\vdash$ as either $\vdash_{Pac}$ or $\vdash_{Kl}$!

All the consequence relations described above were defined in abstract terms, using semantics. The rest of this paper is devoted to various methods for *syntactically* characterizing consequence relations. It is possible, of course, for a C.R. to be defined directly in syntactical terms, and often this is (or was) the case. Thus, the two kinds of modal C.R.'s given above were in use (at least for important special cases) much before the above semantic description was known!

[5] $A + B \overset{\text{def}}{=} \neg A \to B$. $B_1 + \cdots + B_n \overset{\text{def}}{=} \bot$ in case n=0.

4 Classification of C.R. according to their Basic Connectives

Our main object in this section is to provide a syntactical characterization of some propositional C.R.'s in terms of the connectives which are definable in them. We examine for this purpose two classes of connectives which are important in the present framework and see what rules they should obey. All these connectives and rules will be found to be quite familiar. (In fact, the set of primitive connectives of practically any seriously investigated logic forms (more or less) a subset of the set of the general connectives introduced below). We then use these connectives for characterizing several well-known logics.

4.1 Intensional connectives

The main group which we discuss is that of intensional connectives relative to a given C.R.. They can be characterized as connectives that enable one to transform a given sequent to an equivalent one which has a special required form. By "equivalent" here we mean just that one sequent obtains iff the other does (but in most important cases it can be interpreted in a much stronger sense).

In what follows assume $\vdash$ to be a fix C.R., all notions defined are taken to be relative to $\vdash$.

Intensional Disjunction: We call a binary connective $+$ an intensional disjunction if for all Γ, Δ, A, B:

$$\Gamma \vdash \Delta, A, B \quad \text{iff} \quad \Gamma \vdash \Delta, A + B \ .$$

Intensional Conjunction: We call a binary connective $\circ$ an intensional conjunction if for all Γ, Δ, A, B:

$$\Gamma, A, B \vdash \Delta \quad \text{iff} \quad \Gamma, A \circ B \vdash \Delta \ .$$

Intensional Negation: We call a unary connective $\neg$ a right intensional negation if for all Γ, Δ, A:

$$\Gamma, A \vdash \Delta \quad \text{iff} \quad \Gamma \vdash \Delta, \neg A \ .$$

We call a unary connective $\neg$ a left intensional negation if for all Γ, Δ, A:

$$\Gamma \vdash \Delta, A \quad \text{iff} \quad \Gamma, \neg A \vdash \Delta \ .$$

Since $\neg$ is a right intensional negation iff it is a left one, we therefore use the term *intensional negation* to mean either.

Intensional Implications: We call a binary connective $\rightarrow$ a weak intensional implication if for all A, B:

$$A \vdash B \quad \text{iff} \quad \vdash A \rightarrow B\ .$$

We call a binary connective $\rightarrow$ an intensional implication if for all Γ, A, B:

$$\Gamma, A \vdash B \quad \text{iff} \quad \Gamma \vdash A \rightarrow B\ .$$

We call a binary connective $\rightarrow$ a *strong* intensional implication if for all Γ, Δ, A, B:

$$\Gamma, A \vdash \Delta, B \quad \text{iff} \quad \Gamma \vdash \Delta, A \rightarrow B\ .$$

Intensional Truth: We call a 0-ary connective $\top$ an intensional truth if for all Γ, Δ:

$$\Gamma \vdash \Delta \quad \text{iff} \quad \top, \Gamma \vdash \Delta\ .$$

Intensional Falsehood: We call a 0-ary connective $\perp$ an intensional falsehood if for all Γ, Δ:

$$\Gamma \vdash \Delta \quad \text{iff} \quad \Gamma \vdash \Delta, \perp\ .$$

Proposition: The following are immediate consequences of the above definitions:

1. If $\vdash$ has a (primitive or definable) intensional disjunction then any sequent $\Gamma \vdash \Delta$ such that Δ is not empty is equivalent to a sequent of the form: $\Gamma \vdash A$. If $\vdash$ has also an intensional falsehood then this is true for *any* sequent.

2. If $\vdash$ has a (primitive or definable) intensional conjunction then any sequent $\Gamma \vdash \Delta$ such that Γ is not empty is equivalent to a sequent of the form: $A \vdash \Delta$. If $\vdash$ has also an intensional truth then this is true for *any* sequent.

3. If $\vdash$ has intensional disjunction, conjunction, falsehood and weak implication or intensional disjunction, implication and falsehood then for all Γ, Δ there is A such that $\Gamma \vdash \Delta$ is equivalent to $\vdash A$. Problems concerning such consequence relations can therefore be reduced to problems about *theoremhood* of formulas.

4. If $\vdash$ has a right (left) intensional negation then any sequent is equivalent to one in which the left (right) hand side is empty. If in addition it has also an intensional disjunction (conjunction) then every non-empty sequent is equivalent to one of the form $\vdash A$ ($A \vdash$).

5. If $\vdash$ has a strong implication then every sequent in which the right-hand side is not empty is equivalent to one with the left-hand side empty.

4.2 Characterizing the Intensional Connectives by Gentzen-Type Rules

In the previous subsection we have introduced several intensional connectives. Their definition can be split into two rules that are the converse of each other. In each case one of the two rules does not have what Gentzen has called "the subformula property". In this subsection we describe a uniform method for deriving rules *with* the subformula property which characterize the same connectives. All the rules we shall find are quite standard in Gentzen-type systems. As an example, we treat the case of strong implication in detail. We then list the rules which are obtained for the other connectives by using the same method.

The definition above directly entails that → is a strong implication iff $\vdash$ is closed under the rules:

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \quad \frac{\Gamma \vdash A \to B, \Delta}{\Gamma, A \vdash B, \Delta} .$$

The first of these rules already has the subformula property. The second does not. In order to find an appropriate substitute for it we use the reflexivity and transitivity of $\vdash$. The reflexivity condition, applied to a formula with → as the principal connective yields:

$$A \to B \vdash A \to B .$$

Hence the second condition above implies that:

$$A, A \to B \vdash B .$$

Taking $A \to B$ to be the principal formula in the last sequent, we proceed next to eliminate the others using cuts. Suppose, accordingly, that $\Gamma_1 \vdash \Delta_1, A$ and $B, \Gamma_2 \vdash \Delta_2$. Then two cuts of these two sequents with the last sequent above result with $\Gamma_1, \Gamma_2, A \to B \vdash \Delta_1, \Delta_2$. We obtain, therefore, that in order for → to be a strong implication relative to $\vdash$ this relation should be closed under the rule:

$$\frac{\Gamma_1 \vdash \Delta_1, A \quad B, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \to B \vdash \Delta_1, \Delta_2} .$$

Conversely, the closure of $\vdash$ under this rule implies the provability of $A, A \to B \vdash B$, and so if $\Gamma \vdash A \to B, \Delta$ then also $\Gamma, A \vdash B, \Delta$ (using a cut).

By using the same analysis we get the following

Proposition:

1. → is a strong intensional implication iff $\vdash$ is closed under the rules:

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \quad \frac{\Gamma_1 \vdash \Delta_1, A \quad B, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \to B \vdash \Delta_1, \Delta_2} .$$

Similarly we can show:

2. $+$ is an intensional disjunction iff $\vdash$ is closed under the rules:

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta} \quad \frac{A, \Gamma_1 \vdash \Delta_1 \quad B, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A + B \vdash \Delta_1, \Delta_2}.$$

3. $\circ$ is an intensional conjunction iff $\vdash$ is closed under the rules:

$$\frac{\Gamma_1 \vdash \Delta_1, A \quad \Gamma_2 \vdash \Delta_2, B}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A \circ B} \quad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \circ B \vdash \Delta}.$$

4. $\rightarrow$ is an intensional implication iff $\vdash$ is closed under the rules:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad \frac{\Gamma_1 \vdash \Delta_1, A \quad B, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \rightarrow B \vdash \Delta_1, \Delta_2}.$$

5. $\bot$ is an intensional falsehood iff $\vdash$ is closed under the rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot} \quad \bot \vdash .$$

6. $\top$ is an intensional truth iff $\vdash$ is closed under the rules:

$$\vdash \top \quad \frac{\Gamma \vdash \Delta}{\top, \Gamma \vdash \Delta}.$$

7. The following conditions are equivalent:

   (a) $\neg$ is an intensional negation.

   (b) $\neg$ is a right intensional negation.

   (c) $\neg$ is a left intensional negation.

   (d) $\vdash$ is closed under the rules:

$$\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A} \quad \frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}.$$

The following observations are immediate from these characterizations. They are familiar facts about the connectives of classical logic. The foregoing discussion and the examples at the end of this section show, however, that they are not peculiar to classical logic or to truth-functional conntectives.

- If $\vdash$ has an intensional negation $\neg$ and an intensional disjunction $+$ then (as in classical logic) it also has intensional conjunction and strong implication, defined by $\neg(\neg A + \neg B)$ and $\neg A + B$ respectively. Similar results obtain if $\vdash$ has an intensional negation and either an intensional conjunction or a strong intensional implication.

- If $\vdash$ has a *strong* intensional implication $\rightarrow$ and an intensional falsehood $\bot$ then it also has an intensional negation, defined by $A \rightarrow \bot$.

- If $\vdash$ is closed under weakenings and has *theorems* (i.e. formulas A such that $\vdash A$), then each of these theorems is an intensional truth. (For having theorems it suffices, e.g. that $\vdash$ has a weak intensional implication, since then $A \rightarrow A$ is a theorem for every A.) If such $\vdash$ has also an intensional negation then the negation of each theorem is an intensional falsehood.

4.3 The combining connectives

The function of the intensional connectives is to enable certain operations on a single sequent in order to transform it to some desirable form. We now study connectives that have a different function: To enable the combination of two sequents into a single one. one which contains exactly the same information as the original two. [6] In the list above of the rules for the intensional connectives there are, of course, rules that take two sequents and return a single one. The resulting combination is not reversible, though: the premises cannot always be recovered from the conclusion. Indeed, one cannot expect the *exact* combination of any two sequents always to be possible. It *is* possible, though, in one important case: When the two sequents to be combined are identical in all formulas except perhaps the two that are actually connected by the combining connective. Now there are three possible positions that these exceptional formulas might occupy and according to them we get the following three combining connectives:

Combining Conjunction: We call a connective $\wedge$ a combining conjunction iff for all Γ, Δ, A, B:

$$\Gamma \vdash \Delta, A \wedge B \quad \text{iff} \quad \Gamma \vdash \Delta, A \ \text{ and } \ \Gamma \vdash \Delta, B\,.$$

(In this case both exceptional formulas are succedents).

Combining Disjunction: We call a connective $\vee$ a combining disjunction iff for all Γ, Δ, A, B:

$$A \vee B, \Gamma \vdash \Delta \quad \text{iff} \quad A, \Gamma \vdash \Delta \ \text{ and } \ B, \Gamma \vdash \Delta\,.$$

(In this case both exceptional formulas are antecedents).

[6] The connectives we discuss are usually called "extensional" by the relevantists, while Girard calls them "additive" (see [Gi]). We find the term "combining" more suggestive.

Combining Implication: We call a connective $\supset$ a combining implication iff for all Γ, Δ, A, B:

$$A \supset B, \Gamma \vdash \Delta \quad \text{iff} \quad \Gamma \vdash \Delta, A \text{ and } B, \Gamma \vdash \Delta\ .$$

(In this case the exceptional formulas are on different sides of $\vdash$).

The choice of these three connectives was guided by tradition. Obviously, there are three other possibilities. Thus we could characterize the "Sheffer stroke" by:

$$\Gamma, A|B \vdash \Delta \quad \text{iff} \quad \Gamma \vdash \Delta, A \text{ and } \Gamma \vdash \Delta, B\ .$$

It is not difficult, indeed, to carry the analysis below to either this connective or to the others. We shall be satisfied, though, with the above three. (Note, by the way, that if an intensional negation is available then the existence of one combining connective entails the existence of the rest).

By using exactly the same method we have above applied for investigating the intensional connectives we can now show:

1. $\vee$ is a combining disjunction iff $\vdash$ is closed under the rules:

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \quad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B} \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ .$$

2. $\wedge$ is a combining conjunction iff $\vdash$ is closed under the rules:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \quad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\ .$$

3. $\supset$ is a combining implication iff $\vdash$ is closed under the rules:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, A \supset B} \quad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B} \qquad \frac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A \supset B}\ .$$

It is easy to see that relative to an *ordinary* C.R. (i.e. regular and closed under weakenings) a connective is a combining disjunction (conjunction, implication) iff it is an intensional disjunction (conjunction, strong implication). This is not the case in general, though, as the examples of Linear Logic and Relevance Logic below show.

4.4 Characterization of some Known Logics

As the title suggests, we shall try to use the various notions introduced so far for characterizing several well-known logics. We shall examine these logics according to three criteria:

- Regular or not.
- Closure under weakening.
- The intensional and combining connectives which are available in them.

"Multiplicative" Linear Logic: [7] This is the logic which corresponds to the *minimal* C.R. which includes all the intensional connectives (half of them will do, of course). It is possible to delete the intensional truth and falsehood to get the minimal C.R. which includes the others. (The system is conservative with respect to this fragment). In both versions the system is neither regular nor closed under weakenings.

Propositional Linear Logic (without the "exponentials" and the propositional constants T and F): This is the minimal consequence relation which contains all the connectives introduced above. Again— it is neither regular nor closed under weakening. It is important to note that its intensional connectives behave quite differently from its combining ones!

$R_{\overset{\sim}{\rightarrow}}$ **-the Intensional Fragment of the Relevant Logic** R**:** [8] This corresponds to the minimal C.R. which contains all the intensional connectives (the intensional truth and falsehood are again optional) and is *closed under contraction.* It is still not regular since the converse of contraction (and so also weakening) fails for it. (A word is in order here of what C.R. associated with $R_{\overset{\sim}{\rightarrow}}$ we have in mind, since there is more than one candidate. The answer is: that which the standard Gentzen type formulation of this system defines. An equivalent definition is: $A_1, \ldots, A_n \vdash_{R_{\overset{\sim}{\rightarrow}}} B_1, \ldots, B_m$ iff $A_1 \rightarrow (A_2 \rightarrow \cdots (A_n \rightarrow B_1 + \cdots + B_m) \ldots)$ is a theorem of this system, where $A + B$ is $\sim A \rightarrow B$ and $(A_n \rightarrow B_1 + \cdots + B_m)$ is $\sim A_n$ in case $m = 0$. The last interpretation is the one we have in mind also with respect to the other systems in the Relevance/Linear family).

R^t **without Distribution:** This corresponds to the minimal C.R. which contains all the connectives which were described above and is closed under contraction. $\bot$ and $\top$ are again optional.

[7] see [Gi].

[8] See [AB] or [Du].

$RMI_{\underset{\rightarrow}{\sim}}$: [9] this corresponds to the minimal *regular* C.R. which contains intensional negation, disjunction, conjunction and strong implication.

$RM^t_{\underset{\rightarrow}{\sim}}$: [10] This corresponds to the minimal regular C.R. which includes *all* intensional connectives described above.

In contrast to Linear Logic and $R_{\underset{\rightarrow}{\sim}}$, $RM_{\underset{\rightarrow}{\sim}}$ is *not* a conservative extension of $RMI_{\underset{\rightarrow}{\sim}}$. This means that the addition of the intensional truth (or falsehood) to $RMI_{\underset{\rightarrow}{\sim}}$ forces new sequents in the language of this system to obtain. The reason is that the intensional constants bring with them part of the power of weakening. This happens to be harmless in the context of Linear logic and R, but together with the converse of contraction, (which is available, of course in every regular C.R. but can also be taken as a *very* special case of weakening) it causes sequents like $A, B \Rightarrow A, B$ to be provable. This is true in fact for every regular C.R. which has an intensional implication: Starting with $\Rightarrow \top, \top$, two applications of $(\rightarrow\Rightarrow)$ give $\top \rightarrow A, \top \rightarrow B \Rightarrow A, B$. But since $\top, A \Rightarrow A$ is provable, so are $A \Rightarrow \top \rightarrow A$ and $B \Rightarrow \top \rightarrow B$. Hence two cuts give $A, B \Rightarrow A, B$ (In order to get a cut-free representation of this system it is necessary to add the following "mingle" rule: from $\Gamma_1 \Rightarrow \Delta_1$ and $\Gamma_2 \Rightarrow \Delta_2$ infer $\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2$)[11].

RM^t without Distribution: [12] This corresponds to the minimal regular C.R. which has all the connectives described above. It is not closed under weakening. Again, the intensional constants here are optional.

Classical Propositional Logic: This of course corresponds to the minimal ordinary C.R. which has all the above connectives. Needless to say, there is no difference in it between the combining connectives and the corresponding intensional ones.

Intuitionistic Logic: It is a common belief that the main difference between classical and intuitionistic logic is that the later is essentially single-conclusioned while the former is essentially multiple-conclusioned. (The origin of this belief is the way in which Gentzen has formulated his sequential version of intuitionistic logic. That version differs from his formalism for classical logic only by limiting sequents to have at most one formula in the succedent.) Assuming for a moment this belief is true, it is straightforward to define

[9] see [AB] pp. 148-9 and [Av1].

[10] This is the intensional fragment of the system RM^t (see [AB] or [Du]). Without the propositional intensional constants this is Sobocinski 3-valued logic (see **2.4** above), also called $RMI^1_{\underset{\rightarrow}{\sim}}$ in [Av1].

[11] For a proof and for more information about this system see [Av2].

[12] see [AB] or [Du].

for the single-conclusioned case the notions of a regular and an ordinary C.R., the combining connectives and the intensional conjunction, implication, truth and falsehood (but not strong implication!). It is easy then to see that:

> The usual single-conclusioned C.R. associated with intuitionistic logic is the minimal ordinary single-conclusioned C.R. which has intensional implication and falsehood and combining disjunction and conjunction.

We proceed now to offer what we believe to be a deeper analysis, in which we need not treat intuitionistic logic differently from the other logics we have considered so far. What we seek therefore is a multiple-conclusioned conservative extension of the above single-conclusioned C.R., which has the same types of basic connectives. The unique solution to this problem is easy to find once we recall that for *ordinary* C.R. a combining disjunction is also an intensional one. Accordingly, we define:

> $A_1, \ldots, A_n \vdash_{Int} B_1, \ldots, B_m$ iff there is a proof of $B_1 \vee \cdots \vee B_m$ from $A_1, \ldots, A_n$ in one of the usual formalisms for intuitionistic logic.

It is easy now to see that:

> $\vdash_{Int}$ is the minimal ordinary C.R. which has intensional disjunction, conjunction, falsehood and implication. [13]

It is illuminating to compare this to the following possible characterization of classical logic:

> The standard C.R. associated with classical logic is the minimal ordinary C.R. which has intensional disjunction, conjunction, falsehood and *strong* implication.

According to these two characterizations classical and intuitionistic logic differ mainly with respect to their *implication* connective, while their disjunctions are the same! Indeed, it is easy to show that exactly the same sequents which involve only $\vee, \wedge$ and $\perp$ are valid in both (note that there are no theorems, i.e. sequents of the form $\vdash A$, among these sequents!). There is however another crucial difference between the two logics that is somewhat hidden in these characterizations:

[13] This characterization corresponds to Maehera's multiple-conclusioned, cut-free Gentzen-type formulation of Int. logic which appears in ch. 1 of [Tak].

Intuitionistic logic does not contain any intensional negation: There is no sentence $N(p)$ in the $\{\vee, \wedge, \rightarrow, \perp\}$-language (in which only the atomic formula p occurs) such that $p, N(p) \vdash$ and $\vdash p, N(p)$ are both valid. (This is an immediate consequence of the disjunction property of $\vdash_{Int}$). The usual definition of $\neg A$ as $A \rightarrow \perp$ works well when $\rightarrow$ is a strong implication (as is in the classical case) but not otherwise. This fact might explain why the rules for negation look so nasty compared to the other rules in the context of intuitionistic natural deduction, and why authors like Prawitz and Schroeder-Heister prefer to take $\perp$ rather than negation as a primitive connective of intuitionistic logic. [14]

4.5 On the Meanings of the Propositional Connectives

There is a long tradition, originated already in Gentzen ([Gen]), about the introduction and elimination rules of Natural Deduction as providing the meanings of the propositional connectives. The famous [Pri] has forced the followers of this tradition to be more careful about this issue, and so today the emphasis is usually put on the *introduction* rules as those which define the meaning of a connective. Concerning the elimination rules the general principle is taken to be that one should not be able to get more out of a formula than the introduction rules can put into it. This principle was used, e.g., by Schroeder-Heister in [SH] for developing an explicit method for *deriving* the (unique) elimination rule for a connective from the corresponding set of introduction rules. [15]

This all is very nice. Unfortunately, it does not seem to work beyond the realm of intuitionistic logic. A particularly important connective that seems to escape this type of characterization is the negation. One illuminating fact about it in this respect is that intuitionists usually raise the above principle to attack the excluded middle, why relevantists (like Dunn in [Du] p. 152) use it for justifying *their* rejection of the disjunctive syllogism. Neither the intuitionists nor the relevantists reject both laws, though.

The intuitionists usually try to avoid the problem of characterizing negation by introducing instead as primitive a (quite artificial) constant for intensional falsehood, and then defining (even more artificially) negation as $A \rightarrow \perp$[16]. As we show above, this might work well in the context of *classical* logic, but in no way defines an *appropriate* negation in the intuitionistic case (nor is there any other way of defining a decent negation in the context of intuitionistic logic).

[14] See [Pra2] and [SH].

[15] For more explanations about this tradition—see [Sud2] and the extensive literature cited there.

[16] I should admit that I do not believe that in an ordinary discourse, when someone denies something, he means that what he denies implies "der false"...

The present paper suggests another method of taking rules as defining the meaning of connectives. It had the following two main properties:

- The meaning of a connective is always something which is *relative* to some C.R..

- What defines a connective is not a set of "introduction rules" but a single rule which is *reversible.*

The reversible rule which defines a connective might introduce it in either of the succedent or the antecedent of a sequent. In the first case it usually corresponds to an "introduction" rule of N.D., in the second—to an elimination rule. There is no priority therefore, at least in this context, to introduction rules over elimination rules. Indeed, the combining disjunction, for example, is characterized by what is usually taken as the elimination rule for disjunction. (We are prepared to argue, in fact, that actual uses of disjunctions are usually made when one wants to infer something from them without being in a position to assert either of the disjuncts!)

5 Uniform Representations of C.R's

The notion of a C.R., as defined in the first section and exemplified in the second is an *abstract* one. We have seen above several ways, semantical as well as syntactical, of defining or characterizing C.R.'s. However, in order to use a certain abstract C.R. in practice one needs a *concrete* way of *representing* it. This is usually done by using a formal system [17]. There are two basic demands that such representations of a C.R. $\vdash$ should meet. These are:

Faithfulness: If the representation can be used to show that $\Gamma \vdash \Delta$ then this is actually the case.

Effectiveness: If someone uses the representation to show that $\Gamma \vdash \Delta$ then it can mechanically be checked that he really does so.

There is also a third property that we would like an adequate representation of $\vdash$ to have, but is not always achievable:

Completeness: Whenever $\Gamma \vdash \Delta$ the representation can be used to show this.

Note: If we accept Church's thesis, then the effectivity demand means that the set of of sequents that can be shown to hold by a formal representation is an r.e.

[17] In fact, Hodges ([Ho],p.26) defines a formal system (or a "formal proof calculus") to be "a device for proving sequents in a language L."

set. Completeness can in principle be achieved, therefore, only if the represented C.R. is r.e.. Otherwise we can only expect a *partial* representation.

As was emphasized several times above, we understand a C.R. $\vdash$ to be an abstract relation, and so whenever we write a sequent $\Gamma \vdash \Delta$ this is a meta-claim about $\vdash$. Now it is important to realize that while dealing with a C.R. $\vdash$ the premises of a sequent are no less important than the conclusions. A full representation of $\vdash$ should reflect this. Hence it should include in its language a formal counterpart "$\Gamma \vdash \Delta$" for each $\Gamma \vdash \Delta$, so that officially the system can show that $\Gamma \vdash \Delta$ iff "$\Gamma \vdash \Delta$" is derivable in it. Obviously, the most direct way of achieving such a formal counterpart for each abstract sequent is to have in the formal language a formal symbol which directly corresponds to $\vdash$. We shall use henceforth "$\Rightarrow$" to denote such a symbol, reserving "$\vdash$" to denote (in the metalanguage of our discussion) the represented C.R. [18]. Accordingly we shall call creatures of the form $\Gamma \Rightarrow \Delta$ "formal sequents". In order to unify our treatment of the usual various formal systems, we shall assume below that such a formal symbol alway exists. If officially it does not, then this assumption means at worst that we are considering an *extended* language in which it does, and an *extended* formal system in which the connections between the old one and $\vdash$ are made a part of the formal machinery (while in the original system they should be explained in the meta-language). For example, an explanation in the metalanguage of the form: "$A_1, \ldots, A_n \vdash B$ iff there is a proof of B from $A_1, \ldots, A_n$" will be translated (in the extended system) to: "From a formal proof of B from $A_1, \ldots, A_n$ infer $A_1, \ldots, A_n \Rightarrow B$". (formulas, *formal* sequents and proofs (in the original system) of formulas from other formulas will all be formal objects of such an extended formal system, possibly with different types!) This might look a little bit complex, but it is absolutely necessary (in some form or another) for a real full representation (that can, e.g. be computerized). Anyway, in the usual cases a much simpler translation will be provided below. It will allow us to regard Hilbert-type systems for provability, natural deduction systems and Gentzen type systems all as *axiomatic systems* (see 1.1) in which the "theorems" are formal sequents.

5.1 Using Axiomatic Systems for Representations

The oldest way of representing a C.R. $\vdash$ is by using an axiomatic system (see **1.1**) which have the same language L (i.e. the same well- formed formulas) as $\vdash$. Now axiomatic systems are designed to prove *theorems*, and so they can be used only indirectly for representing C.R.'s . There are two main methods for doing this:

[18] It is also customary to use $\models$ for the C.R. and $\vdash$ for its formal counterpart.

The Interpretation Method: One defines a correspondence between sequents of L and sets of wffs of L so that $\Gamma \vdash \Delta$ iff at least one of the sentences of the corresponding set (or perhaps all of them) is a theorem of the corresponding axiomatic system. Usually, the corresponding set is just a singleton, and so every sequent is translated into some formula of the language. Such a translation is straightforward if $\vdash$ has the needed intensional connectives. This was really the case for all the logics that were investigated in previous stages of the development of mathematical logic (like classical and intuitionistic logics) and so it suffices at these stages to concentrate on the notion of theoremhood while doing logic. An interpretation using intensional connectives has the property that the interpretation of the formal sequent $\Rightarrow A$ is just A. This should not always be the case, though. A famous example for this is Gödel interpretation of classical logic within intuitionistic logic. An example in which the interpretation does not use just singletons may be provided by the intuitionistic pure implicational C.R.. Here $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ iff *for some i*, $A_1 \rightarrow (A_2 \rightarrow \ldots \rightarrow (A_n \rightarrow B_i) \ldots)$ is a theorem (of intuitionistic logic).

The Extension Method: Here we say that $A_1, \ldots, A_n \vdash B$ iff B is a theorem of the axiomatic system which is obtained by adding $A_1, \ldots, A_n$ to the axioms of the given one. This second method is the origin and the prototype of what is known as Hilbert-type systems. In the next subsection we shall treat this type of systems in detail. For the time being let us just mention two properties of every C.R. $\vdash$ that can be represented using this method: First, such a C.R. is single-valued. Second, it is what we call above ordinary, i.e: closed under weakenings (on the left) and relates sets (rather than multisets) of formulas.

5.2 Hilbert Type Representations

If one uses an axiomatic system AS together with the extension method in order to show that $A_1, \ldots A_n \vdash B$ than one should provide a proof of B in the new axiomatic system $AS + \{A_1, \ldots A_n\}$. Of course, we do not seriously mean that we are using an infinite number of new axiomatic systems. We just use one which each time is (temporarily) augmented by a finite number of axioms in order to derive some sequent. If we want both to make this explicit and to unify our handling of this "infinite number" of axiomatic systems, then we should work directly with formal sequents. What we get is the following *axiomatic system for sequents:*

Axioms:

1. $A \Rightarrow A$ for every A.
2. $\Rightarrow A$ wherever A is an axiom of AS.

Rules:

1.
$$\frac{\Gamma \Rightarrow A}{\Delta, \Gamma \Rightarrow A} \quad \text{(weakening)}$$

2.
$$\frac{\Gamma_1 \Rightarrow A_1 \ldots, \Gamma_n \Rightarrow A_n}{\Gamma_1, \ldots \Gamma_n \Rightarrow B}$$

   where
$$\frac{A_1, \ldots, A_n}{B}$$
   is (an instance of) a rule of AS.

The main fact to note concerning this sequential system is that all the "activity" is made on the right-hand side of the ⇒. Besides the structural rule of weakening and the axioms which reflect the reflexivity condition— the set of wffs on the left-hand side of a conclusion of a rule is always the union of the left-hand sides of the premises. This is, in fact, the main thing that is common to all "Hilbert-type formalisms" that can be found in the literature. Accordingly we can define this kind of formalism quite generally as follows:

Definition: *A Hilbert-type system for consequence* in the language L is an axiomatic system the "formulas" of which are formal sequents of L and:

1. The axioms includes $A \Rightarrow A$ for all A. All the other axioms are of the form $\Rightarrow A$.
2. With the possible exception of cut and weakening, the *set* of formulas which appear on the left-hand side of a conclusion of a rule is the union of the *sets* of formulas which appear on the left-hand side of the premises. (We shall call this property the *left-hand side property*).

If we take axioms as rules with 0 premises then Hilbert representations can be characterized as those systems which ***have besides the basic reflexivity and transitivity rules only structural rules and/or rules with the left-hand side property.***

It is important to realize in what ways does this definition of a Hilbert-type representation really generalizes from the above Hilbert-type system which was derived by the extension method. There are, in fact three new issues involved:

- The above system was ordinary. In the general case we deal, however, with multisets. We have not demand, though, that the multiset of formulas on the left-hand side of a conclusion should be the ***multiset union*** of the l.h.s. of the premises. We require these multisets only to be identical as ***sets.*** This allows for rules like

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

  (which characterizes the combining conjunction and are important in the context of, e.g., linear and relevance logics).

- The system above has the property that whenever its rules allow (for some $\Delta, \Gamma_1, \ldots, \Gamma_n$) the inference of $\Delta \Rightarrow B$ from $\Gamma_1 \Rightarrow A_1, \ldots \Gamma_n \Rightarrow A_n$ then for ***all*** $\Gamma'_1 \ldots, \Gamma'_n$ we may infer $\Gamma'_1, \ldots, \Gamma'_n \Rightarrow B$ from $\Gamma'_1 \Rightarrow A_1, \ldots, \Gamma'_n \Rightarrow A_n$. We shall call Hilbert type systems with this property ***pure*** . In such systems we can just take all these inferences to be instances of the schema:

$$\frac{A_1 \ldots A_n}{B}.$$

  Obviously pure systems are the most frequent type of systems that one finds in practice. As we shall see, this is true also for natural deduction and Gentzen- type systems, so we delay the discussion of this crucial property until we have it defined also for the other types of systems. For the moment let us give only one quite famous example of an impure system. Take some normal modal logic with the associated ***truth*** C.R. (see 2.3). A standard Hilbert-type representation will have some axioms and the following two general rules of inference:

$$\frac{\Gamma_1 \Rightarrow A \qquad \Gamma_2 \Rightarrow A \rightarrow B}{\Gamma_1, \Gamma_2 \Rightarrow B}$$

$$\frac{\Rightarrow A}{\Rightarrow \Box A}.$$

  The first one (M.P) is pure [19], the other is not. Hence this Hilbert-type system is impure. (The material-implication → of $S4$, say, is an intensional

[19] It should be clear from the definition above what the definition of a pure rule is, and so we omit it here and elsewhere.

strong implication relative to *this* C.R. and so it is quite common to represent this C.R. by using the interpretation method, applied to the standard axiomatic system for S4-theorems).

- In the general definition of a Hilbert-type system we allow cut to be one of the primitive rules. This is unnecessary for pure systems, since they are always closed under this rule (this can easily be proved by induction on the length of the proof of that premise of the cut in which the cut formula occurrs on the left). It is true also for many impure systems, like that for modal logics described above. This might explain why the cut is never discussed in the context of Hilbert-type systems, and I believe that this state of affairs should be taken as normative: A reasonable Hilbert-type formalism should always allow cut-elimination.

Hilbert-type systems, as defined above, always represent a single conclusioned C.R.. There is nothing in the discussion above that depends on this feature, though. A natural generalization will be, therefore, to remove this limitation and to allow multiple-conclusioned Hilbert-type systems. (In the last section we shall present a practical use of such a system.)

Remark: It is a common belief that Hilbert-type systems have only to do with *provability*, not with deducibility. [20] Sometimes this belief is expressed just before the author proceeds to prove a deduction theorem about deducibility in some Hilbert-type system...The reason for this belief are partially historical: In the past logicians happened to be interested mainly in proving logical theorems and used for this Hilbert-systems for theoremhood, i.e. axiomatic systems. Thus most textbooks about modal logics present them as axiomatic systems and are quite confused about what amount to a deduction from assumptions in them. This confusion is due to the fact that there might exist *more than one reasonable way* to define a C.R. which is compatible with a given axiomatic system (i.e.: has the same set of *theorems*). The pure one, obtained by the extension method, is always a candidate, but they might be more. We have seen already the example of the impure Hilbert-type representations of the truth C.R. in normal modal logics (the pure extension, by the way, corresponds to the *validity* C.R.), and we shall see more below. What makes this phenomenon possible is that in Hilbert-type system a proof of a sequent $\Rightarrow \Delta$ always consists of sequents with the same form. This allows for a certain degree of freedom while deciding what *other* sequents should be taken as valid!

[20] see, e.g., [sud1] pp. 134-5.

5.3 Natural-Deduction Representations

The next type of formal systems that we are about to examine is natural- deduction . We start with a very general definition of this type of systems which closely resembles that given above for Hilbert-type formalisms:

Definition: *A* Natural-deduction system in the language L is an axiomatic system such that:

1. The formulas are formal sequents of L.
2. $A \Rightarrow A$ is an axiom for each formula A.
3. All the other rules (including 0-premises rules!) has the following property: The set of formulas that appear on the left hand side of their conclusion is a *subset* of the union of the left-hand side of the premises.

If we compare this definition to that of Hilbert-type systems we see that the only difference is that in N.D. (Natural-Deduction) systems we allow certain formulas of the left-hand side of a premise of a rule to disappear (or to "be *discharged*") from the left-hand side of the conclusion.

The notion of "pure" system can now be generalized to N.D. systems as follows.

Definition: We call a N.D. system "pure" if whenever $\Gamma \Rightarrow \Delta$ can be derivable from $\Gamma_1 \Rightarrow \Delta_1, \ldots, \Gamma_n \Rightarrow \Delta_n$ then there are sub-multisets $\Gamma'_1, \ldots, \Gamma'_n, \Delta'_1, \ldots, \Delta'_n$ of $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (respectively) and a submultiset Δ' of Δ such that: for every $\Gamma''_1, \ldots, \Gamma''_n, \Delta''_1, \ldots, \Delta''_n$ we can infer

$$\Gamma''_1, \ldots, \Gamma''_n \Rightarrow \Delta', \Delta''_1, \ldots, \Delta''_n$$

from

$$(\Gamma''_1, \Gamma'_1 \Rightarrow \Delta'_1, \Delta''_1), \ldots, (\Gamma''_n, \Gamma'_n \Rightarrow \Delta'_n, \Delta''_n) \ .$$

Pure N.D. systems are again the most usual kind of N.D. systems. In such systems we can take all the inferences to be applications of rules of the form:

$$\frac{\begin{matrix}[\Gamma'_1] \\ \Delta'_1\end{matrix} \quad \cdots \quad \begin{matrix}[\Gamma'_n] \\ \Delta'_n\end{matrix}}{\Delta'}$$

(where the Γ'_1 etc. are like in the above definition). A well known example of a N.D. which is not pure is Prawitz N.D. system for S4 (see [Pra1]). In this system one can infer $\Gamma \Rightarrow \Box A$ from $\Gamma \Rightarrow A$ only iff *all* the formulas in Γ begin with $\Box$, and so this rule is impure.

There are important differences that should be noted between pure Hilbert systems and pure N.D. systems. While using pure Hilbert systems one can (and does) write down in a proof-tree of a sequent only the r.h.s of the sequents which participate in that proof. He can then quite easily find out at the end which *sequent* was actually proved, by just looking at the root and the leaves of the tree. One need not know for this what intermediate sequents were proved before the final one was derived. Moreover: One can check each part of the proof separately without needing to examine what happens before that part. All these are not true for N.D. proofs—not even pure ones. Here one is *forced to keep track at each stage of a proof of what sequent was actually derived in it.* This is the case whether actually formal sequents are employed while implementing the system or other devices are used for this goal. Whatever the method is, the fact remains that N.D. systems are *essentially sequents calculi.*

Since natural-deduction systems enable us to manipulate sequents and not only formulas, they provide us within the formal machinery an access to methods of proofs that necessarily belong to the *meta*-theory in the case of Hilbert-type systems. An obvious example of this ability is the deduction theorem. For many Hilbert-type systems this is an important meta-theorem which is extensively used for indirectly showing that something is provable, without actually proving it. In natural deduction systems this method of proof is usually incorporated into the system as one of its rules. The ability to do such things is the main source of power for N.D. systems and their crucial advantage over Hilbert-type systems.

At this point a natural objection may be raised against our definition of natural deduction systems: According to our presentation, every Hilbert-type system is a N.D. system, but not vice versa. This seems to render as pointless the frequent problem of finding a natural-deduction presentation for a C.R. to which a Hilbert-type representation is known. Something essential seems to be missing: the notion of introduction and elimination rules which [Sud1], for example, takes to be the second major component (besides the possibility of discharging assumptions) of natural deduction systems. I was unable, though, to find a sufficiently general definition of this notion which will not rule out from the class of N.D. formalisms systems that are taken to be such in the literature. In fact the very first N.D. representation of classical logic in [Gen] included as axiom the excluded middle, which cannot be characterized in terms of introduction and elimination rules. It really seems that almost only intuitionistic logic, together with a careful choice of the basic connectives, admits single-conclusioned N.D. representation consisting only of matching introduction and elimination rules. (The truth is, however, that some fragments of Linear and relevance logics admit such as well, but the corresponding C.R. is not ordinary). As for classical logic, the only way to do so is by using *multiple-conclusioned* N.D. systems, but this certainly is not the *standard* procedure!

In my opinion, therefore, the demand for using matching introduction and elimination rules belongs to the methodology of constructing *good* N.D. representations, not to their definition. This raise the question what makes one formal representation of a C.R. better then another. These can be judged according to the following two criterions:

- ease of finding (and also checking) proofs of sequents.
- usefulness for (constructively) showing significant properties of the represented C.R. (An example of such a property in many logics is Craig's interpolation theorem).

Past experience indicates that in the context of N.D. systems the above two goals are best achieved by first formulating such a system using introduction and (matching) elimination rules. Then using these rules for defining a notion of a "normal proof" with nice properties, and finally proving a "normalization theorem" to the effect that every proof can be converted into a normal proof of the same end sequent. This method seems to be successful only when the connectives involved are intensional [21] and when the represented C.R. can be characterized in terms of them— as in the examples of the last section. The method is even more limited if we confine ourselves to single-conclusioned C.R.'s. The *exact* characterization of the C.R.'s to which this method is applicable is an interesting topic which we are not going to pursue here. Let us just mention that some authors, especially those with intuitionistic tendencies, believe the method of introduction and elimination rules, as well as the concept of a normal proof, to be of a crucial philosophical importance. They believe, accordingly, that there are deeper reasons for the success of this method than the above description suggests. The interested reader is referred to the enormous literature on the subject like: [Pra1], [Pra2], [Sud2], [SH] and (of course!) the original paper of Gentzen ([Gen]). (We shall return to this issue in the next subsection as well.)

A final point which we like to discuss in this subsection is the status of the cut rule in the context of N.D. systems. It can easily be checked that the definition we gave above does not exclude cut from being one of the rules of a N.D. system. For pure N.D. system its eliminability is as easy to show as for pure Hilbert-type systems (with the same method of proof). For impure systems it might be less easy and should not be taken for granted. Let us give an example of such a system for which cut-elimination is true but not *completely* trivial: Consider the

[21] To a lesser degree—also to combining connectives. The resulting N.D. system is usually impure, though, when the combining connectives are not also intensional—as is the case in relevance and linear logics.

$\{\neg, \Box\}$ fragment of the N.D. system for S4. It is immediate that in this system we have that $\neg\neg\Box A \vdash \Box A$ and that $\Box A \vdash \Box\Box A$. It is not, at first glance, obvious that $\neg\neg\Box A \vdash \Box\Box A$, since because of the side conditions on the $(\Rightarrow \Box)$ rule, the proofs of these two sequents cannot directly be combined to produce a proof of the third. (This is strongly related to the fact that normalization fails for this version of the system- see [Pra1]). The system does admit cut-elimination, though, but this requires at least *some* efforts. It is not inconceivable that more complicated impure N.D. systems might offer more serious difficulties while proving cut- elimination, or even that it just may fail for them!

5.4 Gentzen-Type Representations

As we argue above, already pure N.D. systems essentially carry us from proofs of formulas to proofs of sequents. The next obvious step is therefore to take full advantage of the use of sequents by allowing the rules of a system to make significant changes also in the antecedent of a sequent:

Definition: A Gentzen-type representation of a given C.R. $\vdash$ in a language L is an axiomatic system such that:

1. The formulas of it are formal sequents of L.

2. A formal sequent $\Gamma \Rightarrow \Delta$ is a theorem iff $\Gamma \vdash \Delta$.

The concept of a *pure* representation can be naturally extended to Gentzen -type representations as follows:

Definition: We call a Gentzen-type system *pure* if whenever its rules allow the inference of $\Gamma_0 \Rightarrow \Delta_0$ from $\Gamma_1 \Rightarrow \Delta_1, \ldots, \Gamma_n \Rightarrow \Delta_n$ then there are subsets Γ'_i, Δ'_i of Γ_i, Δ_i (respectively) such that for every Γ''_i, Δ''_i we can infer $\Gamma''_1, \ldots, \Gamma''_n, \Gamma'_0 \Rightarrow \Delta'_0, \Delta''_1, \ldots, \Delta''_n$ from $\Gamma''_i, \Gamma'_i \Rightarrow \Delta'_i, \Delta''_i$ $(i = 1, \ldots, n)$.

The definition of pure Gentzen-type systems and of pure N.D. systems are very close, the only difference being with respect to the possible existence of Γ'_0 and the possibility (unless we deal with a single-conclusioned C.R.) for Δ'_0 to be empty. Accordingly we can, if we wish, use for pure rules almost the same notation that is frequently used for N.D. systems, but in which symmetry is restored between what is allowed to be a premise and what is allowed to be a conclusion. In fact pure rules in Gentzen systems may be written as follows:

$$\frac{\begin{matrix}[\Gamma'_1] \\ \Delta'_1\end{matrix} \;\dots\; \begin{matrix}[\Gamma'_n] \\ \Delta'_n\end{matrix}}{\begin{matrix}[\Gamma'_n] \\ \Delta'_n\end{matrix}}$$

(where the Γ'_i, Δ'_i $(0 \leq i \leq n)$ are like in the definition above).

Examples:

1. the standard $\vee-$ elimination rule of intuitionistic (single- conclusioned) N.D. system, usually written as

$$\frac{A \vee B \quad \begin{matrix}[A] \\ C\end{matrix} \quad \begin{matrix}[B] \\ C\end{matrix}}{C}$$

will have in Gentzen-type systems the form:

$$\frac{\begin{matrix}[A] \\ C\end{matrix} \quad \begin{matrix}[B] \\ C\end{matrix}}{\begin{matrix}[A \vee B] \\ C\end{matrix}}\ .$$

And in classical logic just the form: (when ϕ is the empty set):

$$\frac{\begin{matrix}[A] \\ \phi\end{matrix} \quad \begin{matrix}[B] \\ \phi\end{matrix}}{\begin{matrix}[A \vee B] \\ \phi\end{matrix}}\ .$$

The introduction rule in this context will be just:

$$\frac{A, B}{A \vee B}\ .$$

2. The usual antecedent rule for the conjunction is:

$$\frac{\begin{matrix}[A, B] \\ C\end{matrix}}{\begin{matrix}[A \wedge B] \\ C\end{matrix}}$$

in the intuitionistic case, [22] while in classical logic we can replace C above with the empty set. For both systems the succedent rule for conjunction is just:

$$\frac{A \quad B}{A \wedge B}\,.$$

In my opinion, it is just a historical accident that people are used to write the full sequents involved while doing proofs in Gentzen-type system, while more economical methods are used for N.D. systems. Repeating passive formulas again and again is really what make the use of Gentzen-systems tedious. It is highly desirable therefore to develop methods for avoiding it in *practical* implementations of such systems. A promising recent contribution in this direction is Girard's concepts of *proof-nets* (see [Gi]). The use of tableaux systems can also be regarded as a method of this kind.

To summerize: in pure Hilbert system the rules [23] are of the form

$$\frac{\Delta_1, \ldots, \Delta_n}{\Delta}\,.$$

In pure N.D. systems they have the form:

$$\frac{\begin{matrix}[\Gamma_1] \\ \Delta_1\end{matrix} \;\cdots\; \begin{matrix}[\Gamma_n] \\ \Delta_n\end{matrix}}{\Delta}\,.$$

And in pure Gentzen-type systems:

$$\frac{\begin{matrix}[\Gamma_1] \\ \Delta_1\end{matrix} \;\cdots\; \begin{matrix}[\Gamma_n] \\ \Delta_n\end{matrix}}{\begin{matrix}[\Gamma] \\ \Delta\end{matrix}}\,.$$

[22] This reminds the form which some rules take in Schroeder-Heister's calculus of higher-order rules. More on the connection between this calculus and Gentzen type calculi can be found in [Av3].

[23] The notion of a "rule" is ambiguous in papers discussing topics of the kind dealt with in the present one, since it has both a global and a local meaning. Thus one can talk about the global rule of adjunction which permits the inference of $A \wedge B$ from A and B for every A and B (here A and B are metavariables), and can also (as in [SH]) regard any inference of $A \wedge B$ from *particular* A and B as application of a local rule which allow inferring $A \wedge B$ from A and B for these particular A and B . We have tried our best here to formulate our definitions and claims to be independent of the interpretation of the term "rule" (but at least for axiomatic systems we usually have in mind the global interpretation).

A major fact about Gentzen-type systems, even pure ones, is that unlike pure Hilbert-type and N.D. systems, the transitivity (= cut) is never obvious. In the previous types of systems it was a direct result of the asymmetry in rules between antecedents and succedents. In Gentzen-type systems this asymmetry is abolished, and so the cut-rule should either be taken as an explicit rule (as frequently it should) or else be proved admissible. This explains the fact that cut-elimination which is crucial for *every* formal system, was (and is) investigated only in the context of Gentzen-type systems.

Usually, a Gentzen-type representation of a C.R. has the following form: It has as axioms the reflexivity axioms $A \Rightarrow A$ (sometimes it suffices to take only a subset off these and then derive the rest). The rules are then divided into two groups:

- Structural rules, which operate on the multisets, but do not change the formulas in them (though it may add or delete some). Frequent examples are: weakening, contraction, anticontraction, mingle (from $\Gamma_1 \Rightarrow \Delta_1$ and $\Gamma_2 \Rightarrow \Delta_2$ infer $\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2$) and, of course, cut.

- Rules concerning the constants of the language. Usually they are divided into succedent rules, which specify how a logical constant may be introduced in the succedent of a sequent, and antecedent rules, which do the same for the antecedent.

