Inductive Definitions and Type Theory an Introduction (Preliminary version)

Thierry Coquand and Peter Dybjer*

Programming Methodology Group. Department of Computer Sciences. Chalmers University of Technology and University of Göteborg. S-412 96 Göteborg, Sweden
e-mail coquand,peterd@cs.chalmers.se

Abstract. We give a short introduction to Martin-Löf's Type Theory, seen as a theory of inductive definitions. The first part contains historical remarks that motivate this approach. The second part presents a computational semantics, which explains how proof trees can be represented using the notations of functional programming.

Introduction

Martin-Löf's type theory can be described as an intuitionistic theory of iterated inductive definitions developed in a framework of dependent types. It was originally intended to be a full-scale system for the formalization of constructive mathematics, but has also proved to be a powerful framework for programming. The theory integrates an expressive specification language (its type system) and a functional programming language (where all programs terminate). There now exist several proof-assistants based on type theory, and non-trivial examples from programming, computer science, logic, and mathematics have been implemented using these.

This note contains two parts. The first one is a presentation of some historical roots of Type Theory. We must emphasize that this very preliminary account cannot even pretend to be a first approximation to a history of inductive definitions in mathematics and computer science and hope that anyone who knows more about this fascinating topic will share this knowledge with us.

The second part, written by the first author, is a description of a possible computational semantics of Type Theory. We emphasize its open nature: much like in a standard functional language such as ML or Haskell the user can add new types whenever there is a need for them. We try on some concrete examples to explain how this semantics justifies programs, and how we can in this way use the quite elegant notations designed in functional programming for representing and checking inductive proofs.

* This research has been done within the ESPRIT Basic Research Action "Types for Proofs and Programs". It has been paid by NUTEK, Chalmers and the University of Göteborg.

1 Some Historical Roots

The importance of trying to put conceptual ideas in their historical context cannot be overestimated. Let us for instance cite Webb [39]: "But the more I tried to sort out and understand the arguments, to sift claim and counterclaim, the more I found that most of the central figures, however original they might have seemed, had really gotten key ideas from their teachers and predecessors, sometimes very obscure ones at that. And, most importantly, time after time I found that, *because* of my ignorance of these antecedants, *I had not, nor could have, really understood those ideas.*"

When looking for the roots of Intuitionistic Type Theory we must trace both main components: a functional component, based on λ-calculus, and an inductive component. This separation is quite similar in a programming language like ML, that incorporates both higher-order functions and concrete data types.

1.1 Prehistory

Induction on the natural numbers Reasoning by induction on the natural numbers, though it had been used unconsciously for a long time, seems first to have been made explicit in the XVIIth century, in two different forms, by Pascal and Fermat. As formulated by Pascal and used for proving properties of numbers in his "triangle", to prove that a property P holds for all natural numbers, it is enough to prove that it holds for 1, and that it holds for $n+1$ whenever it holds for n. Fermat formulated it as a principle of "infinite descent" for establishing that a property is universally false: if a property P is such that $P(x)$ implies that there exists $y < x$ such that $P(y)$, then $P(x)$ never holds.

Transfinite induction A completely new kind of induction, transfinite induction, was discovered by Cantor. He was first motivated by his analysis of trigonometry, but it also had an important rôle for general topology in the beginning. A typical instance is the definition of the Cantor-Bendixson index of a subset of the reals, where it is used for proving that a closed subset is either countable or has the cardinality of the continuum.

Transfinite induction gives rise to infinitely branching well-founded structures. There are examples, such as Paris-Harrington's version of Ramsey's theorem, which show that it is a genuinenly new principle (it is non-conservative over Peano arithmetic).

The set-theoretic notion of transfinite induction appears in type theory as generalized induction (as opposed to finitary induction). In the framework of Aczel's rule sets [1], they appear as "infinitary" rules, that are rules with an infinite number of premisses.

The book [5] contains a detailed historical introduction to the notion of inductive definition.

Frege's and Dedekind's analysis Frege and Dedekind independently discovered how inductive definitions can be explained in (impredicative) set theory. This explanation is the one currently used in mathematics. For instance the subgroup of a group G generated by a subset A is defined as the intersection of all subgroups of G containing A.

Skolem showed that a large part of mathematics can be developed without quantification over infinite sets. He essentially used what later came to be called primitive recursive arithmetic, where induction is only over non-quantified formulae.

The possibility of taking as primitive the notion of inductive definition in general is clearly expressed in the work of Lorenzen [23], but does not seem to have been formalised in detail. This possibility is suggested in Aczel's survey paper [1], and one can see type theory as an attempt of actualisation.

1.2 Intuitionism

Historically, the next important step is the work of Brouwer, who, in his intuitionistic analysis of topological notions, formulated the first examples of generalized inductive definitions. For instance, a countable ordinal is seen as an inductively generated object of three possible forms:

- the initial 1 ordinal,
- a successor ordinal $x + 1$
- a limit ordinal $x_1 + x_2 + \ldots$, where (x_n) is a sequence of ordinals.

The principle of induction over this kind of objects was also explicitely formulated and considered primitive. To prove that $P(x)$ holds for any ordinal x, it is sufficient to show that $P(1)$ holds, that $P(x + 1)$ holds if $P(x)$ holds, and that $P(x_1 + x_2 + \ldots)$ holds whenever $P(x_n)$ holds for all n.

