COMPUTATIONAL LOGIC

Maria H. Napierala
M.S. in Computer Science, Technical University of Wroclaw, Poland, 1983

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 1992

The dissertation "Computational Logic" by Maria H. Napierala has been examined and approved by the following Examination Committee:

Richard B. Kieburtz
Professor, Dissertation Advisor
Oregon Graduate Institute

James Hook
Assistant Professor
Oregon Graduate Institute

David B. MacQueen
AT&T Bell Laboratories

David G. Novick
Assistant Professor
Oregon Graduate Institute

ABSTRACT

COMPUTATIONAL LOGIC

Maria H. Napierala, Ph.D.

Oregon Graduate Institute 1992

Supervising Professor: Richard B. Kieburtz

This thesis resulted from the research on relating classical and constructive proofs. It is well known that constructive type theories constitute formal systems for constructive mathematics. In these theories the constructivity is implicit in the restriction to intuitionistic logic. This means that restrictions are placed both on the objects studied and on the methods of proofs which may be applied. Hence, not all logical laws (e.g., law of excluded middle, proof by contradiction) can be used in the proofs of consistency of logical specifications. Thus, constructive type theories are reasoning systems only for purely functional programs. Even though functional programming is mathematically elegant, it lacks expressiveness gained by using classically founded non-local reasoning (e.g., escapes and coroutines). Yet, the constructive type theories possess a desirable property that a mathematical sentence can be interpreted as a program specification, and a constructive proof of such a sentence as a program which meets the specification. It is also well-known that many classical theories do not have this property. Kreisel and Friedman showed that for certain classes of sentences (Π^0_2), the classical theories conservatively extend their constructive counterparts.

In this thesis, we show how to construct well-typed programs with nonfunctional operations. We discuss the connections we have discovered between the second-order encodings (operational interpretation) of logical connectives and the continuation-passing-style (CPS) translation on the data-valued expressions. It is well-known that the operational interpretation of the logical connectives yields the proof methods provided by those connectives. Correspondingly, CPS translation of data yields type-correct programming schemas. To obtain the actual instead of encoded operational interpretation, and consequently the actual

classical programs, we remove the impredicativity of second-order definitions of logical connectives by considering only the top-level contexts of programs. By top-level contexts, we mean those that expect values of basic types or data-structured types, excluding contexts of function-typed values. We present the *top-level, operational interpretation* of logical connectives as a way of extracting the computational content from classical proofs. We introduce a classical program development system, Computational Type Theory (CTT), formalizing this interpretation. CTT is in fact a classical type theory. We demonstrate that the top-level operational interpretation of absurdity corresponds to the algorithmic content of the only constructively problematic rule of classical logic, the rule of double-negation elimination. This algorithmic content is the nonlocal operation *resultis* which abandons the normal evaluation of a program and resumes computation in the existing, top-level context. We demonstrate that CTT formalizes the computational extract of classical Π^0_2 sentences. We extend this result to arithmetical Π^0_2 sentences, and show how to extend it to class Π^0_2 of any finite classical theory.

Our work uncovers the connections and differences between classical and constructive logic much more precisely than those derived from double-negation embeddings of classical into intuitionistic logic. It provides a first-order semantic account of classical reasoning. It is a step towards intergrating nonlocal operators into a type-theoretic explanation of computation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Computer Science has become a field which posed some very important theoretical problems concerning the relation of intuitionistic and classical logic. Automated program development is benefiting greatly from studies in this area. It is through exploring connections between logic, mathematics, and computer programming that advances can be made. "Computational Logic" has grown from the efforts of relating constructive and classical proofs. This thesis resulted from work on relating intuitionistic and classical formal systems for constructive mathematics.

The creation and study of formal systems for constructive mathematics has become one of the most active areas in logic, the philosophy of mathematics, and computer science. In "constructive mathematics" one proves the existence of an entity having certain desired properties by showing how to find it. A formal system is a calculus for manipulating the symbols of an artificial language. The construction of a formal system and the demonstration that it is consistent is a general paradigm used in mathematical logic. The goal of formal systems has been to reduce reasoning to calculation.

The growing interest in constructive formal systems from the point of view of the foundations of mathematics stems from a disagreement about the relationship of logic to mathematics. The discussion has centered on the question of whether logic and mathematics can be developed solely as a system of symbols, or whether they necessarily involve an interpretation of the symbols. As a result, three main schools of the foundations of mathematics have arisen: (1) Formalism, in which mathematics is a symbolic construction that becomes meaningful only when interpreted in ordinary language; (2) Logistics, in which mathematics describes an ideal reality that is independent of knowledge; (3) Intuitionism, where mathematics is a mental activity.

There is an increasing realization that none of these positions may be fully satisfactory. Formalism is unsatisfactory because many concepts of pure mathematics do not have any physical interpretation (e.g., the continuum). Intuitionism and logistics both invoke mathematical intuition ("intuitive" mathematical proof and "intended models" of mathematical structures, respectively) as the ultimate guarantees of consistency. However, they fail to provide a connection between either kind of mathematical intuition and the truth of mathematical statements.

The introduction of computers has brought about a fundamental change in the field of constructive mathematics, by making possible the creation of *working formal systems* in which one can actually write mathematical proofs. As the computer performs more and more complicated tasks, the necessity for it to "explain its reasons" becomes crucial. This requires the development of systems that combine programming languages and logical languages. As a consequence, algorithms and proofs become closely related, and the logic built into such systems should be constructive.

In fact, computer scientists have become interested in logic for its application to mechanical theorem-proving in order to automate the design of computer programs and to verify their consistency with logical specifications. Recently, formal systems for constructive mathematics have been seen in computer science as a basis for a systematic methodology to develop provably correct programs. *Constructive type theories* are accepted as sound logical foundations for developing correct functional programs. *Types* have become an important aspect of programming language design. They constitute a way of incorporating a logic of program specifications into the programming language itself. Thus, types provide a context for the logical development of programs and a framework for their verification. These features are crucial in software reliability and maintainability. Even though functional programming is mathematically elegant, it lacks expressiveness gained by using classically founded non-local reasoning (e.g., escapes and coroutines). In fact, many *natural* programming methodologies do not fit well into the functional programming framework. Yet, the constructive type theories possess a desirable property that a mathematical sentence can be interpreted as a program specification, and a constructive proof of such a sentence as a program which meets the specification. It is also well-known that many classical theories do not have this property.

Kreisel and Friedman showed that for certain classes of sentences (Π_2^0), the classical theories conservatively extend their constructive counterparts.

This thesis proposes a solution to the problem of relating classical and intuitionistic logic in program development. It demonstrates that certain *nonfunctional* programming language features correspond to direct reasoning in classical formal systems. Our work shows how to construct well-typed programs with these nonfunctional operations. It presents a system that formalizes the computational content of classical predicate logic and shows how to extend such a formalization to any finite first-order classical theory. The system distinguishes termination problem and lemmata on data types which are not relevant to computation from the parts of constructive proofs that are relevant to computation. By separating termination from program correctness, the system allows the introduction of general recursion operator without destroying the property that all well-typed programs terminate.

At its core, this thesis deals with the problem of a valid way of expressing and of reasoning about computation. The need for expressive reasoning systems and expressive programming systems is widely recognized. The fact that all computation is expressible using Turing machines and that number theory is sufficient for almost all reasoning about computation is here of little help. As a result of our research, we found that programs extracted from a specific, well-defined class of proofs in classical finite theories are provably correct. We built a system for constructing such programs.

To develop a calculus for defining totally correct programs (*computational logic*), it is not sufficient to formally construct only formulas or deductions. One has to construct derivations or proofs. In a logistic system (Russell, Frege, Whitehead), one can define formulas that are true but not provable (Gödel incompleteness theorem). Such a system is concerned with classical validity, i.e., truth in all models. On the other hand, in a system that constructs deductions (Gentzen), one can define formulas that are provable but whose proofs may not provide *evidence* in a constructive sense. More precisely, a constructive proof always provides a way to *construct* an object which is proven to exist. This property of constructive proofs is referred to informally as the *evidence* property. Such evidence is possible only in a system that constructs derivations (proofs). According to the Gödel incompleteness theorem, all methods of proofs (recognized as correct) can never be formalized in a single formal

system. But the operational content of deduction (i.e., of truth "modulo hypotheses") can be interpreted in the second-order λ-calculus or Girard's system **F** [Girard70]. Such an interpretation *defines* the operations that one performs to convince oneself of the truth of proposition *A* "modulo" hypotheses *H*. These operations have been shown to terminate [Girard70]. They are the building blocks for constructing proofs.

Terms justifying propositional quantifications (exhibiting the reasons for which those formulas are true) program deductions. For example, if ϕ is an arbitrary formula, then $x{:}\phi \vdash x{:}\phi$ is a method of justifying the truth of $\phi \supset \phi$ which is expressed by the following propositional quantification:

$$\forall X{:}prop.X \Rightarrow X$$

A Greek letter ϕ was used to represent formulas in order to distinguish the deduction $x{:}\phi \vdash x{:}\phi$ from its interpretation. The Λ-term $\Lambda X{:}prop.\lambda x{:}X.x$ of type $\forall X{:}prop.X \Rightarrow X$ is an identity algorithm. It represents one of the methods of justifying the truth modulo hypotheses, namely when a derived formula *is* the only hypothesis.

Since *all* data are functions in system **F**, the distinction between computation rules isn't so clear as it is in a language that has constants of basic types. System **F** identifies propositions, data types, and control structures. This means that algorithms are coded in terms of typed data objects. The ambiguity of meaning of the second-order universal quantification is known to be algorithmically consistent [Girard70]. Its negative side is that the algorithms are evaluated "by-values" only, that is, functions decompose their arguments completely according to the primitive structural induction. This is not economical and very bad for a programmer who wants to have an actual not a coded algorithm.

We will introduce classical type theories with a clear distinction between different computation rules. In these theories, data types will be distinguished from control structures in order to allow the introduction of actual programs. However, before introducing specific basic types (like integers, lists, trees, etc.), or more precisely, the theories associated with them, logic has to be formalized. A system called *Computational Type Theory* will be introduced to interpret predicate logic and propositional connectives. To internalize the logical connectives, it is not necessary to preserve the purely functional framework by extending the polymorphic λ-calculus, as it is done in the calculus of constructions. In fact,

the correctness of proofs of existential quantification, conjunction, and disjunction *should not* be verified through type-checking in a functional calculus since this implies that the logic is restricted to be intuitionistic. Instead, proofs can be defined as terms in a *predicative* type theory that internalizes the *operational* interpretation of logic at the top-level of proofs. This interpretation is in agreement with the semantics of constructive evidence [Constable85] but does not restrict the logic to be intuitionistic.

The research contribution of this thesis is twofold. It gives a sound explanation of classically-founded computation. Namely, it introduces a *type theory* suitable for programming with non-local control. Moreover, our work answers the objections to the translation techniques used to replace nonlocal control with the functional versions of them. Such translations (double-negation translations on proofs and continuation-passing-style translations on programs) often yield hard to understand and prove correct programs. We show how to construct *directly* programs with nonlocal control and show that the reasoning system for such programs is classical.

This chapter serves as a general introduction to the problem we wish to address. That is, before we begin with the technical material, the background and the context in which this thesis can be understood need to be presented. In the next four subsections we introduce the taxonomy of logics and type theories, we introduce the concepts of continuation and continuation-passing-style translation, and, finally, we survey the results relating intuitionistic and classical logic.

1.1 Constructive Type Theories

There are two kinds of constructive type theories: *predicative* and *impredicative* constructive type theories. The former differs from the latter in significant ways. Yet, understanding the distinction between predicativity and impredicativity in the study of constructive mathematics has not been settled. The notion of predicative type theory comes form Russell who introduced it to avoid paradoxes. His vicious-circle principle: "No totality can contain members defined in terms of themselves" is a rejection of *impredicative* definitions, i.e., definitions which in defining an object refer to some totality to which the object being defined belongs. This self-reference was eliminated in Russell's type theory by arranging sets in a

hierarchy of levels or types, and forbidding propositional expressions of order "i" to contain variables ranging over a type of level greater than "i."

In general, a *predicative* type theory is one in which the objects of discourse must be defined before they can be used. In contrast, when proving the propositional universal quantification $\forall X{:}prop.X \Rightarrow X$ in the system **F**, we are extending the definition of *prop* as we are in the process of quantifying over it. This self-reference is the essence of the impredicative system **F**.

An instance of predicative type theory is Martin-Löf's intuitionistic theory of types (ITT) [Martin-Löf82]. In his theory, Martin-Löf attempted to reconstruct the foundations of mathematics. His intuitionistic mathematics has become the theoretical basis of the system Nuprl [Constable86] in which one develops provably correct functional programs. Predicative as well as impredicative type theories can interpret first-, second-, and higher-order logics. For example, Martin-Löf type theories without universes interpret first-order predicate logic. When extended with a hierarchy of universes, they interpret second- and higher-order logics [Martin-Löf84]. A logically-founded *impredicative* higher-order system was developed by Coquand as a *calculus of constructions* (CC) [Coquand86, CoqHu86]. CC itself interprets not only first-order logic but higher-order logic as well. In CC, however, there is a distinction between propositions-as-types (terms of type *Prop*) and other types (terms of type *Type*). To avoid paradoxes it is possible to quantify *only* propositions over all types but *not* other types. The generalized calculus of constructions (GCC) [Coquand86] is an extension of CC with a hierarchy of universes like in Martin-Löf's type system. What follows is a discussion of each kind of theory, starting with ITT.

Martin-Löf's type theory, which is based on the intuitionistic philosophy of mathematics, admits intuition as decisive in recognizing that a mathematical proof is convincing. Intuitive evidence ensures the correctness of mathematical reasoning. An *evident* proof of a proposition in an intuitionistic logic implies that the proof is noncircular and finite. Since the law of the excluded middle is not intuitionistically valid, ITT, as a programming logic, is suitable only for purely functional programs. The importance of Martin-Löf's intuitionistic theory of types lies in the connection which it makes between computer programming and constructive mathematics, namely that algorithms should naturally accompany existence proofs. However, by identifying programs with proofs, this theory forces unwanted constructions into

synthesised programs. The solution adopted for this problem in ITT causes lack of *unicity* of types even in the first universe of ITT's hierarchy of types[1] and puts the underlying principle of identifying programs with proof-objects in question. More specifically, an object $a \in A$ that satisfies property B also becomes a member of a subset type $a \in \{x \in A \mid B(x)\}$, which was introduced to remove computationally irrelevant details from programs [NoPe83].

The calculus of constructions (CC) is an extension of the polymorphic λ-calculus (Girard's system **F**) with a part that interprets first-order predicate logic, and a part that interprets higher-order propositional logic to allow the binding of propositional schemas. Here, the consistency of proof is assured by encoding the intuitionistic semantics of propositional connectives and existential quantification in second-order propositional calculus. The universal quantifier is constructive. For example, Russell's encoding of the conjunction of propositions P and Q ($P \wedge Q$) is interpreted by the following quantification:

$$P \;\&\; Q \equiv \forall A.(P \Rightarrow Q \Rightarrow A) \Rightarrow A$$

Such an encoding formalizes the normalization of intuitive proof of conjunction. More precisely, to prove the above universal quantification, one has to possess a cut-free proof of P and a cut-free proof of Q, i.e., one has to have a cut-free proof of $P \wedge Q$. Since a cut-free proof corresponds to a deduction in a normal form [Prawitz70], the quantification $P \;\&\; Q$ encodes the condition for (natural deduction) proofs of conjunction to be reducible to normal forms. Normal proof of a conjunctive or disjunctive formula is coded as a *proof method* provided by the formula. Such a proof method is expressed by making the intuitive evidence *uniform* or universal. For instance, the proposition $P \wedge Q$ is a method of proving any proposition C provided one has a proof that C follows from *both* P and Q. This is accomplished by the use of second-order universal quantification (*universal* type), which operates without any information about the domain of quantification. Its objects — *universal abstractions* — are functions defined for arbitrary types.

---

1. Unicity of types is also a problem with respect to equality of types at higher universes.

One of the fundamental properties of CC is that all programs with legal types terminate and are correct with respect to the specifications, i.e., they are *totally* correct. The same is true of Martin-Löf's type theory in which $a \in A$ means that the program a terminates with a value in A. In fact, to develop programs in the framework of any constructive type theory requires the verification of termination of programs. Checking termination is done at the same time as checking correctness. This check may either be an implicit restriction on the expressive power of terms as in CC, or an explicit termination argument as in extensions to ITT exploited in, for example, [Nordström87]. Since a type theory is a finite formal system, not all computable functions can be developed in a single type theory in which all programs terminate.

Martin-Löf's type theories [Martin-Löf84] and Coquand's calculus of constructions [Coquand85] are examples of type theories with *decidable judgements*. One can add to the taxonomy of type theories the distinction between type theories with *decidable judgements* and type theories with *undecidable judgements*. Type theories with decidable judgements restrict expressive power of terms to primitive recursive functions. In such theories correctness of programs is verified through type checking and the general recursion operator is not available. In theories with undecidable judgements, it is no longer decidable whether an object a is in type A, i.e., whether $a{:}A$. Hence, it is no longer possible to use automatic type checking in the ordinary sense, where the compiler checks the type correctness of the program. An example of such a theory is an extension of Martin-Löf's type theory with *subset* type former $\{x{:}A \mid P(x)\}$ [NoSm84], which was already mentioned above. Hence, a belongs to two different types, that is, $a{:}A$ and $a{:}\{x{:}A \mid P(x)\}$. In Martin-Löf type theory with undecidable judgements, it is not even possible to check the program during execution since the predicate P in $\{x{:}A \mid P(x)\}$ does not have to be decidable. A restriction to decidable predicates would seriously weaken the expressive power of the specifications. Instead, the programmer must provide justification for the type correctness of the program. Such justification can be formalized and even the formal proof can be constructed, as was shown in [Petersson82]. In type theories with undecidable judgements is it possible to separate the termination proof of the program from the proof of other properties. By separating types (program specification) and propositions (e.g., $P(x)$ in the subset type $\{x{:}A \mid P(x)\}$), a general recursion operator can be introduced without destroying the property that all well-typed programs terminate [Nordström87].

Another feature of both the ITT and CC theories is that they are based on the identification of propositions, types, and specifications. Martin-Löf's type theory is basically a theory about sets in which it is possible to interpret a logic. A proposition is identified with the set of its proof objects and a specification is identified with the set of all programs satisfying the specification. Similarly, in the calculus of constructions a proposition is identified with the type of its proofs, and proofs are identified with elements of types (programs).

However, there are problems with looking at programs-as-proofs in constructive type theories. For programs-as-proofs to be executed on a computer, they have to be completely formalized including the proof of consistency of program specifications. In actual fact, such a degree of formalization is too deep. Informal reasoning about programs will always be important. Systems that identify programs with proofs in a type theoretic framework force unwanted constructions into synthesised programs. More specifically, if a type $(\Sigma x \in A)B(x)$ is used to interpret an existential proposition, it must be shown that there exists a program of type A that satisfies the property B. To do this in a type theory, one must construct a program $a \in A$ as well as a proof-object $b \in B(a)$, showing that a satisfies the specification B. When the existential statement is read as a specification of a program, only the program a and not the particular proof-object b, is of interest.

Therefore, to make a distinction between proofs and programs, the computationally irrelevant details (formal "comments") must be cut away from the formalized program specifications. No constructive content should be assigned to the part of program development that deals with the consistency of logical specifications: the goal is to be able to write an integer program using the knowledge that its argument is positive without demanding an extra argument at run-time to justify this fact. All the power of classical logic should be available in the consistency proofs. To automatically eliminate unwanted proofs from programs, an understanding of the contexts in which classical logic or intuitionistic logic are appropriate in a program development system is necessary.

The other problem with the programs-as-proofs principle is related to the difference between the repetitive constructs in programs and proofs. In programming languages such constructs do not always terminate (e.g., **while**-loops, **repeat**-loops). The repetitive constructs in proofs, on the other hand, are usually based on some principle of induction which by its very nature

always terminates. A typical example is mathematical induction. Thus, programmers often have to provide a separate proof of termination for their programs, while mathematicians usually justify the termination of proofs by referring to the correctness of the induction principle they use.

In constructive type theories with decidable judgements general recursion is not available but only primitive recursion. From a theoretical point of view, this is not a problem. The primitive recursion operator and higher-order functions together provide a way of expressing all functions provably total in Peano Arithmetic. However, this forces a programmer to prove termination of a program at the same time as the program is derived or restricts the programmer to a small set of recursion schemes. The termination proof of a program should be separated from the rest of the correctness proof. This can be accomplished by introducing a rule of well-founded recursion, and hence not giving up the idea that all well-typed programs terminate.

1.2 Natural Deduction and Sequent Calculus

Gentzen systems of *natural deduction* formalize the deductive role (or essential logical content) of different logical constants. Gentzen discovered that these atomic deductive steps are of two kinds, namely introductions and eliminations, standing in a certain symmetrical relation to each other. Gentzen analysis may be understood as an attempt to characterize the notion of proof.

In systems of natural deduction, proofs are represented as *derivations* written in a tree form. The top formulas of the tree are the assumptions. Any other formula of the tree is to follow from the one immediately above it by one of the *inference rules*. A formula written in square brackets above a premiss is to indicate that it is discharged at the inference. An inference rule is labeled with the logical constant it deals with, followed by "I" when it is an introduction and "E" when it is an elimination. In an inference by an application of an "E"-rule, the premiss in which the constant in question is exhibited is called the *major premiss* of the inference and the other premiss(es) if any, is(are) called the *minor premiss(es)*.

$$(\&I)\quad \frac{A \quad B}{A \,\&\, B} \qquad\qquad (\&E)\quad \frac{A \,\&\, B}{A} \qquad \frac{A \,\&\, B}{B}$$

$$(\vee\mathrm{I})\quad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad\qquad (\vee\mathrm{E})\quad \frac{A \vee B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C}$$

$$(\supset\mathrm{I})\quad \frac{\begin{matrix}[A]\\ B\end{matrix}}{A \supset B} \qquad\qquad (\supset\mathrm{E})\quad \frac{A \quad A \supset B}{B}$$

$$(\forall\mathrm{I})\quad \frac{A(a)}{\forall x A(x)} \qquad\qquad (\forall\mathrm{E})\quad \frac{\forall x A(x)}{A(t)}$$

$$(\exists\mathrm{I})\quad \frac{A(t)}{\exists x A(x)} \qquad\qquad (\exists\mathrm{E})\quad \frac{\exists x A(x) \quad \begin{matrix}[A(a)]\\ B\end{matrix}}{B}$$

We assume that the first-order languages contain a constant Λ for absurdity (or falsehood) and that $\neg A$ is a shorthand for $A \supset \Lambda$. The introduction and elimination rules for negation are then as follows:

$$(\neg\mathrm{I})\quad \frac{\begin{matrix}[A]\\ \Lambda\end{matrix}}{\neg A} \qquad\qquad (\neg\mathrm{E})\quad \frac{A \quad \neg A}{\Lambda}$$

The rules given above determine the system of natural deduction for first-order *minimal logic*, abbreviated **M**. By adding the rule Λ_I (intuitionistic absurdity rule)

$$\frac{\Lambda}{A} \qquad (\Lambda_{\mathrm{I}})$$

where A is an atomic proposition different from Λ, we get the system of natural deduction for first-order *intuitionistic logic* (**I**).

The system of natural deduction for first-order *classical logic* (**C**) is obtained by adding to the rules of **M** the rule Λ_C (classical absurdity rule)

$$\frac{\begin{matrix}[\neg A]\\ \Lambda\end{matrix}}{A} \qquad (\Lambda_{\mathrm{C}})$$

where A is atomic and different from Λ.

The sense of the symmetry between introduction and elimination rules is that the conclusion obtained by an elimination does not state anything more than what must have already been obtained if the major premiss of the elimination was inferred by an introduction. In other words, a proof of the conclusion of an elimination is already "contained" in the proofs of the premisses when the major premiss is inferred by introduction. This correspondence between introduction and elimination rules is referred to as the *inversion principle*. Since nothing new is gained by an elimination immediately following an introduction, it suggests that such sequences of inference can be eliminated. A formula occurrence in a derivation that stands at the same time as the conclusion of an introduction and as the major premiss of an elimination is called a *maximum formula*. The inversion principle implies that a maximum formula is an unnecessary detour in a derivation and can be removed. A derivation is defined as *normal* or said to be in *normal form* when it contains no maximum formula. The **normalization theorem** states that every derivation in **M**, **I**, or **C** reduces to normal form. The **strong normalization theorem** states that every derivation Π in **M**, **I**, or **C** reduces to a unique normal derivation Π' and every reduction sequence starting from Π terminates (in Π').

There is a close correspondence between the constructive meaning of a logical connective and its introduction rule. For instance, implication $A \supset B$ is constructively understood as the assertion of the existence of a construction of B from A. Natural deduction systems allow the inference of $A \supset B$ given a proof of B from A. However, a proof of B from A is not the same as a construction of B from A. It is rather a special kind of it, namely a *uniform* construction transforming constructions of A to constructions of B. Thus, there isn't a complete agreement but a close correspondence between the constructive meaning of logical constants and the introduction rules.

As remarked above, Gentzen's operational interpretation of implication, which is based on an introduction rule, is much stronger than the usual constructive interpretation. The same holds for universal quantification. As a consequence the interpretation of the axioms of first order arithmetic fails under Gentzen's interpretation. More precisely, the mathematical induction is not valid under such an interpretation. Since the introductions and eliminations are inverses of each other, Gentzen's idea to justify the eliminations by the meaning given to the logical constants by the introductions may be reversed [Prawitz70]. Instead of interpreting the

constants as asserting the existence of certain constructions that build up formulas with these constants, one may interpret them as stating the performability of certain operations. That is, the eliminations become the definitions of logical constants and the introductions are justified according to these definitions. This is what we may call the *extensional definition* of logical constants. Such a definition yields the usual constructive meanings of implication and universal quantification. We note that the interpretation of logical connectives in second-order logic is in agreement with this extensional interpretation. The second-order propositions "program" the constructive meanings of logical connectives.

1.2.1 Sequent Calculus

Gentzen's *sequent calculus* is an alternative formulation of natural deduction. Instead of deductions $\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}$ one considers *sequents* $\Gamma \vdash A$. More generally, sequents have the form $\Gamma \vdash \Delta$, where $\Gamma\ (=A_1, \ldots, A_n)$ and $\Delta\ (=B_1, \ldots, B_m)$ are finite sequences of formulas. The intended meaning of $\Gamma \vdash \Delta$ is that

$$A_1 \textit{ and } \cdots \textit{ and } A_n \textit{ imply } B_1 \textit{ or } \cdots \textit{ or } B_m.$$

While the main purpose of systems of natural deduction was to *define* the notion of proof, sequent calculus is a system of *derived* rules *about* proofs. Thus, sequent calculus is intended for studying the properties of proofs. Every proof in sequent calculus induces a unique natural deduction, and conversely, every natural deduction comes from sequent calculus proof, but the latter is not unique.

Sequent calculus is divided into three groups of rules: identity, structural, and logical. The three standard structural rules (*exchange*, *weakening*, *contraction*) are all of the form

$$\frac{\Gamma \vdash \Delta}{\Gamma_1 \vdash \Delta_1}$$

We will not introduce the rules of sequent calculus in the thesis. They can be found in many textbooks on mathematical logic and proof theory, e.g., [Szabo69, Kleene52, GirLa89]. We will only describe their meaning.

The exchange rule allows permutation of formulas on either side of the symbol " $\vdash$". The weakening rules allows replacement of a sequent by a weaker one. The contraction rule expresses the idempotence of conjunction and disjunction.

The *logical rules* introduce the logical connectives: conjunction, disjunction, negation, implication, universal and existential quantifications. However, instead of elimination rules, logical rules include the rules for operating on the formulas to the left of the deduction sign $\vdash$. In fact, the rule of sequent calculus are more or less complex combinations of rules of natural deductions. More precisely, the logical rules on the "right" correspond to *introductions*, and those on the "left" to *eliminations*.

The *identity* group has two rules: the *identity axiom* $C \vdash C$ and the *cut rule*

$$\frac{\Gamma \vdash A,B \quad \Delta,A \vdash C}{\Gamma,\Delta \vdash B,C} \qquad \textbf{(cut)}$$

The identity axiom is necessary to start off proofs. The *cut rule* is another way of expressing the identity. It is a dual or symmetric aspect of identity which can be eliminated from a proof. This *cut elimination* process has the same content as the normalization theorem for natural deduction systems; they only differ in the syntactic presentation. That is, a cut-free proof gives a normal deduction.

The *cut-elimination theorem* (Hauptsatz) of Gentzen [Szabo69]

Theorem (Gentzen, 1934). The cut rule is redundant in sequent calculus.

for sequent calculus corresponds to the normalization theorem for natural deduction. The former is technically more complicated than the latter because of lesser purity of syntax. The aim is to eliminate cuts of the special form

$$\frac{A \vdash C,B \quad A_1,C \vdash B_1}{A,A_1 \vdash B,B_1}$$

where the left premiss is the right logical rule and the right premiss is the left logical rule, so that both introduce the main formula C.

1.3 Continuations, CPS Translation, and Escapes

Continuation semantics has become a standard method of specifying program semantics. It is based on the denotational approach to program semantics [MS76] pioneered by Dana Scott. A *continuation* in denotational semantics is a function that can be applied to a value and/or store to yield a result of the entire computation. Program fragments are seen as continuation transformers, which take continuations as arguments and return other continuations. Continuation semantics is well suited for nonfunctional languages, since a nonlocal exit from a program context, e.g., "**goto** *l*", can be described as using a continuation stored under the label *l* instead of using the sequential continuation.

The use of continuations in compilation was extended further by Fischer [Fis72], who showed a translation from a lambda-calculus back into itself such that the resulting program contains as explicit representation of continuations. Such a translation is called a *continuation passing style* transformation, or simply CPS transformation. CPS transforms mimic the operational semantics of evaluation of the original term. A CPS transform $\overline{e}$ of a lambda-expression e is defined as follows [Griffin90]:

$$\overline{x} = \lambda k.kx$$

$$\overline{\lambda x.M} = \lambda k.k(\lambda x.\overline{M})$$

$$\overline{MN} = \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk))$$

Felleisen and others [FFED86] used this translation to convert programs with nonlocal control operations into purely functional programs. To allow the programmer to access the current continuation, the nonfunctional control operator $\boldsymbol{C}$ (pronounced *control*) was invented. The evaluation of $\boldsymbol{C}(M)$ abandons the current continuation (*control context*) and applies M to a procedural abstraction of this context. This allows M, when it finishes its evaluation, to either resume execution at the original evaluation context, or resume at another evaluation context, if it is available. In order to express the CPS translation of $\boldsymbol{C}(M)$, additional control operator $\boldsymbol{A}$ (*abort*) is introduced. The evaluation of $\boldsymbol{A}(M)$ throws away the current continuation and continues with the evaluation of M at the top-level (i.e., empty) context. The CPS transform for $\boldsymbol{C}$ is defined as follows:

$$\overline{\boldsymbol{C}(M)} = \lambda k.\overline{M}(\lambda m.m(\lambda z.\lambda d.kz)\lambda x.\boldsymbol{A}(x))$$

The $\boldsymbol{C}$ operator is a relative of Scheme's call/cc[2] [RC86] that provides access to the current

control context. This operator provides Scheme with labels and jumps to express nonlocal exits, allowing programs that are more efficient than functional programs when executed on a von Neumann style computer.

Griffin [Griffin90] observed that, in the typed setting, one could assign the operator *control* (*C*) a type $\neg\neg(P) \supset P$. He observed that *control* is the computational content of the double-negation elimination rule. Griffin also showed that that CPS-translation was a translation from classical propositional logic into a constructive propositional logic.

Often it is efficient to abort computation or exit a program instead of evaluating the whole program, only to discard its result. Tree search is a very natural candidate for using non-local continuations. For example, we want to write a program that sums up the values associated with nodes in a binary tree, but which returns zero if any particular node contained zero for its value. The most efficient implementation would be to exit the program when the first zero is detected. The operation of aborting a computation is not expressible in purely functional languages. A pure functional program would compute out the sums of all the subtrees, only to discard them.

These nonfunctional operations are in fact *nonlocal control operations*. Examples of nonlocal control operations in command-based languages are nonlocal **goto**'s (the most typical explicit control), and escaping primitives with names like "exit", "return", or "break." Examples of nonlocal control in expression-based languages are Scheme's call/cc, LISP's catch/throw, Reynolds' **escape** [Reynolds72].