In order to judge how good a given Gentzen-type presentation of a given C.R. is, we should apply exactly the same criterions as we did for N.D. systems. However, since we allow more in Gentzen-type systems we are usually entitled to demand more. Hence we usually should expect it to be easier to demonstrate the validity of a formal sequent in a Gentzen-type formalism than it is in a corresponding N.D. system. Also we can expect nice proof theoretic properties to hold. such properties are usually proved by induction, the major step of which is showing that the various rules preserve the property under discussion. Usually the cut rule is the major obstacle while providing such a proof. A very good example for this is the important subformula property. This property is preserved by all the rules we found as characteristic for the various intentional and combining connectives. It is preserved also by all the structural rules considered in the last section—except cut. If we delete cut from the set of rules of each of the systems we discuss in the last section we get therefore formal systems with the subformula property. The cut-elimination theorem (which obtains for each of those systems) mean that these new systems are still representations of C.R.'s and that the C.R.'s they represent are identical to the original ones.

Note: Since the antecedent and succedent rules completely specify how a connective can be introduced into sequents, it can be argued that they operationally define that connective. This claim is strongly related to the analogous claim concerning introduction and elimination rules in N.D. systems. I believe that a much better case can be made to this claim in the present context than in the previous one.[24] Nevertheless, I still think that it is tenable only for special types of connectives. After all, the rules of Gentzen-type systems (or even N.D. systems) do not necessarily treat one connective at a time. The following rule, e.g. is frequently a useful one (see next section),but it treats two:

$$\frac{\Gamma_1 \Rightarrow \Delta_1, \neg A \quad \Gamma_2 \Rightarrow \Delta_2, \neg B}{\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2, \neg(A \wedge B)}\ .$$

In systems with rules like the last one we might be forced to say that the various connectives are simultaneously defined by the set of the rules (taken as a whole). This does not seem to be very helpful, though. (The distinction between antecedent and succedent rules is not so sharp, by the way. In modal logics, for example, we frequently have rules which introduce new connectives on *both* sides of a sequent, like the following rule of the minimal normal modal logic K:

$$\frac{A_1, \ldots, A_n \Rightarrow B}{\Box A_1, \ldots, \Box A_n \Rightarrow \Box B}\ .$$

It might be claimed that this rule completely defines the box in the context of K, but such a claim might be more difficult to swallow than similar claims for more regular rules. Note that rules like the last one is not available in N.D. formalisms!)

5.5 Three Degrees of Impurity

In the previous subsections a great deal of importance was attached to what we call pure systems, The most general definition of which was given while discussing Gentzen-type representations. Now the reasons for the importance of pure systems is due to the fact that they are relatively easy to implement in an economical way, and it is easy also to check proofs in them. [25] However, there are also known representations which are not pure. Our purpose now is to try to identify the most usual forms that impurity may take. We assume, accordingly, that we are dealing with a formal system of one of the types described above. Moreover, we assume that the inference rules leading from sequents to sequents are given

[24] See, in relation to this claim the first chapter of [Ga] and the paper of Hacking [Ha].

[25] Every ordinary, pure single-conclusioned N.D. system can, e.g., quite easily be implemented on the Edinburgh LF, which is a general proof-development environment, based on a general Logical Framework which was developed in the computer science department of Edinburgh university.

by some global rule-schemes of the form described after the definition of a pure Gentzen-type system (in the last subsection), but that there are side conditions on the applicability of some of these schemes which make them impure. We shall examine three possible ways in which this might happen:

Level 1: At this level the side conditions are related to the *structure* of the multisets of the side-formulas (Γ_i'', Δ_i'' above). Examples of such side conditions are:

- Demanding that there be no side formulas. In Hilbert-type systems rules with this conditions are usually known as *rules of proof.* The best known example is the necessitation rule in traditional formulations of normal modal logics. Other example is the adjunction rule of many relevance logics (from A and B infer $A \wedge B$) which frequently is taken to be only a rule of proof.
- Demanding, in a multiple-conclusioned C.R., the antecedents of the conclusion and the hypotheses of a rule to be singletons. Examples are the rule for the $\Box$ in the Gentzen-type system for the minimal normal modal logic K (see above) and the succedent rule for the intuitionistic implication in the multiple-conclusioned Gentzen-type version which we mention in section 4.4.
- Demanding all the hypotheses of a rule to have *exactly* the same side-formulas. Examples are provided by the rules for the combining connectives in section 4.3.. For logics without contraction or weakening, like relevance logics and linear logic, these rules are impure (and this explains why Girard has found (in [Gi]) the "multiplicative" fragment of his logic, which is pure, to be better understood than the "additive" one, which is not).

Level 2: Here an applicability of a rule might depend also on the structure of the side-formulas. Examples are:

- The introduction rule for the $\Box$ in Prawitz N.D. system for $S4$. This rule permits the inference of $\Gamma \Rightarrow \Box A$ from $\Gamma \Rightarrow A$ only if all the formulas in Γ begin with $\Box$.
- The introduction rule for $\forall$ in the N.D. systems for classical and intuitionistic logics. Here one can infer $\Gamma \Rightarrow \forall x A$ from $\Gamma \Rightarrow A$ only if x does not occur free in Γ. (This is not the whole story, though, since this rule might become pure in the context of non-simple C.R.'s. There are in fact no problems to deal with *this* kind of impurity in the Edinburgh LF!).

Level 3: Here an applicability of a rule might depend not only on its potential premises, but also *on their proofs*. A possible example may be provided by an attempt to base a N.D. system on the following version of the deduction theorem in classical first-order logic: [26] $\Gamma \vdash A \rightarrow B$ iff there is a proof of B from Γ, A in which no inference of $\forall x C$ from C is made in which x is free in A and C depends on A (in that proof).

The third level of impurity carries us, in fact, beyond the class of *uniform* systems [27], with which we were dealing so far. The main properties of uniform systems are that they treat exactly one C.R., and that once a sequent was derived in them one can completely forget about *how* it was derived while using it for deriving other sequents. For systems of the third level of impurity this is no longer the case. Such systems are very inefficient in the time and space required for proof checking. It is advisable, therefore to avoid the use of such systems. In the next section we shall suggest some possible methods for doing this when uniform representations are not available or the available ones are inefficient.

6 Non-Uniform Representations

As we have seen in the previous section, all the standard proof systems are examples of uniform representations of consequence relations. However, if we accept that a formal system is essentially a device for deriving correct sequents (of a C.R. in which we happen to be interested) then there is no reason to limit ourselves to uniform formal systems. I believe that this observation might open the door to a new area of investigations with a wealth of promising possibilities. To demonstrate the potential of non-uniform representations we shall describe now two methods of developing such representations together with examples of their applicability. We hope that other efficient methods will be developed in the future.

6.1 Treating Several Consequence Relations Simultaneously

A major feature of uniform representations is that they treat only a single C.R.. In mathematics it is often much more efficient to simultaneously solve *several*

[26] See, e.g., in [Men]. This theorem is true for the pure Hilbert type system for *validity* which is presented there.

[27] A formal definition of a uniform system will be identical to that of a Gentzen-type system and was already given in 5.4.

related problems. The problems of representing related consequence relations should not be an exception in this respect. A good example for this is provided by the two C.R.s which we have associated with modal logics in 3.3 . Obviously, there are strong connections between them. On the other hand it is difficult to provide a nice pure representation of either, since each lacks some important properties (which the other has). It is reasonable, therefore, to try to represent them together in one formal system. This really can be done (at least for natural modal systems like $S4, T, K4$ and so on). Examples of rules of the resulting system in the case of $S4$ are:

$$\frac{\Gamma \vdash_v A}{\Gamma \vdash_v \Box A} \quad \frac{A, \Gamma \vdash_t B, \Delta}{\Gamma \vdash_t A \to B, \Delta} \quad \frac{\Gamma, A \vdash_v B}{\Gamma \vdash_v \Box A \to B} \quad \frac{\Gamma \vdash_t A}{\Gamma \vdash_v A} \quad \frac{\vdash_v A}{\vdash_t A} \,.$$

In this system $\vdash_t$ is taken as multiple conclusioned while $\vdash_v$ as single conclusioned. All the impurities of the usual representations are eliminated in the combined one (or—depending on the meaning of "pure" in this context— are reduced to impurities of the first degree - see 5.5.)

It is worth noting that a suitable variation of this system was actually used by the author in order to solve the problem of efficiently internalizing $S4$ in the computerized Edinburgh LF system (see [AHM]).

6.2 Higher-Order Sequents

We start with the following observation: Uniform representations are axiomatic systems in which the wffs are formal sequents. Past experience shows that it has frequently been useful to extend such a system to a multiple-conclusioned one even if the ultimate interest remained proving theoremhood of wffs. It is natural therefore to try applying the same process in the present case. This naturally leads us to consider "hypersequents" which are sequents of sequents. It might be enough to start by considering only one-side hypersequents . Can they be useful? The answer is "yes", and in what follows we provide two examples which demonstrate this claim.

In order to understand the examples, let us return to the five 3-valued consequence relations of 3.4. It is not difficult to provide uniform, cut free Gentzen-type presentations of the first three. It is enough, e.g., to take as logical rules for all of them the usual rules for the (combining) $\vee$ and $\wedge$, as well as the obvious rules for $\neg\vee$, $\neg\wedge$ and $\neg\neg$ (on both sides). The systems will differ then only with respect to the axioms ($P \Rightarrow P$ (P atomic) will be axioms for all. In addition we have $P, \neg P \Rightarrow_{Kl}$, $\Rightarrow_{PAC} P, \neg P$ and $\neg P, P \Rightarrow_{Lt} \neg Q, Q$). For the other two we should work with hypersequents, though! The needed changes are:

1. We should work with multisets of sequents, both sides of which are multisets of formulas.

2. We should add the *external* structural rules:

   External Contraction

$$\frac{(\Gamma_1 \Rightarrow \Delta_1), \ldots, (\Gamma_n \Rightarrow \Delta_n)}{(\Gamma \Rightarrow \Delta), (\Gamma_1 \Rightarrow \Delta_1), \ldots, (\Gamma_n \Rightarrow \Delta_n)} \; .$$

   External Weakening

$$\frac{(\Gamma \Rightarrow \Delta), (\Gamma \Rightarrow \Delta), (\Gamma_1 \Rightarrow \Delta_1), \ldots, (\Gamma_n \Rightarrow \Delta_n)}{(\Gamma \Rightarrow \Delta), (\Gamma_1 \Rightarrow \Delta_1), \ldots, (\Gamma_n \Rightarrow \Delta_n)} \; .$$

3. We replace the six $\neg$ rules by the usual two rules for internal negation.

4. Internal weakening is not a rule in the case of $\vdash_{Sob}$ while internal contraction is not a rule in the case of $\vdash_{Luk}$.

5. The logical rules and the permitted internal structural rules should be reformulated so that "side-sequents" are allowed. For example $\vee \Rightarrow$ takes the form:

$$\frac{(A, \Gamma \Rightarrow \Delta), (\Gamma_1 \Rightarrow \Delta_1), \ldots \quad (B, \Gamma \Rightarrow \Delta), (\Gamma_1' \Rightarrow \Delta_1'), \ldots}{(A \vee B, \Gamma \Rightarrow \Delta), (\Gamma_1 \Rightarrow \Delta_1), \ldots, (\Gamma_1' \Rightarrow \Delta_1'), \ldots} \; .$$

6. Instead of the deleted internal structural rules we should add:

   For $\vdash_{Sob}$: The following weaker versions of weakening *(with possible side-sequents)*:

$$\frac{(\Gamma_1 \Rightarrow \Delta_1) \quad (\Gamma_2 \Rightarrow \Delta_2)}{(\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2)}$$

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2}{(\Gamma_1 \Rightarrow \Delta_1), (\Gamma_2, \Gamma' \Rightarrow \Delta_2, \Delta')}$$

   For $\vdash_{Luk}$: We add the following *splitting* rule (again with possible side-sequents):

$$\frac{(\Gamma_1, \Gamma_2, \Gamma_3 \Rightarrow \Delta_1, \Delta_2, \Delta_3) \quad (\Gamma_1', \Gamma_2', \Gamma_3' \Rightarrow \Delta_1', \Delta_2', \Delta_3')}{(\Gamma_1, \Gamma_1' \Rightarrow \Delta_1, \Delta_1'), (\Gamma_2, \Gamma_2' \Rightarrow \Delta_2, \Delta_2'), (\Gamma_3, \Gamma_3' \Rightarrow \Delta_3, \Delta_3')} \; .$$

Notes:

1. The proofs that by this method we really get sound and complete representation in which cut-elimination is admissible are not too difficult, since easy reduction steps are available. It is much harder, yet possible, to prove the same properties if we add internal implication to both (getting the full systems RM_3 and Lukasiewicz L_3, respectively). We shall present all these proofs in a forthcoming paper.

2. Cut-free Hypersequential calculi were first introduced in [Pot] and, independently, in [Av2]. In the forthcoming paper mentioned above we shall extend the method to other Logics (e.g. the system LC of Dummet) and supply all the proofs.

3. It is possible, of course, to consider further iterations of the above procedure.[28] The examples we have given should suffice for clarifying the main idea, though.

7 Acknowledgements

I like to thank Furio Honsell and Ian Mason for the encouragement they have given me while working on this paper and for a lot of stimulating discussions and helpful comments. This paper would have never been written without them. Thanks also for the other participants of the LF project: B. Harper and G. Plotkin for their comments and help.

8 References

[**AB**] Anderson A.R. and Belnap N.D., **Entailment** vol. 1, Princeton University Press, Princeton,N.J., 1975.

[**AHM**] Avron A., Honsell F. and Mason I. *Using judgements to implement logics on a machine*, Technical Report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1987. (In preparation).

[**Av1**] Avron A. *Relevant entailment: semantics and formal systems*, J.S.L. vol. 49 (1984), pp. 334-342.

[**Av2**] Avron A. *A constructive analysis of RM*, to appear in the J.S.L.

[**Av3**] Avron A. *Gentzenizing Schroeder-Heister's Natural extension of natural deduction*, to appear.

[**Do**] Došen K. *Sequent-systems for modal logic*, J.S.L., vol. 50 (1985), pp. 149-168.

[**Du**] Dunn J.M. *Relevant logic and entailment*, in: **Handbook of Philosophical Logic**, Vol III, ed. by D. Gabbay and F. Guenthner, Reidel: Dordrecht, Holland; Boston: U.S.A. (1984).

[28] In the literature one can already find uses of sequents of arbitrary degree of nesting in [SH] and [Do].

[Ga] Gabbay D. **Semantical investigations in Heyting's intuitionistic logic** , Reidel: Dordrecht, Holland; Boston: U.S.A. (1981).

[Gen] Gentzen G. *Investigations into logical deduction*, in: **The collected work of Gerhard Gentzen**, edited by M.E. Szabo, North-Holland, Amsterdam, (1969).

[Gi] Girard J.Y. *Linear Logic*, to appear in T.C.S.

[Ha] Hacking I. *What is logic*, The journal of philosophy, vol. 76 (1979), pp. 285-318.

[Ho] Hodges W. *Elementary predicate calculus*, in: **Handbook of Philosophical Logic**, Vol I, ed. by D. Gabbay and F. Guenthner, Reidel: Dordrecht, Holland; Boston: U.S.A. (1983).

[HHP] Harper R., Honsell F., and Plotkin G., *A Framework for Defining Logics*, Proceedings of the second annual conference on Logic in Computer science, Cornell, (1987).

[Men] Mendelson E. **Introduction to Mathematical Logic**, Princeton, Van Nostrad,(1964).

[Pot] Pottinger G. *Uniform, Cut-free formulations of T, S4 and S5*, (abstract), J.S.L., vol. 48 (1983), p. 900.

[Pra1] Prawitz D. **Natural Deduction**, Almqvist&Wiksell,Stockholm (1965).

[Pra2] Prawitz D.*Ideas and results in Proof theory*, in: J.E. Fenstad (ed.), **Proceedings of the second scandinavian Logic symposium**, North-Holland, Amsterdam, pp. 235-307, (1973).

[Pri] Prior A.N. *The runabout inference-ticket*, Analysis, vol. 21 (1960), pp. 38-39.

[Sc1] Scott D. *Rules and derived rules*, in: Stenlund S. (ed.), **Logical theory and semantical analysis**, Reidel: Dordrecht (1974), pp. 147-161.

[Sc2] Scott D. *Completeness and axiomatizability in many-valued logic*, in: **Proceeding of the Tarski Symposium**, Proceeding of Symposia in Pure Mathematics, vol. XXV, American Mathematical Society, Rhode Island, (1974), pp. 411-435.

[SH] Schroeder-Heister P. *A natural extension of natural deduction*J.S.L., vol. 49 (1984), pp. 1284-1300.

[**Sob**] Sobociński B. *Axiomatization of partial system of three-valued calculus of propositions*, The journal of computing systems, vol 11. 1 (1952), pp. 23-55.

[**Sud1**] Sundholm G. *Systems of deduction*, in: **Handbook of Philosophical Logic**, Vol I, ed. by D. Gabbay and F. Guenthner, Reidel: Dordrecht, Holland; Boston: U.S.A. (1983).

[**Sud1**] Sundholm G. *Proof theory and meaning*, in: **Handbook of Philosophical Logic**, Vol III, ed. by D. Gabbay and F. Guenthner, Reidel: Dordrecht, Holland; Boston: U.S.A. (1984).

[**Tak**] Takeuti G. **Proof theory**, North-Holland, Amsterdam, (1975).

[**Ur**] Urquhart A. *Many-valued Logic*, in: **Handbook of Philosophical Logic**, Vol III, ed. by D. Gabbay and F. Guenthner, Reidel: Dordrecht, Holland; Boston: U.S.A. (1984).

Contents

1 Introduction 1

2 The Notion of a Consequence Relation 2

2.1 Axiomatic Systems 2

2.2 Consequence Relations 3

2.3 Remarks and variations 3

3 Some Examples of Abstract Consequence Relations 5

3.1 Classical Propositional Logic 5

3.2 First-order logic 5

3.3 Propositional Modal Logic 6

3.4 Three-Valued Logic 6

4 Classification of C.R. according to their Basic Connectives 8

4.1 Intensional connectives 8

4.2 Characterizing the Intensional Connectives by Gentzen-Type Rules 10

4.3 The combining connectives 12

4.4 Characterization of some Known Logics 14

4.5 On the Meanings of the Propositional Ponnectives 17

5 Uniform Representations of C.R's 18

5.1 Using Axiomatic Systems for Representations 19

5.2 Hilbert Type Representations 20

5.3 Natural-Deduction Representations 24

5.4 Gentzen-Type Representations 27

5.5 Three Degrees of Impurity 31

6 Non-Uniform Representations 33

6.1 Treating Several Consequence Relations Simultaneously 33

6.2 Higher-Order Sequents 34

7 Acknowledgements 36

8 References 36

Some remarks about interactive proof-checkers

Th. COQUAND

Edinburgh, february 87

8

. project started in 85

. to have a "natural" representation of higher-ord

logic in an Automath-like language

. N. De Bruijn Aut 4 (74)

syntax : the one of Automath

$. M = x \mid M(M) \mid (\lambda x{:}M)M \mid (\Pi x{:}M)M$

. two special identifiers (kinds)

Prop and Type

. all we have to do to contain higher-ord

logic + $\left\{\begin{matrix}\text{Girard} \\ \text{McCracken}\end{matrix}\right.$'s type system is to

declare Prop : Type

<u>valid context</u> . $\emptyset$ valid

. $\Gamma \vdash A : K \rightarrow \Gamma, x : A$ valid

<u>typing rules</u>

. $\Gamma \vdash x : A$ if $x : A \in \Gamma$

. $\Gamma, x : A \vdash t : B \longrightarrow \Gamma \vdash (\lambda x : A) t : (x : A) B$

. $\Gamma, x : A \vdash B : \kappa \longrightarrow \Gamma \vdash (x : A) B : \kappa$

. $\Gamma \vdash t : (x : A) B, \Gamma \vdash u : A \rightarrow \Gamma \vdash t(u) : [u / x] B$

. $\Gamma \vdash t : A, \Gamma \vdash B : \kappa, A = B \rightarrow \Gamma \vdash t : B$

Martin-Löf 71 inductive definition

. β-conversion : simplicity

. Do we need η-conversion ?

. Definition mechanism ?

Why higher-order logic?

. historically, type-system which contains Girard's type system (hence Reynold's one)

. more powerful <u>and</u> as "easy" to implement as Automath ("system" independant)

. nice to study foundation problems as Girard's paradox (without "coding")

. Real problem where the computer is used to handle combinatorial complexity (on proofs)

. Problem : add to Church's type theory

. type variable $\alpha, \beta ..$

. product over types $\Pi\alpha . \alpha \to \alpha$

. quantification over types

then show that this logic is inconsistent

. behaviour of the proof of the inconsistency by cut-elimination

. <u>definition mechanism</u> : it's possible to do this analysis only with a type-checker

. comparaison with D. Howe NPrl

Problems

- substitution , equality
- memory allocation
- theory

Polymorphism

- user's facility : the core type system is the same
- example

pairing A: Type , B: Type , x: A , y: B ...

witness A: Type, B: (A) Prop, x: A, p: B(x) —

pair (A | Type) (B | Type) (x: A) (y: B) A × B

- Huet 's course notes.
- typical ambiguity , inclusion
- enough for a lot of applications e.g. category theory

A more interactive approach

. goal : Automath + Isabelle

. each identifier is the name of a rule , its type

. this rule is a generalized Horn clauses

$(x_1 : A_1) \dots (x_p : A_p)\, A$

. "higher-level" Prolog

Tarski fixed point theorem

$A : \text{Type}$, $f : (A)A$, $R : (A)(A)\,\text{Prop}$, $E : (A)(A)\,\text{Prop}$

eq $\quad (x:A)(y:A)(R(x,y))(R(y,x))\quad E(x,y)$

trans $\quad (x:A)(y:A)(z:A)(R(x,y))(R(y,z))\quad R(x,z)$

incr $\quad (x:A)(y:A)\ (R(x,y))\quad R(f(x),f(y))$

upp $\quad (x:A)\ (R(x,f(x)))\quad R(x,M)$

low $\quad (x:A)((y:A)(R(y,f(y)))\ R(y,x))\ R(M,x)$

Problem $\quad ?x \quad E(x,f(x))$

Prolog where the set of rules and constants is dynamic.

```
***************************************************************************************************************
***************************************************************************************************************
**                                                                                                           **
**                                  tfhc - steve - Feb 21 10:33 - exemple5.ML                                **
**                                                                                                           **
***************************************************************************************************************
***************************************************************************************************************
```

```
(* Tarski where we say that we consider the set
  {x \in A / x R f(x)}
  in the usual set notation. Notice the way that we express that
  M is the LEAST upper bound. This is expressed as a RULE with
  nested implication and universal quantification *)

(* The data *)

Decl"P0""Prop";
Decl"A""Type";Decl"R""(A)(A)Prop";Decl"E""(A)(A)Prop";
Decl"f""(A)A";Decl"M""A";

(* the axioms, actually, inference rule *)

Decl "eq" "(x:A)(y:A)(R(x,y))(R(y,x))E(x,y)";

Decl "trans"    "(x:A)(y:A)(z:A)(R(y,z))(R(x,y))R(x,z)";

Decl"incr"      "(x:A)(y:A)(R(x,y))R(f(x),f(y))";

Decl"upp"       "(x:A)(R(x,f(x)))R(x,M)";

Decl"low"       "(x:A)((y:A)(R(y,f(y)))R(y,x))R(M,x)";

Decl"H"         "(x:A)(E(x,f(x)))P0";

(*
?x2 : ((?x1)f)(?x1)E
?x1 : A

    ----------------------------

?x6 : (?x3)((?x3)f)R
?x5 : ((?x3)f)(?x3)R
?x3 : A

    ----------------------------

?x8 : (((M)f)f)((M)f)R
?x5 : ((M)f)(M)R

    ----------------------------

?x11 : ((M)f)(M)R
?x5 : ((M)f)(M)R

    ----------------------------

?x13 : (x1:A)(x2:((x1)f)(x1)R)((M)f)(x1)R

    ----------------------------

 Introduction de X0

 Introduction de X1
?x20 : (?x17)(X0)R
?x19 : ((M)f)(?x17)R
?x17 : A

    ----------------------------

?x19 : ((M)f)((X0)f)R

    ----------------------------

?x23 : (M)(X0)R

    ----------------------------

?x25 : ((X0)f)(X0)R

    ----------------------------

    ----------------------------
j'ai trouve
la reponse est :
(((((Lx1.Lx2.(x2)(((x2)(x1)upp)(M)(x1)incr)((M)f)((x1)f)(x1)trans)((M)f)low)((M)f)(M)incr)((M)f)upp)((Lx1.Lx2.(
x2)(((x2)(x1)upp)(M)(x1)incr)((M)f)((x1)f)(x1)trans)((M)f)low)((M)f)(M)eq)(M)H
Timing  compile-0.9s (0.8s+0.0s+0.0s+0.0s+0.1s) run-3.9s

*)
```

→ AUTO();
with trace

```
************************************************************************************************************************
**                                                                                                                    **
**                                   tfhc - steve - Feb 21 10:34 - Logic.ML                                           **
**                                                                                                                    **
************************************************************************************************************************
************************************************************************************************************************

(* The conjonction in second-order minimal calculus *)

fun DEF c s = (eval c;Egal s) ;

Decl "p""Prop";Decl "q" "Prop";

Goal "(p)((p)q)q";AUTO();SAVE1"cut";

Goal "((p)q)(p)q";AUTO();SAVE1"cut1";

fun CUT h = (LET ("cut("^h^")") "l";SOLVE "l");

DEF"(r:Prop)((p)(q)r)r""and";

(* un exemple : la construction des projections *)

Goal"(and(p,q))p";INTRO "h1";SIMPLIFY"h1"["h11","h12"];AUTO();SAVE1"p1";

Goal"(and(p,q))q";INTRO "h1":SIMPLIFY"h1"["h11"."h12"];AUTO();SAVE1"p2";

Goal"(p)(q)and(p,q)";INTRO"h1";INTRO"h2";REDUCT();AUTO();SAVE1"intro_and";

(* the equivalence *)

eval "and((p)q,(q)p)";Egal"equiv";

(* how to prove an equivalence *)

Goal"((p)q)((q)p)equiv(p,q)";INTRO"h1";INTRO"h2";
REDUCT():SOLVE"intro_and";AUTO();SAVE1"equiv_intro";

Goal "(p)(equiv(p,q))q";
SAVE1"equiv1";fun TAC_equiv1 h = (SOLVE "equiv1";SOLVE h);

Goal "(p)(equiv(q,p))q";
SAVE1"equiv2";fun TAC_equiv2 h = (SOLVE "equiv2";SOLVE h);

(* the equivalence is transitive *)

Decl "r" "Prop";
Goal "(equiv(p,q))(equiv(q,r))equiv(p,r)";
INTRO"h1";INTRO"h2";SIMPLIFY "h1" ["h12","h11"];SIMPLIFY "h2" ["h22","h21"];
SOLVE"equiv_intro";
INTRO"h3";SOLVE"h11";SOLVE"h21";SOLVE"h3";PAS();
INTRO"h3";SOLVE"h22";SOLVE"h12";SOLVE"h3";AUTO();
SAVE1"trans_equiv";

Goal "(equiv(q,r))(equiv(p,q))equiv(p,r)";
INTRO"h1";INTRO"h2";SOLVE"trans_equiv";AUTO();SAVE1"trans1_equiv";

Abs"p";

(* the inclusion *)

Decl"A""Type";Decl"P""(A)Prop";Decl"Q""(A)Prop";

DEF "(x:A)(P(x))Q(x)" "inclus";

Decl"R""(A)Prop";Goal"(inclus(A,Q,R))(inclus(A,P,Q))inclus(A,P,R)";
INT1["h1","h2"];REDUCT();INT1["x","h3"];SIMPLIFY"h1"[];SIMPLIFY"h2"[];
AUTO();SAVE1"trans_inclus";

Abs"R";Abs"Q";Goal"inclus(A,P,P)";REDUCT();AUTO();SAVE1"ref_inclus";

Decl"Q""(A)Prop";DEF"[x:A]and(P(x),Q(x))""inter";

Goal"inclus(A,inter(A,P,Q),P)";
REDUCT();INT1["x","h1"];SIMPLIFY"h1"["h11","h12"];AUTO();SAVE1"int1";

Goal"inclus(A,inter(A,P,Q),Q)";
REDUCT();INT1["x","h1"];SIMPLIFY"h1"["h11","h12"];AUTO();SAVE1"int2";

Decl"R""(A)Prop";Goal"(inclus(A,R,P))(inclus(A,R,Q))inclus(A,R,inter(A,P,Q))";
INT1["h1","h2"];REDUCT();INT1["x","h3"];REDUCT();SOLVE"intro_and";
SIMPLIFY"h2"[];SOLVE"h3";SIMPLIFY"h1"[];SOLVE"h3";AUTO();SAVE1"inter_least";

Abs"R";Goal"(x:A)(P(x))(Q(x))inter(A,P,Q,x)";
```

```
INT1["x","h1","h2"];REDUCT();SOLVE"intro_and";AUTO();SAVE1"inter_intro";

Abs"Q";
DEF "(p:Prop)((x:A)(P(x))p)p""exists";

Goal"(x:A)(P(x))exists(A,P)";
INT1["x","h"];REDUCT();AUTO();
SAVE1"witness";

Abs"A";

DEF "(p:Prop)p" "abs" ;
```

```
********************************************************************************************
********************************************************************************************
**                                                                                        **
**                      tfhc - steve - Feb 21 10:33 - Scott.ML                            **
**                                                                                        **
********************************************************************************************
********************************************************************************************

Decl"A""Type";
Decl"R""(A)(A)Prop";

LET"[P:(A)Prop][x:A][y:A]exists(A,inter(A,P,inter(A,R(x),R(y))))""confluent";

Goal"(P:(A)Prop)(x:A)(y:A)(z:A)(P(z))(R(x,z))(R(y,z))confluent(P,x,y)";
INT1["P","x","y","z","h1","h2","h3"];REDUCT();AUTO();SAVE"confluent_intro";

fun ELIM_confluent s z h1 h2 h3 =
(SIMPLIFY s [z,"a"];SIMPLIFY "a" [h1,"a"];SIMPLIFY "a" [h2,h3]) ;

LET"[P:(A)Prop](x:A)(P(x))(y:A)(P(y))confluent(P,x,y)""directed";

Decl"f""(A)A";
LET"[P:(A)Prop][x:A](Q:(A)Prop)((y:A)(P(y))Q(f(y)))Q(x)""Image";

Goal"(P:(A)Prop)(x:A)(P(x))Image(P,f(x))";
INT1["P","x","h1"];REDUCT();AUTO();SAVE"Image_intro";

fun ELIM_Image s z h1 = SIMPLIFY s [z,h1];

Decl "Incr""(x:A)(y:A)(R(x,y))R(f(x),f(y))";

Goal"(P:(A)Prop)(directed(P))directed(Image(P))";
INT1["P","h1"];REDUCT();INT1["x","h2","y","h3"];
LET"[u:A]confluent(Image(P),u,y)""P1";EGAL"P1(x)";ELIM_Image"h2""x1""h21";
REDUCT();ELIM_Image"h3""y1""h31";LET"h1(x1,h21,y1,h31)""l";
ELIM_confluent"l""z""h4""h5""h6";AUTO();SAVE"lemme1";

Decl"Lim""((A)Prop)A";

Decl"Upperb""(P:(A)Prop)(x:A)(P(x))R(x,Lim(P))";
Decl"Least" "(P:(A)Prop)(x:A)((y:A)(P(y))R(y,x))R(Lim(P),x)";

Goal"(P:(A)Prop)R(Lim(Image(P)),f(Lim(P)))";
INTRO"P";SOLVE"Least";INT1["x","h1"];LET"[u:A]R(u,f(Lim(P)))""P1";EGAL"P1(x)";
ELIM_Image"h1""x1""h11";REDUCT();AUTO();SAVE"lemme2";

LET "[x:A](p:Prop)p""empty";

fun ELIM_empty s = SIMPLIFY s [];

Goal"directed(empty)";
REDUCT();INT1["x","h1","y","h2"];ELIM_empty"h1";AUTO();SAVE"lemme3";

LET"Lim(empty)""bottom";

Goal"(x:A)R(bottom,x)";
INTRO"x";EGAL"R(Lim(empty),x)";PAS();INT1["y","h1"];ELIM_empty"h1";AUTO();
SAVE"minimum";

LET"[x:A](P:(A)Prop)((x:A)(P(x))P(f(x)))(P(bottom))P(x)""Approx";

fun ELIM_Approx s = SIMPLIFY s [] ;

Goal"Approx(bottom)";REDUCT();AUTO();SAVE"Approx_bottom";

Goal"(x:A)(Approx(x))Approx(f(x))";
INT1["x","h1"];REDUCT();INT1["P","h2","h3"];PAS();ELIM_Approx "h1";AUTO();
SAVE"Approx_stable";

Goal"(x:A)(Approx(x))R(x,f(x))";
INT1["x","h1"];LET"[u:A]R(u,f(u))""P1";EGAL"P1(x)";ELIM_Approx "h1";
REDUCT();PAS();REDUCT();INT1["y","h2"];RED"h2";REDUCT();AUTO();
SAVE"proposition1";

Decl"Ref""(x:A)R(x,x)";
Decl"Trans""(x:A)(y:A)(z:A)(R(y,z))(R(x,y))R(x,z)";

Goal"directed(Approx)";
REDUCT();INT1["x","h1","y","h2"];ELIM_Approx"h2";PAS();SOLVE"minimum";
SOLVE"x";SOLVE"Ref";PAS();INT1["z","h3"];ELIM_confluent"h3""t""h4""h5""h6";
PAS();SOLVE"Incr";PAS();SOLVE"Trans";SOLVE"proposition1";PAS();SOLVE"Incr";
AUTO();SAVE"proposition2";

Goal"(P1:(A)Prop)(P2:(A)Prop)(inclus(A,P1,P2))R(Lim(P1),Lim(P2))";
INT1["P1","P2","h1"]:RED"h1";SOLVE"Least":INT1["x"."h2"]:SOLVE"Upperb";
```

```
AUTO();SAVE"lemme4";

Decl"Cont""(P:(A)Prop)(directed(P))R(f(Lim(P)),Lim(Image(P)))";

Goal"and(R(Lim(Approx),f(Lim(Approx))),R(f(Lim(Approx)),Lim(Approx)))";
PAS();PAS();SOLVE"Cont";PAS();PAS();REDUCT();INT1["x","h1"];
ELIM_Image"h1""x1""h11";PAS();PAS();AUTO2();
SOLVE"Least";INT1["x","h1"];PAS();SOLVE"proposition1";PAS();SOLVE"Incr";
SOLVE"Upperb";AUTO();SAVE"theoreme1";
```

```
(* classical logic. Uses Logic *)

fun PRIM c s = (Goal c;SAVE1 s) ;

PRIM "(p:Prop)(((p)abs)abs)p" "cl";

DEF "[p:Prop](p)abs" "not";

DEF "[p:Prop][q:Prop](not(p))(not(q))abs" "vel";

Goal "(p:Prop)(q:Prop)(p)vel(p,q)";
INT1["p","q","h1"];REDUCT ();INT1["h2","h3"];
LET_INTRO"h2(h1)""l";AUTO();SAVE1 "or_intro1";

Goal "(p:Prop)(q:Prop)(q)vel(p,q)";
INT1["p","q","h1"];REDUCT ();INT1["h2","h3"];
LET_INTRO"h3(h1)""l";AUTO();SAVE1 "or_intro2";

fun TAC_cl h = (SOLVE"cl";INTRO h) ;

Goal "(p:Prop)(q:Prop)(r:Prop)((p)r)((q)r)(vel(p,q))r";
INT1["p","q","r","h1","h2","h3"];RED"h3";TAC_cl"h4";
SOLVE"h3";
REDUCT();INTRO"h5";SOLVE"h4";SOLVE"h2";SOLVE"h5";AUTO2();
REDUCT();INTRO"h5";SOLVE"h4";SOLVE"h1";SOLVE"h5";AUTO();
SAVE1"case";

fun ELIM_vel h1 = (SOLVE "case";SOLVE h1);

Goal "(p:Prop)(q:Prop)(vel(p,q))vel(q,p)";
INT1["p","q","h1"];SOLVE"case";SOLVE"h1";
INTRO"h2";SOLVE"or_intro1";SOLVE"h2";AUTO2();
INTRO"h2";SOLVE"or_intro2";AUTO();SAVE1"sym_vel";

Goal "(p:Prop)equiv(vel(p,p),p)";
INTRO"p";SOLVE"equiv_intro";INTRO"h1";SOLVE"or_intro1";SOLVE"h1";AUTO2();
INTRO"h1";SOLVE"case";SOLVE"h1";AUTO();SAVE1"idem_vel";

Goal "(p:Prop)(vel(p,p))p";


Goal "(p:Prop)(q:Prop)(vel(p,q))(not(p))q";
INT1["p","q","h1","h2"];TAC_cl "h3";RED"h1";SOLVE"h1";REDUCT();
INTRO"h4";SOLVE"h3";AUTO();SAVE1"vel_elim1";

Goal "(p:Prop)(q:Prop)(vel(p,q))(not(q))p";
INT1["p","q","h1","h2"];TAC_cl "h3";RED"h1";SOLVE"h1";SOLVE"h2";
REDUCT();INTRO"h4";SOLVE"h3";AUTO();SAVE1"vel_elim2";

Decl"A""Type";Decl"P""(A)Prop";
DEF "not((x:A)not(P(x)))" "E";
Goal "(x:A)(P(x))E(A,P)";
INT1["x","h1"];DUP REDUCT ();INTRO"h2";LET_INTRO"h2(x,h1)""l";AUTO();
SAVE1"witness";

Goal"(p:Prop)((x:A)(P(x))p)(E(A,P))p";
INT1["p","h1","h2"];TAC_cl"h3";RED"h2";RED"h2";SOLVE"h2";
INTRO"x";REDUCT();INTRO"h4";LET_INTRO"h3(h1(x,h4))""l";AUTO();SAVE1"elim_E";

fun ELIM_E h1 x h2 = (SOLVE "elim_E";SOLVE h1;INT1 [x,h2]);

Abs "A";
```

```
(* this file must contain the beginning of set theory, up to the
   definition of functions, and the definition of ordinals. Uses
   classique *)

PRIM "((V)Prop)V" "tau";

PRIM "(P:(V)Prop)(E(V,P))P(tau(P))" "choix";

PRIM "(P:(V)Prop)(Q:(V)Prop)((x:V)equiv(P(x),Q(x)))=(tau(P),tau(Q))" "S7";

PRIM "(V)(V)Prop" "mem";
```

```
****************************************************************************************************
**                                                                                                **
**                        tfhc - steve - Feb 21 10:34 - inclusion1.ML                             **
**                                                                                                **
****************************************************************************************************
****************************************************************************************************

(* the inclusion. Uses set. *)

DEF "[x:V][y:V](z:V)(mem(z,x))mem(z,y)" "inclus";

Goal "(x:V)inclus(x,x)";
INTRO"x";REDUCT();AUTO();SAVE1 "proposition1";

Goal "(x:V)(y:V)(z:V)(inclus(x,y))(inclus(y,z))inclus(x,z)";
INT1["x","y","z","h1","h2"];REDUCT();INT1["u","h3"];RED"h1";RED"h2";
AUTO();SAVE1 "proposition2";

PRIM "(x:V)(y:V)(inclus(x,y))(inclus(y,x))=(x,y)" "ext";

Goal "(x:V)inclus(x,x)";
INTRO"x";REDUCT();AUTO();SAVE1 "rem1";

Goal "(x:V)(y:V)(=(x,y))inclus(x,y)";
INT1["x","y","h1"];SIMPLIFY"h1"[];SOLVE"rem1";AUTO();SAVE1 "rem2";

Goal "(x:V)(y:V)(=(x,y))inclus(y,x)";
INT1["x","y","h1"];LET_INTRO"[u:V]inclus(u,x)""P0";
EGAL"P0(y)";SIMPLIFY"h1"[];REDUCT();SOLVE"rem1";AUTO();SAVE1 "rem3";

PRIM "(V)(V)V" "upair";
PRIM "(x:V)(y:V)(u:V)(mem(u,upair(x,y)))vel(=(u,x),=(u,y))" "ax1_upair";
PRIM "(x:V)(y:V)(u:V)(vel(=(u,x),=(u,y)))mem(u,upair(x,y))" "ax2_upair":

Goal "(x:V)(y:V)=(upair(x,y),upair(y,x))":
INT1["x","y"];SOLVE"ext";
REDUCT();INT1["u","h1"];MATCH_MP "ax1_upair" "h1" "l";
SOLVE"ax2_upair";SOLVE"sym_vel";SOLVE"l";AUTO2();
REDUCT();INT1["u","h1"];MATCH_MP "ax1_upair" "h1" "l";
SOLVE"ax2_upair";SOLVE"sym_vel";SOLVE"l";AUTO();SAVE1"comm_upair";

DEF "[x:V]upair(x,x)" "singl";

(* les regles qui suivent sont des regles de simplification *)

Goal "(x:V)(y:V)(mem(y,singl(x)))=(y,x)";
EXPAND "singl";INT1["x","y","h1"];MATCH_MP "ax1_upair" "h1" "l";
SOLVE"case";SOLVE"l";AUTO();SAVE1"remarque1";

Goal "(x:V)mem(x,singl(x))";
INTRO"x";EXPAND"singl";SOLVE"ax2_upair";SOLVE"or_intro1";SOLVE"ref_=";
AUTO();SAVE1"remarque2";

Goal "(x:V)(X:V)(inclus(singl(x),X))mem(x,X)";
INT1["x","X","h1"];SIMPLIFY"h1"[];SOLVE"remarque2";AUTO();
SAVE1"remarque3";

Goal "(x:V)(y:V)(=(singl(x),singl(y)))=(x,y)";
INT1 ["x","y","h1"];SOLVE"remarque1";
LET_INTRO "[u:V]mem(x,u)" "P0";EGAL "P0(singl(y))";SIMPLIFY"h1"[];
REDUCT();SOLVE"remarque2";AUTO();SAVE1 "remarque4";

Goal "(x:V)(y:V)mem(x,upair(x,y))";
INT1["x","y"];SOLVE"ax2_upair";SOLVE"or_intro1";SOLVE"ref_=";AUTO();
SAVE1"rem4";

Goal "(x:V)(y:V)mem(y,upair(x,y))";
INT1["x","y"];SOLVE"ax2_upair";SOLVE"or_intro2";SOLVE"ref_=";AUTO();
SAVE1"rem5";

Goal "(x:V)(y:V)(X:V)(inclus(upair(x,y),X))mem(x,X)";
INT1["x","y","X","h1"];SIMPLIFY"h1"[];SOLVE"rem4";AUTO();
SAVE1"remarque6";

Goal "(x:V)(y:V)(X:V)(inclus(upair(x,y),X))mem(y,X)";
INT1["x","y","X","h1"];SIMPLIFY"h1"[];SOLVE"rem5";AUTO();
SAVE1"remarque7";

Goal "(x:V)(y:V)(=(x,y))inclus(x,y)":
INT1["x","y","h1"];SIMPLIFY"h1"[]:SOLVE"proposition1";AUTO();
SAVE1"rem6";

Goal "(x:V)(y:V)(=(x,y))inclus(y,x)":
INT1["x","y","h1"];LET"[u:V]inclus(u,x)""P0":EGAL"P0(y)":
```

```
SIMPLIFY "h1" [];REDUCE();SOLVE"proposition1";AUTO();
SAVE1"rem7";

fun ELIM_egal h h1 h2 = (MATCH_MP "rem6" h h1;MATCH_MP "rem7" h h2;ENLEVE h);

Goal "(x:V)(y:V)(y1:V)(=(singl(x),upair(y,y1)))and(=(y1,x),=(y,x))";
INT1["x","y","y1","h1"];ELIM_egal "h1" "h11" "h12";
MATCH_MP "remarque6" "h12" "l";MATCH_MP "remarque7" "h12" "l1";
MATCH_MP "remarque1" "l" "r";MATCH_MP "remarque1" "l1" "r1";
SOLVE"intro_and";AUTO();SAVE1 "remarque8";
```

```
************************************************************************************************************************************
**                                                                                                                                **
**                                         tfhc - steve - Feb 22 10:35 - instant.ML                                               **
**                                                                                                                                **
************************************************************************************************************************************
************************************************************************************************************************************
```

```
(* definition of instants, following Russell and Wiener *)

Decl "A" "Type";Decl "P" "(A)(A)Prop";
Decl "dec" "(x:A)(y:A)or(P(x,y),not(P(x,y)))";

fun LET s1 s2 = (eval s2;Let (! C) s1) ;

LET "S" "[x:A][y:A]and(not(P(x,y)),not(P(y,x)))";

Decl "hyp1" "(x:A)(y:A)(z:A)(P(x,y))(S(y,z))P(x,z)";
Decl "hyp2" "(x:A)(y:A)(z:A)(S(x,y))(P(y,z))P(x,z)";

Goal "(x:A)(y:A)(S(x,y))S(y,x)";
INT1["x","y","h"];RED"h":SIMPLIFY"h"["h1","h2"];REDUCT();AUTO();
SAVE"sym_S";

fun ELIM_S h l = (RED h:SIMPLIFY h l) ;

fun ELIM_not h = (RED h:SIMPLIFY h []):

Goal"(x:A)(y:A)(z:A)(S(x.y))(S(y.z))S(x.z)";
INT1["x","y","z","h1"."h2"];REDUCT();PAS():LET"l""dec(z,x)";SIMPLIFY"l"[];
Step();Step();Step();
INTRO"h3";REDUCT();INTRO"h4";ELIM_S"h1"["h11","h12"];ELIM_not "h12";
SOLVE"hyp2";Step();Step();AUTO2();LET"l""dec(x.z)":SIMPLIFY"l"[];
Step();Step();Step();
INTRO"h3";REDUCT();INTRO"h4";ELIM_S"h1"["h11","h12"];ELIM_not "h11";
SOLVE"hyp1";SOLVE"sym_S";AUTO():
SAVE"trans_S":
```

tfhc - steve - Feb 22 10:02 - categorie1.ML

```
(* un exemple avec des categories *)
```

with polymorphism

```
Decl"Obj""Type";Decl"Hom""(A:Obj)(B:Obj)Type";
Decl"comp""(A:Obj)(B:Obj)(C:Obj)(f:Hom(A,B))(g:Hom(B,C))Hom(A,C)";
Decl"id""(A:Obj)Hom(A,A)";
Decl"Rel""(A:Obj)(B:Obj)(f:Hom(A,B))(g:Hom(A,B))Prop";

Decl"assoc""(A:Obj)(B:Obj)(C:Obj)(D:Obj)(f:Hom(A,B))(g:Hom(B,C))(h:Hom(C,D))\
\Rel(comp(comp(f,g),h),comp(f,comp(g,h)))";

Decl"cong""(A:Obj)(B:Obj)(C:Obj)\
          \(f:Hom(A,B))(g:Hom(B,C))(f1:Hom(A,B))(g1:Hom(B,C))\
          \(Rel(f,f1))(Rel(g,g1))\
          \Rel(comp(f,g),comp(f1,g1))";

Decl"ref""(A:Obj)(B:Obj)(f:Hom(A,B))Rel(f,f)";
Decl"trans""(A:Obj)(B:Obj)(f:Hom(A,B))(g:Hom(A,B))(h:Hom(A,B))\
       \(Rel(h,f))(Rel(g,f))Rel(g,h)";
Decl"id1""(A:Obj)(B:Obj)(f:Hom(A,B))Rel(comp(id(A),f),f)";
Decl"id2""(A:Obj)(B:Obj)(f:Hom(A,B))Rel(comp(f,id(B)),f)";

Decl"T""Obj";Decl"nil""(A:Obj)Hom(A,T)";
Decl"eg_nil""(A:Obj)(f:Hom(A,T))Rel(f,nil(A))";

Decl"f""Hom(T,T)";Goal"Rel(f,id(T))";

Abs"f";
Decl"B""Obj";Decl"f""Hom(B,T)";
Decl"mono""(A:Obj)(g:Hom(A,B))(h:Hom(A,B))\
    \(Rel(comp(g,f),comp(h,f))Rel(g,h)";
Decl"g""Hom(T,B)";Goal"Rel(comp(g,f),id(T))";

Goal"Rel(comp(f,g),id(B))";
```

```
********************************************************************************
**                                                                            **
**                 tfhc - steve - Feb 22 10:02 - monoide.ML                   **
**                                                                            **
********************************************************************************
********************************************************************************
```

```
(* This example shows
    .how to represent a first-order signature
    .how to deal with equality *)

Decl "G" "Type";
Decl "Eq" "(G)(G)Prop";
Decl "f"  "(G)(G)G";     (* composition *)
Decl "e"  "G";           (* element neutre *)

Decl "trans" "(x:G)(y:G)(z:G)(Eq(y,z))(Eq(x,z))Eq(x,y)";
Decl "ref"   "(x:G)Eq(x,x)";

Decl "assoc" "(x:G)(y:G)(z:G)Eq(f(x,f(y,z)),f(f(x,y),z))";
Decl "id1"   "(x:G)Eq(f(e,x),x)";
Decl "id2"   "(x:G)Eq(f(x,e),x)";

Decl "cong"  "(x:G)(y:G)(z:G)(t:G)(Eq(y,t))(Eq(x,z))Eq(f(x,y),f(z,t))";

Decl"x""G";Decl"y""G";Decl"z""G";

Goal "Eq(f(x,f(f(e,y),z)),f(f(x,y),f(z,e)))";
SOLVE "trans";SOLVE "assoc";SOLVE "trans";SOLVE "assoc";
SOLVE "trans";SOLVE "cong";SOLVE "assoc";SOLVE "ref";SOLVE "trans";
SOLVE "id2";SOLVE "trans";SOLVE "cong";SOLVE "cong";SOLVE "id2";
SOLVE "ref";SOLVE "ref";SOLVE "ref";AUTO();

(*
j'ai trouve
la reponse est :
((z)((y)(e)f)(x)assoc)(((e)(z)((y)(x)f)assoc)((((y)(e)(x)assoc)((z)ref)(z)((y)((e)(x)f)f)(z)(((y)(e)f)(x)f)cong
)((((z)((y)(x)f)f)id2)(((((x)id2)((y)ref)(y)(x)(y)((e)(x)f)cong)((z)ref)(z)((y)(x)f)(z)((y)((e)(x)f)f)cong)(((z
)((y)(x)f)f)ref)((z)((y)(x)f)f)((z)((y)(x)f)f)((z)((y)((e)(x)f)f)f)trans)((z)((y)(x)f)f)((z)((y)((e)(x)f)f)f)((
e)((z)((y)(x)f)f)f)trans)((z)((y)((e)(x)f)f)f)((e)((z)((y)(x)f)f)f)((z)(((y)(e)f)(x)f)f)trans)((e)((z)((y)(x)f)
f)f)((z)(((y)(e)f)(x)f)f)(((e)(z)f)((y)(x)f)f)trans)((z)(((y)(e)f)(x)f)f)(((e)(z)f)((y)(x)f)f)(((z)((y)(e)f)f)(
x)f)trans
*)

val l1 = ["ref","assoc","id1","id2","trans"];
val l2 = ["assoc","id1","id2"];

fun tac1 () = RESOLVE l1;fun tac2 () = RESOLVE l2;
fun tac3 () = SOLVE "cong";fun tac4 () = SOLVE "ref";

fun tac5 () = (tac2 ORELSE (tac3 AND (DUP tac5)) ORELSE tac4) ();

val tac = tac1 AND tac5;

fun TAC () = LOOP tac () handle fail with
 "RESOLVE"                 =>
(prs "j'ai trouve\nla reponse est : \n";imp (!COM))
|s                         => raise fail with s;

Goal "Eq(f(x,f(f(e,y),z)),f(f(x,y),f(z,e)))";
TAC();

(*
((z)((y)(e)f)(x)assoc)(((e)(z)((y)(x)f)assoc)((((y)(e)(x)assoc)((z)ref)(z)((y)((e)(x)f)f)(z)(((y)(e)f)(x)f)cong
)((((z)((y)(x)f)f)id2)(((((x)id2)((y)ref)(y)(x)(y)((e)(x)f)cong)((z)ref)(z)((y)(x)f)(z)((y)((e)(x)f)f)cong)(((z
)((y)(x)f)f)ref)((z)((y)(x)f)f)((z)((y)(x)f)f)((z)((y)((e)(x)f)f)f)trans)((z)((y)(x)f)f)((z)((y)((e)(x)f)f)f)((
e)((z)((y)(x)f)f)f)trans)((z)((y)((e)(x)f)f)f)((e)((z)((y)(x)f)f)f)((z)(((y)(e)f)(x)f)f)trans)((e)((z)((y)(x)f)
f)f)((z)(((y)(e)f)(x)f)f)(((e)(z)f)((y)(x)f)f)trans)((z)(((y)(e)f)(x)f)f)(((e)(z)f)((y)(x)f)f)(((z)((y)(e)f)f)(
x)f)trans

Timing  compile-0.1s (0.0s+0.0s+0.1s+0.0s+0.0s) run-1.3s
*)
```

. One example developped: beginning of Scott's theory of domain in higher-order intuitionistic logic (~ topos)

. topos theory }
type theory } as a fundation of mathematics?

