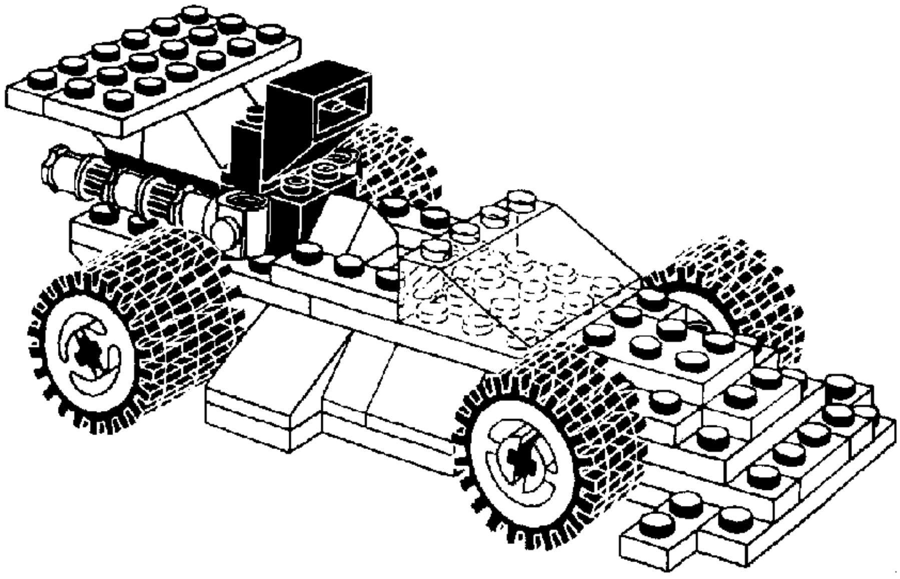
Proof by Calculation



Doctoraalscriptie *Wiskunde & Informatica*

385

Martijn Oostdijk

(begeleiding H.P. Barendregt)

1996

Contents

Chapter 1

Introduction

1.1 What is science?

It is far beyond the scope of this paper to fully answer the question in the title of this section. But there are some things that we must say about science in order to understand the need for automated proof verification which is the subject of this thesis. It is difficult to catch all the activities of all scientists in one definition, but the least we can say is that science has something to do with finding valid assertions concerning some reality. Therefore it is of importance to scientists to have methods for verifying the validity of assertions. Physicists are dependent on the so-called *empirical method*.

- Given an assertion: Try to conceive experiments which will succeed in case the assertion holds. (the difference between success and failure of an experiment has to be measurable).

- An assertion is assumed to be valid if none of the experiments fail.

This method has one drawback. It is only useful to determine if assertions do not hold. When none of the experiments fail, we are still not sure of the truth of the statement. Maybe someone can think of a better experiment which will clearly show that our statement is false.

Mathematicians investigate an abstract reality consisting of abstract objects. For example the discipline of geometry is concerned with points and lines. No one has ever seen a geometrical point, let alone experimented with one. Mathematical objects cannot be subjected to experimenting. Therefore the above empirical method is not applicable. Instead we use the *deductive method* (or *axiomatic method*) formulated by Aristotle.

- Design a formal *language*, containing primitive terms, to express the assertions.

- Introducing new concepts using definitions.

- Conceive a set of *axioms*, i.e. assertions that are valid by definition. These axioms describe the meaning of the abstract objects in our reality. In geometry, for example, we could state that two different points determine exactly one line.
- Conceive *deduction rules* which transform assertions into new assertions.
- An assertion is called derivable if it can be retrieved by repeated application of the deduction rules, starting with axioms. A transcription of such a derivation is called a *proof*.

An assertion is assumed to be valid if it is derivable in the deduction system.

There is a drawback to this method as well. We can only describe abstract reality. There are objects in our real physical world that look a lot like points and lines. We have however no guarantee that they will behave in any way like the points and lines in our abstract world. The validity of derived assertions, relative to the (approximations of) points and lines in the physical world, depends greatly on the validity of the axioms and deduction rules.

1.2 Proof verification and proof inference

Given a statement A, the question: "Does A hold?" is replaced by the question: "Is there a proof for A?". Given also a proof-object p, another question we could ask is: "Is p a proof of A?". The first question seems harder. In fact we have:

- When given a proposition and a candidate proof, it is easy to check whether the proof demonstrates the truth of the proposition. This task is called *proof checking*.
- When given a proposition, it is not always easy to find an argument that proves it. This task is called *theorem proving*.

These facts have been known since Aristotle. The first one is easy indeed. We only have to check whether the steps in the proof are valid applications of the various deduction rules and whether the conclusion is the proposition we have to prove. Church formalized the notion of the impossibility of theorem proving in 1936.

1.3 Automated proof verification

So proof verification is simple in theory. But in practice checking a real proof written by real mathematicians is a hard task. Often these proofs are not written in a formal language. Mathematicians often use terms like "trivial", "the reader can verify that" etc. Moreover, proofs tend to get very large and detailed. Proof verification is simple in principle but that does not make it an easy task for humans. Automated proof verification could be useful here.

Since proof verification is simple in principle, it can be done by a machine. Yet, this is only possible if the propositions and proofs are written down in a formal way. The necessity for formal notation of proofs forms a new problem for human mathematicians. The object of automated proof verification is to help mathematicians. Still, the price we pay is that proofs need to be very precise and formal.

1.4 Interactive proof development

In order to overcome this problem yet another computer program is written: A *proof development system*. Mathematicians can use this program to develop proofs interactively. To find a proof, the mathematician provides the program with tactical information which is comparable to the steps he would take in an informal argument. The program takes care of the details. At the end of an interactive session the program yields a formal proof-object for the proposition that we wanted to prove. This proof-object can then be sent to other mathematicians who can test its validity using trusted automatic proof checkers.

So the human mathematician writes his proof in a high level language comparable to high level programming languages while the machine builds a formal proof-object in a language comparable to assembly-language.

1.5 Calculations and algorithms

Sometimes calculations play an important role in mathematical proofs. The actual calculation itself is not considered to be part of the argument. Yet, the outcome of a calculation may be used to demonstrate for example the existence of some mathematical object. When attempting to formalize a proof in which calculations play a role we are sometimes forced to perform the actual calculations ourselves. On the low level of proof-objects this may lead to large proof-objects. We do not want the calculations to become part of the proof.

If our proof development system is based on *type theory* a solution can be found. The solution to this problem is to view the calculations as algorithms and to prove these algorithms correct i.e. the algorithm does the calculations we need. We can now let the machine do the calculations (computers are very good at this) and this will no longer show in the size of the proof-object.

An application of this idea is to view *predicates* as syntactical objects and generate an algorithm that decides the predicate. In general this is not possible, but we will restrict the class of admissible predicates. We can then generate a correctness proof of the algorithm relative to the predicate. This can be used to automatically prove propositions without generating huge proof-objects.

1.6 Outline of the rest of this thesis

We will first discuss a theoretical framework about proof checking and proof development in general and proof checking based on type theory. This will be done in chapter 2. Chapter 3 is about the two-level approach discussed in the last section. The specification of the automatic theorem prover is given in chapter 4 while details about an implementation in the functional programming language Clean are given in chapter 5.

Chapter 2

Mathematics and Computer Science

In this chapter we will introduce some basic theory on foundations. Computer science deals with programs and calculations while mathematics is about proofs and propositions. We will say something about *recursion theory* which is the foundation of computer science and about *logic* which is the foundation of mathematics. Then we will give a unified implementation of both by means of type theory through the well known *propositions-as-types* semantics by Curry and Howard.

