Constructive natural deduction and its 'ω-set' interpretation

Giuseppe Longo and Eugenio Moggi

Mathematical Structures in Computer Science / Volume 1 / Issue 02 / July 1991, pp 215 - 254
DOI: 10.1017/S0960129500001298, Published online: 04 March 2009

Math. Struct. in Comp. Science (1991), *vol.* 1, *pp.* 215–254

Constructive natural deduction and its 'ω-set' interpretation†

GIUSEPPE LONGO‡§ and EUGENIO MOGGI¶

§ *Laboratoire d'Informatique, CNRS Ecole Normale Superieure, Paris, France*
¶ *LFCS, Computer Science Dept., University of Edinburgh, Edinburgh, UK*

Received 28 May, 1990; revised 24 October 1990

Dedicato a Carlo nel suo settantaseiesimo compleanno

Various Theories of Types are introduced, by stressing the analogy 'propositions-as-types': from propositional to higher order types (and Logic). In accordance with this, proofs are described as terms of various calculi, in particular of polymorphic (second order) λ-calculus. A semantic explanation is then given by interpreting individual types and the collection of all types in two simple categories built out of the natural numbers (the modest sets and the universe of ω-sets). The first part of this paper (syntax) may be viewed as a short tutorial with a constructive understanding of the deduction theorem and some work on the expressive power of first and second order quantification. Also in the second part (semantics, §§6–7) the presentation is meant to be elementary, even though we introduce some new facts on types as quotient sets in order to interpret 'explicit polymorphism'. (The experienced reader in Type Theory may directly go, at first reading, to §§6–8).

0. Remarks on 'mathematical semantics'

The mathematical relevance of some recent work on the semantics of Programming Language suggests the need for some general preliminary observations on the role of the mathematical investigation of linguistic concepts in computer science.

It is worth recalling that formal notions in mathematics came after, in time and intellectual attitude, the work concerning concrete mathematical structures, possibly intended as entities of an independent reality. Namely, only by the end of last century and mainly as the result of the work of Frege and Hilbert, an attempt to achieve a 'complete' formalization of mathematical thinking gained some relevance. Even nowadays working mathematicians think in terms, more or less platonistic, of concrete structures: sets or algebraic or geometric spaces of various kinds.

Following this pattern, the vast majority of mathematical training begins by first giving the concrete feeling of structures and extensional operations on them, followed, sometimes, by some general issues on axiomatization and provability. Rarely is a detailed formalization and proof theoretic investigation of the systems, which have been introduced, carried out.

† Research supported in part by the Joint Collaboration Contract # ST2J-0374 – C (EDB) of the European Economic Community. The original part of this paper is based on an invited lecture delivered at the workshop 'Semantics of Natural and Computer Languages' Half-Moon-Bay (Stanford), March 1987.

‡ On leave from Dip. di Informatica, Università di Pisa. This author's work has been made possible also by the generous hospitality of the Computer Science Dept. of Carnegie Mellon University, while teaching in this University during the academic year 1987/88, and by the support of the French CNRS during summer 1990.

This intellectual itinerary, when it does not lead to some deep misunderstanding of the role of mathematical logic, is sufficiently fair and productive: it seems to help the formation of the intuition and 'almost visual' insight into structures that most good mathematicians have.

However, a major event has taken place in the last forty years: the birth of computer science. Computer scientists first think in terms of formal languages and effective, finitistically describable, procedures; then, sometimes, they may try to add meaning to these over more or less traditional mathematical structures: sets, algebras, functions *in extenso*. The vast majority of computer science courses first introduce a programming language (or the theory of formal or programming languages) and algorithms (or their theory), then give some hints on how these abstract notions can be understood in mathematical terms. We may say that, in this respect, computer science reversed the prevailing gnoseological paradigm of mathematics: first formal notions, then their (intuitive) interpretation.

The reasons for this are clear to everybody: computers are rather stupid fellows and only deal with operations formally described in full detail. It requires a great deal of intelligence and work to have them accomplish the smallest task. For this purpose, the formal notions one essentially deals with in computing are, first, the programming languages and the algorithms, required for the computer–man interaction; then the formal systems aimed at the investigation of languages, algorithms and their properties, as well as at their automatic synthesis. It is clear that these latter aspects form a crucial research area: Type Theory (and this paper) is part of it.

This inversion of methodological approach, i.e. first formalisms then meaning, which started with computer science, turned mathematical logic into an applied sector of mathematics and brought it into the limelight. Indeed, the crucial distinction between syntax and semantics (Frege's 'pure calculus of signs' and Tarski set-theoretic interpretation) is directly inherited from logic into the theory of programming languages. The same evolution occurred in the investigation of effectiveness, provability and feasibility.

Of course, this phenomenon had two immediate consequences: it greatly enlarged the audience of interested researchers and it raised new, unforeseen, problems and issues.

The main system presented below, for example, was first invented, for purely foundational purposes, by Girard, Troelstra and Martin-Löf (see references). These authors were mostly interested in constructive approaches to mathematics and its formalization. Nowadays, since the results of De Bruijn, Reynolds, Burstall and Lampson, Coquand and Huet, the groups in Göteborgh and Cornell and many others, most of the relevant work in this field is carried on within the computer science community, for the purposes of writing (possibly correct) software or implementing deduction (see also Scott (1970) for some early work).

One of the points of formalization, as intended in computer science, is the finitistic nature of the methods involved. The Hilbertian approach to formal systems is surely the main heritage, often corrected by more or less mild versions of Brower's constructivism. The systems to be considered will entirely refer to a constructive approach to formalization and

they are based on the 'propositions-as-types' paradigm, which is the current motto of Type Theory.

The discussion though is open as to how the relevant aspects of Type Theory should be given meaning and made understandable by translating them into possibly independent mathematical constructions.

Martin-Löf and his followers (even more so) insist on providing strictly constructive explanations of Type Theory, based on the solid grounds of a strictly finitistic approach to Mathematics: only finite or finitistically defined structures are allowed, functions are just effective procedures in some formalized language .... This is clearly expressed in Nördstrom (1986).

Indeed, the strict interaction between meaning and denotation, both growing out of a proposal for a constructive foundation of mathematics, suggested at several stages new systems or variations of previous ones, all embedded into a unifying philosophical perspective.

A further motivation for this understanding of types and terms goes back to the same observation which makes formal systems for computations refer to constructive approaches to logic: computers, which are finite machines, run (finite) programs for computable functions, thus any programming construct should be explained in these terms. This seems less convincing, as the meaning of software should also be provided to human beings, who may (fortunately) have a variety of approaches to life and understanding. As long as our brains will not be entirely computerized, the understanding of a strictly constructive, but complicated system may also derive from highly non-constructive, but conceptually simple, intellectual experiences.

The point is that programming languages are getting almost as complicated as real life and the more approaches we have to their semantics the better we may hope to deal with their unexpected riches. Thus, everybody must welcome the fruitful direct interaction between denotation and meaning provided by finitistic view points in semantics, but one should also appreciate the deep insight given by the limit or ideal objects in mathematics, particularly when they aid in understanding the complicated stepwise constructions of automated computing. Functions *in extenso* do not need to exist, but everybody understands by a one second sketch on a blackboard the asymptotic behaviour of the logarithmic function, say, much better than by looking at the effective generation of a hundred thousand outputs. This is simply because our geometric intuition is based on the most solid grounds of over two thousand years of mathematical thinking, since Nemecmo.

Similarly, the sketchy design of a partially ordered set or of a tree, even with limit points, provides an excellent intuition of what is meant by the notion of approximation or by order completeness and a secure guidance to descriptions by constructive formalizations. On the other hand, when a formal system is given and the issue of its 'meaning' is raised, a most informative semantics may easily come from approaches which are based on entirely different grounds. Relevant scientific explanations are precisely provided when apparently unrelated systems are brought together and unexpected links are suggested. By this, semantic explanations in terms of topologies or categories enrich our knowledge in both the source and target areas, guarantee the relative consistency of complicated formal systems

and suggest extensions or even the design of original systems. Indeed, this is the story of the most relevant applications of Scott–Strachey denotational semantics, to which, for example, the design of Edinburgh language ML is indebted (see Plotkin, 1977; Milner, 1978), and it seems to be the intellectual itinerary which brought Girard from system F to Linear Logic as a growing system.

It should be clear that the previously considered perspective gives a major role to informal explanations and intuitive meaning, possibly based on a vaguely defined 'mathematical insight' into structures (topological, algebraic ...). However, this customary informality of mathematics always hides complete rigour and restricts constructions and understanding to the limited cage of mathematical formalism: lack of ambiguities and precision in meaning is the core of it. In other words, it is intended that any informal understanding can be made very precise, if time allows. The formal systems and their semantics below is a typical example of this method and of the forceful formalist environment.

1. Minimal deductions

The general idea on which Gentzen–Prawitz systems are based is the concept of **derivation** as a formalization of the mathematical practice of theorem proving by stepwise deductions. The 'atomic' step in a derivation is given by the application of a **deduction rule**. Each inference rule describes the deduction of certain consequences from given premises. The premises may be formulae, which are assumed, or deductions, i.e. compositions of inference rules. Thus they may have the structure:

$$\frac{A_1\ A_2\ \ A_n}{C} \qquad \frac{\begin{array}{c}A_1\ A_2\ \ A_n\\ \vdots\\ B\end{array}}{C}.$$

In the latter case a fundamental concept may be used, the concept of 'cancellation'. To explain it, assume that you have deduced that if it rains, then it is humid. In a semiformal way, this lets you infer 'rain → humidity'. Now, the latter implication is 'true' independently of the truth of the assumption 'it rains'; namely, as an implication, it holds even when you mention it as an obvious meteorological remark on a sunny day. Thus, you may omit or cancel 'rain' when inferring 'rain → humidity'.

We first discuss one of the simplest deductive systems, the **Positive Natural calculus** ($\mathbf{PN}_0$), as a basis and paradigm for stronger logical extensions. $\mathbf{PN}_0$ is also known as Minimal Logic. $\mathbf{PN}_0$ is only based on implication ('→') as a key concept in deductions. Formulae are inductively defined as follows.

An **atomic predicate** is a formula.
If A and B are formulae, then $(A \to B)$ is a formula.

The inference rules introduce or eliminate '→' from formulae and, thus, tell us how the only connective is formally handled.

Introduction rule	**Elimination rule**
$(\to \mathrm{I})\ \dfrac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \to B}$	$(\to \mathrm{E})\ \dfrac{A \quad A \to B}{B}$.

In $(\to \mathrm{I})$, A is 'cancelled'.

We are now in the position to formally define derivations as trees. The rigorous reader may view the definitions of inference rule and of derivation as given by combined induction.

1.1. Definition. (1) The one element tree $\{A\}$ is a **derivation** for all predicates A.

$(2 \to)$ If $\begin{array}{c}A\\ \vdots\\ B\end{array}$ is a derivation, then $\dfrac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \to B}$ is a derivation.

If $\begin{array}{c}D\\ A\end{array}$ and $\begin{array}{c}D'\\ A \to B\end{array}$ are derivations, then $\dfrac{\begin{array}{cc}D & D'\\ A & A \to B\end{array}}{B}$ is a derivation.

The root or bottom formula of a derivation is the **conclusion**. In this natural deduction system, the provability relation $\Gamma \vdash A$ between lists of formulae and formulae is defined as follows.

1.2. Definition. $\Gamma \vdash A$ if there is a derivation, according to the previously given rules, with all (uncancelled) hypotheses included in Γ and with conclusion A.

If $\Gamma \vdash A$, we say that A is **derivable** from Γ. Note that, by definition, Γ may contain several copies of the same formula and superfluous 'hypotheses', i.e. if $\Gamma \supseteq \Delta \vdash A$, then $\Gamma \vdash A$. Γ, A stands for $\Gamma \cup \{A\}$. If $\Gamma \varnothing$, we write $\vdash A$ and say that A is a **theorem**. Sometimes, the system above is equivalently presented by defining directly $\Gamma \vdash A$, that is by

$$(A)\ \frac{}{\Gamma \vdash A} \quad \text{for } A \text{ in } \Gamma$$

$$(\to \mathrm{I})\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad (\to \mathrm{E})\ \frac{\Gamma \vdash A \quad \Gamma \vdash A \to B}{\Gamma \vdash B}.$$

2. Constructive $\mathbf{PN_0}$

We shall soon see the relevance for functional languages for the very weak system $\mathbf{PN_0}$. $\mathbf{PN_0}$ though is just the implicative fragment of the propositional part of a well known logical system: intuitionistic logic. This logic was formalized by Heyting in the 30s as a logical basis for Brouwer's approach to mathematics. In Brouwer's view, only the subjective intuition of the stepwise generation of the natural numbers could serve as a basis for mathematical

reasoning. That reasoning, moreover, could only be safe if based on finitistic notions and structures, the only ones perfectly handled by the human mind. In particular, given a finite structure and a property one could safely check the truth of the latter in the structure, by finite inspection; while this cannot be done over infinite structures, where the truth of an assertion would require an infinite verification. However, assertions on infinite structures can be made, provided that they are based on finite descriptions and arguments: finitary languages and finite **proofs** only make sense. In other words, an assertion is true when it comes together with a (finitistic) proof. As a consequence of constructivism, following Brouwer (and Heyting), the truth of $A \vee \sim A$ holds only if it is verified over a finite set; in general, $A \vee \sim A$ strictly requires a proof, namely a proof of A or a proof of $\sim A$. Similarly, $A \to B$ is true if finitistically provable, where a proof of $A \to B$ means an effective procedure that converts any proof of A into a proof of B.

In this section we formalize the previously mentioned concept: C holds iff there is a proof c of C. In this case we write $c:C$. In particular, the proof c of an implicative assertion $A \to B$ must be an effective procedure which takes any proof of A into a proof of B. This will be the key idea in the rewriting below of $(\to \mathrm{I})$ and $(\to \mathrm{E})$ in an explicitly constructive way, where proofs are handled as part of the language.

For this purpose, we construct a language for proofs. This language contains variables, as $x:A$ means that there is an 'hypothetical' proof of A and it is used in (possibly cancelled) assumptions. Suppose, for example, that, if an hypothetical proof x of A is assumed, $x:A$, then a proof b of B may be derived, $b:B$. By $(\to \mathrm{I})$, one has $A \to B$ and, now, also a proof of it; namely, a transformation $\lambda x:A.b$ which takes any proof of A into a proof of B. The independence of $A \to B$ from A, which is cancelled, corresponds to the fact that the function $\lambda x:A.b$ does not depend on the (name of the) variable x.

Indeed, the variable binder λ is an 'abstraction operator' such as $\{x \mid \ldots\}$ in $\{x \mid A(x)\}$, or as $\forall x$ in a quantified formula $\forall x.A(x)$: the set or the formula do not depend on the bound variable x. Thus, the variable binder λ is introduced in the presence of cancelled assumptions:

$$(\to \mathrm{I}) \quad \frac{\begin{array}{c}[x:A] \\ \vdots \\ b:B\end{array}}{\lambda x:A.b:A \to B}.$$

Consider now $(\to \mathrm{E})$, i.e. Modus Ponens. In our constructive approach it says that from a proof $a:A$ and a proof $c:A \to B$ we may obtain a proof of B; indeed, such a proof is constructed from c and a.