In general, a nonlocal escape may result in a *type error* caused by "escaping" from deep within expression to the top-level of the program. This means that the reduction-rule reasoning cannot be applied to a program that escapes, since the type of the program may not be preserved by the reduction. For example, consider a program for every boolean function $f{:}Nat \rightarrow \{0,1\}$ (where $0<1$) to attain a minimum [Murthy91]. The specification for the

program is the following proposition:

$$\exists n \in Nat.\forall m \in Nat.f(n) \leq f(m)$$

Intuitively, the program will make a "guess" that N=0 is the desired minimum. Then, given a number m, it will check if $f(0) \leq f(m)$. If so, then it will report success. If not, then $f(m) < f(0)$ which means that $f(m) = 0$. So the program will unwind the context back to that before it chose 0, and instead it will choose m. That is, the program *escapes* from deep within the expression to the top-level. Hence, it is not decidable whether a term (program fragment) is evidence for the proposition $\forall m \in Nat.f(N) \leq f(m)$. As a consequence, an escaping term cannot be assigned the proposition $\forall m \in Nat.f(m) \leq f(N)$ as its type. Such a term is not type-correct.

1.4 Double-Negation Translations and Friedman A-translation

The double-negation translations address the problem of exactly which classical logics can be embedded into their intuitionistic counterparts. The Gödel-translation [Gödel65] solves this problem for predicate logic, and in fact for number theory. That is, the Gödel-translation is an embedding of classical number theory into constructive number theory. If we denote the translation by $(-)^{\circ}$, then what Gödel proved is that

$$\text{if } \vdash_{PA} \Phi \text{ then } \vdash_{HA} \Phi^{\circ}$$

where $\vdash_{PA}$ is a deduction in Peano Arithmetic and $\vdash_{HA}$ is a deduction in Heyting Arithmetic. The Gödel translation is as follows:

$$(A \vee B)^{\circ} \rightarrowtail \neg\neg(A^{\circ} \vee B^{\circ})$$

$$(A \& B)^{\circ} \rightarrowtail A^{\circ} \& B^{\circ}$$

$$(\exists x \in A.B)^{\circ} \rightarrowtail \neg\neg(\exists x \in A.B^{\circ})$$

$$(\forall x \in A.B)^{\circ} \rightarrowtail \forall x \in A.B^{\circ}$$

$$(A \supset B)^{\circ} \rightarrowtail A^{\circ} \supset B^{\circ}$$

---

2. Call/cc abbreviates call-with-current-continuation.

$$P^{\circ} \quad \rightarrowtail \quad \neg\neg(P) \quad (P \text{ prime})$$

Another double-negation translation is the Kolmogorov translation [Kolmogorov67], which in contrast with Gödel translation, double-negates every propositional symbol. This makes it easier to work with. If we denote the Kolmogorov translation by $\overline{(-)}$, then it is defined as follows:

$$\begin{aligned}
\overline{(A \vee B)} &\quad \rightarrowtail \quad \neg\neg(\bar{A} \vee \bar{B}) \\
\overline{(A \& B)} &\quad \rightarrowtail \quad \neg\neg(\bar{A} \& \bar{B}) \\
\overline{(\exists x \in A.B)} &\quad \rightarrowtail \quad \neg\neg(\exists x \in A.\bar{B}) \\
\overline{(\forall x \in A.B)} &\quad \rightarrowtail \quad \neg\neg(\forall x \in A.\bar{B}) \\
\overline{(A \supset B)} &\quad \rightarrowtail \quad \neg\neg(\bar{A} \supset \bar{B}) \\
\bar{P} &\quad \rightarrowtail \quad \neg\neg(P) \quad (P \text{ prime})
\end{aligned}$$

The double-negation embedding result for Kolmogorov translation is the following theorem:

Theorem (Double-Negation Embedding) If $\vdash_C \Phi$, then $\vdash_I \bar{\Phi}$

where $\vdash_C$ is a deduction in classical logic and $\vdash_I$ is a deduction in intuitionistic logic. This theorem tells us that the Kolmogorov translation converts a classically provable sentence into a constructively provable sentence, but it does not tell us what form the constructive proof will take. In other words, once we have a constructive proof of the double-negation-translated sentence, we need to recover a proof of the original sentence. This problem was solved by Friedman [Friedman78] and the solution is described below.

A constructive proof always provides a way to *construct* an object which is proven to exist. In other words, in constructive systems the notions of existence and computability are identified. This is not true about classical systems. A classical proof of an existential sentence does not in general *constructs* the witness. So one might think that classical reasoning is not suitable for reasoning about computation. In 1977, though, Harvey Friedman [Friedman78] showed by a simple syntactic argument called A-translation, that a classical proof of a Π^0_2 sentence indeed gave a constructive proof. More precisely, he discovered that one could replace instances of falsehood (Λ) with an arbitrary proposition, in particular Φ. Friedman A-translation is based on the observation that there is a simple mapping from intuitionistic theories to their minimal counterparts. We recall that a minimal logic is one in which there is no rule of the form:

$$\Lambda \vdash A$$

where A is arbitrary and Λ is the absurdity. This means that Λ is treated as an uninterpreted propositional symbol and, hence, it can be replaced with a new one, A.

Theorem (Friedman A-Translation) If $\vdash_{PA} \Phi$, where Φ is Σ_1^0, i.e., it is Λ-free, then $\vdash_{HA} \overline{\Phi}[\Phi/\Lambda]$.

Friedman then showed:

Theorem (Conservative Extension) If we have a proof $\vdash_{PA} \Phi$, where Φ is Σ_1^0, then we can construct a proof of $\vdash_{HA} \Phi$.

The conservative extension result for Π_2^0 sentences follows by considering free variables. This is one of the most interesting metamathematical results connected with constructivity. It simply says that every Turing machine program that, provably in Peano Arithmetic, converges at all arguments, also terminates provably in Heyting Arithmetic.

Friedman showed how to translate the classical proof in a straightforward manner into a constructive proof of the same sentence. A-translation replaced uses of the double-negation elimination rule with constructive reasoning. Yet, often, the proofs obtained through this translation were hard to read and understand. It was even more difficult to extract programs from these opaque constructions.

As an attempt to address these objections, Murthy [Murthy90] showed that Friedman's result is a proof-theoretic version of CPS-translation from a non-functional programming language (with the "control" C) to a functional programming language. He discovered that one could assign programs to classical proofs in a direct manner, by giving the rule of classical absurdity an algorithmic meaning. Murthy proved that the rule of double-negation elimination is the proof-theoretic form of the (nonlocal control) operator C. He showed that every classical proof can be regarded as a computation; however, as he demonstrated, only sometimes are these computations correct. They are correct only for sentences for which Friedman's translation succeeds, namely, for sentences belonging to Π_2^0 class.

1.5 The Research Contribution of This Work

This thesis has a similar aim as Murthy's work, namely, to demonstrate the algorithmic nature of classical reasoning. However, our work is not based on double-negation embedding but on the *operational interpretation* of propositional connectives, existential witness, and the lack of existential witness. Such an interpretation is formalizable in the second-order propositional logic or Girard's system **F**. The operational interpretation, which consists in performability of certain operations, is in agreement with the intuitionistic semantics of the logical connectives but it is equally applicable to constructive as well as classical logic. In other words, the second-order encodings of logical connectives formalize the constructive interpretation of existence in a manner applicable to classical logic. It should be clear that even though algorithms do not always accompany classical existence proofs, this doesn't mean that classical formal systems are incapable of reasoning about computation. It is rather that in constructive systems the notions of existence and computability are *identified*, and that there are certain practical and mathematical advantages to such an identification. But, again, the identification of proofs with programs is not necessary for proofs to possess the *evidence* property.

The double-negation *translations* convert classically provable sentences into constructively provable sentences, while the second-order *encoding* of the logical connectives represents sentences formed by those connectives *operationally*. That is, the double-negation translations don't distinguish between sentences formed from different logical constants. They are all translated in the same manner, i.e., simply double-negated. On the other hand, the operational interpretation depends on a logical connective that forms a sentence. There is a fundamental difference between the two approaches. The double-negation/A-translation yields a programming language which is inherently higher-order and semantically complex. Such a language doesn't distinguish between continuations and functions. More precisely, the semantic complexity of nonlocal control operator $\mathcal{C}$ completely hides the nature of continuations. A continuation is here simply a "normal" function that will perform a jump when applied. In other words, a continuation is introduced as an "imperative" add-on to a "declarative" language. In contrast, the analysis of classical computation based on the second-order encodings of the logical connectives treats continuations as a purely declarative

concept.

One would like to have a language where explicit access to continuations is a central "declarative" concept. The language yielded by the operational semantics exploited in this thesis is such a language. We will have a separated syntax for discarding the current continuation and making the continuation of a program module accessible for escaping. Our research yields a structured and statically-scoped framework for non-local exists. It lies a foundation for a practical language where a non-local continuation would be considered just as natural, simple, and declarative as a non-local value. One of the consequences of a declarative treatment of continuations is that non-termination can be viewed as a special case of escaping: from the point of call it makes no difference whether the called function is looping forever or has jumped somewhere else. Hence, no new facility is needed to express terminating general recursion.

The consequences of treating continuations as imperative functions may run deep. In principle, a non-local control operator $\mathcal{C}$ is powerful enough to express any function definable in our type theory. However, in practice the translation may not be obvious and may generate hard to read programs, especially when the general recursion is involved. CPS translation replaced nonlocal control operations with their functional programming versions. In addressing the objections to such a translation, namely that they generate hard to understand programs, Murthy's work settles on replacing nonlocal control with imperative functions. Hence, Murthy's work doesn't quite answer the objections that translations can render programs and proofs unintelligible.

The double-negation/A-translation doesn't provide direct classical methods of proof for translated sentences. The second-order definitions, on the other hand, encode operations which are the building blocks for constructing computational proofs. Friedman's *A*-translation tells only how to *recover* the proof of the original sentence after translation. On the other hand, by applying the "top-level" analysis to the second-order encodings of logical connectives, we can construct the computational content of proofs of classical sentences directly.

This thesis offers a new and somewhat unconventional approach to type theory and to understanding existing language constructs and concepts. This approach is as naturally dictated by classical reasoning as the approach to Martin-Löf's intuitionistic type theories was dictated by constructive reasoning.

1.6 Motivation

The double-negation translations, such as those of Gödel and Kolmogorov, translate classical formulas into intuitionistic formulas. The computational significance of this result was not exploited until recently [Murthy90], as was mentioned in the previous section. Similarly, it is well-known that second-order propositional logic encodes propositional connectives into their intuitionistic semantics [Prawitz65]. Definition of the existential witness is also possible in this logic. Even though implicit in this interpretation, its computational content has not been investigated. It is exploited in this thesis.

Another well-known result is that all functions provably total in the second-order Peano arithmetic, PA^2, are representable in the system **F** [Girard70], a formal system of typed terms that encodes the proof theory of second-order propositional logic. The power of system **F** comes from the operation of abstraction on types. For example, if a predecessor function, defined by the following equations

$$pred(0) = 0 \qquad pred(Sx) = x$$

(where S is a successor function) is programmed in **F**, the second equation will only be satisfied when x is a numeral $\bar{n}$. This means that the program decomposes the argument x completely to $SSS...S0$ (with n occurrences of S), then reconstructs it leaving out the last symbol S [GLT89]. Of course we would like to remove the first S instead. This would not change the result of computation and it would make it economical. If it were required that x always evaluates to a numeral, the second equation would be satisfied by a computation that subtracts 1 from its argument.

The *universal abstraction* makes the programs expressible in system **F** decompose their arguments completely. We shall call such computations "by-value". This is characteristic of primitive structural induction as contrasted with more general recursion schemas. Computation "by values only" is a defect of the system **F**, a price paid for its power.

As another example let us consider *factorial* which is inductively defined by the following equations:

$$fact(0) = 1, \qquad fact(Sx) = (Sx)*fact(x)$$

The program for this function in system **F** decomposes x to a numeral $\bar{n}$, then computes a new integer by multiplying together all integers $\bar{i}$ (where $i=1,..,n$) obtained during the reconstruction of $\bar{n}$, except that it takes 1 as a factor instead of 0. If it were required that x *always* computes to a numeral, then the following definition would be equivalent to the given above

$$fact(0) = 1, \qquad fact(x) = x*fact(x-1)$$

For $x\neq 0$, *fact* invokes itself at $x-1$ and the result of this invocation is multiplied by x. If we assume a language with a recursion operator ***rec***, then factorial can be expressed in that language as a recursive functional closure (a recursive value from the function space):

$$\textit{rec}\ fact = \lambda x.\text{if } x=0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ x*fact(x-1)$$

However, the recursion operator introduces a possibility of nontermination. Let us recall from the section 1.3 that in *continuation semantics* [StarWad74] a term is evaluated in a context which represents "the rest of the computation". In continuation semantics, a function of type $A\rightarrow B$ can be seen as a continuation accepting a pair that consists of a value of type A and a B-accepting continuation. Assuming that our hypothetical language has a pairing operation $(\_,\_)$, factorial can be defined as a recursive continuation as follows:

$$\textit{rec}\ fact = \lambda(x,c).\text{if } x=0\ \textbf{\textit{then}}\ c\leftarrow 1\ \textbf{\textit{else}}\ fact\leftarrow(x-1,c')\ \textbf{\textit{where}}\ c' = \lambda k.c\leftarrow(x*k)$$

where the left arrow "$c\leftarrow r$" designates that the result r is sent to continuation c. This notation is introduced to distinguish the accepting of a value by a continuation from a function application. If $x=0$, factorial immediately sends a result to continuation c which will carry on when the computation of factorial is completed. When $x\neq 0$, factorial invokes itself at $x-1$ with a freshly created continuation c' which multiplies the result it receives by x before passing it on to c. If the computation of factorial is the result of the entire computation, then c in the above definition is the initial (or top-level) continuation defined by $\lambda y.y$. The only (non-trivial) continuation left is c' which is generated by *fact* when a subcomputation occurs.

A continuation accepting a pair of an argument value and a continuation is called a *context-typed* continuation, as such a pair is the context of a function application [Fil89]. In the above

definition of factorial, we have used λ-abstraction to designate the context-typed continuation. A context-typed continuation is in fact the (continuation) semantics of a function typed expression. We would like to distinguish between top-level (or empty) context-typed continuation, which is the application context of the entire computation, and non-top-level (local) context-typed continuations, generated when subcomputations occur. The reason for such a distinction is that we want to prevent a classical subcomputation from being accessible for escaping, as such an escape may result in a non-type-correct program (*cf* section 1.3). Our work will demonstrate that in a classical setting, an expression of a functional type corresponds to a value of a function space only when it is "outside" of any recursion. In order to construct a classical type theory, we must (temporarily) abandon the λ-calculus amalgamation of functions and values and distinguish sharply between these two different syntactical classes. The *syntactical abstraction* notation

$$x.e$$

will be used for the expression obtained from e by symbolizing its empty *holes* (requests for data) with the variable x. The only purpose of the prefix "x." is to show what variable is used as the placeholder. The difference between λ-abstraction $\lambda x.e$ and the abstraction $x.e$ is that the former denotes a function space object while the latter describes a function as an expression with holes in it. In other words, a λ-abstraction denotes a value of functional type while a syntactical abstraction is used to designate a fundamental notion of a function. This conceptual distinction will allow us to reserve the λ-notation for expressions denoting entire programs or program modules (i.e., computations accessible for escaping) and use the syntactical abstraction to express subcomputation.

So, assuming a primitive recursion operator *natrec*, the factorial can be defined by

$$\lambda n \in Nat.natrec(n,1,(x,y).x*y)$$

the operator *natrec* being applied to the variable n of type natural numbers, the number 1, and the syntactical abstraction $(x,y).x*y$. The last expression is a *local* context-typed continuation generated by (the definition of) factorial. The result or *nonlocal* continuation is assumed here to be empty, i.e., the factorial is the result of the entire computation. The expression $(x,y).x*y$ multiplies the subresult it receives in y by an number stored in the hole designated by x.

In order that the expression $natrec(e_n, 1, (x,y).x*y)$ evaluates to a number, its first argument, the expression e_n, must also be computable to a number. The operational interpretation of the operator *natrec* is given by a purely mechanical procedure of finding its value when it is applied to all arguments. This procedure is as follows:

if the value of n is 0 then the value of *natrec(n,b,e)* is the value of b;
if the value of n is $k \neq 0$ then the value of *natrec(n,b,e)* is the value of $e(k-1, natrec(k-1,b,e))$.

We want to distinguish types whose values cannot contain unevaluated computations. We want to derive a system of typed terms and reduction rules, such that the reduction process always terminates in an explicit data value. Since we are not restricted to intuitionistic logic, the programs will represent those classical proofs of sentences which provide evidence in a constructive sense [Constable85]. The class of such sentences will be determined by the types of programs which will be derived. We shall see that the computed values are not only integers but include all other usual data types used in computer science, like lists of values, pairs and sums of values, trees, etc. The notion of a value is extended to functions that when given a value, return a value.

1.7 Overview of the Thesis

This thesis consists of seven Chapters. Chapter 2 formalizes the operational content of deduction. The rules of such a formalization constitute an inference system, the Calculus of Impredicative Derivations (COID), for second-order propositional logic. COID is essentially Girard's system F [Girard70]. The propositions of COID represent the algorithmic content of data types. Chapter 3 introduces a distinction between data types and "types" in Computational Type Theory (CTT), i.e., between explicit values and continuations. The dependent function type (Π-type) is introduced into CTT in order to formalize classical types. In Chapter 4, classical predicate logic is interpreted in CTT. The types of existential witness, disjunction, conjunction, and the lack of existential witness are introduced to CTT. These types formalize the operational semantics of the logical connectives at the top-level of proofs. In Chapter 5, CTT is extended to the type theories with booleans, natural numbers, and binary trees. The control operators and program schemas associated with these theories are

introduced. Most importantly, CTT extended with the natural number type, i.e., CTT+Nat formalizes the operational interpretation of arithmetical sentences. Chapter 6 contains examples of computable functions on natural numbers expressed in CTT+Nat. Finally, in Chapter 7 we present conclusions, comparison with other work, and directions for future research.

CHAPTER 2

CALCULUS OF IMPREDICATIVE DERIVATIONS

Many methods have been proposed for investigating the consistency of logical and mathematical theories. All these methods belong to mathematical logic. Their major characteristic is that they are formalistic methods (*calculi*), i.e., they abstract from the meaning of words or symbols. The essence of formalism is that logical validity does not depend on the interpretation of the symbols but on the laws of their combination. However, the understanding and the scope of formalization is different in different methods. Frege, Russell and Whitehead retain in their *logistic* methods the usual meanings of all logical symbols. Hilbert, in his *metamathematics*, takes a more formalistic point of view by considering *all* the symbols of the deductive system under study as meaningless. Thus, Hilbert treats formulas with strict formalism but the methods of inference and deduction are interpreted. Gentzen, in his formal systems of *natural deduction*, separated two roles of the symbol of implication: its role in deductions (usually symbolized by ' $\vdash$ ') and its role as a component symbol of a formula to be proved (symbolized by ' $\supset$ '). While the former role of implication is treated informally by Hilbert (i.e., it is interpreted), Gentzen introduced a new formal symbol for the role of implication in deductions. He extended the formalism to deduction. The premises and conclusions of inference rules in Gentzen's systems are not formulas but formal deductions.

The aim of Hilbert's method is to study provability in the formal system, while Gentzen attempts to define the notion of proof by isolating the essential deductive operations. In particular, under Gentzen's interpretation $A \supset B$ means that there exists a *uniform* procedure for transforming proofs of A to proofs of B.

Gentzen's formal system corresponds closely to the "intuitive meanings" of logical constants but is not in complete agreement. According to the "intuitive explanation", $A \supset B$ is

not in complete agreement. According to the "intuitive explanation", $A \supset B$ is interpreted as asserting the existence of a construction by which any *given* proof of A can be transformed to a proof of B. This is weaker than Gentzen's derivation of B from A as a hypothesis

$$\frac{H, A \vdash B}{H \vdash A \supset B} \qquad (\supset\textit{-intro})$$

(where H is a set of formulas) which is just a special case of such a construction. For example, let $\boldsymbol{N}$ be a one-place predicate constant (for the property of being natural number) defined by the rules

$$\boldsymbol{N}0$$

$$\frac{\boldsymbol{N}a}{\boldsymbol{N}a'}$$

where 0 is an individual constant (denoting zero) and ' is a 1-place operational constant (denoting the successor function). Although valid derivations of $A(0)$ and of $\forall x(A(x) \supset A(x'))$ guarantee the existence of a derivation of $\boldsymbol{N}t \supset A(t)$, valid for every numeral t, there may be no uniformly valid derivation of $\boldsymbol{N}a \supset A(a)$, where a is a parameter, as required by Gentzen's interpretation of implication. The same holds for universal quantification. For example, consider mathematical induction: although valid derivations of $A(0)$ and of $\forall x(A(x) \supset A(x'))$ guarantee the existence of a derivation of $A(t)$, valid for every numeral t, there may be no uniformly valid derivation of $A(a)$, where a is a parameter, as required by Gentzen's interpretation of universal quantification:

$$\frac{A(a)}{\forall x A(x)}$$

This is an indication that the essential constructive deductive operations have not been isolated.

By "marking" proofs and hypotheses in Gentzen's natural deduction systems, one can exhibit the reason for which a formula is true, namely in the form of a λ-term. For example, the reason for the truth of the formula $A \supset A$ is represented by the λ-term $\lambda x{:}A.x$. The calculus that JUSTIFIES the truth of formulas consists of expressions of the from

$$M \vdash t{:}A$$

where A is a formula, t is a λ-term and M is a list of marked hypotheses $x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n$ [Coquand88]. According to the analogy between formulas in natural deduction systems and types in a functional calculus [CurryFeys58, Howard80], one

can demonstrate the consistency of the inference

$$If\ A \vdash A\ then \vdash A \supset A \qquad \textbf{(Id)}$$

by showing that the term $\lambda x{:}A.x$ is strongly normalizable. Explicit consideration of the justification of a formula leads to a more refined study of the notion of truth modulo hypotheses [Coquand88]. For example, the decidability of such a notion of truth reduces to the normalization of λ-terms.

Our ultimate goal is to develop a *computational logic* — a calculus for reasoning about totally correct programs. A program will correspond to a constructive proof (of its specification) that computes *evidence* in a constructive sense [Constable85] for the specification. A program will be a functional term that codes a deductive method used in the proof. Thus, to develop computational logic it is not sufficient to justify formulas, but one has also to code deduction.

What follows is the introduction of a system which **defines** the operational content of deduction, i.e., that defines the **methods** used in the operations performed to convince oneself of the truth modulo hypotheses. All variables (including propositional variables) in such a system will be bound and will have types. The terms that code the deduction methods are the terms of the second-order λ-calculus (or equivalently, terms of Girard's system **F** [Girard72] or Reynold's polymorphic λ-calculus [Reynolds84]). The type language for polymorphic λ-calculus is a second-order propositional logic. Thus, the terms justifying the truth of second-order formulas formalize the deduction methods. Among the methods defined by the terms of polymorphic λ-calculus are the methods used in the operations performed to justify the truth of propositional quantifications themselves. This means that the "universal abstractions", i.e., the terms of the polymorphic λ-calculus, are impredicativite.

2.1 Assumption of Logical Completeness

A system that codes deduction constitutes a calculus of formally constructed proofs. In a system where proofs or derivations are formally constructed, a formula is true only when it is provable, i.e., the system is intuitionistic. Hence, to build a system of formally constructed derivations is to identify truth with provability (i.e., to define *constructive truth*). The notion of constructive truth is formalized by translating into functional terms the operations that one performs to convince oneself of the validity of the assumption of logical completeness,

namely that every true proposition is logically provable. In this way, one obtains a system with a purely syntactical notion of truth, i.e., where the types of derivations are *identified* with true formulas.

Let us call the system of justification of derivations the Calculus of Impredicative Derivations (COID). We shall justify this name shortly. The calculus COID is essentially Girard's system **F** [Girard70]. We use a different name, however, as we give an intuitive interpretation to derivations in **F**.

If the term *prop* is a constant of the calculus and ' $\vdash$ ' is a formal symbol (a mark) of intuitionistic deduction, then the expression

$$\vdash \mathit{prop} \qquad (\varnothing)$$

means that *prop* is a type of propositions. This interpretation is extraneous to the presentation of the formal system being constructed. However, it guides one in achieving the ultimate goal, which is the formal construction of proofs. From the point of view of the formal system, the expression ($\varnothing$) is a construction of an empty *context*. A *context* is a sequence of bindings of propositional and proof variables. Contexts are constructive versions of assumptions.

To justify the assumption of logical completeness, first the validity of an assumption of truth of a proposition has to be justified. Let the symbol ':' designate the *typing* of a variable, i.e., "marking" a variable with its type. Then, if x is a propositional variable, the expression 'x:*prop*' designates a (free) variable of type *prop*. Let the symbol '[_:_]' designate a universal quantifier. Then the expression '[x:*prop*]' designates the "binding" of a variable x.[1]

The assumption of truth of a proposition is justified by a binding of a propositional variable in the context:

$$[x\text{:}\mathit{prop}] \vdash \mathit{prop} \qquad \textbf{(context')}$$

The values of a variable x:*prop* are formulas of COID. From the point of view of a formal

1. A variable is free if the value of a proposition depends on it; otherwise it is bound.

system, the expression (**context'**) is a construction of the context $[x{:}prop]$.

The justification of the assumption of provability of a proposition is expressed by the following rule in COID:

$$\frac{[x{:}prop] \vdash prop}{[x{:}prop][y{:}x] \vdash prop} \qquad \textbf{(context'')}$$

We represent a proof of a true proposition x by introducing a bound variable y with a type x. The expression "$[x{:}prop][y{:}x] \vdash prop$" asserts logical completeness.

If the Greek letters Γ, Δ denote finite juxtapositions of bindings of variables, then the following rule is the generalization of the rule (**context''**)

$$\frac{\Gamma[x{:}prop]\Delta \vdash prop}{\Gamma[x{:}prop]\Delta[y{:}x] \vdash prop} \qquad \textbf{(context1)}$$

2.2 Impredicative Quantification

In this section the assumption of logical completeness will be internalized as an object of type *prop*. The expression "$[x{:}prop][y{:}x] \vdash prop$" asserts that y is a proof of a proposition x. We can infer from it that x is a proposition:

$$[x{:}prop][y{:}x] \vdash x{:}prop \qquad \textbf{(Var')}$$

The derivation (**Var'**) generalizes to an introduction of a bound variable of type M, where M is *prop* or M is of type *prop*:

$$\frac{\Gamma[x{:}M]\Delta \vdash prop}{\Gamma[x{:}M]\Delta \vdash x{:}\,M} \qquad \textbf{(Var)}$$

The expression $[x{:}prop][y{:}x] \vdash x{:}prop$ is a derivation of a proposition. We can internalize this construction as an object of type *prop*. Let '$\Rightarrow$' be a sign of constructive conditional. Then the following inference rule introduces a *constructive implication*:

$$\frac{[x{:}prop][y{:}x] \vdash x{:}\,prop}{[x{:}prop] \vdash x{\Rightarrow}x{:}\,prop}$$

If $M{:}prop$, then the above rule generalizes as follows

$$\frac{\Gamma \vdash M{:}prop,\ \Gamma \vdash N{:}\,prop}{\Gamma \vdash M{\Rightarrow}N{:}\,prop} \qquad \textbf{(Impl)}$$

The following is the formation rule for second-order constructive universal quantifications:

$$\frac{\Gamma[x{:}prop] \vdash N{:}\ prop}{\Gamma \vdash [x{:}prop]N{:}\ prop} \qquad \textbf{(Univ)}$$

The expression $N{:}prop$ above may have free occurrences of the variable x.

The justification of the assumption of provability of a proposition $M{:}prop$ is expressed by the following rule:

$$\frac{\Gamma \vdash M{:}prop}{\Gamma[x{:}M] \vdash prop} \qquad \textbf{(context2)}$$

Let the symbol 'λ' designate constructive inference. The following rule introduces constructive derivations:

$$\frac{\Gamma[x{:}M] \vdash P{:}\ N}{\Gamma \vdash \lambda x{:}M.P{:}\ [x{:}M]N} \qquad \textbf{(Abstr)}$$

If M is of type *prop* in **(Abstr)**, then $[x{:}M]N$ reduces to the implication $M \Rightarrow N$.

An argument by constructive inference is represented by juxtaposition of the expression for the inference and the expression for its premise. In other words, the expression representing the argument is an *application* in second-order λ-calculus:

$$\frac{\Gamma \vdash P{:}\ [x{:}M]N,\ \Gamma \vdash Q{:}\ M}{\Gamma \vdash PQ{:}\ N[Q/x]} \qquad \textbf{(Appl)}$$

$N[Q/x]$ denotes the expression obtained by substituting Q for the variable x in N. This operation is formally definable [deBruijn72]. We shall also write $N(Q)$ instead of $N[Q/x]$.

Let '*red*' denote the relation between an application of a constructive inference to an argument and the value which is constructively derived from that argument. This relation corresponds to β-reduction in second-order λ-calculus:

$$\frac{\Gamma[x{:}M] \vdash P{:}\ N,\ \Gamma \vdash Q{:}\ M}{\Gamma \vdash (\lambda x{:}M.P)Q\ red\ P[Q/x]} \qquad (\beta)$$

2.3 Examples of Basic Types

Constructions of atomic propositions represent the usual data structures (booleans, integers, lists, trees, etc.). Since all terms of COID are functional objects, there is no distinction between data structure and control structure (e.g., booleans implement conditionals, integers

implement loops, etc.).

For example, the quantification

$$id \equiv [x{:}prop]\, x \Rightarrow x$$

where '≡' denoted definitional equality, constitutes the formalization of the concept of '*identity*'. There is only one canonical (i.e., irreducible) object of type *id*:

$$self \equiv \lambda x{:}prop.\lambda e{:}x.e$$

Another formal construction

$$bool \equiv [x{:}prop]\, x \Rightarrow (x \Rightarrow x)$$

expresses a concept of a '*boolean*'. There are two canonical objects of *bool*

$$true \equiv \lambda x{:}prop.\lambda t{:}x.\lambda f{:}x.t$$

$$false \equiv \lambda x{:}prop.\lambda t{:}x.\lambda f{:}x.f$$

which implement conditional.

The following is the type of polymorphic iterators

$$nat \equiv [x{:}prop](x \Rightarrow x) \Rightarrow x \Rightarrow x$$

whose objects are encodings of numbers.

The following quantification

$$tree \equiv [x{:}prop](x \Rightarrow x \Rightarrow x) \Rightarrow x \Rightarrow x$$

represents the logical function of disjunctive conditional. An example of a proof of the proposition *tree*, which constitutes the presentations of the instances of the concept '*binary tree*', is a construction of an empty tree:

$$null \equiv \lambda x{:}prop.\lambda b{:}x \Rightarrow x \Rightarrow x.\lambda n{:}x.n$$

2.4 Impredicative Definitions of Provable Propositions

The calculus COID, defined in the previous sections, does not introduce constructions of predicates and relations. In this calculus, predicates (and relations) are represented by constructive, not computational, implications. For example, let '→' be a symbol of implication, then the system can be extended with the following formation rule for types of predicates on natural numbers:

$$\frac{[x{:}nat] \vdash prop}{\vdash nat \rightarrow prop} \qquad \textbf{(Nat\_predicate)}$$

If the Greek letters Γ, Δ denote finite juxtapositions of constructions of bound variables, and letters M, N denote constructive predicates, relations, etc., then we can generalize the rules **(Nat_predicate)** and **(context'')**, respectively, to the following:

$$\frac{\Gamma[x{:}M] \vdash N}{\Gamma \vdash M \rightarrow N} \qquad \textbf{(predicate)}$$

$$\frac{\Gamma \vdash M}{\Gamma[x{:}M] \vdash prop} \qquad \textbf{(context3)}$$

The language for the operations in the system containing rules **(predicate)** and **(context3)** is an extension of second-order λ-calculus to a higher-order λ-calculus with dependent types [Coquand85, CoqHu86]. The dependent types add extra logical information about terms which may be useless for computation. For example, the following expression represents mathematical induction:

$$NAT \equiv [P{:}nat \rightarrow prop][[m{:}nat]Pm \Rightarrow P(succ\ m)] \Rightarrow Pzero \Rightarrow [n{:}nat]Pn$$

where

$$zero \equiv \lambda x{:}prop.\lambda s{:}x \Rightarrow x.\lambda z{:}x.z$$

$$succ \equiv \lambda n{:}nat.\lambda x{:}prop.\lambda s{:}x \Rightarrow x.\lambda z{:}x.nxsz$$

There is no term of type NAT because the objects of type $nat \rightarrow prop$ are not computational.