2.1 Recursion theory

Recursion theory is a theory about functions. It tries to answer questions about computability of functions. A first attempt to characterize the class of computable functions led to the class of *primitive recursive functions.* This class turned out to be too small. There are computable total functions that are not primitive recursive.

Yet, a lot of useful functions are primitive recursive, and it is not hard to construct a programming language in which a programmer can specify only the primitive recursive functions. Programs written in this language have nice properties, for example it holds that all programs are total.

We will define the class of primitive recursive functions on natural numbers. The class of partial recursive functions will also be defined. Moreover, we will recapitulate some theorems from recursion theory without proof.

Definition 2.1.1 *The initial functions are the following: The function* $Z, S : \mathbb{N} \to \mathbb{N}$ *defined by*

$$\begin{aligned} Z(x) &= 0 \\ S(x) &= x + 1 \end{aligned}$$

and for all $n, k \in \mathbb{N}$ with $0 \leq k \leq n$ the function P_n^k defined by

$$P_n^k(x_1, \ldots, x_n) \quad = \quad x_k$$

The function P_0^0 is defined as the constant 0.

Definition 2.1.2 *The class of primitive recursive functions is the smallest class of functions for which it holds that:*

1. *The initial functions belong to the class.*
2. *If $f : \mathbb{N}^n \to \mathbb{N}$ belongs to the class and $g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}$ belong to the class, then $h : \mathbb{N}^m \to \mathbb{N}$, defined by*

   $$h(x_1, \ldots . x_m) \quad = \quad f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

   belongs to the class. We write: $h = \mathrm{Comp}[f, g_1, \ldots, g_n]$.
3. *If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ belong to the class, then $h : \mathbb{N}^{n+1} \to \mathbb{N}$, defined by*

   $$\begin{aligned} h(0, x_1, \ldots, x_n) &= f(x_1, \ldots, x_n) \\ h(y+1, x_1, \ldots, x_n) &= g(h(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n) \end{aligned}$$

   belongs to the class. We write: $h = \mathrm{Prim}[f, g]$.

Many useful functions turn out to be primitive recursive:

Example 2.1.3 *The following functions are all primitive recursive:*

$$\begin{aligned} \mathrm{Add} &= \mathrm{Prim}[P_1^1, \mathrm{Comp}[S, P_3^1]] \\ \mathrm{Mult} &= \mathrm{Prim}[Z, \mathrm{Comp}[\mathrm{Add}, P_3^1, P_3^3]] \\ \mathrm{sg} &= \mathrm{Prim}[P_0^0, \mathrm{Comp}[S, \mathrm{Comp}[Z, P_2^2]]] \\ \overline{\mathrm{sg}} &= \mathrm{Prim}[\mathrm{Comp}[S, P_0^0], \mathrm{Comp}[Z, P_2^2]] \end{aligned}$$

But not all computable functions are primitive recursive. The modified Ackermann function, for example, is clearly computable but it does not belong to the class of primitive recursive functions.

Proposition 2.1.4 *The modified Ackermann function A defined by*

$$\begin{aligned} A(0, x) &= 2x \\ A(k+1, 0) &= 1 \\ A(k+1, x+1) &= A(k, A(k+1, x)) \end{aligned}$$

is not primitive recursive.

A new scheme called *minimalization* is added. This results in the class of partial recursive functions. The objects in this class are partial functions since the new scheme does not guarantee the totality of functions. Kleene proved that recursively defined functions, like the Ackermann function A, are partial recursive.

Definition 2.1.5 *The class of partial recursive functions is the smallest class of partial functions for which it holds that:*

1. *The initial functions belong to the class.*
2. *If $f : \mathbb{N}^n \to \mathbb{N}$ belongs to the class and $g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}$ belong to the class, then $h : \mathbb{N}^m \to \mathbb{N}$, defined by*
$$h(x_1, \ldots, x_m) \quad = \quad f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$
belongs to the class. We write: $h = \mathrm{Comp}[f, g_1, \ldots, g_n]$.
3. *If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ belong to the class, then $h : \mathbb{N}^{n+1} \to \mathbb{N}$, defined by*
$$\begin{aligned} h(0, x_1, \ldots, x_n) &= f(x_1, \ldots, x_n) \\ h(y+1, x_1, \ldots, x_n) &= g(h(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n) \end{aligned}$$
belongs to the class. We write: $h = \mathrm{Prim}[f, g]$.
4. *If $f : \mathbb{N}^{n+1} \to \mathbb{N}$ belongs to the class, then $h : \mathbb{N}^n \to \mathbb{N}$, defined by*
$$h(x_1, \ldots, x_n) \quad = \quad \mu y[f(y, x_1, \ldots, x_n) = 0]$$
belongs to the class. We write $h = \mathrm{Min}[f]$.

We now define primitive recursiveness of predicates.

Definition 2.1.6 *Let $P \subseteq \mathbb{N}^n$ be a predicate. The characteristic function χ_P is defined as follows: For all $\vec{x} \in \mathbb{N}^n$:*

$$\chi_P(\vec{x}) = \begin{cases} 1 & \textit{if} \quad P(\vec{x}) \\ 0 & \textit{if} \quad \neg P(\vec{x}) \end{cases}$$

We call P primitive recursive if χ_P is primitive recursive. We call P recursive if χ_P is partial recursive (note that χ_P is total by definition).

Lemma 2.1.7
1. *There exist primitive recursive functions $\langle\ \rangle : \mathbb{N}^* \to \mathbb{N}$ for encoding and $(\ )_0, (\ )_1, \ldots : \mathbb{N} \to \mathbb{N}$ for decoding of finite sequences.*
2. *The partial recursive functions can be encoded as natural numbers. We write φ_e for the partial recursive function with code number e.*

Theorem 2.1.8 *There exists a primitive recursive predicate $T \subseteq \mathbb{N}^3$ (Kleene's T-predicate) and a primitive recursive function $U : \mathbb{N} \to \mathbb{N}$ (outcome) such that*

$$\varphi_e(\vec{x}) = n \iff \exists_{z:\mathbb{N}}[T(e, \langle\vec{x}\rangle, z) \ \&\ U(z) = n]$$

Proof:

See [11].

□

2.2 Mathematical logic

As was said in chapter 1 mathematicians use the deductive method to test the validity of statements. We will describe this process for the theory of arithmetic.

First we have to define the language in which propositions about arithmetic can be expressed. This is necessary since natural languages are too rich and ambiguous. Next we have to single out some propositions which will serve as axioms. Finally we have to give deduction rules which transform propositions into other propositions. The deduction rules, which will be dealt with here, do not specifically apply to arithmetic. They are the rules of *first order intuitionistic logic.*

The language is defined inductively out of basic propositions using connectives and quantifiers. Of course the language must have constructions to describe the mathematical objects that interest us (numbers, functions, triangles, equations etc.) The objects that are allowed here are *relations*, *functions* and *constants* over some set called the *domain.* Take the following structure: $\mathcal{N} = \langle \mathbb{N}, <, S, +, \cdot, 0, 1 \rangle$. The domain is $\mathbb{N}$ (the set of natural numbers) and we are interested in the binary relation $<$ (less than) and binary functions for addition and multiplication. Two constants, 0 and 1, play a special role. We therefore use symbols for them.