1.3 Metamathematical Analysis

Post Canonical Systems *The canonical systems* were invented by Post [33] in an attempt to find the most general form of a formal system, such as for example Principia Mathematica. An economical presentation is given in Martin-Löf's notes on constructive mathematics [24]. As shown for instance in Lorenzen's of Martin-Löf's book [23, 24], the use of canonical Post systems allows for a short and elegant presentation of the basic notions of recursivity.

Tarski's Truth Definition Natural examples of inductive definitions appeared in Tarski's work on truth definitions. First, one needed to give a precise definition of the syntax of the language. It is interesting that Tarski presented it as directly justified (though he points out that one may define it using an impredicative definition similar to the ones used in Principia Mathematica). Second, the truth definition was a complex example of a recursively defined function over this syntax.

1.4 Proof Theory

Functional Systems Hilbert, in his study of the infinite, formulated the notion of higher-type primitive recursion (or primitive recursive functional). It was conjectured by Hilbert and shown by Ackermann that this gives a strict hierarchy of functions indexed by the complexity of the types: Ackermann's function can be defined by higher-type primitive recursion, but is not primitive recursive.

Closely connected to the consideration of these functional systems is the presentation of arithmetic given by Herbrand [16]. This presentation, which inspired the "Herbrand-Gödel" notion of general recursive functions, is quite interesting in the light of Martin-Löf's type theory. Indeed, Herbrand's presentation of arithmetic as an *open* system is similar to our presentation of type theory: he allows the addition of new function symbols and new computation rules, provided it is possible "to prove intuitionistically" that these computation rules are total. His system may be seen as a precursor of the usual presentation of Peano arithmetic as a first-order theory, but in such a presentation, the "open character" of the system has been lost.

This notion of primitive recursive functional was used in Gödel's Dialectica interpretation [11]. The goal was to give a consistency proof for arithmetic based on this notion. Spector extended this to a consistency proof of second-order arithmetic, by introducing the notion of bar-recursion.

The analysis of these systems seen as functional systems, and in particular, a meta-theoretical proof that all the functions defined in these systems are total, by Tait and Howard [36, 18], has been historically an important step in the discovery of the deep analogy between functional systems and proof systems.

Realizability Recursive realizability was introduced by Kleene [20] in order to relate precisely two quite different approaches to the notion of effective processes: the constructive functions in intuitionism, and the notion of recursive functions. Modified realizability was introduced by Kreisel in [21], as a tool to show the non derivability of Markov's principle in Heyting's arithmetic. The main difference is that extracted computations are programs of a programming language that has the normalisation property.

(Modified) realizability can be seen as a natural projection operation on formulae (that become types) and proofs. This is explained for instance in [31].

What is important here is that realizability suggests strongly an elegant approach to rigourous programming: to prove (constructively) a proposition can be seen as a way to achieve the "hand-in-hand" development of programs together with proofs that they satisfy their specification, advocated by Dijkstra or Gries [9][2]. The book [4] contains a detailed discussion of realizability and of its potential interest for computer science.

[2] Let us cite Gries [13]. "The study of program correctness proofs has led to the discovery and elucidation of methods of developing programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way."

Natural Deduction Martin-Löf's type theory is formulated in the natural deduction style introduced by Gentzen. Reduction rules for natural deduction proofs were studied by Prawitz.

A general theory of inductive definitions in predicate logic in predicate logic was formulated by Martin-Löf [25]. He also proved a normalization theorem. This theory was an immediate precursor to intuitionistic type theory the first version of which appeared shortly afterwards.

1.5 Programming Languages

Inductively defined notions are permeate theoretical computer science too. One fundamental example is the notion of a language in the sense of formal language theory. But we shall focus on two aspects of programming languages here: inductive programming language constructs and inductive constructs for reasoning about programs.

Data types, Constructors, Structural Induction The definition of data types in terms of their constructors were presented by Burstall [6], who introduced in this paper also the quite elegant case notation for the definition of functions over data types. He also discussed the general method of structural induction for provig properties of programs which manipulate such data types, and pointed out, fact which is going to be further illustrated in Type Theory, the analogy of structure between the form of the proofs and the programs to which they refer.

This method of structural induction was further analyzed in Hoare's paper on recursive data structures [17]. Earlier, McCarthy and Painter [29] had used structural induction for proving the correctness of a compiler for arithmetic expressions.

Initial Algebras In category theory inductive definitions are modelled by initial algebras. This is the basis for the algebraic approach to the semantics of data types, whereby one considers the initial algebra on a signature of constructor operations [12]. These ideas also influenced the design of programming languages such as OBJ, NPL, Hope, and Standard ML, where data types are defined by listing the constructors with their types.

Logic Programming A natural interpretation of the definite Horn clauses of a logic program is as the introductory clauses of a (simulteneous) inductive definition of predicates. Hagiya and Sakurai [14] used this view to explain the foundation of logic programming, including negation as failure, and also pointed to the connection with Martin-Löf's theory of iterated inductive definitions.

Operational and Natural Semantics Natural Semantics [19, 8], inspired from previous works by Plotkin [32] and by Martin-Löf [27], can be seen as an

illustration of the use of inductive definitions in computer science. The programs are seen as inductively defined relations between inductively defined objects, and the proofs are done by structural induction [8].