For *mathematical* applications, set theory as in Bourbaki {
- classical logic
. τ operator

example in Isabelle (Ph. de Groote)

Do we need . higher-order unification?
. unification of λ-terms *not* modulo β-conversio

example HOL

Questions and challenges

. set theory versus direct formalisation ?

. is it possible / interesting to see the type system as a type system for the meta-language ?

. do we need dependant products (Isabelle) ?

. use of proofs ? "no-counter example" method for unwinding a proof.

. proof of normalisation on a computer

Terminating General Recursion

Bengt Nordström
Department of Computer Science
University of Göteborg/Chalmers
S-412 96 Göteborg
Sweden

First Draft, February 1987

Abstract

In Martin-Löf's type theory, general recursion is not available. The only iterating constructs are primitive recursion over natural numbers and similarly defined inductive types. The paper describes a way to allow a general recursion operator in type theory (extended with propositions). A proof rule for the new operator is presented. The addition of the new operator will not distroy the property that all well-typed programs terminate. An advantage of the new program construct is that it is possible to separate the termination proof of the program from the proof of other properties.

10

1 Introduction

Martin-Lõf's type theory [1,?,?] is a programming logic for a functional programming language. There is a formal system in which it is possible to express not only programs and their specifications but also derivations of programs. Versions of type theory have been implemented in Gõteborg [4], Cornell [5] and Cambridge [6]. Using these systems it is possible to use the computer to check the correctness of program derivations.

Type theory relies heavily on the identification between types, propositions and specifications.

The judgement

$$a \in A$$

can be read as:

1. a is a construction for the proposition A
2. a is a solution of the problem A
3. a is a program which satisfies the specification A
4. a is an implementation of the abstract data type specification A [8].

It is a theory for total correctness, that $a \in A$ means that the program a terminates with a value in A. Compared with other functional languages it has a very rich type structure in that the type system can be used to completely express the task of the program. For instance, the type of a program solving Fermat's last theorem is

$$(\Pi x \in \mathsf{N}).\{n \in \mathsf{N} | n > 2 \& \exists x, y, \overset{z}{\cancel{z}} \in \mathsf{N}. x^n + y^n = z^n\}$$

Wether there is an element in this type only God (and Fermat) knows.

If the type structure is very strong, the program forming operations may seem a little weak, since general recursion is not available, only primitive recursion. From a metamathematical point of view, this is not a serious problem. We know that primitive recursion together with higher order functions give us a way to express all provably (in Peano's arithmetic) total functions. This, however, gives no relief to a programmer who really wants to write down a program! From a programming methodological point of view, it forces the programmer to prove the termination of the program at the same time as the program is derived. It is often convenient to be able to separate the termination proof from the rest of the correctness proof. Another serious problem is that it forces the programmer to in some sense estimate the number of iterations the program will make. This information is not always available, an example of this is the lambo-function. (This introduction will be expanded later).

The purpose of this paper is to show that it is possible to extend type theory with an operator for general recursion, while still not giving up the requirement on termination.

2 The Syntax of Type Theory

Expressions in type theory are built up from constants and variables using application and abstraction. An expression which cannot be applied to an argument is called *saturated*. For instance, in the Natural Induction rule below, p, N, d, $C(0)$, $e(x, y)$, $C(\mathsf{succ}(x))$, x, y, $\mathsf{natrec}(p, d, e)$, $C(p)$ are saturated. The expression C is unsaturated, it expects one (saturated) argument, the expression e expects two saturated arguments and, finally, the primitive constant natrec expects two saturated arguments and one unsaturated argument which expects two saturated arguments.

If x is a variable and e is an expression then

$$x.e$$

will denote the abstraction of e with respect to x.

3 Primitive Recursion and Matematical Induction

Consider the rule for natural induction in type theory:

Natural Induction:

$$\frac{p \in \mathsf{N} \qquad d \in C(0) \qquad e(x,y) \in C(\mathsf{succ}(x))\ [x \in \mathsf{N},\ y \in C(x)]}{\mathsf{natrec}(p,d,e) \in C(p)}$$

The rule can be read in the following way: We may draw the conclusion $\mathsf{natrec}(p, d, e) \in C(p)$ if $p \in \mathsf{N}$, $d \in C(0)$ and if $e(x, y) \in C(\mathsf{succ}(x))$ under the assumptions that $x \in \mathsf{N}$ and $y \in C(x)$.

Using currying, we can slightly rewrite the rule as:

Natural Induction':

$$\frac{p \in \mathsf{N} \qquad d \in C(0) \qquad e(x) \in C(x) \to C(\mathsf{succ}(x))\ [x \in \mathsf{N}]}{\mathsf{natrec}(p,d,e) \in C(p)}$$

So the problem $C(p)$ is solved by the program $\mathsf{natrec}(p, d, e)$ if d solves the problem $C(0)$ and $e(x)$ is a "step-function" taking a solution of the problem $C(x)$ to a solution of the problem $C(\mathsf{succ}(x))$. The justification of the rule is based on the semantics of type theory and the computation rule for the primitive recursion operator natrec:

If the value of p is 0 then the value of $\mathsf{natrec}(p, d, e)$ is the value of d which solves the problem $C(0)$.

If the value of p is $\mathsf{succ}(0)$ then the value of $\mathsf{natrec}(p, d, e)$ is the value of $e(0)(d)$ which solves the problem $C(\mathsf{succ}(0))$ since

$$e(0) \in C(0) \to C(\mathsf{succ}(0))$$

and

$$d \in C(0).$$

If the value of p is $\mathsf{succ}(\mathsf{succ}(0))$ then the value of $\mathsf{natrec}(p, d, e)$ is the value of $e(\mathsf{succ}(0))(e(0)(d))$ which solves the problem $C(\mathsf{succ}(0))$ since

$$e(\mathsf{succ}(0)) \in C(\mathsf{succ}(0)) \to C(\mathsf{succ}(\mathsf{succ}(0)))$$

and

$$e(0)(d) \in C(\mathsf{succ}(0)).$$

In general, if the value of p is $\mathsf{succ}(a)$, then the value of $\mathsf{natrec}(p, d, e)$ is the value of $e(a)(\mathsf{natrec}(a, d, e))$ which solves the problem $C(\mathsf{succ}(a))$ since

$$e(a) \in C(a) \to C(\mathsf{succ}(a))$$

and

$$\mathsf{natrec}(a, d, e) \in C(a).$$

4 Course - of - values recursion and Complete Induction

In course – of – values recursion we have a step-function $e(x)$ which takes a solution of all problems $C(0), C(\mathsf{succ}(0)), \ldots, C(x)$ to a solution of the problem $C(\mathsf{succ}(x))$. How can we express this? We want to have a function $e(x)$ which as argument takes a list or a tuple $\langle g_0, g_1, \ldots, g_x \rangle$, where $g_i \in C(i)$. A convenient way to express this is that the argument of $e(x)$ is a function g such that $g(i) \in C(i)$ for $i \leq x$. This is an element in a Cartesian product of a family of types, i.e.

$$e(x) \in (\prod_{i \leq x} C(i)) \to C(\mathsf{succ}(x))$$

where $\prod_{i \leq x} C(i)$ is the type of functions b such that $b(i) \in C(i)$ for $i \leq x$.

We can also express the requirement on the step-function $e(x)$ as:

$$e(x, y) \in C(\mathsf{succ}(x)) \; [x \in \mathsf{N},\; y(z) \in C(z) \; [z \leq x]]$$

So we obtain the following rule:

Course-of-values Induction 1:

$$\frac{p \in \mathsf{N} \qquad d \in C(0) \qquad e(x, y) \in C(\mathsf{succ}(x)) \; [x \in N,\; y(z) \in C(z) \; [z \leq x]]}{\mathsf{covrec}(p, d, e) \in C(p)}$$

where covrec is a new primitive constant which is computed in the following way:

The value of $\mathsf{covrec}(p,d,e)$ is obtained by first computing the value of p. If the result is 0 then the value of $\mathsf{covrec}(p,d,e)$ is the value of d. If the result is $\mathsf{succ}(a)$, then the value of $\mathsf{covrec}(p,d,e)$ is the value of $e(a,z.\mathsf{covrec}(z,d,e))$, where $z.e$ is the notation for the abstraction of e with respect to the variable z.

So if the value of p is 0 then the value of $\mathsf{covrec}(p,d,e)$ is the the value of d which solves the problem $C(0)$. If the value of p is $\mathsf{succ}(0)$, then the value of $\mathsf{covrec}(p,d,e)$ is the value of $e(0,z.\mathsf{covrec}(z,d,e))$. But $e(0,y) \in C(\mathsf{succ}(0))$ if y is a function such that $y(z) \in C(z)$, for $z \leq 0$. But $z.\mathsf{covrec}(z,d,e)$ is such a function since $\mathsf{covrec}(0,d,e) \in C(0)$.

If the value of p is $\mathsf{succ}(\mathsf{succ}(0)$ then the value of $\mathsf{covrec}(p,d,e)$ is the value of $e(\mathsf{succ}(0),z.\mathsf{covrec}(z,d,e))$ but $e(\mathsf{succ}(0),y) \in C(\mathsf{succ}(\mathsf{succ}(0)))$ if y is a function such that $y(z) \in C(z)$ for $z \leq \mathsf{succ}(0)$. And $z.\mathsf{covrec}(z,d,e)$ is such a function since $\mathsf{covrec}(0,d,e) \in C(0)$ and $\mathsf{covrec}(\mathsf{succ}(0),d,e) \in C(\mathsf{succ}(0))$.

We can simplify the rule for course-of-values induction if we instead have a step function $e(x)$ which takes a solution of all problems strictly smaller than x to a solution of $C(x)$. We can then drop the second premise:

Course-of-values Induction 2:

$$\frac{p \in \mathsf{N} \qquad e(x,y) \in C(x)\ [x \in \mathsf{N},\ y(z) \in C(z)\ [z < x]]}{\mathsf{rec}'(p,e) \in C(p)}$$

where the value of $\mathsf{rec}'(p,e)$ is computed by computing the value of $e(p,z.\mathsf{rec}'(z,e))$.

We get the rule for complete induction if we take away some of the constructions in the previous rule:

Complete Induction:

$$\frac{p \in \mathsf{N} \qquad C(x)\mathsf{true}\ [x \in \mathsf{N},\ C(z)\mathsf{true}\ [z < x\mathsf{true}]]}{C(p)\mathsf{true}}$$

5 General Recursion and Well-founded Induction

There is nothing in the rule for course-of-values induction which is particular to the set of natural numbers. The reason the rule works is that N is well-ordered by $<$. We can generalize the rule to an arbitrary set A which is well-ordered by a relation $<_A$. The computation rule becomes a little simpler if we change the order of the arguments:

Recursion rule:

$$\frac{\mathsf{Well\text{-}ordered}(A,<_A) \qquad p\in A \qquad e(x,y)\in C(x)\ [x\in A,\ y(z)\in C(z)\ [z<x]]}{\mathsf{rec}(e,p)\in C(p)}$$

with the following computation rule for rec:

The value of $\mathsf{rec}(e,p)$ is the value of $e(p,\mathsf{rec}(e))$.

Notice that we get the ordinary fixpoint operator by defining

$$\mathsf{Y}(e,p)\equiv \mathsf{rec}(x.y.e(y,x),p)$$

because then $\mathsf{Y}(e)$ is a fixpoint of e, i.e. $e(\mathsf{Y}(e))=\mathsf{Y}(e)$, i.e. $e(\mathsf{Y}(e))(p)=\mathsf{Y}(e)(p)$ since

$$\begin{aligned}
\mathsf{Y}(e)(p) &\equiv \mathsf{Y}(e,p)\\
&\equiv \mathsf{rec}(x.y.e(y,x),p)\\
&\rightarrow_1 (x.y.e(y,x))(p,\mathsf{rec}(x.y.e(y,x)))\\
&\equiv e(y,x)\ [x:=p, y:=\mathsf{rec}(x.y.e(y,x))]\\
&\equiv e(\mathsf{rec}(x.y.e(y,x)),p)\\
&\equiv e(\mathsf{Y}(e),p)
\end{aligned}$$

where

$\rightarrow_1$ stands for "computes in one step to"

$e[x:=a,y:=b]$ stands for the expression obtained from e by simultaneous substitiution of the variable x with a and the variable y with b.

So we get the fixpoint operator by swapping the arguments of the first argument to rec. If we try to formulate the previous rule using the Y-combinator directly we get the following

Wrong recursion rule :

$$\frac{\mathsf{Well\text{-}ordered}(A,<_A) \qquad p\in A \qquad e(y,x)\in C(x)\ [y(z)\in C(z)\ [z<x],\ x\in A]}{\mathsf{Y}(e,p)\in C(p)}$$

which doesn't make sense since the first argument of e depends on the second.

6 A simple example: The termination of quicksort

The scheme which will be used for solving recursion equations is that if

$$f(z)=e(z,f)$$

is a recursion equation then it can be solved by defining

$$f \equiv \mathsf{rec}(e)$$

where we in e have abstracted the two variables z and f. This is correct provided that the requirements on the parameters hold as expressed by the recursion rule. In the simple case that the family $C(x)$ type $[x \in A]$ does not depend on x, the requirements are that in the equation $f(z) = e(z, f)$, f is a function from A to C, z is an element in A and f must only be applied to arguments smaller than z on the right hand side of the equation. These requirements are exactly those which are used by programmers of functional languages in informal termination proofs.

Let's look at a termination proof of quicksort. The recursion equations for quicksort are:

$$\begin{aligned} \mathsf{quick}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{quick}(\mathsf{cons}(a, s)) &= \mathsf{quick}(\mathsf{less}(s, a)) \diamondsuit \mathsf{cons}(a, \mathsf{quick}(\mathsf{gt}(s, a))) \end{aligned}$$

where nil is the empty list and cons(a, s) is the list with a as the first element and the list s as the rest, less(s, a) is the list obtained from s by taking away all elements which are greater than a and gt(s, a) is the list where all elements of s strictly smaller than a have been removed. To solve the equations we have to first rewrite them using the listcases- expression: [1]

```
quick(p)= listcases(p,
                    nil,
                    a.s.( quick(less(s, a)) ◇ cons(a, quick(gt(s, a)))))
```

This equation can be solved by making the definition:

```
quick≡ rec x . y . listcases(x,
                            nil,
                            a.s.(y(less(s, a)) ◇ cons(a, y(gt(s, a)))) )
```

In order to show that quick terminates it is enough to show that [2]

$$\mathsf{quick} \in \mathsf{List}(A) \rightarrow \mathsf{List}(A)$$

In order to show this we have to show that

[1] The primitive constant listcases is used to express pattern-matching over lists. The value of listcases(p, d, e) is computed by first computing the value of p. If the value of p is nil, then the value of listcases(p, d, e) is the value of d. If the value of p is cons(a, s) then the value of listcases(p, d, e) is the value of $e(a, s)$.

[2] The meaning of $a \in A$ is that the computation of a *terminates* with a value in A.

$$\begin{aligned}&\mathsf{listcases}(\mathsf{x},\\ &\qquad \mathsf{nil},\\ &\qquad \mathsf{a.s.(y(less(s,\ a))} \diamond \mathsf{cons(a,\ y(gt(s,\ a))))} \in \mathsf{List(A)}\end{aligned}$$

under the assumptions that $x \in \mathsf{List}(A)$, $y(z) \in \mathsf{List}(A)$ $[z \in \mathsf{List}(A),\ z < x]$ for some well founded relation $<$.

This can be shown by induction over x. The base case is trivial. If $x = \mathsf{cons}(a, s)$ we have to show that

$$y(\mathsf{less}(s,a)) \diamond \mathsf{cons}(a, y(\mathsf{gt}(s,a)))) \in \mathsf{List}(A)$$

which holds if

$$\begin{cases} y(\mathsf{less}(s,a)) \in \mathsf{List}(A) \\ y(\mathsf{gt}(s,a)) \in \mathsf{List}(A) \end{cases}$$

since the concatenation operator $\diamond$ takes two lists to a list. But this follows from the induction assumption if we can find a wellordering $<$ on $\mathsf{List}(A)$ such that

$$\mathsf{less}(s,a) < \mathsf{cons}(a,s)$$

and

$$\mathsf{gt}(s,a) < \mathsf{cons}(a,s)$$

but this holds if we define $<$ to be

$$a < b \equiv \#a <_{\mathsf{N}} \#b$$

where $\#a$ is the number of elements in the list a and $<_{\mathsf{N}}$ is the usual order on natural numbers.

Theorem 1 *All iterating constructs in type theory can be reduced to pattern matching and the general recursion operator* rec.

Proof outline: For instance, if we define [3]

$$\mathsf{natrec}(p,d,e) \equiv \mathsf{rec}(x.y.\mathsf{natcases}(x,d,z.e(z,y(z))),p)$$

The value of $\mathsf{natrec}(p,d,e)$ is then the value of

$$\begin{aligned} & \mathsf{natcases}(x,d,z.e(z,y(z)))\ [x := p, y := \mathsf{rec}(x'.y'.\mathsf{natcases}(x',d,z.e(z,y'(z)))] \\ \equiv\ & \mathsf{natcases}(p,d,z.e(z,\mathsf{rec}(x'.y'.\mathsf{natcases}(x',d,z.e(z,y'(z))),z))) \\ \equiv\ & \mathsf{natcases}(p,d,z.e(z,\mathsf{natrec}(z,d,e))) \qquad (1) \end{aligned}$$

[3] The primitive constant natcases is used to express pattern-matching over natural numbers. The value of natcases(p,d,e) is computed by first computing the value of p. If the value of p is 0, then the value of natcases(p,d,e) is the value of d. If the value of p is succ(a) then the value of natcases(p,d,e) is the value of $e(a)$. A more readable syntax for the natcases-expression would be

$$\mathrm{cases} p \mathrm{of} 0 : d, \mathsf{succ}(x) : e' \mathrm{endcases}$$

where all free occurrences of the variable x becomes bound in the expression e'.

If the value of p is 0 then the value of 1 is the value of d. If the value of p is $\mathsf{succ}(a)$, then the value of 1 is the value of $e(a, \mathsf{natrec}(a, de))$. So we have shown that natrec as defined above is computed in the same way as the traditional primitive recursion operator in type theory.

The listrec-operator and the w-rec-operator can be defined similarly. □

7 Remark

The general framework within which this is expressed is the logical theory presented by Martin-Lõf in Siena 1983. This is the first draft of the paper: more examples have to be added (for instance the lambo-function and binary search) and the propositional function which expresses wellordering has to be made precise.

References

[1] P. Martin-Lõf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, North-Holland, 1975.*

[2] P. Martin-Lõf. Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science, pp. 153-175. North-Holland, 1982.*

[3] P. Martin-Lõf. Intuitionistic Type Theory. *Studies in Proof Theory, Lecture Notes, Bibliopolis, Napoli, 1984.*

[4] K. Petersson. A Programming System for Type Theory. Programming Methodology Group report 9, 1982, University of Gõteborg and Chalmers Universtiy of Technology, Gõteborg, Sweden.

[5] R. Constable. Implementing Mathematics with the Nuprl Proof Development System, Englewood Cliffs: Prentice Hall.

[6] L. Paulson. Natural Deduction Proof as Higher-Order Resulution. University of Cambridge Computer Laboratory. Technical Report No. 82

[7] L. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. J. Symbolic Computation(1986) 2, 325-355.

[8] B. Nordstrõm, K. Petersson.The Semantics of Module Specifications in Martin-Lõf's Type Theory. Programming Methodology Group report. 1984.

FURIO HONSELL

The Metatheory of the LF Type System

11

The LF Type System

is a formal system for deriving assertions of the shape

$$\Gamma \vdash_{\Sigma} K$$

$$\Gamma \vdash_{\Sigma} A : K$$

$$\Gamma \vdash_{\Sigma} M : A$$

where $K ::= \mathrm{Type} \mid \Pi x{:}A.\,K$

$A ::= c \mid \Pi x{:}A.\,B \mid \lambda x{:}A.\,B \mid AM$

$M ::= x \mid c \mid \lambda x{:}A.\,M \mid MN$

$\Sigma ::= S_1, \ldots, S_n$

$S ::= c : K \mid c : A$

$\Gamma ::= G_1, \ldots, G_n$

$G ::= x : A$

with standard conventions

Rules

1. $$\vdash \text{Type}$$

2. $$\frac{\vdash_{\Sigma} K}{\vdash_{\Sigma, c:K} \text{Type}} \qquad c \text{ fresh}$$

3. $$\frac{\Gamma \vdash_{\Sigma} A : \text{Type}}{\Gamma, x:A \vdash_{\Sigma} \text{Typee}} \qquad \frac{\vdash_{\Sigma} A : \text{Type}}{\vdash_{\Sigma, c:A} \text{Type}}$$

x, c fresh

4. $$\frac{\Gamma \vdash_{\Sigma} \text{Type}}{\Gamma \vdash_{\Sigma} U : V} \qquad U:V \in \Sigma \cup \Gamma$$

5. $$\frac{\Gamma, x:A \vdash_{\Sigma} K}{\Gamma \vdash_{\Sigma} \Pi x:A.K}$$

6. $$\frac{\Gamma, x:A \vdash_{\Sigma} B : K}{\Gamma \vdash_{\Sigma} \lambda x:A.B : \Pi x:A.K}$$

7. $$\frac{\Gamma \vdash_{\Sigma} A : \Pi x:B.K \qquad \Gamma \vdash_{\Sigma} M : B}{\Gamma \vdash_{\Sigma} AM : K[{}^{x}/_{M}]}$$

8. $$\frac{\Gamma, x:A \vdash_2 B : Type}{\Gamma \vdash_2 \Pi x:A.B : Type}$$

9. $$\frac{\Gamma, x:A \vdash_2 M : B}{\Gamma \vdash_2 \lambda x:A.M : \Pi x:A.B}$$

10. $$\frac{\Gamma \vdash_2 M : \Pi x:B.A \qquad \Gamma \vdash_2 N : B}{\Gamma \vdash_2 MN : A[x/N]}$$

13. $$\frac{\Gamma \vdash_2 A : K \qquad \Gamma \vdash_2 K' \qquad K \approx K'}{\Gamma \vdash_2 A : K'}$$

14. $$\frac{\Gamma \vdash_2 M : A \qquad \Gamma \vdash_2 A' : Type \qquad A \approx A'}{\Gamma \vdash_2 M : A'}$$

Problem : how do we define "$\approx$"

1. congruent closure of $\beta\eta$- contraction ($\rightarrow_{\beta\eta}$)

2. $A \approx A'$ iff $\exists A''$ $A \xrightarrow{*} A'' \leftarrow A'$
for K similarly

3. $A \approx A'$ iff A conv$_{\beta\eta}$ A'
for K similarly

The 3 choices are equivalent if.
A and K are <u>Church Rosser</u> w.r.t.
$\beta\eta$ - reduction

i.e.

$$\begin{tikzcd}
& A'' \arrow[dr] & & \cdots \arrow[dr, "*"] & \\
A \arrow[ur, "*"] \arrow[drr, "*"] & & A''' \arrow[ur, "*"] & & A' \arrow[dll, "*"] \\
& & A^{\circ} & &
\end{tikzcd}$$

Unfortunately this is not the case

$$\begin{tikzcd}
& \lambda x{:}A.\ (\lambda x{:}B.\ C)\, x \arrow[dl, "\beta"] \arrow[dr, "\eta"] & \\
\lambda x{:}A.\ C & & \lambda x{:}B.\ C
\end{tikzcd} \qquad x \notin FV\ B, C$$

$1 \iff 2$

$3 \overset{?}{\iff} 1$ is an open problem

What is the logic of analytic judgements we have formalized?

Which are the properties of the consequence relation

list of judgements $\vdash_{\Sigma}$ judgement ?

$$\Gamma \vdash_{\Sigma} M : A \quad ?$$

these properties will influence the <u>naturalness</u> of an LF presentation of a given consequence relation and of the rules under which it is closed

Rule 4

$$\Gamma \vdash_{\Sigma} x : A \qquad \text{if} \qquad x : A \in \Gamma$$

$$\Gamma \vdash_{\Sigma} c : A \qquad \text{if} \qquad c : A \in \Sigma$$

reflexivity

This is an admissible rule

$$\frac{\Gamma \vdash_{\Sigma} M : A \quad , \quad \Gamma, \Gamma' \vdash_{\Sigma, \Sigma'} \mathrm{Type}}{\Gamma, \Gamma' \vdash_{\Sigma, \Sigma'} M : A}$$

thinning (weakening)

judgements are open concepts !

given a logic $\mathcal{L}$
no proof of the derivability of a rule in $\mathcal{L}$ done by induction on the structure of formulae or proofs is directly derivable in any adequate signature for $\mathcal{L}$

Eg. a) substitutivity of equals in F.O.L.

compare this to the derivability of S.H. elimination rule for $\wedge$ in the classical signature of FOL

b) $\nvdash_{\text{Hilbert-FOL}} \quad M : \Pi A, B : o \left(\mathrm{True}(A) \to \mathrm{True}(B) \right) \to \mathrm{True}(A \supset B)$

This is another admissible rule of the system

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A, \Delta \vdash U : V}{\Gamma, \Delta[x/M] \vdash U[x/M] : V[x/M]}$$

transitivity (cut)
+
contraction

- difficulty in modeling naively
  rules of proof
  relevance logics

- extremely natural to model in the LF
  the hypothetical and general
  judgements of standard logics
  in Hilbert, N.D.

- what about natural versions of
  multi valued sequent calculi ?

Why analytic judgements?

$\boxed{\text{The LF Type System is decidable}}$

- proofs and basic tactics (refinements) can be checked automatically

The following rule is admissible

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma' \vdash_{\Sigma'} \mathrm{Type}}{\Gamma' \vdash_{\Sigma'} M : A}$$

provided $FV\{M\} \cup FV\{A\} \subseteq \Gamma'$

$\mathrm{Constants}\{M\} \cup \mathrm{Constants}\{A\} \subseteq \Sigma'$

Γ', Σ' are subsequences of Γ and Σ, resp.

strengthening (AUTOMATH)

Proof.
extremely complicated
decidability has to be proved first ☒

It is best to assume this rule from the very beginning, and proove it admissible for the original system only after basic metatheoretic results for the extended system, such as decidability, have been derived

Free variables or constants in the proof term are essential :

- empty domains and free logic

$$\Gamma,\ P : \iota \to o\ ,\ x : \mathrm{True}\,(\forall P) \vdash_{\Sigma_{FOL}} M : \mathrm{True}\,(\exists P)$$

is derivable only if
either $x : \iota \in \Gamma$ or $c : \iota \in \Sigma_{FOL}$

$$M \equiv \exists_I\, P c\, (\forall_E\, P c\, x)$$

Moral : a proof of the adequacy of a given signature for a given logic is needed

main tools in such proofs are

- Strong Normalization for correct expressions (η reduction)
- Church Rosser for correct expressions
- a clear understanding of the consequence relations that are modeled by the various synthetic judgements

Eg. F.O.L.

$$\Gamma, x_1 : True(A_1), \ldots, x_n : True(A_n) \vdash_{\Sigma_{FOL}} M : True(A$$

iff

$$A_1, \ldots, A_n \underset{\uparrow}{\vdash} A$$

consequence relation of truth

where $x : A \in \Gamma \Rightarrow A \equiv \iota$ or o

Sketch of the decision Algorithm

$a(\vdash_{\Sigma, c:K} \mathrm{Type})$ check $a(\vdash_{\Sigma} K)$

$a(\vdash_{\Sigma, c:A} \mathrm{Type})$ check $a(\vdash_{\Sigma} A : \mathrm{Type})$

$a(\Gamma, x : A \vdash_{\Sigma} \mathrm{Type})$ check $a(\Gamma \vdash_{\Sigma} A : \mathrm{Type})$

$a(\Gamma \vdash_{\Sigma} A : K)$ check $a(\Gamma \vdash_{\Sigma} K)$
evaluate $\tau^{\Gamma}_{\Sigma}(A)$
check $\tau^{\Gamma}_{\Sigma}(A) \ \mathrm{conv}_{\beta\eta} \ K$

$a(\Gamma \vdash_{\Sigma} M : A)$ check $a(\Gamma \vdash_{\Sigma} A : \mathrm{Type})$
evaluate $\tau^{\Gamma}_{\Sigma}(M)$
check $\tau^{\Gamma}_{\Sigma}(M) \ \mathrm{conv}_{\beta\eta} \ A$

Definition of $\tau_{\Sigma}^{\Gamma}(A)$ and $\tau_{\Sigma}^{\Gamma}(M)$

```
$\tau_{\Sigma}^{\Gamma}(\Pi x{:}A.\,B)$        check $\alpha(\Gamma \vdash_{\Sigma} A : Type)$
                             check $\alpha(\Gamma, x{:}A \vdash_{\Sigma} B : Type)$
                             return $Type$

$\tau_{\Sigma}^{\Gamma}(c)$                 return $K$  iff  $c : K \in \Sigma$

$\tau_{\Sigma}^{\Gamma}(\lambda x{:}A.\,B)$         check $\alpha(\Gamma \vdash_{\Sigma} A : Type)$
                    return $\Pi x{:}A.\ \tau_{\Sigma}^{\Gamma, x:A}(B)$

$\tau_{\Sigma}^{\Gamma}(A\,M)$              evaluate $\tau_{\Sigma}^{\Gamma}(A)$
                   if $\tau_{\Sigma}^{\Gamma}(A) = \Pi x{:}B.\,C$
                      evaluate $\tau_{\Sigma}^{\Gamma}(M)$
                         if $\tau_{\Sigma}^{\Gamma}(M)$ conv$_{\beta\eta}$ $B$
                            return $C[M]$

$\tau_{\Sigma}^{\Gamma}(x)$                 return $A$  iff  $x : A \in \Gamma$
```

. . . . . .

Remarks

type labels in abstractions is essential $\lambda x{:}\boxed{A}\,M$

In order to decide tests of the type

$$\tau_{\Sigma}^{\Gamma}(A) \text{ conv}_{\beta\eta}\ K$$

we need

Strong Normalization of A and K
and uniqueness of normal forms

For the algorithm to be proved sound and complete we need

1) Subject Reduction Theorem

if $\Gamma \vdash_{\Sigma} U : V$ and $U \xrightarrow{*}_{\beta\eta} U'$

then $\Gamma \vdash_{\Sigma} U' : V$

the strengthening rule is essential to prove this

2) SN
3) CR

The issue of transforming proofs by suffering into insightful proofs.

Eg. Strong Normalization can be proved at a very early stage although type labels are present. as a corollary it yields that

if $\Gamma \vdash_{\Sigma} M : A$

then Untype(M) can be typed in ML where Untype(M) denotes the type erasure of M

Proof (G. Plotkin)

define $\tau : K \cup A \longrightarrow$ Simple Types

$$\tau(\mathrm{Type}) = \alpha$$

$$\tau(\Pi x{:}A.K) = \tau(A) \rightarrow \tau(K)$$

$$\tau(c) = \alpha$$

$$\tau(\lambda x{:}A.B) = \tau(B)$$

$$\tau(A\,M) = \tau(A)$$

$$\tau(\Pi x{:}A.B) = \tau(A) \rightarrow \tau(B)$$

Fact

if. $A \ \mathrm{conv}_{\beta\eta} \ A'$

$$\text{then} \quad \tau(A) = \tau(A')$$

if. $K \ \mathrm{conv}_{\beta\eta} \ K'$

$$\text{then} \quad \tau(K) = \tau(K')$$

Definition

$$\sim : A \cup M \longrightarrow \text{untyped } \lambda\text{-calculus}$$

$$\tilde{C} = C$$

$$\tilde{c} = c$$

$$\tilde{x} = x$$

$$\widetilde{AM} = \tilde{A}\,\tilde{M}$$

$$\widetilde{MN} = \tilde{M}\,\tilde{N}$$

$$\widetilde{\Pi x{:}A.B} = \Pi\,\tilde{A}\,\tilde{B}$$

$$\widetilde{\lambda x{:}A.M} = (\lambda y.\ \lambda x.\tilde{M})\,\tilde{A} \qquad y \notin FV(\lambda x M)$$

$$\widetilde{\lambda x{:}A.B} = (\lambda y.\ \lambda x.\tilde{B})\,\tilde{A} \qquad y \notin FV(\lambda x.A)$$

Proposition

$$\Gamma \vdash A : K \quad \text{iff} \quad \tau(\Gamma) \vdash \widetilde{A} : \tau(K)$$

$$\Gamma \vdash M : A \quad \text{iff} \quad \tau(\Gamma) \vdash \widetilde{M} : \tau(A)$$

$$\tau(\Gamma, x : A) \equiv \tau(\Gamma), x : \tau(A)$$

Proof. by structural induction on proofs where $\Pi : \alpha \to \alpha \to \alpha$

☒

Fact if $A \xrightarrow[\beta\eta]{} A'$ $\quad \widetilde{A} \xrightarrow[\beta\eta]{*} \widetilde{A'}$

$M \xrightarrow[\beta\eta]{} M'$ $\quad \widetilde{M} \xrightarrow[\beta\eta]{*} \widetilde{M'}$

☒

Church Rosser for correct terms can be proved using

D. VAN DAALEN's trick.

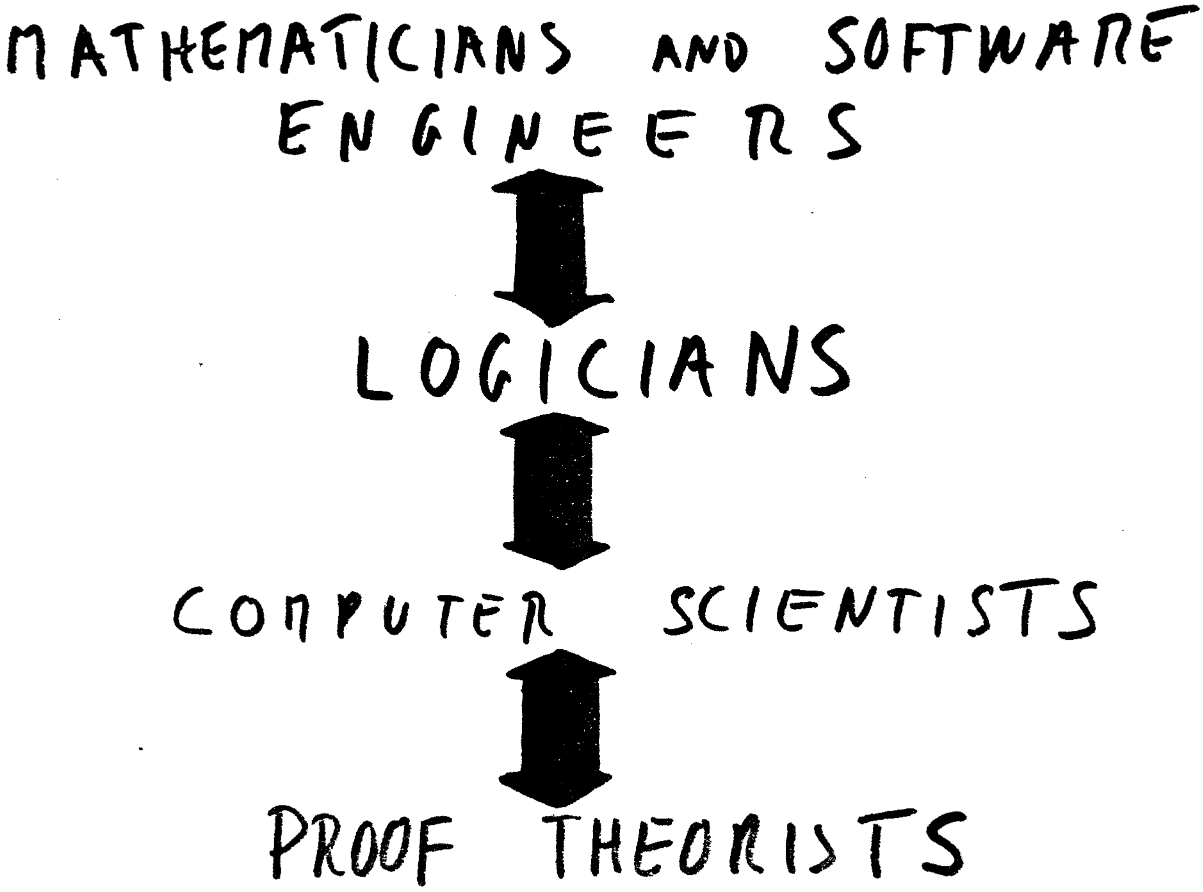
TOWARDS SEARCH SPACES FOR THE EDINBURGH LOGICAL FRAMEWORK.

12

GORDON PLOTKIN

PROOF CHECKING / EDITING / DISCOVERY

THE SOCIAL FRAMEWORK



MATHEMATICIANS AND SOFTWARE ENGINEERS

LOGICIANS

COMPUTER SCIENTISTS

PROOF THEORISTS

ARTIFICIAL INTELLIGENTSIA
LINGUISTS

A CALCULUS OF PROBLEMS

WORK IN LF EXTENDED WITH $A ::= B \to C$

IE FUNCTIONAL TYPES, NOT CONSIDERED AS AN ABBREVIATION. ALSO, FORGET THE SIGNATURES AND ALLOW KIND DECLARATIONS IN CONTEXTS.

PROBLEMS ARE $\Gamma \vdash A$

SUCH THAT $\Gamma \vdash A : \mathrm{Type}$

SOLUTIONS ARE OBJECTS M

SUCH THAT $\Gamma \vdash M : A$

(WILL IDENTIFY UP TO $=_{\beta\eta}$)

THE CALCULUS

PRIM $\Gamma, x{:}A, \Gamma' \vdash A$

→I
$$\frac{\Gamma, x{:}A \vdash B}{\Gamma \vdash A \underset{\to}{\bullet} B}$$

ΠI
$$\frac{\Gamma, x{:}A \vdash B}{\Gamma \vdash \prod_{x:A} B}$$

→E
$$\frac{\Gamma \vdash B \qquad\qquad \Gamma, z{:}C \vdash A}{\Gamma \vdash A}$$

(IF $f : B \to C$ IN Γ)

ΠE
$$\frac{\Gamma, z : C[t] \vdash A}{\Gamma \vdash A}$$

(IF $f : \prod_{y:B} C[y]$ IN Γ, AND $\Gamma \vdash t : B$)

SOLUTIONS ⑥

$\longrightarrow x$

$\longleftarrow \quad t$

$\longrightarrow \quad \lambda x{:}A.t$

$\longleftarrow t$

$\longrightarrow \lambda x{:}A.t$

$\swarrow t$ $\qquad\qquad$ $\swarrow v(z)$

$\longrightarrow v(f(t))$

$\swarrow v(z)$

$\longrightarrow v(f(t))$

<u>CONJECTURE</u> THE CALCULUS IS COMPLETE

THE SEARCH-SPACE VIEW

GOALS ~ PROBLEMS

RULES ~ OPERATORS

(REFINEMENT RULES,
PRIMITIVE TACTICS)

OPERATORS

NORMALISATION

$$\Gamma \vdash A \to B$$

$$|$$

$$\Gamma, x : A \vdash B$$

(AND ONE MORE)

SIMPLE MIDDLE OUT

$$\Gamma \vdash A$$

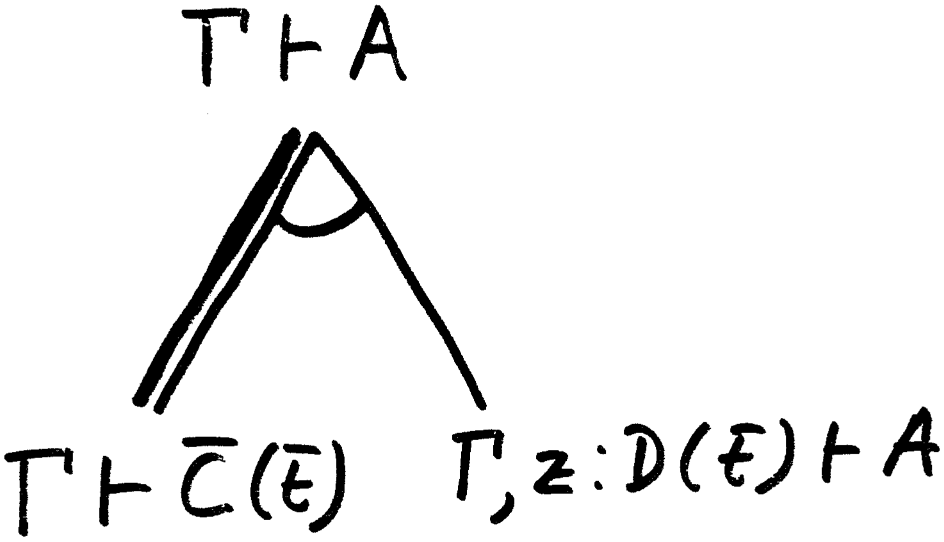
$$\Gamma \vdash B \qquad \Gamma, z : C \vdash A$$

$(f : B \to C \text{ in } \Gamma)$

CONJECTURE CAN ALWAYS NORMALISE
- KEEPS COMPLETENESS.

A DERIVED OPERATOR

MIDDLE-OUT


$\Gamma \vdash A$
$\Gamma \vdash \bar{C}(\bar{t})$
$\Gamma, z : D(\bar{t}) \vdash A$

(WHEN $f : \prod_{\bar{y}:\bar{B}} \bar{C}(\bar{y}) \to D(\bar{y})$ IS IN Γ AND $\Gamma \vdash \bar{t} : \bar{B}$)

CONJECTURE FOR JUDGEMENTS OF THIS FORM IN Γ NEED ONLY USE THE DERIVED OPERATOR.