Definition 2.2.1 *The language L consists of formulas constructed as follows:*

1. *Terms are built inductively as follows:*
   (a) *Each variable x is a term*
   (b) *The two constants $\underline{0}$ and $\underline{1}$ are terms*
   (c) *If T is a term, then so is $f(T)$*
   (d) *If T_1 and T_2 are terms, then so are $T_1 \oplus T_2$ and $T_1 \odot T_2$*
2. *Formulas are built inductively from terms:*
   (a) *If T_1 and T_2 are terms, then so are $T_1 \sqsubset T_2$ and $T_1 = T_2$*
   (b) *The symbol $\bot$ (falsum) is a formula. If A is a formula, we write $\neg A$ for $A \to \bot$.*
   (c) *If A and B are formulas, then so are $A \,\&\, B$, $A \vee B$ and $A \to B$*
   (d) *If A is a formula and x a variable, then $\forall x[A]$ and $\exists x[A]$ are formulas.*

To define the language L, we do not need to know the meaning of the various symbols. The language we define is not specific for $\mathcal{N}$ alone. We used symbols like f and $\oplus$ to denote the functions. These symbols can be interpreted as S and $+$. This is convenient because we can talk about formulas on a higher level. For example the formula $\varphi = \exists y \forall x[(y \sqsubset x) \vee (y = x)]$ can be interpreted in $\mathcal{N}$ (where it is valid) and also in the structure $\mathcal{Z} = \langle \mathbb{Z}, <, S, +, \cdot, 0, 1 \rangle$ (where φ is not valid).

Now we have a language in which we can express propositions. The deductive method requires us to give axioms for $\mathcal{N}$. Axioms are specifically chosen formulas in L.

Definition 2.2.2 *As axioms of $\mathcal{N}$ we choose the following: First we have axioms describing equality:*

$$\begin{array}{l}
\forall_x[x = x], \\
\forall_x\forall_y[x = y \to y = x], \\
\forall_x\forall_y\forall_z[x = y \;\&\; y = z \to x = z], \\
\forall_x\forall_y[x = y \to f(x) = f(y)], \\
\forall_x\forall_y\forall_z\forall_v[(x = y \;\&\; z = v) \to x \oplus z = y \oplus v], \\
\forall_x\forall_y\forall_z\forall_v[(x = y \;\&\; z = v) \to x \odot z = y \odot v]
\end{array}$$

Next we have, for each of the functions, axioms describing important properties of that function. For the successor function we demand that it never yields zero, that it is injective and that it is is 'almost' surjective:

$$\begin{array}{l}
\forall_x[\neg(f(x) = \underline{0})], \\
\forall_x\forall_y[f(x) = f(y) \to x = y], \\
\forall_x[\neg(x = \underline{0}) \to \exists_y[f(y) = x]]
\end{array}$$

The axioms describing addition and multiplication are very similar to their primitive recursive definitions:

$$\begin{array}{l}
\forall_x[x \oplus \underline{0} = x], \\
\forall_x\forall_y[x \oplus \mathsf{S}(y) = \mathsf{S}(x \oplus y)], \\
\forall_x[x \odot \underline{0} = \underline{0}], \\
\forall_x\forall_y[x \odot f(y) = (x \odot y) \oplus x]
\end{array}$$

The ordering is defined using equality and addition.

$$\begin{array}{l}
\forall_x\forall_y[x \sqsubset y \to \exists_z[x \oplus f(z) = y]], \\
\forall_x\forall_y[\exists_z[x \oplus f(z) = y]] \to x \sqsubset y]
\end{array}$$

The deductive method requires us to give deduction rules for the formulas of L. Let A be a formula of L. We call A derivable (notation $\Gamma_{\mathcal{N}} \vdash A$) if it can be derived from the axioms by finite application of the following rules:

Definition 2.2.3 *The deduction rules for first order intuitionistic logic are defined as follows:*

(start-rule)	$\dfrac{A \in \Gamma}{\Gamma \vdash A}$
($\perp$-rule)	$\dfrac{\Gamma \vdash \perp}{\Gamma \vdash C}$
(& -intro)	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$
(& -elim)	$\dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A} \qquad \dfrac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}$
($\vee$-intro)	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$
($\vee$-elim)	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$
($\rightarrow$-intro)	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$
($\rightarrow$-elim)	$\dfrac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$
($\forall$-intro)	$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x[A(x)]}$
($\forall$-elim)	$\dfrac{\Gamma \vdash \forall x[A(x)]}{\Gamma \vdash A(t)}$
($\exists$-intro)	$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x[A(x)]}$
($\exists$-elim)	$\dfrac{\Gamma \vdash \exists x[A(x)] \quad \Gamma, A(x) \vdash B}{\Gamma \vdash B}$

The formulas in our language L are only syntax. We have some semantical notion of the basic formulas as they can be interpreted in a structure of appropriate key, for example in the structure $\mathcal{N}$, but not for compound formulas. There are various interpretations that give meaning to the first order formulas based on meaning of the basic formulas. The truth-table semantics, in which a formula is assigned the meaning 0 or 1, is perhaps the best known method. We will discuss the so-called intuitionistic semantics due to Heyting briefly here.

They are based on the following insight.

Take a look at the $\rightarrow$-intro rule. We can read this rule as: "A proof of $A \rightarrow B$ is a method to transform any proof of A into a proof of B." In the same manner we can read the &-intro rule as: "A proof of A & B is an ordered pair of a proof of A and a proof of B."

The notions of method and ordered pair are well known in computer science. Similar interpretations can be given for the other connectives. This strong interpretation of the connectives leads us to view propositions as types and proofs as programs.

2.3 Type theory

Lambda calculus (with types) can be used as an implementation for proof-checkers. The key idea behind implementations like these, is the *Curry-Howard-de Bruijn isomorphism* between types and propositions as described above. In this section we will look at different systems and their expressive power both as a computation model and as a formalism to express mathematics.

2.3.1 Type free lambda calculus

Lambda calculus was intended as a computation model. The idea was to capture two important notions from function theory: abstraction and application. Abstraction is used to construct function terms which can be applied to argument terms in order to get a result. Lambda terms are built from variables by abstraction and application. See [1] for a detailed description.

Definition 2.3.1 *The type free lambda calculus is defined as follows:*

1. *Let V be a set of variables. The set of lambda terms (Λ) is defined as follows:*

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \Lambda \\ M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda \\ M \in \Lambda, x \in V & \Rightarrow & (\lambda x.M) \in \Lambda \end{array}$$

2. *The reduction relation is given by:*

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

we write $=_\beta$ for the reflexive transitive closure of $\rightarrow_\beta$.

What functions (on natural numbers) can be represented in the type-free lambda calculus? First, we need a way to denote natural numbers and their functions:

Definition 2.3.2 *Natural numbers and functions on natural numbers are represented in lambda calculus as follows:*

1. *For each* $n \in \mathbb{N}$ *the Church numeral* c_n *is defined by* $c_n \equiv \lambda xf.f^n x$.
2. *Let* $f : \mathbb{N}^k \to \mathbb{N}$ *be a function.* f *is said to be* λ*-definable if there exists a lambda term* F*, such that*

$$Fc_{n_1} \cdots c_{n_k} = c_{f(n_1,\ldots,n_k)}$$

   for all $n_1, \ldots, n_k \in \mathbb{N}$

Recursive functions are implemented in the type-free lambda-calculus using the following Y-combinator.