For the purpose of the development of programs, the system without the rules **(predicate)** and **(context3)** is sufficient. It constitutes the first step in the process of defining a *computational logic*, i.e., a calculus for constructing correct programs. In this first step, the *basic* types are introduced. They constitute the basis for the second step in the process which is to extend the second-order λ-calculus with *only* such logical content as is useful for computation (in contrast with [Coquand85]). Such an extension is equivalent to the introduction of computational definitions of logical constants. We should note that the second-order λ-calculus can express all programs representing functions provably total in second-order Peano arithmetic, but it is not a *language* of correct programs since its terms lack the logical part that assures the correctness.

2.5 Convertibility

With the notion of β-reduction, '*red*', is associated the notion of *convertibility* '*conv*'. What follows are the rules of inference representing reflexivity, symmetry, transitivity and substitutivity of the *convertibility* relation:

Reflexivity.

$$\frac{\Gamma \vdash prop}{\Gamma \vdash prop\ conv\ prop} \quad \textbf{(RefM)}$$

$$\frac{\Gamma \vdash M{:}N}{\Gamma \vdash M\ conv\ M} \quad \textbf{(RefF)}$$

Symmetry.

$$\frac{\Gamma \vdash M\ conv\ N}{\Gamma \vdash N\ conv\ M} \quad \textbf{(Sym)}$$

Transitivity.

$$\frac{\Gamma \vdash M\ conv\ N \qquad \Gamma \vdash N\ conv\ P}{\Gamma \vdash M\ conv\ P} \quad \textbf{(Trans)}$$

Substitution_in_implication.

$$\frac{\Gamma \vdash P_1\ conv\ P_2 \qquad \Gamma[x{:}P_1] \vdash M_1\ conv\ M_2}{\Gamma \vdash [x{:}P_1]M_1\ conv\ [x{:}P_2]M_2} \quad \textbf{(SubCond)}$$

Substitution_in_abstraction.

$$\frac{\Gamma \vdash P_1\ conv\ P_2 \qquad \Gamma[x{:}P_1] \vdash M_1\ conv\ M_2 \qquad \Gamma[x{:}P_1] \vdash M_1{:}\,N}{\Gamma \vdash \lambda x{:}P_1.M_1\ conv\ \lambda x{:}P_2.M_2} \quad \textbf{(SubAbs)}$$

Substitution_in_application.

$$\frac{\Gamma \vdash M_1N_1{:}\,P \qquad \Gamma \vdash M_1\ conv\ M_2 \qquad \Gamma \vdash N_1\ conv\ N_2}{\Gamma \vdash M_1N_1\ conv\ M_2N_2} \quad \textbf{(SubApp)}$$

The rule (β) has the following rule of conversion as its counterpart:

$$\frac{\Gamma[x{:}M] \vdash P{:}\,N,\ \Gamma \vdash Q{:}\,M}{\Gamma \vdash (\lambda x{:}M.P)\ Q\ conv\ P[Q/x]}$$

The relation of convertibility is used in a crucial way in the argument of the form: [2]

If a is an instance of the expression A and A *conv* B, then a is also an instance of B

We shall add this argument as the following conversion rule:

$$\frac{\Gamma \vdash M:P \qquad \Gamma \vdash P \text{ conv } Q}{\Gamma \vdash M:Q} \qquad \textbf{(TermEq)}$$

2. The argument is a version of Martin-Löf's principle of "equality of types": if a is an object of type A and A is definitionally equal to type B, then a is an object of type B [Martin-Löf72].

CHAPTER 3

COMPUTATIONAL TYPE THEORY

DEPENDENT FUNCTION TYPE

The derivations in the system **F** constitute the formalizations of the operational content of deduction. The propositions provable in the system **F** have their proofs *identified* with their *algorithmic content*, i.e, with the deduction methods provided by the propositional quantifications. These deduction methods correspond to the operational interpretation of data types. Such an interpretation is in agreement with intuitionistic semantics of data types but it is equally applicable to classical logic. This means that proofs of the universal propositional quantifications represent the results of classical programs. Hence, the second-order lambda-calculus or system **F** provide a basis for the connection between logic and computation without constraining the specification logic of computation to be intuitionistic.

The proofs of impredicative universal quantifications are functions defined for arbitrary types, i.e., they are *universal abstractions* [Girard89]. A function of universal type must be "uniform", i.e., do the same thing on all types. Such uniform functions operate without any information about their arguments. Hence, function and data are *identified.* Since *all* data are functions in the system **F**, the distinction between computation rules isn't so clear as it is in a language that has constants of basic types. In the system **F** the algorithms are coded in terms of the data type objects. For example, the natural iteration is confused with natural numbers. The ambiguity of the meaning of the second-order universal quantification is known to be algorithmically consistent.[1] Its negative side is that coded algorithms are different from the

original ones.

We will introduce classical type theories with a clear distinction between different computation rules. In these theories, data types will be distinguished from control structures in order to allow the introduction of actual programs. The separation of control structures from data types will be accomplished by forgetting the internal structure of the normal (irreducible) derivations in system **F**. In this way, the atomic types will be obtained. However, before introducing specific basic types (like integers, lists, trees, etc.), or more precisely, the type theories associated with them, logic has to be formalized. In Chapter 4, classical predicate logic is interpreted in a total-correctness type theory. We call this theory a Computational Type Theory or CTT for short. We will introduce in CTT the classical conjunction and disjunction types, the existential witness type (operational content of existential quantification), and the lack of existential witness type or classical absurdity type. These types will express the operational interpretation of logical connectives. The interpretation of the universal quantification is obtained as simple extension by allowing free atomic variables. However, in order to formalize the operational interpretation of logical connectives, the introduction of a dependent function type is necessary. It will be introduced in this chapter.

Type theory CTT is similar to a declarative programming language with the classical absurdity type for abandoning the normal evaluation and resuming computation in the context of an entire program. We will demonstrate that this kind of "escape" is admissible in a total-correctness framework. All programs expressible in CTT extended to a theory of natural numbers (i.e., CTT+Nat) and other inductive data types are correct and terminate. We will demonstrate that the programs expressible in CTT+Nat are terminating, general recursive programs. But a theory with CTT logic is more than just a programming language with

1. The proof of strong normalization of the second-order λ-calculus is due to [Girard70].

clearly defined operational semantics. Since all the computation rules expressible in CTT will be logically justified, it is possible to develop in it provable correct programs. Thus, CTT+Nat, for instance, should not be compared with a programming language but with a formalized programming logic.

The symbol *prop* in the system **F** played the role of a type of propositions. The constant *data* will be the corresponding type of data types in CTT. Then, there is a judgement in CTT that *data is* a type:

$$data \text{ type}$$

The judgement "$A \in data$" is rendered in words "*A is a data-type*". The constant *data* is a set of all *data-types* of computer science. By a *data-type* we mean a type whose values (but not expressions) do not contain any unevaluated computations, i.e., its *values* are explicit. Expressions of such types can be always computed to explicit values. More precisely, values are integers, booleans, trees, lists, etc. An integer expression, for instance, even if it contains an unevaluated computations, can always be "computed out" to a numeral. The notion of a value is extended to pairs of values, injections of values, and functions from values to values. We will call simple data types (like booleans, identity, etc) and inductive data types (like integers, trees, lists, etc.) *ground* types to distinguish them from structured data types (binary product, binary sum, existential witness type, functional type).

3.1 Prawitz+ Encoding

The following propositional quantification formalizes the condition for existential results over *A* to be witnessed by explicit solutions without restricting the logic to be intuitionistic:

$$(Exists\ A) \equiv [C{:}prop](A \Rightarrow C) \Rightarrow C$$

The proposition schema (*Exists A*) encodes the operational interpretation of any data type *A*. Its objects encode *implicit* values of type *A* since in order to prove (*Exists A*), we have to possess a closed term of type *A*. For example, if *nat* is the type of natural numbers in the system **F**

$$nat \equiv [x{:}prop](x \Rightarrow x) \Rightarrow x \Rightarrow x$$

then (*Exists nat*) is the type of implicit integers. That is, to prove the proposition (*Exists nat*), we have to have a closed term of type *nat*. Such a term reduces to a numeral

$$\Lambda X{:}prop.\lambda z{:}X.\lambda s{:}X{\Rightarrow}X.s(s(s...(sz)...))$$

with n occurrences of s.

This encoding can be extended to structured types (i.e., pairs, sums). The following proposition schemas encode the operational interpretation of conjunction and disjunction:

$$P \ \& \ Q \equiv [C{:}prop](P{\Rightarrow}Q{\Rightarrow}C){\Rightarrow}C$$

$$P \ or \ Q \equiv [C{:}prop](P{\Rightarrow}C){\Rightarrow}(Q{\Rightarrow}C){\Rightarrow}C$$

The proofs of $P\&Q$ and $PorQ$ encode implicit binary pairs and binary injections, respectively. Similarly, the following proposition schema is an example of an inductive type schema, a list of objects of type A:

$$(List\ A) \equiv [C{:}prop](A{\Rightarrow}C{\Rightarrow}C){\Rightarrow}C{\Rightarrow}C$$

The second-order definitions of data types and data type schemas are the types of *implicit* values. We shall refer to the second-order definitions of the propositional connectives, extended with the definitions of the existential witness and with inductive type schemas as "Prawitz+ encoding".

3.2 Data-Types vs. Types, Values vs. Continuations

The propositional schemas (*Exists A*), (*P or Q*), and (*P & Q*) define the proof methods provided by existential witness, disjunction, and conjunction, respectively. Such proof methods are simple, abstract operations: "recover the same," "recover either of two," "recover both." These simple operations are the building blocks for formalizing the full notion of a classically-founded computation. When combined with the operational semantics of ground objects, simple operations will produce new complex abstract operations. For instance, by combining disjunction with the inductive definition of natural numbers, the natural iteration will be introduced in Chapter 5. Similarly, by combining abstract recursion operator expressed by conjunction with the intuitionistic interpretation of natural numbers, the primitive recursion operator and the terminating, general recursion operator will be defined in

Chapter 5. The recursion operators associated with the inductive types of binary trees and lists will be also introduced. We will refer to such constructs as *abstract, mechanical* operations. The recursion operators will be implemented by the totally-correct, classically-founded program schemas. We will show that total, recursive functions correspond to classical proofs of the sentences belonging to class Π_2^0. We will demonstrate that the notion of a total, mechanical operation corresponds to the notion of a total, recursive function.

As we mentioned before, all data, including pairs and injections, are functions in system **F**. As a consequence, there is no clear distinction between different logical connectives as well as between different computation rules. CTT will separate pairs and injections from the algorithmic content of conjunction and disjunction, respectively. This algorithmic content corresponds to the operational semantics of the logical connectives at the *top-level* of proofs. By top-level contexts of proofs, we mean those that expect atomic types, a disjunction of atomic types, or a conjunction of atomic types. Similarly, CTT will separate atomic data from their operational interpretation, i.e., from control structures.

To formalize the separation of data from their operational interpretation, CTT distinguishes between *data-types*, i.e., members of *data*, and what we shall call "types". Data-types are types of requests for explicit ground data. "Types" are the types of structured data value-expecting continuations. A data value-expecting continuation is the context of an entire computation. We will refer to such continuations as the "top-level" continuations. If A is a data-type, then

$$a \in A$$

expresses in CTT that

a is a request for a datum of a ground type A

If A is a "type", on the other hand, then

$$a \in A$$

expresses that

a is a request for a structured data value of type A

In other words, the objects of CTT represent "top-level" continuations of structured types and

requests for ground data. A top-level continuation of a structured type, and most importantly, of the type of the existential quantification $\exists x \in A.P(x)$ expresses a program for computing a witness $a \in A$. Hence, the distinction between "types" and data-types is the distinction between programs and "plain" data.

The judgement "A type" is rendered in words *A is a set.* Not just any set can be a "type" but only a "completely presented set", i.e., a set whose members carry the necessary "witnessing data" by means of the *mechanical operation* of their membership [Beeson80]. In other words, a "type" is what is meant by "type" in the intuitionistic type theories (e.g., Martin-Löf's type theories) except that the meaning of the "mechanical operation" is different. In intuitionistic type theories, a mechanical operation is identified with a function. Hence, a function is an *intensional* function, i.e., the function given together with a description or a rule. In contrast, we will make a clear distinction between functions ("functional graphs") and operations (i.e., top-level continuations).

The semantics of proofs in intuitionistic type theories can be summarized by the slogan

proofs as programs

This *identification* of proofs and programs dictates that the programs use only local reasoning (i.e., they are purely functional) and that the classical laws cannot be used in the proofs. Since the paradigm "proofs as functions" is clearly deficient, we propose a different paradigm, namely

proofs as requests for programs

As we stated above, the judgement $a \in A$, where A is a "type" A, expresses that a is a request for a data value of a structured type A. If "type" A is thought of as a proposition, then the judgement $a \in A$ expresses that

a is a request for a proof of the proposition A

A request for a proof is represented by an object of a constructive system. Hence, only when a proof provides a witness, that witness is the computational content of the proof. Such an interpretation of propositions is applicable to classical logic since the concepts of proof and witness (program) are *not* identified. Hence, by adopting the "*propositions-as-types*"

paradigm (known as Curry-Howard correspondence) but interpreting propositions operationally, we can extend the Curry-Howard interpretation of computation to include nonlocal programming constructs and classical laws. As a consequence, we can reason about programs using such constructs in a totally-correct manner.

Since the top-level continuations are defined in a constructive system, they are presented by constructive objects. Yet, such a treatment of the top-level continuations does not imply that the underlying logic of computation is intuitionistic. Unlike *expressions* which may not denote values because they escape, every *syntactic* continuation also denotes a *semantic* continuation. However, only first-order continuations can be presented in a constructive system without loosing the applicability to classical logic. The reason is that a context-typed continuation (*cf.* Introduction) is the operational semantics of a function typed expression. Hence, allowing the construction of higher-order continuations would be equivalent to treating functions as first-class objects, i.e., allowing values of functional types. Thus, if the setting is constructive, functions would also be constructive. CTT doesn't need to introduce higher-order constructive functions since the top-level operational interpretation of disjunction, conjunction, and existential quantification is formalized by the first-order constructions.

The operational semantics of the underlying language of Computational Type Theory is defined by a set of *reduction* or *evaluation* rules. Each type has terms of two categories: *canonical* and *noncanonical*. In CTT the canonical (irreducible) objects of a "type" represent *implicit values*, i.e., the top-level continuations of structured types. If propositions are interpreted as types, the implicit values correspond to requests for proofs. A request for a proof of a proposition P is formalized in a constructive system, namely in CTT. Hence, if the request is satisfied, i.e., a proof is constructed, that proof is a program (a witness) of type P. This thesis will demonstrate that the computational extracts of the proofs that provide the evidence for propositions in a constructive sense represent total recursive functions. The noncanonical terms in CTT represent the operational content of the top-level contexts of structured types, i.e., they express the *implicit* mechanical operations. When CTT is extended to a particular, first-order theory the canonical terms become explicit values like integers, trees, lists, pairs of integers, injections of integers, total functions from integers to integers,

etc. The noncanonical terms become the actual programs.

Disambiguating the Prawitz+ encoding is a three-step process. First, the types referred to as the single existential witness, the left- and the right-disjunct, and the left- and the right-conjunct will be introduced. Those types provide the building blocks for formalizing full operational semantics of the existential quantification, disjunction, and conjunction. Next, these building blocks are used to define the classical types of conjunction, disjunction, and existential quantification. Finally, in the third step, the classically-founded control structures and program schemas will be introduced and some programming examples will be shown.

3.3 Dependent Function Type

To formalize classical types in CTT, a dependent function type is necessary. The following hypothetical judgement is used to introduce the dependent function types:

$$B(x) \text{ type } [x \in A]$$

This judgement means that for an arbitrary object a of data type A, $B(a)$ is a type. Let 'Π' be the dependent function type constructor. The following is a formation rule for dependent function types:

$$\frac{A \in \mathit{data},\ B(x) \text{ type } [x \in A]}{\Pi x \in A.B(x) \text{ type}_1} \qquad (\Pi\text{-}\mathit{form})$$

The typing of a Π-type as "type$_1$" instead of "type" prevents the construction of higher-order functional terms. The possibility of constructing higher-order functions would have identified the notion of a mechanical operation with an intensional function and, hence, restricted the interpreted logic to be intuitionistic.

If B doesn't depend on A, then $\Pi x \in A.B(x)$ reduces to $A \rightarrow B$ with "B type" or $B \in \mathit{data}$. An object of a dependent function type represents a function whose values are implicit. The objects of a Π-type are introduced by the rule:

$$\frac{b(x) \in B(x)\ [x \in A]}{\lambda x \in A.b(x)\ \in \Pi x \in A.B(x)} \qquad (\Pi\text{-}\mathit{intro})$$

The expression $\lambda x \in A.b(x)$ is *fully evaluated* in CTT. More precisely, an expression is "fully" or "completely" evaluated in CTT when it is in canonical form and all its binding-free intermediate subterms are fully evaluated. Hence, $b(x)$ in $\lambda x \in A.b(x)$ is not to be evaluated since doing so would have been like trying to execute a program which expects an input but the input data is not provided.

The non-canonical constant for the Π-type is the application represented by the juxtaposition of an object of a $\Pi x \in A.B(x)$-type and an object of A:

$$\frac{f \in \Pi x \in A.B(x),\ a \in A}{fa \in B(a)} \qquad (\Pi\text{-}elim)$$

The evaluation rule associated with the Π-type is defined by the following one-step ($\rightarrow_1$) reduction rule:

$$\frac{b(x) \in B(x)\ [x \in A],\ a \in A}{(\lambda x \in A.b(x))a \rightarrow_1 b[a/x]} \qquad (\Pi\text{-}red)$$

CHAPTER 4

CLASSICAL LOGIC AS A TYPE THEORY

Our goal is to find type theories based on classical logic that can be viewed as total correctness specification logics, much like constructive type theories (e.g., HA, Martin-Löf's type theories). When classical systems are viewed as type theories, not every proof normalizes to a term of the type suggested by the propositions-as-types correspondence. Some proofs do not compute evidence, i.e., they "escape" in an incorrect manner (*cf* Introduction). We are looking for classical types (formulas) whose terms (proofs) do not escape within the deep of the expression to the top level. Such escapes mean either type errors (resulting from reliance on classical negation) or nontermination. As a consequence, either the program is not correct with respect to its specification or it contains infinite recursion. Yet, certain applications of the classical laws *are* correct and compute evidence. We will isolate such applications.

To carry out this task, we use the second-order propositional logic to express the *encoding* of classical logic into intuitionistic logic. Other techniques for embedding classical logic into intuitionistic logic include the double-negation *translations* of Gödel [Gödel65] and Kolmogorov [Kolmogorov67] (*cf.* Introduction). As we already pointed out in the Introduction, there is a fundamental difference between the double-negation translation of classical sentences and the second-order encoding of the logical connectives. The double-negation translations don't distinguish between sentences formed from different logical constants. They are all translated in the same manner, i.e., simply double-negated. On the other hand, the operational interpretation depends on a logical connective that forms a sentence. In other words, the operational interpretation discussed in this thesis provides a structured analysis of the classically-founded computation.

In the second-order logic, the correctness of proofs of conjunction, disjunction, and existential quantification is assured by "coding" or "programming" their intuitionistic semantics (*cf.* Introduction). The "programmed" intuitionistic interpretation encodes the *operational use* of (i.e., a proof method provided by) conjunctive, disjunctive, and existentially quantified propositions. For instance, a conjunction $P \wedge Q$ is a method for proving any proposition A, provided one has a proof that A follows from P and Q. This operational interpretation of the logical constants is in agreement with their intuitionistic semantics but it is equally applicable to both classical and intuitionistic logic. The second-order encoding of logical connectives is a continuation-passing-style (CPS) translation applied only to the data-type terms. In other words, only the computationally relevant part of a proof is being analyzed. The universality of the second-order abstractions for the "programmed" logical connectives encodes a condition that the proofs of formulas formed by the connectives are reducible to cut-free forms in all contexts. The computational counterpart of this condition would be that the corresponding programs are totally correct in all contexts. We cannot hope to arrive at a semantics of evidence for classical proofs in arbitrary contexts. But what we really want is to assure that the elimination of the applications of the cut rule preserves the constructive evidence at the top-level of proofs. Similarly for programs, reduction should preserve typing exactly at the top-level of a program. By the top-level of a proof we mean a context consisting of axioms and/or hypotheses whose proofs provide evidence in a constructive sense. Similarly, a top-level context of a program is either empty (a counterpart of axioms) or it is a data value-expecting. Proofs of axioms are collapsed to a unit type which cannot be computationally analyzed.

In this chapter, we will formalize the operational interpretation of existential witness, conjunction, and disjunction in a manner that preserves evidence at the top-level of proofs. The top-level analysis of a proof depends on the structure of the formula to be proved. First, the types specifying the components of the structured contexts in which programs always evaluate to pairs of values, injections of values, and ground values are introduced into CTT. Second, these types are used as building blocks in formalizing the full operational semantics of logical connectives. In this chapter we will also introduce into CTT the classical absurdity

type and a nonlocal control operator of that type to obtain a classical programming logic. We will show that CTT expresses the operational interpretation of Π_2^0 sentences. We will prove the strong normalization of CTT.

4.1 Single Existential Witness Type

The quantifier "$\exists$" can be interpreted in the second-order predicate logic [Prawitz65] as follows:

$$Sig(R,Q) \equiv \forall^2 X.(\forall x{:}R.Q(x) \supset X) \supset X.$$

But the operational interpretation of a witness of the existential quantification can be formalized in the second-order propositional logic, namely by the quantification

$$(Exists\ P) \equiv [C{:}prop](P \Rightarrow C) \Rightarrow C. \qquad \text{(E)}$$

The quantification (*Exists P*) encodes the condition for the existential results to be witnessed by explicit solutions. The explicit solutions to existential formulas imply that all application of the cut rule can be eliminated from their proofs. When we have a cut-free proof of "$\vdash A$," then the last rule applied in the deduction must be a logical rule (vs. structural rule in sequent calculus). This has immediate consequences, e.g., if A is $\exists y\ B$, then $B(t)$ has been proved for some t, and similarly for the other logical connectives [Girard89]. In other words, a cut-free proof of a formula provides the evidence for that formula in a constructive sense. Thus, a normal proof of the quantification (*Exists P*) encodes the constructive meaning of the existential quantification even when the logic includes the classical rules for negation and disjunction.

The definition of the existential witness in F is based on the elimination rule ($\exists$E) for the existential quantification rather than on its introduction rule ($\exists$I) presented in Chapter 1. That is, the proposition schema (E) encodes the elimination rule of the existential witness. Existential quantification is interpreted as stating the performability of certain operation instead of asserting the existence of a value for which certain property holds. The interpretation of the existential witness based on the elimination rule is in agreement with the intuitionistic semantics but it is equally applicable to classical logic. This is the *operational*

interpretation of the existential quantification.

The following deduction in system F represents the introduction rule for (*Exists P*):

$$[P{:}prop][x{:}P] \vdash \Lambda C{:}prop.\lambda u{:}P \Rightarrow C.(u\ x) \qquad (Wit)$$

We note that if P represents a data type then the formula (*Exists P*) is a continuation-passing-style (CPS) translation of that type [Murthy90]. The implication $P \Rightarrow C$ is the type of a continuation-representing function u of a program that evaluates to type P in any context C. However, we cannot arrive at a semantics of evidence in arbitrary contexts. Hence, instead of an arbitrary context, we would like to explicitly specify that *there is* a meaningful continuation. We can accomplish this by making u return a data-typed result, since an evaluated computation does not contain control side-effects. This also means that as a function, u will always return.

Since CTT distinguishes between data-types and types (like Π-type), we can express in it the requirement that the type variable C in (*Exists P*) ranges over (non-empty) data-types. A continuation-representing function u: $P \Rightarrow C$ that always returns with an arbitrary data-typed result, is represented in CTT as an inference rule that ignores its premise. Such a rule contains no information. Only when the type of the result of u becomes a specific data-type, does the corresponding rule carry information. We cannot yet, however, make C specific since (E) is not a proposition, but a proposition schema — a macro of the meta-language expanding into a proposition. This means that C and P are expressed on different levels of notation: P is introduced on the meta-level while C is expressed in the object language (i.e., in the second-order λ-calculus). First, we need to internalize in CTT the proposition schema (E) before C can be made specific.

The context (i.e., the sequence of bindings on the left-hand side of the deduction sign) of the deduction (*Wit*) *describes* what is needed to prove the proposition (*Exists P*), namely a witness to which the proof of the existential quantification evaluated. The proposition schema (*Wit*) is represented in the formalism of CTT as a type schema specifying the kind of context in which proofs of existential quantification always reduce to cut-free forms. We will refer to such a context as a "top-level context" since it is a context in which a proof reducible to a cut-

free form concluded. Computationally, such a context corresponds to the context of an entire computation. A cut-free proof concluded in a top-level context is a "top-level proof". In other words, a top-level context is the *operational interpretation* of a top-level proof. This operational analysis of the top-level proofs is carried out in this thesis.

Let the data type *Id* be a "unit" type which has a unique value *it*. The following are the formation and the introduction rules for *Id* in CTT:

$$Id \in data \qquad (Id\text{-}form)$$

$$it \in Id \qquad (Id\text{-}intro)$$

The data type *Id* formalizes in CTT the notion of "classical" evidence, i.e., of "classical" constructive truth. It can represent any proposition whose proofs provide evidence in the data value-expecting contexts. Let A be a CTT variable corresponding to P and $a \in A$ be a witness. Let 'Σ^*' be the symbol of the type schema in question. The following rules internalize the schema (*Wit*) into the formalism of CTT:

$$\frac{A \in data, \quad a \in A}{\Sigma^*(A,a) \text{ type}} \qquad (\Sigma^*\text{-}form)$$

$$\frac{C \in data, \quad c \in C}{[c]_C \in \Sigma^*(Id,it)} \qquad (\Sigma^*\text{-}intro)$$

$$\frac{C \in data, \quad d \in \Sigma^*(A,a)}{split_C(d) \in C} \qquad (\Sigma^*\text{-}elim)$$

$$\frac{C \in data, \quad c \in C}{split_C([c]_C) \rightarrow_1 c} \qquad (\Sigma^*\text{-}red)$$

The Σ^*-type schema only internalizes the *form* of a context in which proofs of the existential quantification are always reducible to cut-free forms, namely it tells us that this context is expecting a data-type object. It does *not* internalize, however, the witness of the existential quantification. It only says that when a proof of a sentence provides a witness, then in any data type-expecting context, that sentence can be represented by the type *Id*. Hence, the notions of proof and evidence are not identified, making the Σ^*-type schema applicable to

classical logic.

The object $[c]_C$ represents an implicit value of an arbitrary data-type C. The evaluation of $split_C([c]_C)$ to c represents an *arbitrary computation* which recovers an arbitrary value c to which a program evaluated.

Under the operational interpretation provided by the second-order encodings of the logical connectives, the type of the top-level context of a proof depends on the structure of the formula we are proving. In the case of the existential quantification, the construction of the top-level, operational interpretation of the existential witness is composed of value-tagged constructions, each handling one of the witnesses. Thus, a type has to be introduced that formalizes this basic component of the top-level interpretation of existential witness. We will call this type a *single existential witness type*. It corresponds to the propositional universal quantification (*Exists P*) being instanciated to P. The following rules define a single existential witness type in CTT:

$$\frac{A \in data, \;\; a \in A}{\Sigma(A,a) \text{ type}} \qquad (\Sigma\text{-}form)$$

$$\frac{a \in A}{[a] \in \Sigma(A,a)} \qquad (\Sigma\text{-}intro)$$

$$\frac{d \in \Sigma(A,a)}{split(d) \in A} \qquad (\Sigma\text{-}elim)$$

$$\frac{a \in A}{split([a]) \rightarrow_1 a} \qquad (\Sigma\text{-}red)$$

The type $\Sigma(A,a)$ is tagged with a data value a, a witness of $\exists x \in A.P(x)$, to assure that the last rule applied in the proof of $\exists x \in A.P(x)$ is the introduction ($\exists$ I) which has an immediate consequence that $P(a)$ was proved. Yet, such a definition of the existential witness type is applicable to classical logic since the concepts of proof and witness are *not* identified. Rather, only when a proof provides a witness, that witness is the computational content of the proof. The type $\Sigma(A,a)$ expresses the construction of the computational content of the top-level cut-free proofs of the existential quantification for the single witness a. The type $\Sigma(A,a)$ is necessary to start off the construction of the operational interpretation of classical proofs.

The canonical object $[a]$ of type $\Sigma(A,a)$ represents an implicit value of data-type A. For instance, if A is the type of natural numbers *Nat*, [0] is an implicit integer from which the explicit value 0 is simply "read off". The evaluation of *split*([0]) to 0 represents an *empty computation* which simply recovers the same value to which a program evaluated. The reduction rule for Σ-type expresses the primitive operation of identity, i.e., of "recovering the same".

4.2 Left- and Right-Disjunct Types

The impredicative construction of the operational interpretation of disjunction is as follows:

$$P \text{ or } Q \equiv [C{:}prop](P \Rightarrow C) \Rightarrow (Q \Rightarrow C) \Rightarrow C \qquad \textbf{(or)}$$

This definition of disjunction is based on the elimination rule (∨E) rather than on the introduction rules (∨I) for disjunction introduced in Chapter 1. That is, disjunction is interpreted as stating the performability of certain operations instead of asserting the existence of certain constructions. The latter is the usual way a disjunction and other logical constants are interpreted. The interpretation of disjunction based on the elimination rule is in agreement with its intuitionistic semantics but it is equally applicable to classical logic. It constitutes a method for inferring certain formulas, namely the proposition $P \vee Q$ is a *method* of proving any proposition C provided one has either a proof that C follows from P or a proof that C follows from Q. This is the *operational interpretation* of disjunction. The following deductions of the system **F** introduce disjunction:

$$[P{:}prop][Q{:}prop][x{:}P] \vdash \Lambda C{:}prop.\lambda u{:}P \Rightarrow C.\lambda v{:}Q \Rightarrow C.(u\ x) : (P \text{ or } Q) \quad (Inl)$$

$$[P{:}prop][Q{:}prop][y{:}Q] \vdash \Lambda C{:}prop.\lambda u{:}P \Rightarrow C.\lambda v{:}Q \Rightarrow C.(v\ y) : (P \text{ or } Q) \quad (Inr)$$

The type schema $(P \text{ or } Q)$ encodes the method of construction of cut-free proofs of disjunction from cut-free proofs of P and Q[1].

We note that if P and Q represent data types then the formula (P *or* Q) is a CPS-translation of a binary sum of P and Q. The implications $P \Rightarrow C$ and $Q \Rightarrow C$ are the types of continuation-representing functions u and v of a program that evaluates to either a p of type P or a q of type Q in any context C. Both u and v return results of the *same* type. This means that either of the parts of the program computing the two disjuncts will return to the same place of call, provided that at least one part returns. The universal quantification in the formula (**or**) assures that such a program is type-correct and terminates. Such a program corresponds to a proof that provides evidence in a constructive sense for a disjunctive formula.

As in the case of the existential witness, we will restrict the context C in the definition of disjunction to be a data value-expecting. We will skip the case of disjunction schema, where the context is an arbitrary non-empty data type. This case would have corresponded to, what we may call, the *arbitrary sum injections*. Instead, we will restrict the context C to be either P or Q and obtain two components of the top-level interpretation of disjunction.