We cannot resist to cite J. Despeyroux comments on the use of this method, because they fit perfectly with the representation of natural semantics in type theory, that we shall describe in the second part. Furthermore, we feel that type theory is a way to make precise these intuitive remarks. "To prove the validity of an inference rule, we have to prove that for each proof of the premisses, we have a proof of the conclusion. For that, we shall use *induction on the length of the proofs* of the premisses... We are allowed to use induction on the length of the proof for we ...only consider syntactic models."

1.6 The Curry-Howard Correspondence

In the 60s, people started to realize the deep analogies between functional systems and proof theoretical systems. For instance, the techniques developped by Tait for proving the totality of functions defined in Gödel's system T, could be used essentially to prove the normalisation property of systems of natural deduction. A typical example of this situation was the normalisation property for system F. Though it was found having in mind a termination theorem for a functional system, it was later realized that it can be applied, as by magic, to give a proof of Takeuti's conjecture, that was a conjecture about cut-elimination in a sequent calculus of second-order arithmetic.

These analogy were used by Prawitz in his study of "validity" of proofs [?]. The main idea, according to Prawitz, is the analysis of inferences in term of introduction and elimination rules. The introduction rules are understood as conferring a meaning of the logical constants. Elimination rules are justified w.r.t. the given introduction rules. At about the same time, de Bruijn, Scott, and Martin-Löf showed that these analogy could be given a common expression in a general type theory with dependent types, unifying for instance the notion of proofs by induction and functions defined by induction.

Ideas closely related to the Curry-Howard correspondence had also appeared earlier in Lawvere's work on categorical logic from the 60s. His hyperdoctrines [22], which are categorical models of first order predicate logic, model propositions as objects and proofs as morphisms, and has a quite similar structure to type theory.

The notions of constructor and concrete data type turned out to be similar to the notions of introduction rule and logical constant respectively. Thus the close connection between functional systems studied in metamathematics and proof theory could be extended to a connection to functional programming languages. The notion of admissible rule corresponds closely to the notion of recursively defined functions. It should be noted for instance that the introduction of data types such as product or disjoint union, which seem so natural in a functional programming language, did not appear in the literature of functional systems before the 70s.

1.7 Rereading the history from a type-theoretic perspective

In the previous sections we have pointed to several traditions which have been brought together by the Curry-Howard isomorphism. It is interesting to reread this history in the light of the structure of type-theory. We note for example that there are several different levels of complexity of inductive definitions in type theory:

- First order datatypes and type constructors.
  - Simple type constructors such as disjoint union and cartesian product, and their logical analogues conjunction and disjunction.
  - Natural numbers.
  - Other simple datatypes such as lists and binary trees.
- Higher type constructors and transfinite types.
  - The function space construction and implication.
  - Brouwer ordinals.
- Dependent types.
  - Disjoint union and cartesian product of a family of types and their logical analogues the quantifiers.
  - The equality relation.
  - Other inductively defined relations, such as the transitive closure of a relation. Inductively defined families of types such as vectors (lists of a certain length).
- Universes. Metamathematical reflection.

There is also another kind of stratification coming from the distinction between ordinary primitive recursive definitions and primitive recursive functionals.

We may then structure the history by asking for some particular concept:

- Where does it come from? How has it been used informally?
- How was it explained/justified mathematically?
- When do its rules appear as part of a formal system?

In each case we have to look at both sides of the Curry-Howard isomorphism:

- Logical systems, induction principles.
- Programming languages and type systems, definition by recursion.

For example, analysing Π and Σ this way, we note that the cartesian product and disjoint union of a family of sets are well-known mathematical concepts with a clear set-theoretic definition. The formal type-theoretic rules for Π and Σ were used by Scott [35] who gives the following background [35][p 240]: "Next de Bruijn made good use of *cartesian products* of species (formation of function spaces) in connection with the *universal quantifier* - an idea also familiar to Lawvere and to a certain extent to Läuchli - and the author took this at once. Now dual to products (as Lawvere knows) are *disjoint sums* which must be used for the interpretation of the *existential quantifier* (cf. Kreisel - Goodman). These sums were not employed by de Bruijn, but it would be easy to add them to his

system.". Later Martin-Löf introduced the eliminators *split* [26] and *funsplit* [28].

On the logical side we first have Heyting's and Kolmogov's explanations of the logical constants in terms of proofs and problems respectively. For example, according to Heyting, a proof of $\forall x.P[x]$ is a function that given an arbitrary element a returns a proof of $P[a]$, and a proof of $\exists x.P[x]$ is a pair consisting of an element a and a proof of $P[a]$. These explanations and also Gentzen's idea that logical constants are defined by their introduction rules can be read as saying that the logical constants are inductively defined sets of proofs. Gentzen also formulated the system NJ of natural deduction in intuitionistic logic. Rules for reduction of proofs in this system were given by Prawitz.