Theorem 2.3.3 *There exists a closed* λ*-term* $\mathbf{Y}$ *such that for all* $F \in \Lambda$*:*

$$\mathbf{Y}F =_\beta F(\mathbf{Y}F)$$

Proof:

Take $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. *(For details see [1].)*

□

Theorem 2.3.4 *All recursive functions are* λ*-definable.*
Proof:

1. *The initial functions are* λ*-definable.*
2. *The class of* λ*-definable functions is closed under composition (using application).*
3. *The class of* λ*-definable functions is closed under primitive recursion and minimalization (using theorem 2.3.3).*

(For details see [1].)

□

2.3.2 Simple types

The system $\lambda\!\to$ (the simply typed lambda-calculus) is due to Church. It is an extension of the untyped lambda calculus. Terms are annotated with types and we define type assignment in such a way that terms can only be applied to arguments of the appropriate types. The type system is called simple because we only allow one type constructor for the construction of function types.

Definition 2.3.5 *The system* $\lambda\!\to$ *is defined as:*

1. *Types:* $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$
2. *Pseudo-terms:* $\Lambda_\mathbb{T} = V \mid \Lambda_\mathbb{T}\Lambda_\mathbb{T} \mid \lambda V : \mathbb{T}.\Lambda_\mathbb{T}$
3. *Contraction rule:* $(\lambda x : A.M)N \to_\beta M[x := N]$

4. *Type assignment:*

(start-rule)	$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}$$
($\to$-intro)	$$\frac{\Gamma, x:A\vdash M:B}{\Gamma\vdash(\lambda x:A.M):(A\to B)}$$
($\to$-elim)	$$\frac{\Gamma\vdash M:(A\to B)\quad \Gamma\vdash N:A}{\Gamma\vdash(MN):B}$$

We can extend the definition of Church numerals to $\lambda\!\!\to$. The definition of λ-definability is also extended. Of course we need to annotate the abstraction variables with types. We single out a variable α for this purpose.

Definition 2.3.6 *Let $\alpha\in\mathbb{V}$ be a fixed type variable.*

1. *Define the following terms*

$$\begin{array}{rcl}
\mathbf{N} & \equiv & \alpha\to(\alpha\to\alpha)\to\alpha \\
c_n & \equiv & \lambda x:\alpha.\lambda f:\alpha\to\alpha.f^n x \\
\mathbf{0} & \equiv & c_0 \\
\mathbf{S} & \equiv & \lambda n:\mathbf{N}.\lambda x:\alpha.\lambda f:\alpha\to\alpha.f(nxf)
\end{array}$$

2. *Let $f:\mathbb{N}^k\to\mathbb{N}$ be a function, f is said to be $\lambda\!\!\to$-definable if there exists a term of type $\mathbf{N}^k\to\mathbf{N}$ such that*

$$Fc_{n_1}\cdots c_{n_k}=c_{f(n_1,\ldots,n_k)}$$

for all $n_1,\ldots,n_k\in\mathbb{N}$.

The question now arises: What functions on natural numbers are definable in $\lambda\!\!\to$? We define the class of multi-variable polynomials.

Definition 2.3.7 *The class of extended polynomial functions is the least class for which it holds that:*

1. *P^k_n is a member of the class*
2. Add *and* Mult *are members of the class*
3. sg *and* $\overline{\text{sg}}$ *are members of the class*
4. *The class is closed under composition*

Theorem 2.3.8 *The class of extended polynomial functions is exactly the class of functions definable in $\lambda\!\!\to$.*
Proof:

See [10]. A more elaborate version of this proof can be found in [4]. Statman proved this result simultaneously.

□

The class of definable functions in $\lambda\!\rightarrow$ is even smaller than the class of primitive recursive functions. We therefore cannot use the simply typed lambda-calculus as a serious programming language.

2.3.3 Gödel's T

The system λT due to Gödel (see [5]) is an extension of $\lambda\!\rightarrow$ with a type constant for natural numbers. New deduction rules are added to introduce and eliminate the new type, but this is not the essential extension. More important are the new contraction rules. These allow for a much larger set of definable functions.

Definition 2.3.9 *The system λT is defined as follows:*

1. *Types:* $\mathbb{T} = \mathbb{V} \mid \mathbf{N} \mid \mathbb{T} \rightarrow \mathbb{T}$
2. *Pseudo-terms:* $\Lambda_{\mathbb{T}} = V \mid \mathbf{0} \mid \mathbf{S} \mid \mathbf{R}_{\mathbb{T}} \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \lambda V : \mathbb{T}.\Lambda_{\mathbb{T}}$
3. *Contraction rules:*

$$\begin{array}{rcl} (\lambda x : A.M)N & \rightarrow_{\beta} & M[x := N] \\ \mathbf{R}_A\, PQ\mathbf{0} & \rightarrow_{\iota} & P \\ \mathbf{R}_A\, PQ(\mathbf{S}\, M) & \rightarrow_{\iota} & Q(\mathbf{R}_A\, PQM)M \end{array}$$

4. *Type assignment:*

(start-rule)	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
($\rightarrow$-intro)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \rightarrow B)}$
($\rightarrow$-elim)	$\dfrac{\Gamma \vdash M : (A \rightarrow B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$
($\mathbf{N}$-intro)	$\dfrac{}{\Gamma \vdash \mathbf{0} : \mathbf{N}} \qquad \dfrac{\Gamma \vdash M : \mathbf{N}}{\Gamma \vdash \mathbf{S}\, M : \mathbf{N}}$
($\mathbf{N}$-elim)	$\dfrac{\Gamma \vdash P : A \quad \Gamma \vdash Q : A \rightarrow \mathbf{N} \rightarrow A \quad \Gamma \vdash M : \mathbf{N}}{\Gamma \vdash \mathbf{R}_A\, PQM : A}$

The definition of λ-definability in λT is clear. Which functions can be represented by terms in λT?

Definition 2.3.10 *A recursive function $f : \mathbb{N} \to \mathbb{N}$ is called provably total in a system of arithmetic $\mathcal{A}$ if $\mathcal{A}$ proves the formula which expresses $\exists z[T(e, n, z)]$ (Kleene's T predicate was defined in 2.1.7, the above formula means "For all n, the program e, with input n terminates.") where e is the code of an algorithm representing f.*

Theorem 2.3.11 *The class of λT-definable functions turns out to be the class of provably total functions in Peano arithmetic* PA*.*
Proof:

See [5].

□

From a functional point of view this class is rich enough for our purposes. It contains the class of primitive recursive functions for we can implement Prim using $\mathbf{R}_{\mathbf{N}}$.

Proposition 2.3.12 *Let $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ be functions. Let f be λT-defined by F and g be λT-defined by G. There exists a λT-term H which λT-defines* $\mathrm{Prim}[f, g]$.
Proof:

Take

$$\mathsf{H} \equiv \lambda \vec{x} : \mathbf{N}^k . \lambda y : \mathbf{N}.\, \mathbf{R}_{\mathbf{N}}\, (\mathsf{F}\, \vec{x})\, (\lambda r : \mathbf{N}. \lambda z : \mathbf{N}. (\mathsf{G}\, \vec{x}\, z\, r))\, y$$

then

$$\begin{array}{rcl} \mathsf{H}\, \vec{x}\, \mathbf{0} & \twoheadrightarrow_{\beta\iota} & \mathsf{F}\, \vec{x} \\ \mathsf{H}\, \vec{x}\, (\mathbf{S}\, y) & \twoheadrightarrow_{\beta\iota} & \mathsf{G}(\mathsf{H}\, \vec{x}\, y)\, \vec{x}\, y \end{array}$$

□

Yet, there are λT-definable functions that are not primitive recursive. So the class of λT-definable functions properly contains the class of primitive recursive functions.