Since the type of the top-level context of a proof depends on the structure of the formula being proved, two types have to be introduced in CTT to define disjunction. They formalize two components of the top-level interpretation of disjunction. Two tagged types are necessary to handle each of the disjuncts, i.e., to specify whether the evidence of (P *or* Q) comes from the evidence of the left disjunct P or the right disjunct Q. We will refer to these component types as *left-* and *right-disjunct types*. These types on their own are not sufficient to express disjunction. They will be used as building blocks in defining the full top-level, operational semantics of disjunction. Their introduction is necessary in order that disjunction type be applicable to classical logic. More precisely, they prevent the general *identification* of proofs of disjunction with sum injections. Rather, only when a proof of one of two disjuncts

---

1. When we have a cut-free proof of $\vdash A \vee B$ then the last rule applied in the deduction must be a logical rule. This has immediate consequence, namely that A has been proved or that B has been proved, and that there is a tag telling us which disjunct we were getting evidence for.

provides a witness, the injection of that witness is the evidence for the proof. Hence, we need a type tagged not only with a witness but also with the information what disjunct it is a witness for.

Let '$\vee_L$' and '$\vee_R$' be the symbols of "left" and "right" disjunct types, respectively. Let A and B be the data-type variables in CTT corresponding to P and Q. The following rules define the components of the operational interpretation of disjunction type in CTT:

$$\frac{A \in data,\ B \in data,\ a \in A}{\vee_L(A,B,a)\ \text{type}} \qquad (\vee_L\text{-}form)$$

$$\frac{A \in data,\ B \in data,\ b \in B}{\vee_R(A,B,b)\ \text{type}} \qquad (\vee_R\text{-}form)$$

$$\frac{a \in A}{inl(a) \in \vee_L(A,B,a)} \qquad (\vee_L\text{-}intro)$$

$$\frac{b \in B}{inr(b) \in \vee_R(A,B,b)} \qquad (\vee_R\text{-}intro)$$

$$\frac{d \in \vee_L(A,B,a)}{outl(d) \in A} \qquad (\vee_L\text{-}elim)$$

$$\frac{d \in \vee_R(A,B,b)}{outr(d) \in B} \qquad (\vee_R\text{-}intro)$$

$$\frac{a \in A}{outl(inl(a)) \rightarrow_1 a} \qquad (\vee_L\text{-}red)$$

$$\frac{b \in B}{outr(inr(b)) \rightarrow_1 b} \qquad (\vee_R\text{-}red)$$

The type $\vee_L(A,B,a)$ expresses the construction of a cut-free proof of the left disjunct in the context of that disjunct. Such a construction assures that the last rule applied in a proof is the left of the two introductions in ($\vee$I) which has an immediate consequence that A has been proved and that there is a tag telling us that it is the left disjunct we were getting evidence for. Similarly, the type $\vee_R(A,B,b)$ expresses the construction of a cut-free proof of the right disjunct in the context of that disjunct. The types $\vee_L(A,B,a)$ and $\vee_R(A,B,b)$ provide the

building blocks for the construction of cut-free proofs of full disjunction. More precisely, one has a top-level cut-free proof of $A \vee B$ constructed from cut-free proofs of either A or B, if either

$$\dfrac{\dfrac{A}{A \vee B} \quad \begin{matrix}[A] \\ A\end{matrix} \quad \dfrac{\begin{matrix}[B] \\ \Sigma_A\end{matrix}}{A}}{A} \qquad (\Pi_L)$$

or

$$\dfrac{\dfrac{B}{A \vee B} \quad \begin{matrix}[B] \\ B\end{matrix} \quad \dfrac{\begin{matrix}[A] \\ \Sigma_B\end{matrix}}{B}}{B} \qquad (\Pi_R)$$

since the derivations Π_L and Π_R reduce to axioms A and B, respectively. Σ_A and Σ_B above denote finite sequences of derivations of A and B, respectively. We note that in Π_L the proof of $A \vee B$ from B may not be reducible to a cut-free form since the proof of B may not be cut-free. Similarly, for the derivation Π_R. The types $\vee_L(A,B,a)$ and $\vee_R(A,B,b)$ define the components of a primitive, abstract operation expressed by disjunction of recovering either a value of data-type A or a value of data-type B.

4.3 Left- and Right-Conjunct Types

The following second-order propositional quantification expresses the operational interpretation of conjunction:

$$P \;\&\; Q \equiv [C{:}prop](P \Rightarrow (Q \Rightarrow C)) \Rightarrow C \qquad (\&)$$

This *procedural* interpretation of conjunction is in agreement with its intuitionistic semantics. The proposition $P \wedge Q$ is a *method* of proving any proposition C provided one has a proof that C follows from P and Q. The proposition schema $(P \;\&\; Q)$ encodes the condition for proofs of conjunction to be reducible to a cut-free form[2].

The following deduction in the system **F** represents the introduction rule for conjunction:

$[P{:}prop][Q{:}prop][x{:}P][y{:}Q] \vdash \Lambda C{:}prop.\lambda h{:}P \Rightarrow Q \Rightarrow C.(h\ x\ y) : (P\ \&\ Q)$ (Pair)

We note again that if P and Q represent data types then the formula $(P\ or\ Q)$ is a CPS-translation of a binary product of P and Q. The implication $P \Rightarrow Q \Rightarrow C$ is the type of a continuation-representing function h of a program that evaluates to a pair of values of types P and Q in any context C. The uniformity of second-order quantifications assures that such a program does not contain control side-effects (i.e., it is type-correct and terminates). Such a program corresponds to a proof that provides evidence for a conjunctive formula in a constructive sense. The program computes the pair sequentially. That is, first, it computes one of the conjuncts in the initial context, and then it computes the other conjunct in the context of the first computation. We can choose either the left or the right conjunct to be computed first. This specifies the evaluation order for pairs, i.e., whether they are evaluated from left-to-right or from right-to-left.

We want to define classical type of conjunction directly, not through second-order encoding. We note that the continuation-representing function $h{:}P \Rightarrow (Q \Rightarrow C)$ is higher-order, i.e., it returns a function of type $Q \Rightarrow C$. In other words, the computation of type Q is not at the top-level of the program of type of conjunction of P and Q, and if it escapes, the program may not be type correct. Thus, in order that a program for computing a value of type P in the context of the computation of a value q of type Q always returns, the computation of q cannot escape. To assure this, the type of the context of the computation of q has to be Q, that is, q represents an evaluated computation. As a consequence, with each value of type Q there is associated a value p of type P, assuming the evaluation from left-to-right. The same is true when the right-to-left order of evaluation is assumed, where with each value of type P there is

---

2. When we have a cut-free proof of $\vdash A \wedge B$ then the last rule applied in the deduction must be a logical rule, and this has immediate consequence, namely that A has been proved and that B has been proved.

associated a value of type Q.

The direct method of construction of cut-free proofs of conjunction (P *or* Q) requires that the initial context is of data type P or of data type Q. This corresponds to instanciating the propositional universal quantification in (P & Q) to either P or Q. When (P & Q) is instanciated to P, then the continuation-representing function $h{:}P{\Rightarrow}(Q{\Rightarrow}P)$ ignores its second argument and returns its first argument. Similarly, when (P & Q) is instanciated to Q, then the continuation-representing function $h{:}P{\Rightarrow}(Q{\Rightarrow}Q)$ ignores its first argument and returns the second. This corresponds to introducing the first and the second projections of a pair. In other words, the definition of conjunction is based here on the elimination rules ($\wedge$E) rather than on the introduction rule ($\wedge$I) introduced in Chapter 1. Conjunction is interpreted as stating the performability of certain operation instead of asserting the existence of a certain construction.

As we have already pointed out, the type of the top-level context of a proof depends on the structure of the formula we are proving. To internalize the schema (&) into the formalism of CTT, two types have to be introduced which formalize two components of the top-level, operational interpretation of conjunction. Two tagged types are necessary to handle each of the conjuncts, i.e., to specify whether the top-level context of the proof of (P & Q) is that of the left conjunct P or the right conjunct Q. We will refer to these component types as the *left-* and the *right-conjunct types*. These types specify the components of a structured context in which proofs of conjunction always reduce to cut-free forms. As in the disjunctive case, the left- and right-conjunct types will not be able to express conjunction directly but will be used subsequently to do so in later sections of this chapter. The introduction of the left- and the right-conjunct types is necessary to define a full *classical* conjunction type. The left- and the right-conjunct types prevent the general *identification* of proofs of conjunction with pairs. Such an identification restricts the underlying logic to be intuitionistic. Rather, only when there are two proofs, obtained in a sequence, each providing a witness for one of the conjuncts, a *pair* of these witnesses is the evidence for the proof. Whichever proof is the first in the sequence, it has to return to its initial context in order that the second witness can be computed. Yet, in classical logic a proof can "escape" as the result of the application of the

rule of double-negation elimination. At this stage of construction of CTT, we can only formalize the fact that either one of the two witnesses was obtained. Hence, we need a type tagged not only with a witness but also with the information what conjunct it is a witness for.

Let $\wedge_L$ and $\wedge_R$ be the symbols of the "left" and the "right" conjunct types, respectively. The subscripts specify whether the initial context is of a type of the left or of the right conjunct. If the initial context is of a type of the left conjunct A (a type variable in CTT corresponding to P), then the program for the second component of the pair can escape. Similarly, if the result of h is of type of the right conjunct B (a type variable in CTT corresponding to Q), the program for the first component of the pair may not be totally correct. In either case, on the logical side, the proofs of conjunction may contain irreducible applications of the cut rule, i.e., they may not provide the evidence for conjunction in a constructive sense.

The following rules define the left- and right- conjunct types:

$$\frac{A \in data,\ B \in data,\ b \in B,\ a \in A}{\wedge_L(A,B,a,b)} \qquad (\wedge_L\text{-}form)$$

$$\frac{A \in data,\ B \in data,\ b \in B,\ a \in A}{\wedge_R(A,B,a,b)} \qquad (\wedge_R\text{-}form)$$

$$\frac{a \in A}{(a,\mathit{tt}) \in \wedge_L(A,Id,a,\mathit{tt})} \qquad (\wedge_L\text{-}intro)$$

$$\frac{b \in B}{(\mathit{tt},b) \in \wedge_R(Id,B,\mathit{tt},b)} \qquad (\wedge_R\text{-}intro)$$

$$\frac{d \in \wedge_L(A,B,a,b)}{fst(d) \in A} \qquad (\wedge_L\text{-}elim)$$

$$\frac{d \in \wedge_R(A,B,a,b)}{snd(d) \in B} \qquad (\wedge_R\text{-}elim)$$

$$\frac{a \in A}{fst((a,\mathit{tt})) \rightarrow_1 a} \qquad (\wedge_L\text{-}red)$$

$$\frac{b \in B}{snd((\mathit{it},b)) \rightarrow_1 b} \qquad (\wedge_R\text{-}red)$$

We recall that the one-element type *Id* can represent in CTT an arbitrary non-empty data-type. The type $\wedge_L(A,Id,a,\mathit{it})$ formalizes the introduction of the first witness for conjunction and merely the existence of a witness for the second conjunct. In other words, the computation of the second witness may escape. This means that the proof of the second conjunct may be by contradiction, in which case it is not of type *B*. Similarly, the type $\wedge_R(Id,B,\mathit{it},b)$ formalizes the introduction of the second witness and the existence of a witness for the first conjunct. In this case, the first witness may not be of type *A*.

The $\wedge_L$- and $\wedge_R$-types formalize only the operations of "reading off" either the first or the second component of a pair. The operation of "reading off" both values has not yet been completely formalized. The type $\wedge_L(A,Id,a,\mathit{it})$ is a "suspended type" in the value of the second component. Similarly, the $\wedge_R$-type defines a pair that "is suspended" (*lazy*) in its first component. This means, that a program returning just one component of a pair may delay the evaluation of the other component. If this other component, say *c*, is never requested, the entire program is still correct even when the evaluation of *c* would never terminate. One can easily see that this reflects the technique used to obtain call-by-name behaviour in a call-by-value language by delaying evaluation [Plotkin75].

The types $\wedge_L$ and $\wedge_R$ express the construction of two cut-free components of top-level cut-free proofs of conjunction. The left component is constructed in the context of the left conjunct. Similarly, the right component is constructed in the context of the right conjunct. This assures that the last rule applied in a proof of conjunction *contains* the introduction of the left or of the right conjunct which has an immediate consequence that either *A* has been proved or that *B* has been proved, and that there is a tag telling us which conjunct we were getting evidence for. The types $\wedge_L(A,B,a,b)$ and $\wedge_R(A,B,a,b)$ provide the building blocks for the construction of cut-free proofs of full conjunction. More precisely, one has a top-level (but not necessarily yet cut-free) proof of $A \wedge B$ constructed from cut-free proofs of either *A* or *B*, if either

$$\dfrac{\dfrac{A \quad \dfrac{\Sigma_B}{B}}{A \wedge B}}{A} \qquad (\Pi_L)$$

or

$$\dfrac{\dfrac{\dfrac{\Sigma_A}{A} \quad B}{A \wedge B}}{B} \qquad (\Pi_R)$$

since the derivations Π_L and Π_R reduce to axioms A and B, respectively. As before, Σ_A and Σ_B above denote finite sequences of derivations of A and B, respectively. We note that in the derivations Π_L the proof of $A \wedge B$ may not be reducible to a cut-free form since the proof of B may not be cut-free. Similarly, for the derivation Π_R where the proof of A may not be cut-free.

4.4 Classical Types

In the previous section of this chapter, we have introduced the second-order encodings of the methods of construction of cut-free proofs of conjunction, disjunction, and existential quantification. These encodings constituted the CPS-translation of the binary sum, the binary product, and the existential witness types. The second-order λ-abstraction required that the proofs of conjunction, disjunction, and existential quantification be reducible to a cut-free form in an arbitrary context. As we pointed out before, we cannot hope to arrive at a semantics of evidence for classical proofs in such an arbitrary setting. We can only assure that the applications of the cut rule preserve the constructive evidence at the top-level of proofs. In order to apply the top-level evaluation strategy to the operational interpretation of the logical connectives, we have introduced in the previous section left- and right-conjunct types, left- and right-disjunct types, and single existential witness type. These types specify the components of the structured top-level contexts in which proofs of formulas constructed of these logical connectives always reduce to cut-free forms. They provide the building blocks for program schemas implementing the full operational semantics of these connectives. In this section we will use these building blocks to define classical disjunction, conjunction, and

existential witness types. These types will express the full operational interpretation of classical connectives in a manner that preserves the constructive evidence at the top-level of proofs. The type universal quantification results as a simple extension of allowing free data-typed variables in the interpretation of existential quantification.

We will also introduce the type for the double-negation elimination rule applied in an atomic formula-expecting context. This is the classical absurdity type. We will show that the computational content of the classical absurdity type is the abandoning a normal evaluation of a program and resuming computation at a ground-value expecting context. We will introduce a programming construct "*resultis*" for abandoning the current path of evaluation and installing new final result of a program.

4.4.1 Disjunction Type

In this section we will introduce a type for classical disjunction in a manner that preserves the constructive evidence at the top-level of proofs. This new type will be constructed from the left- and right-disjunct types, each handling the top-level construction of one of the disjuncts. Let "+" be the symbol of the type of disjunction. Its definition is as follows:

$$\frac{C \in data,\ A \in data,\ B \in data}{A +_C B \text{ type}} \qquad (+\text{-}form)$$

$$\frac{f \in C \to \vee_L(A,B,a),\ g \in C \to \vee_R(A,B,b)}{<f,g> \in A +_C B} \qquad (+\text{-}intro)$$

$$\frac{d \in A +_C B}{d^0 \in C \to A,\ d^1 \in C \to B} \qquad (+\text{-}elim)$$

$$\frac{f \in C \to \vee_L(A,B,a),\ g \in C \to \vee_R(A,B,b)}{<f,g>^0 \to_1 \lambda c \in C.outl(fc),\ <f,g>^1 \to_1 \lambda c \in C.outr(gc)} \qquad (+\text{-}red)$$

Disjunction type is the type of top-level proofs of disjunction. The functions f and g in $(+-intro)$ represent the component parts of the top-level contexts for disjunction. These functions are constructive (i.e., they are objects of constructive $\to$-type) in order to assure that the proofs of disjunction are reducible to cut-free forms, i.e., that the witnesses of the left- and

the right-disjunct are requested in a data value-expecting context. The data type C plays the role of this context. C is distributed over a pair of types, each handling one of the disjuncts. The construction $<f,g>$ represents a structured, top-level context for disjunction. In other words, CTT provides a structured framework for the top-level contexts of classical programs. In contrast, the treatment of the top-level contexts for classical types in [Murthy90] is powerful but unstructured. There, continuations are presented as "normal" functions while CTT conceptually distinguishes between contexts (represented by constructive functions) and classical functions.

From the point of view of proof theory, the type $A +_C B$ expresses the construction of cut-free proofs of disjunction in the context C. As we pointed out before, the definition of disjunction (and other logical connectives) in CTT is based on the elimination rather than on the introduction rule of natural deduction. More precisely, one has a cut-free proof of $A \vee B$ constructed from cut-free proofs of A or B in the context of an atomic proposition C, if

$$\frac{\begin{array}{ccc} X & [A] & [B] \\ A \vee B & C & C \end{array}}{C} \qquad X = A \text{ or } X = B \qquad (\Pi_C)$$

since Π_C reduces to a normal derivation $\begin{matrix}[X] \\ C\end{matrix}$. The definition of +-type assures that the proofs of A and B are reducible to a cut-free form since the subterms $outl(fc)$ and $outr(gc)$ have to reduce to atomic objects a and b respectively. The proofs of A and B are implicit which is expressed by the fact that the terms $\lambda c \in C.outl(fc)$ and $\lambda c \in C.outr(gc)$ are fully evaluated in CTT.

When C is A in $A +_C B$ then the specialized type $A +_L B$ is obtained and is defined as follows:

$$\frac{A \in data, \quad B \in data}{A +_L B \text{ type}} \qquad (+_L\text{-}form)$$

$$\frac{f \in \Pi a \in A.\vee_L(A,B,a), \quad g \in A \rightarrow \vee_R(A,B,b)}{<f,g> \in \; A +_L B} \qquad (+_L\text{-}intro)$$

$$\frac{d \in A +_L B}{d^0 \in A \to A,\ d^1 \in A \to B} \qquad (+_L\text{-}elim)$$

$$\frac{f \in \Pi a \in A.\vee_L(A,B,a),\ g \in A \to \vee_R(A,B,b)}{<f,g>^0 \to_1 \lambda a \in A.a,\ <f,g>^1 \to_1 \lambda a \in A.outr(ga)} \qquad (+_L\text{-}red)$$

From an operational point of view, the type $A +_L B$ internalizes the operations of *duplicating* a data type A-expecting continuation and of requesting the normal result of a program of type B which is evaluated in the context of type A. The first operation is represented by the term of type $\Pi a \in A.\vee_L(A,B,a)$, the second is a term of type $A \to \vee_R(A,B,b)$. Duplication of a context implies preserving a state of computation that can be reset in case an escape occurs. Hence, if the computation in B escapes it can resume at the preserved context of type A.

From a categorical point of view, a term of type $A +_L B$ yields an *iterated* function of type $A \to B$ by mapping a continuation of type A either to itself or to a continuation of type B. If the result is of type A, it can be applied again, etc., until it yields an object in B. This corresponds to the categorical notion of *iteration*. In CTT which interprets pure predicate logic every term represents a continuation. There are no classical, i.e., explicit values in CTT. Categorical iteration by itself doesn't yet compute any value. It is an abstract operation which can operate on values (if they are provided) as well as on continuations. Here, iteration operates on continuations. Nothing yet distinguishes consecutive iteration steps (i.e., reapplication of a continuation of type A) except that during those reapplications classical reasoning is not allowed. Hence, the context stays local (escape-free) until a context, in which a result is requested in b, is reached. This context is the top-level context of an entire computation. Since operationally a local context corresponds to an unchanged context, the operation on a local context is an identity. In CTT, nothing yet is known about computation. CTT provides only a type-theoretic framework of the distinction between a top-level and local continuation, i.e., between a top-level computation and a subcomputation. A top-level computation yields the result of an entire program. In general, a type-correct subcomputation cannot be abandoned, that is, it is only based on local or constructive reasoning. A top-level computation, on the other hand, can be abandoned and an evaluation of a program can resume at an existing, data value-expecting context. In other words, a computation may use non-local

or classically-founded reasoning at its top-level. The type $A +_L B$ delimits classical reasoning. It formalizes a general schema for isolating the instances of correct applications of classical laws. In CTT, the notion of computation is still in an embryonic stage. Yet, the introduction of CTT is necessary to start off the introduction of actual, classical programs.

The type $A +_L B$ expresses the construction of cut-free proofs of disjunction in the context of the first disjunct. Similarly, a type $A +_R B$ can be defined to express the construction of cut-free proofs of disjunction in the context of the second disjunct. More precisely, one has a cut-free proof of $A \vee B$ constructed from cut-free proofs of A or B in the context A, if

$$\frac{\begin{array}{ccc} X & [A] & [B] \\ A \vee B & A & A \end{array}}{A} \qquad X = A \text{ or } B \qquad (\Pi_A)$$

since Π_A reduces to a normal derivation $\begin{matrix}[X] \\ A\end{matrix}$. Here, the proof of A in the context A is an axiom.

4.4.2 Conjunction Type

In this section, we will formalize classical conjunction in a manner that preserves the constructive evidence at the top-level of proofs. This new type will be constructed from either the left- or right-conjunct type. The definition of the conjunction type requires that the context in which cut-free proofs of conjunction are being constructed is not an arbitrary non-empty data type but the left or the right conjunct. This is the only way we can assure that if the cut-free proof of the first (in sequence) conjunct is obtained, the proof of the second conjunct doesn't escape. The definition of conjunction type and its symbol depend on whether the context in which cut-free proofs of conjunction are being constructed is the left or the right conjunct. If the left conjunct is this initial context, then "$\times_{LR}$" is the symbol of conjunction type. The tag "*LR*" denotes the order of reduction from left-to-right for pairs of proofs. The definition of $\times_{LR}$-type is as follows:

$$\frac{A \in data, \quad B \in data}{A \times_{LR} B \ type} \qquad (\times_{LR}\text{-}form)$$

$$\frac{h \in \Pi a \in A.\Lambda_R(A,B,a,b)}{\pi(h) \in A \times_{LR} B} \qquad (\times_{LR}\text{-}intro)$$

$$\frac{c \in A \times_{LR} B}{c_0 \in A \to A,\ c_1 \in A \to B} \qquad (\times_{LR}\text{-}elim)$$

$$\frac{h \in \Pi a \in A.\Lambda_R(A,B,a,b)}{(\pi(h))_0 \to_1 \lambda a \in A.a,\ (\pi(h))_1 \to_1 \lambda a \in A.snd(ha)} \qquad (\times_{LR}\text{-}red)$$

Conjunction type is the type of the operational content of the top-level, classical proofs of conjunction. The constructive function h in ($\times_{LR}$-*intro*) assures that the computation of a witness of the right-conjunct B in a data value a-expecting context uses only local reasoning. The construction $\pi(h)$ represents a structured, top-level context of the classical conjunction type.

From an operational point of view, the type $A \times_{LR} B$ internalizes the operations of *duplicating* a data value-expecting continuation represented by a and of requesting the normal result of a program of type B which is evaluated in the context a. As we pointed out before, duplication of a context implies preserving a state of computation that can be reset in case an escape occurs. Hence, if the computation in B escapes it will resume at the preserved context a. The operational difference between the type of conjunction and the type of disjunction $A +_L B$ is that the latter allows the computation in B to resume evaluation in an arbitrary data value-expecting context while the former allows it to resume only in the preserved context of its normal evaluation.

From the categorical point of view, a term of type $A \times_{LR} B$ maps an object (a continuation) of type A both to itself and to an object (a continuation) of type B. One seeks an operation $f \in A \to B$ defining only b in terms of a by mapping an object of type A to itself and to an object of type B. The continuation of type A can be applied again, etc., until a continuation of type B is reached. This defines a categorical repetition operation known as *recursion*. As in the case of categorical iteration, categorical recursion by itself doesn't yet compute any value. The difference between iteration defined in the section 4.4.1 and recursion is that the latter preserves the context of type A when the result of type B is requested while the former doesn't.

From the point of view of proof theory, the type $A \times_{LR} B$ expresses the construction of cut-free proofs of conjunction in the context of the first conjunct. More precisely, one has a cut-free proof of $A \wedge B$ constructed from cut-free proofs of A and B in the context A, if

$$\frac{\dfrac{A \quad B}{A \wedge B}}{A} \qquad (\Pi_A)$$

since Π_A reduces to axiom A. The definition of $\times_{LR}$-type assures that the proofs of B are cut-free since the subterm $snd(ha)$ has to reduce to a data value b. The proofs of B are implicit which is expressed by the fact that the term $\lambda a \in A.snd(ha)$ is fully evaluated in CTT.

4.4.3 Existential Witness Type

In this section we will formalize the existential witness type in a manner that preserves the constructive evidence at the top-level of proofs. This new type will be constructed from a single existential witness type introduced in the section 4.1. Let '$\{\ \}$' be the symbol of the type in question. Its definition is as follows:

$$\frac{A \in data, \quad B \in data}{\{A\}_B \text{ type}} \qquad (\{\}\text{-}form)$$

$$\frac{h \in B \to \Sigma(A,a)}{\rho(h) \in \{A\}_B} \qquad (\{\}\text{-}intro)$$

$$\frac{d \in \{A\}_B}{d_0 \in B \to A} \qquad (\{\}\text{-}elim)$$

$$\frac{h \in B \to \Sigma(A,a)}{(\rho(h))_0 \to_1 \lambda b \in B.split(hb)} \qquad (\{\}\text{-}red)$$

Existential witness type is the type of computational content of top-level proofs of the existential quantification. The constructive function h assures that the computation of a witness a in a data type B-expecting context uses only local reasoning, i.e., that it is a *normal* computation. This, in turn, assures that proofs of existential quantification are reducible to cut-free forms and the corresponding programs are totally-correct. The type B is distributed over every construction, each handling one of the witnesses $a \in A$. The construction $\rho(h)$

represents a structured, top-level context for classical existential quantification.

$\{A\}_B$ is the type of the evidence of cut-free proofs of the existential quantification carried out in the context of data type B. More precisely, one has a (computational content of) cut-free proof of the existential quantification over A in an atomic context B, if

$$\frac{\begin{array}{cc} A & \begin{array}{c}[A]\\ B\end{array}\end{array}}{B} \qquad (\Pi_B)$$

since Π_B reduces to a normal derivation $\begin{array}{c}[A]\\ B\end{array}$. The definition of $\{\}$-type assures that the proofs of existential quantification are reducible to cut-free forms since the subterm $split(hb)$ has to reduce to an atomic object a. An existential witness a of data-type A is implicit since the expression $\lambda b \in B.split(hb)$ is fully evaluated in CTT.

4.4.4 Classical Absurdity Type

The following universal quantification expresses the operational content of a lack of an existential witness:

$$(NExists\ A) \equiv \forall C{:}prop.((A \Rightarrow Empty) \Rightarrow C) \Rightarrow C$$

where $Empty \equiv \forall X{:}prop.X$. The implication $A \Rightarrow Empty$ is a type of a function that never returns to a point of call, i.e., it escapes. The fact that a continuation-representing function of type $A \Rightarrow Empty$ never returns to place of call implies that there is no meaningful continuation (or context) in which a value of type A can be computed.

We want to assure that a program that escapes preserves its type. In order to assure that escapes in a program of type A preserve typing, we must take A as the context of the program. Hence, the type variable C in $(NExists\ A)$ has to be instanciated to A. The resulting type is the type of call/cc which was shown to be the algorithmic content of double-negation elimination rule (the classical absurdity rule) [Grif90]:

$$\frac{\begin{array}{c}\neg A\\ \Lambda\end{array}}{A} \qquad (\Lambda_C)$$

If C in $(NExists\ A)$ were instanciated to some other data type than A, this would have left

open the possibility for a program of type A to escape to a context in which it might have computed a result of a different type from A. The logical counterpart of such a program is a proof which uses the rule of absurdity elimination $\Lambda \vdash_C B$ since it is the type of arbitrary escapes. It is known that the sentences that do not contain applications of absurdity elimination (are Λ-free) belong to the class Π^0_2.

The proposition (*NExists* A) is the type of a continuation of a program of type A that fails to compute a value in any context. When instanciated to A, it is a type of a continuation of a program of type A that abandons its normal evaluation and requests a new final result of type A.

We will introduce in CTT a type expressing the top-level, operational interpretation of the constant for *absurdity* or falsehood. The absurdity type whose formation rule in CTT is as follows:

$$\nabla \in \mathit{data} \qquad (\nabla\text{-}\mathit{form})$$

formalizes the notion of falsehood in CTT. It represents any proposition whose proofs do not provide evidence in a constructive sense. Its operational interpretation, i.e., its elimination rule, corresponds to abandoning a proof entirely in any context. Its *top-level* operational interpretation corresponds to *abandoning* an entire derivation and *resuming* a proof at the top-level. This interpretation corresponds the computational content of the double-negation elimination rule applied in the context of an entire proof. The top-level operational interpretation of absurdity is characterized by a unique function $\mathit{Resultis}(a) \in (A \to \nabla) \to A$:

$$\frac{A \in \mathit{data}, \quad a \in A}{\mathit{Resultis}(a) \in (A \to \nabla) \to A} \qquad (\nabla^{\mathrm{top}}\text{-}\mathit{elim})$$

The rule $(\nabla^{top} - elim)$ represents the rule of absurdity elimination applied only at the top-level of proofs. The type $(A \to \nabla) \to A$ is the *classical absurdity type.* We note that the type $(A \to \nabla) \to A$ does not have noncanonical terms, i.e., its terms occur only in normal forms. Hence, it is an atomic or a data type in CTT, namely for any $A \in \mathit{data}$, $(A \to \nabla) \to A \in \mathit{data}$. The construct $\mathit{Resultis}(a)$ formalizes abandoning the current path of evaluation in a program of type A and returning a new *final* result requested in $a \in A$. In order to assure classical type-

soundness, namely that reduction preserves typing, the classical absurdity has to preserve the operational semantics of terms. This is expressed by the following one-step reduction rule representing the operational interpretation of *Resultis*:

$$\frac{A \in data,\ \ a \in A,\ \ f \in ((A \to \nabla) \to A) \to A}{f(Resultis(a)) \to_1 a} \qquad (\nabla^{top}\text{-}red)$$

The rules (∇^{top}-*elim*) and (∇^{top}-*red*) formalize the computational content of absurdity at the top-level of proofs. CTT with the rules for classical absurdity is a classical programming logic. In other words, it interprets classical predicate logic as a total-correctness type theory. A type-preserving escape in a program whose normal execution results in a value of type A, is formalized as $Resultis(a) \in (A \to \nabla) \to A$.

We will introduce a programming construct *resultis* to implement *Resultis*, i.e., to implement the operational interpretation of the classical absurdity type. The construct *Resultis* is defined on atomic objects "a" of CTT that represent continuations. The term *resultis*, on the other hand, is defined on (expressions that reduce to) constants "c_a" (e.g., numerals) that stand for explicit data. If a *resultis* expression is ever evaluated, the value of its argument will become the final result of the program. In a program p, whose normal execution results in a value of type A, the type of the argument of any application of *resultis* in p has to be A as well. This is required in order that p be well-typed.

As we have already mentioned in the Introduction, another way of interpreting classical proofs as programs was introduced by Chetan Murthy in his thesis [Murthy90]. Murthy used the double-negation/A-translation to extract program content of classical proofs. He proved that the rule of double-negation elimination is the proof-theoretic form of the (nonlocal control) operator $\boldsymbol{C}$. The difference between the operator $\boldsymbol{C}$ (a relative of Scheme's call/cc) and *resultis* lies in semantical complexity. Let us recall the type of the operator C:

$$((A \to B) \to A) \to A$$

There are two key features of $\boldsymbol{C}$: the *duplication* of the continuation at the point where call/cc is called (which can be a subexpression of a program) and *discarding* of the current continuation. The construct *resultis* doesn't have the duplication aspect since we always

restart with the initial continuation. That is, *resultis* always returns its argument as a final result of a program. Its introduction is necessary to start things off. First, we have to define classical program fragments whose execution can be abandoned without destroying well-typeness. Only having defined the correct programming schemas, the scope of *resultis* can be limited so it can specify the result of a subexpression and not necessarily the whole program. This will introduce a language with a block structure. The delimited version of *resultis* could be obtain with two constructs:

block b is e

resultof b is e

Here *b* is a continuation of a block (a "label"). The expression in a block is evaluated normally. However, inside that expression we can use the construct "*resultof b is e*" to abandon execution of the block expression and return alternate result. That is, after we encounter a *resultof*, we want to reactivate the continuation of the block. But first, we have to know how such a block expression is formed without destroying the well-typeness of the program, which is an endevour taken by the thesis.