FURTHER OPERATORS

TOP-DOWN

$$\begin{tikzcd}
\Gamma \vdash A \arrow[d, equal, "\text{(WHEN } D(\bar{t})=A\text{)}"] \\
\Gamma \vdash \bar{C}(\bar{t})
\end{tikzcd}$$

BOTTOM-UP

$$\begin{tikzcd}
\Gamma \vdash A \arrow[d, dash, "\text{(WHEN } \bar{C}(\bar{t}) \in \Gamma\text{)}"] \\
\Gamma, z : D(\bar{t}) \vdash A
\end{tikzcd}$$

$$\begin{tikzcd} {} & v(f(E,\bar{u})) \arrow[l] \\ {} \arrow[d] & {} \arrow[d] \\ \bar{u} & v(z) \end{tikzcd}$$

EXAMPLE

Γ = PROPLOGIC

INCLUDES : $\supset I : \prod_{P,Q:o} (P \to Q) \to (P \supset Q)$

$$\Gamma \vdash \prod_{A,B:o} A \supset (B \supset A)$$

| NORMALISATION

$$\Gamma, A:o, B:o \vdash A \supset (B \supset A)$$

$\Gamma' \stackrel{d}{=}$

| TOP-DOWN ON $\supset I$
WITH $P \mapsto A$, $Q \mapsto B \supset A$

$$\Gamma' \vdash A \to (B \supset A)$$

| NORMALISATION

$$\Gamma', A \vdash B \supset A$$

| TOP-DOWN ON $\supset I$
WITH $P \mapsto B$, $Q \mapsto A$

$$\Gamma', A \vdash B \to A$$

| NORMALISATION

$$\Gamma', A, B \vdash A$$ PRIMITIVE

INTRODUCTION RULES FOR UNARY JUDGEMENTS

$\$I$ $$\overline{C}(\bar{x},\bar{y}) \vdash_{\bar{x}:\bar{A},\,\bar{y}:\overline{B}(\bar{x})} T(\$(\bar{x}))$$

TOP-DOWN OPERATOR

$$\begin{array}{c} \Gamma \vdash \$(\bar{t}) \\ \Big\Vert\, \bar{u} \\ \text{NORMAL FORMOF}(\Gamma \vdash \overline{C}(\bar{t},\bar{u})) \end{array} \qquad (\text{IF } \Gamma \vdash \overline{B}(\bar{u}))$$

NOTE REMOVES NEED FOR MATCHING.

*CAN ONE ALWAYS RESTRICT TO THE TOP-DOWN OPERATOR? - WHATEVER THAT MEANS!

*WHAT ABOUT MULTIARY JUDGEMENTS?

*WHAT ABOUT INDUCTIVELY DEFINED OPERATORS (SEVERAL SUCH $\$I$ AND $\$$ OCCURS IN $\overline{C}$)

EXAMPLES

1. $\supset I$ $\qquad P \to Q \;\; \vdash_{P,Q:o} P \supset Q$

OPERATOR:

$$\begin{tikzcd}
\Gamma \vdash A \supset B \arrow[d, dash] \\
\Gamma, A \vdash B
\end{tikzcd}$$

$\forall I$ $\qquad \vdash_{x:\iota} P(x) \;\; \vdash_{P:\iota\to o} \forall x{:}\iota.\, P(x)$

OPERATOR

$$\begin{tikzcd}
\Gamma \vdash \forall x{:}\iota\; F(x) \arrow[d, dash] \\
\Gamma, x{:}\iota \vdash F(x)
\end{tikzcd}$$

$\exists I$ $\qquad P(x) \;\; \vdash_{P:\iota\to o,\, x:\iota} \exists x{:}\iota\; P(x)$

OPERATOR

$$\begin{tikzcd}
\Gamma \vdash \exists x{:}\iota\; F(x) \arrow[d, dash, "t"] \\
\Gamma \vdash F(t)
\end{tikzcd}$$

(IF $\Gamma \vdash t : \iota$)

ELIMINATION RULES

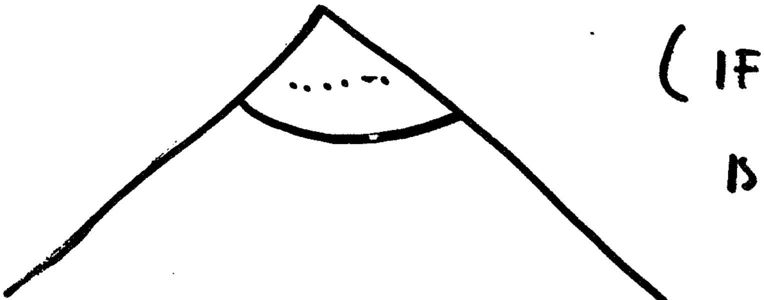
FIRST FORM

$E

$$\bar x : \bar A, \bar w : D(\bar x)\ \frac{T(\$(\bar x)) \qquad \begin{array}{l} \bar C_1(\bar x,\bar w,\bar y_1) \\ |\ \bar y_1 : \bar B_1(\bar x,\bar w) \\ J(\bar x,\bar w) \end{array} \ \cdots\cdots \ \begin{array}{l} \bar C_m(\bar x,\bar w,\bar y_m) \\ |\ \bar y_m : \bar B_m(\bar x,\bar w) \\ J(\bar x,\bar w) \end{array}}{J(\bar x,\bar w)}$$

OPERATOR

$$\Gamma \vdash J(\bar t,\bar u)$$



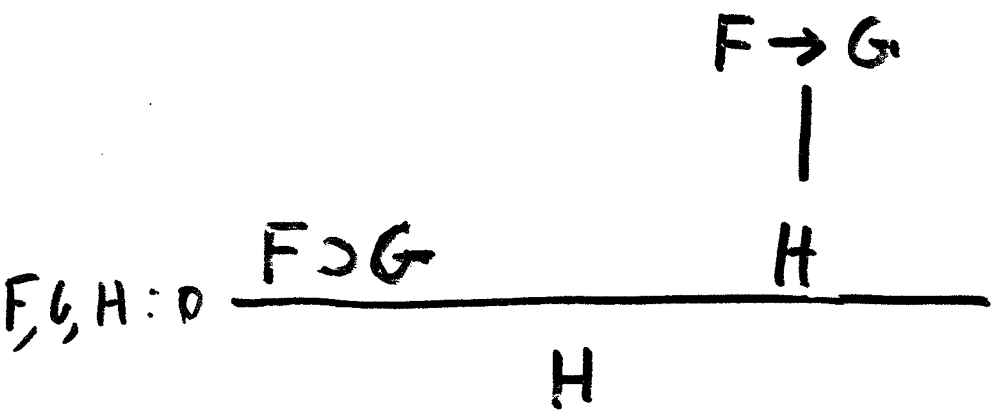
(IF $T(\$(\bar t))$ IS IN Γ)

$$\Gamma,(\bar y_1 : \bar B_1(\bar t,\bar u)),(\bar z_1 : \bar C_1(\bar t,\bar u,\bar y_1)) \vdash J(\bar t,\bar u)$$

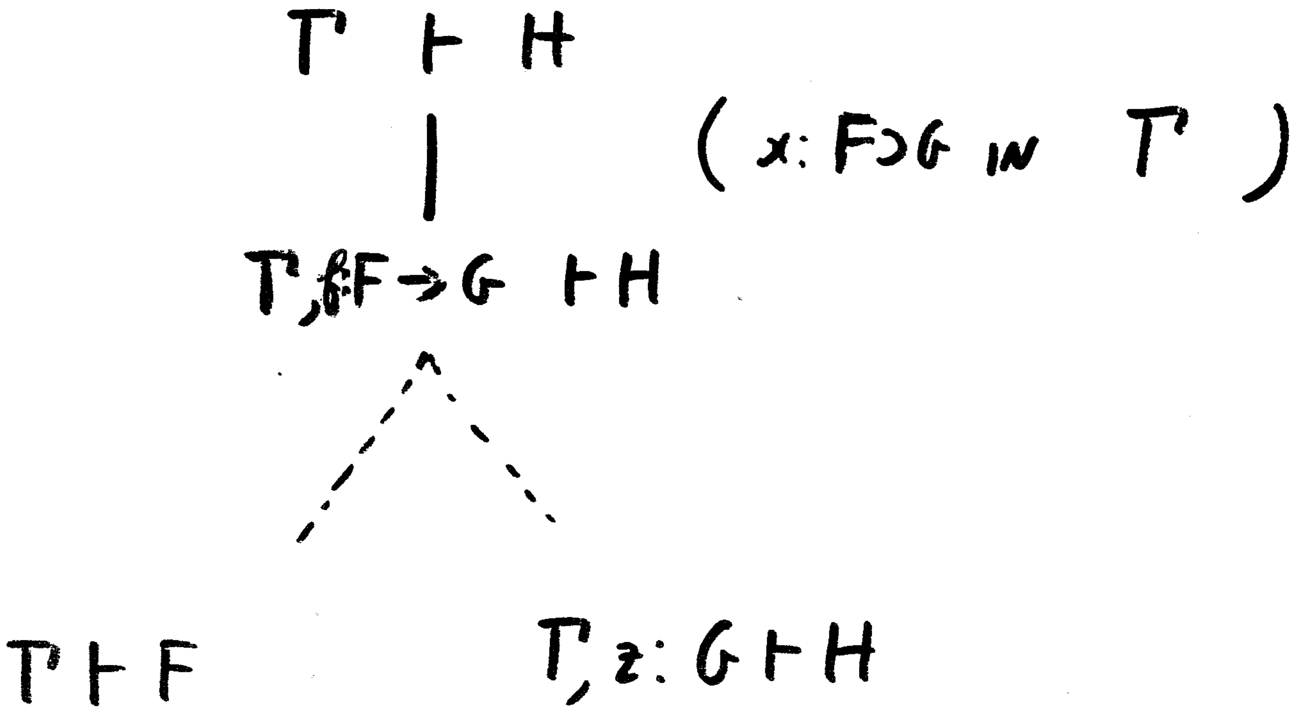
$$\Gamma, \bar y_m : \bar B_m(\bar t,\bar u), \bar z_m : \bar C_m(\bar t,\bar u,\bar y_m) \vdash J(\bar t,\bar u)$$

EXAMPLE

⊃E



OPERATOR



∃E

$$P:\iota\to o,\ Q:o\ \dfrac{\exists x{:}\iota\ P(x) \qquad \begin{array}{c} P(y) \\ \Big|\ y:\iota \\ Q \end{array}}{Q}$$

OPERATOR

$$\begin{array}{c} \Gamma \vdash Q \\ \Big| \\ \Gamma,\ y:\iota,\ z:F(y) \vdash Q \end{array} \qquad (u : \exists x{:}\iota.\ F(x) \text{ in } \Gamma)$$

∀E

$$P{:}\iota\to o,\ Q{:}o\ \cfrac{\forall x{:}\iota\ P(x) \qquad \begin{array}{c}\vdash_{x:\iota} P(x)\\ |\\ Q\end{array}}{Q}$$

OPERATOR

$$\begin{array}{c}\Gamma \vdash Q\\ |\\ \Gamma, (z : \vdash_{x:\iota} F(x)) \vdash Q\end{array} \qquad (\text{IF } x : \forall x{:}\iota\ F(x) \text{ IS IN } \Gamma)$$

WITNESSES AND UNIFICATION

IDEA DO NOT SUPPLY EXTRA INFORMATION (LIKE ON P9) BUT LEAVE OPEN WITH VARIABLES, CAN BE FILLED IN LATER BY SUBSTITUTION, GUESSED PERHAPS BY UNIFICATION.

$$\begin{tikzcd}
\Gamma \vdash \$(\bar{t}) \arrow[d, dash] \\
\Gamma, V\bar{y} : \bar{B}(\bar{t}) \vdash C(\bar{x}, \bar{y})
\end{tikzcd}$$

EXAMPLE

$$a : \exists y\, \forall x\, R(x,y) \vdash \forall x\, \exists y\, R(x,y)$$

$$|$$

$$a : \exists y\, \forall x\, R(x,y),\, x : \iota \vdash \exists y\, R(x,y)$$

$$|$$

$$a : \exists y\, \forall x\, R(x,y),\, x : \iota,\, y : \iota,\, b : \forall x\, R(x,y) \vdash \exists y R(x$$

$$|$$

$$\sim\sim,\, x : \iota,\, y : \iota,\, b : \forall x\, R(x,y),\, Vy' : \iota \vdash R(x,y')$$

$$|$$

$$\sim\sim,\, x : \iota,\, y : \iota,\, \sim\sim\sim,\, Vy' : \iota,\, Vx' : \iota,\, R(x',y) \vdash R(x,y')$$

PRIMITIVELY SOLVABLE

AS $x' \mapsto x$ UNIFIES $R(x',y)$
$y' \mapsto y$

AND $R(x,y')$

AND PROPERLY SO AS.

$$\sim\sim,\, x : \iota,\, y : \iota,\, \sim\sim \vdash y : \iota$$

$$\sim\sim,\, x : \iota,\, y : \iota,\, \sim\sim \vdash x : \iota$$

COUNTEREXAMPLE

$$a : \forall y \, \exists x \, R(x,y) \vdash \exists y \, \forall x \, R(x,y)$$

$$a : \forall y \, \exists x \, R(x,y), \; Vy' : \iota \vdash \forall x . R(x,y')$$

$$a : \forall y \, \exists x \, R(x,y), \; Vy' : \iota, \; x : \iota \vdash R(x,y')$$

$$\sim\sim\sim, \; Vy' : \iota, \; x : \iota, \; y : \iota, \; Vx' : \iota \; R(x',y) \vdash R(x,y')$$

NOT PRIMITIVELY SOLVABLE

AS THE UNIFICATION

$$x' \mapsto x$$
$$y' \mapsto y$$

IS **NOT** PROPER AS

$$\sim\sim\sim \not\vdash y : \iota$$

Implementation of Substitution in the AUTOMATH Verification Program

L.S. Jutting

AUTOMATH	de Bruijn 1967 - now	proofchecking feasable (which is more than decidable)
Verification Program	Zandleven 1971 - 1976	- simulating pointers in Burroughs Algol - controlling memory acces - interactive - big (20,000 lines)
Question	Coquand. Huet 1986	How is substitution implemented?

AUTOMATH

Based on 1. Definitions.

2. λ-calculus.

1. Application of definitions : δ-reduction

if $d(x_n, \ldots, x_0) := D$

then $d(A_n, \ldots, A_0) > s^{x_n \ldots x_0}_{A_n \ldots A_0} D$

2. Application of functions to arguments: β-reduction

$(\lambda x:A.BC) > s^x_C B$

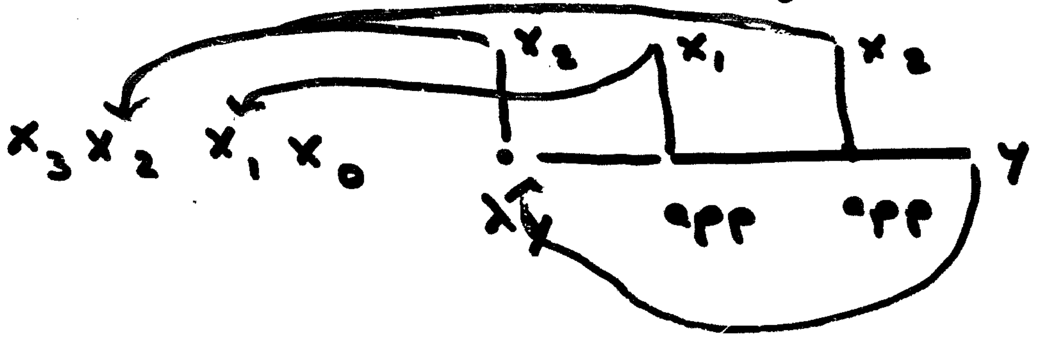
Difficulties with Substitution

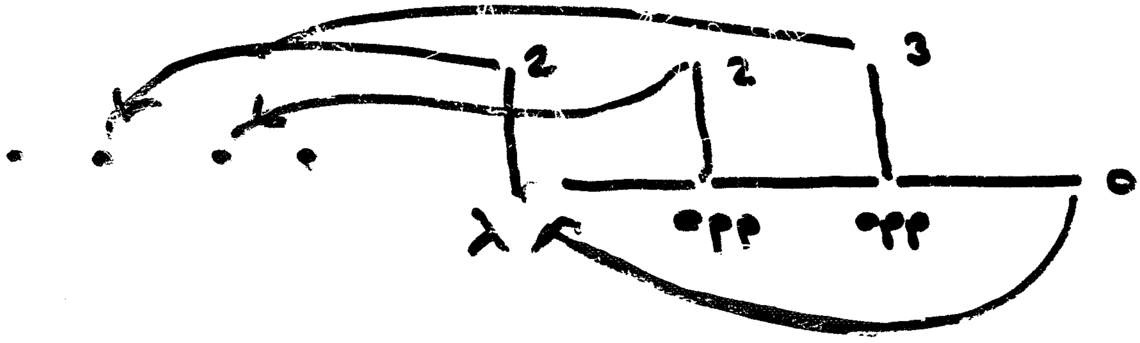
1. α-conversion.

2. copying.

Solutions 1. de Bruijn indices.

(x_3, x_2, x_1, x_0) $\lambda y : x_2 ((y\ x_2) x_1)$





( ) $\lambda 2 . ((0\ 3) 2)$

I: Definition Language

(PAL · SEMIPAL)

variables: $0, 1, 2, 3, \dots$

constants: $p, q, r, \dots \in \mathcal{P}$

expressions:

$\widetilde{T}_d$ | variables

| $p(\bar{A})$ for any sequence $\bar{A} = \dots A_n, \dots A_1, A_0$ of expressions.

<u>Simultaneous substitution:</u> if $\bar{B} = \dots B_n, \dots B_1, B_0$ is a sequence of expressions

then

$$s_{\bar{B}}\, i = B_i$$

$$s_{\bar{B}}\, p(\bar{A}) = p(s_{\bar{B}}\, \bar{A})$$

Thm: $s_{\bar{B}}\, s_{\bar{C}} = s_{s_{\bar{B}} \bar{C}}$

δ reduction

Δ a (partial) function : $\mathcal{P} \to \mathcal{T}$

$$p(\bar{A}) > s_{\bar{A}}\, \Delta(p)$$

Zandleven implementation

contexts : functions $\Gamma : k\omega \to \mathcal{T}_d$

$k \geq 0$ $\mathcal{T}_d$ is the set of terms

$\ldots, B_n, \ldots B_1, B_0, \ldots, A_n, \ldots A_1, A_0$ C

$\leftarrow \ldots$ context Γ $\ldots \rightarrow$

C should be interpreted w.r.t. Γ

operations on contexts:

Extension: $\delta_{\bar{A}}$

Cutting : γ_ω

interpretation: $\mathcal{I}(C, \Gamma) \in \mathcal{T}_d$

$$\mathcal{I}(i, \Gamma) = \begin{cases} \mathcal{I}(\Gamma(i), \gamma_\omega \Gamma) & \text{if } \mathrm{dom}(\Gamma) \neq \emptyset \\ i & \text{if } \mathrm{dom}(\Gamma) = \emptyset \end{cases}$$

$$\mathcal{I}(p(\bar{A}), \Gamma) = p(\mathcal{I}(\bar{A}, \Gamma))$$

Thm $\mathcal{I}(C, \delta_{\bar{A}} \Gamma) = S_{\mathcal{I}(\bar{A}, \Gamma)} C$

Application to δ-reduction

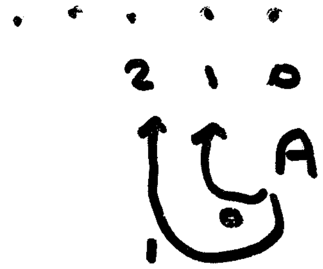
$$I(p(\bar{A}), \Gamma) >_{\delta}$$

$$I(\Delta(p), \delta_{\bar{A}} \Gamma)$$

II: λ-calculus

$\mathcal{T}_\lambda$ | variables: 0, 1, 2, 3, ...
application: (AB)
abstraction: λA.B

Substitution S^n_A



$\lambda B.C$

$$\begin{cases} S^n_A\, i = \begin{cases} i & \text{if } i \neq n \\ U^{n+1}_0 A & \text{if } i = n \end{cases} \\ S^n_A(BC) = (S^n_A B\ S^n_A C) \\ S^n_A\, \lambda B.\underline{C} = \lambda S^n_A B.\, S^{n+1}_A C \end{cases}$$

Updating U^n_k

remains 0

$$\begin{cases} U^n_k\, i = \begin{cases} i & \text{if } i < k \\ n+i & \text{if } i \geq k \end{cases} \\ U^n_k(BC) = (U^n_k B\ U^n_k C) \\ U^n_k\, \lambda B.C = \lambda U^n_k B.\, U^n_{k+1} C \end{cases}$$

Thm's

1. $$U^n_k U^m_l = \begin{cases} U^m_l U^n_{k-m} & k \geq l+m \\ U^{n+m}_l & l \leq k \leq l+m \end{cases}$$

2. $$S^n_A U^m_l = \begin{cases} U^m_l S^{n-m}_A & n \geq l+m \\ U^m_l & l \leq n < l+m \end{cases}$$

3. $$S^n_A S^m_B = S^m_{S^{n-m-1}_A B} S^n_A \qquad n > m$$

β-reduction $(\lambda A.B\ C) > S^0_C B$

(remark: Here the context should be extended)

Zandleven implementation

:$\omega \to (\mathcal{T} \times \omega)$

Contexts: partial functions Γ ~~$\to (\mathcal{T} \times \omega)$~~

$\# \, dom(\Gamma)$ finite

$\bullet \quad \bullet \quad \bullet \; \langle A, l \rangle \bullet \langle B, m \rangle \; \bullet \quad \bullet \qquad\qquad C$

C should be interpreted w.r.t. Γ.

operations on contexts:

Extension: ε^k

(placing extra dots: $\bullet \quad \bullet \quad \bullet$)

Cutting: γ_k

(removing the final entries)

Substitution: $\sigma^{n}_{\langle A, l \rangle}$ (for $n \notin dom(\Gamma)$)

(extending the domain to n)

interpretation $I(C, \Gamma) \in \mathcal{T}_\lambda$

$$I(i, \Gamma) = \begin{cases} U_0^{i+\ell+1} I(B, \gamma_{i+\ell+1} \Gamma) & \text{if } i \in dom(\Gamma),\ \Gamma(i) = \langle B, \ell \rangle \\ i & \text{if } i \notin dom(\Gamma) \end{cases}$$

$$I((AB), \Gamma) = (I(A, \Gamma)\ I(B, \Gamma))$$

$$I(\lambda A . B, \Gamma) = \lambda I(A, \Gamma) . I(B, \varepsilon^1 \Gamma)$$

Thm's 1. $I(U_k^n A, \varepsilon^k \Gamma) = U_k^n I(A, \varepsilon^k \gamma^n \Gamma)$

corr $I(U_0^n A, \Gamma) = U_0^n I(A, \gamma^n \Gamma)$

2. $I(A, \sigma^n_{\langle B, \ell \rangle} \Gamma) = s^n_{B_0}\ I(A, \Gamma)$

where $B_0 = I(U_0^\ell B, \gamma^n \Gamma)$

Application to β-reduction

let

$$\cdots\ \langle\rangle\langle\rangle\cdot\langle\rangle\cdot\langle\rangle\cdot \qquad (AB)$$

$$I(A) \geq I'(\lambda A_1 . A_2)$$

$$\cdots\ \langle\ \rangle\langle\ \rangle\cdot\langle\ \rangle\cdot\langle\ \rangle\cdot \left\{ \underbrace{\langle\ \rangle\langle\ \rangle\langle\ \rangle\langle\ \rangle}_{\ell} \right. \qquad \lambda A_1 . A_2$$

then $I((AB)) \geq I''(A_2)$

$$\cdots\ \langle\ \rangle\langle\ \rangle\cdot\langle\ \rangle\cdot\langle\ \rangle\cdot \left\{ \langle\ \rangle\langle\ \rangle\langle\ \rangle\langle\ \rangle\langle B,\ell\rangle \right. \qquad A_2$$

or: If $I(A,\Gamma) \geq I(\lambda A_1 . A_2, \Gamma')$

$\Gamma = \gamma^{\ell}\Gamma'$

then $I((AB),\Gamma) \geq I(A_2, s^{0}_{\langle B,\ell\rangle} c'\Gamma')$

Combination of I & II

This is straightforward.
Takes some time.

Combination of Zandleven implementations.

[The same
A paper will appear.

Moreover:

Also typing (of variables, constants and expressions) can be implemented using this device.

Discussion of results

1. Zandleven implementation is adequate to describe substitution & typing

2. Zandleven implementation is adequate to describe outside β-reduction & δ-reduction

3. Zandleven implementation does not require copying, and therefore reduces the required memory space.

4. As to aspects of performance I have no hard facts.
   The AUTOMATH checker is rather good for a program over 10 years old.

L.S. Jutting
dept. of Mathematics &
Computing Science
Univ. of Technology, Eindhoven
the Netherlands.

Partial inductive definitions

by

Lars Hallnäs

Swedish Institute of Computer Science

Box 1263, 163 13 Spånga, Sweden

Let us assume that f is a function defined by a set of equations E:

$$f(0) = a$$

$$f(n+1) = g(h(n+1), f(n))$$

.

.

.

f(n) is *defined* if we can compute f(n) in E into a unique canonical value in a finite number of steps using usual rules of substitution. One could say that f(3) in this case is *totally* defined since given an equation $f(3) = g(h(3),f(2))$ f(3) is at least *partially* "defined" in some sense even if we cannot compute a unique canonical value in a finite number of steps. This distinction is basically motivated from an intensional point of view. It *is* somehow a matter of definability. When we write down the equations in E we *intend* to define a certain function and if we read E with this act of intent in mind "f(n) is undefined" ought simply to mean that there is no equation f(n) = . . . in E. The situation is similar when it comes to define semantics for formal languages and the like. Take the usual situation: we define True(A) by recursion on some wellfounded relation. This corresponds to defining f by primitive recursion. Somehow this not only consists of an act of intent, but also a proof that this act totally fulfills the given intentions. That is of course something much more complex than just writing down your intentions. But just as in the case with a function f there is a basic difference between being partially defined and not being defined at all. This intensionally based distinction is perhaps of interest when we focus on the definitions themselves rather than the abstract objects they should present. So we do not just consider partial definitions in the sense that the definition does not cover the whole universe U, but also objects in U that are only partially defined by the given definition. The question of isolating total fragments that one can believe in seems often enough to have very little to do with the basic intuitions on how to define True(A) for a

15

given A of some form. So perhaps the basic theory with its syntax and semantics may be simpler if we accept partial objects in defining propositions. We know that the problems concerning establishing that certain theories really has a totally defined notion of truth as a basis is not a formal matter, but a question of belief. The question is if not the task to present a fomal language with its syntax and semantics in a certain sense is a "formal" matter and hence elementary?

In this note I will try to discuss some aspects of partially defined propositions based on an attempt to interpret a certain class of eventually non monotone inductive definitions as partial inductive definitions. The basic interpretation studied here is closely connected with work in general proof theory done by Martin-Löf,Prawitz and Schroeder-Heister (See [M1,3],[P1,2,3,4],[SH1,2,3]) and generalizes Aczels characterization of monotone inductive definitions (See [A]).

1. Partial inductive definitions

Syntax

Let U be a given universe of atoms a,b,c. . . Intuitively we may think of U as a universe of propositions.

If E is a class of clauses over U and atoms in U and e is an atom in U, then $E \Rightarrow e$ is a clause over U.

Let Clause(U) denote the class of clauses over U. Let F,E,. . . denote classes of clauses and atoms and let A,B,. . . denote an arbitrary clause or atom.

The level of a clause or atom A is given as follows:

$\text{level}(a) = 0$

$\text{level}(E \Rightarrow e) = \max\{\text{level}(C) \mid C \in E\} + 1$

$\text{level}(E) = \max\{\text{level}(C) \mid C \in E\}$

Syntactically then a partial inductive definition is a class P of clauses over some universe U. Monotone inductive definitions corresponds to partial inductive definitions of level ≤ 1. See the interpretation given by Aczel in [A].

Semantics

To give a semantics in this context means to explain what predicate Def(P) a given definition P defines. The standard interpretation of monotone inductive definitions gives the following characterization of Def(P):

(*) $e \in \text{Def}(P)$ iff there is a clause $E \Rightarrow e$ in P such that $E \subseteq \text{Def}(P)$.

It seems reasonable to think of (*) as giving the basic intuitions concerning the interpretation of inductive definitions. Intuitively (*) says that e satisfies Def(P) iff there is a clause $E \Rightarrow e$ in P such that all $C \in E$ "satisfy" Def(P). The question is now how to interpret this "satisfy" when $\text{level}(C) > 1$. The intuition behind the interpretation given here is simply that "$E \Rightarrow e$ satisfies Def(P)" means that "e follows from E in P". More precisely we think of P as implicitly defining a consequence relation $\vdash$ and thereby give a local meaning to "follows from". The meaning of "C follows from E in P" - $E \models_P C$ - will be given by a calculus of sequents generated by P in a uniform manner. For the motivation of the different rules we need the notion of a total object relative to $\vdash$. Intuitively the class of total objects w.r.t. $\vdash$ is the class of objects for which $\vdash$ make sense as "follows from".

A predicate C is a totality candidate w.r.t. P and $\vdash$ if it satisfy the following conditions:

(i) if $e \in C$ and $(E \Rightarrow e) \in P$, then $E \subseteq C$

(ii) if $(E \Rightarrow e) \in C$, then $E \cup \{e\} \subseteq C$

(iii) if $A \in C$, then $F \vdash A$ and $G,A \vdash B$ implies $F,G \vdash B$ for all F and G.

Let $T(P) = \cup\{C \mid C$ is a totality candidate$\}$.

Clearly T(P) is a totality candidate and thus the largest totality candidate.

A relation $\vdash$ is called a consequence realtion if it satisfy the following:

Let F,E be short for $F \cup E$ and F,C short for $F \cup \{C\}$.

$F \vdash E$ will be short for $\ldots F \vdash C \ldots (C \in E)$.

$$\vdash\Rightarrow \qquad \frac{F,E \vdash e}{F \vdash E \Rightarrow e}$$

$$\Rightarrow\vdash \qquad \frac{F \vdash E \qquad F,e \vdash C}{F,E \Rightarrow e \vdash C}$$

The $\vdash\Rightarrow$ rule explains the meaning of "$E \Rightarrow e$ holds in P" namely that e follows from E in P. The $\Rightarrow\vdash$ rule explains what it means to assume $E \Rightarrow e$. If $E \Rightarrow e$ holds, then e follows from E.

If $E \Rightarrow e$ is a total object this means that if E follows from F anything that follows from F,e must follow from $F,E \Rightarrow e$: $F \vdash E$ and $F,E \vdash e$ implies $F \vdash e$ and given $F,e \vdash C$ this implies $F \vdash C$.

A consequence relation $\vdash$ is called P - closed if it satisfy the following conditions:

let $D(e) = \{E \mid (E \Rightarrow e) \in P\}$ and let $F,D(e) \vdash C$ be short for $\ldots F,E \vdash C \ldots (E \in D(e))$.

$$\vdash P \qquad \frac{F \vdash E}{F \vdash e}$$

where $(E \Rightarrow e) \in P$

$$P\vdash \qquad \frac{F, D(e) \vdash C}{F,e \vdash C}$$

The $\vdash$P rule explains the meaning of "e holds in P" namely that there is a definitional clause $E \Rightarrow e$ such that definiens E holds in P. The P$\vdash$ rule explains what it means to assume e. If e holds, then E holds for some definitional clause $E \Rightarrow e$. So assume e is a total object and that e holds beacuse E holds for $E \Rightarrow e$ in P. Then $F \vdash E$ and $D(e) \vdash C$ implies $F \vdash C$.

So to assume an object C does not only mean that we assume C holds, but also that the meaning P gives to C make sense.

Now let $\models_P$ be the smallest P-closed consequence relation containing all $F, e \vdash e$.

<u>Proposition 1.1</u> $\models_P e$ iff there is a clause $E \Rightarrow e$ in P such that $\models_P E$.

<u>Proof</u>: trivial.

So let $\mathrm{Def}(P) = \{e \mid \models_P e\}$.

Clearly this interpretation will cover also the standard interpretation of monotone inductive definitions. $\models_P$ will satisfy the usual conditions on reflexivity and monotonicity, but not necessarily transitivity. Since $\models_P$ is implicitly defined by P itself this eventual lack of transitivity shows that the meaning P gives to various objects is not allways the intended one. Note that there is a sharp distinction between $F, A \models_P B$ and $\models_{P,A} B$. To assume something holds given a certain definition is not the same as adding new clauses to a given definition.

Let us call P <u>total</u> if $T(P) = \mathrm{Clause}(U) \cup U$.

Consider the definition P consisting of the single clause $(a \Rightarrow b) \Rightarrow a$. This definition gives the general structure of paradoxes like Russell's paradox and provide a simple example of partial objects: $\models_P a$ and $\models_P a \Rightarrow b$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{a \vdash a} \qquad \overline{b \vdash b}}{a, a \Rightarrow b \vdash b}\,(\Rightarrow\vdash)}{a \vdash b}\,(P\vdash)}{\vdash a \Rightarrow b}\,(\vdash\Rightarrow)}{\vdash a}\,(\vdash P)$$

But not $\models_P b$ since $D(b) = \varnothing$. So there is no totality candidate C relative to P and $\models_P$ such that $a \in C$.

Let us call A <u>false</u> in P if $\{A\} \models_P B$ for all B. P will then be referred to as a <u>complete</u> definition if for all a inU either $\models_P a$ or a is false in P.

So $(a \Rightarrow b) \Rightarrow a$ also give a simple example of a complete definition which is not total.

Proposition 1.2

(i) If P is complete, then $\models_P C$ or C is false in P for all C.

(ii) P is total and complete iff $\models_P$ means "if. . .then. . .".

Proof:

(i) By a simple induction on C.

Assume C is $E \Rightarrow e$, then if $\models_P e$ by monotonicity $E \models_P e$. So assune e is false in P. If $\models_P E$

$$\frac{\vdash E \qquad e \vdash C}{E \Rightarrow e \vdash C}$$

If some $C \in E$ is false in P, then $C \models_P e$, so by monotonicity $C,E \models_P e$.

(ii) Assume that P is total and complete. If $F \models_P C$ and $\models_P F$, then $\models_P C$ since P is total. So assume that $\models_P F$ implies $\models_P C$. If $\models_P F$, then $\models_P C$ and by monotonicity $F \models_P C$. If $\models_P F$ does not hold, then $F \models_P C$ follows since P is complete.

Assume that $F \models_P C$ is equivalent with $\models_P F$ implies $\models_P C$. If $F \models_P C'$ for all $C' \in G$, $G \models_P C$ and not $F \models_P C$, then $\models_P F$ and not $\models_P C$. This is absurd since $\models_P F$ implies $\models_P G$ and thus $\models_P C$. Trivially P has to be complete since if not $\models_P C$ by assumption $\{C\} \models_P C'$ for any C'.■

Let P be given by

$\Rightarrow T(p)$ (p a given propositional variable)

$(T(X) \Rightarrow T(Y)) \Rightarrow T(X \rightarrow Y)$ (X,Y propositional sentences)

Clearly P is complete. A slight modification of a standard argument for cut elimination in the implication calculus shows that P also is total. Let P' be the definition we get from P by deleting $\Rightarrow T(p)$ and adding $T(p) \Rightarrow T(p)$ for all propositional variables p. P' is clearly total, but not complete. T(p) does not hold and T(p) is not false in P'. P' gives a definition of a notion of logical truth in the implication calculus while P defines truth for the interpretation where only p is given the truth value true.

The natural operator $\Phi_P : P(U) \rightarrow P(U)$ associated with P is given as follows:

If $e \in X$, then X is e-closed.

If X is C-closed for all $C \in E$ implies $e \in X$, then X is $E \Rightarrow e$ - closed.

$\Phi_P(X) = \{e \mid$ there is a clause $E \Rightarrow e$ in P such that X is C-closed for all $C \in E\}$.

Proposition 1.3

(i) If P is total and complete, then Def(P) is the smallest fixed point of Φ_P.

(ii) If Φ_P has a fixed point X, then $\mathrm{Def}(P) \subseteq X$.

Proof:

(i) First note that if P is total and complete, then $\models_P C$ is the same as saying that Def(P) is closed under C. We use induction on C to see this:

Assume $\models_P E \Rightarrow e$. If Def(P) is closed under all $C \in E$, then by IH $\models_P E$ and since P is total we have $e \in \mathrm{Def}(P)$.

If on the other hand Def(P) is closed under $E \Rightarrow e$ we may assume $\models_P E$ and by IH it follows that Def(P) is closed under all $C \in E$, so $e \in \mathrm{Def}(P)$. Since P is complete this means $\models_P E \Rightarrow e$.

So assume $e \in \mathrm{Def}(P)$, then $\models_P E$ for some $(E \Rightarrow e) \in P$ which means that Def(P) is closed under all $C \in E$, so $e \in \Phi_P(\mathrm{Def}(P))$. If $e \in \Phi_P(\mathrm{Def}(P))$, then Def(P) has to be closed under all $C \in E$ for some $(E \Rightarrow e) \in P$ thus $\models_P E$, so $e \in \mathrm{Def}(P)$.

Now let X be another fixed point of Φ_P. Define $F \vdash_X C$ to hold iff if X is closed under all $C' \in F$, then X is closed under C. Then $\vdash_X$ is a P - closed consequence relation:

Clearly $\vdash_X$ is a consequence relation.

If $F \vdash_X E$ for some $(E \Rightarrow e) \in P$, then if X is closed under all $C \in F$ we have $e \in \Phi_P(X) = X$, so $F \vdash_X e$. If $F, E' \vdash_X C$ for all $E' \in D(e)$ and X is closed under all $C' \in F \cup \{ e \}$, then since X = $= \Phi_P(X)$ there must be some $(E' \Rightarrow e) \in X$ such that X is closed under all $C' \in E$, and thus X is closed under C. Since $\models_P$ is the smallest P - closed consequence relation we have $\mathrm{Def}(P) \subseteq X$.

(ii) Follows from the argument for (i). ∎

We will think of a property over U as a partal inductive definition over U. Thus to prove by induction that P is "included" in P' means simply to prove that $\models_{P'}$ is P-closed.

2 Natural deduction - total objects

Let P be a partial inductive definition. The basic associated natural deduction calculus NP is given by the following rules:

assumptions $\qquad C$

$\Rightarrow$ I

$$\dfrac{\begin{array}{c}\cancel{E}\\ \vdots\\ e\end{array}}{E \Rightarrow e}$$

$\Rightarrow$ E

$$\dfrac{\begin{array}{c}\vdots\\ \ldots C \ldots\end{array}\;(C \in E) \quad \begin{array}{c}\vdots\\ E \Rightarrow e\end{array}}{e}$$

P I

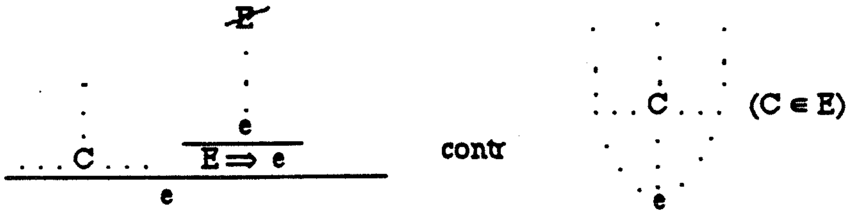
$$\dfrac{\begin{array}{c}\vdots\\ \ldots C \ldots\end{array}}{e} \quad (C \in E)$$

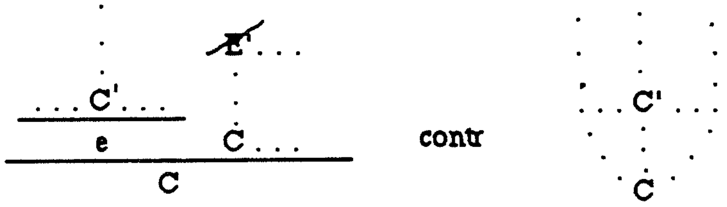
for $(E \Rightarrow e)$ in P

P E

$$\dfrac{\begin{array}{c}\vdots\\ e\end{array} \quad \begin{array}{c}\cancel{E'} \ldots\\ \vdots\\ C \ldots\end{array}}{C} \quad E' \in D(e)$$

A deduction of C from F in NP is as usual built up from assumptions using the rules of inference. $E \Rightarrow e$ is the major premise in $\Rightarrow$ E inferences and e the major premise in PE inferences. A cut in a deduction is an atom or a clause which is the conclusion of an application of a I inference and at the same time the major premise of a E inference. If a deduction *D* ends with a E inference, then we may follow major premises of E inferences upwards in *D* until we reach a cut - the main cut of *D* - or an assumption. The branch we followed in *D* is called the main branch of *D* . We have the following rules of contraction for eliminating cuts:





If *D* has a main cut let Con(*D*) denote the contraction of *D* at this cut.

Given this notion of contraction we define what it means for a deduction *D* to be normalizable.

If *D* is an assumption, then *D* is normalizable.

If *D* ends with a I inference, then *D* is normalizable if the premisedeductions of this inference are all normalizable.

If *D* ends with a E inference, then *D* is normalizable if *D* has no main cut and all minor deductions along the main branch are normalizable or *D* has a main cut and Con(*D*) is normalizable.

D is said to be normal if *D* is cut free. If all deductions in NP are normalizable, then if there is a deduction of C from F in NP there is also a normal deduction of C from F in NP.

Proposition 2.1 $F \models_p C$ iff there is a normal deduction of C from F in NP.

Proof: If $F \models_p C$, then we have a proof in a certain cut free calculus of sequents. The standard translation of proofs in such a calculus into deductions in a system of natural deduction gives us a normal deduction of C from F. We argue by a simple induction on the given proof. Let us just consider a case in the induction step to illustrate how this translation is carried out:

assume our proof ends with an application of the PE rule. By IH we have normal deductions of C from E' for all E' $\in$ D(e), so

$$\dfrac{e \qquad \begin{array}{c}\cancel{E'}\ldots\\ \vdots\\ C\ldots\end{array}}{C}$$

gives us a normal deduction of C from F,e.

So assume we have a normal deduction *D* of C from F in NP. It follows then easily by induction on the lenght of this deduction that $F \models_p C$.

If *D* = C, then obvously $F \models_p C$.

If *D* ends with an application of the $\Rightarrow$ I or PI rule, then it follows directly from IH that $F \models_p C$.

If *D* ends with a $\Rightarrow$ E or PE inference, then it has a main branch since it is normal. *D* will then have one of the two following forms:

$$\begin{array}{c}\dfrac{E \Rightarrow e \qquad \begin{array}{c}\vdots\\ \ldots C' \ldots\end{array}}{e}\ (C' \in E)\\ \vdots\\ C'\end{array}$$

$$\begin{array}{c}\dfrac{e \qquad \begin{array}{c}\cancel{E'}\ldots\\ \vdots\\ C'\ldots\end{array}}{C'}\\ \vdots\\ C\end{array}$$

In the first case by IH $F,e \models_p C$ and $F \models_p C'$ for $C' \in E$, so $F \models_p C$. In the second case we have

$$\begin{array}{c} E' \\ \vdots \\ C' \\ \vdots \\ C \end{array}$$

Apply IH to this deduction and we have $F,E' \models_P C$ for $E' \in D(e)$ and thus $F \models_P C$.■

So the property of being a total definition corresponds to the normal form property of NP: if $F \models_P A$ and $G,A \models_P B$, then we have normal deductions of A from F and B from G,A. Put these deductions together and we have a deduction of B from F,G. The normal form property assure that there is a normal deduction of B from F,G thus $F,G \models_P B$.

One may conversely interpret the normal form theorem as stating that the definition shown by the introduction rules of a natural deduction calculus is total.

The NP calculus is a formal one in the sense that the rules can be given a semantical motivation based on P only if P is a total definition. So the "real" calculus is to be thought of as embedded in NP. We will consider a calculus NT(P) based on the notion of a total object:

assumptions $\qquad C$

$\Rightarrow$ I
$$\frac{\begin{array}{c} \cancel{E} \\ \vdots \\ e \end{array}}{E \Rightarrow e}$$

provided $E \Rightarrow e$ is in T(P)

$\Rightarrow$ E
$$\frac{\begin{array}{c} \vdots \\ \ldots C \ldots \end{array} : (C \in E) \qquad \begin{array}{c} \vdots \\ E \Rightarrow e \end{array}}{e}$$

P I

$$\dfrac{\overset{\vdots}{\ldots C \ldots}}{e} \quad (C \in E)$$

for $(E \Rightarrow e)$ in P

where e is in T(P)

P E

$$\dfrac{\overset{\vdots}{e} \qquad \overset{\cancel{E'}\ldots}{\overset{\vdots}{C}} \ldots}{C} \quad E' \in D(e)$$

The notions of contraction, cut, normalizability etc. . . are formulated as for NP.

Proposition 2.2 If D is a deduction of C from F in NT(P) and F consists of total objects, then C is also a total object.

Proof: By induction on the lenght of D .

If D is just an assumption, then we are done.

If D ends with an I inference, then by definition of NT(P) C has to be a total object.

So assume D ends with an E inference.

$$\dfrac{\overset{\vdots}{\ldots C \ldots} (C \in E) \qquad \overset{\vdots}{E \Rightarrow e}}{e}$$

By IH $E \Rightarrow e$ is a total ɔbject and thus e is also a total object.

$$\dfrac{\overset{\vdots}{e} \qquad \overset{\cancel{E'}\ldots}{\overset{\vdots}{C}} \ldots}{C} \quad E' \in D(e)$$

By IH e is a total object, so if E' is in D(e), then E' has also to be a total object. By IH then C has to be a total object. ■

Proposition 2.3 If $F \cup \{C\}$ consists of total objects, then

(i) if $F \models_P C$, then there is a deduction of C from F in NT(P),

(ii) if there is a deduction in NT(P) of C from F, then $F \models_P C$.

Proof:

(i) By an easy induction on the lenght of the proof of $F \models_P C$. In order to apply IH in the $\vdash \Rightarrow$ and $\vdash P$ cases we use the fact that if $E \Rightarrow e$ is in T(P), so are all objects in $E \cup \{e\}$ and if e is in T(P) and $E \Rightarrow e$ is in P, then all objects in E are in T(P).

(ii) So assume there is a deduction *D* of C from F in NT(P).

If *D* consist of just an assumtion then clearly $F \models_P C$.

Assume *D* ends with an I inference. consider the PI rule as an example:

Since e is total, then E consists of total objects. So by IH $F \models_P E$. Thus $F \models_P e$.