Proposition 2.3.13 *The Ackermann function A defined in 2.1.4 is definable in λT.*
Proof:

Take

$$\begin{array}{lcl} \mathsf{Add} & \equiv & \lambda x : \mathbf{N}. \lambda y : \mathbf{N}.\, \mathbf{R}_{\mathbf{N}}\, y\, (\lambda r : \mathbf{N}. \lambda z : \mathbf{N}.\, \mathbf{S}\, r)\, x \\ \mathsf{A} & \equiv & \mathbf{R}_{\mathbf{N} \to \mathbf{N}} (\lambda x : \mathbf{N}.\, \mathsf{Add}\, x\, x) \\ & & (\lambda f : \mathbf{N} \to \mathbf{N}. \lambda k : \mathbf{N}. (\mathbf{R}_{\mathbf{N}} (\mathbf{S}\, \mathbf{0}) \\ & & (\lambda r : \mathbf{N}. \lambda z : \mathbf{N}. (f\, r)))) \end{array}$$

□

So λT provides a suitable functional language. If viewed as a language to express logical propositions, however, it is poor. There is only implication as a connective and no quantifiers at all.

2.3.4 System F

The system $\lambda 2$ of second order typed lambda-calculus (also known as *System F*, see [5]) due to Girard is an extension of $\lambda\!\!\rightarrow$ with polymorphism. An abstraction on types is added which enlarges the class of definable functions considerably.

Definition 2.3.14 *The system $\lambda 2$ is defined as follows:*

1. *Types:* $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \rightarrow \mathbb{T} \mid \forall \mathbb{V}.\mathbb{T}$
2. *Pseudo-terms:* $\Lambda_{\mathbb{T}} = V \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \Lambda_{\mathbb{T}}\mathbb{T} \mid \lambda V : \mathbb{T}.\Lambda_{\mathbb{T}} \mid \Lambda\mathbb{V}.\Lambda_{\mathbb{T}}$
3. *Contraction rules:*

$$\begin{aligned} (\lambda x : A.M)N &\rightarrow_{\beta} M[x := N] \\ (\Lambda\alpha.M)A &\rightarrow_{\beta} M[\alpha := A] \end{aligned}$$

4. *Type assignment:*

(start-rule)	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
($\rightarrow$-intro)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \rightarrow B)}$
($\rightarrow$-elim)	$\dfrac{\Gamma \vdash M : (A \rightarrow B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$
($\forall$-intro)	$\dfrac{\Gamma \vdash M : A}{\Gamma \vdash (\Lambda\alpha.M) : (\forall\alpha.A)}, \alpha \notin \mathrm{FV}(\Gamma)$
($\forall$-elim)	$\dfrac{\Gamma \vdash M : (\forall\alpha.A)}{\Gamma \vdash MB : A[\alpha := B]}, B \in \mathbb{T}$

What functions are definable in $\lambda 2$? We can extend the definition of Church-numerals to a polymorphic type in $\lambda 2$.

Definition 2.3.15 *Numerals in $\lambda 2$:*

1. *Define*

$$\begin{array}{rcl} \mathbf{N} & \equiv & \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha \\ c_n & \equiv & \Lambda\alpha.\lambda x:\alpha.\lambda f:\alpha \to \alpha.f^n x \\ \mathbf{0} & \equiv & c_0 \\ \mathbf{S} & \equiv & \lambda n:\mathbf{N}.\Lambda\alpha.\lambda x:\alpha.\lambda f:\alpha \to \alpha.f(n\alpha x f) \\ \mathsf{It}_A & \equiv & \lambda a:A.\lambda f:A \to A.\lambda n:\mathbf{N}.(nAaf) \end{array}$$

2. *Let $f:\mathbb{N}^k \to \mathbb{N}$ be a function, f is said to be $\lambda 2$-definable if there exists a term of type $\mathbf{N}^k \to \mathbf{N}$ such that*

$$Fc_{n_1}\cdots c_{n_k} = c_{f(n_1,\ldots,n_k)}$$

for all $n_1,\ldots,n_k \in \mathbb{N}$.

The definition of $\mathbf{N}$ has a peculiarity. The type of $\mathbf{N}$ is $*$. But $\mathbf{N}$ is constructed by quantifying over all elements of $*$. So $\mathbf{N}$ is defined by quantification over an index type that includes $\mathbf{N}$ itself. Definitions like this are called impredicative.

In terms of definability the iterator It_A and the recursor $\mathbf{R}_A$ of λT are equally strong. The recursor is stronger in the following sense, though: In λT we can define the predecessor function

$$\mathsf{Pred} \equiv \mathbf{R}_{\mathbf{N}}\,\mathbf{0}\,(\lambda r:\mathbf{N}.\lambda p:\mathbf{N}.p)$$

and $\mathsf{Pred}(\mathbf{S}\,x)$ will reduce to x (where x is a variable). There is a corresponding implementation in $\lambda 2$ using the iterator but it will work on values only. I.e. in $\lambda 2$ the term $\mathsf{Pred}(\mathbf{S}\,\mathbf{S}\,\mathbf{0})$ will reduce to $\mathbf{S}\,\mathbf{0}$ but $\mathsf{Pred}(\mathbf{S}\,\mathbf{S}\,x)$ will not reduce to $\mathbf{S}\,x$.

Theorem 2.3.16 *The class of functions representable in $\lambda 2$ is exactly the class of functions provably total in second order Peano arithmetic* PA_2*.*
Proof:

See [5].

□

This includes all primitive recursive functions. Yet, there are no special contraction rules like in λT. In $\lambda 2$ we have second order quantification i.e. quantification over the kind of all types. We cannot quantify over natural numbers. So the logical language is still very poor.

2.3.5 Calculus of Constructions

The system λC (the calculus of constructions, see [2]) is the strongest system in the cube.

Definition 2.3.17 *The system λC is defined as follows:*

1. *Pseudo-expressions:* $\mathbb{T} = \mathbb{V} \mid * \mid \Box \mid \mathbb{T}\mathbb{T} \mid \lambda\mathbb{V} : \mathbb{T}.\mathbb{T} \mid \Pi\mathbb{V} : \mathbb{T}.\mathbb{T}$
2. *Contraction rule:* $(\lambda x : A.B)C \to_\beta B[x := C]$
3. *Type assignment: The kind-variables s, s_1, s_2 in the following rules range over $*$ and $\Box$.*

(axiom)	$\langle\rangle \vdash * : \Box$
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}, x \notin \Gamma$
(weakening)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}, x \notin \Gamma$
(Π-intro)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}$
(abstraction)	$\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)}$
(application)	$\dfrac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$
(conversion)	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash A' : s \quad A =_\beta A'}{\Gamma \vdash M : A'}$

Restricting s_1 and s_2 in the rule Π-intro leads to subsystems of λC. For example $\lambda\!\to$ can be obtained by demanding that $(s_1, s_2) \in \{(*, *)\}$ and $\lambda 2$ can be obtained by demanding that $(s_1, s_2) \in \{(*, *), (\Box, *)\}$. The eight subsystems that appear in this way can be ordered in the λ-cube (see [2]).

From the logical point of view this system is strong enough for our purposes.