2 A Computational Model of Type Theory

In this section, we give a computational semantics for Type Theory. We believe that to present such a semantics is useful for the understanding of Type Theory. While writing this semantics, we realized that our treatment is quite close to the one described in Sato's paper [34]. The main difference is in the way inductive and co-inductive proofs are handled (that would be more related to the treatement of inductive definitions in [15]). It is also quite similar in many respect to the semantics of NuPrl [2], the main difference, behind the treatment of inductive proofs, being the fact that the present theory is with an intensional equality, so that types are interpreted as predicates over programs and not as partial equivalence relations.

2.1 The general form of this interpretation

The computational interpretation we shall give has the following structure. We start with a general language for computations, which defines an equational theory $(\Lambda, =)$. On top of this language, we can define a language of types, built from a special constant Set, primitive type $\mathsf{Elem}(u)$, and dependent product and sum. This describes the syntax of the language of type theory.

In order to interpret this language, we have to give the semantics of the constant Set and of each primitive type $\mathsf{Elem}(u)$. The semantics of Set will be a subset $\mathcal{S}$ of Λ, and the semantics of $\mathsf{Elem}(u)$ will be given by an interpretation function $\phi(u)$, for $u \in \mathcal{S}$, which is a set of expressions. It is then direct to extend this semantics to a general type, which gets thus interpreted as a subset of Λ.

A large number of properties of such a semantics are independent of the subset $\mathcal{S}$ and of the semantic function ϕ.

2.2 Expressions of Martin-Löf's Logical Framework

A short description may be that it is a lazy functional language with dependent types. We insist on the word *lazy* because we will assume that the equational law

$$(\lambda x.u)(v) = [v/x]u$$

is valid on terms, which is not the case with a strict language[3]. Also, it may be possible to extend our treatment to infinite objects, and for this, the restriction to a lazy language is essential. There are two levels of expressions, *terms* and *types*. We write $E(x_1, \ldots, x_n)$ if E is an expression that may contain the free variables $x_1, \ldots, x_n$. We write $[E_1/x]E_2$ for the substitution of E_1 for x in E_2, or, if the variable x is clear from the context, simply $E_2[E_1]$.

Term expressions The terms of the language can be thought of as programs of a lazy functional language. They have following syntax[4], inspired by the elegant presentation of mini-ml in [19]

$$u \;:=\; x \mid C\,u_1 \ldots u_r \mid \lambda p.u \mid u\,u \mid () \mid (u,u) \mid \mathsf{case}(u; C_1\,u_1, \ldots, C_n\,u_n)$$

where p is a pattern, that is a special term defined by

$$p \;:=\; x \mid (p,p) \mid ().$$

We make a distinction between variables, or identifiers, $x, y, \ldots$ that may be free or bound, and terms built out of *constructors* $C\,u_1 \ldots u_p$. Each constructor is supposed to have a fixed arity. We use $t, u, v, \ldots$ for meta-variables representing terms. We write $(e_1, \ldots, e_n)$ for $(e_1, (e_2, \ldots, e_n))$.

We assume known the notion of α-conversion, and terms are thought of modulo α-conversion. We assume defined the operation of (simultaneous) substitution $[u/p]v$ which takes care of the problem of clashes of variables. For instance $[(u,z)/(x,y)]\lambda z.\lambda x.x\ y = \lambda z'.\lambda x.x\ z$.

We write Term the set of (closed) terms; we define β, η-equality on Term by taking the congruence generated by the laws

1. η-conversion: $u = \lambda p.(u\ p)$ if no identifiers of p are free in u,
2. β-conversion: $(\lambda p.u)\ v = [v/p]u$,
3. case conversion: $\mathsf{case}(C_i\ v_1 \ldots v_p; C_1\ u_1, \ldots, C_n\ u_n) = u_i\ v_1 \ldots v_p$.

---

[3] In a strict language, $(\lambda x.0)(v)$ may diverge if v diverges, while $[v/x]0 = 0$ is a convergent expression.

[4] We should also include let and letrec expressions, that are useful both for programs and for proofs; they correspond in the last case to the possibility of having local lemmas. Another needed extension is the addition of defined constants. We omit these two extensions in order to simplify the presentation.

This forms a consistent equational theory. An elegant way to establish this consistency is to use denotational semantics for building a model of this theory [5].

Another way is to define a reduction relation on terms, namely β, η-reduction, and to prove the Church-Rosser property. It follows from such a proof that we have the two fundamental consistency properties:

1. if C_1 and C_2 are different constructors, we have $C_1\ u_1 \ldots u_p \neq C_2\ v_1 \ldots v_q$ for all u_i, v_j,
2. if C is a constructor, and $C\ u_1 \ldots u_p = C\ v_1 \ldots v_p$, then $u_1 = v_1, \ldots, u_p = v_p$.

We say that an element $u \in \mathsf{Term}$ is *canonical* iff there exists a constructor C and terms $v_1 \ldots v_p \in \mathsf{Term}$ such that $u = C\ v_1 \ldots v_p$. By the first consistency property, the constructor C is then uniquely determined.

Type expressions The types of the language are formed following the rules

$$A \ := \ \mathsf{Set} \ \mid \ \mathsf{Elem}(u) \ \mid \ (x : A)A \ \mid \ (x : A, A).$$

The type $(x : A)B$ is called the dependent product and $(x : A, B)$ the dependent sum of the family of type $B(x)$ over the type A.