4.5 Classical Logic as Specification Logic

Let us summarize the preceding sections of this chapter. We are only concerned with the computationally relevant parts of classical proofs. We already know that conjunction, disjunction, existential witness, and lack of existential witness are definable in the intuitionistic second-order propositional logic. Since absurdity is also definable by the second-order propositional quantification, there is no distinction between minimal and intuitionistic second-order logic. These second-order definitions *encode* operational semantics of conjunction, disjunction, and existential quantification, which is in agreement with their intuitionistic interpretation but equally applicable to classical logic. CTT results from the top-level analysis of the second-order encodings of logical connectives. It is a formalization of the top-level operational interpretation of classical logic. It formalizes (the computational extracts of) those classical proofs that provide evidence for formulas in a constructive sense.

Friedman A-translation establishes the following provability result (*cf* Introduction):

$$\text{if } \Gamma \vdash_I \Phi \text{ then } \Gamma^A \vdash_I \Phi^A$$

where X^A is A-translation of X and $\vdash_I$ denotes provability in intuitionistic logic. As we have seen in the previous section, the top-level operational interpretation of absurdity is definable in CTT and corresponds to A-translation of double-negation elimination rule. A-translation is safely applicable only to formulas that do not contain applications of the absurdity elimination rule. This implies that only the proofs of formulas that do not contain applications of absurdity elimination rule are representable in CTT. Such formulas belong to class Σ^0_1. Below, we will show that CTT expresses the operational interpretation of Σ^0_1 sentences:

Theorem 1 (CTT **Interpretation of Σ^0_1 Sentences**) If we have a proof $\vdash_C \phi$, where $\vdash_C$ denotes provability in classical logic, and ϕ is Σ^0_1, and if we have a decision procedure for propositional sentences, then we can express the operational interpretation of the proof of ϕ in CTT.

Proof: Every Σ^0_1 sentence can be written in the form $\phi \equiv \exists y \in B.P(y)$, where P is a computable predicate and B is a data type. There are two possible cases, depending on whether a proof of ϕ is direct or by a counter-example. If a proof is direct, then the operational interpretation of a classical proof of ϕ is represented in CTT by a term T of type $\{A\}_B$. According to (*{ }-intro*), the term T is definitionally equal to $\rho(h)$, where $h \in B \to \Sigma(A,a)$. According to (*{ }-red*), the following one step reduction is the operational content of the construct $\rho(h)$:

$$(\rho(h))_0 \to_1 \lambda x \in B.split(hx)$$

where $\lambda x \in B.split(hx) \in B \to A$. Provided $b \in B$ that has property P, the type $\{A\}_B$ requires that $(\lambda x \in B.split(hx))\,b$ evaluates to a.

If a proof of ϕ is by a counter-example, then the top-level operational interpretation of a classical proof of ϕ is represented in CTT by a term T of type $\{A\}_{(A\to\nabla)\to A}$. According to (*{ }-intro*), T is definitionally equal to $\rho(h)$ where $h \in ((A\to\nabla)\to A)\to\Sigma(A,a)$. According to (*{ }-red*),

$$(\rho(h))_0 \rightarrow_1 \lambda g \in (A \rightarrow \nabla) \rightarrow A.split(hg)$$

where $\lambda g \in (A \rightarrow \nabla) \rightarrow A.split(hg) \in ((A \rightarrow \nabla) \rightarrow \nabla) \rightarrow A$. The type $\{A\}_{(A \rightarrow \nabla) \rightarrow A}$ formalizes the case when $b \in B$ does not have the property P and requires that $(\lambda g \in (A \rightarrow \nabla) \rightarrow A.split(hg))Resultis(a_r)$ reduces to a new value $a_r \in A$. □

The similar theorem can be stated for the Π_2^0 sentences and it is as a corollary of the previous theorem, where free variables on atomic types are allowed:

Corollary 1 (CTT Interpretation of Π_2^0 Sentences) If we have a proof $\vdash_C \phi$, where $\vdash_C$ denotes provability in classical logic, and ϕ is Π_2^0, and if we have a decision procedure for propositional sentences, then we can express the operational interpretation of the proof of ϕ in CTT.

For the systems, where the type symbols and the terms are generated separately from each other (e.g., Gödel's theory **T**, Girard's system **F**), the method used in the proof of strong normalization consists of two stages. First, the abstract notion of *reducibility* or of *reducibility candidate* is defined by induction on the construction of a type symbol, and, second, the reducibility of a term is proved by induction on its construction. More precisely, the strong normalization proof for simply typed λ-calculus (which easily extends to strong normalization proof for Gödel's theory **T** of primitive recursive functionals of finite types) uses the method of reducibility due to [Tait67]. Reducibility is an *abstract* notion used to formulate a strong induction hypothesis to make the proof work. A method to prove strong normalization of an impredicative system uses even more abstract notion, *reducibility* candidate, which is an extension of Tait's method. This proof was discovered by Girard [Girard70] to prove strong normalization of system **F**. In CTT, however, the definition of the notion of reducibility and the proof that an arbitrary term is reducible can no longer be separated because the type symbols and the terms are generated simultaneously. If a term t converts (reduces in one step) to t', then t is called the *redex* and t' the *contractum*. The types in CTT are defined in such a way that the contractum of the redex of a particular type is a part of the formation of that type. Moreover, such redeces are always objects of atomic types whose normalization is immediate. In other words, CTT is constructed in such a way that its

terms are reducible to normal forms by their definitions. The strong normalization theorem for CTT is stated and proved as follows:

Theorem 2 (Strong Normalization of CTT) Every derivation in CTT reduces to a unique normal form.

Proof: Each of Σ, $\wedge_i$, and $\vee_i$ (where $i=L,R$) is tagged with an atomic object(s) to which its noncanonical objects reduce in one-step of reduction. That is, the normalization and uniqueness of terms of those types are parts of their definitions. In particular, the term $split([a])$ is strongly normalizable since it converts to an atomic object a. There is no other possibility: $split([u])$ cannot convert to $split([u'])$ with u' one step from u since there are no introduction, elimination, or reduction rules for data types in CTT. For the similar reasons, the only possible conversions for redeces of left- and right-conjunct types and left- and right-disjunct types are as follows:

$fst(<a,\#>) \rightarrow_1 a$

$snd(<\#,b>) \rightarrow_1 b$

$outl(inl(a)) \rightarrow_1 a$

$outr(inr(b)) \rightarrow_1 b$

The types $\{\ \}$, $+_i$, and $\times_{ij}$ (where $i,j=L,R$ and $i \neq j$) are formed of the particular instances of Π-type in such a way that the antecedent is always a data type, say D, and consequent is always one of the Σ, $\vee_i$, or $\wedge_i$ types, respectively. Except the type of classical absurdity, these are the *only* instances of a Π-type used in CTT derivations. A noncanonical term t of any of the types formed from $\{\ \}$, $+_i$, and $\times_{ij}$ is reducible in one-step of reduction to an abstraction(s) t' such that t' is one of the following:

1. An identity $\lambda x \in D.x$.

2. A term $\lambda x \in D.outr(fx)$ or a term $\lambda x \in D.outl(gx)$, where f and g are variables of types $\Pi x \in D.\vee_R(D,A,a)$ and $\Pi x \in D.\vee_L(A,D,a)$, respectively.

3. A term $\lambda x \in D.snd(gx)$ or a term $\lambda x \in D.fst(fx)$, where g and f are variables of types $\Pi x \in D.\wedge_R(D,A,x,a)$ and $\Pi x \in D.\wedge_L(A,D,a,x)$, respectively.

Terms fx and gx above are in normal forms since they are not redeces: f and g are not of the form $\lambda x \in D.v$. Similarly, terms $outr(fx)$, $outl(gx)$, $snd(gx)$, and $fst(fx)$ are in normal forms since they are not redeces: fx is not of the form $inr(u)$ or $<u,\mathit{it}>$ and gx is not of the form $inl(v)$ or $<\mathit{it},v>$. In other words, t' is in normal form. There are no other possibilities since the term $<f,g>^i$ (where $i=0,1$) cannot convert to $<f',g>^i$, with f' one step from f (or $<f,g'>^i$, with g' one step from g). Such a conversion would have implied that f, for instance, is a redex of the from $(\lambda y \in E.\lambda x \in D.u)e$ with e a member of some data type E and $u \in \vee_j(D,A,a)$ (where $j=L,R$). But there are no rules for constructing higher-order functional terms (except the absurdity type) in CTT. For the similar reasons, the only possible conversions for redeces of conjunction and existential witness types are those listed above.

The only other functional type used in CTT derivations whose normalization has to be shown, is the classical absurdity type. We note that the type $(D\rightarrow\nabla)\rightarrow D$ does not have noncanonical terms, i.e., its terms occur only in normal forms. The terms of the antecedent are constant functions $Resultis(d)$, where d is a data value of type D. A redex t associated with the classical absurdity type is of the type $((D\rightarrow\nabla)\rightarrow D)\rightarrow D$. A term t is defined in such a way that for any $d \in D$, $(t\ Resultis(d))$ converts to d. There are no other possibilities, for $(t\ Resultis(d))$ cannot convert to $(t'\ Resultis(d))$. Such a conversion would have implied that $t \equiv (\lambda y \in E.u)e$ with e a member of some data type E and $u \in ((D\rightarrow\nabla)\rightarrow D)\rightarrow D$. However, there are no rules for constructing higher-order functional terms in CTT other than the absurdity type itself. □

CHAPTER 5

CLASSICAL THEORIES AS PROGRAMMING LOGICS

In Chapter 4, we have introduced the classical types of disjunction, conjunction, existential quantification, and absurdity. These types constitute the purely logical part of any classical type theory. In this chapter we will show how to extend CTT to a first-order theory. We will extend CTT to first-order theories with natural numbers, booleans, and binary trees. CTT extended to a first-order theory of natural numbers, CTT+Nat, for instance, is a programming logic for classical arithmetic. It provides top-level operational semantics of numerical functions. CTT+Nat is a programming logic for Peano Arithmetic in the same manner as Martin-Löf's type theories are programming logics for Heyting Arithmetic[1] and other constructive reasoning systems. In fact, CTT+Nat *interprets* Peano Arithmetic.

The purpose of this thesis is to find computations that can be safely abandoned and resumed in a total-correctness, type-theoretic framework. In other words, we are looking for programs for which resetting an existing context or "escaping at the top-level" is type-correct. This "safety check" is done in a total-correctness framework provided by CTT. In fact, CTT formalizes a classically-founded computation in an embryonic stage.

Before, we proceed further to derive concrete classical programming schemas, we need to clarify the meaning given to the word 'escape' in this thesis. More precisely, in the thesis the word 'escape' is used as a generic word for resetting an existing top-level context rather than

1. Constructive or Heyting Arithmetic (HA) [DT89] is essentially Peano Arithmetic (PA), without an axiom of excluded middle.

for abandoning a computation entirely. The latter is the usual meaning given to the word 'escape' in the literature on programming languages and continuation semantics. We want to sharply distinguish our general meaning of the word 'escape' from the other commonly used meaning. Also, in most of the work on continuations, a *top-level context* (i.e., the context of an entire program) cannot be analyzed. In other words, it is assumed to be expecting always an atomic value. This thesis demonstrates that a top-level context can be structured. The most refined, structured top-level context is that of the existential witness type since it is a context of the most basic, entire computation. In this context, a program may either evaluate normally or may abandon its normal evaluation and resume its computation at the existing, data value-preserving context. We will refer to the resumed computation as an "escaped" computation. An "escaped" computation is of the classical absurdity type introduced and it is implemented by the *resultis* construct introduced in Chapter 4. Hence, the top-level context of the existential witness type has two components. One component represent the context of a normal (i.e., local) evaluation of a program, the other represents an alternative top-level context which is the context of the "escaped" computation. In other words, both contexts are available but only one of them will be used. It is an exact analogue of the computer instruction "IF ... THEN ... ELSE ..." where the parts "THEN ..." and "ELSE ..." are both available but only one of them will be executed. In fact, we will demonstrate in this chapter that *conditional* (a control structure associated with a two-element type) implements the operational semantics of the classical disjunction type.

We emphasize that in this thesis a phrase "an escape at the top-level of a program" means continuing a computation by resuming in an existing, data value-expecting context. In other words, a normal evaluation of a program is abandoned at its top-level and the program resumes its evaluation in the existing top-level context which is the context of an "escaped" computation. In contrast, the commonly given meaning to this phrase in the literature is that a computation is abandoned and concluded.

The type $A +_L B$ introduced in Chapter 4 formalizes a general schema for isolating the instances of correct applications of classical laws. In this chapter we will introduce several classically-founded, local control operators. By choosing a data-value expecting context A in

the type $A +_L A$ to be a particular ground type, classically-founded local control operators can be defined. We will also define local control operators associated with types of identity, booleans, natural numbers, and binary trees. Similarly, we will introduce classically-founded recursion operators by specializing a data-value expecting context in classical conjunction type $A \times_{LR} B$ to natural number and binary tree types.

The recursion operators will be implemented by classical program schemas. We will define totally-correct classical program schemas associated with natural numbers and binary trees. These are primitive recursion computation schema, terminating general recursion computation schema, and binary tree primitive recursion computation schema. We will identify Π^0_2 as the class of sentences of Peano Arithmetic (PA) whose classical proofs provide evidence in a constructive sense, and we will show how to extend this result to other theories.

We will prove the strong normalization of CTT+Nat. We will show CTT+Nat formalizes the operational interpretation the arithmetical class Π^0_2.

Constructive type systems *identify* proofs of conjunction, disjunction and existential quantification with algorithms computing pairs of values, injections of values, and ground values, respectively. This identification dictates that one must verify that a (functional) program terminates through type-checking, and hence many terminating functions may be "missed". Checking termination is usually the hardest part of program verification and should be distinguished from others. We will demonstrate that by separating proofs from programs, it is possible to express in CTT+Nat all functions provably total in PA.

The system F *identifies* data objects with their operational interpretation. The price paid for identifying data types with control structures is that there is no clear distinction between different computation rules. In this chapter, we will separate data objects from control structures for several classes of objects. Such a separation is accomplished by forgetting the internal structure of a derivation of a data type in F and preserving only its intuitionistic interpretation. In this way an *atomic* type is obtained. An extension of CTT with an atomic type A and with the rules for reasoning about its objects corresponds to the operational interpretation of a classical theory of objects of this type. Such an extension is a classical type

theory CTT+A. In this chapter, we will extend CTT to several type theories interpreting classical, first-order theories of booleans, natural numbers, and binary trees.

The *schemata* for introducing basic types is as follows: let $\vdash M:T$ be provable in the system F and let a normal term M correspond to a data object of a particular data-type T. Forgetting the derivation of T and M in F is expressed by *naming* the constructions M and T. If ' ' is a map from terms in the system F to strings, then let $"m"='M'$ and $"t"='T'$. Introducing constants a and A to CTT+A, such that

$$a \equiv "m"$$

$$A \equiv "t"$$

yields a fundamental form of a *judgement* (a "naturally" correct mathematical assertion) in CTT+A:

$$a \in A$$

namely, that an object a is of type A. The epsilon ($\in$) notation represents the relation between an object and its type.

5.1 Identity

As we recall from Chapter 4, we have used the unit type *Id* to define top-level operational semantics of propositional connectives. We have used this atomic type to represent the truth in CTT. In this section we will formalize the computational content of a theory associated with the unit, i.e., we will introduce a control structure associated with the unit type. The identity type has the following impredicative construction:

$$self \equiv [X:prop][x:X]x : id$$

where

$$id \equiv [X:prop]X \Rightarrow X.$$

The data type "identity" can be distinguished from the algorithm associated with it by forgetting its derivation in F:

$$Id \equiv 'id'$$

$$t \equiv 'self'$$

In Chapter 4, we have introduced the formation and the introduction rules for *Id*. A control structure associated with the type *Id* is a construction of the computational content of a classical axiom, i.e., of a cut-free proof of an arbitrary formula with that formula as hypothesis. The proof can be direct or by contradiction. The control structure in question yields a classical program that given a data value a it evaluates to a. Yet, the normal evaluation of the program can be abandoned at its top-level and the program will resume its ("escaped") computation in the existing, data value-expecting context. The abandoning the normal computation at the top-level corresponds to abandoning the *request* for its normal result. In other words, if the identity program "escapes" it "escapes" always at the context of its normal evaluation. As we have already pointed out in the beginning of this chapter, the meaning given to the word 'escape' in this thesis is more general than the commonly used meaning. By 'escaping' we mean resuming computation at the existing, data value-expecting context.

We want to define a constructive translation, i.e., the operational semantics of a classical identity proof. In order that a request for a normal computation be safely abandoned in a total-correctness framework, the top-level context of the normal computation has to be *duplicated*. Duplication of a context implies preserving a state of computation that can be reset in case an escape occurs. We will refer to the preserved context as being *accessible for escaping*. As we recall from Chapter 4, the specialized disjunction type $B +_L A$ formalizes the notions of *duplicating* a context as well as of requesting the normal result of a program. Since a classical identity proof corresponds to an arbitrary computation, we can represent its top-level context by an object of the unit type *Id*. The unit type interprets the truth in CTT, i.e., it represents any non-empty classical type. Its unique object tt can represent a top-level context of an arbitrary computation. Hence, we can define the operational translation of classical identity proof by choosing the type B in $B +_L A^2$ to be *Id*. The resulting type, $Id + A$ (we don't need to the use the subscript L) will allow us to duplicate the top-level context of an arbitrary computation as well as request its normal result.

According to ($+_L$-*elim*), the rule of type $Id \rightarrow A$ is defined from $d \in Id + A$. By ($+_L - intro$), $d = <f,g>$ where

$$f \in \Pi i \in Id.\vee_L(Id,A,i)$$

$$g \in Id \rightarrow \vee_R(Id,A,a)$$

According to ($+_L - elim$), in order for the proofs of disjunction to be reducible to a cut-free from, the following one-step reductions have to take place:

$$<f,g>^0 \rightarrow_1 \lambda i \in Id.i,$$

$$<f,g>^1 \rightarrow_1 \lambda i \in Id.outr(gi)$$

When the assumption $[i \in Id]$ is discharged by taking $i = \mathit{tt}$, two things happen: tt is duplicated and the following 1-step reduction takes place

$$outr(g\,\mathit{tt}) \rightarrow_1 a$$

Any value requested in $a \in A$ is computed normally. However, the duplication of tt preserves the context of a normal evaluation (i.e., a top-level context) of an arbitrary program. In other words, tt is the name of the context in which *any* computation in a data value-expecting context can resume evaluation after abandoning the request for the normal result. Thus, for any data type A, we can represent $a \in A$ as being the request for normal result of a program through a unique $h \in Id \rightarrow A$, such that:

$$h\,\mathit{tt} \rightarrow_1 a$$

We can introduce the following local control operator into CTT+Id:

$$\frac{A \in data,\ a \in A}{\{a\}_A \in Id \rightarrow A} \qquad \textbf{(Id)}$$

The rule of identity $\{a\}_A$ is defined by the following rule of one-step reduction:

$$\frac{A \in data,\ a \in A}{\{a\}_A\,\mathit{tt} \rightarrow_1 a} \qquad \textbf{(Id-}\textit{red}\textbf{)}$$

---

2. Equivalently, we could take A to be Id in $B +_R A$.

In the system **F** all data are functions, i.e., (universal) λ-abstractions. In CTT extended with a data domain *Id* there is a distinction between a data type *Id* and a control structure associated with that type, namely the rule **Id**. In other words, in CTT extended with atomic types, the notion of computation, which is obscured in the system **F**, becomes clearer and its top-level context "accessible for escaping."

5.2 Booleans

In this section we formalize the computational content of a classical theory associated with the "boolean" reasoning, i.e., we will introduce a control structure called the *conditional*. We will give a predicative definition of conditional whose impredicative construction was the second-order quantification *bool*:

$$bool \equiv [X{:}prop]X \Rightarrow X \Rightarrow X$$

The data type "boolean" and the algorithm associated with it, i.e, a conditional, are separated by forgetting its derivation in **F**:

$$Bool \equiv \text{`}bool\text{'}$$
$$true \equiv \text{`}T\text{'}$$
$$false \equiv \text{`}F\text{'}$$

The following judgements are introduced in CTT+Bool, i.e., CTT extended to the theory of booleans:

$$Bool \in data \qquad (Bool\text{-}form)$$

$$true \in Bool \qquad (Bool\text{-}intro1)$$

$$false \in Bool \qquad (Bool\text{-}intro2)$$

A control structure associated with the type *Bool* is a construction of either a cut-free proof of one arbitrary formula with that formula being a hypothesis or a cut-free proof of another arbitrary formula, again with itself as hypothesis. Such a proof can be either direct or by contradiction. It corresponds to a classical program that evaluates to either of two values v_1 or v_1. Yet, the normal evaluation of either v_1 or v_2 can be abandoned at their top-levels and the evaluation can resume its computation in an existing, data value-expecting context. This

context is either the top-level context of the computation resulting in v_a or the top-level context of the computation resulting in v_2. In other words, we will construct a conditional which may have non-local, type-correct jumps.

We want to define an operational interpretation of a classical conditional proof. To assure that if the computation requested in a or b escapes it escapes always at the top-level, we have to preserve the context of the normal computation requested in a or the context of the normal computation requested in b. Since a classical conditional proof corresponds to a sum of two arbitrary computations, we can represent its top-level contexts by objects of type *Bool*. The "boolean" type has two distinct, unique objects which can represent the top-level contexts of two disjoint arbitrary computations. We can define the operational interpretation of conditional proof by taking the type B in $B +_L A$[3] to be *Bool* as this type will allow us to duplicate the top-level context as well as request a normal result of a program, whether the result is a value requested in a or a value requested in b.

According to ($+_L$-*elim*), the rule of type $Bool \rightarrow A$ is defined from $d \in Bool + A$. We know that $d = <f,g>$ where

$$f \in \Pi b \in Bool.\vee_L(Bool,A,b)$$

$$g \in Bool \rightarrow \vee_R(Bool,A,a)$$

According to ($+_L - elim$), the following one-step reductions have to take place:

$$<f,g>^0 \rightarrow_1 \lambda b \in Bool.b,$$

$$<f,g>^1 \rightarrow_1 \lambda b \in Bool.outr(gb)$$

When the assumption $[b \in Bool]$ is discharged by taking b to be either *true* or *false*, two things happen: b is duplicated and the following 1-step reductions take place

$$outr(g\ true) \rightarrow_1 a_1$$

$$outr(g\ false) \rightarrow_1 a_2,$$

---

3. See footnote 2 in this chapter.

that is, $a = a_1$ for $b = \mathit{false}$ and $a = a_2$ for $b = \mathit{true}$. When the normal evaluation of a program results in a value requested in $a_1 \in A$, the duplication of its top-level context "*true*" implies that this context is preserved. That is, if the request a_1 for the normal result is abandoned, the program will resume its evaluation at the context "*true*." Similarly, the duplication of the context "*false*" of a normal computation whose result is requested in a_2 will allow to reset this top-level context in case an escape occurs. In other words, there are two possible top-level computations but only one will be carried out. Correspondingly, there are two top-level contexts available but only one will be used. Thus, for any data type A, we can represent either $a_1 \in A$ or $a_2 \in A$ as requesting the final result of a program through a unique $h \in \mathit{Bool} \rightarrow A$, such that :

$$h\ \mathit{true} \rightarrow_1 a_1$$

$$h\ \mathit{false} \rightarrow_1 a_2$$

We introduce a new control operator [,] into CTT+Bool associated with "boolean" reasoning:

$$\frac{A \in \mathit{data},\ a_1 \in A,\ a_2 \in A}{[a_1, a_2]_A \in \mathit{Bool} \rightarrow A} \qquad \textbf{(Cond)}$$

We refer to the control operator [,] as *conditional*. In fact, the conditional implements the top-level, operational interpretation of classical disjunction type. The conditional is defined by the following one-step reductions:

$$\frac{A \in \mathit{data},\ a_1 \in A,\ a_2 \in A}{[a_1, a_2]_A\, \mathit{true} \rightarrow_1 a_1} \qquad \textbf{(Cond-}\mathit{red1}\textbf{)}$$

$$\frac{A \in \mathit{data},\ a_1 \in A,\ a_2 \in A}{[a_1, a_2]_A\, \mathit{false} \rightarrow_1 a_2} \qquad \textbf{(Cond-}\mathit{red2}\textbf{)}$$

5.3 Natural Numbers

In this section we will introduce a type theory, CTT+Nat, that formalizes the computational content of classical arithmetic. We will define a control structure associated with arithmetical reasoning (natural iteration). We will introduce the primitive recursion operator and

terminating general recursion operator. All these constructions will be defined in a manner applicable to classical reasoning. Finally, we will implement primitive and general recursion operators by programming schemas with a clear and direct operational semantics. These schemas will provide a top-level, operational semantics of numerical, primitive and general recursive functions. We begin with the introduction of natural numbers to CTT+Nat.

The second-order quantification

$$nat \equiv [X{:}prop](X \Rightarrow X) \Rightarrow X \Rightarrow X$$

is an impredicative construction of type of natural numbers. It confuses the data type of natural numbers with iteration. We shall introduce the predicative construction of numbers:

$$Nat \equiv \text{`}nat\text{'}$$

We introduce the following rule of formation for natural numbers to CTT+Nat:

$$Nat \in data \qquad (Nat\text{-}form)$$

The constants k_n will be introduced in CTT extended with natural numbers to represent non-negative integers n:

$$k_n \equiv \text{`}\Lambda X{:}prop.\lambda z{:}X.\lambda s{:}X \Rightarrow X.s(s(s...(sz)...))\text{'}$$

with n applications of s. The integer $0 \equiv$ '*zero*' is the only number equal to the constant k_0 by definition.

The methods for defining totally-correct numerical functions are definitions by *induction*. These definitions are based on an inductive definition of the class of natural numbers, i.e., on a class of objects generated from one primitive object 0 by means of one primitive operation "successor" or "+1". Such a definition of natural numbers is in agreement with their intuitionistic semantics. We have to preserve this inductive interpretation of natural numbers in order to formalize their operational interpretation. The successor function is represented in F by the following construction:

$$succ \equiv \lambda n{:}[X{:}prop](X \Rightarrow X) \Rightarrow X \Rightarrow X.\Lambda Y{:}prop.\lambda s{:}Y \Rightarrow Y.\lambda z{:}Y.s(nYsz)$$

of type $nat \Rightarrow nat$. The term *succ*, the natural iterator, when applied to a (representation of) natural number n, reduces to a universal abstraction representing the number $n+1$. The successor in CTT+Nat is the function constant S such that $S(k_n) \equiv$ '*succ N*', where $N \in nat$

represents n in **F**. Since (*succ N*) *red M*, where $M \in nat$ represents the number $n+1$ in **F**, the constant S is defined in CTT+Nat by the following one-step reduction:

$$S(k_n) \rightarrow_1 k_{n+1} \qquad (S\text{-}red)$$

5.3.1 Natural Iteration

A control structure associated with the type *Nat* is an abstract construction of a cut-free proof of an arbitrary formula obtained after a finite number of steps of reduction. The proof can be direct or by contradiction. It corresponds to an arbitrary computation that evaluates to a data value after some finite number n of steps of computation. Yet, the normal evaluation of the program can be abandoned at its top-level and the program will resume its computation in the existing, data value-expecting context. Such an escape is well-typed if the top-level context is preserved.

We want to define an abstract, operational interpretation of a classical, arithmetical proof. We want to allow an n-step computation to "escape" at a data value-expecting context of its normal evaluation. In other words, a request for the normal result of an n-step computation can be abandoned and the program can resume its computation in the existing, data value-expecting context. To accomplish this, we have to duplicate this top-level context so it can be reset when an escape occurs. Since a classical proof of an arbitrary formula whose cut-free form is reached in an n-step reduction corresponds to an arbitrary n-step computation, we can represent its top-level context by an object $n \in Nat$. The only distinction among arbitrary, numerical computations is the number of evaluation steps needed to reach the top-level of a computation. We will construct the top-level operational interpretation of an arithmetical proof by taking the type B in $B +_L A^4$ to be *Nat*. The resulting type $Nat + A$ will allow us to duplicate the top-level context as well as request a normal result of a program obtained after a finite number of computation steps.

According to ($+_L$ *-elim*), the rule of type $Nat \rightarrow A$ is defined from $d \in Nat + A$. We know that $d = \langle f, g \rangle$ where

$$f \in \Pi n \in Nat. \vee_L(Nat, A, n)$$

$$g \in \in Nat \rightarrow \vee_R(Nat, A, a)$$

According to ($+_L - elim$), in order for the proofs of disjunction to be reducible to cut-free froms, the following one-step reductions have to take place:

$$\langle f, g \rangle^0 \rightarrow_1 \lambda n \in Nat.n$$

$$\langle f, g \rangle^1 \rightarrow_1 \lambda n \in Nat.outr(gn)$$

When the assumption $[n \in Nat]$ is discharged by a natural number N, two things happen: N is duplicated and the following N-step reduction takes place

$$outr(gN) \rightarrow_N a$$

Every value requested in $a \in A$, obtained after some finite number N of computation steps, is computed normally. However, the duplication of the context N of the normal evaluation implies that the context N is preserved. In other words, each N is the name of a top-level context at which an arbitrary N-step evaluation may escape in a well-typed manner. Thus, for any data type A, we can represent an $a \in A$ as being a request for a normal result of a finite number of steps computation through a unique $h \in Nat \rightarrow A$ such that :

$$h0 \rightarrow_1 a_0, \qquad h(n+1) \rightarrow_1 k(h(n)),$$

assuming the constant $a_0 \in A$ and the function constant $k \in A \rightarrow A$. We shall introduce a new local control operator I, the *natural iterator*, into CTT+Nat:

$$\frac{A \in data, \quad a_0 \in A, \quad k \in A \rightarrow A}{Ia_0k \in Nat \rightarrow A} \qquad \textbf{(Iter)}$$

The natural iterator I is defined by the following rules of one-step reduction:

---

4. See footnote 2 in this chapter.

$$\frac{A \in data,\ a_0 \in A,\ k \in A \to A,\ n \in Nat}{IAa_0k(n+1) \to_1 k(IAa_0kn)} \qquad \text{(Iter-}red1\text{)}$$

$$\frac{A \in data,\ a_0 \in A,\ k \in A \to A}{IAa_0k0 \to_1 a_0} \qquad \text{(Iter-}red2\text{)}$$

The normal form of the term IAa_0km is reached in m steps of reduction, namely

$$IAa_0km \to_m kk \cdots ka_0$$

with a finite number m of applications of k. The function constant k is a constructive function since it formalizes the request for a normal (i.e., local) computation. In other words, k represents the operational content of a part of an arbitrary arithmetical proof where classical rules are disallowed. That is, we do not have access to the internal structure of k. Hence, the *contractum* $kk \cdots ka_0$ (with m applications of k) is the normal form of the redex IAa_0km.

The construction (IAa_0kn) is operationally equal to the following "for-loop":

$$z := a_0 \text{ ; for } i=0 \text{ to } n-1 \text{ do } z := kz.$$

5.3.2 Primitive Recursion Operator

A method for defining totally-correct number-theoretic functions corresponds to a *construction schema* of a classical cut-free proof of an arbitrary formula obtained after a finite number of steps of reduction. Such a proof corresponds to a classical program that evaluates to a data value after some finite number n of steps of computation and such that if it abandons its normal evaluation and resumes another, it does so always at the top-level context. For such an "escape" to be type-correct, the top-level context has to be preserved. When the request for the normal result is abandoned, the program will resume its evaluation at the preserved context. The primitive recursion operator will preserve the top-level context as a part of a normal evaluation of a numerical program. In contrast, the duplication of a top-level context of numerical program in not formalized as a part of the definition of the natural iterator.

The methods for defining totally-correct numerical functions are definitions by *induction* (also called *recursive definitions*). These recursive definitions are based on an inductive definition of the class of natural numbers, i.e., on a class of objects generated from one primitive object 0

by means of one primitive operation "+1". Such a definition of natural numbers is in agreement with their intuitionistic semantics. In this section we will derive an elementary inductive method of defining total numeric functions, i.e., *primitive recursion. Primitive recursion operator* will be defined by combining product-based operator, i.e., recursion with the intuitionistic interpretation of natural numbers.