Definition 2.3.18 *We now have the full power of first order intuitionistic logic. Write:*

$$
\begin{array}{rcl}
A \to B & \equiv & \Pi x : A.B \quad (x \notin \mathrm{FV}(B)) \\
\forall x : A.B & \equiv & \Pi x : A.B \\
\bot & \equiv & \Pi\alpha : *.\alpha \\
\neg A & \equiv & A \to \bot \\
A \,\&\, B & \equiv & \Pi\gamma : *.(A \to B \to \gamma) \to \gamma \\
A \vee B & \equiv & \Pi\gamma : *.(A \to \gamma) \to (B \to \gamma) \to \gamma \\
\exists x : A.B & \equiv & \Pi\gamma : *.(\Pi x : A.(B \to \gamma)) \to \gamma
\end{array}
$$

The definitions for $\perp$, & ,$\vee$ and $\exists$ are *impredicative*. We construct an element of $*$ by quantifying over all elements of $*$ (including the element we are constructing). This has the same drawbacks as the impredicative coding of natural numbers we saw earlier in $\lambda 2$.

The functional power of this calculus is even greater than that of $\lambda 2$. We use the same impredicative definition for **N**.

Definition 2.3.19 *Numerals in λC:*

1. *Define*

$$\begin{array}{rcl} \mathbf{N} & \equiv & \Pi\alpha : *.\alpha \to (\alpha \to \alpha) \to \alpha \\ c_n & \equiv & \lambda\alpha : *.\lambda x : \alpha.\lambda f : \alpha \to \alpha.f^n x \\ \mathbf{0} & \equiv & c_0 \\ \mathbf{S} & \equiv & \lambda n : \mathbf{N}.\lambda\alpha : *.\lambda x : \alpha.\lambda f : \alpha \to \alpha.f(n\alpha x f) \\ \mathsf{It}_A & \equiv & \lambda a : A.\lambda f : A \to A.\lambda n : \mathbf{N}.(nAaf) \end{array}$$

2. *Let $f : \mathbb{N}^k \to \mathbb{N}$ be a function, f is said to be λC-definable if there exists a term of type $\mathbf{N}^k \to \mathbf{N}$ such that*

$$Fc_{n_1} \cdots c_{n_k} = c_{f(n_1,\ldots,n_k)}$$

for all $n_1, \ldots, n_k \in \mathbb{N}$.

What functions are λC-definable?

Theorem 2.3.20 *The class of functions representable in λC is exactly the class of functions provably total in higher order Peano arithmetic* PA_ω.
Proof:

See [5].

□

This of course includes all primitive recursive functions. So the calculus of construction seems to be the ideal choice for our purposes. But it uses the same impredicative encoding for natural numbers we saw in $\lambda 2$. Therefore we will define an extended version of this system which is implemented in the LEGO proof development system.

2.3.6 Extended Calculus of Constructions

The system ECC (extended calculus of constructions, see [7]) is an extension of λC with sigma-types and an infinite type-hierarchy. This means that there is a partial ordering on types.

Definition 2.3.21 *The type cummulativity relation $\preceq$ is defined inductively as follows:*

1. $* \preceq \Box_0 \preceq \Box_1 \ldots$

2. *If* $A =_\beta A'$ *and* $B \preceq B'$ *then* $\Pi x : A.B \preceq \Pi x : A'.B'$

3. *If* $A \preceq A'$ *and* $B \preceq B'$ *then* $\Sigma x : A.B \preceq \Sigma x : A'.B'$

4. (a) $A \preceq A$

   (b) *If* $A \preceq B$ *and* $B \preceq C$ *then* $A \preceq C$

The Σ-types like the Π-types are a way to construct dependent types. Addition of all Σ-types to λC leads to paradoxes. This is caused by the fact that by using Σ-types we can construct a type which is isomorphic to $*$. Yet, it is also an inhabitant of $*$ which leads to a Russell-like paradox. This is why the type-hierarchy is introduced. In ECC we do not allow so-called large Σ-types. The type of a Σ-type $\Sigma x : A.B$ is always larger than the index-type A.

Definition 2.3.22 *The system* ECC *is defined as follows:*

1. *Pseudo-expressions:*

$$\begin{array}{rcl} \mathbb{T} & = & \mathbb{V} \mid * \mid \Box_{\mathbb{N}} \mid \mathbb{T}\mathbb{T} \mid \lambda\mathbb{V} : \mathbb{T}.\mathbb{T} \mid \Pi\mathbb{V} : \mathbb{T}.\mathbb{T} \mid \\ & & \pi_1(\mathbb{T}) \mid \pi_2(\mathbb{T}) \mid \mathbf{pair}_{\mathbb{T}}(\mathbb{T},\mathbb{T}) \mid \Sigma\mathbb{V} : \mathbb{T}.\mathbb{T} \end{array}$$

2. *Contraction rule:*

$$\begin{array}{rcl} (\lambda x : A.B)C & \to_\beta & B[x := C] \\ \pi_1(\mathbf{pair}_{\Sigma x:A.B}(M,N)) & \to_\beta & M \\ \pi_2(\mathbf{pair}_{\Sigma x:A.B}(M,N)) & \to_\beta & N \end{array}$$

3. *Type assignment: In the rule* Π-intro (s_1, s_2) *ranges over* $(*, *)$, $(\Box_i, *)$ *and* $(\Box_i, \Box_i)$. *But a similar rule for* $(*, \Box_i)$ *can be deduced using the rule of* cummulativity.

(axioms)	$\langle\rangle \vdash * : \Box_0 \qquad \langle\rangle \vdash \Box_i : \Box_{i+1}$
(start-rule)	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
(Π-intro)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$
(abstraction)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)}$
(application)	$\dfrac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$
(Σ-intro)	$\dfrac{\Gamma \vdash A : \Box_i \quad \Gamma, x : A \vdash B : \Box_i}{\Gamma \vdash (\Sigma x : A.B) : \Box_i}$
(pairing)	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[x := M]}{\Gamma \vdash \mathbf{pair}_{\Sigma x:A.B}(M, N) : (\Sigma x : A.B)}$
(projection)	$\dfrac{\Gamma \vdash M : (\Sigma x : A.B)}{\Gamma \vdash \pi_1(M) : A} \qquad \dfrac{\Gamma \vdash M : (\Sigma x : A.B)}{\Gamma \vdash \pi_2(M) : B[x := \pi_1(M)]}$
(cummulativity)	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash A' : \Box_i}{\Gamma \vdash M : A'}, A \preceq A'$

2.3.7 Lego

The proof checker LEGO (see [8]) is based on the extended calculus of constructions. The notation used in LEGO is a little different because computers cannot 'read' Greek symbols (See table 2.1).

Some extras have been added to make the system more convenient to use. One of these extras is *implicit polymorphism*. By a special abstraction operator, functions can be made dependent on a type which need not be explicitly given when applying the function to its arguments. The definition for Leibniz equality for example would read in LEGO something like:

```
[Q = [T|Type][x,y:T]{P:T->Prop}(P x)->P y];
```

Where `[T|Type][x,y:T]` means that `Q` can be applied to two terms of the same type T but we do not need to give T as an extra argument. This will be done by the system.