We write $A \rightarrow B$ for $(x : A)B$ if x does not appear in B, and $(x_1 : A_1, x_2 : A_2 \ldots, A_n)$ for $(x_1 : A_1, (x_2 : A_2, \ldots, A_n))$. Furthermore, we write $(x_1 : A_1, \ldots, x_n : A_n)A$ for $((x_1, \ldots, x_n) : (x_1 : A_1, \ldots, A_n))A$.

The equational theory on terms extend directly on types. We write Type the set of all (closed) types. We can also extend the notion of β, η-reduction, and still prove Church-Rosser. In particular, if $\mathsf{Elem}(u) = \mathsf{Elem}(v)$ then $u = v$.

We often write u instead of $\mathsf{Elem}(u)$ if there is no possible confusion. A type which does mention Set is called a *small* type.

2.3 Interpretation

The logical framework is an *open* system. We capture this mathematically by parametrising our interpretation by a function ϕ of domain $\mathcal{S} \equiv \mathsf{Dom}(\phi) \subseteq \mathsf{Term}$ and of values subsets of Term, that is $\phi(u) \subseteq \mathsf{Term}$ for $u \in \mathcal{S}$. The subset $\mathcal{S}$ represents the set of terms of type Set[6]. The subset $\phi(u)$ represents intuitively the set of terms of type $\mathsf{Elem}(u)$.

Given ϕ, we lift it to a function Φ, defined on $\mathcal{T} \subseteq \mathsf{Type}$, such that $\Phi(A) \subseteq \mathsf{Term}$ for $A \in \mathcal{T}$. Intuitively, $A \in \mathcal{T}$ means that A is a correct type at level ϕ, and in this case, $\Phi(A)$ represents the set of terms of type A.

[5] It is actually possible to get operational results from this denotational results, using adequacy theorems; see Winskel's text book [40]. The advantage of this approach is that we don't rely on Church-Rosser property. A typical example is surjective pairing in an untyped framework: it is not Church-Rosser, but its equational consistency can be establish by use of denotational models; see for instance [3].

[6] All subsets of Term are implicitely closed under β, η-equality.

1. Set is a correct type, and $\Phi(\mathsf{Set}) \equiv \mathcal{S}$,
2. if $u \in \mathcal{S}$, then $\mathsf{Elem}(u)$ is a correct type, and $\Phi(\mathsf{Elem}(u)) \equiv \phi(u)$,
3. if A is a correct type, and if $[a/x]B$ is a correct type for $a \in \Phi(A)$, then both $(x : A)B$ and $(x : A, B)$ are correct types. Furthermore, $\Phi((x : A)B)$ is the set of terms $u \in \mathsf{Term}$ such that $u(a) \in \Phi([a/x]B)$ if $a \in \Phi(A)$ and $\Phi((x : A, B))$ is the set of terms $u \in \mathsf{Term}$ such that there exists $a \in \Phi(A)$ and $b \in \Phi([a/x]B)$ such that $u = (a, b)$. [7]

The next proposition follows directly from the definition.

Stability Property: If A is a small type, and ϕ' extends ϕ, then $\Phi'(A) = \Phi(A)$.

This is a simple, but important property, closely connected to the notion of "sufficient completeness" and "fully specified types" used in algebraic specification [30]. Notice that this property needs not to be true for any type. In particular this is not the case for the type Set, since we increase $\Phi(\mathsf{Set})$ at each addition of a new data type.

2.4 Computational Evidence of a Type

If A is a correct type, and $u \in \Phi(A)$, we say that the term u is a *computational evidence* for the type A. We write this relation $u \epsilon A$. More generally, let Γ is a set of conditions on variables $x_1, \ldots, x_n$ of the form $v \epsilon B$ or $v_1 = v_2$. We write

$$u \epsilon A \quad \Gamma$$

meaning that, for any sequence $a_1, \ldots, a_n$ such that the conditions $\Gamma[a_1, \ldots, a_n]$ are all satisfied, $A[a_1, \ldots, a_n]$ is a correct type and $u[a_1, \ldots, a_n] \epsilon A[a_1, \ldots, a_n]$.

By definition, we have

1. if $u \epsilon A$ and B is a correct type such that $A = B$, then $u \epsilon B$,
2. if $u \epsilon A$ and $v = u$, then $v \epsilon A$,
3. if $v \epsilon B$ $[x \epsilon A]$, then $\lambda x.v \epsilon (x : A)B$,
4. if $v \epsilon (x : A)B$ and $u \epsilon A$, then $v\ u \epsilon B[u]$.

This is clear, except maybe for the third clause: we have to check that $(\lambda x.v)(u) \epsilon B[u]$ if $u \epsilon A$ knowing that $v[u] \epsilon B[u]$ for $u \epsilon A$. Given $u \epsilon A$, we have $(\lambda x.v)(u) = v[u]$, hence the result by the second clause.

2.5 Some examples

At any level ϕ, we can show $\lambda A.\lambda x.x \epsilon (A : \mathsf{Set})(x : A)A$. Indeed, by the third clause, it is enough to show $x \epsilon A$ if $A \epsilon \mathsf{Set}$ and $x \epsilon A$, which is a tautology.