Let us write (ca) for it. That is:

$$(\to \mathrm{E}) \quad \frac{a:A \qquad c:A \to B}{ca:B}.$$

In conclusion the language of proofs of $\mathbf{PN}_0$ is based only on variables $x, y, z, \ldots$, a variable binder λ and on application. Indeed, a **typed** language, as $a:A$ also means that 'a has type A'. By this, $\mathbf{PN}_0$ is really the first step in the analogy between types and formulae: the intuitive

meaning of $\lambda x{:}A.b{:}A \to B$ is that $\lambda x{:}A.b$ is a function from (the meaning of) A to B; indeed, a computable function as it is explicitly defined by finitistic syntax. Then $\lambda x{:}A.b$ can be applied to a term a of type A and gives an output of type B. Note that λ is a variable binder and binds variables in cancelled assumptions; thus a term does not depend on (the name of) a bound variable similarly as an implication does not depend on the truth of the premise. Conversely, if a term or proof of a proposition contains free variables, then the proposition depends on uncalled hypotheses or, correspondingly, those free variables must have an (uncancelled or) explicit type declaration.

We note now that, at this stage, proofs and terms may be viewed in a slightly different way. Namely, one may simply write $\lambda x.b$ as a proof of $A \to B$ by erasing all type information (by induction, i.e. also in b) in $\lambda x{:}A.b$. This is possible, at this stage, by the strong decidability properties mentioned in the Remark below (on typability). The advantage is twofold. First, it sets the base for ML programming styles, where one may write type-free programs which are typed at compile time by the system. Second, a term 'proves' many propositions: e.g. $\lambda x.x$ proves $A \to A$, or has type $A \to A$, for *all* instances of A (whereas $\lambda x{:}A.x$ is a different term for each type A). By this, we have a notion of type-schema corresponding to the notion of (axiom) schema in logic. This is not possible in the higher order systems described in the sequel.

More formally, then, terms and types are defined as follows:

2.1. Definition. λ – **terms** are inductively defined as:

$x, y, z, \ldots$	are terms
$\lambda x.b$	is a term, when b is a term
(ca)	is a term, when c, a are terms.

Types are inductively defined as:

a given set AT of atomic letters	are types
$A \to B$	is a type, whenever A and B are types.

Conventions. For terms: $\lambda x_1 x_2 \ldots x_n.a \equiv \lambda x_1.\lambda x_2.\ldots\lambda x_n.a$

$a_1 a_2 \ldots a_n \equiv ((\ldots(a_1 a_2)\ldots a_n)$;

For types: $A \to B \to C \equiv (A \to B) \to C$.

2.2. Definition. A list $\Gamma = \{(x_1{:}A_2), \ldots, (x_n{:}A_n)\}$ of pairs (variable:Type) is a well-formed set of assumptions or of type assignments if $x_i \neq x_j$, for $1 \leqslant i < j \leqslant n$.

Provability in $\mathbf{PN}_0$, i.e. $\Gamma \vdash a{:}A$, is defined as for the classical case (see 1.2), w.r.t. ($\to$ I) and ($\to$ E) above. A useful exercise now is to give effective proofs of simple theorems of $\mathbf{PN}_0$.

2.3. Lemma.

$$\mathbf{PN}_0 \vdash \lambda xy.x{:}A \to (B \to A)$$
$$\mathbf{PN}_0 \vdash \lambda xyz.xz(yz){:}A \to (B \to C) \to ((A \to B) \to (A \to C)).$$

Proof.

$$\dfrac{\dfrac{\dfrac{[x:A]\quad[y:B]}{x:A}}{\lambda y.x:B\to A}\ (\to \mathrm{I})}{\lambda xy.x:A\to(B\to A)}\ (\to \mathrm{I})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[z:A]\quad[x:A\to(B\to C)]}{xz:B\to C}\ (\to \mathrm{E})\qquad\dfrac{[z:A]\quad[y:A\to B]}{yz:B}\ (\to \mathrm{E})}{xz(yz):C}\ (\to \mathrm{E})}{\lambda z.xz(yz):A\to C}\ (\to \mathrm{E})}{\lambda yz.xz(yz):(A\to B)\to(A\to C)}\ (\to \mathrm{I})}{\lambda xyz.xz(yz):A\to(B\to C)\to((A\to B)\to(A\to C))}\ (\to \mathrm{I})\qquad \Delta$$

Notation. $K_\lambda \equiv \lambda xy.x; S_\lambda \equiv \lambda xyz.xz(yz)$.

Note that λ-terms code proofs of $\mathbf{PN}_0$: each λ corresponds to the use of an $(\to \mathrm{I})$ rule, each application to $(\to \mathrm{E})$ rule. Thus by looking at a term, with the right parentheses, one may reconstruct the theorem it proves.

Remark (On typability). Not all terms, as defined, have a type (e.g. xx). However, it is decidable whether a type-free term has a type and one can uniformly effectively generate its 'most general' type (principal type schema; see Hindley, 1969; Milner, 1978). These facts are the core of type-inference and type-checking systems in functional programming. This is why it is worth looking both at the typed and the type free calculi, in the $\mathbf{PN}_0$ case (see §6, the paragraph on 'Semantics of type assignment and of typed terms').

The rest of the first part of this paper will be mostly concerned with the language of proofs in 2.1 and its extensions; when defined by a collection of axioms, which specify how terms reduce to each other, it is also known as the λ-calculus (see §3.3). However, it may be worth spending a few lines on a simple variant, which actually originated the topic by the work of Shönfinkel and Curry in the 20s and 30s. The correspondence between these two systems also clarifies the correspondence between the deductive approach of Natural Deduction and Hilbert's axiomatic systems.

We assume that the reader is familiar with the classical Hilbert-style presentation of propositional calculus; that is with the system of logic based on axioms and inference rules. Indeed, Hilbert's presentation of logic differs from Gentzen–Prawitz in that it assumes some generally valid truth (the axioms or **axiom schemata**) and deduces other truth from these by means of inference rules. As should be clear, no assumptions are made in the system $\mathbf{PN}_0$, which only relies on the concept of inference. The two different perspectives in the following will suggest two different calculi of proofs. We recall one of the many equivalent axiomatic systems for a fragment of Hilbert's Propositional Calculus.

2.4. Definition. The **Positive Propositional Calculus ($\mathbf{PP}_0$)** is given by the following axioms and rule:

$$Ax.1) \quad A \to (B \to A)$$
$$Ax.2) \quad A \to (B \to C) \to ((A \to B) \to (A \to C))$$
$$\text{(MP)} \quad \frac{A \quad A \to B}{B}.$$

In this case, of course, derivability is defined w.r.t. the given axioms.

Recall that $\mathbf{PN}_0$ makes no assumptions, except for the structure of deductions, while $\mathbf{PP}_0$ relies on two axioms, i.e. on two underived assumptions. This corresponds to having constants in the language of proofs. That is, in the constructive positive case, it is assumed once and for all that there are 'proofs' of the two propositions (or schemata) in $Ax.1$ and 2. Let us call K and S these (constant) proofs (cf, K_λ and S_λ in §2.3):

$$(K) \quad K\colon A \to (B \to A)$$
$$(S) \quad S\colon A \to (B \to C) \to ((A \to B) \to (A \to C)).$$

2.5. Definition. Combinators are inductively defined as:

$x, y, z, \ldots$	are combinators
K and S	are combinators
(ca)	is a combinator, when c, a are combinators.

Variables should be understood as proofs of hypothetical assumptions, if needed. K and S are constant as above and application (ca) is motivated by $(\to \text{E})$. The crucial point is that one can get rid of variables in any compound combinator and this will be exactly the following deduction theorem.

Recall now that by axiom we actually mean axiom schema. Thus, K and S are proofs of any formula obtained from $ax.1$ and $ax.2$ by consistent instantiation. Or, equivalently, they have the **type schema** described in (K) and (S).

2.6. Lemma. $\mathbf{PP}_0 \vdash SKK\colon A \to A$.

Proof. In the proof, the types of S and K are instantiated as to allow the application of (MP), i.e. of the rule $(\to \text{E})$.

$$\frac{\dfrac{S\colon (A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A)) \quad K\colon A \to ((A \to A) \to A)}{SK\colon (A \to (A \to A)) \to (A \to A) \qquad K\colon A \to (A \to A)}}{SKK\colon A \to A}.$$

Δ

Set $I \equiv SKK$, then I is a proof of $A \to A$. Its intended meaning is the identity function of type $A \to A$ (see later). Note that combinators code proofs in $\mathbf{PP}_0$. For example $A \to A$ is proved by first applying $(\to \text{E})$ to (an instance of) axiom S to axiom K: this gives (SK). Then apply $(\to \text{E})$ again to the theorem coded by (SK) and to axiom K: this gives $((SK)K)$, a proof of $A \to A$.

2.7. Lemma ('Constructive' Deduction Theorem for $\mathbf{PP}_0$). *Let* $\Gamma, x:A \vdash b:B$ *then there exists a term* $[x].b$, *such that*

$$x \notin FV([x].b) \quad \textit{and} \quad \Gamma \vdash [x].b : A \to B.$$

Proof. (By induction on the structure of b.)
(1) $b \equiv x$. Then $A \equiv B$. Recall that $I:A \to A$ and set $[x].b \equiv I$.
(2) $x \notin FV(b)$. Then the assumption $x:A$ is not used in the derivation, i.e. $\Gamma \vdash b:B$. Now $K:B \to (A \to B)$, thus $\Gamma \vdash Kb:A \to B$ and set $[x].b = Kb$. (Clearly, the case $b \equiv y \neq x$ could *suffice* for the inductive proof.)
(3) $b \equiv cd$. Then

$$\frac{c:D \to B \quad d:D}{cd:B}$$

has been applied.

Observe that, by induction, $[x].c:A \to (D \to B)$ and $[x].d:A \to D$. Thus

$$\frac{\dfrac{S:A \to (D \to B) \to ((A \to D) \to (A \to B)) \quad [x].c:A \to (D \to B)}{S([x].c):(A \to D) \to (A \to B)} \qquad [x].d:A \to D}{S([x].c)([x].d):A \to B}.$$

Set then $[x].b = S([x].c)([x].d)$. Δ

2.8. Theorem. *If* $\mathbf{PP}_0 \vdash a:A$, *then there exists a closed* λ*-term* a' *s.t.*

$$\mathbf{PN}_0 \vdash a':A.$$

Conversely, if $\mathbf{PN}_0 \vdash a:A$, *then there exists a closed SK-term* a° *s.t.*

$$\mathbf{PP}_0 \vdash a^\circ:A.$$

Proof. Use K_λ and S_λ to translate combinators into λ-terms. For the converse, translate $\lambda x \ldots$ by $[x] \ldots$ and use induction and the lemma. Δ

For the reader familiar with Shoenfinkel–Curry's abstraction algorithms, by which one may embed λ-terms into combinators, it should be clear that they are just a constructive investigation of the classical deduction theorem, as pointed out by the last two results (see also Huet, 1986).

3. Reductions of proofs and terms

In the previous section we were able to look at proofs as terms of a very simple language, indeed a functional language. By this, the inhabited types of that language turned out to be exactly the theorems of $\mathbf{PN}_0$. The first application of this analogy is now a connection between reducing proofs and terms.

Let us define formally the usual mathematical concept of instantiation:

3.1. Definition. Let b be a λ-term or an SK-term, then the term $[a/x]b$ is inductively defined with a free for x in b, in the following way:
$-\ b \equiv x$ then $[a/x]b \equiv a$
$-\ x \notin FV(b)$ then $[a/x]b \equiv b$
$-\ b \equiv \lambda y.c$ then $[a/x]b \equiv \lambda y.[a/x]c$
$-\ b \equiv cd$ then $[a/x]b \equiv ([a/x]c)([a/x]d)$.
(As usual, a free for x in b if there is no free occurrence of x in b, which is in the scope bound by a variable free in a).

3.2. Remark. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{[x:A]}{\vdots}}{\dfrac{b:B}{\lambda x.b:A\to B}(\to\mathrm{I})} \qquad a:A}{(\lambda x.b)\,a:B}(\to\mathrm{E}). \tag{P.1}$$

As $x:A$ is 'any hypothetical proof of A' and x may occur in b, this simplifies (**reduces**) to:

$$\begin{array}{c} a:A \\ \vdots \\ [a/x]b:B. \end{array} \tag{P.2}$$

This passage corresponds to the following practice in mathematics. In order to obtain a proof $[a/x]b$ of B, depending on a specific proof a of A, one may more easily prove in general that, 'for any $x:A$', $b:B$ can be deduced and, then, apply the general proof $\lambda x.b$ of $A\to B$ to the specific proof a of A.

However, this indirect argument, logically reduces to the derivation of $[a/x]b:B$ from $a:A$. In conclusion, the proof (P.1) reduces to (P.2) and the terms $(\lambda x.b)a$, $[a/x]b$ respectively, code these proofs; this suggests a calculus of proofs axiomatizing this concept.

3.3. Definition. **λ-calculus** ($\lambda\beta_>$ or λ_0) is the calculus of proofs defined as λ-terms and axiomatized by (α) and (β) plus the inference rules below which turn '$>$' into a *reflexive*, *transitive* and *substitutive* relation:

(α) $\lambda x.a > \lambda y.[y/x]a$

(β) $(\lambda x.b)a > [a/x]b$ provided that a is free for x in b.

Rules:

(ρ) $a > a$

(τ) $\dfrac{a > b \qquad b > c}{a > c}$

($\mu\nu$) $\dfrac{a > b \qquad a' > b'}{aa' > bb'}$ (ξ) $\dfrac{a > b}{\lambda x.a > \lambda x.b}$.

Conventions. Recall that λ is a variable binder and bound variables may be freely renamed, by (α). Therefore, given a, x and b, we may always consider b free for x in a, by renaming the bound variables in a.

Remark. Consider the following derivation:

$$\begin{array}{ll} (\to \mathrm{E}) & \dfrac{c:A\to B \qquad [x:A]}{cx:B} \\ (\to \mathrm{I}) & \dfrac{}{\lambda x.cx:A\to B} \end{array}. \tag{P.3}$$

Similarly as before, one shows that $\lambda x.cx$ is a proof of $A \to B$, under the uncancelled assumption that $c:A \to B$. Thus, one may simplify (P.3) to

$$c:A\to B. \tag{P.2}$$

This corresponds to the $\lambda\beta\eta_>$ calculus which has the extra axiom:

$$(\eta) \quad \lambda x.cx > c \text{ provided that } x \text{ is free in } c.$$

3.4. Definition. A λ-term a **is in normal form** iff it does not contain subterms of the form $(\lambda x.b)c$.

3.5. Definition. A λ-term a **has normal form** iff there exists a term a' in normal form such that $a > a'$.

In natural deduction we say that a proof is in normal form if it does not contain a ($\to$ I) rule followed by a ($\to$ E) rule. This corresponds exactly to say that a term, viewed as a code for a proof, cannot have subterms such as $(\lambda x.b)c$. Indeed, $(\lambda x.b)c$ is formed first by abstraction, i.e. ($\to$ I), then by application, i.e. ($\to$ E).

Notice now that, if a term a is in normal form, there are only two possibilities:

(i) $a \equiv xb_1 \ldots b_n$ with $b_1 \ldots b_n$ in normal form,

(ii) $a \equiv \lambda x.b$ with b in normal form.

3.6. Proposition.

(1) $K_\lambda ab > a$, $S_\lambda abc > ac(bc)$.

(2) *Let a and b be closed terms in normal form. Assume that $a:(A\to(B\to A))$ and $b:(A\to(B\to C))\to((A\to B)\to(A\to C))$. Then $a \equiv K_\lambda$ and $b \equiv S_\lambda$.*

Proof.