Other extras are the inductive types. These are a generalization of the type for natural numbers we introduced in λT. To specify an inductive type

Notation in ECC	Notation in LEGO
$*$	`Prop`
$\Box_i$	`Type(`i`)`
$\Pi x : A.B$	`{x:A}B`
$A \to B$	`A->B`
$\lambda x : A.B$	`[x:A]B`
AB	`A B`
$\Sigma x : A.B$	`<x:A>B`
$A \times B$	`A#B`
(a, b)	`(a,b)`
$\mathbf{pair}_{\Sigma x:A.B}(a, b)$	`(a,b):<x:A>B`
$\pi_1(M)$	`M.1`
$\pi_2(M)$	`M.2`

Table 2.1: Translation of ECC-expressions to LEGO-expressions.

we provide names and types for the constructor terms (for introduction of the type). The system will provide the necessary destructor terms (for elimination of the type). The type of natural numbers can be specified in LEGO as follows:

```
Inductive [nat:Type(0)]
Constructors [zero:nat][succ:nat->nat];
```

By this definition the context will be enriched with the following constants:

```
nat: Type(0);
zero: nat;
succ: nat->nat;
nat_elim: {C:nat->TYPE} (C zero) ->
            ({x:nat}(C x)->(C (succ x)) -> {z:nat} C z;
```

The term `nat_elim` behaves like the recursor $\mathbf{R}_A$ we saw earlier in λT. It can be used both as a recursion operator $\mathbf{R}_{\mathbf{N}}$ for constructing recursive functions and as an induction operator $\mathbf{R}_*$ for constructing inductive proof-objects. Since we are only interested in natural numbers and functions on them, we will not need the full power of inductive types. We will use the above definitions to implement our natural numbers in LEGO. To avoid having to use the LEGO syntax throughout the rest of this thesis, we define the following.

Definition 2.3.23 *In the sequel when we say that something is derivable in* LEGO*, we mean that it can be derived in* ECC *extended with the inductive type* $\mathbf{N}$ *we introduced in* λT*. The set of all* LEGO*-expressions will be called* Λ_L*.*

Chapter 3

Proof by calculation

3.1 The problem

In chapter 1 the necessity to formalize informal arguments in order to allow for automated proof-checking, was discussed. In chapter 2 we chose an appropriate formalism for this task. Now the question arises how we should tackle specific formalizations. There are many choices to be made when formalizing mathematics. How can we choose the right representation of objects and concepts in type theory? Here is one way to check this: A representation is good, if the informal arguments translate to formal proof-objects in a straightforward way.

Some theorems, which are easy to prove using an informal argument, are very hard to prove when using a proof development system. Take a look at the following two example theorems:

Proposition 3.1.1 *Five is a prime*

Proposition 3.1.2 *Zero has no predecessor in* $\mathbb{N}$

Clearly, the first theorem is easy to prove. We only have to check all natural numbers less than five to see that among them there are no proper divisors. The second theorem looks much harder. It claims a property of a function, which is essentially an infinite object. We would expect that proving the first theorem using a computer would be easier than proving the second one.

Yet in LEGO the proof of the second theorem is not that hard. An easy proof is obtained by defining (using the recursor) a function of type $\mathbb{N} \to *$ which on 0 yields $\perp$ and on all other numbers yields $\top$. (Where $\perp = \forall\alpha : *.\alpha$ and $\top = \forall\alpha : *.\alpha \to \alpha$.) The hypothesis $0 = S(n)$ now implies $\perp = \top$. Since we can prove $\top$ we can prove $\perp$. Therefore $0 = S(n)$ implies $\perp$.

```
Goal {n:nat} not (Q zero (succ n));
   Intros n h;
   [C = nat_elim ([_:nat]Prop)
        absurd
```

```
        [n:nat][P:Prop] ({x:Prop}x->x)];
    Claim Q absurd ({x:Prop}x->x);
    Qrepl ?+1;
    Refine [P:Prop][x:P]x;
    Equiv Q (C zero) (C (succ n));
    Refine Q_resp;
    Immed;
Save zero_not_succ;
```

A formalization of the proof of the first theorem takes much more effort. We would have to prove that $1|5 \to 1 = 1$ and $2|5 \to 2 = 1$ and $3|5 \to 3 = 1$ and $4|5 \to 4 = 1$ etc. Furthermore we restricted ourselves to only checking numbers less than five, so we would also have to convince the proof checker of the fact that no numbers larger than five could be proper divisors of five. Now imagine having to prove that 61 is a prime. The proof-object would get 30 times larger. There must be a better way to translate our informal argument.

3.2 The two level approach

The idea behind our informal proof of the primality of five was the existence of an algorithm for primality checking. We encountered problems when we tried to formalize this idea because the proof checkers forced us to emulate a computer running the algorithm. In mathematics theorems are often proved using algorithms but the calculations made by a computer (or a human) are not considered to be part of that proof.

Mathematicians who are interested in algebra for example write theorems about polynomials. Certain operations on polynomials, such as factorizing them as product of polynomials of lesser degree, are difficult to perform manually. To assist these mathematicians computer algebra systems were developed. These consist of a large library of algorithms useful to mathematicians. As an example look at the following theorem:

Proposition 3.2.1 *For all $n \in \mathbb{N}$ it holds that $n^2 + n$ is even.*
Proof:

One way to prove this is:

1. $n^2 + n = n(n + 1)$
2. *Case distinction:*
   (a) *Either n is even, in which case $n(n + 1)$ is even*
   (b) *Or n is odd, in which case $n+1$ is even and hence $n(n+1)$ is even*

□

The essential step in this proof is proving: $n^2+n = n(n+1)$. In an informal argument this step needs no more justification, but a proof checker will need a proof for this. We can prove $n^2 + n = n(n + 1)$ by hand (using axioms of associativity, commutativity and distributivity etc.) but this is not what we want. A better way is to view $n^2 + n$ as a polynomial (a syntactical description of the expression $n^2 + n$, as opposed to $n^2 + n$ as just an element of $\mathbb{N}$) and use our computer algebra system to factorize $n^2 + n$ intro $n(n + 1)$. We need to prove the correctness of our factorization algorithm to convince the proof checker. I.e. we have to prove that for all polynomials p it holds that

$$\forall_{n:\mathbb{N}}[\mathsf{eval}(p(n)) = \mathsf{eval}(\mathsf{factor}(p(n)))]$$

This may be a large proof, but once this is done we can use the same method to prove for example that $n^5 + 2n^4 + 3n^3 + n^2 + 5n$ is even for all $n \in \mathbb{N}$ and the proof will be of the same size as in the former case. Note that this approach is only possible if algorithms live in the same space as propositions and proofs. This is the case with proof checkers based on type theory.

This method of viewing mathematical objects on two different levels is also applicable to the primality problem in the example. The primality predicate can be viewed syntactically and from this a primality checking program can be derived. A proof of the correctness of this program relative to the predicate can also be constructed. The predicate, the program, and the proof can then be translated to the semantical level, where the primality of five can be proved by running the program on five. We want to generalize this idea to all primitive recursive predicates i.e. prove $P(\vec{n})$ for each primitive recursive predicate $P \subseteq \mathbb{N}^k$ and each $\vec{n} \in \mathbb{N}^k$ using the following scheme:

1. Describe a language in which you can express the primitive recursive predicates.

2. Construct for each predicate expression P an expression for its characteristic function χ_P.

3. Construct a proof-object b_1 for the correctness statement

$$\forall_{\vec{x}:\mathbb{N}^k}[P(\vec{x}) \iff \chi_P(\vec{x}) = 1]$$

   The formula claims the correctness of the constructed algorithm.