[7] The exact definition is a little more complex, because we have to take into account the level ϕ. The idea is to look at the level ϕ as a Kripke world. With this modified definition, the interpretation $\Phi(A)$ is monotonic in ϕ, that is such that $\Phi(A) \subseteq \Phi'(A)$ if ϕ' extends ϕ.

We can add to the set $\mathcal{S}$ the constant $\bot$ and define $\phi(\bot)$ to be the empty set.

We can introduce $\mathsf{N} \in \mathsf{Set}$, with the constructors $\mathsf{0} \in \mathsf{N}$ and $\mathsf{S} \in (x : \mathsf{N})\mathsf{N}$. This means that the set $\mathcal{S}$ contains now N and $\phi(\mathsf{N}) \equiv \{\mathsf{S}^k(\mathsf{0}) \mid k = 0, 1, \ldots\}$.

We can add the disjunction on sets: we close $\mathcal{S}$ by the operation $A \vee B \equiv \mathsf{Plus}\ A\ B$. We define

$$\phi(A \vee B) \equiv \{\mathsf{Inl}\ u \mid u \in \phi(A)\} \cup \{\mathsf{Inr}\ v \mid v \in \phi(B)\}.$$

This is an example of inductive definition with parameters[10].

We can add to the set $\mathcal{S}$ the terms $\mathsf{Leq}\ u\ v$, for $u \in \mathsf{N}$ and $v \in \mathsf{N}$. We define $\phi(\mathsf{Leq}\ u\ v)$ to be the set of terms w such that $w = C_0\ v$ and $u = \mathsf{0}$, or $w = C_1\ u'\ v'\ w'$, and $u = \mathsf{S}\ u'$, $v = \mathsf{S}\ v'$ and $w' \in \phi(\mathsf{Leq}\ u'\ v')$.

2.6 Data types, components

Data types All the previous examples have the following form. We suppose given a type $T \in \mathcal{T}$, and we introduce a new family $C\ a_1 \ldots a_n \in \mathcal{S}$ for $(a_1, \ldots, a_n) \in \Phi(T)$. The constructor C will be called a *set constructor*, the intention being that $C\ a_1 \ldots a_n$ is a set whenever $(a_1, \ldots, a_n) \in \Phi(T)$.[8]

In order to define $\phi(C\ a_1 \ldots a_n)$, we suppose that we are given a list of constructors $C_1, \ldots, C_k$ together with a list of typing conditions

$$C_i\ x_1 \ldots x_{n_i} \in \phi(C\ a_1^i \ldots a_n^i)$$

for $(x_1, \ldots, x_{n_i}) \in T_i(C)$. We suppose that C occurs positively in T_i. This can be seen as defining a monotone operator on the poset of family of sets over $\Phi(T)$. We then define $\phi(C\ a_1 \ldots a_n)$ using the least fixed-point of this operator.

In the example Leq above, the type T is the type $(x : \mathsf{N}, \mathsf{N})$. The two constructors are C_0 and C_1, and the typing conditions are

$$C_0\ v \in \mathsf{Leq}\ \mathsf{0}\ v$$

for $v \in \mathsf{N}$ and

$$C_1\ u\ v\ w \in \mathsf{Leq}\ (\mathsf{S}\ u)\ (\mathsf{S}\ v)$$

for $(u, v, w) \in (u : \mathsf{N}, v : \mathsf{N}, \mathsf{Leq}\ u\ v)$.

The definition of ϕ using a least-fixed point is standard [1]. We only want to stress next one important consequence of this definition, which is used crucially in checking the correctness of the definition of a function over such a data type.

[8] In order to simplify the discussion, we do not consider the case where there are parameters in this definition, as in the definition of Plus above.

Components Given a typed term $t \in A$, we define when another typed term is a *component* of $t \in A$. There are four cases.

1. First $t \in A$ is a component of itself.
2. Next, if A is a type $(x : A_1)A_2$, any component of $t\ a \in A_2[a]$ is a component of $t \in A$ for all a of type A_1.
3. If A is $(x : A_1, A_2)$, then $t = (t_1, t_2)$ and any component of t_i is a component of t.
4. Finally, if $A = \mathsf{Elem}(C\ a_1 \ldots a_n)$, then $t = C_i\ u_1 \ldots u_p$ and any component of u_j is a component of t.

If C is a set constructor, we define "t_1 is a C-component of t_2" as meaning that t_1 is a component of t_2, and that in the proof that t_1 is a component of t_2, we have used at least one time the fourth clause with a type of the form $\mathsf{Elem}(C\ a_1 \ldots a_n)$.

The main remark is now that the C-component relation is well-founded.

Finite Type Theory A restricted version of Type Theory consists in the one where $\phi(u)$ consist only of "finite terms", that is, terms that are equal to a term built out only with constructors. This is already a powerful fragment of Type Theory. This corresponds to the following syntactical restriction in defining data types: no function types are admitted. Almost all data types and inductively defined predicates or relations are of this form.

A typical example of a data type that is not finite is

$$\mathsf{Ord} \ = \ 0 \mid \mathsf{S}\ \mathsf{Ord} \mid \mathsf{Lim}\ (\mathsf{N} \rightarrow \mathsf{Ord});$$

the elements of this type do not consist in general of only finite elements built only from constructors: for instance $\mathsf{Lim}\ (\lambda x.x) \in \mathsf{Ord}$.