(1) By sequential applications of the (β) axiom:

$$K_\lambda ab \equiv (\lambda xy.x)ab > (\lambda y.a)b > a.$$

Analogously:

$$S_\lambda abc \equiv (\lambda xyz.xz(yz))abc > (\lambda yz.az(yz))bc > (\lambda z.az(bz))c > ac(bc).$$

(2) Let a be a closed term in normal form whose type-scheme is: $A\to(B\to A)$. Since a is in normal form, it has one of the forms:

$$xb_1 \ldots b_n \quad \text{or} \quad \lambda x.b \ (n \geqslant 0).$$

The first form is impossible, since a is closed. Hence, for some b, $a \equiv \lambda x.b$; that is, $(\rightarrow \mathrm{I})$ has been used in the deduction of type-scheme of a:

$$(\rightarrow \mathrm{I}) \quad \frac{\begin{array}{c}[x:A] \\ \vdots \\ b:(B \rightarrow A)\end{array}}{(\lambda x.b):A \rightarrow (B \rightarrow A)}.$$

Since x has a type A which may be atomic, x must occur in b in a non-function-position, i.e. b cannot be of the form $(xc_1 \ldots c_{m+1})$. On the other hand, b cannot be of the form $(\lambda y.c)$ $c_1 \ldots c_{m+1}$ as it would not be in normal form; nor of the form $zc_1 \ldots c_m$ because z would be free in b, while we assumed that b contains at most x free.

Hence b has one of the forms:

– $b \equiv x$, then $a \equiv \lambda x.x:A \rightarrow A$ which is impossible because this is a scheme different from $A \rightarrow (B \rightarrow A)$.

– $b \equiv \lambda y.c$, then the deduction coded by a had to be:

$$\begin{array}{ll} & \begin{array}{cc}[x:A] & [y:B] \\ \ddots & \iddots \end{array} \\ (\rightarrow \mathrm{I}) & \dfrac{c:A}{b \equiv \lambda y.c:B \rightarrow A} \\ (\rightarrow \mathrm{I}) & \dfrac{}{a \equiv \lambda xy.c:A \rightarrow (B \rightarrow A)}. \end{array}$$

Consider now the structure of c. It cannot be of the form $(\lambda w.d)$ because it would give a composite type-scheme. In addition c must be in normal form, because a is in normal form; it must not contain free variables different from x and y because it has to be closed. Thus $c \equiv x$ or $c \equiv y$. Let us suppose that $c \equiv y$, then $a \equiv \lambda xy.y:A \rightarrow (B \rightarrow B)$ since:

$$\dfrac{\dfrac{\begin{array}{cc}[x:A] & [y:B] \\ \ddots & \iddots \end{array}}{y:B}}{\lambda x.y:B \rightarrow B}$$
$$a \equiv \lambda xy.y:(A \rightarrow (B \rightarrow B))$$

which is impossible. Thus $c \equiv x$ and then $a > \lambda xy.x \equiv K_\lambda$. (Analogously for S_λ.) Δ

Thus K_λ and S_λ are the only closed normal forms of type schemas corresponding to the axioms for $\mathbf{PP_0}$. 3.6.1 gives their operational behaviour. In accordance to these facts, define the theory of proofs of $\mathbf{PP_0}$ as follows.

3.7. Definition. Combinatory Logic ($\mathbf{CL_>}$) is the calculus of proofs of $\mathbf{PP_0}$ axiomatized by:

$$Kab > a$$
$$Sabc > ac(bc)$$

plus inference rules (ρ), $(\mu\nu)$ and (τ) in 3.3.

3.8. Corollary. *Let* $[x].b$ *be defined as in* 2.7. *Then* $([x].b)\,a > [a/x]\,b$.

Proof. (By induction on the structure of b.) Consider the possible cases for b and use the constructive version of the deduction theorem in $\mathbf{PP}_0$.
– If $b = x$ then $[x].b \equiv SKK \equiv I$, thus $(SKK)\,a > Ka(Ka) > a = [a/x]\,b$.
– *If* $x \notin FV(b)$ then $[x].b \equiv Kb$ and $KBa > b \equiv [a/x]\,b$.
– $b = cd$; then $[x].b \equiv S([x].c)([x].d)$ and $S([x].c)([x].d)\,a > ([x].c)\,a(([x].d)\,a) > [a/x]\,(cd)$. Δ

The ' = ' relation is the symmetric and transitive closure of ' > '. In the calculi below we will directly define equality between terms.

4. First order logic or types depending on terms

As usual in logic and mathematics, most relevant facts are expressed by an essential use of assertions on arbitrary elements (of a given set or type). That is, mathematical practice relies on the use of predicates which include variables and on quantification with respect to variables. A type-theoretic foundation of this common practice has been given by the work of Martin-Löf. This method, of course, is widely used also in computer science (see Bruce and Longo, 1988; for a recent application to the description and interpretation of 'record types'). As before, we only describe the logic 'core' of first order natural deduction.

The **expressions** of the language we are going to define are terms and types (or formulae). They will be defined by combined induction as terms may contain types and types may contain terms. We write capital letters for expressions which are types. **A:Tp** is short for 'A is a type'. In contrast to the propositional case, we only consider typed terms (see the remark on typability, in §2). The base is given by a collection of variables $x, y, z, \ldots$ and of atomic types or predicates.

Conventions. λ, $\forall$, $\exists$ are variables binders. An unbound variable x in exp is **free** in exp (*notation*: $x \in FV(\text{exp})$). The **substitution** of a for x in exp is defined by induction, provided that a is free for x in exp, as usual.

Here is a (combined) inductive definition of term and type expressions. The formation and deduction rules below will select the (well) typed and well-formed terms and types.

Term expressions: $a := \mathsf{var} \mid \mathsf{fst}\,a \mid \mathsf{snd}\,a \mid (aa) \mid \langle a, a\rangle \mid (\lambda\,\mathsf{var}{:}A\,.\,a)$
Type expressions: $A := \mathsf{Atomic} \mid (\forall\,\mathsf{var}{:}A\,.\,A) \mid (\exists\,\mathsf{var}{:}A\,.\,A)$.

The derivations from lists of assumptions (or assignment of types to variables, see below) are summarized as **judgements**, following Martin-Löf terminology. $\Gamma(x{:}A)$ stands for the ordered list Γ extended with $(x{:}A)$.

Judgements: $\Gamma\ \mathsf{ok}$; $\Gamma \vdash A{:}Tp$; $\Gamma \vdash A = B{:}Tp$; $\Gamma \vdash a{:}A$; $\Gamma \vdash a = b{:}A$.

Well-formed assumptions:

(Empty) $\varnothing$ ok (the empty assignment is well formed)

$$\text{(Context)}\quad \frac{\Gamma \text{ ok} \qquad \Gamma \vdash A:Tp}{\Gamma(x:A) \text{ ok}} \quad \text{for } x \notin FV(\Gamma)$$

$$\text{(Assump)}\quad \frac{\Gamma \text{ ok}}{\Gamma \vdash (x:A)} \quad \text{for } (x:A) \in \Gamma.$$

Circular clashes are avoided by the assignment rules, e.g. $\{(x:A(y)), y:B)\}$ is not an ok assignment.

Equality ' $=$ ' is a reflexive, symmetric and transitive relation, which is congruent w.r.t. types according to the following rules:

$$\text{(Congr.1)}\quad \frac{a:A \qquad A = B:Tp}{a:B}$$

$$\text{(Congr.2)}\quad \frac{a = b:A \qquad A = B:Tp}{a = b:B}.$$

Remark. The following rules are admissible. For any judgement J,

$$\text{(Sub)}\quad \frac{\Gamma \vdash a:A \qquad \Gamma, (x:A) \vdash J}{\Gamma \vdash [a/x]J}$$

$$\text{(Congr)}\quad \frac{\Gamma \vdash a = b:A \qquad \Gamma \vdash [a/x]J}{\Gamma \vdash [b/x]J}.$$

The last rule is needed, as may others below, since atomic types may contain term variables, and, hence, type expressions may do so. Substitution of variables by terms (or terms by equal terms) for atomic types (i.e. predicates) will be discussed in the general setting of mathematical extensions of the system below (see the **Intermezzo**).

Type Formation and Equality:

$$\frac{A:Tp \qquad [x:A] \vdash B:Tp}{(\forall x:A.B):Tp^{(*)}} \qquad \frac{A:Tp \qquad [x:A] \vdash B:Tp}{(\exists x:A.B):Tp^{(*)}}$$

$$\frac{A = C:Tp \qquad [x:A] \vdash B = D:Tp}{(\forall x:A.B) = (\forall x:C.D):Tp} \qquad \frac{A = C:Tp \qquad [x:A] \vdash B = D:Tp}{(\exists x:A.B) = (\exists x:C.D):Tp}.$$

Remark. As for $(*)$, observe that in B there is no free variable whose type depends on x.

Conventions.

1 – If $x \notin FV(B)$, set $A \to B$ for $\forall x:A.B$ and $A \times B$ for $\exists x:A.B$.

2 – In the rules we are only writing the part of the assumptions which is discharged in the conclusion. Thus a judgement J in a rule stands for $\Gamma \vdash J$ (see §4.1(ii)).

Dependent products (*introduction, elimination, equality*):

$$(\forall \mathrm{I}) \quad \frac{[x{:}A] \vdash b{:}B}{(\lambda x{:}A.b){:}(\forall x{:}A.B)}$$

$$(\forall \mathrm{I}=) \quad \frac{A = C \qquad [x{:}A] \vdash b = d{:}B}{(\lambda x{:}A.b) = (\lambda x{:}C.d){:}(\forall x{:}A.B)}$$

$$(\forall \mathrm{E}) \quad \frac{f{:}(\forall x{:}A.B) \qquad a{:}A}{(fa){:}[a/x]B}$$

$$(\forall \mathrm{E}=) \quad \frac{f = g{:}(\forall x{:}A.B) \qquad a = b{:}A}{(fa) = (gb){:}[a/x]B}$$

$$(\forall \beta) \quad \frac{[x{:}A] \vdash b{:}B \qquad a{:}A}{(\lambda x{:}A.b)a = [a/x]b{:}[a/x]B}$$

$$(\forall \eta) \quad \frac{f{:}(\forall x{:}A.B)}{(\lambda x{:}A.fx) = f{:}(\forall x{:}A.B)} \quad x \text{ not free in } f.$$

Dependent sums (*introduction, elimination, equality*):

$$(\exists \mathrm{I}) \quad \frac{a{:}A \qquad b{:}[a/x]B}{\langle a, b\rangle{:}\exists x{:}A.B}$$

$$(\exists \mathrm{I}=) \quad \frac{a = c{:}A \qquad b = d{:}[a/x]B}{\langle a, b\rangle = \langle c, d\rangle{:}\exists x{:}A.B}$$

$$(\exists \mathrm{E}) \quad \frac{b{:}\exists x{:}A.B}{\mathrm{fst}\, b{:}A \quad \mathrm{snd}\, b{:}[\mathrm{fst}\, b/x]B}$$

$$(\exists \mathrm{E}=) \quad \frac{b = c{:}\exists x{:}A.B}{\mathrm{fst}\, b = \mathrm{fst}\, c{:}A \quad \mathrm{snd}\, b = \mathrm{snd}\, c{:}[\mathrm{fst}\, b/x]B}$$

$$(\exists \beta) \quad \frac{a{:}A \qquad b{:}[a/x]B}{\mathrm{fst}\langle a, b\rangle = a{:}A \quad \mathrm{snd}\langle a, b\rangle = b{:}[a/x]B}$$

$$(\exists \eta) \quad \frac{c{:}\exists x{:}A.B}{\langle \mathrm{fst}\, c, \mathrm{snd}\, c\rangle = c{:}\exists x{:}A.B}.$$

4.1. Remarks.

(i) By the rules, $x{:}A$ is allowed in judgements only if $x \notin FV(A)$.

(ii) Note that ($\forall$I) could be equivalently written as

$$(\forall \mathrm{I}) \quad \frac{\Gamma \vdash A{:}Tp \qquad \Gamma(x{:}A) \vdash a{:}B}{\Gamma \vdash (\lambda x{:}A.a){:}(\forall x{:}A.B)}$$

thus, by the rules, when ($\forall$I) is applied, x is not free in uncancelled hypothesis. Similarly in the related cases.

(iii) One may equivalently define $\mathbf{PN}_1$ as an extension of $\mathbf{PN}_0$, but the expressions and rules of $\mathbf{PN}_0$ may be derived from the presentation just given. Since, for $x \notin FV(B)$, $A \to B \equiv \forall x{:}A\,.\,B$, one has that $(\to \mathrm{I})$ and $(\to \mathrm{E})$ in $\mathbf{PN}_0$ are special cases of $(\forall \mathrm{I})$ and $(\forall \mathrm{E})$. Observe that, if there are no atomic types with free variables, $\mathbf{PN}_1$ would coincide with $\mathbf{PN}_0$.

4.2. Definition. $\mathbf{PN}_1$ is the deductive system defined above and $\lambda\mathbf{P}$ is the calculus of its proofs (as terms).

The reader may recognize two familiar properties of first-order logic in the theorems in proposition 4.3: proposition 4.3 explicitly gives their constructive proofs.

4.3. Proposition. (1) *If* $x \notin FV(B)$, *then exists* c *such that*:

$$\mathbf{PN}_1 \vdash c{:}(\forall x{:}A\,.\,(B \to C)) \to (B \to \forall x{:}A\,.\,C).$$

(2) *If* $x \notin FV(C)$, *there exists* c *such that*:

$$\mathbf{PN}_1 \vdash c{:}(\forall x{:}A\,.\,(B \to C)) \to ((\exists x{:}A\,.\,B) \to C).$$

Proof.

(1)
$$\dfrac{\dfrac{\dfrac{\dfrac{[z{:}(\forall x{:}A\,.\,(B \to C))] \qquad [x{:}A]}{[y{:}B]^{(*)} \qquad zx{:}B \to C}\ (\to \mathrm{E})}{zxy{:}C}}{\lambda x{:}A\,.\,zxy{:}(\forall x{:}A\,.\,C)}\ (\forall \mathrm{I})}{(\lambda y{:}B\,.\,\lambda x{:}A\,.\,zxy){:}(B \to (\forall x{:}A\,.\,C))}\ (\to \mathrm{I})}{(\lambda z{:}(\forall x{:}A\,.\,(B \to C))\,.\,\lambda y{:}B\,.\,\lambda x{:}A\,.\,zxy){:}(\forall x){:}A\,.\,(B \to C)) \to (B \to (\forall x{:}A\,.\,C))}\ (\to \mathrm{I})$$

where y is a fresh variable s.t. $y \notin FV(BCA)$; $^{(*)}x \notin FV(B)$ as $(y{:}B)$ follows, in the ordered set of assumptions, $(x{:}A)$.

(2)
$$\dfrac{\dfrac{\dfrac{\dfrac{[z{:}(\forall x{:}A\,.\,(B \to C))] \qquad \dfrac{[y{:}(\exists x{:}A\,.\,B)]}{\mathrm{fst}\,y{:}A \quad \mathrm{snd}\,y{:}[\mathrm{fst}\,y/x]B}\ (\exists \mathrm{E})}{z(\mathrm{fst}\,y){:}([\mathrm{fst}\,y/x]B) \to C^{(**)} \quad \downarrow}\ (\forall \mathrm{E})}{z(\mathrm{fst}\,y)(\mathrm{snd}\,y){:}C}\ (\to \mathrm{E})}{\lambda y{:}(\exists x{:}A\,.\,B)\,.\,z(\mathrm{fst}\,y)(\mathrm{snd}\,y){:}((\exists x{:}A\,.\,B) \to C)}\ (\to \mathrm{I})}{\lambda z{:}(\forall x{:}A\,.\,(B \to C))\,.\,\lambda y{:}(\exists x{:}A\,.\,B)\,.\,z(\mathrm{fst}\,y)(\mathrm{snd}\,y){:}(\forall x{:}A\,.\,(B \to C)) \to ((\exists x{:}A\,.\,B) \to C)}\ (\to \mathrm{I})$$

$^{(**)}$ by assumption $x \notin FV(C)$ Δ.