Assume *D* ends with an E inference:

$$\frac{\begin{matrix} \vdots & \vdots \\ \ldots C \ldots (C \in E) & E \Rightarrow e \end{matrix}}{e}$$

Since F consists of total objects by proposition 2.2 $E \cup \{E \Rightarrow e\}$ consists of total objects. Thus we may apply IH to get $F \models_P E$ and $F \models_P E \Rightarrow e$. Clearly $F, E \models_P e$. Now E consists of total objects and thus $F \models_P e$.

$$\frac{\begin{matrix} & E' \ldots \\ \vdots & \vdots \\ e & C \ldots \end{matrix}}{C} \quad E' \in D(e)$$

Using proposition 2.2 we see that e is a total object, each E' will consists of total objects. So we may apply IH to get $F \models_P e$ and $F, E' \models_P C \ldots$ So $F, e \models_P C$ and since e is a total object we have $F \models_P C$. ■

3 Clauses as types

NP is a system of natural deduction in the usual sense, so we may consider some notion of realizability similar to the one studied by Martin-Löf and others in the tradition of Curry and Howards. So we think of clauses and atoms as types and the objects realizing these types will be descriptions of the structure of deductions giving the clauses and atoms as conclusions:

assumptions $\quad x : C$

$$\lambda \qquad \begin{array}{c} x \cancel{: C} \ldots \qquad (C \in E) \\ \vdots \quad \vdots \\ \dfrac{b(x\ldots) : e}{\lambda\, x\ldots b(x\ldots) : E \Rightarrow e} \end{array}$$

$$\text{Ap} \qquad \begin{array}{c} \vdots \qquad\qquad \vdots \\ \dfrac{a : E \Rightarrow e \qquad b : C \ldots}{a(b\ldots) : e} \quad (C \in E) \end{array}$$

$$(\;) \qquad \begin{array}{c} \vdots \\ \dfrac{a : C \ldots}{(a\ldots) : e} \quad (C \in E) \end{array}$$

$$[\;] \qquad \begin{array}{c} \qquad\qquad x \cancel{: C'} \ldots \quad (C' \in E') \\ \vdots \qquad\qquad \vdots \qquad \vdots \qquad\qquad\qquad (E' \in D(e)) \\ \dfrac{a : e \qquad b(x\ldots) : C \ldots}{[\,\lambda\, x\ldots b(x\ldots)\ldots\,]\, a : C} \end{array}$$

An object is called canonical if it is of the form (a. . .) or the form λx. . .b(x. . .). The computation rules for non canonical objects are as usual based on the rules of contraction:

$$\dfrac{\dfrac{\begin{matrix} x \cancel{: C} \ldots \\ \vdots \\ b(x\ldots) : e \end{matrix}}{\lambda x\ldots b(x\ldots) : E \Rightarrow e} \qquad c : C\ldots}{\lambda x\ldots b(x\ldots)(c\ldots) : e} \qquad \text{contr} \qquad \begin{matrix} \vdots \\ c : C\ldots \\ \vdots \\ b(c\ldots) : e \end{matrix}$$

$$\dfrac{\dfrac{\begin{matrix} \vdots \\ a : C'\ldots \end{matrix}}{(a\ldots) : e} \qquad \begin{matrix} x \cancel{: C'} \ldots \\ \vdots \\ b(x\ldots) : C\ldots \end{matrix}}{[\lambda x\ldots b(x\ldots)\ldots](a\ldots) : C} \qquad \text{contr} \qquad \begin{matrix} \vdots \\ a : C'\ldots \\ \vdots \\ b(a\ldots) : C \end{matrix}$$

As usual these computationrules preserve types. So we have what corresponds to partial correctness: if a realizes C and can be computed into a canonical object b, then also b realizes C.

4 Notions of computability an validity

Tait introduced an elegant and powerful method for proving normalization in systems of typed λ–calculus by giving an intensional notion of computability for terms. (See [T]). This method was adopted for proving normalization in systems of natural deduction by Martin-Löf. (See [ML1]). Usually this notion of computability is defined by recursion on some wellfounded relation. This means that in order to give such an intensional notion of computability our systems must be built up in a certain syntactically wellfounded manner. It seems to me that the mere possibilty of defining such a notion should not depend on syntactical properties of the rules of a given system. What should be important is that we are concerned with an interpretation of these rules based on the notion of substitution that is in terms of a system of natural deduction or a system of λ-terms. So let us consider a general notion of computability and validity as given by a partial inductive definition:

$$\Rightarrow I \qquad \dfrac{\begin{array}{c}\cancel{E}\\ \vdots \\ e\end{array}}{E \Rightarrow e}$$

is valid if

$$\begin{array}{c}\vdots\\ E\\ \vdots\\ e\end{array}$$

is valid for all valid deductions of clauses in E.

$$(E \Rightarrow e) \qquad \dfrac{\begin{array}{c}\vdots\\ \ldots C \ldots\end{array}}{e} \qquad (C \in E)$$

is valid if all the premise deductions of . . .C. . . are valid.

A deduction *D* on ⇒ E or PE form is valid if it has a main cut and Con(*D*) is valid.

Let V be the class of clauses associated with this inductive definition.

Proposition 4.1

If $\models_V D$ and D is on $\Rightarrow$ E or PE form, then D reduces to some D' on $\Rightarrow$ I or PIform such that $\models_V D'$.

Proof: If $\models_V D$ this can only be beacuse $\models_V$Con(D), so in finitely many steps we must reach some D' such that D' is on $\Rightarrow$ I or PI form and $\models_V D'$.■

Let us write $\vdash_V$C for "there is a deduction D of C from F which is valid" .

Proposition 4.2

$\vdash_V$e iff there is a clause E $\Rightarrow$ e in P such that $\vdash_V$E.

Proof: Assume $\vdash_V$e. So we have a deduction D of e from some F which is valid. Now if D is not already on (E $\Rightarrow$ e) form for some E it must reduce to such a deduction D' according to 4.1. By definition it then follows that there are valid deductions of the clauses in E, so $\vdash_V$E.

And trivially if we have valid deductions of the clauses in E for some E $\Rightarrow$ e in P, then we may apply the PI rule to form a valid deduction of e.■

Prawitz has used such a general notion of validity as a central concept in discussions on a foundation of a general proof theory. (See [P1,2,3]). But his notion is based on recursively defined concepts and so not directly an example of a partial definition.

Let us form a definition V' by adding the following clauses to V:

all assumptions C satisfy V'

if D is on E form have a cut free main branch and all minor deductions along the main branch are normalizable, then D satisfies V'.

This means we add a base to V. V' is Martin-Löfs notion of a computable deduction seen as a partial inductive definition.

Proposition 4.3 If $\models_{V'} D$, then D is normalizable.

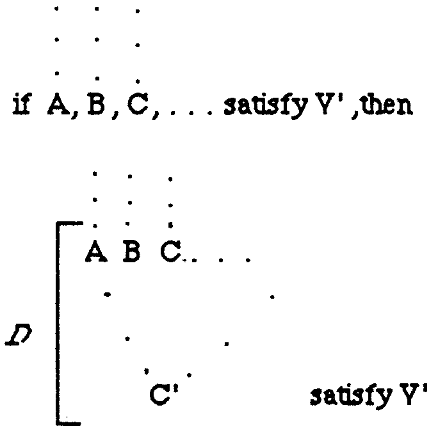
Proof: A simple induction on V'. Let us say that a clause E $\Rightarrow D$ is normalizable if D is normalizable provided all C $\in$ E are normalizable. Then let $\vdash$ be the consequence relation given by "if all C $\in$ E are normalizable, then C' is normalizable". What we have to show is that $\vdash$ is V' closed. We have to inspect the various cases. Let us as an example consider the case where D

ends with a PI inference. What we have to show is that $F \vdash \mathcal{D}$ provided $F \vdash \mathcal{D}'. \dots$ for premise deductions of $\mathcal{D}$. Clearly if all clauses and atoms in F are normalizable, then $\mathcal{D}'. \dots$ are all normalizable and so is then $\mathcal{D}$ by definition. On the other hand assume $F,\mathbf{D}(\mathcal{D}\ ) \vdash C$ holds and that all clauses and atoms in $F,\{\mathcal{D}\ \}$ are normalizable, then clearly all deductions in $\mathbf{D}(\mathcal{D}\ )$ are normalizable and thus C holds. ■

<u>Proposition 4.4</u> If V' is a total and complete definition, then $\models_{V'}\mathcal{D}$ for all deductions $\mathcal{D}$.

<u>Proof</u>: This is proved in the usual way using the fact that if V' is a total and complete definition, then $\models_{V'}$ simply means "if. . .then. . .".

Thus by induction on the lenght of $\mathcal{D}$ we prove:



Since V' is total and complete V' corresponds exactly to the what we intended in writing "if. . .then. . ." in our informal presentation of V'. Thus it easy to see that the induction goes through in the usual way. (See Martin-Löf [ML1] and Prawitz [P1] for canonical examples of such proofs). ■

For λ-terms we have the corresponding notion of computability:

Assume we have defined a suitable one step reduction procedure for non canonical objects c → c'.

(a. . .) is <u>computable</u> if a. . . are all computable,

λx. . .b(x. . .) is <u>computable</u> if b(c. . .) is computable for all computable c. . . of appropriate type,

a non canonical object c containing a redex is <u>computable</u> if c' is computable for c → c'.

Let us say that a clause C is realized by c if c : C holds.

4.4 <u>Proposition</u>

If C is realized by a computable object c, then C is realized by a computable canonical object c'.

Proof: Any appropriate notion of a one - step reduction based on the contraction rules should satisfy:

if a : C and a → b, then b : C.

If c : C and c is a computable non canonical object, then clearly c → . . . → c' where c' is a computable canonical object and c' : C.■

5 Generalized propositional logic

Let U be a universe of propositional variables. Let Prop be the class of propositional sentences built up from U using the connectives $\bot,\wedge,\vee,\rightarrow$. As interpretations of the propositional variables we will consider partial inductive definitions over U. The semantics of Prop will iteslf be given in terms of a partial inductive definition over Prop:

$T(X),T(Y) \Rightarrow T(X\wedge Y)$

$T(X) \Rightarrow T(X\vee Y)$

$T(Y) \Rightarrow T(X\vee Y)$

$(T(X) \Rightarrow T(Y)) \Rightarrow T(X\rightarrow Y)$

So if P is an interpretation, then $P \models A$ iff $\models_{T,P} A$ for A in Prop.

This propositional logic is generalized in the sense that it covers a lot of different interpretations.

First of all it covers standard classical propositional logic:

consider the interpretations P consisting of clauses $\Rightarrow p$. Then it is easily to see that T,P is a total and complete definition thus:

$T,P \models p$ iff $\Rightarrow p$ is in P

$T,P \models A\wedge B$ iff $T,P \models A$ and $T,P \models B$

$T,P \models A\vee B$ iff $T,P \models A$ or $T,P \models B$

$T,P \models A\rightarrow B$ iff if $T,P \models A$, then $T,P \models B$.

If P consists of all clauses $p \Rightarrow p$, then T,P gives the standard interpretation of logical consequence in intuitionistic propositional logic.

Consider all interpretations P where we have clauses $\Rightarrow p$ and $p \Rightarrow p$ such that P does not contain two clauses with the same conclusion. The we will have a certain three valued logic where $p \Rightarrow p$ means that the truthvalue of p is not known. (See [H2] for a discussion of this type of interpretations in connection with non monotonic reasoning).

And of course we will have a lot of interpretations that will give a non standard interpretation of implication.

It is easy to see how to also consider predicate logic in this manner. Partial inductive definitions will then define predicates over a given Herbrand universe.

6 Two examples

When we think of the semantics of a formal system as given by a partial inductive definition we think of the semantics in iteslf as something elementary given. The true complexity of the semantics enter into the picture when we try to isolate the total objects of the definition. A canonical example that illustrates this situation is of course the syntax and semantics of naive set theory:

$\mathrm{Prop}(\bot)$

$(\mathrm{Prop}(A),\mathrm{Prop}(B)) \Rightarrow \mathrm{Prop}(A\wedge B)$

$(\mathrm{Prop}(A),\mathrm{Prop}(B)) \Rightarrow \mathrm{Prop}(A\vee B)$

$(\mathrm{Prop}(A),\mathrm{Prop}(B)) \Rightarrow \mathrm{Prop}(A\rightarrow B)$

$\{\mathrm{Set}(a) \Rightarrow \mathrm{Prop}(A(a)) \mid a \in U\} \Rightarrow \mathrm{Prop}(\forall x A(x))$

$\{\mathrm{Set}(a) \Rightarrow \mathrm{Prop}(A(a)) \mid a \in U\} \Rightarrow \mathrm{Prop}(\exists x A(x))$

$(\mathrm{Set}(a),\mathrm{Set}(b)) \Rightarrow \mathrm{Prop}(a=b)$

$(\mathrm{Set}(a),\mathrm{Set}(b)) \Rightarrow \mathrm{Prop}(a\in b)$

$\{\mathrm{Set}(a) \Rightarrow \mathrm{Prop}(A(a)) \mid a \in U\} \Rightarrow \mathrm{Set}(\{x \mid A(x)\})$

$(\mathrm{True}(A),\mathrm{True}(B)) \Rightarrow \mathrm{True}(A\wedge B)$

$\mathrm{True}(A) \Rightarrow \mathrm{True}(A\vee B)$

$\mathrm{True}(B) \Rightarrow \mathrm{True}(A\vee B)$

$(\mathrm{True}(A) \Rightarrow \mathrm{True}(B)) \Rightarrow \mathrm{True}(A\rightarrow B)$

$\{\mathrm{Set}(a) \Rightarrow \mathrm{True}(A(a)) \mid a \in U\} \Rightarrow \mathrm{True}(\forall x A(x))$

$(\mathrm{Set}(a),\mathrm{True}(A(a))) \Rightarrow \mathrm{True}(\exists x A(x)) \quad (a \in U)$

$\{\mathrm{Set}(a) \Rightarrow (\mathrm{True}(a\in b) \Rightarrow \mathrm{True}(a\in c),\mathrm{True}(a\in c) \Rightarrow \mathrm{True}(a\in b)) \mid a \in U\} \Rightarrow \mathrm{True}(b=c)$

$(\mathrm{Set}(a),\mathrm{True}(A(a))) \Rightarrow \mathrm{True}(a\in \{x \mid A(x)\})$

Where U is a large enough universe of expressions.

The usual distinction between sets and classes will here be embedded in the distinction between total and partial objects. What we get here is a view of set theory as a language which is very different from set theory as an axiomatic description of the universe of sets.

The syntax and semantics of Peano arithmetic can be given as a partial inductive definition in the following manner:

$(\mathrm{True}(A),\mathrm{True}(B)) \Rightarrow \mathrm{True}(A\wedge B)$

$\mathrm{True}(A) \Rightarrow \mathrm{True}(A\vee B)$

$\mathrm{True}(B) \Rightarrow \mathrm{True}(A \vee B)$

$(\mathrm{True}(A) \Rightarrow \mathrm{True}(B)) \Rightarrow \mathrm{True}(A \rightarrow B)$

$\Rightarrow \mathrm{True}(N(O))$

$\mathrm{True}(N(a)) \Rightarrow \mathrm{True}(N(s(a))) \quad (a \in U)$

$(\mathrm{True}(N(a)), \mathrm{True}(A(a))) \Rightarrow \mathrm{True}(\exists x A(x)))$

$(\mathrm{True}(A(0)), \{\mathrm{True}(N(a)) \Rightarrow (\mathrm{True}(A(a)) \Rightarrow \mathrm{True}(A(s(a)))) \mid a \in U\}) \Rightarrow \mathrm{True}(\forall x A(x))$

$\mathrm{True}(a=a) \quad (a \in U)$

$\mathrm{True}(a=b) \Rightarrow \mathrm{True}(s(a)=s(b))$

Note that $\mathrm{True}(0=s(0)) \Rightarrow \perp$ will hold. This follows by an application of the $P\vdash$ rule.

This definition is then a total definition.

References:

[A] Aczel P. An introduction to inductive definitions, in:Handbook of mathematical logic, ed.J. Barwise(North Holland, Amsterdam 1977)

[H1] Hallnäs L. On normalization of proofs in set theory (to appear in Dissertationes Mathematicae)

[H2] Hallnäs L. A note on non monotonic reasoning, in: Proceedings of the 1987 workshop on the frame problem (Morgan Kaufmann Publishers, Inc, Los Altos 1987)

[M1] Martin-Löf P. Haupsatz for the intuitionstic theory of iterated inductive definitions, in: Proceedings of the second scandinavian logic symposium ,ed. J.E. Fenstad (North Holland, Amsterdam 1971)

[M2] Martin-Löf P. Constructive mathematics and computer programming, in: Logic,Methodology and Philosophy of Science VI, ed. L.J. Cohen, J.Los, H. Pfeiffer and K-P.Podewski (North Holland, Amsterdam 1982)

[M3] Martin-Löf P. On the meanings of the logical constants and the justifications of the logical laws (Preprint Dept. of mathematics, University of Stockholm 1984)

[P1] Prawitz D. Ideas and results in proof theory,in: Proceedings of the second scandinavian logic symposium, ed J.E. Fenstad (North Holland, Amsterdam 1971)

[P2] Prawitz D. Towards a foundation of a general proof theory, in: Logic.Methodology and the Philosophy of Science IV, ed. P. Suppes(North Holland, Amsterdam 1973)

[P3] Prawitz D. On the idea of a general proof theory, Synthese 27, 1974

[P4] Prawitz D. Remarks on some approaches to the concept of logical consequence, Synthese 62, 1985

[S1] Schroeder-Heister P. Untersuchungen zur regellogischen deutung von aussagenverknupfungen, Dissertation, Bonn 1981

[S2] Schroeder-Heister P. A natural extension of natural deduction, Journal of symbolic logic Vol 50 number 1, 1985

[S3] Schroeder-Heister P. Judgements of higher level and standa dized rules for logical constants in Martin-Löf's theory of logic, manuscript, 1985

[T] Tait W. Intensional interpretation of functionals of finite type, Journal of symbolic logic 32,1967

① **Internalizing S4 and S5 in the LF** (December 86)

0. Preliminaries

(I) The Hilbert-type formulation of S4

Axioms

S4.1. $A \supset B \supset A$

S4.2. $A \supset (B \supset C) \supset (A \supset B) \supset (A \supset C)$

S4.3. $(\sim A \supset \sim B) \supset (B \supset A)$

S4.4. $\Box A \supset A$

S4.5. $\Box (A \supset B) \supset (\Box A \supset \Box B)$

S4.6. $\Box A \supset \Box\Box A$

Rules

$$\frac{A \quad A \to B}{B}$$

$$\frac{A}{\Box A}$$ (Necessitation)

II Corresponding consequence relations

(i) The internal $\vdash_I$: Take in the system above mecessitation to be only a rule of proof (i.e. if $\vdash_I A$ then $\vdash_I \Box A$ but $A \nvdash_I \Box A$)

semantical interpretation: $A_1, \dots, A_n \vdash_I B$ iff in every S4-frame and in every world in this frame in which $A_1, \dots, A_n$ are all true - so is B.

(ii) The external $\vdash_E$: This is the pure consequence relation defined by the system above (all rules are taken as rules for derivability)

semantical interpretation: $A_1, \dots, A_n \vdash_E B$ iff B is valid in every S4-frame in which $A_1, \dots, A_n$ are all valid (valid = true in all worlds of the frame).

① III <u>Deduction theorems</u>

<u>①</u> $\Gamma, A \vdash_{I} B$ iff $\Gamma \vdash_{I} A \supset B$

<u>②</u> a) $\Gamma, A \vdash_{E} B$ iff $\Gamma \vdash_{E} \Box A. \supset B$

b) $\Gamma, \Box A \vdash_{E} B$ iff $\Gamma \vdash_{E} \Box A. \supset B$

2b) can be generalized as follows: Define a formula to be <u>essentially modal</u> by:

(i) $\Box A$ is essentially modal

(ii) If A and B are essentially modal, so are $\sim A$, $A \vee B$, $A \wedge B$, $A \supset B$.

we have then:

② c): If A is essentially modal then
$\Gamma, A \vdash_{E} B$ iff $\Gamma \vdash_{E} A \supset B$

<u>Note:</u> 2a) implies 2b) because of axiom S4.6 while 2b) implies 2a) because of axiom S4.4. 2b) and 2c) are also easily seen to be equivalent.

<u>IV</u> <u>Prawitz Natural Deduction formulation of S4 (PNDS4)</u>:

(i) <u>version 1</u>: All the usual rules for the truth-functional connectives and in addition:

□-introduction $\dfrac{A}{\Box A}$, provided all assumptions on which A depends are boxed.

□-

□-elimination: $\dfrac{\Box A}{A}$

(ii) <u>version 2</u>: Like version 1, except □int, which is allowed whenever all assumptions on which A depends are <u>essentially modal</u>.

1 · Internalizing the Hilbert-type system

The idea is rather simple: We employ two different judgments, corresponding to the two consequence relations defined above. "Valid" corresponds to $\vdash_E$, "true" to $\vdash_I$. It is important to note that although we don't need the "true" judgement in order to internalize $\vdash_E$ we do need "valid" in order to overcome the difficulties connected with the impurity of $\vdash_I$!

The resulting signature is:

$$\mathrm{true} : \underset{A:o}{\pi}\ \mathrm{type}$$

$$\mathrm{valid} : \underset{A:o}{\pi}\ \mathrm{type}$$

$$c : \underset{A:o}{\pi}\ \mathrm{valid}\ A \to \mathrm{true}\ A$$

$$Ax_1 : \underset{A:o}{\pi}\ \underset{B:o}{\pi}\ \mathrm{valid}\ A \supset (B \supset A)$$

$(Ax_2, \ldots, Ax_6$ are similarly defined$)$

$$MP_t : \underset{A:o}{\pi}\ \underset{B:o}{\pi}\ \mathrm{true}\,(A \supset B) \to (\mathrm{true}\ A \to \mathrm{true}\ B)$$

$$MP_v : \underset{A:o}{\pi}\ \underset{B:o}{\pi}\ \mathrm{valid}\,(A \supset B) \to (\mathrm{valid}\ A \to (\mathrm{valid}\ B))$$

$$Nec : \underset{A:o}{\pi}\ \mathrm{valid}\ A \to \mathrm{valid}\ \Box A.$$

It is easy(?) then to see (using syntactic and/or semantic considerations) that:

(i) $A_1, \ldots A_n \vdash_I B$ iff there exists a term t such that
$x_1 : \mathrm{true}\ A_1, \ldots\ x_n : \mathrm{true}\ A_n \vdash_{LF} c : \mathrm{true}\ B$

(ii) $A_1, \ldots A_n \vdash_E B$ iff there exist a term t s.t.
~~x_1: valid A_1, ... x_n valid~~

...... x_1: valid A_1, ..., x_n: valid $A_n \vdash_{LF}$ t: valid B

(... might contain assumptions of the form p:O we shall ignore such assumptions in what follows)

(iii) More generally, we have that there is a term t s.t:

x_1: valid A_1, ..., x_n: valid A_n, y_1: true B_1, ..., y_m: true $A_m \vdash t$: true C

iff in every S4-frame in which $A_1, ..., A_n$ are valid and in every world in this frame in which $B_1, ..., B_m$ are true C is true as well

2. The problems with PNDS4 and their solutions

A natural-deduction system always explicitely defines an obvious consequence relation, and the above PNDS4 is no exception. The question is now: to what C.R. the above system corresponds: to $\vdash_I$ ("truth") or $\vdash_E$ ("validity"). At first sight it seems that to neither. The reason is that the □-intr. rule (from A infer □A) is sound for $\vdash_E$, but not for $\vdash_I$, while the ⊃ int is sound for $\vdash_I$, but not for $\vdash_E$. A closer inspection reveals, however, that □-int is here not $\frac{A}{\Box A}$, but a "meta" rule in seqhents:

$$\frac{\Box A_1, ..., \Box A_n \vdash B}{\Box A_1, ..., \Box A_n \vdash \Box B}$$

. This metarule is sound for $\vdash_I$. ~~as well~~ (i.e: by interpreting $\vdash$ as $\vdash_I$ we get an admissible metarule) It follows easily, therefore, that there is a proof of B from the assumptions $A_1, ..., A_n$ in NDS4 iff $A_1, ..., A_n \vdash_I B$. FNDS4 is, accordingly, essentially a sequential calculus. In the LF we cannot, however, directly internalize seqhents, nor can we directly handle the impurity caused by the side condition on □-int. It is not useful, therefore, to interprete $A_1, .., A_n \vdash_{MOS4} B$ as meaning true $A_1, ...,$ true $A_n \vdash$ true B (despite its being the most obvious interpretation) since this caus unsolvable problems concerning □-int.

(5) Fortunately the $\vdash_I$ is not the only possible interpretation of PWDS4. In fact it is trivial that $A_1, \ldots A_n \vdash_I B$ entails that $A_1, \ldots, A_n \vdash_E B$. Hence whenever there is a proof of B from $A_1, \ldots A_n$ in PWDS4 ~~it~~ it is the case that valid $A_1, \ldots$ valid A_n entail valid B. The converse fails, though,* unless all the A_i's are boxed or, more generally, essentially modal (including the important case n=0!)' This failure is due to ⊃int, which is not sound ~~true~~ for $\vdash_E$. The above observations, as well as the deduction theorems for $\vdash_E$ and the formulations of the condition on □-int suggest limiting ⊃int to the case in which ~~the~~ the discharged assumption is boxed (or, for ~~to~~ the second ~~second~~ version, essentially modal). Under this limitation, e.g., the ~~above~~ proof below of ~~A5~~ S4.5 is translatable straightforwardly:

1) □(A⊃B) (Assumption - dischared in 8)
2) A⊃B (1, □elim)
3) □A (Assumption - discharged in 7)
4) A 3, □elim.
5) B 2, 4, M.P.
6) □B 5, □-int.
7) □A ⊃ □B (6, ⊃int)
8) □(A⊃B) ⊃ (□A ⊃ □B) (7, ⊃int)

Limiting ⊃int as suggested above causes new problems, though, since even valid (A⊃A) is not derivable any more. It is desirable, therefore, to allow unlimited use of ⊃int when only truth-functional inferences are involved. ~~For~~ This can be done by introducing a new judgement, taut, corresponding to provability using propositional calculus. Actually, also □-Elim is not a real obstacle, since

(*) A complete characterization is: $\Box A_1, \ldots \Box A_n, B_1 \ldots B_m \vdash_{PWDS4} C$ iff $\Box A_1, \ldots, \Box A_n \vdash_E B_1 \supset B_2 \supset \ldots \supset B_m \supset C$. More generally: if $A_1 \ldots A_n$ are all essentially modal then $A_1 \ldots A_n, B_1 \ldots B_m \vdash_{PWDS4} C$ iff $A_1 \ldots A_n \vdash_E B_1 \supset B_2 \supset \ldots \supset B_m \supset E$

(c) $\Box A \supset A$ is valid. We may therefore extend "taut" to mean "provable using only the pure rules of NDS4". This leads to the following signature for internalizing PND S4:

LFS4

(1) valid : $o \to$ type

(2) taut : $o \to$ type

(3) V : $\prod_{A:o}$ taut $A \to$ valid A

(4) $\supset int_{ta}$: $\prod_{A:o} \prod_{B:o}$ (taut $A \to$ taut B) $\to$ (taut $A \supset B$)

(5) $\supset int_{va}$: $\prod_{A:o} \prod_{B:o}$ (valid $\Box A \to$ valid B) $\to$ valid ($\Box A \supset B$)

(6) $\supset Elim_{ta}$: $\prod_{A:o} \prod_{B:o}$ taut ($A \supset B$) $\to$ (taut $A \to$ taut B)

(7) $\supset Elim_{v}$ $\prod_{A:o} \prod_{B:o}$ valid ($A \supset B$) $\to$ (valid $A \to$ valid B)

(8) $\Box$-int : $\prod_{A:o}$ valid $A \to$ valid $\Box A$

(9) $\Box$elim : $\prod_{A:o}$ taut $\Box A \to$ taut A.

(4) - (9) constitute a formulation of the intuitionistic implication-necessitation fragment of the first version of pNDS4. To get the full classical system one needs just to add the usual rules for the other connectives (especially $\bot$) using taut.

To internalize the second version we should add another judgement, essen-modal, with the corresponding constants, and replace (5) by:

(5') $\supset Int'_{va}$: $\prod_{A:o} \prod_{B:o}$ essen-modal $(A) \to [($valid $A \to$ valid $B) \to$ valid $(A \supset B)$ ~~$A \supset B$~~

To see the soundness of these rules use the following interpretation.

taut A : There is a proof of A in the Hilbert-type system for S4 which uses only S4.1 - S4.4 and M.P. (call this system HS

valid A : $\vdash_{S4} A$.

(7) Important! The above "interpretations" are not absolute, but context-dependent. This is clear ~~w~~ from the observation that a formula of the form $\Box A$ can never be provable using M.P from S4.1 - S4.4 alone (since it is not difficult to show that ~~i~~ a formula has such a proof iff it is true for any assignment v of ~~boolean~~ boolean values to formulas satisfying: a) $v(A \supset B) = v(A) \supset v(B)$ b) $v(\bot) = f$ c) $v(\Box A) = f$ if $v(A) = f$). ~~It~~ It is, nevertheless, perfectly acceptable to assume x: taut($\Box A$) and to build then terms belonging ~~[illegible]~~ ~~belonging~~ to taut B where B is ~~another~~ formula (e.g. $\Box A$ itself) ~~And~~ But we surely don't want this to be possible for every B! (i.e we don't want taut $\Box A \rightarrow$ taut B to be always non-empty!). It will be more accurate to say, e.g. that taut A is a type the cannonical elements of which are proofs ~~[illegible]~~ of A (from certain assumptions) in MS4 (the Hilbert systemd ~~[illegible]~~ which has S4.1 - S4.4 as axioms and MP as the only rule of inference).

It is necessary then to go on and to explain what are the cannonical elements of taut $A \rightarrow$ taut B and so on. This can[*] be done in the ordinary tradition of Martin-Löf.[**]

It is (really!) easy to show completeness of the above ~~s~~ internalization. By this we mean, e.g., that

(1) If $A_1, \ldots A_n \vdash_E B$ then there is a term t s.t.: (assumptions for showing $A_i \in \sigma$ - - -), ~~x_1: taut A_1, x_2: taut A~~ x_1: valid A_1, ... x_n: valid $A_n \vdash t$: valid B

(2) If $A_1 \ldots A_n \vdash_{MS4}$ ~~[illegible]~~ B then there is a term t s.t (assumptions for showing $A_i \in \Theta$), x_1: taut A_1, ... , x_n: taut $A_n \vdash t$: taut B.

(*) ?

(**) Between us: the meaning of all these explanations and claims are far from being clear to the author! Please help him! The real troubles begin, however, when we try to give meaning to types like taut $A \rightarrow$ valid B, ~~as~~ ~~we~~ see below!

(*) Before discussing the relation between the above signature and the original PNDS4 there is one other question that might be asked: what about $\vdash_I$? Well this can ~~[illegible]~~ easily be internalized too by introducing a new judgement, "true". For achieving ~~[illegible]~~ soundness and completeness we need them just two extra constants:

$$t : \prod_{A:o} \text{valid}\ A \to \text{true}\ A$$

$$\supset Elim_{tr} : \prod_{A:o} \prod_{B:o} \text{true}(A \supset B) \to (\text{true}\ A \to \text{true}\ B)$$

It is natural to add also constants similar to those of tant (like $\supset int_{tr}$, $\Box elim_{tr}$ and so on), but those terms of the desired type are definable already in the resulting system.

(3) The relation between LFS4 and PNDS4.

Our problem in this section is: to what extent and in what sense is LFS4 really an internalization of PNDS4. Obviously, it is not an internalization of the Hilbert-type system, since no constant of LFS4 corresponds to any axiom, while the various $\supset$int constants and their corresponding higher-order judgements are typical for natural-deduction systems (Our ~~[illegible]~~ explanations of the meaning of the various judgements as well as the justification for introducing the various constants were in terms of the Hilbert-type system, but this is a general feature of N.D. systems: that they usually result from a formalization of the meta-theory of Hilbert-type system. The meaning of $A_1, \ldots, A_n \vdash_{PNDS4} B$ was also given in terms of $\vdash_I$ or $\vdash_E$!). However LFS4 might just corresponds to a new N.D. formulation of S4, different from that of Prawitz

(*) This possibility was suggested by Fabio.

In order to definitely settle this question we need a precise notion of "faithful internalization", which we do not have at the moment. Demanding, e.g. 1-1 correspondence between what can be proved in the internalzed system and its suggested internalization (or even between proofs in the first and proof-terms in the second) is too strong and too weak. It is too strong, since the LF, with its higher-order judgements is always capable of expressing and proving claims that are beyond the power of the internalized system. Moreover, LFS4, with its three different judgements surely internalizes more then just PNDS4. On the other hand the above suggestion is also too weak, since ~~st~~ for every proof of B from the assumption $A_1, \ldots A_n$ in PNDS4 we can uniformly ~~constant~~ construct (in 1-1 way) a term t of the signature for the Hilbert system (LFH) s.t.

$$x_1 : \text{true } A_1, \ldots, x_n : \text{true } A_n \vdash_{LFH} t : \text{true } B$$

I don't think, though, that it is desirable to take this signature as a faithful internalization of PNDS4. We feel that the term t should somehow really represent in a natural way the given proof in PNDS4, ~~But~~ It is difficult though to give a precise characterization of this intuition.

We shall leave the general problem of precisely defining "faithful internalization" to further investigations. For present purposes we shall be content with providing a procedure which given a PNDS4 proof(*) of B from $A_1, \ldots, A_n$ returns a term t of LFS4 satisfying

$$\boxed{x_1 : \text{valid } A_1, \ldots, x_n : \text{valid } A_n \vdash_{LFS4} t : \text{valid } B}$$

So that t is actually constructed from the given proof, intuitively reflects it and naturally represents it. We believe that this construction establishes an internalization of PNDS4 within LFS4.

The construction of t is done in three stages:

(*) We give a construction for the first version. A very similar construction is applicable to the second one.

(10) <u>Stage 1</u> : This stage reflects the "taut" inferences of the given IND S4 proof. We construct in it a term t'_C for every formula C occurring in the given proof s.t:

(*) $\ldots x_i : \text{taut } D_i \ldots \quad y_j : \text{taut } \Box E_{j,1} \to \text{taut } \Box E_{j,2} \to \ldots \to \text{taut } \Box E_{j,k_j} \to \text{taut } \Box F_j \ldots \vdash_{\lambda S4} t'_C : \text{taut } C$
$(1 \leq i \leq m) \qquad (1 \leq j \leq p)$

where $D_1, \ldots, D_m$ are all the assumptions on which C depends in the given proof, and for $1 \leq j \leq p$ there is above C an inference of the form: $\frac{F_j}{\Box F_j}$ and $\Box E_{j,\ell}$ $(1 \leq \ell \leq k_j)$ are the assumptions on which F_j depends.

Note The need to use term of type $\text{taut } \Box E_{j,1} \to \ldots \to \text{taut } \Box F_j$ reflects the fact that in $\Box$-int the whole segment involved should be known before applying ~~this~~ the rule.

The construction of t'_C is by induction on the length of the proof of C. There are two basic cases: (i) C is a hypothesis. In this case $t'_C = x$. (ii) $C = \Box F$ and was inferred from F. Let then $\Box D_1, \ldots, \Box D_n$ be all the assumptions on which F (and C) depend. We have:

$$x_1 \in \text{taut } \Box D_1, \ldots, x_n \in \text{taut } \Box D_n, y \in (\text{taut } \Box D_1 \to \text{taut } \Box D_2 \to \ldots \to \text{taut } \Box D_n \to \text{taut } \Box F) \vdash_{\lambda S4} y x_1 x_2 \ldots x_n \in \text{taut}$$

Hence we can take t'_C to be $y x_1 x_2 \ldots x_n$ in this case.

The inductive construction of t'_C in other cases is obvious. For example, suppose $C = A' \supset B'$ was inferred by $\supset$-int. We can take t'_C then to be $\supset\text{-int}_{to}(\lambda x : \text{taut } A' . t'_{B'})$, where $t'_{B'}$ is given by the induction hypothesis. Other cases are even simpler.

<u>Stage 2</u> In this stage we construct from t'_C (for each C) a term t''_C such that

(**) $\ldots x_i : \text{valid } D_i \ldots y_j : \text{valid } \Box E_{j,1} \to \text{valid } \Box E_{j,2} \to \ldots \to \text{valid } \Box E_{j,k_j} \to \text{valid } \Box F_j \ldots \vdash t''_C : \text{va}$

where D_i, $E_{j,\ell}$ are like in (*)

(11) The construction of t_C^* is done as follows:

(i) Using $\supset int_{to}$ and $\supset Elim_{to}$ we construct from t_C'' a term t_C^1 s.t:

$$\vdash_{LFS4} t_C^1 : \text{tont}\ \underbrace{(D_1 \supset D_2 \supset \ldots \supset D_n \supset \ldots \supset (\Box E_{j,1} \supset \ldots \supset \Box E_{j,k_j} \supset \Box F_j) \supset \ldots \supset C}_{T}.$$

(ii) Let $t_C^2 = V(t_C^1)$(*). then $\vdash t_C^2 : \text{valid}\ T$

(T = the formula s.t $t_C^1 : \text{tont}\ T$)

(iii) By ~~rep~~ repeated applications of $\supset Elim_{va}$ we get from t_C^2 a term t_C^3 s.t:

$$\vdash_{LFS4} t_C^3 : \text{valid}\ D_1 \rightarrow \ldots \text{valid}\ D_n \rightarrow \ldots \rightarrow \text{valid}\ (\Box E_{j,1} \supset \ldots \supset \Box E_{j,k_j} \supset \Box F_j) \rightarrow \ldots \rightarrow \text{valid}\ C$$

(iv) Using $\supset int_{va}$ enough times we get, for each j a term t_j s.t:

$$\vdash_{LFS4} t_j : (\text{valid}\ \Box E_{j,1} \rightarrow \text{valid}\ \Box E_{j,2} \ldots \rightarrow \text{valid}\ \Box E_{j,k_j} \rightarrow \text{valid}\ \Box F_j) \rightarrow \text{valid}\ (\Box E_{j,1} \supset \ldots \supset \Box F_j).$$

(v) From t_C^2 and t_j we easily get, finally, a term t_C^* as in (**)

Stage 3. The term t_B'' obtained in stage 2 differed from the desired by having extra free variables corresponding to "assumptions" of the form $\Box E_{j,1} \rightarrow \ldots \rightarrow \Box E_{j,k_j} \rightarrow \Box F_j$ s.t the given proof of B from $A_1, \ldots, A_n$ contains a proof of F_j from ~~$\Box E_{j,1} \rightarrow \ldots \Box E_{j,k_j} \rightarrow E$~~ $E_{j,1} \ldots E_{j,k_j}$ - This sub-proof is a proper subproof of the given one, and so we may assume that our procedure provides us with a term $t_F : \text{valid}\ \Box E_{j,1} \rightarrow \text{valid}\ \Box E_{j,2} \rightarrow \ldots \rightarrow \text{valid}\ \Box E_{j,k_j} \rightarrow \text{valid}\ F$.) $\Box int \otimes t_F$ can then be substitute for the extra free variables to provide us, at last, the desired t. (Note that t_F should not be constructed from scratch but from t_F'' using stage 3 also!)

To summerize The internalization of a given INDS4-proof was done in three stages. Stage 1 treat all application of the

(*) See definition of V in p. C

phre rules of PROS4 (using "taut"). Stage 3 – the applications of the impure □-int rule (using "valid") while stage 2 connect the two stages by passing from taut to valid.

Note As mentioned above, the main thing done in stage 2 is the global passage (in sequents of the form (x)) from taut to valid. Now this passage consists of two main parts: One is the passage from assumptions of the form $x{:}\ \text{valid}(\Box A_1 \supset \ldots \Box A_n \supset \Box B)$ to $x{:}\ \text{valid}\,\Box A_1 \to \text{valid}\,\Box A_2 \ldots \to \text{valid}\,\Box A_n \overset{\text{valid }\Box B}{\supset \Box B}$ And this passage is handle by our □-int $_{\text{ru}}$. The other part is the passage from $\text{taut}\,A_1 \to \ldots \to \text{taut}\,A_n$ to $\text{valid}\,A_1 \to \text{valid}\,A_2 \to \ldots \to \text{valid}\,A_n$. This passage is done uniformly and should be taken as a basic tactic for LFS4. However, despite the obvious uniformity we need a separate tactic for each n. For example, suppose we apply to ~~valid A → valid B → valid C the tactic~~ taut A → (taut B → taut C) just the tactic leading from taut B → taut C to valid B → valid C. We get then taut A → (valid B → valid C) and so also valid B → (taut A → valid C). Here we are forced to stop, unless we have a tactic leading from (taut A → valid C) to valid A → valid C. Unfortunately no corresponding term seems to exists in LFS4 (although for each n there is a term $t_n : (\text{taut}\,A_1 \to \ldots \to \text{taut}\,A_n) \to (\text{valid}\,A_1 \to \ldots \to \text{valid}\,A_n)$ One might consider adding a new constant with this property. The intuitive gustification might be that cannonicals terms of type taut A → valid B as well as valid A → valid B) should correspond to a proofs of B from A according to $\vdash_E$, and ~~how A itself is obtained~~ the question what exact fragment of S4 is used to obtain A itself is irrelevant. However the intuitions concerning the intuitive meaning of hybrids like taut A → valid C are not clear at the moment, and so one cannot feel sure that by adding corresponding constants one ~~we~~ still have S4, and not something else (i.e: that the extension is conservative in the obvious sense). We consider this problem to be a very ~~[illegible]~~ important area for further investigations.

4 Internalizing S5

S5 is obtained from S4 by adding the axiom

$$\sim\Box A \supset \Box\sim\Box A.$$

The N.D system of Prawitz for S5 is similar to that for S4 (second version) and the only difference is in the definition of "essentially-modal". For S5 we have:

(i) $\Box A$ is essentially modal

(ii) If A and B are essentially-modal, so are $\sim A$, $A \vee B$, $A \wedge B$ and $A \supset B$.

According to this, it is obvious that the internalization of PRDS5 will be exactly as that of PNDS4 (second version) and the only differences will involve the set of constants corresponding to the "essent modal" judgement.

AN INTRODUCTION TO SET NOTATIONS

J.R. Abrial

17

TABLE OF CONTENTS

1. BASIC SET CONSTRUCTS 1
    1.1. INDEXATION 1
    1.2. CARTESIAN PRODUCT 2
    1.3. POWER SET 2
    1.4. SET MEMBERSHIP 2
    1.5. SET INCLUSION 2
    1.6. SET EQUALITY 3
2. DERIVED CONSTRUCTS 3
    2.1. COMPREHENSION 3
    2.2. EXTENSION 3
    2.3. UNION AND INTERSECTION 3
    2.4. COMPLEMENT 4
3. BINARY RELATIONS 5
    3.1. DEFINITION 5
    3.2. DOMAIN AND CO-DOMAIN 5
    3.3. COMPOSITION 5
    3.4. EMPTY AND IDENTITY RELATIONS 6
    3.5. RESTRICTION AND CO-RESTRICTION 6
    3.6. CONVERSE 7
    3.7. IMAGE 7
    3.8. UNION AND INTERSECTION 8
    3.9. OVERRIDING 8
4. FUNCTIONS 9
    4.1. PARTIALITY VS TOTALITY 9
    4.2. EVALUATION 9
    4.3. CONSTRUCTION 10
    4.4. INJECTIVE FUNCTIONS 10
5. CONSTRUCTING MATHEMATICAL OBJECTS 10
6. FINITE SETS 13
7. INFINITE SETS 14
8. NATURAL NUMBERS 14
    8.1. DEFINITION 14
    8.2. RECURSION ON NATURAL NUMBERS 16
    8.3. ARITHMETIC 17
9. SEQUENCES 18
10. TREES 20
    10.1. BINARY TREES 20
    10.2. LABELED BINARY TREES 21
    10.3. N-ARY TREES 22
    10.4. LABELED N-ARY TREES 23

REFERENCES

Appendix 1: SUMMARY OF NOTATIONS
Appendix 2: SUMMARY OF AXIOMS
Appendix 3: SUMMARY OF DEFINITIONS

AN INTRODUCTION TO SET NOTATIONS

In this note, we introduce various elementary set theoretical notations

1. BASIC SET CONSTRUCTS

We may denote a set by a symbol. Examples of these are

{}	The EMPTY SET (with no member)
NAT	The set of natural numbers

Note that we shall later construct NAT (see section 8) so that the only set which is so far (God) given is the empty set {}.

We may also denote sets by means of various notations intended to capture the idea of constructing new sets from already known ones. There are three such constructs called INDEXATION, CARTESIAN PRODUCT, and POWER SET.

N.B: All notations used in this presentation are summarized in Appendix 1.

1.1. INDEXATION

Constructing a set by indexation consists in giving the general 'form' of its members. The form in question is supposed to be an expression indexed by a variable ranging over all members of another set. For instance, the following set is the set of perfect squares of natural numbers; that is, the set whose members have the general form 'n*n' for all natural numbers n

```
$n.(n:NAT | n*n)
```

More generally, given a variable x, a set s, a predicate P and an expression E, the set whose members have the 'form' E for x in s such that P is denoted by

```
$x.(x:s & P | E)
```

When P is TRUE, we simplify the construct as follows

```
$x.(x:s | E) =  $x.(x:s & TRUE | E)            DEFINITION
```

Note that we have not yet axiomatise the (membership) predicate "x:s". This will be done in section 1.4.

N.B: All definitions are summarized in Appendix 3.

1.2. CARTESIAN PRODUCT

The cartesian product of two sets s and t is the set whose members are all ORDERED PAIRS of members of s and t; it is denoted by 's*t'. As a notation for the pairing operator, we shall use either ',' or '|->'; more precisely 'x,y' and 'x|->y' are alternate notations for the pair made of x and y in that order. Note that pair equality is axiomatised DIRECTLY as follows

```
(a,b)=(c,d)  =  (a=c) & (b=d)                    AXIOM
```

N.B: All axioms are summarized in Appendix 2

1.3. POWER SET

The power set of a set s is the set whose members are all sets INCLUDED in s (see section 1.5); it is denoted by "POW(s)".

1.4. SET MEMBERSHIP

As we have implicitely admitted in previous sections, a SET IS CHARACTERIZED BY ITS MEMBERS. Set membership is formally defined by means of the predicate "x:s" which can be read 'x is a member of s'. Membership is defined recursively for the three previous constructs (as well as for the empty set) using Predicate Calculus as follows

```
y:$x.(x:s & P | E)  =  #x.(x:s & P & y=E)    AXIOM

(x,y):s*t           =  x:s & y:t              AXIOM

s:POW(t)            =  !x.(x:s => x:t)        AXIOM

not #x.(x:{})                                 AXIOM
```

1.5. SET INCLUSION

Membership of a power set is called set inclusion, formally

```
s inc t  =  s:POW(t)                          DEFINITION
```

Set inclusion is a PRE-order since it is obviously refexive and transitive, formally

```
s inc s
s inc t  &  t inc u  =>  s inc u
```

1.6. SET EQUALITY

Since a set is characterized by its members, we have

```
!x.(x:s = x:t) => (s=t)                    AXIOM
```

This leads to tne following which is easier to use in practice and makes set inclusion a PARTIAL order

```
t inc s  &  s inc t  =>  (s=t)
```

2. DERIVED CONSTRUCTS

We define now a collection of derived constructs; namely, set comprehension, set extension, union, intersection and complementation.