2.7 Representation of proofs

We can now consider the correct small type

$$A \ = \ (x : \mathsf{N}, y : \mathsf{N}, z : \mathsf{N}, t : \mathsf{Leq}\ x\ y, u : \mathsf{Leq}\ y\ z)\mathsf{Leq}\ x\ z.$$

As a typical example of our representation of proofs as functional programs, we are going to show that the following functional program f is an evidence of the type A. The definition of f is recursive:

$f\ (x, y, z, t, u) \ =$
case t
$C_0\ y' \quad \rightarrow C_0\ z$
$C_1\ x'\ y'\ t' \quad \rightarrow$ **case** u
$C_0\ z' \quad \rightarrow$ "any term"
$C_1\ y''\ z'\ u' \rightarrow C_1\ x'\ z'\ (f\ (x', y', z', t', u'))$

To show that f is an evidence of A is to show that $f(x, y, z, t, u)$ is an evidence of $\mathsf{Leq}\ x\ z$ whenever (x, y, z, t, u) is an evidence of $(x : \mathsf{N}, y : \mathsf{N}, z : \mathsf{N}, t : \mathsf{Leq}\ x\ y, u : \mathsf{Leq}\ y\ z)$. We prove this by structural induction on t.

By definition of $\phi(\mathsf{Leq}\ x\ y)$, there are two cases. Either $t = C_0\ y'$ and $x = 0$, with $y' = y$, or $t = C_1\ x'\ y'\ t'$ and $x = \mathsf{S}\ x'$, $y = \mathsf{S}\ y'$ for some x', y' in $\phi(\mathsf{N})$.

In the first case, $C_0\ z$ is in $\phi(\mathsf{Leq}\ x\ z)$ since $x = 0$.

In the second case, we have two cases by definition of $\phi(\mathsf{Leq}\ y\ z)$. The first case, where $y = 0$ is impossible since $y = \mathsf{S}\ y'$. This is why we can put any term in this branch: because this case can never actually happen. We should have instead $u = C_1\ y''\ z'\ u'$ with $y = \mathsf{S}\ y''$, $z = \mathsf{S}\ z'$ and u' in $\phi(\mathsf{Leq}\ y''\ z')$. One can notice that $y'' = y'$, since both $\mathsf{S}\ y'$ and $\mathsf{S}\ y''$ are equal to y and constructors are one-to-one. It follows that (x', y', z', t', u') is an evidence for $(x : \mathsf{N}, y : \mathsf{N}, z : \mathsf{N}, t : \mathsf{Leq}\ x\ y, u : \mathsf{Leq}\ y\ z)$. By structural induction, $f(x', y', z', t', u')$ is in $\phi(\mathsf{Leq}\ x'\ z')$ and thus $C_1\ x'\ z'\ (f\ (x', y', z', t', u'))$ is in $\phi(\mathsf{Leq}\ x\ z)$.

This is a typical example of a proof by analysis on the structure of proofs [40]. The fact that each C-component relation is well-founded can be seen as a general way of justifying structural induction. The reader can compare this proof to a proof by rule induction, notion that is defined in Winskel's text book [40]. We think that the use of a functional notation is helpful here. First, we can visualise the overall structure of the proof. Second, we can see that the recursive call of the function f is well-founded, since t' is a Leq-component of t. We see that functional terms, with the notion of constructors, provide a good notation for proof trees in operational/natural semantics [40, 8].

2.8 Main Properties

The present computational semantics can be seen as an alternative to truth value semantics, where the notion of truth is replaced by the notion of computational evidence.

All the following properties are direct consequence of the stability property.

Since $\Phi(\bot)$ is empty, we know that it is impossible to have $u \in \bot$. This expresses the *consistency* of type theory.

By construction, existence and disjunction have constructive content. Let us look at the disjunction property. If we have $w \in A \vee B$, we known that $w = \mathsf{Inl}\ u$ with $u \in A$ or that $w = \mathsf{Inr}\ v$ with $v \in B$. We have furthermore a *method* for computing this, which is to reduce the proof term w by head-reduction. In this way, we get $w = \mathsf{Inl}\ u$ with $u \in A$ or $w = \mathsf{Inr}\ v$ with $v \in B$. Notice that the term u or v that we get in this way is not necessarily canonical.

We can embed a system such as HA^ω [38] in type theory. A *formula* A of HA^ω is translated as a *set*, and a proof of A becomes a term u such that $u \in \phi(A)$. This translation is closely connected to the modified realisability interpretation [38].

2.9 Variations

By changing the function ϕ, we obtain variations of type theory. If we allow non terminating computation, we obtain partial type theory, not consistent (any type is inhabited by any program that is not canonical), and if we allow infinite (stream) computations, we obtain an extension of type theory with infinite objects, still consistent, and with constructive existence. Such an extension is described in [7]. See also [37] for a realizability of co-inductive definitions by streams.