Intermezzo (on mathematical theories)

We have presented so far only the 'logical rules' for propositional and first-order systems, within the constructive perspective of type theory. Mathematical practice, though, and the

needs of computer science require the investigation and use of specific extensions of these core theories. In this regard, in constructive natural deduction, i.e. in this type-theoretic formalization of first-order deduction, one has more expressive power than in the usual approaches to natural deduction. Besides 'proof terms', a crucial difference is in the representation of sorts and propositions both as types, while sorts and formulae are different entities in logic.

Observe that the extensions are usually given by adding *constant symbols*, i.e. *predicates* and *functions*, of various *sorts*, whose behaviour is given by telling their '*arity*' and by a bunch of *axioms*. That is, sorts, predicates, functions and axioms of a first-order theory can be represented in a *signature* (Σ, say) in the following way (see Harper *et al.*, 1987). Judgements, then, also depend on signatures.

Judgements.

Σ ok; $\Sigma \vdash \Gamma$ ok; $\Sigma, \Gamma \vdash A : Tp$; $\Sigma, \Gamma \vdash A = B : Tp$; $\Sigma, \Gamma \vdash a : A$; $\Sigma, \Gamma \vdash a = b : A$; $X : [\ldots]A$ gives the 'arity' of X.

Well-formed signatures.

(Axiom) $\varnothing$ ok (the empty signature is well formed)

(Sign.1) $$\frac{\Sigma \vdash \Gamma \text{ ok}}{\Sigma(X : [\Gamma]\, Tp) \text{ ok}}$$ for X not in Σ

(Sign.2) $$\frac{\Sigma, \Gamma \vdash A : Tp}{\Sigma(X : [\Gamma] A) \text{ ok}}$$ for X not in Σ.

Constants.

Sorts: $X : [\,]\, Tp$

Predicates: $X : [x_1 : A_1, \ldots, x_n : A_n]\, Tp$ where $A_i : Tp$, for all $i \leqslant n$;

Functions: $X : [x_1 : A_1, \ldots, x_n : A_n] A_0$ where $A_i : Tp$, for all $i \leqslant n$;

Axioms: $X : [\,] A$ where $A : Tp$ with no free variables.

The various formation, elimination and introduction rules follow the pattern below, where $K : Tp$ or $K = Tp$ and the constant X appears as $X : [x_1 : A_1, \ldots, x_n : A_n] K$ in Σ:

(X) $$\frac{\Sigma \text{ ok} \quad \Sigma, \Gamma \vdash a_i : [a_1/x_1, \ldots, a_{i-1}/x_{i-1}] A_i \quad 1 \leqslant i \leqslant n}{\Sigma, \Gamma \vdash X(a_1, \ldots, a_n) : [a_1/x_1, \ldots, a_n/x_n] K}$$

(X =) $$\frac{\Sigma \text{ ok} \quad \Sigma, \Gamma \vdash a_i = b_i : [a_1/x_1, \ldots, a_{i-1}/x_{i-1}] A_i \quad 1 \leqslant i \leqslant n}{\Sigma, \Gamma \vdash X(a_1, \ldots, a_n) = X(b_1, \ldots, b_n) : [a_1/x_1, \ldots, a_n/x_n] K}.$$

4.4. Remark. Another aspect of the greater expressive power of type theory, is given by the ($\exists$E) rule. The 'there exists elimination' rule is usually formalized in natural deduction as

$$(\exists E)_1 \quad \frac{\exists x : A \,.\, B \qquad B \vdash C}{C} \quad \text{with } x \notin FV(C)$$

while the (∃E) in **dependent sums** above is equivalent to

$$(\exists E)_2 \quad \frac{a:\exists x:A.B \qquad x:A, y:B \vdash c:[\langle x,y\rangle/z]C}{(Let\ \langle x,y\rangle = a\ in\ c):[a/z]C}$$

where z may occur free in C. Thus $(\exists E)_2$ is much stronger than $(\exists E)_1$, since, by the proof terms, one has both first and second projections, as in (∃E). This requires a little proof. First observe that rules corresponding to (∃E =), ($\exists\beta$) and ($\exists\eta$) for the (*Let* ... *in* ...) may be easily given, e.g.

$$(\exists\beta)_2 \quad \frac{\langle a,b\rangle:\exists x:A.B \quad x:A, y:B \vdash c:[\langle x,y\rangle/z]C}{(Let\ \langle x,y\rangle = \langle a,b\rangle\ in\ c) = [a/x, b/y]c:[\langle a,b\rangle/z]C}$$

$$(\exists\eta)_2 \quad \frac{c:\exists x:A.B}{(Let\ \langle x,y\rangle = c\ in\ \langle x,y\rangle) = c:\exists x:A.B}.$$

(∃E =) is obvious. Then define, for $a:\exists x:A.B$,

$$\begin{aligned} \mathrm{fst}\, a &= Let\ \langle x,y\rangle = a\ in\ x:A \\ \mathrm{snd}\, a &= Let\ \langle x,y\rangle = a\ in\ y:[\mathrm{fst}\, a/x]B. \end{aligned}$$

What needs to be proved is that (snd a) is well typed. Indeed,

$$\begin{aligned} &x:A, y:B \vdash x = \mathrm{fst}\langle x,y\rangle:A \quad \text{by } (\exists\beta)_2 \\ &x:A, y:B \vdash B = [\mathrm{fst}\langle x,y\rangle/x]B:Tp \quad \text{by } B = [x/x]B \\ &x:A, y:B \vdash y:B \\ &x:A, y:B \vdash y:[\mathrm{fst}\langle x,y\rangle/x]B, \end{aligned}$$

which is identical to $[\langle x,y\rangle/z][\mathrm{fst}\, z/x]B$. In conclusion, the rules $(\exists E)_2$, $(\exists E =)_2$, $(\exists\beta)_2$ and $(\exists\eta)_2$ instead of the corresponding rules for $\mathbf{PN}_1$ above, give an equivalent system, which is strictly stronger than the usual natural deduction approach to existential quantification.

5. Second-order logic and explicit polymorphism

The working mathematician often makes assertions concerning arbitrary functions in a given collection (when describing integration, say) or even about arbitrary sets within a given category or class of sets (*all* c.p.o.'s have a least element ...).

In the previous section we have been dealing with a language for higher-type functions. Functional abstraction (i.e. λx ...) was defined w.r.t. to variables ranging over ground elements, functions, functionals and so on, in any finite higher type. Note that functional abstraction may be understood as a form of quantification; thus, as each boolean-valued function determines a set, abstracting w.r.t. a variable which ranges over boolean-valued functions is like quantifying over sets of a *given* type. However, we were not allowed to quantify explicitly over *types*. Indeed, there is some implicit quantification over types in the systems mentioned so far. Church–Curry types are defined as type schemata: e.g. the identity $\lambda x.x$ has type schema $\alpha\to\alpha$, i.e. $\lambda x.x$ has type $\sigma\to\sigma$ for *any* type σ, or the

collection of its possible types is obtained by consistently instantiating α in $\alpha \to \alpha$ by every type.

Mathematical practice and this implicit use of quantification suggest a language where one could explicitly consider all types: thus, a higher-order language, with quantification over types. This language is the core of the nowadays widely known approach to 'modularity' in functional programming via second-order or 'explicit' polymorphism.

The expressions of the language, i.e. terms and types (or formulae), will be defined by simultaneous induction as before, with the further possibility of including the expression Tp in their formation. We write capital letters to stress that an expression is a type; lower-case letters, though, may also be types. $A:Tp$ is a short for 'A is a type'.

The base is given by a collection of variables $x, y, z \ldots X, Y, Z \ldots$ and of atomic types or predicates. We maintain the usual conventions on variable binders (λ and $\forall$), free variables and substitution. A definition, by combined induction, of term and type expressions is first given, as in the first-order case. (We only give the 'proper' second order terms.) The rules pick up the legal ones.

Term expressions.

$$a := \mathsf{var} \mid (aa) \mid (aA) \mid (\lambda\, var{:}A\,.\,a) \mid (\lambda\, \mathsf{Var}{:}Tp\,.\,a).$$

Type expressions.

$$A := \mathsf{Var} \mid \mathsf{Atomic} \mid (A \to A) \mid (\forall\, \mathsf{Var}{:}Tp\,.\,A).$$

The judgements and related conventions are the same as for $\mathbf{PN}_1$. As $\mathbf{PN}_2$ below is a proper extension of $\mathbf{PN}_1$, we only summarize the new rules. (Recall that $A \to A$ is a special case of $(\forall\, \mathsf{Var}{:}A\,.\,A)$.)

Well-formed assumptions.

$$(\text{Context}^2) \quad \frac{\Gamma\ \mathsf{ok}}{\Gamma(X{:}Tp)\ \mathsf{ok}} \quad \text{for } X \notin FV(\Gamma)$$

$$(\text{Assump}^2) \quad \frac{\Gamma\ \mathsf{ok}}{\Gamma \vdash (X{:}Tp)} \quad \text{for } (X{:}Tp) \in \Gamma.$$

Equality '$=$' is a congruence as in the I order case. The II order type formation rules include the following crucial impredicative cases:

Type formation and equality.

$$\frac{[X{:}Tp] \vdash B{:}Tp}{(\forall X{:}Tp\,.\,B){:}Tp^{(*)}}$$

$$\frac{[X{:}Tp] \vdash B = D{:}Tp}{(\forall X{:}Tp\,.\,B) = (\forall X{:}Tp\,.\,D){:}Tp}.$$

(*) In B there is no free variable whose type depends on X.

2nd-order dependent products (*introduction, elimination, equality*):

$$(\forall^2\mathrm{I}) \quad \frac{[X:Tp] \vdash b:B}{(\lambda X:Tp.b):(\forall X:Tp.B)}$$

$$(\forall^2\mathrm{I}=) \quad \frac{[X:Tp] \vdash b = d:B}{(\lambda X:Tp.b) = (\lambda X:Tp.d):(\forall X:Tp.B)}$$

$$(\forall^2\mathrm{E}) \quad \frac{f:(\forall X:Tp.B) \qquad A:Tp}{(fA):[A/X]B}$$

$$(\forall^2\mathrm{E}=) \quad \frac{f = g:(\forall X:Tp.B) \qquad A = B:Tp}{(fA) = (gB):[A/X]B}$$

$$(\forall^2\beta) \quad \frac{[X:Tp] \vdash b:B \qquad A:Tp}{(\lambda X:Tp.b)A = [A/x]b:[A/X]B}$$

$$(\forall^2\eta) \quad \frac{f:(\forall X:Tp.B)}{(\lambda X:Tp.fx) = f:(\forall X:Tp.B)} \quad X \text{ not free in } f.$$

5.1. Definition. $\mathbf{PN}_2$ is the deductive system defined above and λP_2 is the calculus of its proofs (as terms).

The system $\mathbf{PN}_2$ is a (revisited) fragment of Coquand–Huet's 'Calculus of Constructions' (see Hyland and Pitts, 1987).

5.2. Remark.

(i) When ($\forall$I) is used w.r.t. to a non-empty list Γ of assumptions, X is not free in uncancelled assumptions, by the same argument as in (4.1).

(ii) If there are no atomic types with free variables, λP_2 coincides with Girard–Reynolds' second-order λ-calculus, since $A \to B \equiv \forall x:A.B$.

The very elegant and relevant fact, in second-order logic, is that also the connectives $\{\wedge, \vee, \exists\}$ and the absurdum $\bot$, are derivable in this purely implicative fragment, $\mathbf{PN}_2$, based only on $\forall$ and $\to$. This was discovered, in logic, long ago. Observe that second-order quantification is essential for this.

5.3. Definition. $\bot \equiv \forall X:Tp.X$.

It is sound to consider $\bot$ the 'absurdum' or constantly false predicate, in view of the following theorem. It proves that '*ex falso quodlibet*' (1) and that 'if one can prove any proposition (type) then one can prove falsum' (2).

5.4. Theorem.

$$(1) \quad \frac{\bot \quad A:Tp}{A}; \qquad (2) \quad \frac{X:Tp \vdash b:X}{(\lambda X:Tp.b):\bot}.$$

Proof. Indeed, we can give a constructive version of (1), the absurdum rule of intuitionistic logic, as a derived rule.

(1) Let $a:(\forall X:Tp.X) \equiv \bot$, fixed a generic $A:Tp$, we have:

$$\frac{a:(\forall X:Tp.X) \qquad A:Tp}{(aA):A}\ (\forall^2\mathrm{E}).$$

(2) This second inference is just a trivial application of $(\forall^2\mathrm{I})$. Δ

5.5. Definition.

$$A \wedge B \equiv \forall Y:Tp.((A \to (B \to Y)) \to Y)$$
$$A \vee B \equiv \forall Y:Tp.((A \to Y) \wedge (B \to Y) \to Y).$$

5.6. Theorem.

$$(1)\ \frac{a:A \qquad b:B}{\lambda X:Tp.\lambda p:(A \to (B \to X)).pab:(A \wedge B)}; \qquad (2)\ \frac{a:A \wedge B}{aAK_\lambda:A}.$$

Proof.

(1)
$$\frac{a:A \qquad [p:(A \to (B \to X))]}{\dfrac{\dfrac{b:B \qquad pa:(B \to X)}{pab:X}\ [X:Tp]}{\dfrac{\lambda p:(A \to (B \to X)).pab:(A \to (B \to X)) \to X}{\lambda X:Tp.\lambda p:(A \to (B \to X)).pab:(\forall X:Tp.((A \to (B \to X)) \to X))}}}$$

$(\to \mathrm{E})$, $(\to \mathrm{E})$, $(\to \mathrm{I})$, $(\forall^2\mathrm{I})$.

(2) Analogously:

$$\frac{\dfrac{a:(\forall X:Tp.((A \to (B \to X)) \to X)) \qquad A:Tp}{aA:(A \to (B \to A)) \to A}}{aAK_\lambda:A}$$

$(\forall^2\mathrm{E})$, $(\to \mathrm{E})$.

Note that A is a generic type and K_λ is uncancelled because its type-scheme is a theorem. Use $K_\lambda I$ to obtain the second projection. Δ

5.7. Definition.

$$\exists X:Tp.A \equiv \forall Y:Tp.((\forall X:Tp.(A \to Y)) \to Y).$$

More generally, one may also derive a first-order existential quantifier, by setting, for $B:Tp$,

$$\exists x:B.A \equiv \forall Y:Tp.((\forall x:B.(A \to Y)) \to Y).$$

In either case, the intuitive meaning of '$\exists$' may be understood by instantiating Y by $\bot$. Informally, if $\sim A\,(\equiv A \to \bot)$ is the negation of A:

$$\exists X:K.A \cong \sim \forall X:K.\sim A$$

for $K:Tp$ or $K = Tp$. However, if defined in this way, i.e. by using '$\sim$', '$\exists$' would not give the results below!

5.8. Definition.

$$\langle a, b\rangle_A \equiv \lambda Y{:}Tp.\lambda x{:}(\forall X{:}Tp.A \to Y).xab.$$

5.9. Remarks.

(1) Note that $[a, b] \equiv \lambda x.xab$ is the classical type-free pairing and may be obtained from $\langle a, b\rangle_A$ by erasing all type information. As in the type-free case, the second-order typed pairing is not surjective, i.e. for no terms as projections fst$([a, b])$ and snd$([a, b])$ one has that [fst$([a, b])$, snd$([a, b])$] $= [a, b]$.