2.1. COMPREHENSION

A set is defined 'in comprehension' when its members are exactly the members of a given set such that a certain predicate holds. Set comprehension can obviously be defined as a special case of indexation: the 'form' of the members is just the indexing variable itself

```
{x | x:s & P} = $x.(x:s & P | x)           DEFINITION
```

2.2. EXTENSION

A set can also be defined 'in extension' by an explicit enumeration of its members supposed to be already members of a given set. Such an enumeration is a special case of comorehension. For instance, if a and b are members of s, then we have

```
{a}   = {x | x:s & (x=a)}                  DEFINITION

{a,b} = {x | x:s & (x=a or x=b)}           DEFINITION
```

2.3. UNION AND INTERSECTION

Given a set u which happens to be a member of POW(POW(s)) for some set s, the (generalized intersection) of u, denoted by "inter(u)" (only defined if u is not empty), is the subset of s whose members are members of all members of u, formally

```
not(u={})  =>  inter(u) = {x | x:s & !y.(y:u => x:y)}     DEFINITION
```

Example

```
      inter({{a,b,c,d},{b,c,e},{d,b,c,e}}) = {b,c}
```

Likewise, the (generalized) union of u, denoted by "union(u)", is the subset of s whose members are members of at least one member of u, formally

```
      union(u) = {x | x:s & #y.(y:u & x:y)}               DEFINITION
```

Example

```
      inter(({{a,b,c,d},{b,c,e},{d,b,c,e}}) = {a,b,c,d,e}
```

Generalized intersections (res. unions) are greatest lower bounds (resp. least upper bounds). More precisely, the intersection (resp. union) of the set (of sets) s is the greatest (resp. least) set which is smaller (resp. greater) than all its members; formally (if s is not empty in the case of intersection)

```
   t:s  =>  inter(s) inc t                        it is a lower bound
   !t.(t:s =>  u inc t)  =>  u inc inter(s))      it is the greatest of them

   t:s  =>  t inc union(s)                        it is an upper bound
   !t.(t:s => t inc u)  =>  union(s) inc u)       it is the least of them
```

The classical operations of (small) union and (small) intersection of sets are now defined as special cases of the corresponding generalized operations. More precisely, given two subsets a and b of s, we have

```
      a\/b = union({a,b})                               DEFINITION

      a/\b = inter({a,b})                               DEFINITION
```

Note that this definition allows us to define set in extension when there are more than two elements, formally

```
      {x,y,z} = {x,y}\/{z}
```

2.4. COMPLEMENT

Given a subset t of s, the complement of t with respect to s is the subset of s whose members are not members of t, formally

```
     s-t = {x | x:s & not(x:t)}                           DEFINITION
```

Note that if s-t is empty then s is included in t, therefore equal to it since t is, by definition, already included in s, formally

```
     s-t = {}  =  not #x.x:s-t
               =  not #x.(x:s & not(x:t))
               =  !x.(x:s => x:t)
               =  s inc t
```

3. BINARY RELATIONS

3.1. DEFINITION

A binary relation r with SOURCE s and DESTINATION t is a subset of the cartesian product of s and t. The set of all such relations from s to t is denoted by s<->t, formally

```
     s<->t = POW(s*t)                                     DEFINITION
```

Example

```
     {2|->a, 3|->d, 4|->b, 4|->d} : {1,2,3,4}<->{a,b,c,d}
```

3.2. DOMAIN AND CO-DOMAIN

Given a relation r of s<->t, the domain of r is the subset of s whose members are related to at least one element (of t). Similarly, the co-domain of r is the subset of t whose members are related to at least one element (of s). Formally

```
     dom(r) = {x | x:s & #y.(x,y):r}                      DEFINITION

     cod(r) = {y | y:t & #x.(x,y):r}                      DEFINITION
```

Example

```
     dom({2|->a, 3|->d, 4|->b, 4|->d}) = {2,3,4}
     cod({2|->a, 3|->d, 4|->b, 4|->d}) = {a,b,d}
```

3.3. COMPOSITION

Given three sets u, v, and w and two relations r and s from u to v and v to w respectively, the composition "r;s" of r and s is a relation with source u and destination w such that

```
     r;s = {x,z | x,z:u*w & #y.((x,y):r & (y,z):s)}          DEFINITION
```

Example

```
    {2|->a, 3|->d, 4|->b, 4|->d};{a|->0, a|->5, c|->1, d|->3}
  = {2|->0, 2|->5, 3|->3, 4|->3}
```

Composition is associative

```
    (r;s);t = r;(s;t)
```

3.4. EMPTY AND IDENTITY RELATIONS

Two special relations play an important role, the empty relation (which is nothing else than the empty set) and the identity relation built on a certain set. Here is the definition of the second one

```
     identity(s) = $x.(x:s | x,x)                            DEFINITION
```

For example, we have

```
     identity({a,b,c}) = {a|->a, b|->b, c|->c}
```

Here are some properties

```
     r;{} = {}
     {};r = {}

     r;identity(cod(r)) = r
     identity(dom(r));r = r

     dom(r;s) = dom(r;identity(dom(s)))
     cod(r;s) = cod(identity(cod(r));s)

     identity(s);identity(t)  =identity(s/\t)
```

3.5. RESTRICTION AND CO-RESTRICTION

Given a relation r of s<->t, a subset u of s, and a subset v of t, the restriction of r to u is defined to be

```
     identity(u);r
```

and the co-restriction of r to v is

```
     r;identity(v)
```

Example

```
     identity({1,2,3});{2|->a, 3|->d, 4|->b, 4|->d} = {2|->a, 3|->d}

     {2|->a, 3|->d, 4|->b, 4|->d};identity({b,e,f}) = {4|->b}
```

3.6. CONVERSE

The converse r~ of a relation r of s<->t is defined as follows

```
     r~ =  {y,x | (y,x):t*s & (x,y):r}                        DEFINITION
```

Here are various prooerties of the converse

```
     (r;s)~   = s~;r~
     r~~      = r
     identity(u)~  = identity(u)
     {}~      = {}
     dom(r~)  = cod(r)
     cod(r~)  = dom(r)
```

3.7. IMAGE

Given a relation r of s<->t, and a subset u of s, the image of u under r is the subset of t whose members are related to at least one member of u, formally

```
     image(r)(u) = {y | y:t & #x.(x:u & (x,y):r)}         DEFINITION
```

Examole

```
     image({2|->a, 3|->d, 4|->b, 4|-> d})({2,4}) = {a,b,d}
```

Here are some properties of images

```
image(r)(u)              = cod(identity(u);r)
image(s)(image(r)(u))  = image(r;s)(u)
image(r)(u\/v)          = image(r)(u)\/image(r)(v)

cod(r) = image(r)(s)
dom(r) = image(r~)(t)
```

3.8. UNION AND INTERSECTION

Relations being sets, it is possible to union and intersect them as in

```
   {2|->a, 3|->d, 4|->b, 4|->d} \/ {1|->f, 2|->a, 3|->c}
 = {2|->a, 3|->d, 4|->b, 4|->d, 1|->f, 3|->c}

   {2|->a, 3|->d, 4|->b, 4|->d} /\ {1|->f, 2|->a, 3|->c}
 = {2|->a}
```

Here are some properties of the union and intersection of relations

```
dom(r\/s)      = dom(r)\/dom(s)
r;(t\/s)       = (r;t)\/(r;s)
(r\/s);t       = (r;t)\/(s;t)
(r\/s)~        = r~\/s~
(r/\s)~        = r~/\t~
identity(u)\/identity(v) = identity(u\/v)
identity(u)/\identity(v) = identity(u/\v)
```

3.9. OVERRIDING

The overriding of relation r by relation s (both relations being members of u<->v) is denoted by "r<+s"; it is also a relation from u to v obtained by removing from r those pairs whose first elements are members of the domain of s and then by making the union of the resulting relation with s, formally

```
r<+s = (identity(u-dom(s));r)\/s                    DEFINITION
```

Example

```
   {2|->a, 3|->d, 4|->b, 4|->d} <+ {2|->b, 3|->e, 1|->5}
 = {2|->b, 3|->e, 4|->b, 4|->d, 1|->5}
```

Overriding is associative

```
      (r<+s) <+ t = r <+ (s<+t)
```

4. FUNCTIONS

4.1. PARTIALITY VS TOTALITY

A partial function with source s and destination t is a relation of s<->t such that no two distinct members of t are related to a single member of s. The set of such functions is denoted by "s+->t". Here is its definition

```
      s+->t = {r | r:s<->t & (r~;r):identity(cod(r))}        DEFINITION
```

A total function from s to t is a partial function from s to t whose domain is equal to s. The set of such functions is denoted by "s-->t", formally

```
      s-->t = {f | f:s+->t & dom(f)=s}                        DEFINITION
```

Properties

```
      f:s+->t & g:t+->u => (f;g):s+->u
      f:s-->t & g:t-->u => (f;g):s-->u

      f:s+->t & g:s+->t => (f+g):s+->t

      f:s+->t & g:s+->t & dom(f)/\dom(g)={} => f\/g:s+->t
```

4.2. EVALUATION

Given a function f in s+->t, and a member x of dom(f), the value of f at x is denoted by f(x). It is defined indirectly by the following axiom

```
      f:s+->t & x:dom(f)  =>  x,f(x):f                        AXIOM
```

Properties

```
      f:s+->t & x:dom(f) => f(x):cod(f)

      f:s+->t & g:t+->u & x:dom(f;g) => (f;g)(x)=g(f(x))
```

4.3. CONSTRUCTION

Given a set s and an expression E with free variable x, the set of pairs of the form (x,E) for all x in s is a function denoted by %x.(x:s | E), formally

```
%x.(x:s | E) = $x.(x:s | x,E)                          DEFINITION
```

Example (the square function)

```
%x.(x:NAT | x*x) = {0|->0, 1|->1, 2|->4, 3|->9,...}
```

The following property relates function evaluation to substitution

```
!x.(x:s => E:t)         => (%x.(x:s | E):s-->t
!x.(x:s => E:t) & a:s => (%x.(x:s | E)(a)=[x:=a]E
```

Example

```
(%x.(x:NAT | x*x)(3) = [x:=3](x*x) = 3*3 = 9
```

4.4. INJECTIVE FUNCTIONS

An injective function is a function whose converse is also a function. We consider partial and total injective functions from s to t denoted respectively by s*+>t and s*->t, formally

```
s*+>t = {f | f:s+->t & f~:t+->s}                    DEFINITION

s*->t = s*+>t /\ s-->t                              DEFINITION
```

Properties

```
f:s*+>t & g:t*+>u => (f;g):s*+>t
f:s*+>t & g:t*+>u => (f+g):s*+>t
```

5. CONSTRUCTING MATHEMATICAL OBJECTS

In Mathematics, objects obeying closed 'definitions' are frequently encountered. For instance

> A finite set is either the empty set or the set obtained by adding a single element to an already given finite set
>
> A Natural Number is either 0 or the number obtained by adding 1 to an already given Natural Number
>
> A finite sequence is either the empty sequence or the sequence obtained by appending an element to the end of an already given sequence
>
> A binary tree is either the empty binary tree or the tree obtained by 'putting together' two already given binary trees

In each case, the idea is that the entire set of objects can be 'generated' in this way. However, the trouble with such definitions is that they do not lead naturally to formal expressions using set comprehension. The best thing we can do is to write down the properties stated informally. For instance, in the case of the set NAT of Natural Numbers, we have

```
0:NAT
!n.(n:NAT => succ(n):NAT)
```

where "succ" is supposed to be the function which 'adds 1 to a number'. Since we 'know' that all natural numbers are characterized in this way, we can simplify the previous properties as follows, by using the image operator "image" defined in section 3.7, yielding

```
NAT = {0}\/image(succ)(NAT)
```

As you can see, the set NAT obeys an equation of the general form

```
NAT = gennat(NAT)
```

where "gennat" is supposed to be a certain 'set function'. For obvious reasons, such an equation is said to be a FIXPOINT equation. As all examples above (and many others) follow this general scheme (that is, obey a fixpoint equation), it is certainly worth investigating the possibility to define a (or maybe 'the') fixpoint of a function. Let 'fix' be this hypothetical operator. In the example of the natural number above, and up to the determination of the generating function "gennat", we have

```
NAT = fix(gennat)     that is, NAT is 'the' set s such that  s=gennat(s)
```

In what follows, we first give the definition of 'fix' and then construct various sets using this technique.

Given a set s, and a total function f from POW(s) to POW(s), we define fix(f) to be the (generalized) intersection (see section 2.3) of all subsets t of s such that f(t) is included in t, formally

fix(f) = inter {t | t:POW(s) & f(t) inc t}   DEFINITION

From the greatest lower bound properties of 'inter' (section 2.3), we can immediately deduce the following

(1) t:POW(s) & f(t) inc t =) fix(f) inc t
(2) !t.((t:POW(s) & f(t) inc t =) u inc t) =) u inc fix(f))

Moreover, we suppose that the function f is MONOTONE, that is

(3) x inc y =) f(x) inc f(y)   for all x and y in POW(s)

We now prove that "fix(f)" is indeed a fixpoint of f. For any t such that "t:POW(s) & f(t) inc t", we have "fix(f) inc t" after (1), hence after (3) we also have "f(fix(f) inc f(t)", therefore "f(fix(f)) inc t" by transitivity of inclusion and since "f(t) inc t"; consequently, after (2) where u is reolaced by "f(fix(f))", we have

(4) f(fix(f) inc fix(f)

Conversely, after (4) and because of (3), we have "f(f(fix(f))) inc f(fix(f))", therefore, after (1) where t is replaced by "f(fix(f))", we have

(5) fix(f) inc f(fix(f)

We have just re-proved Tarski's theorem stating that fix(f) is indeed a fixooint of f if f is monotone. We leave it as an exercise for the reader to prove that fix(f) is the LEAST such fixpoint (hence justifying our informal usage of 'the'), formally

fix(f) = f(fix(f))
t:POW(s) & t=f(t) =) fix(f) inc t

The fact that fix(f) is a lower bound leads directly to the possibility to prove properties of fix(f) BY INDUCTION. More precisely, suppose we like to prove that all members of fix(f) enjoy a certain property P, let t be the subset of fix(f) where the property holds

```
     t = {x | x:fix(f) & P}          we obviouly have:  t inc fix(f)
```

If we are able to prove that f(t) is included in t, then, after (1), fix(f) will be included in t, therefore be equal to it; formally, this corresponds to the following proof method

```
     f({x | x:fix(f) & P}) inc {x | x:fix(f) & P}  =>  !x.(x:fix(f) => P)
```

6. FINITE SETS

Given a set s, we denote by 'FIN(s)' the set of its FINITE subsets. We obviouly like to have 'FIN(s)' enjoying the following properties

```
     {}:FIN(s)
     x:s & t:FIN(s) => ({x}\/t):FIN(s)
```

These properties leads us to the construction the following set function

```
     genfin(s) = %z.(z:POW(POW(s)) | {{}} \/ $(x,t).(x,t:s*z | {x}\/t))
                                                      DEFINITION
```

We define 'FIN(s)' to be the fixpoint of this function

```
     FIN(s) = fix(genfin(s))                          DEFINITION
```

Therefore, and since the above function can be proved to be monotone, we have

```
     FIN(s) = genfin(s)(FIN(s))
            = [z:=FIN(s)]({{}} \/ $(x,t).(x,t:s*z | {x}\/t))
            = {{}} \/ $(x,t).(x,t:s*FIN(s) | {x}\/t)
```

Consequently

```
     {}:FIN(s)
     x:s & t:FIN(s) => ({x}\/t):FIN(s)
```

We have constructed a set with the desired property. As a by-product of this definition, we have an inductive method to be used in order to prove properties of finite sets, formally

```
[t:={}]P & !t.(t:FIN(s) & P => !x.(x:s => [t:={x}\/t]P)) => !t.(t:FIN(s) => P)
```

For instance, we can easily prove the following by induction

```
     t:FIN(s) & u:FIN(s) => (t\/u):FIN(s)
     t:FIN(s) & u:FIN(s) => (t/\u):FIN(s)
```

7. INFINITE SETS

A set is said to be INFINITE if it is not one of its finite subsets, formally

```
     infinite(s) = not(s:FIN(s))                        DEFINITION
```

An important property of infinite sets is that they are indeed infinite; more precisely, if t is a finite subset of an infinite set s, then the complement of t with respect to s (see section 2.4) is not empty since, in this case, t would be equal to s therefore be infinite, formally

```
     infinite(s) & t:FIN(s) => not(s-t={})
```

Finally, we postulate the existence of an infinite (God given) set that we name BIG

```
     infinite(BIG)                                      AXIOM
```

8. NATURAL NUMBERS

8.1. DEFINITION

Let us state again the caracteristic properties of natural numbers that we want to achieve

```
     0:NAT
     n:NAT => succ(n):NAT
```

The problem here is more complicated than in the previous case because we have to define from scratch the function 'succ' as well as the constant '0'. The only sets we know so far are the empty set '{}' and the infinite set 'BIG'. We have also a formal definition of the concept of 'finite subsets' of a given set. The idea is to represent each natural number by a finite subset of BIG; therefore 0 is obviously {}

```
     0 = {}                                                  DEFINITION
```

Given a number n (that is, a finite subset of BIG), we would like to construct its 'successor', an operation which, operationally, can be performed by adding a NEW element to n (that is, an element choosen in BIG but ouside n). Is this always possible? The answer is yes, since, by definition, n is a finite subset of the infinite set BIG (see section 7). The next question is: how are we going to choose such an element? Well, we shall suppose that we always have the possibility to choose an element in a set (provided the set in question is not empty of course) thanks to a (God given) CHOICE FUNCTION called 'tau' and axiomatized as follows

```
     not(s={}) => tau(s):s                                   AXIOM
```

Here is the definition of 'succ'

```
     succ = %n.(n:FIN(BIG) | {tau(BIG-n)}\/n)               DEFINITION
```

The definition of NAT follows; we define a function 'gennat' whose least fixpoint is NAT

```
     gennat = %s.(s:POW(FIN(BIG)) | {0}\/image(succ)(s))              DEFINITION

     NAT = fix(gennat)                                                DEFINITION
```

As a consequence, and since 'gennat' can be proved to be monotone, we have

```
     NAT = gennat(NAT)
         = [s:=NAT]({0}\/image(succ)(s))
         = {0}\/image(succ)(NAT)
```

We have constructed a set with the desired property. In what follows, we use also the constant "NAT1" to denote the set "NAT-{0}".

As we know, this construction gives us the possibility to prove properties of natural numbers by induction; in this case, the induction principle exhibited in section 6. can be re-stated as follows

```
     [n:=0]P & !n.(n:NAT & P => [n:=succ(n)]P) =>  !n.(n:NAT => P)
```

It is also possible to prove the remaining Peano axioms, namely

```
n:NAT => not(succ(n)=0)
n:NAT & m:NAT  =>  (succ(n)=succ(m) => n=m)
```

The proof of the last Peano axiom (the injectivity of 'succ') is not very simple. In fact, it requires the proof of a very important property of natural number which says that two natural numbers are included in each other, formally

```
m:NAT & n:NAT => (m inc n) or (n inc m)
```

The inclusion relation for natural numbers is the usual 'smaller than' relation which is thus a TOTAL order. We can also define the MINIMUM of a non empty set of natural numbers as its generalized intersection, formally

```
s:POW(NAT) & not(s={}) => min(s)=inter(s)
```

Of course, min(s), being a lower bound, is included in all members of s; we can also prove that this greatest lower bound is indeed a member of s (this comes from the fact that all members of s are embedded in each other), formally

```
s:POW(NAT) & not(s={})  =>  min(s):s
```

We have reconstructed the classical properties of the minimum of non-empty subsets of natural numbers and so proved that the 'smaller than' relation is a WELL order (since every non empty subset of NAT has a least element).

8.2. RECURSION ON NATURAL NUMBERS

Given a set s, an element a of s and a total function g from s to s, we would like to construct a total function f from NAT to s, obeying the following specification

```
f(0)         = a
f(succ(n)) = g(f(n))              when it makes sense
```

Of course, we do not know yet whether such a function does exist and even in this case, we do not know what its domain is (so that an expression such as "f(n)" is, for the moment, very dubious). In order to construct f, we shall use the following strategy: first, we construct a relation from NAT to s, and second we prove that this relation is a total function obeying the required specification. For this, we define the following (relation) function 'genf' whose least fixpoint is f

```
     genf = %f.(h:NAT<->s | {0|->a} \/ succ~;h;g)

     f = fix(genf)
```

Since 'genf' is obviously monotone, we have

```
     f = {0|->a} \/ succ~;f;g
```

We now prove by INDUCTION that the domain of f is NAT and that f is indeed a function, formally

```
     !n.(n:NAT => #y.(n,y:f & !z.(n,z:f => z=y))
```

We leave this proof as an exercise for the reader (use the fact that the restriction of "succ~" to NAT1 is a total function from NAT1 onto NAT). As is easily shown, we have eventually constructed a function enjoying the required properties.

8.3. ARITHMETIC

In order to construct arithmetic in a comoletely formal way, we can define addition and multiplication by recursion as follows

```
     m+0 = m
     m+succ(n) = succ(m+n)

     m*0 = 0
     m*succ(n) = m*n+m
```

We could also have define the ITERATE of a relation r with source and destination the same set s, by recursion

```
     iterate(r)(0)         = identity(s)
     iterate(r)(succ(n))   = iterate(r)(n);r              DEFINITION
```

The following properties being easily proved by induction

```
     f:s+->s => iterate(f)(n):s+->s
     f:s-->s => iterate(f)(n):s-->s
```

And then, we could have define addition and multiplication of natural numbers as follows

```
m+n = iterate(succ)(n)(m)
m*n = iterate(iterate(succ)(m))(n)(O)
```

Difference and division can be defined as 'converses' for addition and multiplication

```
(m-n)+n = m        if m is greater than or equal to n
(m/n)*n = m        if n is not equal to O
```

We leave it as an exercice for the reader to prove all elementary arithmetical properties.

For each finite sets of a set s, we can define its CARDINAL as the number of elements it contains

```
t:FIN(s) => card(t) = min {n:NAT | t:iterate(genfin(s))(n+1)({})}
```

In fact it can be proved that the infinite sequence

```
{}, iterate(genfin(s))(1)({}), ... , iterate(genfin(s))(n)({}), ...
```

converges to FIN(s); more precisely, we have

```
FIN(s) = union $n.(n:NAT | iterate(genfin(s))(n)({}))
```

As a consequence, the operator "min" is correctly used in the definition of "card" since the corresponding set is not empty.

9. SEQUENCES

A sequence built on a set s is either the empty sequence or the sequence obtained by 'pushing' amember of x at the beginning of a given sequence. In order to formalise this closed defintion, we need to make precise this idea of 'pushing'. Given a partial function f from NAT1 to a set s and a member x of s, we define "x->s" as follows

```
x->s = {1|->x}\/(succ~;s)                          DEFINITION
```

Informally speaking "x->s" 'pushes' x at the 'beginning' of s; for instance we have

```
x->{1|->y, 2|->z} = {1|->x, 2|->y, 3|->z}
```

The empty sequence (which is nothing else than the empty function) is denoted by (). The set of finite sequences built on s is denoted by "seq(s)"; this is the fixpoint of the function 'genseq(s)' defined as follows

```
     genseq(s) = %f.(g:FIN(NAT1+->s) | {()}\/ $(x,f).(x,f:s*g | x->f))

     seq(s) = fix(genseq(s))                                DEFINITIONS
```

As a consequence, we have

```
     ():seq(s)
     x:s & f:seq(s) => x->f:seq(s)
```

It can be shown that a sequence is a function whose domain is an INTERVAL from 1 to a natural number n (denoted by "1..n"); more precisely, it can be proved that "seq(s)" is equal to the union of the sets of functions having such domains, formally

```
     seq(s) = union $n.(n:NAT | (1..n)-->s)
```

For each sequence s, this number n (which is the cardinal of the domain of s) is called its SIZE, formally

```
     size(s) = card(dom(s))                                 DEFINITION
```

Note that the empty sequence is the function with domain the interval "1..0", hence of size 0; sequences defined in extension are special case of sets defined in extension; consequently, we use a special notation as shown on the following example

```
     (a,b,a) = {1|->a, 2|->b, 3|->a}                        DEFINITION
```

Properties of sequences can be proved by induction. Here is the statement of the corresponding principle

```
[s:=()]P &!s.(s:seq(t) & P => !x.(x:s => [s:=x->s]P)) => !s.(s:seq(s) & P)
```

As for natural numbers, sequence functions can be defined recursively by given their value at "()" and then at "x->s" in terms of their value at s. Examples of these, are functions to concatenate two sequences "s*t", to append an element at the end of a sequence "s<-x", to reverse a sequence "s^", or to do the generalized concatenation of a sequence of sequences "conc(s)", formally

```
    ()*s          = s                                        DEFINITIONS
    (x->s)*t      = x->(s*t)

    ()<-y         = y->()
    (x->s)<-y     = x->(s<-y)

    ()^           = ()
    (x->s)^       = s^<-x

    conc(s)       = ()
    conc(x->s)    = x*conc(s)
```

The following properties can easily be proved by induction

```
    s*()        = s
    s*(t*u)     = (s*t)*u
    s*(t<-x)    = (s*t)<-x
    (s<-x)*t    = s*(x->t)
    (s<-x)^     = x->s^
    (s*t)^      = t^*s^
    conc(s*t)   = conc(s)*conc(t)
```

10. TREES

10.1 BINARY TREES

A BINARY TREE is either the null binary tree or the tree obtained by considering two binary trees in a certain order: one is said to be the LEFT subtree and the other one the RIGHT subtree. This closed definition can be formalised using, of course, a certain fixpoint. Given two sets b1 and b2 of sequences built on the set {0,1}, we define the function "cons" as follows

```
    cons = %(b1,b2).(b1,b2:FIN(seq{0,1})*FIN(seo{0,1}) |
                    {()} \/ $s.(s:b1 | 0->s) \/ $s.(s:b2 | 1->s))

                                                  DEFINITION
```

For example, we have

```
    cons({(),(0),(1)},{(),(0),(0,1)})
  = {(),(0),(0,0),(0,1),(1),(1,0),(1,0,1)}
```

The null tree denoted by "NIL" is simply the empty set.

```
    NIL = {}                                              DEFINITION
```

The set BIN of binary trees can then be defined as the fixpoint of the following function "genbin"

```
    genbin = %s.(s:FIN(seq({0,1}) | {NIL} \/ image(cons)(s*s))

    BIN = fix(genbin)                                     DEFINITIONS
```

The function "genbin" is obviously monotone, so that we have

```
    NIL:BIN
    b1:BIN & b2:BIN => cons(b1,b2):BIN
```

Here is an example of binary tree

```
    {(), <0>, <0,0>, <0,1>, <1>, <1,0>, <1,0,1>}
```

This tree can be pictured as follows

```
         .
        / \
       /\ /
           \
```

An induction principle follows from this definition

```
    [b:=NIL]P &
    !(b1,b2).(b1:BIN & b2:BIN & [b:=b1]P & [b:=b2]P => [b:=cons(b1,b2)]P)  =>
    !b.(b:BIN => P)
```

10.2. LABELED BINARY TREES

We can also form the set of LABELED binary trees "bin(s)" built on a certain set s. Such a tree is the union of all functions whose domain are members of BIN

```
    bin(s) = union $b.(b:BIN | b-->s)
```

Note that "bin(s)" could also have been defined as a fixpoint. In fact the constructing function like "succ(n)", for natural numbers, "x->s" for sequences , or "cons(b[illegible]" for binary trees, is denoted this time by "b1/x\b2", so that we have the following induction principle for proving properties of "bin(s)".

```
      [b:=NIL]P &
      !(b1,b2).(b1:bin(s) & b2:bin(s) & [b:=b1]P & [b:=b2]P => [b:=b1/x\b2]P)  ⇒
      !b.(b:bin(s) => P)
```

We may also define functions on "bin(s)" recursively. Examples of these are the following transforming labeled binary trees into sequences in various ways (pre-, post-, in-order)

```
      pre(NIL) = <>
      pre(b1/x\b2) = x->(pre(b1)*pre(b2))

      in(NIL) = <>
      in(b1/x\b2) = in(b1)*(x->in(b2))

      post(NIL) = <>
      post(b1/x\b2) = post(b1)*post(b2)<-x

      mirror(NIL) = NIL
      mirror(b1/x\b2) = mirror(b2)/x\mirror(b1)
```

The following property can easily be proved by induction·

```
      pre(mirror(b)) = pre(b)^
```

10.3. N-ARY TREES

It is possible to generalise the conceot of finite binary trees to that of finite n-ary trees; that is, trees whose 'nodes' might have more than two outgoing 'branches' as is the case with binary trees. We use the same method as for defining the set BIN; this time, we represent an n-ary tree as a set of sequences built on NAT1; thus, the set "TREE" is a subset of the set "FIN(seq(NAT1))". In order to build a new tree from a sequence "st" of trees, the idea is to 'push' the corresponding index i at the beginning of each sequence of st(i). Here is the correspnding function

```
      build = %st.(st:seq(FIN(seq(NAT))) | {<>} \/
                                                     union $i.(i:dom(st) | i->st(i)))
```

The set TREE is then defined as the fixpoint of the following function

```
      gentree = %s.(s:POW(FIN(seq(NAT1)) | image(build)(seq(s))

      TREE = fix(gentree)                              DEFINITIONS
```

The function "gentree" being obviously monotone, we have

```
TREE = image(build)(seq(TREE))
```

And there is, as usual, an induction and a recursion principle. Note that the function "build" restricted to the set "seq(TREE)" is a bijection so that the set TREE is said to be 'isomorphic' to the set "seq(TREE)".

There is another interseting isomorphism, this time between TREE and BIN. More precisely, we can define the two function 'destruct' and 'construct' such that

```
destruct:  TREE-->BIN
construct: BIN-->TREE
```

as follows

```
destruct(build(<>))      = NIL
destruct(build(t->st))  = cons(destruct(t),destruct(build(st)))

construct(NIL)           = build(<>)
construct(cons(b1,b2))  = build(construct(b1)->build~(construct(b2)))
```

It can be shown by induction that

```
destruct = construct~
```

10.4. LABELED N-ARY TREES

As for binary trees, we can also define the set "tree(s)" of labeled n-ary trees built on a set s. In fact, this is the set of all functions whose domain are members of TREE, formally

```
trees(s) = union $t.(t:TREE |t-->s)                DEFINITION
```

We leave it as an exercise for the reader to develop the various corresponding notations.

REFERENCES

N. BOURBAKI   Theorie des ensembles   (Hermann)
A. LEVY   Basic Set Theory   (Springer-Verlag)
H. ENDERTON   Elements of Set Theory   (Academic Press)

Appendix 1: SUMMARY OF NOTATIONS

LOGIC

P => Q	P implies Q
P & Q	conjunction of P and Q
P or Q	disjunction of P and Q
not P	negation of P
!x.P	for all x, P
#x.P	for some x, P
[x:=E]P	P with free occurences of x reolaced by E

SETS

{}	empty set	1.
$x.(x:s & P | E)	set of objects of the form E for x in s where P	1.1.
$x.(x:s | E)	set of objects of the form E for x in s	1.1.
a,b	ordered pair (a followed by b)	1.2.
a|->b	ordered pair (a followed by b)	1.2.
s*t	cartesian product of s and t	1.2.
POW(s)	set of subsets of s	1.3.
x:s	x is a member of s	1.4.
s inc t	s is included in t	1.5.
{x | x:s & P}	subset of s where P	2.1.
{a}	set made of a	2.2.
{a,b}	set made of a and b	2.2.
inter(u)	intersection of the set (of sets) u	2.3.
union(u)	union of the set (of sets) u	2.3.
s/\t	intersection of s and t	2.3.
s\/t	union of s and t	2.3.
s-t	complement of s with respect to t	2.4.

RELATIONS

s<->t	set of binary relations from s to t	3.1.
dom(r)	domain of r	3.2.
cod(r)	co-domain of r	3.2.
r;s	forward composition of r and s	3.3.
identity(s)	identity on s	3.4.
r~	converse of r	3.6.
image(r)(s)	image of s under r	3.7.
r<+s	overriding of r by s	3.9.

FUNCTIONS

s+->t	set of partial functions from s to t	4.1.
s-->t	set of total functions from s to t	4.1.
%x.(x:s | E)	the function with value E at x (for x in s)	4.3.
s*+>t	set of partial injections from s to t	4.4.
s*->t	set of total injections from s to t	4.4.

FIXPOINT

fix(f)	fixpoint of the (set) function f	5.

FINITE AND INFINITE SETS

FIN(s)	set of finite subsets of s	6.
infinite(s)	s is infinite	7.
BIG	an infinite set (God given)	7.

NATURAL NUMBERS

tau(s)	a fixed element of the non-empty set s	8.1
succ	successor function	8.1.
NAT	set of natural numbers	8.1.
NAT1	set of positive natural numbers	8.1.
min(s)	minimum of a non-empty set of natural numbers	8.1.
iterate(r)(n)	nth iterate of a relation r	8.3.
m+n	sum of m and n	8.3.
m*n	product of m and n	8.3.
m-n	difference of m and n	8.3.
m/n	division of m by n	8.3.
card(t)	the number of elements of the finite set t	8.3.

SEQUENCES

x->f	aopending element x to sequence s	9.
seq(s)	set of sequences built on s	9.
⟨⟩	the emoty sequence	9.
⟨a, b, a⟩	the sequence made of a, b, and c in that order	9.
size(s)	the size of a finite sequence s	9.

BINARY TREES

cons	constructor function for binary trees	10.1.
NIL	the null tree	10.1.
BIN	set of binary trees	10.1.

LABELED BINARY TREES

bin(s)	set of labeled binary trees built on s	10.2.
pre	pre-order function	10.2.
in	in-order function	10.2.
post	post-order function	10.2.
mirror	mirror function	10.2.

N-ARY TREES

build	constructor function for n-ary trees	10.3.
TREE	set of n ary trees	10.3.
destruct	transforming an n-ary tree into a binary tree	10.3.
construct	transforming a binary tree into an n-ary tree	10.3.

LABELED N-ARY TREES

tree(s)	set of n-ary trees built on s	10.4.

Appendix 2: SUMMARY OF AXIOMS

(a,b)=(c,d)	=	a=c & b=d	pair equality	1.2.
y:$x.(x:s & P | E)	=	#x.(x:s & P & y=E)	membership	1.4.
(x,y):s*t	=	x:s & y:t	membership	1.4.
s:POW(t)	=	!x.(x:s => x:t)	membership	1.4.
not #x.(x:{})			empty set	1.4.
!x.(x:s = x:t)	=>	(s=t)	set equality	1.6.
f:s+->t & x:dom(f)	=>	x,f(x):f	function evaluation	4.2.
infinite(BIG)			infinity axiom	7.
not(s={})	=>	tau(s):s	choice function axiom	8.1.

Appendix 3: SUMMARY OF DEFINITIONS

Most of these definitions require some pre-conditions (check with the corresponding section)

```
  SETS

    $x.(x:s | E)   = $x.(x:s & TRUE | E)                                         1.1.
    s inc t        = s:POW(t)                                                    1.5.
    {x | x:s & P} = $x.(x:s & P | x)                                             2.1.
    {a}            = {x | x:s & (x=a)}                                           2.2.
    {a,b}          = {x | x:s & (x=a or x=b)}                                    2.2.
    inter(u)       = {x | x:s & !y.(y:u => x:y)}                                 2.3.
    union(u)       = {x | x:s & #y.(y:u & x:y)                                   2.3.
    a\/b           = union {a,b}                                                 2.3.
    a/\b           = inter {a,b}                                                 2.3.
    s-t            = {x | x:s & not(x:t)}                                        2.4.

  RELATIONS

    s(->t          = POW(s*t)                                                    3.1.
    dom(r)         = {x | x:s & #y.(x,y):r}                                      3.2.
    cod(r)         = {y | y:t & #x.(x,y):r}                                      3.2.
    r;s            = {x,z | x,z:u*w & #y.((x,y):r & (y,z):s)}                    3.3.
    identity(s)    = $x.(x:s | x,x)                                              3.4.
    r~             = {y,x | (y,x):t*s & (x,y):r}                                 3.6.
    image(r)(u)    = {y | y:u & #x.(x:u & (x,y):r)}                              3.7.
    r<+s           = (identity(u-dom(s));r)\/s                                   3.9.

  FUNCTIONS

    s+->t          = {r | r:s(->t & (r~;r):identity(cod(r))}                     4.1.
    s-->t          = {f | f:s+->t & dom(f)=s}                                    4.1.
    %x.(x:s | E)   = $x.(x:s | x,E)                                              4.3.
    s*+>t          = {f | f:s+->t & f~:t+->s}                                    4.4.
    s*->t          = s*+>t /\ s-->t                                              4.4.

  FIXPOINT

    fix(f)         = inter {t | t:POW(s)  &  f(t) inc t}                         5.

  FINITE AND INFINITE SETS

    genfin(s)      = %z.(z:POW(POW(s)) | {{}} \/ $(x,t).(x,t:s*z | {x}\/t))     6.
    FIN(s)         = fix(genfin(s))                                              6.
    infinite(s)    = not(s:FIN(s))                                               7.
```

NATURAL NUMBERS

```
  0                  = {}                                                      8.1.
  succ               = %n.(n:FIN(BIG) | {tau(BIG-n)}\/n)                       8.1.
  gennat             = %s.(s:POW(FIN(BIG)) | {0}\/image(succ)(s))              8.1.
  NAT                = fix(gennat)                                             8.1.
  NAT1               = NAT-{0}                                                 8.1.
  iterate(r)(0) = identity(s)                                                  8.3.
  iterate(r)(succ(n)) = iterate(r)(n);r                                        8.3.
  m+n                = iterate(succ)(n)(m)                                     8.3.
  m*n                = iterate(iterate(succ)(m))(n)(0)                         8.3.
  (m-n)+n            = m                                                       8.3.
  (m/n)*n            = m                                                       8.3.
  card(t)            = min {n:NAT | t:iterate(genfin(s))(n+1)({})}             8.3.
```

SEQUENCES

```
  x->s               = {1|->x}\/(succ~;f)                                      9.
  genseq(s)          = %f.(g:FIN(NAT1+->s) | {{}}\/$(x,f).(x,f:s*g | x->f))    9.
  seq(s)             = fix(genseq(s))                                          9.
  <>                 = {}                                                      9.
  <a,b,a>            = {1|->a, 2|->b, 3|->a}                                   9.
  size(s)            = card(dom(s))                                            9.
  <>*s               = s                                                       9.
  (x->s)*t           = x->(s*t)                                                9.
  <><-y              = y-><>                                                   9.
  (x->s)<-y          = x->(s<-y)                                               9.
  <>^                = <>                                                      9.
  (x->s)^            = s^<-x                                                   9.
  conc(<>)           = <>                                                      9.
  conc(x->s)         = x*conc(s)                                               9.
```

BINARY TREES

```
  cons               = %(b1,b2).(b1,b2:FIN(seq{0,1})*FIN(seq{0,1}) |
                         {<>} \/ $s.(s:b1 | 0->s) \/ $s.(s:b2 | 1->s))        10.1.
  NIL                = {}                                                     10.1.
  genbin             = %s.(s:FIN(seq({0,1}) | {NIL} \/ image(cons)(s*s))      10.1.
  BIN                = fix(genbin)                                            10.1.
```

LABELED BINARY TREES

```
   bin(s)               = union $b.(b:BIN | b-->s)                            10.2.
   pre(NIL)             = <>                                                  10.2.
   pre(b1/x\b2)         = x->(pre(b1)*pre(b2))                                10.2.
   in(NIL)              = <>                                                  10.2.
   in(b1/x\b2)          = in(b1)*(x->in(b2))                                  10.2.
   post(NIL)            = <>                                                  10.2.
   post(b1/x\b2)        = post(b1)*post(b2)<-x                                10.2.
   mirror(NIL)          = NIL                                                 10.2.
   mirror(b1/x\b2)      = mirror(b2)/x\mirror(b1)                             10.2.
```

N-ARY TREES

```
  build                  = %st.(st:seq(FIN(seq(NAT))) | {<>} \/
                           union $i.(i:dom(st) | i->st(i)))           10.3.
  gentree                = %s.(s:POW(FIN(seq(NAT1)) |
                           image(build)(seq(s))                       10.3.
  TREE                   = fix(gentree)                               10.3.
  destruct(build(<>))    = NIL                                        10.3.
  destruct(build(t->st)) = cons(destruct(t),destruct(build(st)))      10.3.
  construct(NIL)         = build(<>)                                  10.3.
  construct(cons(b1,b2)) = build(construct(b1)->build~(construct(b2)))10.3.
```

LABELED N-ARY TREES

```
  tree(s) = union $t.(t:TREE |t-->s)                                  10.4.
```

On the Meaning and Construction of the Rules in Martin-Lõf's Theory of Types

Roland Backhouse

CS 8606

18

On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types

Roland Backhouse
Department of Mathematics and Computing Science
University of Groningen
PO Box 800
9700 AV GRONINGEN
The Netherlands

Abstract We describe a method to construct the elimination and computation rules from the formation and introduction rules for a type in Martin-Löf's theory of types. The construction is based on an understanding of the inference rules in the theory as judgements in a pre-theory. The motivation for the construction is to permit disciplined extensions to the theory as well as to have a deeper understanding of its structure.

0 Introduction

Martin-Lõf's theory of types [ML0] has attracted considerable attention from both logicians and computing scientists, and for a variety of reasons. First, it has considerably enhanced our understanding of constructive proof and the relationship between such proofs and programs. Second, it anticipated the notion of dependent type introduced for example in the language Pebble [BL]. Third, as a formal system it has an elegant structure that is worthy of study in its own right. This paper is largely concerned with the latter aspect, the motivation being that by gaining a deeper understanding of its structure we will be better equipped to adapt the theory to individual needs

The present work grew out of a feeling of discontent with the theory. On first encounter the universal reaction among computing scientists appears to be that the theory is formidable. Indeed, several have specifically referred to the overwhelming number of rules in the theory. On closer examination, however, the theory betrays a rich structure — a structure that is much deeper than the superficial observation that types are defined by introduction, elimination and computation rules. Once recognised this structure considerably reduces the burden of understanding. And yet, to my knowledge, the structure of the theory has not been properly discussed or documented; Martin-Lõf, himself, alludes to the fact that there is a "pattern... in the type forming operations" in the preface to the notes prepared by Giovanni Sambin [ML1], but he does not give a detailed account of the pattern.

So much for the ideological motivations for this paper. At a more practical level it has become increasingly clear to us that there is a need to freely permit *disciplined* extensions to the theory. That the theory is open to extension is a fact that was clearly intended by Martin-Lõf. Indeed, it is a fact that has been exploited by several individuals; Nordström, Petersson and Smith [NPS] have extended the theory to include lists, they and Constable et al [Co] have added subset types and Constable et al have introduced quotient types, Nordström has introduced multi-level functions [No], Chisholm has introduced a very special-purpose type of tree structure [Ch] and Dyckhoff [Dy] has defined the type of categories.

Initially we were against such extensions on the grounds that it is often possible to define them in terms of the W-type (for examples see [Kh]), because they add to the complexity of the theory and because they might undermine the quality of the theory even to the extent of introducing inconsistencies. The experiences and arguments of others have now convinced us that this view is wrong. The view that we now hold is that implementations of type theory (proof checkers, proof editors etc. like Nuprl and the Gothenburg Type Theory System) should permit user-defined extensions to the theory but in a disciplined way. This paper is therefore a first attempt at formulating such a discipline.

The main contribution that we make here is to describe a scheme for computing the elimination rule and computation rules for a newly introduced type. In other words, we show that it suffices to provide the type formation rule and the introduction rules for a new type; together these provide sufficient information from which the remaining details can be deduced. (At this stage in our work we cannot provide such a scheme to cover all type constructors; the limitations of our work are discussed in the conclusion.) The significance of this result is that it has the twin benefits of reducing the burden of understanding and the burden of definition. It reduces the burden of understanding since we now need to understand only the formation and introduction rules and the general scheme for inferring the remaining rules. Conversely, the burden of definition is reduced since it suffices to state the formation and introduction rules, the others being inferred automatically.

A necessary preliminary was to give an explanation of the meaning of the formal rules in the theory. Such an explanation is notably absent from the seminal account of Martin-Lõf's theory [ML0]; although the paper gives a very careful account of the meaning of the various judgement forms, nowhere is it stated how to interpret the rules. Yet, it is fundamental that a type be defined by its rules and that the rules be meaningful in some precise sense. We therefore begin this paper by providing an account, in section 2, of the rules in type theory as judgements in a "pre-theory", that is, a theory that precedes the theory of types itself. Also in section 2 we introduce the notion of internal consistency of a rule. The pre-theory is taken from [NPS], with which we assume some familiarity, and is summarised in section 1.

The main body of the paper is contained in section 3. Here we detail the scheme for computing elimination and computation rules. Several examples of the scheme are also included in this section.

There are many shortcomings in this stage of our work. Some of those of which I am aware are discussed in the conclusions. Needless to say I would be grateful for further criticism and comments.

1 The Pre-Theory

The pre-theory that we need involves an understanding of the theory of expressions and the notion of a category as discussed by Nordström, Petersson and Smith [NPS], and to which we refer the reader for complete details.

The theory of expressions defines the arity of expressions and definitional equality of expressions. For understanding the rules that follow it is necessary to know that different occurences of the same variable in a rule denote definitionally equal expressions. Identical expressions are, of course, definitionally equal but also $((x)P)(x)$ is definitionally equal to P for any expression P and variable x, and $((x)c)(y)$ is definitionally equal to c for any constant c and variables x and y. In particular $((x)\mathit{Type})(y)$ is definitionally equal to *Type*, since *Type* is a constant.

The rules of the pre-theory (and of the theory) prescribe the formation of derivations and from derivations one may abstract judgements. The syntactic form of derivations and judgements is described in essence by the following BNF syntax

$$\begin{aligned}
\langle \mathit{derivation} \rangle &::= \langle \mathit{statement} \rangle * \\
\langle \mathit{statement} \rangle &::= \langle \mathit{primitive\ statement} \rangle \mid \langle \mathit{context} \rangle \\
\langle \mathit{context} \rangle &::= \text{``}\|\text{''} \langle \mathit{assumption} \rangle \text{``}\triangleright\text{''} \langle \mathit{derivation} \rangle \text{``}\|\text{''} \\
\langle \mathit{assumption} \rangle &::= \langle \mathit{statement} \rangle \\
\langle \mathit{judgement} \rangle &::= \langle \mathit{primitive\ statement} \rangle \mid \text{``}\|\text{''} \langle \mathit{assumption} \rangle \text{``}\triangleright\text{''} \langle \mathit{judgement} \rangle \text{``}\|\text{''}
\end{aligned}$$

A derivation is thus a sequence of statements each of which is either a primitive statement or a context. Contexts are delimited by the scope brackets "||" and "||" and consist of an assumption followed by a (sub-) derivation. A judgement is formed from a derivation by the simple process of eliding all but the last statement in the derivation and in all its sub-derivations. For example consider a derivation of the form

$$\begin{array}{l}
p_0 \\
\| \; a_0 \\
\quad \triangleright \; p_1 \\
\quad\quad \| \; a_1 \\
\quad\quad\quad \triangleright \; p_2 \\
\quad\quad \| \\
\|
\end{array}$$

where p_0,p_1 and p_2 are primitive statements and a_0,a_1 are assumptions. Then the judgement obtained by eliding all but the last statement in each derivation is the following.

$$\begin{array}{l} \| \;\; a_0 \\ \quad \triangleright \; \| \;\; a_1 \\ \qquad\qquad \triangleright \; p_2 \\ \qquad\;\; \| \\ \| \end{array}$$

which may be read as "*assuming* a_0 *and assuming* a_1 *then* p_2".

We say that a statement p *precedes* a statement q within a derivation if p is the ith statement of the derivation, for some i, and either (a) q is the jth statement of the derivation for some $j > i$ or (b) the jth statement, for some $j > i$, is a context that includes the statement q. The statement p also precedes the statement q in a derivation if p precedes q in a subderivation of the derivation. Thus in the example above statement p_0 precedes statements a_0,p_1,a_1 and p_2. Also p_1 precedes $\| \; a_1 \; \triangleright \; p_2 \; \|$ and p_2, and so on.

Each rule in the pre-theory (and in the theory) consists of a set of premises and a conclusion, in the usual way. The application of a rule permits a derivation to be *extended* by adding a statement to the end of the derivation or to the end of a subderivation provided that the extended derivation includes statements preceding the added statement that match the premises in the same way that the added statement matches the conclusion. An axiom is a rule that has no premises; thus application of an axiom permits a derivation to be extended at an arbitrary point.

Note that there is considerable freedom in the order of construction of statements in a derivation. The form in which derivations are presented on the printed page will suggest one particular order but it should not be supposed that this is the only order.

Just those rules that we explicitly employ are given below. For these rules we explain their meaning in an ad hoc way. We do not, however, attempt to give any meaning to the word *category*: the reader must accept that certain expressions denote "categories", which expressions being determined by application of the rules. Thus the first rule must be accepted as an axiom - "*Type*" denotes a category.

$$\frac{\qquad\qquad}{\mathit{Type\ cat}} \qquad\qquad \text{Type Formation}$$

"*Type cat*" is a primitive statement and therefore a derivation and a judgement.