2.10 Extension of the programming language

The previous construction uses quite general hypothesis on the programming language. Here is a possible extension. We add an operator Exit u on terms, and a special constant Raise. We extend the conversion rules with the laws

$$(\mathsf{Exit}\ u)\ v = \mathsf{Exit}\ u \qquad \mathsf{case}(\mathsf{Exit}\ u; l) = \mathsf{Exit}\ u \qquad \mathsf{Exit\ Raise} = \mathsf{Raise}.$$

It is possible to extend the notion of head-reduction, of reduction, and to prove Church-Rosser. We change the definition of ϕ accordingly. For instance $\phi(\mathsf{N})$ is the set of terms of the form $\mathsf{S}^k(0)$ or Raise. It will then still be the case that head-reduction will terminate on any object $u \in \phi(A)$, but now u may reduce to the constant Raise. We thus get a notion of "terminating partial proof."

Conclusion

Let us try now to summarize the main points of our presentation. Type Theory can be seen as a lazy functional programming language with dependent types. Important notions of natural deduction get a concrete representation in type theory:

- introduction rules are represented as constructors,
- canonical proofs are represented by terms starting with a constructor,
- the method of producing a canonical proof from a proof is represented by head-reduction,
- doing a proof by induction over an object is seen as a recursive definition of a proof object,
- derived rules are represented as (recursively) defined constants.

The computational semantics of type theory not only ensures, but gives direct and elegant proofs of the following strong properties

- consistency: the absurd proposition $\bot$ is not provable,
- existence property: if a property P of natural numbers can be represented in type theory, and if $\exists x : N.P(x)$ has a proof in type theory, then, from this proof, we can effectively find a natural number n that satisfies the property P.

Furthermore, the notation of functional programming languages, with its notion of constructor, provides a concrete representation of proof trees used in

operational/natural semantics. The illustrates further the great analogy between the development of proofs in type theory, and the development of programs in a (functional) programming language. We consider this analogy as an important heuristic for the design of proof notations, and of interactive proof systems.

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.
2. S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Department of Computer Science, Cornell University, 1987.
3. H. P. Barendregt. *The Lambda Calculus.* North-Holland, 1984. Revised edition.
4. M. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, 1985.
5. F. S. P. W. Buchholz, W. and S. W. *Iterated Inductive Definitions and Subsystems of Analysis.* Springer, Berlin, 1981.
6. R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, (39):135–154, 1985.
7. T. Coquand. Infinite objects in type theory. In *Proceedings of the 1993 TYPES Workshop, Nijmegen, LNCS 806*, 1993.
8. J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science*, pages 193–205, 1986.
9. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
10. P. Dybjer. Inductive families. *Formal Aspects of Computing.* To appear.
11. K. Gödel. *Collected Works, Volumes I and II.* Oxford University Press, 1986.
12. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Prentice Hall, 1978.
13. D. Gries. *The Science of Programming.* Springer Texts and Monographs in Computer Science, 1981.
14. M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.
15. L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 53:335–343, 1991.
16. J. Herbrand. On the consistency of arithmetic. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 618–628. Harvard University Press.
17. C. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4:105 – 124, 1975.
18. W. Howard. Functional interpretation of bar induction by bar recursion. *Compositio Mathematica*, 20:107 – 124.
19. G. Kahn. Natural semantics. Technical Report 601, 1987.
20. S. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.
21. G. Kreisel. Interpretation of analysis by mean of constructive functionals of finite type. In Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.
22. F. W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In A. Heller, editor, *Applications of Categorical Algebra, Proceedings of Symposia in Pure Mathematics.* AMS, 1970.
23. K. Lorenzen. *Métamathématique.* Edition Gauthier-Villars, 1962.

24. P. Martin-Löf. *Notes on Constructive Mathematics.* Almqvist & Wiksell, Stockholm, 1968.
25. P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, 1971.
26. P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.
27. P. Martin-Löf. The domain interpretation of type theory, lecture notes. In K. Karlsson and K. Petersson, editors, *Workshop on Semantics of Programming Languages, Abstracts and Notes*, Chalmers University of Technology and University of Göteborg, August 1983. Programming Methodology Group.
28. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
29. J. McCarthy and J. A. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, pages 33–41. AMS, 1967.
30. D. R. Musser. On proving inductive properties of abstract data types. *POPL*, pages 154–162, 1980.
31. C. Paulin-Mohring. Extracting fω programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symposium on Principles of Programing Languages.* ACM, 1989.
32. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN -19, Computer Science Depratment, Aarhus University, Aarhus, Denmark, 1981.
33. E. Post. Absolutely unsolvable problems and relatively undecidable propositions. account of an anticipation. In M. Davis, editor, *The undecidable.* Raven Press, Hewlett, NY, 1965.
34. M. Sato. Adding proof objects and inductive definition mechanisms to frege structure. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Proceeding of International Conference on Theoretical Aspects of Computer Software*, pages 53–87. Springer-Verlag, LNCS 526, 1991.
35. D. S. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, pages 237–275. Springer Lecture Notes in Mathematics 125, 1970.
36. W. W. Tait. *Normal Derivability in Classical Logic.* Springer, 1968.
37. M. Tatsuta. Realisability interpretation of coinductive definitions and program synthesis with streams. *Proceedings of International Conference on Fifth Generation Computer Systems*, pages 666 – 673, 1992.
38. A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics, an introduction.* North-Holland, 1988.
39. J. Webb. *Mechanism, Mentalism and Metamathematics.* D. Reidel Publishing Company, 1980.
40. G. Winskel. *The Formal Semantics of Programming Languages, an Introduction.* MIT Press, 1993.