(2) Definition 5.8 and the theorems below are only stated for $K = Tp$. They also hold for $K{:}Tp$. In particular, $\langle a, b\rangle_A \equiv \lambda Y{:}Tp.\lambda x{:}(\forall x{:}C.A \to Y).xab$, for $C{:}Tp$, is a first-order pairing.

5.10. Theorem.

$$(\exists^2\mathrm{I})\quad \frac{B{:}Tp \qquad b{:}[B/X]A}{\langle B, b\rangle_A{:}(\exists X{:}Tp.A)}.$$

Proof.

$$\dfrac{\dfrac{\dfrac{[Y{:}Tp] \qquad [x{:}(\forall X{:}Tp.A \to Y)] \qquad B{:}Tp}{xB{:}[B/X]A \to Y \qquad b{:}[B/X]A}\,(\forall^2\mathrm{E})}{xBb{:}Y}\,(\to\mathrm{E})}{(\lambda x{:}(\forall X{:}Tp.A \to Y).xBb){:}((\forall X{:}Tp.A \to Y) \to Y)}\,(\to\mathrm{I})$$

$$\frac{}{(\lambda Y{:}Tp.\lambda x{:}(\forall X{:}Tp.A \to Y).xBb){:}(\forall Y{:}Tp.(\forall X{:}Tp.A \to Y) \to Y)}\,(\forall^2\mathrm{I}).$$

Also $(\exists^2\mathrm{E})$ is a derived rule: Δ

5.11. Theorem.

$$(\exists^2\mathrm{E})\quad \frac{a{:}\exists X{:}Tp.A \qquad [X{:}Tp, x{:}A] \vdash c{:}C}{aC(\lambda X{:}Tp.(\lambda x{:}A.c)){:}C} \quad \text{if } x, X \notin FV(C).$$

Proof. Let $a \equiv \langle B, b\rangle_A \equiv \lambda Y{:}Tp.\lambda x{:}(\forall X{:}Tp.A \to Y).xBb$, i.e.

$$a{:}(\exists X{:}Tp.A) \equiv \forall Y{:}Tp.(\forall X{:}Tp.A \to Y) \to Y), \quad \text{where } B{:}Tp \text{ and } b \text{ is a term.}$$

Compute now:

$$\dfrac{\dfrac{a{:}(\forall Y{:}Tp.(\forall X{:}Tp.(A \to Y) \to Y)) \quad C{:}Tp}{aC{:}(\forall X{:}Tp.(A \to C) \to C)}\,(\forall^2\mathrm{E}) \qquad \dfrac{\dfrac{[X{:}Tp, x{:}A]}{\vdots}}{\dfrac{\lambda x{:}A.c{:}A \to C}{\lambda X{:}Tp.(\lambda x{:}A.c){:}(\forall X{:}Tp.(A \to C))}\,(\forall^2\mathrm{I})}\,(\forall\mathrm{I})^*}{aC(\lambda X{:}Tp.(\lambda x{:}A.c)){:}C}\,(\forall^2\mathrm{E})^*.$$

* Note that in these derivations we used the hypothesis $X, x \notin FV(C)$, as we applied $(\forall I) \equiv (\to I)$ and $(\forall E) \equiv (\to E)$. Δ

5.12. Corollary.

$$\langle B, b\rangle_A\, C(\lambda X{:}Tp.\lambda x{:}A.c) = [B/X, b/x]\, c{:}C$$

Proof.

$$\begin{aligned}
&(\lambda Y{:}Tp.\lambda x{:}(\forall X{:}Tp.A \to Y).xBb)\, C(\lambda X{:}Tp.\lambda x{:}A.c) \quad \text{for } Y \notin FV(A)\\
&\quad = (\lambda x{:}(\forall X{:}Tp.A \to C).xBb)\,(\lambda X{:}Tp.\lambda x{:}A.c)\\
&\quad = (\lambda X{:}Tp.\lambda x{:}A.c)\, Bb\\
&\quad = [B/X, b/x]\, c.
\end{aligned}$$

Δ

5.13. Remark. (On second order 'there exists elimination' rules.)

(i) For a derived second-order existential quantifier it is not possible to obtain projections at all, in the sense, say, of the first-order (∃E) in §4. As for the first projection, one should take $C \equiv Tp$ and $c \equiv X$, but the result would not be in the language. The second projection is impossible since one should take $C \equiv A$, which may depend on X, (cf, 4.4).

Note that, if one does not have predicates with free (term) variables, as in Girard's system ***F***, then, after erasing proof terms, $(\exists^2 E)$ becomes exactly the second-order rule for existential elimination in natural deduction.

(ii) However, one may consistently extend $\mathbf{PN}_2$ by a stronger rule. Namely, define pairing $\langle , \rangle$, as an independent function symbol, by rules such as (∃I) and (∃I =) in §4 (where A becomes Tp). Then set

$$(\exists^2 E)_2 \quad \frac{a{:}(\exists X{:}Tp.B) \qquad [X{:}Tp, x{:}B] \vdash c{:}[\langle X, x\rangle/z]\, C}{(Let\ \langle X, x\rangle = a\ in\ c){:}[a/z]\, C} \quad X, x \notin FV(C)$$

where, now, z may occur in C and, hence, C depends *indirectly* on X, x. By this, $(\exists^2 E)_2$ is not derivable. Clearly, then, the behaviour of the (*Let ... in ...*) terms must be formalized by rules analogous to the (∃E =), (∃β), (∃η) rules in Remark 4.4 (where, again, A is Tp and the dependence of C on X, x must be made indirect as above). By this indirect dependence, though, one cannot have either projection.

5.14. Remark. (On first order 'there exists elimination' rules.) Following Remark 5.9.2, one may derive a first-order 'there-exists-elimination' rule by taking $K = B{:}Tp$. Then 5.11 proves

$$(w\exists E) \quad \frac{\langle b, a\rangle_A{:}(\exists x'{:}B.A) \qquad [x'{:}B, x{:}A] \vdash c{:}C}{(\langle b, a\rangle_A)\, C(\lambda x'{:}B.(\lambda x{:}A.c)){:}C} \quad \text{with } x', x \notin FV(C).$$

By Corollary 5.12, also a corresponding ($w\exists\beta$) is derivable and one may obtain the *first* projection. Just take $c \equiv x'$ and $C \equiv B$, which is possible, as B cannot depend on x', x. Then, by 5.11 and 5.12, one has

$$(\langle b, a\rangle_A)\, C(\lambda x'{:}B.(\lambda x{:}A.x)) = b{:}B.$$

However, these rules are weak and there is no way to derive the second projection, in contrast to the (∃E), (∃β) rules given in §4 for the first-order case, since A may depend on x', and, hence, one cannot set $C \equiv A$. Similarly, one cannot derive an analogue to (∃η).

Note also that, if one does not have predicates with free (term) variables, then ($w\exists$E), with no proof terms, is exactly the first-order rule for existential elimination in natural deduction.

Mathematical extensions are handled similarly as for $\mathbf{PN}_1$, by taking into account that one has higher order arities and sorts.

6. Semantic interpretation

As already mentioned, we have just presented the core theory of 'propositions-as-types'. However, this is where (most of) the semantic challenges come in. The point now is to define or construct sets, categories, objects or the like, in such a way that the formal, linguistic, notions introduced so far will be enriched by meaning and consistency over these structures, which have an independent foundation from type theory.

It is clear that if we can do so for the last theory of types examined, then we would also be able to interpret the other theories, as the latter is an extension of all of them. Moreover, λP_2 presents a very challenging problem: the definition of types is *impredicative*, or $\mathbf{PN}_2$ is an impredicative theory of types or propositions. In short, when defining proper second-order types, the quantified types are defined by using quantification over all types, including the one which is being defined. Indeed, recall

$$(\forall^2\mathrm{I}) \quad \frac{[X:Tp] \vdash b:B}{(\lambda X:Tp.b):(\forall X:Tp.B)}$$

$$(\forall^2\mathrm{E}) \quad \frac{f:(\forall X:Tp.B) \qquad A:Tp}{(fA):[A/X]B}$$

where $(\lambda X:Tp.a):(\forall X:Tp.B)$ also summarizes that $(\forall X:Tp.B):Tp$ (see §5).

Since we only want to focus on the semantic understanding of impredicativity and keep the presentation elementary, in the sequel we will restrict ourselves to Girard's system F, i.e. to the second-order fragment of $\mathbf{PN}_2$, and we will not discuss here the interpretation of dependent types, i.e. $\mathbf{PN}_1$. To be precise, system F, as a pure theory, is equivalent to $\lambda\mathbf{P}_2$. The ideas presented in this paper have brought again to the limelight the 'realizability interpretation' of impredicative type theory, as pointed out in Hyland (1987).

A full categorical account of dependent types, including the theory of constructions, can be now found in Hyland and Pitts (1987) (see also Ehrhard, 1988; and Robinson, 1989).

Call now $\mathbf{M}$ the interpretation of Tp. Then ($\forall^2$I) roughly means that the interpretation of $(\forall X:Tp.B)$ must be an element of $\mathbf{M}$, even though its definition may use $\mathbf{M}$ itself, as $(\forall X:Tp.B)$ contains Tp. Thus, it may not seem possible to construct the entire set or structure $\mathbf{M}$ stepwise, i.e. by defining its elements first, as we know them only after knowing all of $\mathbf{M}$ (see (0) below). The following method for understanding this circularity exemplifies a crucial point of 'meaningful (and informative) interpretations': the impredicative calculus $\lambda\mathbf{P}_2$ is given meaning in the familiar powerset of a (countable) set. As discussed in the introduction, this is meant to provide a conceptual environment for an explanation of impredicative Type Theory, since we understand it independently and better, we claim, than $(\forall X:Tp.B):Tp$.

Semantics of type assignment and of typed terms. A distinction must be made when discussing the semantics of systems like those described in the previous sections. In §2 types are assigned to type-free terms: rule ($\to$I), say, derives type schema $(A \to A)$ for the term $\lambda x.x$, the functional program computing the identity function. $\lambda x.x$, though, carries no type information: it is a type-free term and ($\to$I) tells us that it has all types whose schema is $(A \to A)$.

Thus, the meaning of an assignment of types must be given over a model for the type-free calculus, $(D, \cdot)$, as one needs to interpret first untyped terms. Then a derivation $\vdash a:A$ will hold in D iff the interpretation in D of the type-free term a is in the interpretation of the type A as a subset (or a substructure) of D.

We will not get into the semantics of type assignment as this is a well settled part of the connection between type-free and classically typed λ-calculus (Barendregt *et al.*, 1983, first proved a completeness result; Hindley, 1983, gives an alternative tidy presentation; see Coppo, 1983, Longo and Martini, 1986, Hindley and Seldin, 1986, Mitchell, 1986, 1988, for further references).

In contrast to λ_0, the terms of the calculi of proofs of $\mathbf{PN}_1$ and $\mathbf{PN}_2$ are typed. Then rules such as ($\forall$I) specify their types (or select the legal terms of the typed language together with their types). For example, by ($\forall$I) in $\mathbf{PN}_1$ one may deduce

$$(\lambda x{:}A.x){:}(A \to A)$$

from the derivation $[x:A] \ldots x:A$; that is for type A, $\lambda x{:}A.x$ is the typed identity of type $A \to A$.

In the second order case, $\lambda\mathbf{P}_2$, the metalinguistic or implicit derivation we just made can be explicitly formalized. That is the deduction

'for any type A, $\lambda x{:}A.x$ is the typed identity of type $A \to A$'

becomes $$[X{:}Tp] \ldots (\lambda x{:}X.x){:}(X \to X)$$

and ($\forall^2$I) gives $$(\lambda X{:}Tp.\lambda x{:}X.x){:}(\forall X{:}Tp.X \to X).$$

Next we will discuss the meaning of the later syntactic constructs.

The model investigated is built out of a type-free structure: Kleene's applicative $(\omega, \cdot)$. However, the construction works over any partial combinatory algebra, and, thus, on any model of the type-free λ-calculus; in this case the model would yield also a complete semantics for type assignment, as described by $\mathbf{PN}_0$ (see Scott, 1976, Hindley, 1983, where the modest sets are called 'the quotient semantics of types').

Types as modest sets. There are several general definitions of what the semantics of impredicative II order types should be (see Bruce *et al.*, 1986, Seely, 1986). In this section, we first consider a very simple interpretation of $\lambda\mathbf{P}_2$, which satisfies the requirements in Bruce *et al.* (1986) and is based on the early work of Girard (1971) and Troelstra (1973). In §§6 and 7, though, we also set the basis for a more elaborate understanding of $\lambda\mathbf{P}_2$ as described in §8. There the meaning of (the impredicative core of) $\mathbf{PN}_2$ will be better given by providing a category-theoretic explanation of the (impredicative part of) the previous construction. This is done within a model of IZF (Intuitionistic Set Theory), which allows a 'set-theoretic' understanding of a calculus which has no classical set-theoretic model. Let

us informally summarize the main steps of the abstract notion and hint the 'naïve' interpretation. Later we will construct the model, which is informative even without a general presentation of what a model should be (see Asperti and Longo, 1991 for this). (**Notation**: unless otherwise stated, $f: A \to B$ or $f \in (A \to B)$ is an arbitrary function from the set (or class) A to the set (or class) B. Indeed, we first argue in the category **Set** of sets and functions.)

The intended operational meaning of second-order types is that a term $(\lambda X: Tp\,.\,a)$ of type $(\forall X: Tp\,.\,B)$ is a procedure which takes a type C, say, as input and gives as output the term $[C/x]a$ of type $[C/x]B$. This is exactly what $(\forall^2\mathrm{E})$ says. As for the interpretation over a mathematical structure, assume that Tp is interpreted as the set **M**. Then:

(1) Write first $(\forall X: Tp\,.\,B)$ as $\forall(\lambda X: Tp\,.\,B)$, i.e. understand $\forall$ as a map from $(Tp \to Tp)$ to Tp, as it turns the map $\lambda X: Tp\,.\,B$ in $(Tp \to Tp)$ into a type. Thus the interpretation of $\forall$ should be a map from $\mathbf{M} \to \mathbf{M}$ to **M**.

(2) Define $[c]_\xi$ as the interpretation of the type or term c under environment ξ and set $f = [(\lambda X: Tp\,.\,a)]_\xi$. Then, by $(\forall^2\mathrm{E})$ and the way we understood it before,

$$f: \mathbf{M} \to \bigcup_{C \in \mathbf{M}} [B]_{\xi[C/X]}, \quad \text{with } f(C) \in [B]_{\xi[C/X]}.$$

This idea corresponds to the notion of dependent or indexed product in set theory: any f as above is an element of the set-theoretic product $\Pi_{C \in \mathbf{M}} [B]_{\xi[C/X]}$, indexed over the set **M** of types. In short, assume that $g \in (\mathbf{M} \to \mathbf{M})$ interprets $\lambda X: Tp\,.\,B: (Tp \to Tp)$, then

$$f \in \prod_{C \in \mathbf{M}} g(C) = \prod_{C \in \mathbf{M}} [B]_{\xi[C/X]}$$

and $\Pi_{C \in \mathbf{M}} [B]_{\xi[C/X]}$ is an element of **M**, i.e. a type.

Thus, one has to find a set, **M**, with the following closure property: it must be closed by products indexed over the set **M** itself. That is, for all functions $g: \mathbf{M} \to \mathbf{M}$ we want

$$(0) \quad \prod_{e \in \mathbf{M}} g(e) \in \mathbf{M}.$$

This property interprets in **M** the circularity of $(\forall X: Tp\,.\,B): Tp$, the crucial judgement of impredicative Type Theory. In categorical terms (w.r.t. the category **Set**), this means that we want a full subcategory **M** of **Set** which is cartesian closed, has all products indexed by the set $\mathbf{M}_0$ of objects of **M** and the embedding of **M** in **Set** preserves function spaces, finite products and $\mathbf{M}_0$-indexed products. (For a general definition of categorical model for system F, along this naïve idea, can be found in Pitts, 1987, and Meseguer, 1989; see also Asperti and Longo, 1991.) Reynolds (1984) has shown that this naïve approach, unless given in a suitable categorical environment, leads to a paradox, by an informative result on models of λP_2 (see also Reynolds and Plotkin, 1988). The following is a simple cardinality argument.