Contexts may be introduced into a derivation via the assumption rule.

$$\frac{C \; \mathit{cat}}{\begin{array}{l} \| \;\; x : C \\ \quad\; \triangleright \\ \| \end{array}} \qquad\qquad \text{Assumption}$$

If in a derivation we have a primitive statement of the form C *cat* then it is possible to extend the derivation by adding an assumption of the form $x :\; C$ where x is a variable. Note that the assumption is a particular sort of primitive statement. For clarity it is separated from following statements by the symbol "$\triangleright$".

For each type A the elements of A form a category. Thus we have the rule of element formation.

$$\frac{A : \mathit{Type}}{\mathit{El}(A) \; \mathit{cat}} \qquad\qquad \text{Element formation}$$

The rule permits a derivation that includes a statement of the form $A : \mathit{Type}$ to be extended by adding the statement $\mathit{El}(A)$ *cat* to the derivation. In so doing the context of both statements must be identical.

Function categories are obtained by discharging assumptions.

$$\frac{\begin{array}{l} A \; \mathit{cat} \\ \| \;\; x :\; A \\ \quad \triangleright \; B(x) \; \mathit{cat} \\ \| \end{array}}{F(A,B) \; \mathit{cat}} \qquad\qquad \text{Function formation}$$

$F(A,B)$ is the category of functions that map an object x of the category A into an object of the category $B(x)$. Note that $B(x)$ does not denote an expression containing free occurrences of x, as it would in conventional mathematics, but an expression that is definitionally equal to the application of some expression B of arity $0 \rightarrow 0$ to some variable x. For instance $Type$ takes the form $B(x)$ since it is definitionally equal to $((y)Type)(x)$.

The final rule we need in the pre-theory is the rule of function elimination.

$$\frac{\begin{array}{l} a : A \\ c :\ F(A,B) \end{array}}{c(a) :\ B(a)} \qquad \text{Function Elimination}$$

An example of a derivation using these rules is as follows. Note that the line numbers and material within braces are not part of the derivation but are only included as aids to the reader. Also, the symbol "≡" has been used to denote definitional equality.

```
        {Type formation}
0       $Type\ cat$
        {0, assumption}
1.0     || $X : Type$
           ▷ {1.0, El-formation}
1.1        $El(X)\ cat$
           {1.1, assumption}
1.2.0      || $x : El(X)$
              ▷ {Type formation}
1.2.1         $Type\ cat$
           ||
           {1.1, 1.2, $((x)Type)(x) \equiv Type$, fun-formation}
1.3        $F(El(X), (x)Type)\ cat$
           {1.1, assumption}
1.4.0      || $y : El(X)$
              ▷ {1.3, assumption}
1.4.1.0       || $Y : F(El(X), (x)Type)$
                 ▷ {1.4.0, 1.4.1.0, $((X)Type)(y) \equiv Type$, fun-elim}
1.4.1.1          $Y(y) : Type$
              ||
           ||
        ||
```

The judgement obtained from this derivation by eliding all but the last statement in every subderivation is the following.

```
|| $X : Type$
   ▷ || $y : El(X)$
      ▷ || $Y : F(El(X), (x)Type)$
         ▷ $Y(y) : Type$
        ||
     ||
||
```

In words, assuming X is a type, y is an element of X and Y is a function mapping elements of X into the category of types, then Y applied to y is a type.

2. The Rules of Type Theory

Now that we have discussed the pre-theory we may proceed to explicate the meaning of the rules in type theory itself. We do this by interpreting each rule of type theory as a judgement in the pre-theory. The premises of the rule become assumptions of the pre-theory judgement.

This rather simple idea has far-reaching consequences. It means that we can decide whether the premises of a type-theory rule make sense by constructing a derivation in the pre-theory. We can also check that the conclusion of the rule obeys a certain consistency requirement (called *internal consistency* in the sequel).

Some preliminary examples may help to convey the idea. Let us consider the formation, introduction and elimination rules for the disjoint-sum type.

Below we show the formation rule and the corresponding pre-theory judgement. Here the correspondence is immediate: premises become assumptions and *P type* is replaced by *P*: *Type*.

$$\begin{array}{l} A\ type \\ B\ type \\ \hline A \vee B type \end{array}$$

type-theory rule

$$\begin{array}{l} \| \; A : Type \\ \quad \triangleright \; \| \; B : Type \\ \qquad\quad \triangleright \; A \vee B : Type \\ \qquad\;\; \| \\ \| \end{array}$$

pre-theory judgement

∨-formation

Next consider one of the introduction rules for the disjoint-sum type. Again we exhibit the type-theory rule and the corresponding pre-theory judgement.

$$\begin{array}{l} A\ type \\ B\ type \\ x \in A \\ \hline i(x) \in A \vee B \end{array}$$

type-theory rule

$$\begin{array}{l} \| \; A : Type \\ \quad \triangleright \; \| \; B : Type \\ \qquad\quad \triangleright \; \| \; x : El(A) \\ \qquad\qquad\quad \triangleright \; i(x) : El(A \vee B) \\ \qquad\qquad\;\; \| \\ \qquad\;\; \| \\ \| \end{array}$$

pre-theory judgement

∨-introduction

This example is more illuminating because we can use it to give a preliminary account of what it means for an introduction rule to be internally consistent. Specifically, given an introduction rule with conclusion $e \in E$ we convert the rule into a judgement $El(E)$ *cat* under assumptions derived from (in a manner yet to be described) the premises of the rule. The rule is then said to be internally consistent if the judgement can be verified using the rules of the pre-theory and the formation rules of the type. Thus for our example we verify internal consistency by establishing the judgement

$$\begin{array}{l} \| \; A : Type \\ \quad \triangleright \; \| \; B : Type \\ \qquad\quad \triangleright \; \| \; x : El(A) \\ \qquad\qquad\quad \triangleright \; El(A \vee B)\ cat \\ \qquad\qquad\;\; \| \\ \qquad\;\; \| \\ \| \end{array}$$

This judgement has the following derivation.

$$\begin{array}{ll}
 & \{\text{Type formation}\}\\
0 & \mathit{Type\ cat}\\
 & \{0,\text{assumption}\}\\
1.0 & \| \; A : \mathit{Type}\\
 & \quad \triangleright \{0,\text{ assumption}\}\\
1.1.0 & \qquad \| \; B : \mathit{Type}\\
 & \qquad\quad \triangleright \{1.0,\text{ element formation}\}\\
1.1.1 & \qquad\qquad El(A)\ \mathit{cat}\\
 & \qquad\qquad \{1.1.1,\text{assumption}\}\\
1.1.1.0 & \qquad\qquad \| \; x : El(A)\\
 & \qquad\qquad\quad \triangleright \{1.0,1.1.0,\vee\text{-formation}\}\\
1.1.1.1 & \qquad\qquad\qquad A \vee B : \mathit{Type}\\
 & \qquad\qquad\qquad \{1.1.2.1,\text{element formation}\}\\
1.1.1.1 & \qquad\qquad\qquad El(A \vee B)\ \mathit{cat}\\
 & \qquad\qquad \|\\
 & \qquad \|\\
 & \|
\end{array}$$

Note that the penultimate step makes use of the ∨-formation rule.

Finally consider the elimination rule for disjoint sum. The type-theory rule and corresponding judgement are shown below.

$$\begin{array}{l}
A\ \mathit{type}\\
B\ \mathit{type}\\
\| \; x \in A \vee B\\
\quad \triangleright \; C(x)\ \mathit{type}\\
\|\\
\| \; y \in A\\
\quad \triangleright \; d(y) \in C(i(y))\\
\|\\
\| \; y \in B\\
\quad \triangleright \; e(y) \in C(j(y))\\
\|\\
f \in A \vee B\\
\hline
\omega(f,d,e) \in C(f)
\end{array}$$

type-theory rule

$$\begin{array}{l}
\| \; A : \mathit{Type}\\
\quad \triangleright \; \| \; B : \mathit{Type}\\
\qquad\quad \triangleright \; \| \; C : F(El(A \vee B),(x)\mathit{Type})\\
\qquad\qquad\quad \triangleright \; \| \; d : F(El(A),(y)El(C(i(y))))\\
\qquad\qquad\qquad\quad \triangleright \; \| \; e : F(El(B),(y)El(C(j(y))))\\
\qquad\qquad\qquad\qquad \triangleright \; \| \; f : El(A \vee B)\\
\qquad\qquad\qquad\qquad\quad \triangleright \; \omega(f,d,e) : El(C(f))\\
\qquad\qquad\qquad\qquad \|\\
\qquad\qquad\qquad\quad \|\\
\qquad\qquad\quad \|\\
\qquad\quad \|\\
\quad \|\\
\|
\end{array}$$

pre-theory judgemtnt

∨-elimination

The additional complexity of this example arises from the hypothetical premises (that is, premises involving assumptions). The specific translation process used converts a premise of the form $\| \; x \in A \triangleright J \|$ as follows. First convert the judgement J to, say, $b(x)$: $B(x)$. Then construct the judgement $b : F(El(A), B)$. Thus the premise $\| \; x \in A \vee B \triangleright C(x)\ \mathit{type} \|$ is converted by first converting $C(x)\ \mathit{type}$ to $C(x) : \mathit{Type}$, which is definitionally equal to $C(x) : ((x)\mathit{Type})(x)$. Then the judgement $C : F(El(A \vee B),(x)\mathit{Type})$ is constructed.

As an exercise the reader may wish to verify the internal consistency of the rule by constructing a derivation of the following judgement.

$$\begin{array}{l}
\| \; A : Type \\
\quad \triangleright \; \| \; B : Type \\
\qquad \triangleright \; \| \; C : F(El(A \vee B), (x)Type) \\
\qquad\quad \triangleright \; \| \; d : F(El(A), (y)El(C(i(y)))) \\
\qquad\qquad \triangleright \; \| \; e : F(El(B), (y)El(C(j(y)))) \\
\qquad\qquad\quad \triangleright \; \| \; f : El(A \vee B) \\
\qquad\qquad\qquad \triangleright \; El(C(f)) \; cat \\
\qquad\qquad\quad\;\; \| \\
\qquad\qquad\quad \| \\
\qquad\qquad \| \\
\qquad\quad \| \\
\quad\;\; \| \\
\|
\end{array}$$

2.1 Formalising the Conversion from Type Theory Rules to Pre-Theory Judgements

Converting from type theory rules to pre-theory judgements is a purely syntactic process which we now summarise.

Each type theory rule consists of an ordered sequence of premises and a single inference. The pre-theory judgement consists of an ordered sequence of assumptions (each corresponding to a premise) and a single conclusion. Individual statements (premise or inference) of the rule are converted according to the following algorithm.

(a) "E *type*" is converted to "$E :$ *Type*"
(b) "$e \in E$" is converted to "$e : El(E)$"
(c) given a hypothetical premise of the form $\| \; H \; \triangleright \; S \; \|$ first convert H to the form "$x : E$" and S to the form "$d(x) : D$". Then the premise is converted to "$d : F(E, (x)D)$". Note that definitional equality may be required to complete the conversion of H and/or S to the required form. Note also that the construction permits the hypothesis H to be itself hypothetical.

Consider now the presentation of a new type in the theory. The order of presentation of the rules is (1) formation rules (2) introduction rules (3) elimination rule (4) computation rules. (We do not consider equality rules in this paper although they do not pose additional difficulties.) Suppose the inference of one of these rules is converted to a statement of the form "$e : E$" and the premises are converted to statements of the form "$a_1 : A_1$",...,"$a_n : A_n$". Then we say that the rule is *internally consistent* if and only if there is a derivation of the judgement

$$\| \; a_1 : A_1 \triangleright \; \ldots \triangleright \; \| \; a_n : A_n \triangleright \; E \; cat \;\; \| \cdots \; \|$$

in a system consisting of the pre-theory and those rules governing the type that precede the given rule in the above order.

3 Introducing New Types into the Theory

The mechanism fcr introducing a new type into the theory has three stages. First the formation rule for the type is prescribed followed by the introduction rules. Finally the elimination rules and computation rule are automatically inferred from the formation rule and the introduction rules (provided, of course, that the latter are internally consistent). This is the subject of the next three sections.

3.1 Formation Rules

Each formation rule introduces some new type constructor, Θ say. The premises of such a rule all take the form "*A type*", for some expression A and within some context. The corresponding pre-theory judgement therefore has the form

$$\begin{array}{l}
|[\; A_1 : T_1 \\
\quad \triangleright \; |[\; A_2 : T_2 \\
\qquad \triangleright \; |[\; \cdots \\
\qquad\quad \triangleright \; |[\; A_n : T_n \\
\qquad\qquad \triangleright \; \Theta(A_1,\ldots,A_n) : Type \\
\qquad\quad]| \\
\qquad]| \\
\quad]| \\
]|
\end{array}$$

where the expressions $T_1,\ldots,T_n$ are all elements of a class of expressions *TYPE* where *TYPE* has the following syntax

$$TYPE ::= \text{“}Type\text{”} \;\mid\; \text{“}F(\text{” } expression \text{ “,” } TYPE \text{ “)”}$$

For future reference we refer to $A_1,\ldots,A_n$ as the *formation variables.*

3.2 Introduction Rules

A type Θ may have several introduction rules, each one of which introduces a new canonical-object constructor. Consider one such constructor, θ say. Then the premises of the θ-introduction rule are in two parts. First there are the premises of the Θ-formation rule. (These are rarely stated explicitly.) Second there is a number, m say, of premises each of the form "$b \in B$" for some expressions b and B and within some context. Thus the corresponding pre-theory judgement takes the form

$$\begin{array}{l}
|[\; A_1 : T_1 ; \ldots ; A_n : T_n \\
\quad \triangleright \; |[\; x_1 : S_1 ; \ldots ; x_m : S_m \\
\qquad \triangleright \; \theta(x_1,\ldots,x_m) : El(\Theta(A_1,\ldots,A_n)) \\
\quad]| \\
]|
\end{array}$$

The first set of premises corresponds to the premises of the Θ-formation rule. The expressions $S_1,\ldots,S_m$ in the second set of premises all belong to the syntactic category *ELEMENT* defined as follows:

$$ELEMENT ::= \text{“}El(\text{” } expression \text{ “)”} \;\mid\; \text{“}F(El(\text{” } expression \text{ “),” } abstract\ ELEMENT \text{ “)”}$$

Again for future reference we call the variables $x_1,\ldots,x_m$ the *θ-introduction variables.*

The θ-introduction rule may be recursive: that is, one or more of the premises of the rule may take the form that, in a certain context, $b \in \Theta(l)$ for some expression b and some list of expressions l. (For such a premise to make sense the inference rule must be internally consistent as defined earlier.) If the ith premise is indeed recursive we refer to x_i as a *recursive θ-introduction variable.*

3.3 Elimination Rules

For each type constructor Θ there is exactly one elimination rule. Let us suppose the elimination rule introduces the non-canonical constant Θrec, and that there are k introduction rules defining canonical constants $\theta_1,\ldots,\theta_k$. Then the elimination rule is formed as follows.

0 (premises of Θ-formation)

1 $a \in \Theta(A_1,\ldots,A_n)$

2 $\|$ $w \in \Theta(A_1,\ldots,A_n)$

$\triangleright$ $C(w) type$

$\|$

3 $\|$ (context computed from θ_1-introduction rule)

$\triangleright$ $z_1(l_1, s_1) \in C(\theta_1(l_1))$

$\|$

$\vdots$

$2+k$ $\|$ (context computed from θ_k-introduction rule)

$\triangleright$ $z_k(l_k, s_k) \in C(\theta_k(l_k))$

$\|$

____ Θ-elimination

$\Theta rec(a, z_1, ..., z_k) \in C(a)$

The premises are divided into four parts. In the first part the premises of Θ-formation are repeated once again (and also once again rarely explicitly). The second part postulates the existence of an object "a" of type Θ (where "a" is a new identifier). The third part postulates that C (where "C" is a new identifier) is a family of types indexed by objects of type Θ. Finally the fourth part consists of a premise for each canonical-object constructor θ. (In the above schema $z_1,\ldots,z_k$ are new variables and $s_1,\ldots,s_k,l_1,\ldots,l_k$ are lists of variables constructed from the introduction rules in in a manner to be described shortly.) A summary of the Θ-elimination rule would be that the proof of a property $C(a)$ given object a of type Θ proceeds by structural induction, i.e. by case analysis on the possible form of the canonical value of a.

The premise (referred to later as p_θ) corresponding to the θ-introduction rule takes the form:

$\|$ (context computed from θ-introduction rule)

$\triangleright$ $z(l, s) \in C(\theta(l))$

$\|$

where l is simply the list of θ-introduction variables but where the construction of the context and the list of θ-elimination variables, s, depends on whether the introduction rule is or is not recursive. The details of their construction are as follows.

Suppose that the θ-introduction rule has the following form.

(premises of Θ-formation)

$\|$ (context 1)

$\triangleright$ $b_1 \in B_1$

$\|$

...

$\|$ (context m)

$\triangleright$ $b_m \in B_m$

$\|$

____

$\theta(l) \in \Theta(t)$

where $b_1,\ldots,b_m,B_1,\ldots,B_m$ are expressions, l is the list of θ-introduction variables and t is the list of Θ-formation variables. Then the premise, p_θ, to be included in the Θ-elimination rule has the form

$\|$ $\langle assumption(s)\ 1\rangle$

$;\ \ldots$

$;\ \langle assumption(s)\ m\rangle$

$\triangleright\ z(l, s) \in C(\theta(l))$

$\|$

Here "$\langle$assumption(s) $k\rangle$" refers to either one or two assumptions depending on whether x_k is or is not a recursive θ-introduction variable. In the case that x_k is not recursive then "$\langle$assumption(s) $k\rangle$" is simply a repetition of the corresponding premise in the θ-introduction rule. That is "$\langle$assumption(s) $k\rangle$" is

$\|$ $\langle$context $k\rangle$

$\triangleright\ b_k \in B_k$

$\|$

On the other hand if x_k is a recursive θ-introduction variable then, by definition, the corresponding premise of the θ-introduction rule takes the form

$\|$ $\langle$context $k\rangle$

$\triangleright\ x_k(u_k) \in \Theta(l)$

$\|$

for some list of variables u_k. In this case $\langle$assumption(s) $k\rangle$ consists of

(a) a repetition of the premise as in the case of a non-recursive θ-introduction variable, and
(b) the assumption

$\|$ $\langle$context $k\rangle$

$\triangleright\ y_k(u_k) \in C(x_k(u_k))$

$\|$

where y_k is a new variable.

The list, s, of θ-elimination variables is then just the list of variables, y_k, that are introduced by the recursive θ-introduction variables.

3.4 Computation Rules

The computation rules for type Θ are in (1-1) correspondence with the introduction rules for Θ. Thus for each canonical-object-constructor, θ say, there is exactly one computation rule which prescribes how to apply Θrec to a θ-object. Again the method of constructing the rule is complicated by the presence of recursive introduction variables.

For the purpose of this discussion let us identify θ with its index in the list of canonical-object constructors for the type Θ. Also let us denote by l_θ the list of θ-introduction variables.

In general the computation rule for θ-objects is a combination of the θ-introduction rule and the Θ-elimination rule. A schema for its construction is as follows.

$$\begin{array}{lll} 0 & \langle\text{premises of }\Theta\text{-formation}\rangle & \\ 1 & \langle\text{premises of }\theta\text{-introduction (excluding }\Theta\text{-formation premises)}\rangle & \\ 2\ldots2{+}k & \langle\text{premises }2\ldots2+k\text{ of }\Theta\text{-elimination}\rangle & \\ \hline & \Theta rec(\theta(l_\theta), z_1, \ldots, z_k) = z_\theta(l_\theta, v) \in C(\theta(l_\theta)) & \theta\text{-computation} \end{array}$$

Apart from the construction of the list of expressions v (which we have yet to describe) the construction of the θ-computation rule is thus very straightforward.

There is an expression in the list v for each recursive θ-introduction variable. Suppose $x_i (1 \leq i < m_\theta)$ is such a variable and the corresponding premise of the θ-introduction rule is

$\|$ $\langle$context $i\rangle$

$\triangleright\ x_i(u_i) \in \Theta(t)$

$\|$

Then the entry in the list v takes the form

$$(u_i)\Theta rec(z_i(u_i), z_1, \ldots, z_k)$$

.

3.5 Examples

We present several examples of the construction of the elimination and computation rules. First consider the disjoint sum type. This has the formation rule:

$$\frac{\begin{array}{l} A_1\ type \\ A_2\ type \end{array}}{A_1 \vee A_2\ type} \qquad \vee\text{-formation}$$

and two introduction rules:

$$\frac{\begin{array}{l} A_1\ type \\ A_2\ type \\ x \in A_1 \end{array}}{i(x) \in A_1 \vee A_2} \qquad \text{i-introduction}$$

$$\frac{\begin{array}{l} A_1\ type \\ A_2\ type \\ x_1 \in A_2 \end{array}}{j(x) \in A_1 \vee A_2} \qquad \text{j-introduction}$$

The list of $\vee$-formation variables is thus (A_1, A_2), the list of i-introduction variables is (x) and the list of j-introduction variables is also (x).

Referring back to section 3.3 we construct the following elimination rule.

$$\begin{array}{ll} 0 & A_1\ type \\ & A_2\ type \\ 1 & a \in A_1 \vee A_2 \\ 2 & \| \ w \in A_1 \vee A_2 \\ & \quad \triangleright\ C(w)\ type \\ & \| \\ 3 & \| \ x \in A_1 \\ & \quad \triangleright\ z_1(x) \in C(i(x)) \\ & \| \\ 4 & \| \ x \in A_2 \\ & \quad \triangleright\ z_2(x) \in C(j(x)) \\ & \| \\ \hline & \vee\text{-}elim(a, z_1, z_2) \in C(a) \end{array} \qquad \vee\text{-elimination}$$

Also, referring to section 3.4 we construct the following computation rules

$$\begin{array}{ll} 0 & A_1\ type \\ & A_2\ type \\ 1 & x \in A_1 \\ 2\ldots4 & \text{(as in $\vee$-elimination)} \\ \hline & \vee\text{-}elim(i(x), z_1, z_2) = z_1(x) \in C(i(x)) \end{array} \qquad \text{i-computation}$$

$$\begin{array}{ll}
0 & A_1\ \mathit{type} \\
 & A_2\ \mathit{type} \\
1 & x \in A_2 \\
2\ldots4 & \text{(as in }\vee\text{-elimination)} \\ \hline
 & \vee\text{-}\mathit{elim}(j(x), z_1, z_2) = z_2(x) \in C(i(x))
\end{array}\qquad \text{j-computation}$$

Our second example is concocted to illustrate the problems of recursive introduction variables. The formation rule is as follows.

$$\begin{array}{l}
A\ \mathit{type} \\ \hline
H(A)\ \mathit{type}
\end{array}\qquad \text{H-formation}$$

The type has one introduction rule

$$\begin{array}{l}
A\ \mathit{type} \\
\|\ \ v \in A \\
\quad \triangleright\ x(v) \in H(A) \\
\| \\ \hline
h(x) \in H(A)
\end{array}\qquad \text{h-introduction}$$

Note that the h-introduction variable x is recursive by virtue of the premise "$x(v) \in H(A)$".

From these two rules we compute the H-elimination rule according to the schema described in section 3.3

$$\begin{array}{ll}
0 & A\ \mathit{type} \\
1 & a \in H(A) \\
2 & \|\ \ w \in H(A) \\
 & \quad \triangleright\ C(w)\ \mathit{type} \\
 & \| \\
3 & \|\ \ \|\ \ v \in A \\
 & \quad\quad \triangleright\ x(v) \in H(A) \\
 & \quad \| \\
 & \quad ;\ \ \|\ \ v \in A \\
 & \quad\quad\quad \triangleright\ y(v) \in C(x(v)) \\
 & \quad\quad \| \\
 & \quad \triangleright\ z(x, y) \in C(h(x)) \\
 & \| \\ \hline
 & H\mathit{rec}(a, z) \in C(a)
\end{array}\qquad \text{H-elimination}$$

Finally the computation rule takes the following form.

$$\begin{array}{ll}
0 & A\ \mathit{type} \\
1 & \|\ \ v \in A \\
 & \quad \triangleright\ x(v) \in H(A) \\
 & \| \\
2\ldots3 & \text{(as in } H\text{-elimination)} \\ \hline
 & H\text{-}\mathit{rec}(h(x), z) = z(x, (v)H\text{-}\mathit{rec}(x(v), z)) \in C(h(x))
\end{array}\qquad \text{h-computation}$$

We conclude our list of examples with the definition of the W-type.

The W-type stands out in Martin-Lōf's presentation of his theory [ML0] because it is the only one that appears to require an understanding of other types in the theory. In particular the rules given there appeal to an understanding of the Π-type and of "$\rightarrow$", a symbol that is not defined (although Martin-Lōf does make its meaning clear elsewhere [ML1]). One important aspect of the rationalisation of the theory, mentioned by Martin-Lōf in the Padova Notes [ML1], detailed by Nordstrōm, Peterson and Smith [NPS] and exploited by us in this paper, is that this lacuna has been overcome. The W-type is also the one that is considered to be the hardest to understand. Our contribution is thus to show that it may be understood solely by understanding its introduction rule and the general scheme for inferring elimination and computation rules.

The W-formation rule is as follows.

$$\begin{array}{l} A_1\ \mathit{type} \\ \|\ \ x \in A_1 \\ \quad \triangleright\ A_2(x)\ \mathit{type} \\ \| \\ \hline W(A_1, A_2)\ \mathit{type} \end{array} \qquad \text{W-formation}$$

The W-type also has just one (recursive) introduction rule:

$$\begin{array}{l} \text{(premises of } W\text{-formation)} \\ x_1 \in A_1 \\ \|\ \ v \in A_2(x_1) \\ \quad \triangleright\ x_2(v) \in W(A_1, A_2) \\ \| \\ \hline \mathit{sup}(x_1, x_2) \in W(A_1, A_2) \end{array} \qquad \text{sup-introduction}$$

According to the schema for its construction the W-elimination rule therefore takes the following form.

$$\begin{array}{ll} 0 & \text{(premises of } W\text{-formation)} \\ 1 & a \in W(A_1, A_2) \\ 2 & \|\ \ w \in W(A_1, A_2) \\ & \quad \triangleright\ C(w)\mathit{type} \\ & \| \\ 3 & \|\ \ x_1 \in A_1 \\ & \quad ;\ \|\ \ v \in A_2(x_1) \\ & \qquad \triangleright\ x_2(v) \in W(A_1, A_2) \\ & \quad\ \ \| \\ & \quad ;\ \|\ \ v \in A_2(x_1) \\ & \qquad \triangleright\ y(v) \in C(x_2(v)) \\ & \quad\ \ \| \\ & \quad \triangleright\ z(x_1, x_2, y) \in C(\mathit{sup}(x_1, x_2)) \\ & \| \\ \hline & W\text{-}\mathit{rec}(a, z) \in C(a) \end{array} \qquad \text{W-elimination}$$

Finally the single computation rule takes the following form.

$$\begin{array}{ll} 0 & \text{(premises of } W\text{-formation)} \\ 1 & x_1 \in A_1 \\ & \|\ \ v \in A_2(x_1) \\ & \quad \triangleright\ x_2(v) \in W(A_1, A_2) \\ & \| \\ 2..3 & \text{(as in } W\text{-elimination)} \\ \hline & W\text{-}\mathit{rec}(\mathit{sup}(x_1, x_2), z) = z(x_1, x_2, (v)W\text{-}\mathit{rec}(x_2(v), z)) \end{array} \qquad \text{sup-computation}$$

Conclusions

Martin-Lōf's theory of types has a rich structure which we hope this paper has helped to expose. Our account must, however, be recognised as very preliminary. This section is therefore devoted to a description of the work that we plan to do in the near future.

To begin with there are certain flaws in the above account. In particular, it has been pointed out to us that additional constraints apply to the use of recursive introduction variables. Thus in the first draft of this paper our example of h-introduction (see section 3.5) had the premise

$$\begin{array}{l} \| \ v \in H(A) \\ \quad \rhd \ x(v) \in H(A) \\ \| \end{array}$$

which should be prohibited on account of the fact that there is a negative occurrence of a recursive introduction variable leading to nonterminating programs. (I am grateful to Per Martin-Lōf for pointing this out.) This highlights a lack of a semantic justification of the scheme we have described, but which we intend to remedy.

Secondly we intend to describe schemes for the construction of derivations of closure properties and uniqueness properties of a type. Closure properties are properties like "every element of a disjoint sum is either of the form $i(a)$ or $j(b)$ for elements a and b of the appropriate type" and uniqueness properties express the fact that objects introduced by distinct introduction rules are always distinct. Thus the two sets of properties reflect the fact that types introduced into the theory are extremal, and, of course, they are fundmental to our understanding. For particular instances of such derivations the reader is referred to [Ba].

Thirdly, we intend to extend the construction to novel type structures such as the subset type [Co,NPS] in which the type introduction rules incur information loss. For the subset type and similar type constructors that we have in mind the way ahead is clear. The quotient type introduced by the PRL group [Co] is much less clear to us.

Finally, we intend to try to provide a collection of examples that illustrate our thesis that the ability to introduce new type constructors is an indispensable feature of the theory - and, consequently, of implementations of the theory.

References

[Ba] R.C. Backhouse "Notes on Martin-Lōf's Theory of Types" Department of Computer Science, University of Essex, England, Section 1, Feb. 1986, Section 2, June 1986.

[Be] M.J. Beeson "Foundations of Constructive Mathematics" Springer-Verlag, Berlin (1985).

[BL] R. Burstall and B.Lampson "A kernel language for abstract data types and modules," in Semantics of Data Types, Eds. G.Kahn, D.B.MacQueen and G.Plotkin, Lecture Notes in Computer Science 173, 1-50 (1984).

[Ch] P. Chisholm "Derivation of a parsing algorithm in Martin-Lōf's Theory of Types", Department of Computer Science, Heriot-Watt University, Scotland, U.K., 1985. (To appear in Science of Computer Programming).

[Co] R.L. Constable et al "Implementing Mathematics with the Nuprl Proof Development System" Prentice-Hall, Inc., Englewood Cliffs, N.J. (1986).

[DF] E.W. Dijkstra and W.H.J. Feijen "Een methode van programmeren", Academic Service, 's-Gravenhage (1984)

[Dy] R. Dyckhoff "Category theory as an extension of Martin-Lōf's Type Theory", University of St. Andrews (1985).

[Kh] A.M.A. Khamiss "Program Construction in Martin-Lōf's Theory of Types", Ph.D. Thesis, University of Essex, Department of Computer Science (1986).

[ML0] P. Martin-Lōf "Constructive Mathematics and Computer Programming", pp. 153-175 in Logic, Methodology and Philosophy of Science, VI, North-Holland Publishing Company, Amsterdam (1982), Proceedings of the 6th International Congress, Hannover, 1979.

[ML1] P. Martin-Lōf "Intuitionistic Type Theory" Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980.

[No] B. Nordström "Multilevel Functions in Type Theory", pp. 206-235 in Workshop on Programs as Objects, Lecture Notes in Computer Science, Vol. 217, Springer-Verlag, Berlin (October 1985)
[NPS] B. Nordström, K. Peterson and J. Smith "An Introduction to Martin-Löf's Type Theory", Draft, midsummer 1986, Programming Methodology Group, Chalmers University of Technology, S-41296 Göteborg, Sweden.

Acknowledgements My thanks go to Bengt Nordström and Per Martin-Löf both of whom have provided helpful criticisms of this report. Thanks go also to Kees Straatman for helping me to win the battle with TEX.

This work was begun whilst I was still employed by the Department of Computer Science of the University of Essex and was supported at that time by a grant from the Science and Engineering Research Council. I am grateful to both organisations for their support.

The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes

Jan M. Smith
Department of Computer Science
University of Göteborg/Chalmers
S-412 96 Göteborg
Sweden

January 1987

Abstract

Martin-Löf's type theory without universes is interpreted in the calculus of truth values. The interpretation shows that no negated equalities can be proved without universes and also gives a finitary consistency proof of type theory without universes.

1 Introduction

In Hilbert-Ackermann [2] there is given a simple proof of the consistency of first order predicate logic by reducing it to propositional logic. Intuitively, the proof is based on interpreting predicate logic in a domain with only one element. Tarski [7] and Gentzen [1] have extended this method to simple type theory by starting with an individual domain consisting of a single element and then interpreting a higher type by the set of truth valued functions on the previous type.

I will use the method of Hilbert and Ackermann on Martin-Löf's type theory without universes to show that $\neg\,\mathsf{Eq}(A,a,b)$ cannot be derived without universes for any type A and any objects a and b of type A. In particular, this proves the conjecture in Martin-Löf [5] that Peano's fourth axiom $(\forall x\in \mathsf{N})\neg\,\mathsf{Eq}(\mathsf{N},0,\mathsf{succ}(x))$ cannot be proved in type theory without universes. The construction will also give a consistency proof by finitary methods of Martin-Löf's type theory without universes. So, without universes, the logic obtained by interpreting propositions as types is surprisingly weak. This is in sharp contrast with type theory as a computational system, since, for instance, the proof that every object of a type can be computed to normal form cannot be formalized in first order arithmetic.

The nonderivability of $\neg\,\mathsf{Eq}(\mathsf{N},0,1)$ for the version of type theory given in Martin-Lõf [4] was already shown in Smith [6] as a corollary to a somewhat less straightforward construction made with a different purpose. The proofs in this paper will work for any of the different formulations of Martin-Lõf's type theory.

2 The construction of the interpretation

We define a truth valued function φ on the types of Martin-Lõf's type theory without universes. Intuitively, $\varphi(A) = \top$ means that the interpretation of the type A is a set with one element and $\varphi(A) = \bot$ means that A is interpreted as the empty set. φ is defined for each type expression $A(x_1, \ldots, x_n)$ by recursion on the length of the derivation of $A(x_1, \ldots, x_n)$ *type* $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1, \ldots, x_{n-1})]$, using the clauses

$$\begin{aligned}
\varphi(\mathsf{N}_0) &= \bot \\
\varphi(\mathsf{N}_k) &= \top \qquad (k = 1, 2, \ldots\ ) \\
\varphi(\mathsf{N}) &= \top \\
\varphi(\mathsf{Eq}(A, a, b)) &= \varphi(A) \\
\varphi(A + B) &= \varphi(A) \vee \varphi(B) \\
\varphi((\Pi x \in A) B(x)) &= \varphi(A) \rightarrow \varphi(B(x)) \\
\varphi((\Sigma x \in A) B(x)) &= \varphi(A) \wedge \varphi(B(x)) \\
\varphi((\mathsf{W} x \in A) B(x)) &= \varphi(A) \wedge (\neg \varphi(B(x))) \\
\varphi(\{x \in A \mid B(x)\}) &= \varphi(A) \wedge \varphi(B(x))
\end{aligned}$$

$\wedge$, $\vee$, $\rightarrow$, and $\neg$ denote the usual boolean operations.

That φ really interprets type theory in the way we have intended is the content of the following theorem.

Theorem. *Let* $a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)$ $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1, \ldots, x_{n-1})]$ *be derivable in type theory without universes. Then* $\varphi(A(x_1, \ldots, x_n)) = \top$ *provided that* $\varphi(A_1) = \cdots = \varphi(A_n(x_1, \ldots, x_{n-1})) = \top$.

In the proof of this theorem we will use two lemmas. The first says that the truth value assigned to a type expression is preserved under substitution. The second lemma says that equality between types is preserved by φ.

Lemma 1. *If* $A(x_1,\ldots,x_n)$ *type* $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})]$ *and* $a_1 \in A_1,\ \ldots,\ a_n \in A_n(a_1,\ldots,a_{n-1})$ *are derivable in type theory without universes, then* $\varphi(A(x_1,\ldots,x_n)) = \varphi(A(a_1,\ldots,a_n))$.

Proof. The proof is by induction on the length of the derivation of $A(x_1,\ldots,x_n)$ *type* $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})]$. The only type forming rule where free variables may be introduced is Eq-formation. Since $\varphi(\mathsf{Eq}(A,a,b)) = \varphi(A)$ the induction hypothesis directly gives the result.

Lemma 2. *If* $A(x_1,\ldots,x_n) = B(x_1,\ldots,x_n)$ $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})]$ *is derivable in type theory without universes, then* $\varphi(A(x_1,\ldots,x_n)) = \varphi(B(x_1,\ldots,x_n))$..

Proof. This lemma is straightforwardly proved by induction on the length of the derivation of $A(x_1,\ldots,x_n) = B(x_1,\ldots,x_n)$ $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})]$. Note that lemma 1 is needed for the rule

$$\frac{a = b \in A \qquad C(x)\ \mathit{type}\ [x \in A]}{C(a) = C(b)}$$

Proof of the theorem. The proof is by induction on the length of the derivation of $a(x_1,\ldots,x_n) \in A(x_1,\ldots,x_n)$ $[x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})]$. I will only discuss a few of the rules; the remaining can be handled in the same way.

Equality of types

$$\frac{a \in A \qquad A = B}{a \in B}$$

By the induction hypothesis we have that $\varphi(A) = \top$ and, by lemma 2, that $\varphi(A) = \varphi(B)$. Hence, $\varphi(B) = \top$.

There are different formulations of the rules for the Eq-type in Martin-Lõf [3] and Martin-Lõf [4,5]. I will here use the earlier formulation which is the one now used by Martin-Lõf since it does not destroy the decidability of the judgemental equality $a = b \in A$.

Eq-introduction

$$\frac{a \in A}{\mathsf{eq}(a) \in \mathsf{Eq}(A,a,a)}$$

Since, by the definition of φ, $\varphi(\mathsf{Eq}(A,a,a)) = \varphi(A)$, the induction hypothesis directly gives $\varphi(\mathsf{Eq}(A,a,a)) = \top$.

Eq-elimination

$$\frac{c \in \mathsf{Eq}(A,a,b) \qquad d(x) \in C(x,x,\mathsf{eq}(x))\ \ [x \in A]}{\mathsf{J}(c,d) \in C(a,b,c)}$$

By the induction hypothesis we have that $\varphi(\mathsf{Eq}(A,a,b)) = \top$ and that $\varphi(C(x,x,\mathsf{eq}(x))) = \top$ if $\varphi(A) = \top$. Hence, since $\varphi(\mathsf{Eq}(A,a,b)) = \varphi(A)$, $\varphi(C(x,x,\mathsf{eq}(x))) = \top$ which, by lemma 1, gives $\varphi(C(a,b,c)) = \top$.

If we instead had considered the Eq-rules in Martin-Lõf [4] we could simplify the definition of φ by putting $\varphi(\mathsf{Eq}(A,a,b)) = \top$.

Π-introduction

$$\frac{b(x) \in B(x) \ \ [x \in A]}{\lambda(b) \in (\Pi x \in A)B(x)}$$

By the induction hypothesis we know that $\varphi(B(x)) = \top$ if $\varphi(A) = \top$. Since $\varphi((\Pi x \in A)B(x)) = \varphi(A) \rightarrow \varphi(B(x))$ this gives that $\varphi((\Pi x \in A)B(x)) = \top$.

Π-elimination

$$\frac{a \in A \qquad c \in (\Pi x \in A)B(x)}{\mathsf{apply}(c,a) \in B(a)}$$

According to the induction hypothesis, we have $\varphi(A) = \top$ and $\varphi((\Pi x \in A)B(x)) = \top$, which, since $\varphi((\Pi x \in A)B(x)) = \varphi(A) \rightarrow \varphi(B(x))$, gives that $\varphi(B(x)) = \top$. Hence, by lemma 1, $\varphi(B(a)) = \top$.

N-elimination

$$\frac{n \in \mathsf{N} \qquad d \in C(0) \qquad e(x,y) \in C(\mathsf{succ}(x)) \ \ [x \in \mathsf{N},\ y \in C(x)]}{\mathsf{rec}(n,d,e) \in C(n)}$$

By the induction hypothesis we have that $\varphi(C(0)) = \top$ which, by lemma 1, gives $\varphi(C(n)) = \top$.

3 Some consequences of the interpretation

3.1 The unprovability of Peano's fourth axiom

By the interpretation we can now see that for no type A and terms a and b does there exist a closed term t such that

$$t \in \neg\,\mathsf{Eq}(A,a,b) \qquad (*)$$

is derivable in type theory without universes. Assume that $(*)$ holds. Then there must exist a derivation of $\mathsf{Eq}(A,a,b)$ *type* and, hence, also a derivation of $a \in A$. So, by the theorem, $\varphi(A) = \top$ which, together with the definitions of φ and $\neg$, gives

$$\varphi(\neg\,\mathsf{Eq}(A,a,b)) = \varphi(\mathsf{Eq}(A,a,b) \rightarrow \mathsf{N}_0) = \varphi(\mathsf{Eq}(A,a,b)) \rightarrow \varphi(\mathsf{N}_0) = \varphi(A) \rightarrow \bot = \bot$$

Hence, by the theorem, $\neg\,\mathsf{Eq}(A,a,b)$ cannot be derived in type theory without universes.

Assume that Peano's fourth axiom can be derived, that is, that we have a derivation of

$$s \in (\Pi x \in \mathsf{N})\neg\,\mathsf{Eq}(\mathsf{N},0,\mathsf{succ}(x))$$

for some closed term s. By Π-elimination we then get $\mathsf{apply}(s,0) \in \neg\,\mathsf{Eq}(\mathsf{N},0,\mathsf{succ}(0))$ which is of the form $(*)$ and therefore impossible to derive in type theory without universes.

That no negated equalities can be proved reflects the intuition behind φ, which is that it interprets type theory in a domain with a single element. We can make this explicit inside type theory by introducing a new constant $\star$ and for each type A such that $\varphi(A) = \top$ adding a new rule

$$\star \in A$$

The theorem can still be proved with this new rule added, so the extension is consistent. Since $\varphi((\Pi x \in A)\mathsf{Eq}(A,x,\star)) = \varphi(A) \to \varphi(A) = \top$ we have that $\star \in (\Pi x \in A)\mathsf{Eq}(A,x,\star)$, that is, all objects of a type are equal to $\star$. Note that the extension is classical because $\star \in A \vee (\neg A)$. Since $\star \in \mathsf{Eq}(\mathsf{N},0,1)$, type theory with universes becomes inconsistent if the $\star$-rule is added.

3.2 Well-orderings

The definition of φ on well-orderings, $\varphi((\mathsf{W}x \in A)B(x)) = \varphi(A) \wedge (\neg\varphi(B(x)))$, is made as to validate the rules in Martin-Löf [4]. The W-introduction rule in [4] does not have a bottom clause $0 \in (\mathsf{W}x \in A)B(x)$ since such a clause can be derived using a universe. We can now see that this use of a universe is necessary. Since $\varphi((\mathsf{W}x \in A)B(x)) = \top$ implies $\varphi(A) = \top$ and $\varphi(B(x)) = \bot$ we get, by the theorem, that if $(\mathsf{W}x \in A)B(x)$ is not empty then $B(a)$ must be empty for all a in A. This gives that all elements of a well-ordering type are initial, that is, without predecessors. So, only very trivial well-orderings can be constructed.

If we add a bottom clause to the W-rules and change the definition of φ by $\varphi((\mathsf{W}x \in A)B(x)) = \top$, we get the full computational strength of the well-ordering types and can still prove our theorem.

3.3 Consistency

That type theory is consistent means that there is no term of type N_0. By the theorem, type theory without universes is consistent since $\varphi(\mathsf{N}_0) = \bot$. Clearly, this consistency proof is finitary in the sense of Hilbert and can be carried out in primitive recursive arithmetic. This may seem surprising since the proof theoretic strength of type theory without universes measured in terms of provable well-orderings is, without well-ordering types, the same as first order arithmetic and, with well-ordering types, even far beyond ε_0. However, this is not in conflict with Gödel's second incompletness theorem, because in order to prove Gödel's theorem, primitive recursive predicates must be numeralwise expressible in the theory and, as we have seen, not even equality is numeralwise expressible in type theory without universes.

3.4 Universes

If φ was extended to a universe, then $\varphi(\mathsf{T}(a))$ has to be defined for each object a of the universe U because of the rule

$$\frac{a \in \mathsf{U}}{\mathsf{T}(a)\ type}$$

which says that if a is the code of a type then $\mathsf{T}(a)$ is the type that a encodes. Let n_0 and n_1 be the codes of N_0 and N_1 respectively. Since

$$\mathsf{T}(\mathsf{n}_0) = \mathsf{N}_0 \quad \text{and} \quad \mathsf{T}(\mathsf{n}_1) = \mathsf{N}_1$$

we must have

$$\varphi(\mathsf{T}(\mathsf{n}_0)) = \bot \quad \text{and} \quad \varphi(\mathsf{T}(\mathsf{n}_1)) = \top$$

Hence, lemma 1, which is crucial for the proof of the theorem, would no longer hold.

An obvious way of extending type theory in order to obtain the strength of first order arithmetic is to add Peano's fourth axiom. This would not, however, follow the general pattern of introduction and elimination rules in type theory which is very natural, particularly when viewing a type as a set and not as a proposition: the elements of a set are defined by the introduction rules and the elimination rule makes it possible to define functions by recursion on the set.

Martin-Löf has instead suggested to extend type theory without universes by using the objects 0_2 and 1_2 of type N_2 as codes for N_0 and N_1 respectively. We then have to add the type formation rule

$$\frac{a \in \mathsf{N}_2}{\mathsf{T}(a)\ type}$$

and the type equalities

$$\mathsf{T}(0_2) = \mathsf{N}_0 \qquad\qquad \mathsf{T}(1_2) = \mathsf{N}_1$$

With these new rules added, Peano's fourth axiom can be proved as in [5]. Another similar way of extending type theory without universes in order to obtain the strength of first order arithmetic, is to add a very small universe U_{Bool} by restricting the first universe U to only have two objects, n_0 and n_1, coding N_0 and N_1 respectively.

References

[1] G. Gentzen. Die Widerspruchsfreiheit der Stufenlogik. *Mathematische Zeitschrift 41, No. 3, 1936, pp.357-366.*

[2] D. Hilbert and W. Ackermann. Grundzüge der Theoretischen Logik. *Springer-Verlag 1928.*

[3] P. Martin-Lõf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, North-Holland, 1975.*

[4] P. Martin-Lõf. Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science, pp. 153-175. North-Holland, 1982.*

[5] P. Martin-Lõf. Intuitionistic Type Theory. *Studies in Proof Theory, Lecture Notes, Bibliopolis, Napoli, 1984.*

[6] J.M. Smith. On a Nonconstructive Type Theory and Program Derivation. To appear in the proceedings of *Conference on Logic and its Applications, Bulgaria 1986 (Pergamon Press).*

[7] A. Tarski. Einige Betrachtungen über die Begriffe der ω-Wiederspruchsfreiheit und der ω-Vollständigkeit. *Monatsh. Math. Phys. 40, 1933. pp.279-295.*

Inductively Defined Sets in Martin-Löf's Set Theory

(Draft)

Peter Dybjer

April 16, 1987

Abstract

There are several possible schemes for introducing inductive definitions in Martin-Löf's intuitionistic set theory. One such scheme is obtained by including a fixed point operator in the theory. The rules for such fixed points are displayed, and it is shown that Aczel's interpretation of Martin-Löf's set theory in a logical theory of constructions can be extended accordingly. Moreover, algorithms are given which derive introduction and elimination rules for particular inductively defined sets and set operators (parameterised sets). Another scheme is obtained by representing inductively defined sets as well-orderings. A theorem shows that this method yields isomorphic representations if one assumes extensional equality of functions.

1 Introduction

Recursive data structures, such as natural numbers, lists, and binary trees, are very important in programming. The corresponding notion in Martin-Löf's intuitionistic set theory (or type theory) is that of an inductively defined set. In [9] there is no general scheme for inductive definitions, however. This is unlike Coquand and Huet's theory of constructions [5], where inductively generated sets can be defined impredicatively by using second order quantification. The only common recursive data structure which is introduced as a primitive is the set of natural numbers. Other recursive data structures can instead be represented in terms of the powerful well-orderings. In [10] it is for example shown how to represent the set of natural numbers and the set of ordinals of the second number class as certain well-orderings.