We want to formalize in CTT+Nat the operational interpretation of an elementary classical arithmetical proof method. We want to formalize as a part of a request for the normal result of a numerical program the duplication of the context of its normal evaluation. We can accomplish this by taking type B in $B \times_{LR} A$[5] to be *Nat* as this type will preserve a top-level context of a numerical program as a part of its normal computation.

According to ($\times_{LR}$-*elim*), the rule of type $Nat \rightarrow A$ is defined from $d \in Nat \times A$. We know that $d = \pi(h)$ where

$$h \in \Pi n \in Nat.\wedge_A(Nat,A,n,a)$$

According to the $\times_{LR} - elim$, in order for the proofs of conjunction to be reducible to cut-free froms, the following one-step reductions have to take place:

$$\pi(h)_0 \rightarrow_1 \lambda n \in Nat.n$$

$$\pi(h)_1 \rightarrow_1 \lambda n \in Nat.snd(hn)$$

When the assumption $[n \in Nat]$ is discharged by a natural number N, two things happen: N is duplicated and made accessible to the normal evaluation process, and the following N-step reduction takes place

$$snd(hn) \rightarrow_N a$$

Every value requested in $a \in A$, obtained after some finite number N of computation steps, is computed normally. A "copy" of a top-level context N is preserved in conjunction with a

5. This is assuming left-to-right order of evaluation. Similarly, we could take $A = Nat$ in $B \times_{RL} A$ when right-to-left evaluation strategy is employed.

request for a normally computed value. Hence, the preserved context is internalized as a part of a normal (i.e., local) computation. Thus, for any data type A, we can represent an $a \in A$ as being a request for a normal result of the entire n-step computation through a unique evaluation rule $k \in Nat \to A$ such that :

$$k0 \to_1 a_0, \qquad k(n+1) \to_1 fn(kn).$$

We will introduce a new local control operator R, the *primitive recursion operator*, into CTT+Nat:

$$\frac{A \in data,\ a_0 \in A,\ f \in Nat \to A \to A}{RAa_0f \in Nat \to A} \qquad \textbf{(Rec)}$$

The operational semantics of primitive recursion operator R is defined by the following one-step reduction rules:

$$\frac{A \in data,\ a_0 \in A,\ f \in Nat \to A \to A,\ n \in Nat}{RAa_0f(n+1) \to_1 fn(RAa_0fn)} \qquad \textbf{(Rec-}\textit{red1}\textbf{)}$$

$$\frac{A \in data,\ a_0 \in A,\ f \in Nat \to A \to A}{RAa_0f0 \to_1 a_0} \qquad \textbf{(Rec-}\textit{red2}\textbf{)}$$

The normal form of the term RAa_0fm is reached in m steps of reduction, namely

$$RAa_0fm \to_m fmf(m-1) \cdots f0a_0$$

with a finite number m of applications of f. The function constant $f \in Nat \to A \to A$ is constructive since it formalizes the processing of a request for a normal (i.e., escape-free) computation. In other words, f formalizes the operational content of a part of arithmetical proof where classical rules are disallowed. That is, $Nat \to A \to A$ is treated as an atomic type. The *contractum* $fmf(m-1) \cdots f0a_0$ (with m applications of f) is the normal form of the redex RAa_0fm.

(RAa_0fn) is operationally equal to the following "for-loop" in a programming language:

z:= a sub 0 ; **for** *i=0* **to** *n-1* **do** *z:=f(i,z);*

which, in contrast to iteration, makes an *explicit use* of an *implicit* value of *i*.

5.3.3 Primitive Recursive Computation Schema

A "classical" program that computes to a value of type A is of type the $\{A\}_C$ of the existential witness over A. A value of type A can be just "read off" when a term of the existential witness over A is supplied with a concrete value $c \in C$. In this section we will derive the primitive recursive program schema.

In CTT+Nat, according to $\{\ \}-intro$, a canonical term $d \in \{A\}_{Nat}$ is defined as $\rho(h)$ where

$$h \in Nat \rightarrow \Sigma(A,a)$$

According to $\{\ \}-red$,

$$\rho(h)_0 \rightarrow_1 \lambda n \in Nat.split(hn)$$

When the assumption $[n \in Nat]$ is discharged by a natural number n, the following n-step reduction has to take place

$$split(hn) \rightarrow_n a$$

Every value requested in $a \in A$, obtained after some finite number n of computation steps, is computed normally. Here, however, a top-level context represented by $n \in Nat$ is not preserved as a part attached to a purely local computation. That is, a top-level continuation is no longer treated as an "imperative" add-on to a declarative language as it was implied by the definition of the primitive recursion operator R. It is treated as a central *declarative* concept, not a parenthesis in the language definition. Correspondingly, an application of the classical absurdity rule at the top-level of a proof is not a classical add-on to (a parenthesis in) a constructive system. Such an application has a direct constructive content. More precisely, the type $\{A\}_{Nat}$ requires that an n-step computation of a witness requested in $a \in A$ always returns at its top-level context after n steps of evaluation, provided that the top-level context represented by n is *not* preserved. In order that an n-step computation, whose top-level context is not preserved, is based only on the local reasoning, the computation requested in n *itself* cannot escape. The only way to assure that, is to require that a computation requested in $n \in Nat$ evaluates to an integer since an evaluated computation does not escape. Hence, we will require that terms representing top-level contexts of numerical programs are integer terms, i.e., terms computable to numerals k_i (i=0,1,2,...). We will introduce a special notation n^D for

integer terms. Then, a computation requested in $a \in A$ whose normal evaluation context is represented by n can be expressed as a *top-level, recursive context-typed continuation* (*cf* Introduction). Such a continuation is the operational interpretation of a function that corresponds to an entire, totally-correct program. The following data-valued expression representing the top-level application context of a numerical function is introduced into CTT+Nat to implement R:

$$natrec(n^D; a_0; i^D, y.f(i^D, y)) \qquad (natrec)$$

where $i,y.f(i,y)$ is the notation for syntactical abstraction introduced in Chapter 1. If a_0 is an integer expression, then *natrec* itself becomes an integer expression.

A value of a function space is known as a functional closure. Yet, in order to formalize a classical, total-correctness type theory, we had to abandon the general amalgamation of functions and values. In continuation semantics, a context-typed continuation is the operational semantics of any value of a function space (*cf* Introduction), whether it denotes a total function or not. On the other hand, a *top-level, recursive context-typed continuation* provides the operational interpretation of the classically-founded, *totally-correct* programs. It uses the separation of functions and values to distinguish between the local and the top-level contexts. A local, context-typed continuation is expressed by a syntactical abstraction (i.e., an expression with holes in it) which represents only local reasoning. In other words, a local, context-typed continuation is represented by a fundamental notion of a function. A local, context-typed continuation by itself doesn't yet denote a functional value. An integer term n^D in *natrec* is the name of the top-level (non-local) context of an entire computation. Since *natrec* is assumed here to express the whole program, the result continuation is not specified.

The operational semantics of the normal evaluation of a numerical program can be expressed in terms of the following reduction rules:

$$natrec(0; a_0; x^D, y.f(x^D, y)) \rightarrow_1 a_0 \qquad (natrec\text{-}red1)$$

$$natrec(k_n; a_0; x^D y.f(x^D, y)) \rightarrow_1 f(k_n, natrec(k_{n-1}; a_0; x^D, y.f(x^D, y))) \qquad (natrec\text{-}red2)$$

The expression *natrec* distinguishes between top-level and local contexts in a declarative

manner. Only a top-level numerical computation can be abandoned and an evaluation can resume in an existing, data-value expecting context. For example, the following expression represents a numerical program that either evaluates normally or abandons its normal computation and resumes an "escaped" computation:

$$natrec(case(b;n^D;resultis(a_r));a_0;x^D,y.f(x^D,y))$$

The expression *case* implements the control structure associated with the type *Bool*, namely the conditional [,], which is defined by the following one-step reductions:

$$[a,b]_A\ true \rightarrow_1 a$$
$$[a,b]_A\ false \rightarrow_1 b$$

where $a,b \in A$. The corresponding programming construct *case* is defined on constants c_a representing explicit data rather than on CTT objects a of an atomic type A. Its operational semantics is given in terms of the following one-step reductions:

$$case(true;c_a;c_b) \rightarrow_1 c_a$$
$$case(false;c_a;c_b) \rightarrow_1 c_b$$

If the value of a boolean expression b in $case(b;n^D;resultis(a_r))$ is *false*, a program will abandon a request for its normal evaluation and will request a data value result a_r of an "escaped" computation. As we discussed before, the values of the type *Bool* can represent the top-level contexts of two disjoint *arbitrary* computations. Here, we are dealing with the concrete first-order classical theory, namely Peano Arithmetic and the computations are the primitive recursive numerical programs. The conditional "[,]" is an abstract control operator manipulating on the atomic objects of CTT that represent data value-expecting continuations. On the other hand, the programming construct "*case*" operates on the explicit data. The two possible values of its first argument are here the names of two disjoint top-level contexts of a primitive recursive, numerical program. One value is the name of the context of a normal evaluation of the program and the other value is the name of the context of its "escaped" computation. In fact, the construct "*case*" implements the operational semantics of the classical disjunction type. In other words, the programming construct "*case*" can be used in the implementation of the top-level context of the existential witness. In the case of the natural existential witness, "*case*" is used to implement a structured, top-level context of any primitive

recursive computation that may perform a well-typed jump. We want to point out that in the above example of the *natrec* expression, we have arbitrarily chosen "*true*" as the name of the context of a normal evaluation and "*false*" as the name of the context of an "escaped" evaluation. We might have as well reversed this choice.

The expression schema *natrec* is an example of an explicit programming schema generated by the type $\{A\}_{Nat}$ of existential witness. Yet, by allowing free integer variables, the class of *primitive recursive functions* is obtained as an example of evidence provided by the classical proofs of universal quantifications over natural numbers. Hence, the derived schema *natrec* provides the *top-level, operational semantics* of primitive recursive functions.

5.3.4 Peano Arithmetic as a Programming Logic

In the previous section, by assuming a domain of positive integers we have derived a classical, type-correct programming schema for primitive recursion. We obtained an extension CTT+Nat of CTT which includes in addition to CTT rules, the following rules, constants and expressions: (*Nat-form*), k_n, S, $(S-red)$, (**Iter**), (**Iter**-*red1*), (**Iter**-*red2*), (**Rec**), (**Rec**-*red1*), (**Rec**-*red2*), (*natrec*), (*natrec-red1*), (*natrec-red2*). In this section, we will extend the results proved in Chapter 4 for CTT to CTT+Nat.

CTT+Nat is strongly normalizable as shown by the following theorem:

Theorem 3 (Strong Normalization of CTT+Nat) Every derivation in CTT+Nat reduces to a unique normal form.

Proof: By Theorem 2 we know that CTT is strongly normalizing. We need to show the normalization of the new terms introduced to CTT+Nat. The normalization of numerals is immediate. The normal form of $S(e^D)$ is reached in $n+1$-steps of reduction, where n is the number of steps in which e^D reduces to a numeral k_n. In the previous sections, we have already demonstrated that the terms RAa_0fn and IAa_0fn are normalizable by n-step reductions. The normalization of the term $natrec(e^D; a_0; i^D, y.f(i^D, y))$ can be proved by induction on one-step reduction "$\rightarrow_1$". Assume the following

(1) $e^D \rightarrow_K k_n$

(2) a_0 is reducible to a data-valued constant c_a in M-step reduction

(3) $f(k_i,c_a)$ (where c_a is a data-valued constant for $a \in A$) is reducible in L-steps to a data-valued constant.

For $n=0$,

$$natrec(e^D;a_0;i^D,y.f(i^D,y)) \rightarrow_1 a_0$$

That is, for $n=0$ the normal form of $natrec(e^D;a_0;i^D,y.f(i^D,y))$ is reached in $K+M+1$ steps.

Assume now that $natrec(k_m;c_a;i^D,y.f(i^D,y))$ is normalizable. We want to show that $natrec(k_{m+1};c_a;i^D,y.f(i^D,y))$ is normalizable. By the definition of *natrec*,

$$natrec(k_{m+1};c_a;x^Dy.f(x^D,y)) \rightarrow_1 f(k_{m+1},natrec(k_m;c_a;x^D,y.f(x^D,y)))$$

By induction hypothesis, we know that $natrec(k_m;c_a;i^D,y.f(i^D,y))$ is normalizable, i.e., reducible to a data-valued constant in $K+M+(L*m)$ steps. Then, from the assumption (3),

$$f(k_{m+1},natrec(k_m;c_a;x^D,y.f(x^D,y)))$$

also reduces to a data-valued constant in $K+M+(L*(m+1))$-steps. Thus, we can conclude that for any n, $natrec(k^n;a_0;i^D,y.f(i^D,y))$ reduces to normal form. □

The operational interpretation result for the arithmetical Σ_1^0 sentences is stated and proved as follows:

Theorem 4 (CTT+Nat **Interpretation of Arithmetical Σ_1^0 Sentences**) If we have a proof $\vdash_{PA} \phi$, where $\vdash_{PA}$ denotes provability in Peano Arithmetic, and ϕ is Σ_1^0, then we can represent the operational interpretation of that proof by a term in CTT+Nat.

Proof: We can easily show that Peano Axioms can be interpreted in CTT+Nat. More precisely, we have five Peano Axioms to cover. The following three are immediate:

(1) 0 is interpreted by k_0

(2) successor *Succ* is interpreted by the function constant S defined on integer terms

(3) fifth Peano Axiom

$$(A(0)\wedge\forall n(A(n)\supset A(Succ(n))))\supset\forall nA(n) \qquad \textbf{(Peano5)}$$

where A is any formula of PA, is interpreted by the expression *natrec*. The formula $A(0)$ is interpreted by a data-valued term and $\forall n(A(n)\supset A(Succ(n)))$ is interpreted by a local context of a function application.

What is left is the third and forth Peano Axioms, namely

$$\forall n\forall m(Succ(n) = Succ(m))\supset(n = m) \qquad \textbf{(Peano3)}$$

$$\forall n(Succ(n)\neq 0) \qquad \textbf{(Peano4)}$$

Peano3 is interpreted in CTT+Nat by (*S-red*). More precisely, $S(e^D)$ reduces to a numeral k_{n+1} only if e^D reduces to k_n. Hence, if $S(e_1^D)$ and $S(e_2^D)$ reduce to the same numeral, also e_1^D and e_2^D reduce to the same numeral as well.

Peano4 can be interpreted in CTT+Nat by the following instance of *natrec* expression using the "escaping" construct *resultis*:

$$natrec(resultis(x^D);0;y^D,z.z)$$

There are many programs for Peano Fourth Axiom and the above expression is just one of them. What is common to all programs for **Peano4** is that their normal computation is abandoned and that they resume evaluation at the existing, integer-expecting context. Such a computation corresponds to an *arbitrary* arithmetical proof by contradiction, i.e., to an arithmetical axiom.

Every arithmetical Σ_1^0 sentence can be written in the form $\phi\equiv\exists y\in Nat.f(y)=0$, where f is primitive recursive. There are two possible cases, depending on whether a proof of ϕ is direct or by a counter-example. If a proof is direct, then the operational interpretation of a classical proof of ϕ is represented in CTT+Nat by a term T of type $\{A\}_{Nat}$. According to (*{ }-intro*), the term T is definitionally equal to $\rho(h)$, where $h\in Nat\rightarrow\Sigma(A,a)$. According to (*{ }-red*), the following one step reduction is the operational content of the construct $\rho(h)$:

$$(\rho(h))_0\rightarrow_1\lambda n\in Nat.split(hn)$$

where $\lambda n\in B.split(hn)\in Nat\rightarrow A$. To represent the operational content of the direct proof proof of ϕ, the type $\{A\}_{Nat}$ has to require that $f(n)=0$ only if a is evaluated in n-steps of

computation.

If a proof of ϕ is by a counter-example, then $f(n) \neq 0$ only if a new value $a_r \in A$ is return as a final result of a program.

Hence, the operational interpretation of a proof of an arithmetical Σ_1^0 sentence can be represented in CTT+Nat by the expression schema

$$natrec(case(b;n^D;resultis(a_r));a_0;x^D,y.f(x^D,y))$$

The characteristic function f is represented in CTT+Nat by a boolean-valued expression b. If the value of b in $case(b;n^D;resultis(a_r))$ is *false*, a program will abandon a request for its normal evaluation and will return a_r as a final result of the program. This represents the operational content of a proof of ϕ when it is the proof by contradiction. If the value of b in $case(b;n^D;resultis(a_r))$ is *true*, a program will evaluate normally. This represents the operational content of a direct proof of ϕ. □

By a simple generalization to free variables, the result of the above theorem extends to Π_2^0 sentences. In this way we have identified Π_2^0 as a class of sentences of Peano Arithmetic all of whose proofs provide evidence in a constructive sense.

Corollary 2 (CTT+Nat Interpretation of Arithmetical Π_2^0 Sentences) If we have a proof $\vdash_{PA} \phi$ and ϕ is Π_2^0, then we can represent the operational interpretation of that proof by a term in CTT+Nat.

An arithmetical formula of the class Π_2^0 can be written $\forall x \in Nat.\exists y \in Nat.f(x,y)=0$[6], where

---

6. Or equivalently, it can be written as $\forall \in Nat.\exists y \in Nat.R(x,y)$, such that $f(x,y)$ is a characteristic function of a computable relation $R(x,y)$. We note that the function f can be presented in normal form:
$$f(x,y)=n \equiv \exists u \in Nat.R'(x,y,0,u)$$
where R' is a primitive recursive relation. It follows that by combining the variables y and u into a pair z, Π_2^0 formula can be written as $\forall x \in Nat.\exists z.f'(x,z)=0$ with f' primitive recursive [Grzegorczyk74].

f is a computable function. An expression of this type is a function F with domain *Nat*, such that $F(n)$ computes evidence of type $\exists y \in Nat.f(n,y)=0$. We can assume that F is of the form $\lambda x \in Nat.M$. When F is applied to $n \in Nat$, $F(n)$ has type $\exists y \in Nat.f(n,y)=0$ (a Σ_1^0 sentence), and hence $F(n)$ will compute evidence for this type. The intuitive reason why F has the type $\forall x \in Nat.\exists y \in Nat.f(x,y)=0$ is that when F is applied to a concrete datum, it returns a concrete datum without any embedded function closures or (equivalently) without any unevaluated computations. Every unevaluated computation represents a potential escape in a program.

The "strength" of CTT+Nat, i.e., a class of algorithms which are representable in it, can be easily shown.

Theorem 5 (Representation in CTT+Nat) The functions representable in CTT+Nat are exactly those which are provably total in PA.

Proof: If f is a closed CTT+Nat-term from integers to integers, it gives rise to a function $|f|$ from N to N (where N is a set of integers) by

$$f(k_n) \rightarrow_k k_{|f|(n)}$$

where k is some finite number of reduction steps.

We note that to prove normalization of any fixed term in CTT+Nat, we need to be able to express a finite number of finite number-steps reductions and (in the case when the term is a *natrec* expression) to reason about them by mathematical induction, which can be done in PA. Since the normalization of f is provable in arithmetic, we say that $|f|$ is provably total in PA.

We need to show also the converse: if F is a recursive function, provably total in PA, then there is a term f from integers to integers in CTT+Nat, such that $F(n) = |f|(n)$ for all n. A recursive function f which is provably total from N to N is called provably total in a system of arithmetic, here PA, if PA proves the formula which expresses :"for all n, the program e with input n, terminates and returns an integer" for some algorithm e representing f. The precise formulation depends how we write programs formally in PA. For example, with Kleene notation:

PA proves $\forall n \exists m\, \mathrm{T}_1(e,n,m)$

where $\mathrm{T}_1(e,n,m)$ means that the program e terminates with output m if given input n. $\mathrm{T}_1(e,n,m)$ may itself be expressed as $\exists m'\ P(n,m,m')$ where P is a primitive recursive predicate and m' is the "transcript" of the computation. The two quantifiers $\exists m \exists m'$ can be replaced by a single one $\exists p$ using some (primitive recursive) coding of pairs. Hence, the termination is expressed as Π^0_2 formula. Now, by Theorem 4 (and Corollary 2), we know that Π^0_2 are the formulas whose computational content is expressible in CTT+Nat. Let t be the term that represents the computational content of termination formula. Then, $t(k_n)$ reduces to a pair $(k_m, k_{m'})$ such that $f(n) = m$. □

5.3.5 Terminating General Recursion

The primitive recursion computation schema *natrec* introduced in the section 5.5.5 defines programs for the computable functions that are primitive recursive. A program for a primitive recursive function always terminates. Yet, nontermination can be viewed as a special case of escaping: from the point of call it makes no difference whether the called function is looping forever or it has jumped somewhere else and never returned. Based on this observation, we will introduce and implement in this section the *terminating, general recursion operator.* This operator will be used to express totally-correct, general recursive programs.

An arithmetical method of proof which is not based on the inductive definition of the number class but on the fact that this class expresses a *well-founded* relation, is called *complete* induction. Its computational content is the terminating, general recursion.

In order to formalize the general recursion operator, the new judgement forms are introduced into CTT. In a constructive type theory, a proof of a dependent function type $\Pi x \in A.B(x)$ (interpreting universal quantification) is a function which, when given a value $a \in A$, returns a value in $B[a/x]$. In a classical type theory, i.e., in a classical logic viewed as a typed programming language, a proof of a $\forall$-type is a witness for a lack of counterexamples for the truth of that type, which is a much weaker statement. Hence, one would like to distinguish between the classical, universally quantified propositions and the dependent function type. This can be done by introducing a new form of a judgement to CTT, namely

$$A \text{ prop}$$

to distinguish propositions from data types. To internalize the universal quantification in a type theory, the following hypothetical judgement is used:

$$B(x) \text{ prop } [x \in A]$$

It yields the following judgement for the universally quantified propositions:

$$\forall x \in A.B(x) \text{ prop}$$

A proposition can be judged to be true or false. We will introduce the following new judgement form to CTT+V, i.e., CTT extended with an atomic data type V:

$$R(v_1, v_2) \text{ true},$$

where $v_i \in V$ (i=1,2) and R is a decidable, well-founded relation. The relation R has to be well-founded in order to preserve termination. It has to be decidable in order to be effectively presentable. For example, if $V = Nat$, R is a well-founded relation "less than" ("<"), which is the usual order on natural numbers.

5.3.5.1 Generalized Natural Function Iterator

In this section, we will introduce a new, natural iterator based on a total recursive function g such that $gn < n$, for any $n \in Nat$, rather than on the natural predecessor. Such a function is a *new, generalized* "predecessor" for recursion.

We want to define an abstract operational interpretation of a classical arithmetical proof based on a total recursive function $g \in Nat \rightarrow Nat$ instead of an intuitionistic interpretation of the class of natural numbers. We will allow a gn-step computation (with an $n \in Nat$) to escape or not terminate at the top-level of a program. In order for such an escape to be type-correct, the top-level context has to be preserved so that when the escape takes place the the top-level context can be reset. We can define the interpretation in question by taking the type B in $B +_L A^?$ to be Nat, as this type will allow us to duplicate the top-level context as well as request a normal result of the program obtained after a finite number of computation steps.

According to ($+_L$ *-elim*), the rule of type $Nat \rightarrow A$ is defined from $d \in Nat + A$. We know that $d = \langle g_1, g_2 \rangle$ where

$$g_1 \in \Pi n \in Nat. \vee_L(Nat, A, n)$$

$$g_2 \in Nat \rightarrow \vee_R(Nat, A, a)$$

According to ($+_L - elim$), in order for the proofs of disjunction to be reducible to cut-free forms, the following one-step reductions have to take place:

$$\langle g_1, g_2 \rangle^0 \rightarrow_1 \lambda n \in Nat.n$$

$$\langle g_1, g_2 \rangle^1 \rightarrow_1 \lambda n \in Nat.outr(g_2 n)$$

When the assumption $[n \in Nat]$ is discharged by a natural number M, two things happen: M is duplicated and the following K-step reduction takes place:

$$outr(g_2 M) \rightarrow_K a,$$

where K is the number of applications of g to M that reaches 0. Every value requested in $a \in A$, obtained after some finite number K of computation steps, is computed normally. Hence, in order to preserve total correctness, g has to be such that it allows 0 to be reached after some finite number of applications. In other words, the trace of g has to be a well-founded sequence of natural numbers, i.e., for any $n \in Nat$, $gn < n$. The duplication of M preserves the top-level context of a K-step evaluation. Hence, if an escape occurs or if a termination condition is not satisfied, the top-level context can be reset and the computation can resume its evaluation. Let $N \in Nat$ be a number such that for any $n \leq N$, gn returns 0. That is, numbers from 0 to N constitute an initial (or base) segment which doesn't have a (generalized) "predecessor". Thus, for any data type A, we can represent an $a \in A$ as a request for the normal result of the entire K-step computation, through a unique $h \in Nat \rightarrow A$ such that:

$$hn \rightarrow_1 a_0, \qquad h(n+N+1) \rightarrow_1 f(h(g(n+N+1))),$$

assuming $a_0 \in A$ and $f \in A \rightarrow A$. We introduce below a new, local control operator I_G into CTT+Nat called the *generalized natural iterator*, which is as an abstract definition of h:

$$\frac{\forall n \in Nat.(gn<n)\ \text{true},\ A \in data,\ a \in A,\ f \in A \to A,\ N \in Nat}{I_G AgafN \in Nat \to A} \quad \text{(GIter)}$$

The operational interpretation of I_G is given by the following rules of one-step reduction:

$$\frac{\forall n \in Nat.(gn<n)\ \text{true},\ A \in data,\ a \in A,\ f \in A \to A,\ ,N \in Nat,\ n \in Nat}{I_G AgafN(n+N+1) \to_1 f(I_G AgafN(g(n+N+1)))}$$

$$\frac{\forall n \in Nat.(gn<n)\ \text{true},\ A \in data,\ a \in A,\ f \in A \to A,\ N \in Nat,\ n \in Nat,\ (n \leq N)\ \text{true}}{I_G Agafn \to_1 a}$$

5.3.5.2 Terminating General Recursion Operator

We want to formalize in CTT+Nat the operational interpretation of a classical, arithmetical proof method based on a total recursive function $g \in Nat \to Nat$ instead of a natural predecessor. Such a proof corresponds to a classical program that evaluates to a datum after some finite number gn of steps of computation and such that if it escapes or doesn't terminate, it does so only in the top-level context. Since an escape or nontermination at the top-level mean resuming computation at the existing, data value-expecting context, they are are type-correct. We will internalize a top-level context as a part of a normal evaluation of a numerical program.

We can define the operational interpretation in question by taking type B in $B \times_{LR} A$[8] to be Nat as this type will make the top-level context accessible to the normal evaluation of the program.

According to ($\times_{LR}$-*elim*), the rule of type $Nat \to A$ is defined from $d \in Nat \times A$. We know that $d = \pi(h)$ where

$$h \in \Pi n \in Nat.\wedge_A(Nat,A,n,a)$$

According to the $\times_{LR}$ – *elim*, in order for the proofs of conjunction to be reducible to cut-free

7. See footnote 2 in this chapter.
8. See footnote 5 in this chapter.

forms, the following one-step reductions have to take place:

$$\pi(h)_0 \rightarrow_1 \lambda n \in Nat.n$$

$$\pi(h)_1 \rightarrow_1 \lambda n \in Nat.snd(hn)$$

When the assumption $[n \in Nat]$ is discharged by a natural number M, two things happen: M is duplicated and made accessible to the normal evaluation process, and the following K-step reduction takes place

$$snd(hM) \rightarrow_K a$$

where K is the number of applications of g to M that reaches 0. Every value requested in $a \in A$, obtained after some finite number K of computation steps, is evaluated normally. That is, g has to be such that it allows 0 to be reached after some finite number of steps of applications. In other words, the trace of g has to be a well-founded sequence of natural numbers, i.e., for any $n \in Nat$, $gn < n$ in order to preserve termination. Let $N \in Nat$ be a number such that for any $n \leq N$, gn returns 0. That is, numbers from 0 to N constitute an initial (or base) segment which doesn't have a (generalized) "predecessor". Thus, for any data type A, we can represent an $a \in A$ as being a request for the final result of the entire K-step computation through a unique evaluation rule $k \in Nat \rightarrow A$, such that :

$$hn \rightarrow_1 a_0, \qquad h(n+N+1) \rightarrow_1 f(n+N+1)(h(g(n+N+1)))$$

The corresponding local control operator is the *terminating general recursion operator* **G** and it is defined as follows:

$$\frac{\forall n \in Nat.(gn < n) \text{ true}, \ A \in data, \ a \in A, \ f \in Nat \rightarrow A \rightarrow A, \ N \in Nat}{GAgafN \in Nat \rightarrow A} \qquad \textbf{(GRec)}$$

The operational interpretation of the operator G is given in terms of the following rules of one-step reduction:

$$\frac{\forall n \in Nat.(gn < n) \text{ true}, \ A \in data, \ a \in A, \ f \in Nat \rightarrow A \rightarrow A, \ N \in Nat, \ n \in Nat}{GAgafN(n+N+1) \rightarrow_1 f(n+N+1)(GAgafN(g(n+N+1)))}$$

$$\frac{\forall n \in Nat.(gn < n) \text{ true}, \ A \in data, \ a \in A, \ f \in A \rightarrow A, \ N \in Nat, \ n \in Nat, \ (n \leq N) \text{true}}{GAgafNn \rightarrow_1 a}$$

5.3.5.3 Terminating General Recursive Computation Schema

In the section 5.3.4 we have defined the primitive recursive computation schema *natrec*. In this section we will define terminating, general recursive computation schema.

According to $\{\ \}-intro$, the canonical $d \in \{A\}_{Nat}$ is defined as $\rho(h)$ where

$$h \in Nat \rightarrow \Sigma(A,a)$$

According to $\{\ \}-red$,

$$\rho(h)_0 \rightarrow_1 \lambda n \in Nat.split(hn)$$

When the assumption $[n \in Nat]$ is discharged by a natural number N, the following K-step reduction has to take place

$$split(hN) \rightarrow_K a,$$

where K is the number of applications of a function g to N that reaches 0. Every value requested in $a \in A$, obtained after a finite number K of computation steps, is computed normally. The same analysis used to derive *natrec* also applies here. That is, the terms $n \in Nat$ representing top-level contexts of numerical programs have to be integer terms. This implies that the generalized predecessor function g has to be such that given an integer it returns an integer. We will use the notation f^D to denote a function $f \in Nat \rightarrow Nat$ from numerals (i.e., the terms k_n) to numerals. However, additionally, the normal, general recursive evaluation requires that for any $n \in Nat$, $gn < n$. If this is not the case, i.e., $gn \geq n$, for some $n \in Nat$, then the program will not evaluate normally. Rather, it will not terminate. Yet, such a top-level nontermination can be represented as a type-preserving escape. Namely, a program will resume an "escaped" computation at the top-level.

A computation based on a total recursive function can be expressed as a recursive, numerical context-typed continuation. Yet, in order to define the relevant computation schema, let us assume that we have a program $is\_less(m^D, n^D)$ that checks whether m is less than n, and such that it returns *true* if $m < n$ and *false* otherwise. Provided such a construction, the following data-valued expression can be introduced into CTT+Bool+Nat to implement G:

$gnatrec^{g^D}(case(is\_less(g^D(n^D),\ n^D);n^D;resultis(a_r));a_0;x^D,y.f(x^D,y))$ (*gnatrec*)

As *natrec*, the expression *gnatrec* distinguishes between local and top-level contexts. Only a top-level context of a numerical program is made accessible for nontermination or escaping. If the function denoted by g^D doesn't satisfy the termination condition, a general recursive program will abandon a request for its normal, non-terminating evaluation and will request a data-valued result a_r of an "escaped" computation. As we discussed before, the values of the type *Bool* can represent the top-level contexts of two disjoint *arbitrary* computations. Here, we are dealing with a concrete first-order classical theory, namely Peano Arithmetic, and the computations are the general recursive, numerical programs. The two possible values of the first argument of the *case* expression in *gnatrec* are the names of two disjoint top-level contexts. One value is the name of the context of an evaluation of a program for which the termination condition is satisfied. The other value is the name of the context of an "escaped" computation caused by violated termination condition. As we pointed out before, a computation which is resumed when a program doesn't terminate at the top-level is just a special case of an "escaped" computation. In other words, the programming construct "*case*" can implement a structured, top-level context of any total, classical computation. In the definition of *gnatrec* given above, we have arbitrarily chosen "*true*" as the name of the context of a normally terminating evaluation and "*false*" as the name of the context of an "escaped" evaluation. Equally, we might have reversed this choice.