Assume that a naïve interpretation of λP_2 could be given, as previously. That is, that there is a set **N** such that

$$A \to B \in \mathbf{N} \quad \text{for } A, B \in N$$

$$\prod_{e \in \mathbf{N}} g(e) \in \mathbf{N} \quad \text{for } g: \mathbf{N} \to \mathbf{N}.$$

The interpretation would be non-trivial if there were a set $A \in \mathbf{N}$ such that $\# A > 1$. Define then $F: \mathrm{Ord} \to \mathbf{N}$ by

$F(0) = A$

$F(\alpha+1) = F(\alpha) \to F(\alpha)$

$F(\gamma) = \Pi_{B \in \mathbf{N}} g(B)$ for a limit ordinal γ, where $g(B) = \textit{if } B \in \{F(\alpha) \mid \alpha < \gamma\} \textit{ then } B, \textit{ else } A$.

By this $\alpha < \beta \Rightarrow \# F(\alpha) < \# F(\beta)$, which is impossible, as Ord is not a set. Therefore for any $A \in \mathbf{N}$, one has $\# A \leqslant 1$. And the model trivializes (similarly as in any naïve attempt to find in **Set** a model for the type-free λ-calculus).

However, both the simple arguments above and the more complex one in Reynolds (1984) are given in classical ZF and do not hold in IZF.

In §8, we prove that the naïve interpretation of second-order types as indexed products can be recovered provided that the model is turned into an internal category of a suitable topos theoretic model of IZF, namely the effective topos **Eff** of Hyland (see Hyland, 1982, 1987; Pitts, 1987). In that frame, products are isomorphic to intersections (Theorem 7.4). For convenience, in the exposition, we may avoid any explicit use of Topos Theory by using as categorical frame the category ω-**Set** below, which has a 'concrete' description as a category of sets with a realizability relation (the correctness of this replacement is explained in §8). By this we intend to provide an elementary categorical justification of the currently well known interpretation of types as quotient subsets of ω, the natural numbers. Let us first review this interpretation of types, simply and *modestly*, as quotient sets, essentially due to Kreisel, Girard, Troelstra and Scott. The current understanding of the higher order types by quotient sets and internal categories, which we develop here, has been first suggested by Moggi and widely developed by several authors in category theory (Rosolini, 1986; Hyland, 1987; Hyland *et al.*, 1987; Carboni *et al.*, 1987; Bainbridge *et al.*, 1987; Robinson, 1989; Asperti and Longo, 1991, ...).

The objects of the category **PER** below are equivalence relations on subsets of the natural numbers or **partial equivalence relations** (p.e.r.'s). Morphisms are defined by Kleene's application: $n \cdot p$ is the result of the application of the nth partial recursive function to the number p. By $n \cdot p \in A$ we always mean that $n \cdot p$ is defined. $\langle , \rangle$ with inverses pr_1, pr_2 is any (effective) and bijective coding of pairs.

(**Notation:** let A be a symmetric and transitive relation on ω. Set then:
$\mathbf{n}\, \mathbf{A}\, \mathbf{m}$ iff n is related to m by A, $\mathbf{dom(A)} = \{n \mid n A n\}$,
$\{\mathbf{n}\}_{\mathbf{A}} = \{m \mid m A n\}$ the equivalence class of n w.r.t. A, $\mathbf{Q(A)} = \{\{n\}_A \mid n \in \mathrm{dom}(A)\}$).

6.1. Definition. The category **PER** has as
objects: $A \in \mathbf{PER}$ iff A is a symmetric and transitive relation on ω,
morphisms: $f \in \mathbf{PER}[A, B]$ iff

$$f: Q(A) \to Q(B) \quad \text{and } \exists n \forall p\, (pAp \Rightarrow f(\{p\}_A) = \{n \cdot p\}_B).$$

Morphisms in **PER** are 'computable' in the sense that they are fully described by partial recursive functions which are total on the domain of the source relation.

Note also that the intersection of any collection $\{A_i\}_{i \in I}$ of objects in **PER** is still in **PER**, by viewing them as sets of pairs (of numbers), that is,

$$n(\bigcap_{i \in I} A_i)\, m \quad \text{iff } \forall i \in I (n A_i m).$$

The model may be defined by saying how to interpret Tp, individual types and type constructors. Then the interpretation of terms, as 'functions' in their types, is given by describing the applicative behaviour of elements of p.e.r.'s, i.e. of equivalence classes. A key point is the mathematical meaning, given in (5), of the application of a term to a type (rule ($\forall^2$E)).

6.2. Definition. (the λ_2-model of partial equivalence relations):
(1) $[Tp] = \mathbf{PER}$,
(2) $\to : (\mathbf{PER} \times \mathbf{PER}) \to \mathbf{PER}$ is the function such that

$$m(R \to S)\, n \Leftrightarrow \forall p, q\, (pRq \Rightarrow m \cdot p\, Sn \cdot q),$$

(3) $\forall : (\mathbf{PER} \to \mathbf{PER}) \to \mathbf{PER}$ is the function such that $\forall(F) = \bigcap_{R \in \mathbf{PER}} F(R)$,
(4) $\mathrm{eval}_{R,S}(\{n\}_{R \to S}, \{m\}_R) = \{n \cdot m\}_S$ where $R, S \in \mathbf{PER}$,
(5) $\mathrm{eval}_F(\{n\}_{\forall(F)}, R) = \{n\}_{F(R)}$ where $F : \mathbf{PER} \to \mathbf{PER}$ (polymorphic application).

The definition of polymorphic application in 6.5 was suggested by Moggi in a discussion on computer mail (Moggi, 1986). This model and similar ones (see Mitchell, 1986; Breazu T. and Coquand, 1987) have been also used to answer consistency questions for extensions of λP_2 (see §9), for which it is essential to allow models also with empty types (see Meyer *et al.*, 1987). However, the first basic ideas for a λ_2-model $\mathbf{HEO}_2$ (with all types non-empty), based on partial equivalence relations, is due to Girard (see also Troelstra, 1972).

What needs to be done now is to relate the above interpretation of polymorphic types as intersections (in 6.4) to the set theoretic intuition of polymorphic types as indexed products. Indexed over **PER** itself, as it should be clear by now. Observe first that there is no way to look at **PER** as an object of **PER** itself (for cardinality reasons!). Indeed, we never required, in the formal theory, Tp to be a type, although in some models (e.g. in Amadio *et al.*, 1986, or Taylor, 1986) this is so. In any case, we must be able to deal with Tp in some 'uniform' way, as we need to interpret universal quantification over types *and* over Tp. For this purpose we consider the following extension of **PER**.

6.3. Definition. The category ω-**Set** has as
objects: $(A, \vdash) \in \omega$-**Set** iff

$$A \text{ is a set and } \vdash \subseteq \omega \times A, \text{ i.e. } \vdash \text{ is a relation in } \omega \times A, \text{ s.t. } \forall a \in A\, \exists n \vdash a.$$

morphisms: $f \in \omega$-**Set**$[A, B]$ iff

$$f : A \to B \text{ and } \exists n \forall a \in A\, \forall p \vdash_A a \quad n \cdot p \vdash_B f(a) \quad (\textbf{notation: } n \vdash_{A \to B} f).$$

Similarly as for **PER**, each morphism in ω-**Set** is 'computed' by a partial recursive function, which is total on $\{p \mid p \vdash_A a\}$, for each $a \in A$.
(**Notation:** we say that 'p realizes a' iff $p \vdash_A a$ in $(A, \vdash)$; in $\vdash_A$ we may omit A if there is no ambiguity.)

The category **PER** is isomorphic to a full subcategory of ω-**Set**. In fact, for every partial equivalence relation (per) A we can define an ω-set $In(A) = (Q(A), \epsilon_A)$, where $Q(A)$ are the equivalence classes of A, as disjoint subsets of ω, and ϵ_A is the usual membership relation

restricted to $\omega \times Q(A)$. Clearly, this defines a realizability relation in the sense of Definition 6.3 and the embedding is full.

Since categorical constructions are defined up to isomorphisms, it is more convenient to consider a full subcategory of ω-**Set** whose objects are exactly the ω-sets X for which there exists a per A s.t. X is isomorphic to $In(A)$. This category is defined as follows.

6.4. Definition. The category **M**, of the **modest sets**, is the subcategory of ω-**Set** whose objects $(X, \vdash)$ have a single-valued relation '$\vdash$'; that is, '$\vdash$' satisfies

$$\text{(SV)} \quad \forall a, b \in X\, n \vdash a \wedge n \vdash b \Rightarrow a = b.$$

Morphisms are defined as for ω-sets.

If A is a p.e.r., then $In(A)$ is clearly a modest set, since the elements of $Q(A)$ are disjoint (non-empty) subsets of ω. Therefore, In defines an embedding from **PER** into **M**. Conversely, for every modest set $(X, \vdash)$ we define a p.e.r. $Out(X)$, by

$$n\, Out(X)\, m \Leftrightarrow \exists a \in X\, n, m \vdash a.$$

In view of (SV), (In, Out) is an equivalence between the categories **PER** and **M**, which cuts down to an isomorphism between p.e.r.'s and ω-sets of the form $(Q(A), \epsilon_A)$.

With the help of **M** we explain the relation among certain products in ω-**Set** and the interpretation of universal quantification in the model defined in 6.2. The crucial point is to look at **PER** also as an *object* of ω-**Set** by setting:

$$\mathbf{M_0} = (\mathbf{PER}, \vdash_M) \in \omega\text{-}\mathbf{Set} \quad \text{where } \vdash_M = \omega \times M.$$

(If there is no ambiguity, we also call **PER** the set of objects of the category **PER**.) This idea will be developed more rigorously in §8, where we turn **PER** into an *internal category* $\mathbf{M}'$ of ω-**Set**, whose 'object of objects' is $\mathbf{M_0}$, and prove that the product below, or the intersection of p.e.r.'s, is the object component of an internal product functor.

There is a straightforward way to define a naïve product in ω-**Set**.

6.5. Definition. Let $(A, \vdash_A) \in \omega$-**Set** and $g: A \to \omega$-**Set**. Define then the ω-set $([\Pi_{a \in A} g(a)], \vdash_{\Pi, g})$ by

(1) for $f \in \Pi_{a \in A} g(a)$, $n \vdash_{\Pi, g} f$ iff $\forall a \in A\, \forall p \vdash_A a\ n \cdot p \vdash_{g(a)} f(a)$.
(2) $f \in [\Pi_{a \in A} g(a)]$ iff $f \in \Pi_{a \in A} g(a)$ and $\exists n\, n \vdash_{\Pi, g} f$.

Remark. Clearly, $([\Pi_{a \in A} g(a)], \vdash)$ in 6.5 is a well-defined object of ω-**Set**. In categorical terms, $([\Pi_{a \in A} g(a)], \vdash)$ is an indexed product of ω-**Set** 'indexed over' itself. However, we may avoid, here, the general notion of indexed category and indexed product and keep the presentation fairly elementary.

What is nice is that, as soon as the range of g is restricted to **M**, this product lives in **M**:

6.6. Theorem. *Let* $(A, \vdash_A) \in \omega$-**Set** *and* $g: A \to \mathbf{M}$.
Then $([\Pi_{a \in A} g(a)], \vdash_{\Pi, g}) \in \mathbf{M}$.

Proof. Let $\vdash_{\Pi, g}$ be defined as in 6.5. All that we need to show is that, under the assumptions, $\vdash_{\Pi, g}$ satisfies (SV) above.

Assume that $n \vdash_{\Pi, g} f \wedge n \vdash_{\Pi, g} h$. We show that $\forall a \in A\ f(a) = h(a)$ and, thus, that $f = h$. By definition $\forall a \in A\, \forall p \vdash_A a\ \ n \cdot p \vdash_{g(a)} f(a) \wedge n \cdot p \vdash_{g(a)} h(a)$ and, thus, $f(a) = h(a)$ since, for all a, $\vdash_{g(a)}$ satisfies (SV) (and any a in A is realized by some number). Δ

Suppose, in a first approximation, that types are interpreted by modest sets, while Tp, the formal collection of types, is interpreted by the ω-set $\mathbf{M}_0 = (\mathbf{PER}, \vdash_M)$, then the naïve interpretation of universally quantified types is

$$[\forall x{:}Tp\,.\,B]_\xi = [\prod_{a \in [Tp]} [B]_{\xi[a/x]}]. \tag{6.7}$$

Hence

$$[\forall X{:}Tp\,.\,B]_\xi = [\prod_{A \in M} [B]_{\xi[A/X]}] \in \mathbf{M}.$$

In §7, we show the relation between this notion of indexed product (over $\mathbf{M}_0$) and intersection of p.e.r.'s, used in the model defined 6.2.

7. Products and intersections

Recall that each $X \in \mathbf{M}$ corresponds exactly to the partial equivalence relation Out(X). By an abuse, we identify objects X of $\mathbf{M}$ and the corresponding p.e.r. Out(X), when this helps in the presentation. Then, the product in 6.6, for $(A, \vdash_A) \in \omega\text{-}\mathbf{Set}$ and $g{:}A \to \mathbf{M}$, may be equivalently written as

$$\begin{aligned} n[\prod_{a \in A} g(a)]\, m \quad &\text{iff } \exists f \in \prod_{a \in A} g(a)\, \forall a \in A\, \forall p, q \vdash_A a \quad n \cdot p, m \cdot q \vdash_{g(a)} f(a) \\ &\text{iff } \forall a \in A \quad \forall p, q \vdash_A a \quad n \cdot p\ g(a)\, m \cdot q. \end{aligned} \tag{7.1}$$

Observe now that the first-order types are interpreted in Troelstra (1973) as follows:

$$n[\forall x{:}A\,.\,B]_\xi\, m \quad \text{iff } \forall p, q (p[A]_\xi\, q \Rightarrow m \cdot p[B]_{\xi[\{p\}_A/x]}\, n \cdot q) \tag{7.2}$$

(where $\{p\}_A$ replaces x in ξ).

That is, '$\ldots m \cdot p$ is equivalent to $n \cdot q$ in the relation $[B]_{\xi[\{p\}_A/x]}$'. When A (or its interpretation) is in $\mathbf{M}$ and g is $g(\{p\}_A) = [B]_{\xi[\{p\}_A/x]}$, then 7.1 and 7.2 are identical. As for the more challenging case, i.e. when $A = Tp$, Girard and Troelstra proposed to interpret $\forall X{:}Tp\,.\,B$ as in 6.2.3, i.e. by

$$n[\forall X{:}Tp\,.\,B]_\xi\, m \quad \text{iff} \quad \text{for all } C \in \mathbf{PER}\ n[B]_{\xi[C/X]}\, m. \tag{7.3}$$

That is by

$$n[\forall X{:}Tp\,.\,B]_\xi\, m \quad \text{iff} \quad n(\bigcap_{C \in \mathbf{PER}} [B]_{\xi[C/X]})\, m.$$

We are now in the position to explain how this intersection fits with the (naïve) meaning of universal quantification as indexed product. (This also hints (naïvely) at a unified framework, which includes the first order quantification, in view of the relation between (7.1) and (7.2).) Surprisingly enough, the intersection over $\mathbf{M}$ in 7.3 is indeed a product. This may be shown when working in ω-**Set**, as 'frame' category. More precisely, Theorem 7.4 below shows that the intersection and the product in 6.5, for $\mathbf{M}_0 = (\mathbf{PER}, \vdash_M) \in \omega\text{-}\mathbf{Set}$ and $g{:}\mathbf{M}_0 \to \mathbf{M}$, are isomorphic in $\mathbf{M}$. Thus, the intersection interpretation of second-order

types fits with their interpretation as products. Moreover, by 6.6 and 7.4, the meaning of functional types (more generally, of first-order quantification) and second-order quantification turns out to be given by the same kind of product. §8 will complete the required category-theoretic explanation of the notion of product we are talking about.