But intuitionistic set theory should not be viewed as a closed framework: new set operators can be added when there is a need for them. For example, in [10] the list former is added as a primitive set forming operation. Another example can be found in [13] where rules for inductively defined sets of multilevel functions are given.

It seems that there are general methods both for representing inductively defined sets in terms of well-orderings and for adding new rules for inductively defined sets. We

*Author's address: Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden shall discuss such methods here for sets inductively generated by certain set operators G, which are built up from the identity and constants by the standard operators $+$, $\times$ and $\rightarrow$ and which are ***strictly positive***, that is, the set variable X must not occur to the left of an arrow in GX. I shall use strictly positive to refer to an operator in this collection in the sequel.

The problem of adding new rules for inductively defined sets has been dealt with by Roland Backhouse. In [3] he gives a general scheme for introducing new set operators by giving formation and introduction rules, and then deriving elimination and equality rules. In section 3 I shall discuss another method based on adding a fixed point operator which can be applied to strictly positive set operators. In section 4 I shall also show how to derive the constructors, and thus the introduction rules, and the selector, and thus the elimination rule, for the set $\mathit{Fix}\, G$ from a G in a certain subcollection of the above collection of strictly positive operators.

The problem of representing inductively defined sets as well-orderings is discussed in section 5, where I show that all inductively defined sets which are generated by a strictly positive operator can be isomorphically represented as well-orderings. However, this theorem assumes the extensional equality relation on sets as given in [9] and [10]. It does not hold in the intensional intuitionistic set theory given by Martin-Löf [8] and which is described in section 2.2.

Acknowledgements. I wish to thank Per Martin-Löf for helpful advice and criticism and in particular for telling me about the general rules for fixed points in intuitionistic set theory. He also suggested that I study algorithms for deriving constructors and recursion operators for inductively defined sets. Moreover, I wish to thank Peter Aczel for telling me that well-orderings could be used for representing a large collection of inductively defined sets.

2 Martin-Löf's Type Theory and Set Theory

I shall in this paper use the notational framework proposed by Martin-Löf in [8]. This framework is similar to the Edinburgh Logical Framework [7] and has its origins in Church's simple type theory, the languages of the AUTOMATH project [6], and the theory of constructions [5]. To make this paper reasonably self-contained, a survey of the notational framework is given in section 2.1. Martin-Löf has proposed to call the framework 'type theory', since it is a theory of 'logical types'. Thus 'type' as in 'intuitionistic type theory' and 'data type' has to be changed to something different, for example, 'set'. In section 2.2 a survey of Martin-Löf's theory of sets in this sense (intuitionistic set theory) is given.

2.1 Type Theory

I essentially follow Martin-Löf's presentations in [8] and at this workshop [11], but write $x_1. \cdots .x_n.e$ instead of $(x_1)\cdots(x_n)e$ for abstraction and $f\, x_1 \cdots x_n$ instead of $f(x_1, \ldots, x_n)$ for application.

There are four kinds of *'analytic' judgements* in this theory:

$$\begin{aligned} \alpha &: \mathit{type}, \\ \alpha = \beta &: \mathit{type}, \end{aligned}$$

$$\begin{array}{rcl} a & : & \alpha, \\ a = b & : & \alpha. \end{array}$$

The theory is a typed λ-calculus with dependent function space types, a ground type ‘set’, and a rule saying that every object of type ‘set’ is also a type.

The rules of type formation are:

$$\mathit{set} : \mathit{type},$$

$$\frac{A : \mathit{set}}{A : \mathit{type}},$$

$$\frac{\alpha : \mathit{type} \qquad \begin{array}{c}(x : \alpha)\\ \beta : \mathit{type}\end{array}}{(x : \alpha)\beta : \mathit{type}}.$$

We may write $(\alpha)\beta = (x : \alpha)\beta$ if β does not depend on x.

The rules of object formation are:

$$\frac{\begin{array}{c}(x : \alpha)\\ b : \beta\end{array}}{x.b : (x : \alpha)\beta},$$

$$\frac{b : (x : \alpha)\beta \qquad a : \alpha}{b\,a : \beta[a/x]},$$

and a rule of assumption. The equality rules are typed β- and η-conversion:

$$\frac{a : \alpha \qquad \begin{array}{c}(x : \alpha)\\ b : \beta\end{array}}{(x.b)\,a = b[a/x] : \beta[a/x]},$$

$$\frac{c : (x : \alpha)\beta}{c = x.c\,x : (x : \alpha)\beta}.$$

Moreover, we may everywhere substitute equals for equals.

There is also a notion of *polymorphism.* The rule of polymorphic type formation is:

$$\frac{\alpha : \mathit{type} \qquad \begin{array}{c}(x : \alpha)\\ \beta : \mathit{type}\end{array}}{\overline{(x : \alpha)}\beta : \mathit{type}}.$$

The rules of object formation are:

$$\frac{\begin{array}{c}(x : \alpha)\\ b : \beta\end{array}}{b : \overline{(x : \alpha)}\beta},$$

where b must not depend on x, and

$$\frac{b : \overline{(x : \alpha)}\beta \qquad a : \alpha}{b : \beta[a/x]}.$$

A *theory* consists of a *signature*, which is a finite list $c_1 : \alpha_1, \ldots, c_n : \alpha_n$ of assignments of types to constants, and a finite list $a_1 = b_1 : \beta_1, \ldots, a_m = b_m : \beta_m$ of *axioms* or *equations*. If we have formulated a *sensible theory* then the four forms of analytic judgements are decidable. (The notion of sensible theory was discussed on several occasions during this workshop. For the judgements to be decidable one assumes that there is no polymorphism and that expressions are kept in normal form.)

Definitions are introduced by writing judgements of the form $c = a : \alpha$ (or sometimes just $c = a$) where c is the definiendum (a new constant), a is the definiens (an object), and α is their type. For example:

$$\circ = g.f.a.g\,(f\,a) : \overline{(A : \mathit{set})(B : \mathit{set})(C : \mathit{set})}((B)C)((A)B)(A)C.$$

I also freely utilise ordinary notational conventions, such as infix notation, and thus write $g \circ f$ instead of $\circ\, g\, f$.

2.2 Intuitionistic Set Theory

Let us represent Martin-Löf's set theory in the notational framework. The rules are more or less those of [10], except that the rules for equality are those for intensional equality from [8]. I have also changed the order of the arguments of the selectors (F, E, D, J, R_n, R, T) so that the principal argument comes last instead of first.

Rules for the cartesian product of a family of sets:

$$\begin{aligned}
\prod &: (A : \mathit{set})((A)\mathit{set})\mathit{set},\\
\lambda &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}((a : A)B\,a)\prod A\,B,\\
F &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}\overline{(C : (\prod A\,B)\mathit{set})}\\
&\quad ((b : (a : A)B\,a)C\,(\lambda b))(c : \prod A\,B)C\,c,\\
d.b.F\,d\,(\lambda b) = d.b.d\,b &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}\overline{(C : (\prod A\,B)\mathit{set})}\\
&\quad ((b : (a : A)B\,a)C\,(\lambda b))(b : (a : A)B\,a)C\,(\lambda b).
\end{aligned}$$

Rules for the disjoint union of a family of sets:

$$\begin{aligned}
\sum &: (A : \mathit{set})((A)\mathit{set})\mathit{set},\\
\langle \_, \_ \rangle &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}(a : A)(B\,a)\sum A\,B,\\
E &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}\overline{(C : (\sum A\,B)\mathit{set})}\\
&\quad ((a : A)(b : B\,a)C\,\langle a, b\rangle)(c : \sum A\,B)C\,c,\\
d.a.b.E\,d\,\langle a, b\rangle = d.a.b.d\,a\,b &: \overline{(A : \mathit{set})(B : (A)\mathit{set})}\overline{(C : (\sum A\,B)\mathit{set})}\\
&\quad ((a : A)(b : B\,a)C\,\langle a, b\rangle)(a : A)(b : B\,a)C\,\langle a, b\rangle.
\end{aligned}$$

Rules for the disjoint union of two sets:

$$\begin{aligned}
+ &: (\mathit{set})(\mathit{set})\mathit{set},\\
i &: \overline{(A : \mathit{set})(B : \mathit{set})}(A)A + B,\\
j &: \overline{(A : \mathit{set})(B : \mathit{set})}(B)A + B,\\
D &: \overline{(A : \mathit{set})(B : \mathit{set})}\overline{(C : A + B)}
\end{aligned}$$

$$
\begin{array}{rcl}
 & & ((a:A)C\,(i\,a))((b:B)C\,(j\,b))(c:A+B)C\,c,\\
d.e.a.D\,d\,(i\,a)=d.e.a.d\,a & : & \overline{(A:set)(B:set)}\overline{(C:(\sum A\,B)set)}\\
 & & ((a:A)C\,(i\,a))((b:B)C\,(j\,b))(a:A)C\,(i\,a),\\
d.e.b.D\,e\,(j\,b)=d.e.b.e\,b & : & \overline{(A:set)(B:set)}\overline{(C:(\sum A\,B)set)}\\
 & & ((a:A)C\,(i\,a))((b:B)C\,(j\,b))(b:B)C\,(j\,b).
\end{array}
$$

Rules for the equality:

$$
\begin{array}{rcl}
I & : & (A:set)(A)(A)set,\\
r & : & \overline{(A:set)}(a:A)I\,A\,a\,a,\\
J & : & \overline{(A:set)(a:A)(b:A)(C:(a:A)(b:A)(I\,A\,a\,b)set)}\\
 & & ((a:A)C\,a\,a\,(r\,a))(c:I\,A\,a\,b)C\,a\,b\,c,\\
d.a.J\,d\,(r\,a)=d.a.d\,a & : & \overline{(A:set)(C:(a:A)(b:A)(I\,A\,a\,b)set)}\\
 & & ((a:A)C\,a\,a\,(r\,a))(a:A)C\,a\,a\,(r\,a).
\end{array}
$$

Rules for finite sets:

$$
\begin{array}{rcl}
N_n & : & set,\\
0_n & : & N_n,\\
 & \vdots & ,\\
(n-1)_n & : & N_n,\\
R_n & : & \overline{(C:(N_n)set)}\\
 & & (C\,0_n)\\
 & & \vdots\\
 & & (C\,(n-1)_n)\\
 & & (c:N_n)\\
 & & C\,c,\\
c_0.\cdots.c_{n-1}.R_n\,c_0\,\cdots\,c_{n-1}\,0_n & & \\
=c_0.\cdots.c_{n-1}.c_0 & : & \overline{(C:(N_n)set)}\\
 & & (C\,0_n)\\
 & & \vdots\\
 & & (C\,(n-1)_n)\\
 & & C\,0_n,\\
 & \vdots & ,\\
c_0.\cdots.c_{n-1}.R_n\,c_0\,\cdots\,c_{n-1}\,(n-1)_n & & \\
=c_0.\cdots.c_{n-1}.c_{n-1} & : & \overline{(C:(N_n)set)}\\
 & & (C\,0_n)\\
 & & \vdots\\
 & & (C\,(n-1)_n)\\
 & & C\,(n-1)_n.
\end{array}
$$

(Note that by letting $n = 0, 1, 2, \ldots$ we get an infinite list of declarations and axioms, and thus not a proper theory in the sense of section 2.1. It is possible to reformulate the rules for finite sets and obtain a finite list of declarations and equations.)

Rules for natural numbers:

$$\begin{array}{rcl}
N & : & \mathit{set},\\
0 & : & N,\\
s & : & (N)N,\\
R & : & \overline{(C : (N)\mathit{set})}\\
 & & (C\,0)((a : N)(C\,a)C\,(s\,a))(c : N)C\,c,\\
d.e.R\,d\,e\,0 = d.e.d & : & \overline{(C : (N)\mathit{set})}\\
 & & (C\,0)((a : N)(C\,a)C\,(s\,a))C\,0,\\
d.e.a.R\,d\,e\,(s\,a) = d.e.a.e\,a\,(R\,d\,e\,a) & : & \overline{(C : (N)\mathit{set})},\\
 & & (C\,0)((a : N)(C\,a)C\,(s\,a))(a : N)C\,(s\,a).
\end{array}$$

Rules for well-orderings:

$$\begin{array}{rcl}
W & : & (A : \mathit{set})((A)\mathit{set})\mathit{set},\\
\mathit{sup} & : & \overline{(A : \mathit{set})(B : (A)\mathit{set})}(a : A)((B\,a)W\,A\,B)W\,A\,B),\\
T & : & \overline{(A : \mathit{set})(B : (A)\mathit{set})(C : (W\,A\,B)\mathit{set})}\\
 & & ((a : A)(b : (B\,a)W\,A\,B)((v : B\,a)C\,(b\,v))C\,(\mathit{sup}\,a\,b))\\
 & & (c : W\,A\,B)\\
 & & C\,c,\\
d.a.b.T\,d\,(\mathit{sup}\,a\,b) & & \\
= d.a.b.d\,a\,b\,((T\,d)\circ b) & : & \overline{(A : \mathit{set})(B : (A)\mathit{set})(C : (W\,A\,B)\mathit{set})}\\
 & & ((a : A)(b : (B\,a)W\,A\,B)((v : B\,a)C\,(b\,v))C\,(\mathit{sup}\,a\,b))\\
 & & (a : A)\\
 & & (b : (B\,a)W\,A\,B)\\
 & & C\,(\mathit{sup}\,a\,b).
\end{array}$$

Rules for the first universe (the formulation 'à la Tarski' [10]):

$$\begin{array}{rcl}
U & : & \mathit{set},\\
S & : & (U)\mathit{set},\\
\pi & : & (a : U)((S\,a)U)U,\\
\sigma & : & (a : U)((S\,a)U)U,\\
+ & : & (U)(U)U,\\
i & : & (a : U)(S\,a)(S\,a)U,\\
n_0 & : & U,\\
n_1 & : & U,\\
 & \vdots & ,\\
n & : & U,
\end{array}$$

$$
\begin{aligned}
w &: (a:U)((S\,a)U)U,\\
a.b.S\,(\pi\,a\,b) = a.b.\prod (S\,a)\,(S\circ b) &: (a:U)((S\,a)U)set,\\
a.b.S\,(\sigma\,a\,b) = a.b.\sum (S\,a)\,(S\circ b) &: (a:U)((S\,a)U)set,\\
a.b.S\,(a+b) = a.b.S\,a + S\,b &: (U)(U)set,\\
a.b.c.S\,(i\,a\,b\,c) = a.b.c.I\,(S\,a)\,b\,c &: (a:U)(S\,a)(S\,a)set,\\
S\,n_0 = N_0 &: set,\\
S\,n_1 = N_1 &: set,\\
&\vdots\;,\\
S\,n = N &: set,\\
a.b.S\,(w\,a\,b) = a.b.W\,(S\,a)\,(S\circ b) &: (a:U)((S\,a)U)set
\end{aligned}
$$

We could then iterate this process and form a second universe, etc, but note the problem with the finiteness of the signature and the axioms.

We also have the following abbreviations (definitions). First we have function application:

$$
Ap = c.a.F\,(b.b\,a)\,c \;:\; \overline{(A:set)(B:(A)set)}(\textstyle\prod AB)(a:A)B\,a
$$

Then we have the left and right projections of a pair:

$$
\begin{aligned}
p = E\,(a.b.a) &: \overline{(A:set)(B:(A)set)}(\textstyle\sum AB)A,\\
q = E\,(a.b.b) &: \overline{(A:set)(B:(A)set)}(c:\textstyle\sum AB)B\,(p\,c)
\end{aligned}
$$

We also have

$$
\begin{aligned}
\rightarrow\, = A.B.\textstyle\prod A\,x.B &: (set)(set)set,\\
\times = A.B.\textstyle\sum A\,x.B &: (set)(set)set.
\end{aligned}
$$

The definitions of the associated constructors and selectors have to be redefined accordingly.

Moreover, we write $a =_A b$, $\sum_{x:A} B\,x$, $\prod_{x:A} B\,x$, and $W_{x:A} B\,x$ instead of $I\,A\,a\,b$, $\sum A\,B$, $\prod A\,B$, and $W\,A\,B$.

3 Adding Fixed Points to Intuitionistic Set Theory

One possibility, suggested by Per Martin-Löf, is to add to intuitionistic set theory a general fixed point operator on the level of sets and a general fixed point operator on the level of elements. Let $G : (set)set$ be a strictly positive set operator. Then we have the following rules:

Fix G-formation:

$$\mathit{Fix}\,G : \mathit{set}$$
$$\mathit{Fix}\,G = \text{[1]}G\,(\mathit{Fix}\,G) : \mathit{set}$$

Fix G-elimination:

$$\begin{aligned} \mathit{fix} \;:\; & \overline{(C : (\mathit{Fix}\,G)\mathit{set})} \\ & (g : (X : \mathit{set})(X \subseteq \mathit{Fix}\,G)\text{[2]}(f : (x : X)C\,x)(y : G\,X)C\,y) \\ & (z : \mathit{Fix}\,G) \\ & C\,z \end{aligned}$$

Fix G-equality:

$$\begin{aligned} \mathit{fix} = g.g\,(\mathit{fix}\,g) \;:\; & \overline{(C : (\mathit{Fix}\,G)\mathit{set})} \\ & (g : (X : \mathit{set})(X \subseteq \mathit{Fix}\,G)(f : (x : X)C\,x)(y : G\,X)C\,y) \\ & (z : \mathit{Fix}\,G) \\ & C\,z \end{aligned}$$

(Similar rules can be found in papers by Constable and Mendler, see [4], [12].)

These rules can be given a semantic justification, informally as in [9], or more formally by an interpretation in a logical theory as suggested by Peter Aczel, see [2], [1], [15]. In order to do this we need to include in the logical theory a new class (or predicate, that is, object of type $(\iota)o$) former ambiguously denoted $\mathit{Fix} : (((\iota)o)(\iota)o)(\iota)o$ (ι is the type of individuals and o is the type of propositions [3] in the logical metalanguage), which may only be applied to strictly positive class operators G, that is, the class variable X may not occur to the left of an implication in $G\,X\,x$. Moreover, we also include a non-canonical program form ambiguously denoted $\mathit{fix} : (((\iota)\iota)(\iota)\iota)(\iota)\iota$, which has the computation rule (in old fashioned notation)

$$\frac{g\,(\mathit{fix}\,g)\,z \Longrightarrow c}{\mathit{fix}\,g\,z \Longrightarrow c}.$$

The idea behind the interpretation is to interpret each set as a class invariant under the computation (or conversion) rules. The equality on the sets is interpreted as convertibility. A strictly positive set operator will be interpreted as a strictly positive class operator. *Fix* and *fix* in intuitionistic set theory will be interpreted as *Fix* and *fix* in the logical theory.

Let us prove informally the correctness of the rules under this interpretation. The formation rules are correct by definition. The elimination rule is proved by induction

[1] This leads to an undecidable equality relation on sets and thus the theory defined is not 'sensible' in the sense of section 2.2. However, this rule does not fit into the notational framework in any other way either.

[2] improper notation meaning that X is a *subtype* of *Fix* G.

[3] We had no type of propositions in the type theory, but we could put $o = \mathit{prop} = \mathit{set}$ by identifying propositions and sets.

on *Fix G*. Let us write the interpreted rule in the ordinary way, using 'set notation' for classes and suppressing some premises.

$$\frac{\begin{array}{c}(X \subseteq \mathit{Fix}\,G, f\,x \in C\,x \quad (x \in X)) \\ g\,f\,y \in C\,y \quad (y \in G\,X)\end{array}}{\mathit{fix}\,g\,z \in C\,z \quad (z \in \mathit{Fix}\,G)}$$

So assume that g satisfies the premise of the rule. Let $P = \{z \in \mathit{Fix}\,G | \mathit{fix}\,g\,z \in C\,z\}$. Our task is then to prove $\mathit{Fix}\,G \subseteq P$. Since $P \subseteq \mathit{Fix}\,G$, and $\mathit{fix}\,g\,z \in C\,z \quad (z \in P)$ we can use the premise to conclude that

$$g\,(\mathit{fix}\,g)\,z \in C\,z \quad (z \in G\,P).$$

But since $\mathit{fix}\,g\,z$ has the same value as $g\,(\mathit{fix}\,g)\,z$ it follows that

$$\mathit{fix}\,g\,z \in C\,z \quad (z \in G\,P).$$

Hence it follows that $\mathit{Fix}\,G \cap G\,P \subseteq P$. But then it follows by induction on $\mathit{Fix}\,G$ that $\mathit{Fix}\,G \subseteq P$.

The equality rule can be proved correct in a similar manner.

4 Deriving Recursion Operators for Inductively Defined Sets

For each inductively defined set there is a principle of primitive (or structural) recursion. For natural numbers we have ordinary primitive recursion and the R-operator of intuitionistic set theory. For lists we have primitive list recursion and the *listrec*-operator. For well-orderings we have transfinite recursion and the T-operator, etc.

We shall now give a method for deriving such recursion operators for a subcollection of the inductively defined set that we defined above. The subcollection is the one obtained by changing the last clause in the definition of strictly positive to the restricted case where $G\,X = K \to X$. Note that this is not a severe restriction; a similar, and in fact stronger, restriction is implicit in Backhouse's method which uses introduction rules to define new sets. If we translate such definitions into definitions in terms of strictly positive operators, this results in having $+$ outermost, then $\times$ (or $\sum$), and $\to$ (or $\prod$) innermost. To find recursion operators for the general case is an open problem. Another algorithm has been proposed by Dag Normann [14]. This algorithm derives recursion operators for for the special case where the new function value is expressed only in terms of the values of the function on the immediate predecessors.

So assume that G is in this subcollection. We wish to determine its n constructors and their types, and hence the introduction rules for *Fix G*. Moreover, we wish to determine the recursion operator and its type, and hence the elimination rule for *Fix G*. From this information the equality rules can also be determined.

The recursion operator and the elimination rule for *Fix G* is obtained by instantiating the general elimination rule of the previous section. We first find a selector (or pattern matching function) sel^G for G with $n+2$ arguments, such that, if we substitute

$sel^G\, d_1 \cdots d_n$ for g then the premise of the general elimination rule for $Fix\, G$ will be satisfied. Thus we can define the recursion operator for $Fix\, G$ as

$$rec^{Fix G}\, d_1\, \cdots\, d_n\, z = fix\, (sel^G\, d_1\, \cdots\, d_n)\, z.$$

For example, if we define the natural numbers as $N = Fix\, X.N_1 + X$, then we get the two constructors

$$\begin{aligned} 0 = i\, 0_1 &\ :\ N, \\ s = j &\ :\ (N)N, \end{aligned}$$

and a recursion operator

$$\begin{aligned} R \ = \ & d_1.d_2.fix\, (f.D\, (R_1\, d_1)\, (x_1.d_2\, x_1\, (f\, x_1))) : \\ & \overline{(C : (N)set)} \\ & (d_1 : C\, 0) \\ & (d_2 : (x_1 : N)(C\, x_1)C\, (s\, x_1)) \\ & (z : N) \\ & C\, z, \end{aligned}$$

which uses that the selector for $X.N_1 + X$ is of the form

$$natcases = d_1.d_2.f.D\, (R_1\, d_1)\, (x_1.d_2\, x_1\, (f\, x_1)).$$

4.1 Finitary induction

The problem of finding a recursion operator for $Fix\, G$ is thus reduced to that of finding a selector for G. It will turn out to be convenient to define the constructors for G at the same time.

We shall begin with a case of finitary induction, where $G\, X$ is built up from the finite sets N_n and the variable X by binary sums $+$ and binary products $\times$.

A k-ary *constructor* for G is a function

$$con^G : \overline{(X : set)}\underbrace{(X) \cdots (X)}_{k} G\, X,$$

which is built up from the constructors of the standard operators, that is, m_n, i, j, $\langle \_, \_ \rangle$.

Note that if we let $X = Fix\, G$ and thus get $G\, X = Fix\, G$, then we get the constructors and introduction rules for $Fix\, G$.

If G has n constructors with arities $k_1, \ldots, k_n$ respectively, then a *selector* for G is a function

$$\begin{aligned} sel^G \ :\ & \overline{(X : set)(C : (X)set)(C' : (G\, X)set)} \\ & (d_1 : (x_1 : X)(C\, x_1) \cdots (x_{k_1} : X)(C\, x_{k_1})C'\, (con_1\, x_1\, \cdots\, x_{k_1})) \\ & \vdots \\ & (d_n : (x_1 : X)(C\, x_1) \cdots (x_{k_n} : X)(C\, x_{k_n})C'\, (con_n\, x_1\, \cdots\, x_{k_n})) \\ & (f : (x : X)C\, x) \\ & (y : G\, X) \\ & C'\, y \end{aligned}$$

which is built up from the selectors for the standard operators, that is, R_n, D and E.

Note that if we let $X \subseteq \mathit{Fix}\, G$ and thus (since G is monotone) get $G\,X \subseteq \mathit{Fix}\, G$ and hence also can have $C' = C : (\mathit{Fix}\, G)set$, then we get the recursion operator

$$\begin{array}{l}
rec^{\mathit{Fix}\, G} = d_1.\cdots.d_n.\mathit{fix}\,(sel^G\, d_1\ \cdots\ d_n) : \\
\overline{(C : (\mathit{Fix}\, G)set)} \\
(d_1 : (x_1 : \mathit{Fix}\, G)(C\, x_1)\cdots(x_{k_1} : \mathit{Fix}\, G)(C\, x_{k_1})C\,(con_1\, x_1\ \cdots\ x_{k_1})) \\
\vdots \\
(d_n : (x_1 : \mathit{Fix}\, G)(C\, x_1)\cdots(x_{k_n} : \mathit{Fix}\, G)(C\, x_{k_n})C\,(con_n\, x_1\ \cdots\ x_{k_n})) \\
(z : \mathit{Fix}\, G) \\
C\, z.
\end{array}$$

We begin with the base case $G = X.N_n$. There are n constructors, all 0-ary:

$$\begin{array}{rcl}
0_n & : & \overline{(X : set)}N_n, \\
 & \vdots & , \\
(n-1)_n & : & \overline{(X : set)}N_n.
\end{array}$$

The selector is

$$\begin{array}{rcl}
d_1.\cdots.d_n.f.R_n\, d_1\ \cdots\ d_n & : & \overline{(X : set)(C : (X)set)(C' : (N_n)set)} \\
 & & (d_1 : C'\, 0_n) \\
 & & \vdots \\
 & & (d_n : C'\,(n-1)_n) \\
 & & (f : (x : X)C\, x) \\
 & & (y : N_n) \\
 & & C'\, y.
\end{array}$$

Note the similarities and differences between the constructors and the selector here, for the case where $X.N_n : (set)set$ is a constant function, and the ordinary constructors 0_n, ..., $(n-1)_n$ and the ordinary selector R_n, for the case where $N_n : set$ is a constant set.

$G = X.X$. There is one constructor

$$x_1.x_1 \ :\ \overline{(X : set)}(X)X,$$

and the selector

$$\begin{array}{rcl}
d_1.f.y.d_1\, y\,(f\, y) & : & \overline{(X : set)(C : (X)set)(C' : (G\,X)set)} \\
 & & (d_1 : (x_1 : X)(C\, x_1)C'\, x_1) \\
 & & (f : (x : X)C\, x) \\
 & & (y : X) \\
 & & C'\, y.
\end{array}$$

For the remaining two cases assume the induction hypothesis that G' has n' constructors $con'_1, \ldots, con'_{n'}$ of arities $k'_1, \ldots, k'_{n'}$, respectively, and the selector sel'; and that G'' has n'' constructors $con''_1, \ldots, con''_{n''}$ of arities $k''_1, \ldots, k''_{n''}$, respectively, and the selector sel''.

$G = X.G'X + G''X$. Then G has the $n' + n''$ constructors

$$\begin{aligned}
x'_1.\cdots.x'_{k'_1}.i(con'_1\, x'_1 \cdots x'_{k'_1}) &: \overline{(X : set)}(X)\cdots(X)G'X + G''X,\\
&\vdots,\\
x'_1.\cdots.x'_{k'_{n'}}.i(con'_{n'}\, x'_1 \cdots x'_{k'_{n'}}) &: \overline{(X : set)}(X)\cdots(X)G'X + G''X,\\
x''_1.\cdots.x''_{k''_1}.j(con''_1\, x''_1 \cdots x''_{k''_1}) &: \overline{(X : set)}(X)\cdots(X)G'X + G''X,\\
&\vdots,\\
x''_1.\cdots.x''_{k''_{n''}}.j(con''_{n''}\, x''_1 \cdots x''_{k''_{n''}}) &: \overline{(X : set)}(X)\cdots(X)G'X + G''X,
\end{aligned}$$

and the selector

$$\begin{aligned}
&d'_1.\cdots.d'_{n'}.d''_1.\cdots.d''_{n''}.f.D\,(sel'\, d'_1 \cdots d'_{n'}\, f)(sel''\, d''_1 \cdots d''_{n''}\, f) :\\
&\overline{(X : set)(C : (X)set)(C' : (G'X + G''X)set)}\\
&(d'_1 : (x'_1 : X)(C\,x'_1)\cdots(x'_{k'_1} : X)(C\,x'_{k'_1})C'(i\,(con'_1\, x'_1 \cdots x'_{k'_1})))\\
&\vdots\\
&(d'_{n'} : (x'_1 : X)(C\,x'_1)\cdots(x'_{k'_{n'}} : X)(C\,x'_{k'_{n'}})C'(i\,(con'_{n'}\, x'_1 \cdots x'_{k'_{n'}})))\\
&(d''_1 : (x''_1 : X)(C\,x''_1)\cdots(x''_{k''_1} : X)(C\,x''_{k''_1})C'(j\,(con''_1\, x''_1 \cdots x''_{k''_1})))\\
&\vdots\\
&(d''_{n''} : (x''_1 : X)(C\,x''_1)\cdots(x''_{k''_{n''}} : X)(C\,x''_{k''_{n''}})C'(j\,(con''_{n''}\, x''_1 \cdots x''_{k''_{n''}})))\\
&(f : (x : X)C\,x)\\
&(y : G'X + G''X)\\
&C'\,y.
\end{aligned}$$

$G = X.G'X \times G''X$. Then G has the $n' \times n''$ constructors

$$x'_1.\cdots.x'_{k'_1}.x''_1.\cdots.x''_{k''_1}.\langle con'_1\, x'_1 \cdots x'_{k'_1}, con''_1\, x''_1 \cdots x''_{k''_1}\rangle,$$

$$\ddots$$

$$\vdots \qquad x'_1.\cdots.x'_{k'_1}.x''_1.\cdots.x''_{k''_{n''}}.\langle con'_1\, x'_1 \cdots x'_{k'_1}, con''_{n''}\, x''_1 \cdots x''_{k''_{n''}}\rangle,$$

$$x'_1.\cdots.x'_{k'_{n'}}.x''_1.\cdots.x''_{k''_1}.\langle con'_{n'}\, x'_1 \cdots x'_{k'_{n'}}, con''_1\, x''_1 \cdots x''_{k''_1}\rangle,$$

$$\ddots$$

$$x'_1.\cdots.x'_{k'_{n'}}.x''_1.\cdots.x''_{k''_{n''}}.\langle con'_{n'}\, x'_1 \cdots x'_{k'_{n'}}, con''_{n''}\, x''_1 \cdots x''_{k''_{n''}}\rangle,$$

and the selector

$$\begin{array}{ccc}
 & d_{11}. \quad \cdots \quad d_{1n''}. & \\
 & \vdots \qquad\qquad \vdots & \\
 & d_{n'1}. \quad \cdots \quad d_{n'n''}. & \\
 & f.y.E\,(y'.y''.sel' & \\
(x_1'.z_1'.\cdots.x_{k_1'}'.z_{k_1'}'.sel''\,(d_{11}\,x_1'\,z_1'\cdots x_{k_1'}'\,z_{k_1'}') & \cdots & (d_{1n''}\,x_1'\,z_1'\cdots x_{k_1'}'\,z_{k_1'}')\,f\,y'') \\
\vdots & & \vdots \\
(x_1'.z_1'.\cdots.x_{k_{n'}'}'.z_{k_{n'}'}'.sel''\,(d_{n'1}\,x_1'\,z_1'\cdots x_{k_{n'}'}'\,z_{k_{n'}'}') & \cdots & (d_{n'n''}\,x_{n''}'\,x_1'\,z_1'\cdots x_{k_{n'}'}'\,z_{k_{n'}'}')\,f\,y'') \\
 & f\,y')y &
\end{array}$$

The types of the constructors and the selector have been omitted for lack of space.

One can now check that this algorithm indeed gives the constructors and the selector for the natural numbers as above.

4.2 Parameterised sets

Some minor extensions to the schemes are needed for parameterised inductively defined sets (or data types). Important examples of such are lists, which are generated by the operator $X.N_1 + A \times X$, and binary trees, which are generated by the operator $X.A + X \times X$, where in both cases A is a parameter.

We can view these generators as functions of the parameters as well. If we extend the notion of a constructor to cope with set operators of arbitrary arity, then we get the following constructors for $A.X.N_1 + A \times X$:

$$\begin{aligned}
nil = i\,0_1 \quad &: \quad \overline{(A : set)(X : set)}N_1 + A \times X, \\
cons = a.x.j\,\langle a, x\rangle \quad &: \quad \overline{(A : set)(X : set)}(A)(X)N_1 + A \times X.
\end{aligned}$$

The notion of a selector can also be modified to a corresponding notion for set operators of arbitrary arity. One of the arguments, the X, is singled out for the recursion. Thus, the selector for $A.X.N_1 + A \times X$ is,

$$\begin{array}{l}
listcases = d_1.d_2.f.D\,(R_1\,d_1)\,(E\,(a_1.x_1.d_2\,a_1\,x_1\,(f\,x_1))) : \\
\overline{(A : set)(X : set)(C : (X)set)(C' : (N_1 + A \times X)set)} \\
(d_1 : C'\,(nil)) \\
(d_2 : (a_1 : A)(x_1 : X)(C\,x_1)C'\,(cons\,a_1\,x_1)) \\
(f : (x : X)C\,x) \\
(y : N_1 + A \times X) \\
C'\,y
\end{array}$$

The binary trees generated by $A.X.A + X \times X$ are treated in a similar way. We get the following two constructors:

$$\begin{aligned}
leaf = i \quad &: \quad \overline{(A : set)(X : set)}A + X \times X, \\
treecons = x_1.x_2.j\,\langle x_1, x_2\rangle \quad &: \quad \overline{(A : set)(X : set)}(A)(X)A + X \times X.
\end{aligned}$$

The selector is

$$\begin{array}{l}
\mathit{treecases} = d_1.d_2.f.D\, d_1\,(E\,(x_1.x_2.d_2\, x_1\,(f\, x_1)\, x_2\,(f\, x_2))) : \\
\overline{(A : \mathit{set})(X : \mathit{set})(C : (X)\mathit{set})(C' : (A + X \times X)\mathit{set})} \\
(d_1 : (a : A)C'(\mathit{leaf}\, a)) \\
(d_2 : (x_1 : X)(C\, x_1)(x_2 : X)(C\, x_2)C'(\mathit{treecons}\, x_1\, x_2)) \\
(f : (x : X)C\, x) \\
(y : A + X \times X) \\
C'\, y.
\end{array}$$

I have already presented some longwinded descriptions of how to form constructors and selectors for the case without parameters, and shall only show the new base case for a parameter $A.X.A$. Then the one constructor is

$$a.a : \overline{(A : \mathit{set})(X : \mathit{set})}(A)A$$

and the selector is

$$\begin{array}{rcl}
d_1.f.y.d_1\, y & : & \overline{(A : \mathit{set})(X : \mathit{set})(C : (X)\mathit{set})(C' : (A)\mathit{set})} \\
& & (d_1 : (a_1 : X)(C\, a_1)C'\, a_1) \\
& & (f : (x : X)C\, x) \\
& & (y : A) \\
& & C'\, y.
\end{array}$$

Note that the only difference between the variable and parameter base cases is that the selector does not depend on $f\, y$ in the parameter case.

4.3 Infinitary induction

Also here the scheme needs to be modified to account for the case where G contains $\rightarrow$ innermost. We need a new base case when $X.K \rightarrow X$ for some constant $K : \mathit{set}$. For this case we have one constructor

$$\lambda : \overline{(X : \mathit{set})}((K)X)K \rightarrow X$$

and the selector

$$\begin{array}{rcl}
d_1.f.F\,(w.d_1\, w\,(f \circ w)) & : & \overline{(X : \mathit{set})(C : (X)\mathit{set})(C' : (K \rightarrow X)\mathit{set})} \\
& & (d_1 : (w_1 : (K)X)((k : K)C\,(w_1\, k))C'(\lambda w_1)) \\
& & (f : (x : X)C\, x) \\
& & (y : K \rightarrow X) \\
& & C'\, y.
\end{array}$$

Again, to get a complete account we need to modify the other cases in a uniform manner.

As an example we consider the ordinals of the second number class, which are generated by $X.N_1 + N \to X$. We get the following constructors:

$$\begin{aligned} 0 = i\,0_1 &\;:\; \overline{(X : set)}N_1 + N \to X, \\ sup = w.j\,(\lambda\,w) &\;:\; \overline{(X : set)}((N)X)N_1 + N \to X. \end{aligned}$$

The selector is

$$\begin{array}{l} \mathit{ordcases} = d_1.d_2.f.D\,(R_1\,d_1)\,(F\,(w.d_2\,w\,(f \circ w))) : \\ \overline{(X : set)(C : (X)set)(C' : (N_1 + N \to X)set)} \\ (d_1 : C'(0)) \\ (d_2 : (w : (N)X)((n : N)C\,(w\,n))C'\,(sup\,w)) \\ (f : (x : X)C\,x) \\ (y : N_1 + N \to X) \\ C'\,y. \end{array}$$

As a final example we look at the well-orderings generated by $X.\sum_{x:A}(B\,x \to X)$. The rules for $\sum$ are just a straightforward modification of the rules for ×. We get the constructor

$$sup = a.w.\langle a, \lambda\,w\rangle : \overline{(A : set)(B : (A)set)(X : set)}\sum_{x:A}(B\,x \to X),$$

and the selector

$$\begin{array}{l} \mathit{transcases} = d.f.E\,(y'.y''.F\,(w.d\,y'\,w\,(f \circ w))y'') : \\ \overline{(A : set)(B : (A)set)(X : set)(C : (X)set)(C' : (\sum_{x:A}(B\,x \to X))set)} \\ (d : (a : A)(w : (B\,a)X)((k : B\,a)C\,(w\,k))C'(sup\,a\,w)) \\ (y : (\sum_{x:A}(B\,x \to X)) \\ C'\,y. \end{array}$$

5 Representing Inductively Defined Sets as Well-Orderings

The well-orderings in intuitionistic set theory are themselves introduced by an inductive definition. As we saw in the last section, a version of $W_{x:A}B\,x$ is obtained by taking the least fixed point of the set operator

$$X.\sum_{x:A}(B\,x \to X).$$

Natural numbers can then be represented by $W_{x:N_2}B_N\,x$, where $B_N\,0_2 = N_0$ and $B_N\,1_2 = N_1$. The ordinals of the second number class can be represented by $W_{x:N_2}B_O\,x$, where $B_O\,0_2 = N_0$, and $B_O\,1_2 = N$. (Both examples are from [10].)

In what sense are these representations correct? A reasonable idea seems to be that the representation is isomorphic to the primitive set. For example,

$$W_{x:N_2} B_N\, x \cong N,$$

where isomorphism of sets is defined in the obvious way, that is, by

$$\begin{array}{rl} & A.B.A \cong B \\ = & A.B.(\exists f : A \rightarrow B)(\exists g : B \rightarrow A) \\ & ((\forall x : A)(Ap\, g\, (Ap\, f\, x)) =_A x) \wedge (\forall y : B)(Ap\, f\, (Ap\, g\, y)) =_B y)). \end{array}$$

The suggested correspondence is between 0 and $sup\, 0_2\, (\lambda\, R_0)$, and between $s\, a$ and $sup\, 1_2\, (\lambda\, x.a')$, where a corresponds to a'. However, the fact that this correspondence determines an isomorphism depends on the extensional equality of functions of [9] and [10]. Every function in $N_0 \rightarrow X$ is equal to $\lambda\, R_0$, that is,

$$(\forall f : N_0 \rightarrow X)(f =_{N_0 \rightarrow X} \lambda\, R_0),$$

provided the equality is extensional. However, if equality is intensional, as in [8], then this proposition can no longer be proved.

The two representations above are more or less instances of a general pattern, and assuming extensional equality in the definition of isomorphism, we have the following theorem.

Theorem 1 *For any strictly positive set operator G, we can find an A : set and a family $B : (A)set$, such that, if X : set, then*

$$G\, X \cong \sum_{x:A} (B\, x \rightarrow X).$$

Proof. Associate with each strictly positive G an A : *set* and a family $B : (A)set$ as follows:

- G is a constant operator $X.K$. Then $A = K$ and $B = x.N_0$.
- G is the identity operator $X.X$. Then $A = N_1$ and $B = x.N_1$.
- $G = X.G'X + G''X$ for strictly positive G' and G'' with associated A', B', and A'', B'' respectively. Then $A = A' + A''$ and B is such that $B \circ i = B' : (A')set$ and $B \circ j = B'' : (A'')set$.
- $G = X.G'X \times G''X$ for strictly positive G' and G'' as above. Then $A = A' \times A''$ and B is such that $x'.x''.B\, \langle x', x'' \rangle = x'.x''.B'x' + B''x'' : (A')(A'')set$
- $G = X.K \rightarrow G'X$ for a constant set K and G' as above. Then $A = K \rightarrow A'$ and $B = f.\sum_{y:K} B'(Ap\, f\, y) : (K \rightarrow A')set$.

First we state some useful isomorphisms.

$$\begin{aligned}
N_1 &\cong N_0 \to A, && (1)\\
A &\cong N_1 \to A, && (2)\\
A &\cong A \times N_1, && (3)\\
A &\cong N_1 \times A, && (4)\\
\prod_{x:A} \sum_{y:B\,x} C\,x\,y &\cong \sum_{f:\prod_{x:A} B\,x} \prod_{x:A} C\,x\,(Ap\,f\,x), && (5)\\
\prod_{x:A} \prod_{y:B\,x} C\,x\,y &\cong \prod_{z:\sum_{x:A} B\,x} C\,(p\,z)\,(q\,z), && (6)\\
\sum_{x':A'} B'\,x' + \sum_{x'':A''} B''\,x'' &\cong \sum_{x:A'+A''} B\,x. && (7)
\end{aligned}$$

where, in the last isomorphism B is such that $B \circ i = B' : (A')set$ and $B \circ j = B'' : (A'')set$.

By using these isomorphisms we can prove the theorem by induction on the structure of G. We have the following cases:

- G is a constant $X.K$. We need to show that

  $$K \cong K \times (N_0 \to X).$$

  (Recall that $A \times B = \sum_{x:A} B$.) But this follows directly from (3) and (1).

- G is the identity $X.X$. We need to show that

  $$X \cong N_1 \times (N_1 \to X).$$

  But this follows directly from (4) and (2).

- $G = X.G'X + G''X$. We need to show that

  $$\sum_{x':A'} (B'\,x' \to X) + \sum_{x'':A''} (B''\,x'' \to X) \cong \sum_{x:A'+A''} (B\,x \to X),$$

  where $B \circ i = B' : (A')set$ and $B \circ j = B'' : (A'')set$. But this follows directly from (7).

- $G = X.G'X \times G''X$. We need to show that

  $$\sum_{x':A'} (B'\,x' \to X) \times \sum_{x'':A''} (B''\,x'' \to X) \cong \sum_{x:A' \times A''} (B\,x \to X),$$

  where $x'.x''.B\,\langle x', x''\rangle = x'.x''.B'\,x' + B''\,x'' : (A')(A'')set$. A typical (canonical) element of the LHS has the form $\langle\langle a', f'\rangle, \langle a'', f''\rangle\rangle$, where $a' : A'$, $f' : B'\,a' \to X$, $a'' : A''$, and $f'' : B''\,a'' \to X$. A typical (canonical) element of the RHS has the form $\langle\langle a', a''\rangle, f\rangle$, where $a' : A'$, $a'' : A''$, and $f : (B'\,a' + B''\,a'') \to X$. Thus we get the isomorphisms

  $$\begin{gathered}
  \lambda x.\langle\langle p\,(p\,x), p\,(q\,x)\rangle, \lambda(D\,(Ap\,(q\,(p\,x)))\,(Ap\,(q\,(q\,x))))\rangle :\\
  \sum_{x':A'} (B'\,x' \to X) \times \sum_{x'':A''} (B''\,x'' \to X) \to \sum_{x:A' \times A''} (B\,x \to X)
  \end{gathered}$$

and

$$\lambda y.\langle\langle p\,(p\,y), \lambda\,((Ap\,(q\,y)) \circ i)\rangle, \langle q\,(p\,y), \lambda\,((Ap\,(q\,y)) \circ j)\rangle\rangle : \\ \sum_{x:A'\times A''}(B\,x \to X) \to \sum_{x':A'}(B'\,x' \to X) \times \sum_{x'':A''}(B''\,x'' \to X).$$

- $G = X.K \to G'\,X$ for a constant set K and G' as above. We get

$$\begin{aligned} K \to \sum_{x':A'}(B'\,x' \to X) &\cong \sum_{f:K\to A'}\prod_{y:K}(B'\,(Ap\,f\,y) \to X) \\ &\cong \sum_{f:K\to A'}(\sum_{y:K} B'\,(Ap\,f\,y) \to X) \end{aligned}$$

by using special cases of (5) and (6) respectively. □

There seems to be no difficulty in extending this theorem to include the cases where G contains $\sum$ and $\prod$.

We can now apply the algorithm to $X.N_1 + X$ and get a representation of the natural numbers as $W_{x:N_1+N_1}B\,x$, where $B\,(i\,0_1) = N_0$ and $B\,(j\,0_1) = N_1$, which is very similar to the representation from [10] given above.

References

[1] P. Aczel. Frege structures and the notions of proposition, truth and set. In *The Kleene Symposium*, pages 31–59, North-Holland, 1980.

[2] P. Aczel. The strength of Martin-Löf's type theory with one universe. In *Proceedings of the Symposium on Mathematical Logic, Oulu, 1974*, pages 1–32, Report No 2, Department of Philosophy, University of Helsinki, 1977.

[3] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Proceedings of the Workshop on General Logic, Edinburgh*, Laboratory for the Foundations of Computer Science, University of Edinburgh, February 1987.

[4] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In *Proceedings of the LICS-Conference, Brooklyn, N.Y., Lecture Notes in Computer Science*, Springer-Verlag, June 1985.

[5] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra, Volume 1: Invited Lectures*, pages 151–184, Springer-Verlag, LNCS 203, 1985.

[6] N. G. de Bruijn. A survey of the project AUTOMATH. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and formalism*, pages 589–606, Academic Press, 1980.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of the 1987 Logic in Computer Science Conference*, June 1987. To appear.

[8] P. Martin-Löf. Amendment to intuitionistic type theory. March 1986. Notes from a lecture given in Göteborg.

[9] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175, North-Holland, 1982.

[10] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[11] P. Martin-Löf. The logic of judgements. February 1987. Notes from a lecture given at the Workshop on General Logic, Edinburgh.

[12] N. P. Mendler. *First- and Second-Order Lambda Calculi with Recursive Types.* Technical Report TR-86-764, Department of Computer Science, Cornell University, Ithaca, N.Y., July 1986.

[13] B. Nordström. Multilevel functions in Martin-Löf's type theory. In N. Jones, editor, *Programs as Data Objects*, pages 206–221, Springer-Verlag, LNCS 217, October 1985.

[14] D. Normann. Inductively and recursively defined types. 1987. A seminar report, Department of Mathematics, University of Oslo.

[15] J. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.