The operational semantics of a normal evaluation of a general recursive program can be expressed in terms of the following one-step reductions:

$gnatrec^{g^D}(case(is\_less(g^D(k_{N-n}),k_{N-n});k_{N-n};resultis(a_r));a;x^D,y.f(x^D,y))$

$\rightarrow_1 a$ (*gnatrec-red1*)

$gnatrec^{g^D}(case(is\_less(g^D(k_{n+N+1}),k_{n+N+1});k_{n+N+1};resultis(a_r));k_{n+N+1};a;$

$x^D,y.f(x^D,y)) \rightarrow_1 f(k_{n+N+1},gnatrec^{g^D}(g^D(k_{n+N+1});a;x^D,y.f(x^D,y)))$ (*gnatrec-red2*)

We have extended the class of classical programs providing the evidence in a constructive sense from primitive to *general recursive*. Of course, this does not change the result that only for Π^0_2 formulas it is always decidable whether their proofs are normalizable.

5.4 Binary Trees

In this section, we will introduce a type theory, CTT+Tree, that formalizes the computational content of a classical theory of the binary trees. We will define a control structure associated with the binary tree reasoning which is the binary tree iteration. We will introduce the primitive recursion operator associated with the binary tree reasoning. These constructions will be defined in a manner applicable to classical reasoning. Finally, we will implement binary tree-based, primitive recursion operator by an expression schema with a clear and direct operational semantics. We begin with the introduction of the binary trees to CTT+Tree.

The second-order quantification

$$tree = [x{:}prop](x{\Rightarrow}x{\Rightarrow}x){\Rightarrow}x{\Rightarrow}x$$

is an impredicative construction of the concept of a binary tree. What follows is the predicative construction of that concept. We separate the binary tree data type from the control structure associated with it by forgetting its derivation in **F**:

$$Tree = \text{'}tree\text{'}$$

We introduce the following rule of formation for binary trees to CTT+Tree, i.e., CTT extended with binary trees:

$$Tree \in data \qquad (Tree\text{-}form)$$

The constants l_t will be introduced to CTT+Tree to represent binary trees $t \in Tree$:

$$l_t \equiv \Lambda X{:}prop.\lambda b{:}X{\Rightarrow}X{\Rightarrow}X.\lambda n{:}X.b\ (...(bnn)...)\ (...(bnn)...)$$

The empty tree $Null \equiv$ '*null*', where

$$null \equiv \Lambda X{:}prop.\lambda b{:}X{\Rightarrow}X{\Rightarrow}X.\lambda n{:}X.n$$

is the only tree definitionally equal to the constant l_{Null}.

The branch function is represented in **F** by the following construction:

$$branch \equiv \lambda left{:}tree.\lambda right{:}tree.\Lambda X{:}prop.\lambda b{:}X{\Rightarrow}X{\Rightarrow}X.$$

$$\lambda n{:}x.(b\ (left\ x\ b\ n)\ (right\ x\ b\ n))$$

of type $tree{\Rightarrow}tree{\Rightarrow}tree$. The term *branch*, when applied to a (representation of) binary tree t and binary tree s, reduces to a universal abstraction representing the tree $t\&s$. The branching

function in CTT+Tree is the function constant *Branch* such that $Branch(l_t,l_s) \equiv$ '*branch T S*', where $T \in$ *tree* and $S \in$ *tree* represent trees t and s in **F**. Since (*branch T S*) *red V*, where $V \in$ *tree* represents the tree $t\&s$ in **F**, the constant *Branch* is defined in CTT+Tree by the following one-step reduction:

$$Branch(l_t,l_s) \rightarrow_1 l_{t\&s} \qquad (Branch\text{-}red)$$

The rest of this chapter is devoted to the top-level, operational interpretation of the binary tree-based reasoning. We will define the binary tree iterator and the binary tree primitive recursion operator. We will implement the latter as a top-level, context-typed continuation schema. The process of formalization of the binary tree reasoning is analogous to the formalization of the arithmetical reasoning. Hence, the definitions introduced in the following sections are not going to be explained in as much detail as it was done for arithmetical reasoning in the previous sections.

5.4.1 Binary Tree Iteration

In order to define a control structure associated with the type *Tree*, one has to show how to construct a cut-free proof of an arbitrary formula about binary trees. In other words, a rule $k \in Tree \rightarrow A$ will be defined for any $A \in$ *data*. We will show below that the type $Tree + A$ yields a control operator which completely characterizes the reasoning about binary trees.

According to ($+_L$-*elim*), the rule of type $Tree \rightarrow A$ is defined from $d \in Tree + A$. We know that $d = \langle f,g \rangle$ where

$$f \in \Pi t \in Tree.\vee_L(Tree,A,t)$$

$$g \in Tree \rightarrow \vee_R(Tree,A,a)$$

According to ($+_L - elim$),

$$\langle f,g \rangle^0 \rightarrow_1 \lambda t \in Tree.t,$$

$$\langle f,g \rangle^1 \rightarrow_1 \lambda t \in Tree.outr(gt)$$

By discharging the assumption $[t \in Tree]$, the following reduction has to take place:

$$outr(gt) \rightarrow_m a$$

where m is the depth of the tree t, and t is duplicated. According to ($+_L$- *red*), the evaluation

rule $k \in Tree \rightarrow A$ for the right inject of the type $Tree + A$ is defined as follows:

$$kNull \rightarrow_1 a_0, \quad k(u\&v) \rightarrow_1 r(ku)(kv),$$

assuming a constant $a_0 \in A$ and a function constant $r \in A \rightarrow A \rightarrow A$. We shall introduce a local control operator T into CTT+Tree:

$$\frac{A \in data, \quad a_0 \in A, \quad r \in A \rightarrow A \rightarrow A}{TAa_0r \in Tree \rightarrow A} \qquad \textbf{(TIter)}$$

The tree iterator T is defined by the following rules of the one-step reduction:

$$\frac{A \in data, \quad a_0 \in A, \quad r \in A \rightarrow A \rightarrow A, \quad s \in Tree, \quad t \in Tree}{TAa_0r(s\&t) \rightarrow_1 r(TAa_0rs)(TAa_0rt)} \qquad \textbf{(TIter-}\textit{red1}\textbf{)}$$

$$\frac{A \in data, \quad a_0 \in A, \quad r \in A \rightarrow A \rightarrow A}{TAa_0r(Null) \rightarrow_1 a_0} \qquad \textbf{(TIter-}\textit{red2}\textbf{)}$$

5.4.2 Binary Tree Recursion Operator

In this section we shall derive a *binary tree recursion operator* by specializing a data-value expecting context in the classical conjunction type $A \times_{LR} B$ to the type *Tree*.

We take the type B in $B \times_{LR} A$ to be *Tree*. According to ($\times_{LR}$-*intro*), the rule of type $Tree \rightarrow A$ is defined from $d \in Tree \times A$. We know that $d = \pi(h)$ where

$$h \in \Pi t \in Tree./\backslash_A(Tree, A, t, a)$$

According to the $\times_{LR}$ – *elim*, the following one-step reductions have to take place:

$$\pi(h)_0 \rightarrow_1 \lambda t \in Tree.t$$

$$\pi(h)_1 \rightarrow_1 \lambda t \in Tree.snd(ht)$$

By discharging the assumption $[t \in Tree]$, the following m-step reduction has to take place, where m is the depth of the tree t:

$$snd(ht) \rightarrow_m a$$

We introduce a new local control operator B into CTT+Tree:

$$\frac{A \in data, \quad a_0 \in A, \quad f \in Tree \rightarrow A \rightarrow Tree \rightarrow A \rightarrow A}{BAa_0f \in Tree \rightarrow A} \qquad \textbf{(TRec)}$$

The operational semantics of the binary tree recursion operator B is defined by the following

one-step reduction rules:

$$\frac{A \in data,\ a_0 \in A,\ f \in Tree \to A \to Tree \to A \to A,\ t \in Tree,\ s \in Tree}{BAa_0f(t\&s) \to_1 ft(BAa_0ft)s(BAa_0fs)} \quad \textbf{(TRec-}red\textbf{)}$$

$$\frac{A \in data,\ a_0 \in A,\ f \in Tree \to A \to Tree \to A \to A}{BAa_0f(Null) \to_1 a_0} \quad \textbf{(TRec-}red\textbf{)}$$

5.4.3 Binary Tree-Based Primitive Recursive Computation Schema

In this section, we will derive a primitive recursive programming schema associated with the classically-founded, binary tree reasoning. This schema will constitute the top-level, operational semantics of primitive recursive functions on binary trees.

In CTT+Tree, according to $\{\ \}-intro$, the canonical $d \in \{A\}_{Tree}$ is defined as $\rho(h)$ where

$$h \in Tree \to \Sigma(A,a)$$

According to $\{\ \}-red$,

$$\rho(h)_0 \to_1 \lambda t \in Tree.split(ht)$$

When the assumption $[t \in Tree]$ is discharged by a binary tree T, the following N-step reduction, where N is the depth of T, has to take place

$$split(hT) \to_N a$$

Every value requested in $a \in A$, obtained after some finite number N of computation steps, is computed normally. Here, the tree T is not duplicated. Hence, the top-level context represented by T is not preserved. If we require that a term representing a top-level context of a program on binary trees is a data-valued term, then the program will always conclude its normal evaluation at the top-level context. We introduce a special notation t^T for the binary tree terms, i.e., terms computable to constants l_t ($t \in Tree$). Hence, the following data-valued expression can be introduced into CTT+Tree to implement $\boldsymbol{B}$:

$$treerec(t^T; a; x_1^T, x_2^T, y_1, y_2.f(x_1^T, x_2^T, y_1, y_2)) \quad (treerec)$$

If a_0 is a binary tree expression, then *treerec* itself becomes a binary tree expression.

The operational interpretation of *treerec* is given in terms of the following reduction rules:

$$treerec(Null;a;x_1^T,x_2^T,y_1,y_2.f(x_1^T,x_2^T,y_1,y_2)) \rightarrow_1 a \qquad (treerec\text{-}red1)$$

$$treerec(l_{t\&s};a;x_1^T,x_2^T,y_1,y_2.f(x_1^T,x_2^T,y_1,y_2)) \rightarrow_1 \qquad (treerec\text{-}red2)$$

$$f(l_t,l_s,treerec(l_t;a;x_1^T,x_2^T,y_1,y_2.f(x_1^T,x_2^T,y_1,y_2)),$$

$$treerec(l_s;a;x_1^T,x_2^T,y_1,y_2.f(x_1^T,x_2^T,y_1,y_2))).$$

One could easily extend the operational interpretation result shown for Peano Arithmetic in the section 5.3.4 to a classical theory of binary trees. As a consequence, not only Peano Arithmetic but other finite, classical theories can serve as programming logics. In other words, there are other data types besides that of natural numbers whose expressions have no control side-effects. These are lists of integers, booleans, streams, etc., with endless possibilities.

CHAPTER 6

IMPLEMENTATION OF RECURSIVE FUNCTIONS

This chapter consists of two major parts. In the first part, we will introduce examples of computable functions on natural numbers. We will show how to implement in CTT+Nat some typical primitive recursive functions like subtraction, addition, factorial, Fibonacci, etc. First, we will express these functions in terms of the natural iterator and pairing. This corresponds to the "by-value-only" computation. Such definitions are also possible in the second-order λ-calculus. Then, we will implement these primitive recursive functions using classical, programming schema *natrec*. We will also show a simple example of a program for a proof by contradiction. This program will either evaluate normally or it will abandon its normal evaluation and resume its computation at the top-level returning a new, final result. We will give an example of a terminating, general recursive function - division by repeated subtraction. A program for this function is an instance of the terminating, general recursive schema *gnatrec*. We will also implement the Ackermann's function which is not primitive recursive but it is constructed by the simultaneous induction on two variables. The Ackermann's function is a generalization of the schema *gnatrec* since the simultaneous induction on two variables corresponds a finite iteration of some reasonable function corresponding to the generalized natural predecessor.

In the second part of this chapter, we will extend CTT with the type of lists of natural numbers. We will introduce a recursive computation scheme associated with lists. We will also illustrate program implementation in CTT extended with lists of natural numbers, i.e., in $\mathsf{CTT+List}_{\mathsf{Nat}}$. The example which we present is a program that computes the smallest element in a list of positive integers. The program illustrates the use of *resultis* expression to handle run-time error when trying to compute the smallest element of an empty list.

6.1 Primitive Recursive Programs

Any primitive recursive function is defined using composition of functions and the following schemata:

$$f(x_1, \cdots, x_n, 0) = d(x_1, \cdots, x_n)$$

$$f(x_1, \cdots, x_n, x+1) = e(x_1, \cdots, x_n, x, f(x_1, \cdots, x_n, x))$$

This schemata is implemented in CTT+Nat by the expression schema *natrec*. Actually, *natrec* itself has only one variable, the one which is inducted upon. Yet, allowing other not inducted upon integer variables x_1^D, .. , x_n^D will not change the correctness of *natrec*. The expression schema *natrec* with additional free integer variables is defined as follows:

$$natrec(n^D; a(x_1^D, \cdots, x_m^D); x^D, y.f(x_1^D, \cdots, x_m^D, x^D))$$

In the following subsections, we will consider several examples of the primitive recursive functions. Each function will be expressed first in terms of the natural iterator and pairing. This is "by-values-only" computation where functions decompose their arguments completely according to the primitive structural induction. Subsequently, we will implement each function using the classical, programming schema *natrec*.

6.1.1 Predecessor and Subtracting One

Let us define an algorithm for *predecessor* computed "by-values-only":

$$pred(0) = 0,\ pred(n+1) = n$$

We will express *pred* in terms of the natural iterator *I*. Let the function $p \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ be defined by the following one-step reduction:

$$p(n,m) \rightarrow_1 (n+1, n)$$

Then, $i \equiv I(Nat \times Nat)(0,0)p \in Nat \rightarrow (Nat \times Nat)$ is an iterator for $Nat \times Nat$. The predecessor computed "by values only" is defined as follows:

$$Pred\ n \equiv snd(in)$$

In order to implement subtracting one from an integer in terms of *natrec*, we need the predecessor constant function *P* defined on numerals, such that

$$P(k_{n+1}) \rightarrow_1 k_n$$

The following λ-abstraction is the definition of the predecessor in the system F:

$$pred \equiv \lambda n \in nat.(n(nat \Rightarrow nat)(\lambda z \in nat \Rightarrow nat.<add\ (succ\ zero)(z\ zero),(z\ zero)>)$$
$$<zero,zero>(succ\ zero))$$

where *add* is the addition

$$add \equiv \lambda n{:}nat.\lambda m{:}nat.\Lambda X{:}prop.\lambda f{:}X \Rightarrow X.\lambda x{:}X.(nXf)(mXfx)$$

and < , > is pairing

$$<a,b> \equiv \lambda s{:}nat.(s\ nat\ (\lambda z{:}nat.a)\ b)$$

We introduce the function constant P to CTT+Nat such that if

$$(pred\ N)\ red\ M$$

in F, then

$$P(k_n) \rightarrow_1 k_{n-1}$$

in CTT+Nat, where N represents the natural number n and M represents $n-1$ in F. If $N = zero$, then also $M = zero$, and $P0 \rightarrow_1 0$.

Provided P as defined above, we can define in CTT+Nat a program for subtracting one from an integer, i.e., for "$n-1$", as follows:

$$n^D - k_1 \equiv natrec(n^D;0;x^D,y.P(x^D))$$

If $n^D \neq 0$, then

$$natrec(n^D;0;x^D,y.P(x^D)) \rightarrow_1 P(n^D)$$

6.1.2 Subtraction

In this section, we will implement the subtraction $Sub \in Nat \rightarrow Nat \rightarrow Nat$, defined by the following equations:

$$Sub(m,0) = m, \quad Sub(m,n+1) = Sub(m-1,n)$$

We will first implement subtraction as "by-values-only" computation. Let one-step function $s \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ be defined as follows:

$$s(l,k) \rightarrow_1 (l+1,k-1)$$

Then, $i \equiv I(Nat \times Nat)(0,m)s \in Nat \rightarrow (Nat \times Nat)$ is an iterator for $Nat \times Nat$. The subtraction computed "by-values-only" is expressed in terms of the natural iterator and pairing as follows:

$$sub\ m\ n \equiv snd(in)$$

The actual program for subtraction is represented in CTT+Nat by the following instance of the programming schema *natrec*:

$$m^D - n^D \equiv natrec(n^D; m^D; x^D, y.y - k_1)$$

We will use the notation sub_n^D for a one argument function from integers to integers such that

$$sub_n^D(m^D) \equiv m^D - n^D$$

6.1.3 Addition and Multiplication

In this section, we will implement addition and multiplication. Addition is defined informally by the following equations:

$$Add(m,0) = m,\ Add(m,n+1) = Add(m,n)+1$$

Let the one-step function $a \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ be defined by the following reduction:

$$a(k,l) \rightarrow_1 (k+1, l+1)$$

Then $i = I(Nat \times Nat)(0,m)f \in Nat \rightarrow (Nat \times Nat)$ is an iterator for $Nat \times Nat$. We can implement addition terms of the natural iterator as follows:

$$add\ m\ n \equiv snd(in)$$

If the arguments n and m for addition $Add(m,n)$ always compute to numerals k_n and k_m, then the program for addition is expressed in CTT+Nat as follows:

$$m^D + n^D \equiv natrec(n^D; m^D; x^D, y.S(y))$$

Multiplication is expressed informally by the following equations:

$$Mult(m,0) = 0,\ Mult(m,n+1) = Add(m, Mult(m,n))$$

To implement *Mult* as an algorithm computed "by-values-only", we will introduce a one-step function $u \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ defined as follows:

$$u(k,l) \rightarrow_1 (k+1,\ add\ m\ l)$$

Then $i \equiv I(Nat \times Nat)(0,0)\,u \in Nat \rightarrow (Nat \times Nat)$ is an iterator for $Nat \times Nat$. Multiplication computed "by-values-only" is implemented by

$$mult\ m\ n \equiv snd(in)$$

If the arguments n and m for addition $Mult(m,n)$ always compute to numerals k_n and k_m, then the program for multiplication is expressed in CTT+Nat as follows:

$$m^D * n^D \equiv natrec(n^D; k_1; x^D, y.m^D + y)$$

6.1.4 Factorial

The factorial function is defined informally by the following equations:

$$Fact(0) = 1, \quad Fact(n+1) = (n+1) * Fact(n)$$

Let the function $f \in (Nat \times Nat) \rightarrow (Nat \times nat)$, be defined by the following one-step reduction:

$$f(n,m) \rightarrow_1 (n+1, mult\ (n+1)\ m)$$

Then $i \equiv I(Nat \times Nat)(0,1)f \in Nat \rightarrow (Nat \times Nat)$ is an iterator for $Nat \times Nat$. Factorial is expressed in terms of the natural iterator and pairing as $snd(in)$.

If it is required that the argument n of factorial always computes to a numeral k_n, factorial can be implemented in CTT+Nat by the following instance of the expression schema *natrec*:

$$natrec(n^D; 1; x^D, y.x^D * y)$$

6.1.5 Fibonacci

The recursive scheme *Fibonacci* is defined by the following equations:

$$Fib(0) = a, \quad Fib(1) = b, \quad Fib(n+2) = F(Fib(n), Fib(n+1))$$

We will expressed Fib in terms of the natural iterator and pairing. Let us assume that

$$a \in A, \quad b \in A, \quad F \in A \rightarrow A \rightarrow A$$

Let the one-step function $f \in (A \times A) \rightarrow (A \times A)$ be defined as follows:

$$f(c,d) \rightarrow_1 (d, Fcd)$$

Then, the following expression is an iterator for $A \times A$:

$$i \equiv I(A \times A)(a,a)(\lambda x \in A \times A.I(A \times A)(a,b)f(Pred\ n)) \in Nat \rightarrow A \times A$$

We can express $Fib(n)$ by $snd(in)$.

In the case of the standard Fibonacci function, Fib_s, being an instance of $Nat \rightarrow Nat$, we take $F = +$, $a = 1$ and $b = 1$. Then, the one-step function $f_s \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ needed to implement Fibonacci in terms of the natural iterator is defined as follows:

$$f_s(n,m) \rightarrow_1 (m, add\ n\ m)$$

The iterator for $Nat \times Nat$ used to implement Fibonacci is defined by

$i_s \equiv I(Nat\times Nat)(1,1)(\lambda x \in Nat\times Nat.\ I(Nat\times Nat)(1,1)f_s(Pred\ n)) \in Nat\rightarrow Nat\times Nat$

We can express $Fib_s n$ in terms of the natural iterator by $snd(i_s n)$.

The actual program for the standard Fibonacci function can be expressed as the following instance of the programming schema *natrec*:

$$natrec(n^D;k_1;x^D,z.natrec(x^D;1;y^D,w.z+w))$$

6.2 Simple Example with a Proof by Contradiction

We will illustrate in the section the use of the programming construct *resultis* which is the operational content of the classical rule of the double-negation elimination. Let us consider the formula

$$\phi \equiv \exists\, n \in Nat.prime(n) \wedge n<100$$

Clearly, there are many proofs of this sentence, and, hence, many realizing programs. We will introduce one such program, namely

$$natrec(resultis(case(is\_prime(k_{104});case(is\_less(k_{104},k_{100});k_{104};resultis(k_2));$$
$$resultis(k_3)));0;x^D.y.x^D) \rightarrow_1 k_2$$

where $is\_prime(k_n)$ is a program which returns *true* when n is prime and *false* otherwise. The program *is_less* checks whether one integer is smaller than the other. It is defined by the following one-step reductions:

$$is\_less(k_m,k_n) \rightarrow_1 true \ \text{ if } m<n \qquad is\_less(k_m,k_n) \rightarrow_1 false \ \text{ if } m\geq n$$

By allowing free integer variables, the formula ϕ becomes a Π^0_2 sentence and its realizing program is as follows:

$$natrec(case(is\_prime(n^D);case(is\_less(n^D,k_{100});n^D;resultis(k_2));resultis(k_3));$$
$$0;x^D.y.x^D)$$

If the above program is supplied with a prime integer less than 100, then the program evaluates normally and returns that integer as the final result. If the program is supplied with a nonprime integer, then the normal execution of the program is abandoned and the program returns the integer 3 as the final result. Finally, if the program is supplied with a prime integer but greater or equal than 100, then again, the program abandons its normal execution and

returns with the result being the integer 2.

6.3 Terminating General Recursion - Division by Repeated Subtraction

In this section we introduce an algorithm for the division by repeated subtraction as an illustration of a terminating, general recursion program. Division by repeated subtraction is an example of an induction based upon a primitive recursive function instead of the natural predecessor. The division function

$$div \in Nat \rightarrow Nat \rightarrow Nat$$

is defined informally as follows:

$$\begin{aligned} div(n,m) \;=\; & \text{if } n<m \textit{ then } 0 \\ & \textit{else if } m=n \textit{ then } 1 \\ & \textit{else } div(n-m,m)+1 \end{aligned}$$

We begin with an implementation of the division in terms of the iteration based on the generalized predecessor being the subtraction *sub*. In order to define the division by repeated subtraction, we need an algorithm for the function "less than" $le \in Nat \rightarrow Nat \rightarrow Bool$ which is defined informally as follows:

$$le(m,n) \;=\; \text{if } m<n \textit{ then true else false}$$

Let the one-step function $g \in (Nat \times (Bool \times Bool)) \rightarrow (Nat \times (Bool \times Bool))$ be defined as follows:

$$g(n,(false,a)) \rightarrow_1 (Pred\ 1,(I\ Bool\ true\ (\lambda x.false)(Pred\ n),a))$$

$$g(n,(true,a)) \rightarrow_1 (n,(true,true))$$

Then $i \equiv I(Nat \times (Bool \times Bool))(m,(true,true))g$ is an iterator for $Nat \times (Bool \times Bool)$. The function "less than" is defined as follows:

$$le\ m\ n \equiv snd(snd(in))$$

Provided the above definition of the "less than" function, the one-step function $d \in (Nat \times Nat) \rightarrow (Nat \times Nat)$ needed to express the division by repeated subtraction in terms of the generalized iteration and pairing is defined as follows:

$$d(n,k) \rightarrow_1 (sub\ n\ m,\ [k,k+1]le\ m\ n)$$

where [,] is the control structure associated with the type *Bool* which was introduced in Chapter 5, and *sub* is the subtraction expressed in terms of the natural iterator, introduced in the section 6.1.2. We can express the division by repeated subtraction in terms of the generalized iterator I_G as follows:

$$div_P\ n\ m = snd(I_G(Nat \times Nat)\, sub(n,0)\, dm)$$

The division by repeated subtraction is not based on the natural predecessor but its construction constitutes the computational content of a proof by the method of complete induction. We can prove for any natural numbers n, m such that $n > m > 0$, that $n - m < n$. In the rule of the general recursive computation scheme *gnatrec*, based on a total function g^D instead of the natural predecessor, we will take g^D to be the subtraction sub_m^D, which was defined in the section 6.1.2. Let the program $is\_less(m,n)$ check whether m is less than n:

$$is\_less(n^D, m^D) \equiv natrec(n^D - m^D; true; x^D, y.false).$$

The actual program for division by repeated subtraction is expressed by the following instance of the general recursive programming schema *gnatrec*:

$$gnatrec^{sub_m^D}(case(is\_less(n^D, m^D); 0; n^D); 0; x^D, y.y + k_1)$$

6.4 Beyond Primitive Recursion - Ackermann's Function

The Ackermann's function is the classical example of a recursive function which is not primitive recursive. It is a double-recursive function that majorizes[1] all primitive recursive functions. Ackermann's function is defined by iterating iteration. More precisely, the proof of its computability uses a technique of double induction:

1. The proposition $A(0,n)$ is proved for all n by induction on n: first $A(0,0)$ is proved, and then assuming $A(0,n)$ one shows that $A(0,n+1)$ follows.

---

1. A function g majorizes a function f if f is computable in time bounded by g.

2. It is assumed that $A(m,n)$ holds for all n (the induction step for m). Then $A(m+1,n)$ is proved by induction on n: first $A(m+1,0)$ is proved, and then assuming $A(m+1,n)$ (the induction step for n), one proves $A(m+1,n+1)$.

This is simply a repeated use of a single induction. This exercise will show how to express the Ackermann's function in CTT+Nat.

Let us consider the following definitions:

$$f_0 \equiv y+x$$
$$f_1 \equiv x*y$$
$$f_2 \equiv x^y \equiv natrec(y;1;x',z.x*z)$$
$$f_3 \equiv x\uparrow\uparrow y \equiv natrec(y;1;x',z.x^z).$$

These functions are the first in the series of functions f_1, f_2,... where $f_{n+1}(x,y)$ is the result of

$$f_n(x,\cdots f_n(x,x)\cdots)) \qquad \text{if } y>0$$

where f_n is applied $y-1$ times. The function $Ack(n,x,y) \equiv f_n(x,y)$ is the Ackermann's function.

The definition of Ackermann's function, $Ack \in Nat\rightarrow Nat\rightarrow Nat\rightarrow Nat$ starts by giving as a primary basis for induction the equations when the first argument is zero :

$$(1)\ Ack(0,x,0) = x \qquad (2)\ Ack(0,x,y+1) = Ack(0,x,y)+1$$

A secondary basis specifies the function for all values of its first argument when the third argument is held at zero:

$$(3)\ Ack(1,x,0) = 0 \qquad (4)\ Ack(n+2,x,0) = 1$$

The definition is completed by the dyadic recursion equation:

$$(5)\quad Ack(n+1,x,y+1) = Ack(n,x,Ack(n+1,x,y))$$

The first two equalities define $f_0(x,y) = x+y$ and the fifth equality defines f_{n+1} in terms of f_n.

First, we will define Ackermann's function as "by-values-only" computed algorithm. Let the one-step function $F \in (Nat\times(Nat\rightarrow Nat))\rightarrow(Nat\times(Nat\rightarrow Nat))$ specifying f_{n+1} in terms of

f_n, be defined as follows:

$$F(n,f) \rightarrow_1 (n+1,\lambda y \in Nat.\ INat(INat0(\lambda m \in Nat.1)n)fy)$$

The innermost iteration $INat0(\lambda m \in Nat.1)n$ on the right-hand side of the above reduction defines the third and fourth Ackermann's equations.

The one-step function F defined above yields the following iterator for the type $Nat\times(Nat\rightarrow Nat)$:

$$h\ x \equiv I(Nat\times(Nat\rightarrow Nat))(0,(add\ x))F$$

We define the Ackermann's function in terms of the nested application of the natural iterator as follows:

$$Ack\ n\ x\ y \equiv (snd(hxn))y.$$

Using this algorithm, we will compute below the Ackermann's function for several different values of its arguments. For clarity, we omit $C\equiv Nat\times(Nat\rightarrow Nat)$ in the definition $ICaf$ of the natural iterator in the following examples:

$Ack\ 0\ x\ 0 = add\ x\ 0 = x$

$Ack\ 0\ x\ (Sy) = add\ x\ (Sy)$

$Ack\ (S0)\ x\ 0 = (snd(F(I(0,add\ x)F0)))0 = (snd(F(0,add\ x)))0 =$

$(snd((S0,\lambda y \in Nat.I(I0(\lambda n \in Nat.(S0))0)(add\ x)y)))0 =$

$(\lambda y \in Nat.I0(add\ x)y)0 = I0(add\ x)0 = 0$

$x+1 = Ack\ (SS0)\ x\ 0 = (snd(FF(0,add\ x)))0 = (snd(F(S0,\lambda y \in Nat.I0(add\ x)y))0 =$

$\lambda y \in Nat.I(I0(\lambda n \in Nat.(S0))(S0))(S0)(\lambda y \in Nat.I0(add\ x)y)y)0 =$

$I0(\lambda n \in Nat.(S0))S0 = S0$

$2*x = Ack\ (S0)\ x\ (SS0) = (snd(F(I(0,add\ x)F0)))(SS0) =$

$(snd(F(0,add\ x)))(SS0) = (snd((S0,\lambda y \in Nat.I(I0(\lambda n \in Nat.(S0))0)(add\ x)y))(SS0)$

$(\lambda y \in Nat.I0(add\ x)y)(SS0) = add\ x\ (add\ x\ (I0(add\ x)0)) = add\ x\ (add\ x\ 0) = add\ x\ x$

$x^2 = Ack\ (SS0)\ x\ (SS0) = (snd(FF(0,add\ x)))(SS0) =$

$(snd(F(S0,\lambda y \in Nat.I0(add\ x)y)))(SS0) =$

$(\lambda y \in Nat.I(I0(\lambda n \in Nat.(S0))(S0))(\lambda y \in Nat.I0(add\ x)y)y)(SS0) =$

$(\lambda y \in Nat.I(S0)(\lambda y \in Nat.I0(add\ x)y)y)(SS0) =$

$(\lambda y \in Nat.I0(add\ x)y)(\lambda y \in Nat.I0(add\ x)y)(S0) = I0(add\ x)x$

We have defined the Ackermann's function $Ack(n,x,y)$ by values only, *i.e.*, for each n separately. More precisely, the Ackermann's function is computed by decomposing its argument n completely to $SS...S0$ (with n occurrences of S) and evaluating the result while reconstructing n. If it is required that n always computes to a numeral, an actual program for computing the Ackermann's function can be defined as it is shown below.