7.4. Theorem. *Let* $(A, \vdash_A) \in \omega$-**Set** *be such that* $\vdash_A = \omega \times A$ *and let* $g: A \to \mathbf{M}$. *Then*

$$[\prod_{a \in A} (g(a))] \cong \mathrm{In}(\bigcap_{a \in A} g(a)), \quad \textit{in } \mathbf{M},$$

where $g(a)$ *is identified with* $\mathrm{Out}(g(a)) \in \mathbf{PER}$.

Proof. Let $S = \bigcap_{a \in A} g(a) \in \mathbf{PER}$. We already know that both $[\prod_{a \in A} g(a)]$ and $\mathrm{In}(S)$ are in $\mathbf{M}$. Thus we need to define a bijection $G: \mathrm{In}(S) \to [\prod_{a \in A} g(a)]$ and prove that it is realized with its inverse.

Let $G(\{n\}_S) = \lambda a \in A . \{n\}_{g(a)}$. Clearly, $G(\{n\}_S) \in \prod_{a \in A} g(a)$ and G is well defined, as $\{n\}_S = \{m\}_S$ implies, for all $a \in A$, $ng(a)\, m$ and, hence, $\{n\}_{g(a)} = \{m\}_{g(a)}$.

Consider now any index k such that $k \cdot p \cdot q = p$, for all $p, q \in \omega$. Then $(k \cdot n)$ realizes $G(\{n\}_S)$, since

$$\forall a \in A \; \forall q \vdash_A a \quad k \cdot n \cdot q = n \in \{n\}_{g(a)} = G(\{n\}_S)(a),$$

and k realizes G. It is easy to observe that G is injective. Let us prove that G is surjective. If $h \in [\prod_{a \in A} g(a)]$, then, by definition, $\exists m \vdash_{\Pi, g} h$, that is,

$$\exists m \; \forall a \in A \; \forall q \vdash_A a \quad m \cdot q \vdash_{g(a)} h(a) \quad \text{or, equivalently,}$$
$$\exists m \; \forall a \in A \; \forall q \in \omega \quad h(a) = \{m \cdot q\}_{g(a)}, \quad \text{as } \vdash_A = \omega \times A.$$

Thus, for $n = m \cdot 0$, we have $\forall a \in A \; ng(a)\, n$, i.e. nSn. In conclusion, $\forall a \in A \; G(\{n\}_S)(a) = \{n\}_{g(a)} = h(a)$, that is $G(\{n\}_S) = h$. Therefore G^{-1} exists and it is realized by any index p such that $p \cdot m = m \cdot 0$, for all $m \in \omega$. Δ

One of the crucial steps in the construction of the isomorphism G in the theorem, i.e. the definition

$$G(\{n\}_S)(a) = \{n\}_{g(a)}, \quad \text{for } S = \bigcap_{a \in A} g(a),$$

is exactly the definition of 'polymorphic' application in 6.2.5. Indeed, take $(A, \vdash_A) = \mathbf{M}_0$. Then, when universally quantified types are interpreted as intersections, terms of these types are interpreted as equivalence classes in the intersections. Thus $G(\{n\}_S)(a) = \{n\}_{g(a)}$ tells us how to apply $\{n\}_S$ to (the interpretation of) a type a.

Next we prove that the intersection is the object component of a product functor (over $\mathbf{M}_0$) for the intended internal category. This provides a categorical justification of $\lambda\mathbf{P}_2$, by using the notion of internal category. As we do not have the space to get into a complete treatment of indexed categories, we refer to Hyland (1987) and Ehrhard (1988) for the categorical understanding of both first- and second-order quantification and, in the following section, we focus only on the crucial impredicative case, the second-order products or quantification over types.

8. Π is the right adjoint of the diagonal functor

As already mentioned, the point of the present interpretation is its naïve 'set-theoretic' flavour, a helpful aspect in applications to the denotational semantics of programming constructs (see, for example, Mitchell, 1986; Bruce and Longo, 1988). (Intuitionistic) set theory could be used, because the internal category we construct lives inside a model of IZF, the Effective Topos, which extends Kleene's realizability interpretation of intuitionistic logic.

It is an easy exercise to check that both the **M** and **ω-Set** are Cartesian closed. A key fact is that **M** is equivalent to an internal category of **ω-Set**, where an internal category is a pair of objects, one representing the collection of objects, such as $\mathbf{M}_0$, the other the morphisms, and a few morphisms between them (see 8.4 below; for example, a category is **small** when it is an internal category of **Set**, i.e. its collections of objects and morphisms are *sets*). The categorical constructions in **ω-Set** that will be used in the sequel are gathered in the following remark. (The formal treatment has been quite elementary up to this point: it will be based on some more technicalities in this final section.)

8.1. Remark. In **ω-Set** products are defined in the obvious way: just use a coding of pairs for the natural numbers. 6.3 suggests how to define function spaces:

$$[A \to B] = (\{f : A \to B \mid f \in \omega\text{-}\mathbf{Set}[A, B]\}, \vdash_{A \to B}).$$

Observe that $[A \to B]$ is also obtained from $([\Pi_{a \in A} g(a)], \vdash)$ in 6.5, when g is constantly equal to B.

The terminal object is simply $(\mathbf{1}, \vdash_1)$, where $\mathbf{1}$ is the singleton set and $\vdash_1 = \omega \times \mathbf{1}$. Equalizers are given by

$$(\{a \in A \mid f(a) = g(a)\}, \vdash_{fg}) \quad \text{where } n \vdash_{fg} a \quad \text{iff } n \vdash_A a.$$

Moreover, **ω-Set** is just a 'piece' of Hyland's Effective Topos **Eff**, a model for IZF.

8.2. Remark. **M** and **ω-Set** may be characterized within **Eff**, namely **M** is equivalent to the category of 'effective objects' and **ω-Set** is equivalent to the category of '$\sim\sim$-separated objects' (Hyland, 1982; Rosolini, 1986). The embedding functor from **ω-Set** to **Eff** is easily given: just take $(A, \vdash_A)$ to $(A, =_A)$ where

$$a =_A b := \text{if } a = b \text{ then } \{n \mid n \vdash_A a\} \text{ else } \varnothing.$$

Recall now that, given a category **C** and an object $A \in \mathbf{C}$, $\mathbf{C}/A$, the **slice over** A, is the comma category, whose objects are the morphisms with target A. Corollary 8.3 below proves that **ω-Set** is 1CCC and its embedding in **Eff** preserves such a structure, i.e. the full embedding of any **ω-Set**$/A$ in **Eff**$/A$ preserves the CCC structure. Thus, the kind of 'constructions' we consider commute w.r.t. the embedding functor from **ω-Set** to **Eff** and, hence, we can safely use **ω-Set** instead of **Eff**.

8.3. Corollary (to Hyland, 1982). *Both* **M** *and* **ω-Set** *are locally Cartesian closed categories, where finite limits and* Π*-types are 'computed' as in* **Eff**.

Proof. The category of effective objects is a full sub-1CCC of **Eff** (see Hyland, 1982; §7),

i.e. it is a full subcategory of **Eff** and inherits the structure of 1CCC from **Eff**. Since **M** and the category of effective objects are equivalent (and the 1CCC structure of a category is preserved by equivalences), **M** inherits the 1CCC structure via the equivalence. A similar argument can be carried out for ω-**Set** (use Hyland, 1982; §6]). Δ

In particular the category **Cat**(ω-**Set**) of internal categories and internal functors of ω-**Set** is a full sub CCC of **Cat**(**Eff**). We now explicitly give the internal version $\mathbf{M}' \in$ **Cat**(ω-**Set**) of **M**.

Definition. $\mathbf{M}' = (\mathbf{M}_0, \mathbf{M}_1, \mathrm{dom}^M, \mathrm{cod}^M, \mathrm{id}^M, \mathrm{comp}^M)$ is given by

1, $\mathbf{M}_0 = (\mathbf{PER}, \vdash_M)$ where $\vdash_M = \omega \times M$ (cf, before 6.5),
2, $\mathbf{M}_1 = (\{\langle A, \{n\}_{A\to B}, B\rangle \mid A, B \in \mathbf{M}, n(A \to B)n\}, \vdash_1)$ where $m \vdash_1 \langle A, \{n\}_{A\to B}, B\rangle$ iff $m(A \to B)n$ (in the sense of 6.2.2),
3, $\mathrm{dom}^M(\langle A, \{n\}_{A\to B}, B\rangle) = A$,
4, $\mathrm{cod}^M(\langle A, \{n\}_{A\to B}, B\rangle) = B$,
5, $\mathrm{id}^M(A) = \langle A, \{n\}_{A\to A}, A\rangle$ where $n = \lambda xx$ is a number for the identity function,
6, $\mathrm{comp}^M(\langle A, \{n\}_{A\to B}, B, \{m\}_{B\to C}, C\rangle) = \langle A, \{b\cdot m\cdot n\}_{A\to C}, C\rangle$ where $b = \lambda xyz.x(yz)$.

We have now to show that $\mathbf{M}'$ has $\mathbf{M}_0$-indexed products, namely that there is a right adjoint $\Pi : [\mathbf{M}_0 \to \mathbf{M}'] \to \mathbf{M}'$ to the diagonal functor $\Delta : \mathbf{M}' \to [\mathbf{M}_0 \to \mathbf{M}']$. Note that in set-theoretic terms, the internal category $[\mathbf{M}_0 \to \mathbf{M}'] = ([\mathbf{M}_0 \to \mathbf{M}_0], [\mathbf{M}_0 \to \mathbf{M}_1], \ldots)$ is the product of $\mathbf{M}_0$ copies of $\mathbf{M}'$, or equivalently is the category of functors from the discrete category, whose objects of objects is $\mathbf{M}_0$, to $\mathbf{M}'$. As for $[\mathbf{M}_0 \to \mathbf{M}_0]$, observe now that ω-**Set**$[\mathbf{M}_0, \mathbf{M}_0]$ is the collection of all set-theoretic functions, by the definition of morphisms in ω-**Set** and of $\vdash_M$ in $\mathbf{M}_0 = (\mathbf{PER}, \vdash_M)$. However, ω-**Set**$[\mathbf{M}_0, \mathbf{M}_1]$ is much smaller.

8.5. Lemma. *If* $\langle F, \tau, G\rangle \in \omega$-**Set**$[\mathbf{M}_0, \mathbf{M}_1]$ *then* $\exists n\, \forall A \in \mathbf{PER}\, \tau(A) = \{n\}_{F(A)\to G(A)}$.

Proof. Suppose that $m \vdash \langle F, \tau, G\rangle$. Since $0 \vdash A$ for any $A \in \mathbf{PER}$, then take $n = m\cdot 0$ and observe that $n \vdash \langle F(A), \tau(A), G(A)\rangle$. Thus $\tau(A) = \{n\}_{F(A)\to G(A)}$. Δ

8.6. Remark. As a matter of fact, some function spaces in ω-**Set** are surprisingly small. For example, while for all $A \in \omega$-**Set**, ω-**Set**$[A, \mathbf{M}_0]$ contains all functions from A to **PER**, when B is a modest set, ω-**Set**$[\mathbf{M}_0, B]$ is just the set of the constant functions from **PER** to B. The reader may have recognized in this a consequence of the validity of the Uniformity Principle (UP) in **Eff**. Indeed, this principle was implicitly used in 7.4. The relevance of (UP) is stressed in Rosolini (1986), where a generalized version of 7.4 is stated for any index object satisfying (UP).

In Definition 6.5 we gave a notion of 'product', $[\Pi_{a\in A} F(a)]$, over an ω-set $(A, \vdash_A)$ (thus, in particular, over $\mathbf{M}_0$). This is the natural candidate for the object part in the definition of the product functor. In view of Theorem 7.4, it is clear now that one can equivalently use intersection as a product. This makes the connection with the **PER** model in 6.2 more apparent (and simplifies the computation of a realizer for Π_1).

8.7. Definition. (The product functor Π.)

1, $\Pi_0 : [\mathbf{M}_0 \to \mathbf{M}_0] \to \mathbf{M}_0$, **the 'object' part of** Π, *is defined by* $\Pi_0(F) = \bigcap_{a \in \mathbf{PER}} F(a)$,
2, $\Pi_1 : [\mathbf{M}_0 \to \mathbf{M}_1] \to \mathbf{M}_1$ is defined as follows. If $m \vdash \langle F, \tau, G \rangle \in \omega\text{-}\mathbf{Set}[\mathbf{M}_0, \mathbf{M}_1]$, set

$$\Pi_1(\langle F, \tau, G \rangle) = \langle \Pi_0(F), \{m \cdot 0\}_{\Pi_0(F) \to \Pi_0(G)}, \Pi_0(G) \rangle.$$

8.8. Fact. Π in 8.7 is well defined. In particular, Π_1 is realized by any number p such that, for all m, $p \cdot m = m \cdot 0$.

Proof. By definition, if $F : \mathbf{M}_0 \to \mathbf{M}_0$, then $\Pi_0(F)$ is in $\mathbf{M}_0$. Π_0 is realized by (any index of) any total recursive function.

Therefore we only need to show that Π_1 is realized by p. Namely, that, if

(1) $m_1, m_2 \vdash \langle F, \tau, G \rangle$,
(2) $n_1 \Pi_0(F) n_2$,
(3) $A \in \mathbf{PER}$,
then one has $(p \cdot m_1 \cdot n_1) G(A) (p \cdot m_2 \cdot n_2)$. Indeed,
(4) $p \cdot m_1 = m_1 \cdot 0$ $(F(A) \to G(A)) m_2 \cdot 0 = p \cdot m_2$, by 1 and Lemma 8.5,
(5) $n_1 F(A) n_2$, by (2) and (3),
and thus $(p \cdot m_1 \cdot n_1) G(A) (p \cdot m_2 \cdot n_2)$ by (4), (5) and the definition of $(F(A) \to G(A)) \in \mathbf{PER}$. Δ

According to the categorical definition of product, given $B \in \mathbf{M}_0$ and $G \in [\mathbf{M}_0 \to \mathbf{M}']$, we must give a natural isomorphism $\varphi_{B,G}$ between $[\mathbf{M}_0 \to \mathbf{M}'][\Delta(B), G]$ and $\mathbf{M}'[B, \Pi_0(G)]$. (In informal lambda notation, one may write $\Delta(B) = \lambda A : \mathbf{M}_0 . B$, the constant function with value B.) Since $\mathbf{M}'$ is an internal category of ω-**Set**, the isomorphism $\varphi_{B,G}$ (and its inverse) must be realized by a natural number, computable from a realizers for B and G. Since B and G are realized by any number (index for a total recursive function), we only need to show that there is a number, independent of B and G, which realizes $\varphi_{B,G}$.

8.9. Definition (of $\varphi_{B,G}$). For $m \vdash \langle \lambda A : \mathbf{M}_0 . B, \tau, G \rangle \in \omega\text{-}\mathbf{Set}[\mathbf{M}_0, \mathbf{M}_1]$, set

$$\varphi_{B,G}(\langle \lambda A : \mathbf{M}_0 . B, \tau, G \rangle) = \langle B, \{m \cdot 0\}_{B \to \Pi_0(G)}, \prod_0(G) \rangle \in \mathbf{M}_1.$$

Conversely, set

$$\varphi^{-1}_{B,G}(\langle B, \{m\}_{B \to \Pi_0(G)}, \prod_0(G) \rangle) = \langle \lambda A : \mathbf{M}_0 . B, \lambda A : \mathbf{M}_0 . \{m\}_{B \to G(A)}, G \rangle \in \omega\text{-}\mathbf{Set}[\mathbf{M}_0, \mathbf{M}_1],$$

$\varphi_{B,G}$ and $\varphi^{-1}_{B,G}$ are realized by p and k (respectively), such that $p \cdot m = m \cdot 0$ and $k \cdot m \cdot n = m$ (cf, 7.4 and 8.8).

Just for curiosity, if one defines Π_1 by using $\Pi_0(F) = [\Pi_{a \in \mathbf{PER}} F(a)]$, then Π_1 is realized by s, with $s \cdot p \cdot q \cdot r = p \cdot r \cdot (q \cdot r)$, the other basic combinator of Combinatory Logic.

The reader may find a detailed analysis of the completeness properties of the internal category $\mathbf{M}'$, necessary to turn it into a model of the calculus of constructions, in Hyland (1987), Ehrhard (1988) and Robinson (1989). The definition of model for system F, by

internal categories, originally due to Moggi, is described and related to other approaches in Asperti and Longo (1991).

8.10. Remark. There is another property of the right adjoint that has to be verified in order to guarantee that substitution is well-behaved, namely the Beck–Chevalley condition (see Hyland, 1987; Hyland and Pitts, 1987). However, this condition is automatically true when quantification is restricted to $\mathbf{M}_0$, because the only right-adjoint required for the interpretation are those to the functors

$$\Delta_A : [A \to \mathbf{M}'] \to [A \times \mathbf{M}_0 \to \mathbf{M}']$$

for any object A in ω-**Set**. (For instance the right adjoint functor to $\Delta_{\mathbf{M}_0}$ applied the interpretation of a type expression $B(X, Y)$, with two type variables, gives the interpretation of $\forall Y{:}Tp\,.\,B$.)

9. Conclusion: methods and applications

Going back to the introductory discussion on 'denotation and meaning', one may argue that the syntax is largely used in the semantics: the partial recursive functions, which define the morphisms in **PER**, **M** and ω-**Set**, are exactly the lambda definable functions! However, one may use the good old λ_0 as a computing device and explain by this $\lambda\mathsf{P}_2$, as it is above. Moreover, one may hide computability in the background or avoid it, as the definition of all the three categories can be relativized to any (partial) combinatory algebra or model of the type-free combinatory logic in 3.7 (see Hyland, 1982; Longo and Moggi, 1990). Thus, one may take a model of λ_0 constructed by topological tools, as done in Scott (1972, 1976), and understand $\lambda\mathsf{P}_2$ in terms of continuous functions.

The case discussed in this paper is an example of an interesting interaction between structural meaning and formal systems. Mathematical structures suggested formal calculi of (typed) functions or the proof theory related to the various lambda calculi. For example, Girard's system F was invented for the proof theoretic investigation of second-order arithmetic, which is commonly understood as the formalization of analysis. System F, in turn, suggested the above-mentioned strong property of the partial equivalence relation on a (sufficiently rich) applicative structures, i.e. their closure under (suitable) products indexed over the set of p.e.r.'s itself, as well as its general category theoretic understanding (in the sense of §8 and Carboni *et al.*, 1987; Hyland, 1987; Hyland *et al.*, 1987; Pitts, 1987; Hyland and Pitts, 1987).

It is worth mentioning that this most recent history was developed by work in computer science. Even though they originated in logic, types as quotient sets or the various categories of 'modest sets' were strongly advocated by Dana Scott in various papers and lectures; without his work and stimulating activity, not much would be known about these structures. Moreover, the present model construction originated as an answer to a very reasonable question in the theory of programming, raised by Albert Meyer: can we extend $\lambda\mathsf{P}_2$ (which is the core of functional languages with explicit polymorphism) by a couple of axioms imposing that there are only two booleans? It is good in programming to know that there will be no rubbish in the type of booleans, i.e. in $\forall X{:}Tp\,.\,X \to (X \to X)$. Moggi, by using **PER** in Moggi (1986), showed that such an extension is consistent. As an exercise,

the reader may check that the interpretation in **M** of the type $\forall X{:}Tp\,.\,X \to X$ of the II order identity $\lambda X{:}Tp\,.\,\lambda x{:}X\,.\,x$ contains only the identity function (or the collection of its indices).

Further 'consistency questions' for interesting extensions of polymorphic languages, such as those related to the issues on 'inheritance' and subtypes, may be the most likely applications of the present interpretation of higher order types (see Bruce and Longo, 1990). This is because types are just (and modestly) sets of numbers and behave very naturally as far as set inclusion and related properties are concerned.

Acknowledgements

Longo is indebted to Roberto Amadio, P. L. Curien, Roger Hindley for numerous discussions. In particular, with P. L. Curien he had a chance to have long and helpful discussions on the categorical approach to higher order typing and on Moggi's understanding of 'polymorphic application'. Pino Rosolini made comments and suggested changes in several messages on computer mail.

References

(An extended and 'organized' bibliography may be found in Longo, 1988.)

Amadio, R., Bruce, K. B. and Longo, G. (1986) The finitary projections model and the solution of higher order domain equations. *IEEE Conference on Logic in Computer Science* (*LICS '86*). Boston, June 1986.

Amadio, R. and Longo, G. (1987) Type-free compiling of parametric types. *IFIP Conference on Formal Description of Programming Concepts* Ebberup (DK). Wirsing, ed. North-Holland.

Asperti, A. and Longo, G. (1991) *Categories, Types and Structures*: *An Introduction to Category Theory for the Working Computer Scientist*. MIT Press.

Bainbridge, E., Freyd, P., Scedrov, A. and Scott, P. J. (1987) *Functional Polymorphism*. University of Texas Institute on Logical foundations of Functional Programming, Austin.

Barendregt, H., Coppo, M. and Dezani, M. (1983) A filter lambda model and the completeness of type assignment. *J. Symb. Logic* **48** 931–40.

Böhm, C. and Berarducci, A. (1985) Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comp. Sci.* **39** 135–54.

Breazu-Tannen, V. and Coquand, T. (1987) Extensional models for polymorphism. TAPSOFT-CFLP, Pisa.

Bruce, K. and Longo, G. (1990) Modest models for inheritance and explicit polymorphism. *Proc. L.I.C.S. '88*, Edinburgh (*revised version*: *Info. & Comp.* **87**).

Bruce, K., Meyer, A. and Mitchell, J. (1986) The semantics of second order polymorphic lambda-calculus. To appear (Preliminary version: *Symposium on Semantics of Data Types*). Kahn, MacQueen and Plotkin, eds, *Springer Lecture Notes in Computer Science* **173**. Springer-Verlag, pp. 131–44.

de Bruijn, N. (1980) A survey of the project AUTHOMATH. In: Hindley and Seldin, eds, *To H. B. Curry*: *Essays in Combinatory Logic, Lambda Calculus and Formalism*. Academic Press.

Burstall, R. M. and Lampson, B. (1984) A kernel language for abstract data types and modules. In: Kahn, MacQueen and Plotkin, eds, *Symposium on Semantics of Data Types. Springer Lecture Notes in Computer Science* **173**. Springer-Verlag, pp. 1–50.

Carboni, A., Freyd, P. and Scedrov, A. (1987) A categorical approach to realizability of polymorphic types. *3rd A.C.M. Symp. Math. Found. Progr. Lang. Semantics, Springer Lecture Notes in Computer Science*. Springer-Verlag.

Cardelli, L. (1986) A polymorphic lambda-calculus with Type:Type. *Preprint*, Syst. Res. Center, Dig. Equip. Corp.

Cardelli, L. and Longo, G. (1989) A semantic basis for **Quest**. Dec-src report 5, Feb. 1990.

Constable, R. L. *et al.* (1986) *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall.

Coppo, M. (1984) Completeness of type assignment in continuous lambda-models. *Theor. Comp. Sci.* **29** 309–24.

Coquand, T. (1985) Une théorie des constructions. Thèse de 3ème cycle, Université Paris VII.

—— (1986) An analysis of Girard paradox. *IEEE Conference on Logic in Computer Science*. Boston, June 1986.

Coquand, T. and Huet, G. (1985) Constructions: a higher order proof system for mechanizing mathematics. Report 401 INRIA, presented at *EUROCAL '85*.

Ehrhard, T. (1988) A categorical semantics of constructions. *Proc. L.I.C.S. '88*, Edinburgh.

Feferman, X. Y. (1985) A theory of variable types. *Rev. Colombiana Matem.* **XIX** 95–105.

Girard, J. Y. (1971) Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In: J. E. Festand, ed, *2nd Scandinavian Logic Symposium*. North-Holland, pp. 63–92.

—— (1972) Interpretation fonctionelle et elimination des coupure dans l'arithmetic d'ordre superieur. Thèse de Doctorat d'Etat, Paris.

Harper, R., Honsell, F. and Plotkin, G. (1987) A framework for defining Logics. *L.I.C.S. '87*, Cornell, June 1987.

Hindley, R. (1969) The principal type-scheme of an object in Combinatory Logic. *Trans. Amer. Math. Soc.* **146** 22–60.

—— (1983) The completeness theorem for typing lambda-terms. *Theor. Comp. Sci.* **22** 1–17 (also *Theoret. Comput. Sci.* **22** 127–33).

Hindley, R. and Seldin, J. (1986) *Introduction to Combinators and Lambda-Calculus*. London Mathematical Society.

Huet, G. (1986) Formal structures for computation and deduction. Lecture Notes, *C.M.U.*

Hyland, M. (1982) The effective Topos. In: Troelstra and Van Dalen eds, *The Brouwer Symposium*. North-Holland.

—— (1988) A small complete category. Lecture delivered at the Conference *Church's Thesis after 50 years*, Zeiss (NL), June 1986. In: *Ann. Pure Appl. Logic* **40**.

Hyland, M. and Pitts, A. (1987) The theory of constructions: categorical semantics and topos theoretic models. *Categories in C.S. and Logic*. Boulder (AMS notes).

Hyland, M., Robinson, E. and Rosolini, P. (1987) The discrete objects in the effective topos. Math. Dept., Cambridge University.

Johnstone, P. (1977) *Topos Theory*. Academic Press.

Knoblock, T. B. and Constable, R. L. (1986) Formalized metareasoning in type theory. *IEEE Conference on Logic in Computer Science*. Ithaca, June 1987, pp. 237–48.

Kreisel, G. (1959) Interpretation of analysis by means of constructive functionals of finite type. In: A. Heyting, ed, *Constructivity in Mathematics*. North-Holland, pp. 101–28.

Lambeck, J. and Scott, P. J. (1980) Intuitionistic type theory and the free topos. *J. Pure Appl. Algebra* **19** 215–57.

Longo, G. (1986) On Church's formal theory of functions and functionals. Lecture delivered at the Conference *Church's Thesis after 50 years*, Zeiss (NL), June 1986. In: *Ann. Pure Appl. Logic* **40** 93–133.

—— (1988) *From type-structures to type theories*, Lecture Notes, Spring semester 1987/8, C.S. Dept., C.M.U.

Longo, G. and Martini, S. (1986) Computability in higher types, Pω and the completeness of type assignment. *Theor. Comp. Sci.* **46** 2–3 (197–218).

Longo, G. and Moggi, E. (1990) A category-theoretic characterization of functional completeness. *Theor. Comp. Sci.* **70**, 2 (*prelim. version. in Math. Found. Comp. Sci.*, Prague 1984, Chytil and Koubek, eds, *Lecture Notes in Computer Science* **176**. Springer-Verlag, pp. 397–406).

Martin-Löf, P. (1971) A theory of types. *Report 71-3*, Dept. of Mathematics, University of Stockholm, February 1971, revised October 1971.

—— (1972) An intuitionistic theory of types. *Report*, Dept. of Mathematics, University of Stockholm, 1972.

—— (1975) An intuitionistic theory of types. In: Rose Shepherdson, ed, *Logic Colloquium* **73**. North-Holland, pp. 73–118.

—— (1982) Constructive logic and computer programming. In: L. J. Cohen *et al.*, eds, *Logic, Methodology and Philosophy of Science VI*. North-Holland, pp. 153–75.

—— (1984) *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Meseguer, J. (1988) Relating models of polymorphism. *POPL '89*.

Meyer, A., Mitchell, J., Moggi, E. and Statman, R. (1987) Empty types in polymorphic lambda calculus. *POPL '87*.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. Sys. Sci.* **3** 348–75.

Mitchell, J. (1986) A type-inference approach to reduction properties and semantics of polymorphic expressions. *ACM Conference on LISP and Functional Programming*. Boston.

—— (1988) Polymorphic type inference and containment. *Info. & Comp.* **76** (2/3) 211–49.

Moggi, E. (1986) Messages on 'Types'. Electronic mailing list.

—— (1988) The partial lambda-calculus. PhD Thesis, Edinburgh.

Nederpelt, R. P. (1980) An approach to theorem proving on the basis of a typed lambda calculus. *LNCS '87*, pp. 182–94.

Nordstrom, B. (1986) Programming in constructive set theory: some examples. In: *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*. Portsmouth, England, pp. 141–53.

—— (1961) Martin-Löf type theory as programming logic. *Prog. Meth. Group, Rep.* **27** Göteborg.

Petersson, K. (1982) *A Programming System for Type Theory*. Chalmers University, Göteborg.

Pitts, A. (1987) Polymorphism is Set Theoretic, constructively. In: Pitt *et al.*, eds. *Symposium on Category Theory and Comp. Sci. Springer Lecture Notes in Computer Science* **283**. Edinburgh.

Plotkin, G. (1977) LCF as a programming language. *Theoret. Comput. Sci.* **5** 223–57.

Reynolds, J. (1984) Polymorphism is not set-theoretic. In: Kahn, MacQueen and Plotkin, eds, *Symposium on Semantics of Data Types. Lecture Notes in Computer Science* **173**. Springer-Verlag.

Reynolds, J. C. (1985) Three approaches to type structures. *Lecture Notes in Computer Science* **185** 145–6.

Reynolds, J. C. and Plotkin, G. (1988) On functors expressible in the polymorphic typed lambda calculus. Preliminary report. C.M.U.

Robinson, E. (1989) How complete is PER? *LICS '89*.

Rosolini, G. (1986) About modest sets. Notes for a talk delivered in Pisa.

Scott, D. (1970) Constructive validity. In: *Symposium on Automatic Demonstration, Lecture Notes in Mathematics* **125**, Springer-Verlag, New York, pp. 237–75.

—— (1972) Continuous lattices. *Toposes, Algebraic Geometry and Logic*. Lavwere, ed, SLNM **274**, Springer-Verlag, pp. 97–136.

—— (1976) Data types as lattices. *SIAM J. Comput.* **5** 522–87.

—— (1980a) Lambda-calculus, some models, some philosophy. In: Barwise *et al.*, eds, *The Kleene Symposium*. North-Holland.

Seely, R. A. G. (1986) Categorical semantics for higher order polymorphic lambda calculus. *JSL* **54**(4) 969–89.

Taylor, P. (1986) Recursive domains, indexed category theory and polymorphism. PhD Thesis, Imperial College, London.

Troesltra, A. S. (1973) Notes in intuitionistic second-order arithmetic. Cambridge Summer School in Math. Logic, *Springer Lecture Notes in Mathematics* **337**, pp. 171–203.