Let us recall from Chapter 5 the control structure associated with the type *Bool* (a type with two objects *true* and *false*), namely the conditional [,] defined by the following one-step reductions:

$$[a,b]_A\ true \rightarrow_1 a$$
$$[a,b]_A\ false \rightarrow_1 b$$

where $a,b \in A$. We will introduce a corresponding programming construct defined on constants c_a (e.g., numerals k_n) rather than on values $a \in A$. This new construct is the term $case(b;c_1;c_2)$ defined as follows:

$$case(true;c_a;c_b) \rightarrow_1 c_a$$
$$case(false;c_a;c_b) \rightarrow_1 c_b$$

For example, if A is *Nat* and $n,m \in Nat$, then $case(b;k_n;k_m)$ is defined as follows:

$$case(true;k_n;k_m) \rightarrow_1 k_n$$
$$case(false;k_n;k_m) \rightarrow_1 k_m$$

We also need a boolean-valued program for checking whether a number is zero:

$$eqzero \equiv I\ Nat\ true\ g$$

where g is defined by the following one-step reduction:

$$gb \rightarrow_1 false$$

The function $eqzero \in Nat \rightarrow Bool$ is implemented by the following term in CTT+Nat+Bool:

$$is\_zero(n^D) \equiv natrec(n^D;true;x^D,y.false)$$

We also have to define a boolean-valued program for checking whether a number is greater

than zero:

$$gtzero \equiv I\ Nat\ false\ h$$

where h is defined by

$$hb \rightarrow_1 true$$

The function $gtzero \in Nat \rightarrow Bool$ is implemented as follows:

$$grt\_than\_zero(n^D) \equiv natrec(n^D; false; x^D, y.true)$$

We note that the first two Ackermann's equalities are defined by the following single equality:

$$Ack(0,x,y) = x+y$$

Similarly, the third and fourth equalities can be implemented using the terms *case* and *is_zero*. More precisely, $Ack(n+1,x,0)$ is implemented by

$$case(is\_zero(k_n);0;k_1)$$

In order to implement the fifth Ackermann's equation, we define n applications of fxx as $do\_app(n,f,x)$ and implement it as

$$natrec(n^D; x^D; y^D, z.f^D(x^D, z))$$

provided that f is a function from integers to integers. This is expressed by marking the term f with a superscript D which is the notation for numerical terms. Then the fifth Ackermann's equation is defined as follows:

$$Ack(n+1,x,y+1) = do\_app(y, \lambda y \in Nat.\lambda z \in Nat.Ack(n,y,z), x)$$

Putting it all together, the function

$$f(n) \equiv \lambda x \in Nat.\lambda y \in Nat.Ack(n,x,y)$$

is defined by the following equations:

$$f(0) = \lambda x \in Nat.\lambda y \in Nat.x+y$$

$$f(n+1) = \lambda x \in Nat.\lambda y \in Nat.[do\_app(y-1, f(n), x),\ [0,1](eqzero\ n)](gtzero\ y)$$

We note that $f(0)$ is a function from integers to integers. We can easily show by induction that if $f(n)$ is from integers to integers, then a finite number of applications of $f(n)$ gives a function from integers to integers as well. We introduce, in sequence, the following notation for the functional terms from numerals to numerals and for the pairs of integers:

$$\lambda x^D.e^D$$

$$(n^D, m^D)$$

The Ackermann's function $Ack(n,x,y)$ is implemented by the nested application of the programming schema *natrec* as follows:

$$(natrec(n^D;\lambda(x_1^D,y_1^D).x_1^D+y_1^D;v^D,z.(\lambda(x_1^D,y_1^D).case(grt\_than\_zero(y_1^D);$$
$$natrec(y_1^D-1;x_1^D;w^D,r.z(x_1^D,r));case(is\_zero(v^D);0;1)))))(x^D,y^D).$$

The Ackermann's function is a generalization of the schema *gnatrec* where the generalized natural predecessor is defined by a finite iteration.

6.5 List Recursion

We can easily extend the result for natural numbers as a domain of quantification for classical sentences to encompass other inductive types like lists, trees, etc. The terms of such types have no control side-effects.

In this section, we will extend CTT with *lists*. The impredicative definition of the type of lists of elements of a data type P is the following propositional schema:

$$(list\ P) \equiv \forall X{:}Prop.X \Rightarrow (P \Rightarrow X \Rightarrow X) \Rightarrow X$$

Its constructors *nil* (empty list constructor) and *cons* (non-empty list constructor) are defined as follows:

$$nil \equiv \Lambda X{:}Prop.\lambda n{:}X.\lambda c{:}P \Rightarrow X \Rightarrow X.n$$

$$(cons\ e\ l) \equiv \Lambda X{:}Prop.\lambda n{:}X.\lambda c{:}P \Rightarrow X \Rightarrow X.(c\ e\ (l\ X\ n\ c))$$

The impredicative definition of lists identifies list objects with their operational interpretation. We want to separate lists from their operational interpretation, i.e., form the control structure associated with lists. We can accomplish this by forgetting the derivations of list constructs (*listP*), *nil*, and (*cons e l*) in **F** and only preserving their intuitionistic interpretation. As a consequence, the following rule of formation for lists is introduced to CTT+List_A:

$$\frac{A \in data}{List_A \in data} \qquad (List\text{-}form)$$

where A is a data type variable in CTT corresponding to P in (*list P*). The constants j_l will be introduced to CTT+List_A to represent lists $l \in List_A$. The empty list $Nil \equiv$ '*nil*' is the only list equal to the constant j_{Nil} by definition.

The constructor *cons*, when applied to a representation of a value h of type A and the representation of a list l of values of type A, reduces to a universal abstraction representing the list $h.l$. The list constructor in $\mathsf{CTT+List_A}$ is the function constant *Cons* such that

$$Cons(c_h, j_r) \equiv \text{`}cons\ H\ R'$$

where R: (*list* P) and H: P represent a list r and an object h of type A in **F**, and c_h is a constant in CTT+A representing $h \in A$. Provided the following reduction in the system **F**

$$(cons\ H\ R)\ red\ L$$

where L: (*list* P) represents in **F** a list $h.r$, the constant function *Cons* is defined in $\mathsf{CTT+List_A}$ by the following one-step reduction:

$$Cons(c_h, j_r) \rightarrow_1 j_{h.r} \qquad (Cons\text{-}red)$$

We will not present in this thesis the actual derivation of the computation schema for lists. The process of formalization of reasoning with lists is analogous to the formalization of arithmetical reasoning. We will only introduce the schema itself. The following expression schema represents the computation schema associated with the reasoning with lists:

$$listrec(l^L; a; x^L, y^A, z.f(x^L, y^A, z))$$

where superscripts L and A annotate terms that represent data.

The operational interpretation of *listrec* is defined by the following one-step reductions:

$$listrec(Nil; a; x^L, y^A, z.f(x^L, y^A, z)) \rightarrow_1 a$$

$$listrec(j_{h.r}; a; x^L, y^A, z.f(x^L, y^A, z)) \rightarrow_1 f(c_h, j_r, listrec(j_r; a; x^L, y^A, z.f(x^L, y^A, z)))$$

Next section presents an example of a program being an instance of the schema *listrec*.

6.5.1 The Smallest Element of a Non-Empty List

As an example of a recursive function on lists, we will implement the function to search for the smallest element in a non-empty list of natural numbers. The specification for the program is given by the following instructions based on the value of the input:

Nil: If the list is empty, the normal execution of the program is abandoned, and the result of the program is the number 100.

n.Nil: If the list contains one element n, the value returned as a result of the entire program is

that element.

n.m.rest: If the list contains more than one element, the following primitive operation is iterated: either m or n is chosen depending on whether $m \leq n$ or whether $n < m$.

Several operations have to be introduced in order to define the program. There are two primitive operations defined on list constants:

$$hd(j_{h.r}) \rightarrow_1 c_h \qquad tl(j_{h.r}) \rightarrow_1 j_r$$

which recover the first element of a list and a list without its first element respectively.

Let us assume that we have an operation *less* that returns the smaller of two integers, namely

$$less(k_m, k_n) \rightarrow_1 k_m \text{ if } m < n \qquad less(k_m, k_n) \rightarrow_1 k_n \text{ if } m \geq n.$$

Finally, we need a program that checks whether a list is empty. This program is defined as an instance of the *listrec* computation schema as follows:

$$is\_Nil(l^L) \equiv listrec(l^L; true; x^L, y^L, z.false).$$

Provided the above programming constructs, the program for searching for a minimum element in a list of integers is defined as follows:

$$MinL(l^L) \equiv listrec(case(is\_Nil(l^L); resultis(k_{100}); tl(l^L)); hd(l^L); x^D, y^L, z.less(x^D, z))$$

The computation of the program $MinL(l^L)$ is based on the following three kinds of conversion:

(1)

$$listrec(case(is\_Nil(Nil); resultis(k_{100}); tl(Nil)); hd(Nil); x^D, y^L, z.less(x^D, z))$$
$$\rightarrow_1 resultis(k_{100}) \rightarrow_1 k_{100}$$

(2)

$$listrec(case(is\_Nil(j_{n.Nil}); resultis(k_{100}); tl(j_{n.Nil})); hd(j_{n.Nil}); x^D, y^L, z.less(x^D, z))$$
$$\rightarrow_1 listrec(Nil; k_n; x^D, y^L, z.less(x^D, z)) \rightarrow_1 k_n$$

(3)

$$listrec(case(is\_Nil(j_{n.m.l}); resultis(k_{100}); tl(j_{n.m.l})); hd(j_{n.m.l}); x^D, y^L, z.less(x^D, z)) \rightarrow_1$$
$$less(k_n, listrec(case(is\_Nil(j_{m.l}); resultis(k_{100}); tl(j_{m.l})); hd(j_{m.l}); x^D, y^L, z.less(x^D, z)))$$

When the argument of the program *MinL* is the empty list, the program will abandon its normal execution and it will return the integer 100 as the final result. This "escaped" computation is expressed in the definition of *MinL* using the *resultis* construct and its interpretation is given in terms of the conversion (1) above. If its argument is a list consisting of one integer, then *MinL* will return this integer as the final result. This is expressed by the conversion (2). Finally, the conversion (3) expresses the recursive execution of a primitive operation of choosing between two given integers.

CHAPTER 7

CONCLUSION

This thesis has been about the top-level operational semantics of logical connectives and its significance in program development. We have argued that so called constructive type theories unnecessarily restrict reasoning in program development to be intuitionistic. As a consequence of this restriction, only purely functional programs can be developed in these theories. Even though functional programming is mathematically elegant, it lacks expressiveness gained by using escapes, coroutines, and other explicit control operations. In this thesis, we have shown how to construct *well-typed* programs with nonfunctional operations presented as purely declarative constructs.

To carry out this endeavour, we have turned to the "program" content of second-order encodings of logical connectives, and shown that it corresponds to an encoded CPS-translation on data-valued expressions of both structured and ground types. In other words, the operational interpretation of types induces a CPS-translation on data-valued expressions. The operational interpretation of logical connectives yields the proof methods provided by those connectives. Correspondingly, a CPS-translation on data yields programming methods. Hence, the operational interpretation of logical connectives yields direct operational semantics for pairs, injections, integer expressions, list expressions, functions from integers to integers, etc. It actually defines the language of classical proofs.

The problem with the second-order definitions of logical constants is that they are merely the *encodings* of their operational semantics. As a consequence, there isn't a clear distinction between different computations rules. To obtain the *actual* operational interpretation, and consequently the *actual* programs, we have removed the impredicativity of definitions of logical connectives by considering only the top-level contexts of proofs. We have presented

the *top-level operational interpretation* of logical connectives as a way of extracting the computational content of classical proofs. We have introduced a classical program development system, Computational Type Theory, formalizing this interpretation. We have demonstrated the equivalence between CTT and the computational extract of the class of classical Π^0_2 sentences by showing that CTT formalizes the operational interpretation of Π^0_2 sentences. We have extended this result to Peano Arithmetic by specializing CTT to a first-order theory, CTT+Nat, about natural numbers. We have defined several programming examples in classical type theories with booleans, natural numbers, and with the lists of natural numbers.

As we have already mentioned in the Introduction, another way of interpreting classical proofs as programs was introduced by Chetan Murthy in his thesis [Murthy90]. Murthy used the double-negation/A-translation to extract the program content of classical proofs. He showed that A-translation corresponds exactly to a continuation-passing-style translation on classical program extractions. This method of extraction is inherently higher-order and semantically complex. It does not distinguish between continuations and functions. It presents the continuation as a "normal" function that may perform a jump when applied. In other words, a continuation is introduced as an "imperative" add-on to a "declarative" language. In contrast, CTT distinguishes between functions and continuations. There is a separate syntax for local continuations, namely the syntactical abstraction facility, and the "*resultis*" construct for implementing non-local continuations, i.e., escapes. These constructs are purely declarative. In other words, in CTT continuations are given a sound declarative meaning as opposed to treating continuations as a powerful but unstructured control primitive, as it is done in [Murthy90]. Our extraction procedure is not an extension of the standard extraction procedure for constructive proofs [Murthy90]. The language of classical proofs is not "guessed" and then verified through translation, but *derived* from Prawitz+ encodings by applying to it a "top-level" analysis. Friedman's A-translation converts a classical proof of a Π^0_2 sentence into a constructive proof that computes evidence for that sentence, but it does *not* gives us a *direct* algorithmic content of classical rules.

Several systems have been built for automatic generation of purely functional programs. There is (1) Nuprl [Const86], already mentioned in the *Introduction* of this dissertation, which is based on Martin-Löf's intuitionistic theory of types; (2) the calculus of constructions with realizations (CCR) [Coquand85, Mohring89] which extends second-order λ-calculus with dependent types; and (3) system **PX** [HayNak88] which extracts computationally meaningful parts of proofs as LISP programs. Hayashi's **PX** uses logics of *partial terms* to obtain computational completeness (i.e., a general recursion operator). The use of a logic of partial terms is necessary since the usual formulations of predicate logic do not permit the formation of terms that do not always denote semantic values. As a consequence of this computational completeness, termination of programs cannot be proved by the proof-checker of **PX**. A proof that respects the separation of the termination condition from the rest of program verification without preserving termination of all well-formed programs, is not a mathematically "good proof": the process of extraction of an algorithm from a constructive proof is not checked for correctness in a logical framework.

The difference between Nuprl and **PX** is that the latter uses a framework of logic and the former uses a framework of a type theory to express computational meanings of constructive proofs. The calculus of constructions with realizations (CCR) [Mohring89] provides a logical framework for **PX** definitions of the notions of computationally uninformative or "type zero" propositions, extraction and realizability. Some of **PX** logical rules, like replacement, are coded and proved sound for program extraction in CCR.

As we pointed out before, Nuprl, **PX**, and CCR are systems for extracting purely functional programs. Constructive mathematics that uses the propositions-as-types principle restricts both the objects studied and the methods of proofs which may be applied. In systems like **PX** or CCR the constructivity is implicit in the restriction to intuitionistic logic. Hence, not all logical laws (e.g., the law of excluded middle, the proof by contradiction) can be used in the proofs of consistency of logical specifications, even when applied to well-determined concepts[1]. This, in turn, means that nonlocal control operations cannot be expressed in the

corresponding languages.

The computational logic of this thesis, in contrast with Nuprl, PX and CCR, does not follow the constructivist philosophy as to the nature of mathematics. In CTT the restrictions are placed only on the objects studied (e.g., functions, sets), but in such a way that all results have *direct* computational significance. CTT does not restrict the means of reasoning employed. For example, the concept of natural number is well-determined so that the application of laws of classical logic is justified (e.g., the law of excluded middle and consequently the proof by contradiction). The minimal requirement for constructivity is that all objects considered must be capable of being *presented* (e.g., functions are presented by rules, sets are presented by defining properties). The minimal requirement for a formal system of constructive mathematics is that existential assertions are witnessed by explicit solutions. CTT fulfills these requirements by considering only *completely presented sets*, formalized in CTT by the type of existential witness $\{A\}_B$. A set is completely presented when the evidence of its membership is carried by the members themselves: the evidence or *witnessing data* can be "read off" from the construction of the member [Feferman79]. That is, the *completeness* of the presentation assures explicit solutions to existential assertions. In CTT, the algorithms $f(y) \in A$, where $y \in B$, compute the evidence for the membership in the set defined by the property P in $\exists x \in A.P(x,y)$. This "works", i.e., $f(y)$ *is* the evidence for $\exists x \in A.P(x,y)$, since the type $\{A\}_B$ assures that $f(y)$ computes a concrete datum. More precisely, we have shown that $f(y)$ has to be a total, recursive operation.

Since programs can be "read off" from completely presented sets, the realizability interpretation is not required to extract the computational meaning from proofs. From the program development perspective, no constructive content is assigned to the part that deals

1. I.e., concepts whose definitions do not involve an implicit assumption of a completed totality of all objects. The questions of truth concerning such concepts are recognized as meaningful and definite (natural numbers, finite graphs, etc.) [Feferman84].

with the consistency of logical specifications: the goal is to be able to write an integer program using the knowledge that its argument is positive without demanding an extra argument at run-time justifying this fact.

CTT separates data types, types, and propositions. Such a separation was already used to extract programs from proofs in the Calculus of Constructions [Mohring89]. However, there is a fundamental difference in the generated program extractions in CTT and CCR, and it should be pointed out. There are three kinds of types in the Calculus of Constructions with Realizations (CCR): the type *Data* of data types, the type *Spec* of specifications, and the type *Prop* of propositions. The separation of *Data* from *Prop* allows to add one "exceptional" element in each data type without getting an inconsistent proof system. In other words, CCR extended each value domain with a bottom element. The problem with this solution is that it is not always clear if, e.g., the domain *nat* already includes a bottom element, or if we must add it ourselves. The problem becomes worse if we want to recover from run-time errors, instead of abandoning execution completely. For example, let the expression

$$\textbf{try}\ e_1\ \textbf{else}\ e_2$$

have the following semantics: first, e_1 is evaluated. If everything goes well, the value of e_1 will be the result of **try** expression, but if evaluation of e_1 fails, e_2 is evaluated instead. For example, the expression:

$$\textbf{try}\ (100\ \mathit{div}\ x)\ \textbf{else}\ 99$$

would evaluate to $[100/x]$ if x is not zero, and 99 otherwise. Such a valuation procedure is not monotonic and may cause problems when recursion is introduced to the language. The problem with CCR is simply that one has not (and could not, since the logic is intuitionistic) formalized from the start the fact that a program can fail (escape) and how to deal with it.

In summary, we have seen how to extract *directly* computational evidence from classical proofs. Extracted programs may return values of structured types, i.e., pair, sums, and functions from values to values. Our research has shown that the top-level operational interpretation of logical connectives provides a first-order account of the operational semantics of "classical" programs. It yields a language with explicit syntax for continuations, including

an explicit nonlocal control operator. From a program specification perspective, CTT is a classically founded program development system - a total-correctness reasoning system for non-functional programs. In comparison with [Murthy90], we have extended the result for Peano Arithmetic to other classical theories, e.g., to finite sequences of integers, trees.

In future work, CTT can be extended with a hierarchy of universes to allow introduction of proofs for sentences of classes higher than Π^0_2. As was pointed out in the Introduction, we cannot hope in general to extract the computational content of classically provable Σ^0_2 sentences. These are the sentences of the form $\exists x.\forall y.R(x,y)$. The reason, as we have discussed, lies in the use of classical reasoning to prove the proposition $\forall y.R(x,y)$. If one could prevent this, then the proof of that proposition would be purely functional. The introduction of universes will allow to treat functional types as ground types and hence disallow to analyze their internal structure. The first-order natural iteration, for example, does not yield all computable functions that might be constructed on natural numbers. This applies also to computation schemas associated with other first-order theories. Yet such constructions must count as a part of reasoning about numbers, lists, trees, etc.

To sum up, this thesis is a first step towards understanding total-correctness reasoning about programs with explicit control operators. The type-theoretic treatment of this subject in the thesis could be developed to useful programming languages. Another interesting task is to investigate the new logics which arise out of such an endeavour.

BIBLIOGRAPHY

[Aristotle28] Aristotle, *Metaphysica*, Oxford Translation, ed. by W.D. Ross, Vol VIII, Oxford, 28.

[Beeson85] Beeson, M.J., *Foundations of Constructive Mathematics, Metamathematical Studies*, Springer-Verlag, 1985.

[Benacerraf73] Benacerraf, P., Mathematical Truth, *The Journal of Philosophy*, **70**, 47-73.

[Bochenski59] Bocheński, J. M., *A Precis of Mathematical Logic*, D. Reidel Publishing Company, Dordrecht-Holland (1959).

[Bochenski61] Bocheński, J. M., *A History of Formal Logic*, University of Notre Dame Press, (1961).

[Boehner52] Boehner, P., *Medieval Logic*, The University of Chicago Press, Chicago 1952.

[Brouwer52] Brouwer, L.E.J., Historical background, principles and methods of intuitionism, *South African Journal of Science*, **49**, 139-46.

[deBruijn72] de Bruijn, N. G., Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indag. Math.*, **34,5**, (1972), 381-392.

[Carnap37] Carnap, R., *The Logical Syntax of Language*, Routledge & Kegan Paul Ltd, London, 1937.

[Constable85] Constable, R., The semantics of evidence, Tech. Report. TR 85-684, Cornell University, Dept. of Computer Science, Ithaca, NY, 1985.

[Constable86] Constable, R. et al., Implementing Mathematics in the NuPrl System, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Coquand85] Coquand, Th., *Une Theórie des Constructions*, these de 3eme cycle, Paris VII (1985).

[Coquand86] Coquand, Th., An analysis of Girard's paradox, *First IEEE Symposium on Logic in Computer Science*, Boston (June 1986), 227-236.

[CoqHu86] Coquand, Th., Huet, G., The Calculus of Constructions, *Information and Control*, 76, 1988.

[CoqHu85] Coquand, Th., Huet, G., Constructions: A Higher Order Proof System for Mechanizing Mathematics, *EUROCAL85*, Linz, Springer-Verlag LNCS 203 (1985).

[Dummett59] Dummett, M., *Truth*, in [Dummett78], 1-24. Originally published in *Proceedings of the Aristotelian Society*, Vol. LIV.

[Dummett73] Dummett, M., The Philosophical Basis of Intuitionistic Logic, in [Dummett78]. Originally published in *Logic Colloquium '73*, eds. Rose, Shepherdson, Bristol, July 1973, 5-40.

[Dummett77] Dummett, M., *Elements of Intuitionism*. Clarendon Press, Oxford 1977.

[Dummett78] Dummett, M., *Truth and Other Enigmas*, Harvard University Press, Cambridge, Mass., 1978.

[Dummett81] Dummett, M., *The Interpretation of Frege's Philosophy*, Harvard University Press, Cambridge, Mass., 1981.

[Feferman60] Feferman, S., Arithmetization of metamathematics in a general setting, *Fundamenta Mathematicae*, 49, 35-92.

[Feferman79] Feferman, S., Constructive theories of functions and classes, *Logic Colloqium 78*, North-Holland, Amsterdam, 159-224.

[Feferman83] Feferman, S., Between Constructive and Classical Mathematics, *Logic Colloqium 78*, Aachen 1983, LNCS, Springer-Verlag, 143-162.

[FFED86] Felleisen, M., Friedman, D., Kohlbecker, E., Duba, B., Reasoning with continuations, in *Proceedings of the First Annual ACM Symposium on Principles of Programming Languages*, 180-190, 1986.

[Fel87] Felleisen, M., The Calculi of λ_v - CS conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages, PhD Thesis, Indiana University, 1987.

[Field72] Field, H., Tarski's Theory of Truth, *The Journal of Philosophy* **69**, 347-75.

[Fil89] Filinski, A., Declarative Continuations and Categorical Duality, Ms. Thesis, Dept. of Computer Science, University of Copenhagen, 1989.

[Fis72] Fischer, M. J., Lambda-calculus schemata, in *Proceedings of the ACM Conference on Proving Assertions about Programs*, vol. 7 of *Sigplan Notices*, pp. 104-109, 1972.

[Friedman78] Friedman, H., Classically and intuitionistically provable recursive functions, in *Higher Set Theory* (Scott, D. S. and Muller, G. H., ed.), vol. 699, *Lecture Notes in Mathematics*, 21-28, Springer-Verlag, 1978.

[GLT89] Girard, J.Y., Lafont, Y., Taylor, P., *Proofs and Types*, Cambridge University Press, 1989.

[Girard70] Girard, J.Y., 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, *Proceedings of the Second Scandinavian Logic Symposium*, Ed. J.E. Fenstad, North Holland, 1970.

[Gentzen35] Gentzen, G., Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift*, **39**, 176-210, 405-431

[Gödel47] Gödel, K., What is Cantor's Continuum Problem?, *American Mathematical Monthly* **54**, 515-25.

[Griffin90] Griffin, T. G., A formulae-as-types notion of control, in *Conference Record of the 17th Annual ACM Symposuim on Principles of Programming Languages*, 1990.

[HayNak88] Hayashi, S., Nakano, H., PX: A computational logic. Preprint RIMS-573, Research Institute for Mathematical Sciences, Kyoto University; MIT Press, Cambridge, Mass., 1988.

[Heyting71] Heyting, A., *Intuitionism: an Introduction*, North-Holland, Amsterdam, 1971.

[Huet87] Huet, G., Induction Principles Formalized in the Calculus of Constructions, *TAPSOFT87*, Springer-Verlag, Pisa, LNCS 249, 276-286.

[Huet90] Huet, G. (ed.), *Logical Foundations of Functional Programming*, Addison-Wesley, 1990.

[JST] *The Material Logic of John of St.Thomas*, Y.R. Simon, J.J. Glanville, G.D. Hollenhorst (trans.), The University of Chicago Press, 1955.

[Kleene52] Kleene, S.C., *Introduction to Metamathematics*, Van Nostrand, Princeton, 1952.

[Kleene67] Kleene, S.C., *Mathematical Logic*, John Wiley & Sons, 1967.

[Kolmogorov67] Kolmogorov, A. N., On the principle of excluded middle, in *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931* (J. van Heijenoort, ed.), pp. 414-437, Cambridge, Mass., Harvard Univerisy Press, 1967.

[Kleene67] Kleene, S.C., *Mathematical Logic*, John Wiley & Sons, 1967.

[Kolmogorov67] Kolmogorov, A. N., On the principle of excluded middle, in *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931* (J. van Heijenoort, ed.), pp. 414-437, Cambridge, Mass., Harvard Univerisy Press, 1967.

[Kotarbinski63] Kotarbiński, T., Spór o desygnat (A Controversy Concerning the Concept of Designatum). *Prace Filozoficzne* XVIII (1963), 1.

[Kreisel65] Kreisel, G., Mathematical logic, in: *Lectures on modern mathematics*, vol. III, ed., Saaty (N.Y., 1965) 95-195.

[Kreisel70] Kreisel, G., A survey of proof theory II, *Proceedings of the Second Scandinavian Logic Symposium*, University of Oslo, 1970.

[Kreisel71] Kreisel, G., Perspective in the Philosophy of Pure Mathematics, in *Logic, Methodology and Philosophy of Science* IV, eds. Suppes, Henkin, Moisil, 1971.

[KrKriv71] Kreisel, G., Krivine, J.L., *Elements of Mathematical Logic*, North-Holland, Amsterdam, 1971.

[Kreisel85] Kreisel, G., Mathematical Logic: Tool and Object Lesson for Science, *Synthese* **62**, 139-51.

[MacLane71] Mac Lane, S., *Categories for Working Mathematician*, Springer-Verlag (1971).

[MS76] Milne, G., J., Strachey, C., *A Theory of Programming Language Semantics*, Chapman and Hall, 1976.

[Murthy90] Murthy, C., Extracting Constructive Content from Classical Proofs, PhD Thesis, Cornell University, Dept. of Computer Science, Ithaca, NY, 1990.

[Murthy91] Murthy, C., An Evaluation Semantics for Classical Proofs, Tech. Report, TR 91-1213, Cornell University, Dept. of Computer Science, Ithaca, NY, 1991.

[Martin-Löf72] Martin-Löf, P., About Models for Intuitionistic Type Theories and the Notion of Definitional Equality, paper read at the Orléans Logic Conference, September 1972.

[Martin-Löf82] Martin-Löf, P., Constructive Mathematics and Computer Programming, *Proc. Sixth International Congress for Logic, Methodology and Philosophy of Science*, North-Holland, Amsterdam, 1982, 153-175.

[Martin-Löf85] Martin-Löf, P., Truth of a Proposition, Evidence of a Judgement, Validity of a Proof, talk given at the workshop Theories of Meaning organized by Centro Fiorentino di Storia e Filosofia della Scienza at the Villa di Mondeggi near Florence, June 1985.

[Martin-Löf85*a*] Martin-Löf, P., On the Meanings of Logical Constants and the Justifications of the Logical Laws, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Technical Report 2., Universita di Siena, 1985.

[Mendelson79] Mendelson, E., *Introduction to Mathematical Logic*, ed. D.Van Nostrand Company, Litton Ed. Pub., 1979.

[Moody53] Moody, E.A., *Truth and Consequence in Mediaeval Logic*, North-Holland, Amsterdam, 1953.

[Mohring89] Paulin-Mohring, C., Extracting Fω's Programs from Proofs in the Calculus of Constructions, *The Proceedings of the 16th Symposium on Programming Languages*, Austin, Texas, 1989.

[Mycielski89] Mycielski, J., The Meaning of Pure Mathematics, *Journal of Philosophical Logic* **18**, pp. 315-320, 1989.

[Nordström87] Nordström, B., Terminating General Recursion, Dept. of Computer Science, University of Göteborg/Chalmers, Sweden, 1987.

[NoPe83] Nordström, B., Petersson, K., Types and specifications, in *Proceedings of IFIP 83*, pp. 915-920, R.E.A. Mason (ed.), Elsevier Science Publishers 1983.

[NoSm84] Nordström, B., Smith, J., Propositions, Types, and Specifications in Martin-Löof's Type Theory, **BIT**, 24(3), 288-301, October 1984.

[Nordström87] Nordström, B., Terminating General Recursion, Dept. of Computer Science, University of Göteborg/Chalmers, Sweden, 1987.

[NoPe83] Nordström, B., Petersson, K., Types and specifications, in *Proceedings of IFIP 83*, pp. 915-920, R.E.A. Mason (ed.), Elsevier Science Publishers 1983.

[NoSm84] Nordström, B., Smith, J., Propositions, Types, and Specifications in

Martin-Löof's Type Theory, **BIT**, 24(3), 288-301, October 1984.

[Petersson82] Petersson, K., A Programming System for Type Theory, PMG Memo 21, Chalmers University of Technology, S-412 96 Göteborg, 1982.

[Pelc81] Pelc, J., The place of the philosophy of language, *Contemporary Philosophy. A new survey.*, Martinus Nijhoff Publishers, Vol. 1, pp. 11-34.

[Prawitz65] Prawitz, D., *Natural Deduction*, Almqvist and Wiksell, Stockholm, 1965.

[Prawitz70] Prawitz, D., Ideas and results in proof theory, *Proceedings of the Second Scandinavian Logic Symposium*, ed. J.E. Fenstad, North Holland, 1970.

[RC86] Rees, J., Clinger, W., The revised report on the algorithmic langugage scheme, *SIGPLAN Notices*, vol. 21, no. 12, 37-79, 1986.

[Sintonen82] Sintonen, M., Realism and Understanding, *Synthese* **52**, 347-78.

[Statman81] Statman, R., Number theoretic functions computable by polymorphic programs, *22nd Symposium on Foundations of Computer Science*, 279-282, IEEE, 1981.

[StarWad74] Starchey, C., Wadsworth, C.P., Continuations: a Mathematical Semantics for Handling Full Jumps, Tech. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

[Sundholm83] Sundholm, G., Constructions, Proofs and The Meanings of Logical Constants, *Journal of Philosophical Logic* **12**, 151-72.

[Szabo69] Szabo, M.E., *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, 1969.

[Tait67] Tait, W.W., Intensional interpretation of functionals of finite type I, *The Journal of Symbolic Logic* **32** (1967), 198-212.

[Tait83] Tait, W.W., Against Intuitionism: Constructive Mathematics Is a Part of Classical Mathematics, *The Journal of Philosophy* **12**, 173-195.

[Tait86] Tait, W.W., Truth and Proof: The Platonism of Mathematics, *Synthese* **69**, 341-370.

[Tarski44] Tarski, A., The Semantic Conception of Truth, *Philosophy and Phenomenological Research*, Vol. IV, March 1944, 341-75.

[PM] Whitehead, A.N., Russell, B., *Principia Mathematica*, Cambridge University Press, 1911.

[Turner76] Turner, D. A., SASL Language Manual, St. Andrews University Technical Report, December 1976.

[Weinstein83] Weinstein, S., The Intended Interpretation of Intuitionistic Logic, *Journal of Philosophical Logic* **12**, 261-70.

BIOGRAPHY

I was born in Poland in 1959. I received my primary and secondary education in Wroclaw, Poland. After the high school graduation I started my studies at the Technical University of Wroclaw in the Institute of Information Systems. I graduated in 1983 with the Master's degree in computer science. In 1984 I started the Ph.D program in the Department of Computer Science and Engineering at the Oregon Graduate Institute in Beaverton, Oregon. I received Ph.D. in computer science in 1992. My Ph.D. research concentrated on automated program development and programming language design. Since 1989 I have been working as a consultant for the AT&T Bell Laboratories. My job assignments have been connected with the analysis of the digital networks and telecommunications software development.