4. Construct a proof-object b_2 for the proof of the calculation

$$\chi_P(\vec{n}) = 1$$

   This proof-object is easy to construct, since $\chi_P(\vec{n})$ will reduce to 1. Therefore $\chi_P(\vec{n}) = 1$ will reduce to $1 = 1$. For this type an inhabitant exists. The conversion rule states that each inhabitant of the latter type is also an inhabitant of the former one.

5. The combination of b_1 and b_2 leads to a proof-object for $P(\vec{n})$.

The steps in this scheme will be taken in the next chapter.

Chapter 4

The prover

4.1 Languages

4.1.1 The language PRIMF

We will now define the language PRIMF. The elements of PRIMF are not the primitive recursive functions of definition 2.1.2, but *function prescriptions* describing the structure of an algorithm computing such a function. The distinction between functions and algorithms is crucial at this point. Two different prescriptions can represent the same primitive recursive function. Function prescriptions are defined inductively, but it is not a context free language. The non-context free restriction of arity is defined inductively at the same time. To allow sharing of function prescriptions, we introduce *identifiers* and *valid contexts of function definitions.*

Definition 4.1.1 *Let $\mathbb{V}$ be an infinite set of identifiers. We define function prescriptions inductively as follows:*

1. `Z` *and* `S` *are function prescriptions of arity* 1 *and for all* $n, k \in \mathbb{N}$ *with* $k \leq n$ `P[`n`,`k`]` *is a function prescription of arity* n.
2. *If* F *and* $G_1, \ldots, G_n$ *are function prescriptions and the arity of* F *is* n *and the arity of all* G_i *is* m, *then* `Comp[`F`,`G_1`, ..., `G_n`]` *is a function prescription of arity* m.
3. *If* F *and* G *are function prescriptions of arity* n *and* $n+2$ *respectively, then* `Prim[`F`,`G`]` *is a function prescription of arity* $n+1$.
4. *An identifier* $\nu \in \mathbb{V}$ *is a function prescription.*

Actually it would be better to talk about *pseudo*-function prescriptions, since a prescription as defined above can contain undefined identifiers. We will now define contexts of function definitions which will only be called valid if all used identifiers in a prescription were defined earlier.

Definition 4.1.2 *Valid contexts of* PRIMF*-function definitions are defined as follows:*

1. *The set of used identifiers in a function prescription H is defined inductively as:*

$$\mathrm{id}(H) = \begin{cases} \emptyset & \text{if } H = \mathtt{Z}, \mathtt{S}, \mathtt{P[}n\mathtt{,}k\mathtt{]} \\ \mathrm{id}(F) \cup \bigcup_{i=1}^{n} \mathrm{id}(G_i) & \text{if } H = \mathtt{Comp[}F\mathtt{,}G_1\mathtt{,}\ \ldots\mathtt{,}\ G_n\mathtt{]} \\ \mathrm{id}(F) \cup \mathrm{id}(G) & \text{if } H = \mathtt{Prim[}F\mathtt{,}G\mathtt{]} \\ \{\nu\} & \text{if } H = \nu \in \mathbb{V} \end{cases}$$

2. *A context of function definitions Δ is a finite sequence of ordered pairs (ν, δ) where $\nu \in \mathbb{V}$ and δ a function prescription.*

3. *The set of used identifiers in a context of function definitions Δ is defined as* $\mathrm{id}(\Delta) = \{\nu \mid (\nu, \delta) \in \Delta$ *for some* $\delta\}$.

4. *Valid contexts of function definitions are defined inductively as follows:*

   (a) $\langle\rangle$ *is valid.*

   (b) *If Δ is a valid context of function definitions, then so is the concatenation of Δ and (ν, δ) provided that $\nu \notin \mathrm{id}(\Delta)$ and $\mathrm{id}(\delta) \subseteq \mathrm{id}(\Delta)$.*

We write arity(F) for the arity of a function prescription F. In the following example arity(`Add`) = arity(`Mult`) = 2.

Example 4.1.3 *A valid context of function definitions for addition, multiplication, sign, and subtraction expressed in the language* PRIMF*:*

```
fun Add   = Prim[P[1,1],Comp[S,P[3,3]]]
fun Mult  = Prim[Z,Comp[Add,P[3,3],P[3,1]]]
fun sg    = Prim[P[0,0],Comp[Comp[S,Z],P[2,2]]]
fun sgbar = Prim[Comp[S,P[0,0]],Comp[Z,P[2 2]]]
fun Pred  = Prim[P[0,0],P[2,2]]
fun rtbus = Prim[P[1,1],Comp[pred,P[3,1]]
fun Subtr = Comp[rtbus,P[2,2],P[2,1]]
```

4.1.2 A streamlined version of PRIMF: PRIMF$^+$

From a compiler-builder's point of view the language PRIMF is very simple. When applying composition and primitive recursion the parser has to check the various arities. The arities form the only context dependent restriction. If it were not for this restriction, the language would have been context-free.

For someone who wants to express complex algorithms PRIMF is too limited. Instead of the prescriptions from example 4.1.3, we would much rather write something like:

Example 4.1.4 *Function definitions for addition and multiplication expressed in the language* PRIMF$^+$*:*

```
fun Add(m,0) = m
    Add(m,n+1) = S(Add(m,n))

fun Mult(m,0) = 0
    Mult(m,n+1) = Add(Mult(m,n),m)
```

Projection functions are no longer needed. We will use formal parameters instead. Composition is described by using nested function calls. Primitive recursion is described using two sub-definitions.

The new language, in which we allow definitions like this, will be called PRIMF$^+$. The class of functions described by the prescriptions of PRIMF$^+$ is of course still the same class of primitive recursive functions.

Definition 4.1.5 *Terms (the elements of* TERM*) are constructed as follows:*

1. *A variable is a term.*
2. *For each natural number n there is a term $\underline{n}$ which is the constant for n.*
3. *If F is a function prescription from* PRIMF$^+$ *of arity n, and $T_1, \ldots, T_n$ are terms, then F*`(`*T_1*`,`*$\ldots$*`,`*T_n*`)` *is a term.*

Definition 4.1.6 *The set of free variables* FV(T) *of a term T is defined as:*

$$\mathrm{FV}(T) = \begin{cases} \{v\} & \textit{if } T \textit{ is the variable } v \\ \emptyset & \textit{if } T \textit{ is a constant} \\ \mathrm{FV}(T_1) \cup \ldots \cup \mathrm{FV}(T_n) & \textit{if } T = F\ (\ T_1\ ,\ \ldots\ ,\ T_n\ ) \end{cases}$$

Definition 4.1.7 *Let $\mathbb{V}$ be an infinite set of identifiers. We define function prescriptions inductively as follows:*

1. `Z` *and* `S` *are function prescriptions of arity* 1 *and for all $n, k \in \mathbb{N}$ with $k \leq n$* `P[`*n*`,`*k*`]` *is a function prescription of arity n.*
2. *If F and $G_1, \ldots, G_n$ are function prescriptions and the arity of F is n and the arity of all G_i is m, then* `Comp[`*F*`,`*G_1*`, ...,` *G_n*`]` *is a function prescription of arity m.*
3. *If F and G are function prescriptions of arity n and $n+2$ respectively, then* `Prim[`*F*`,`*G*`]` *is a function prescription of arity $n+1$.*
4. *If T is a term, then* `(`*x_1*`,`*$\ldots$*`,`*x_n*`)`*T is a function prescription of arity n provided that* FV$(T) \subseteq \{x_1, \ldots, x_n\}$.
5. *An identifier $\nu \in \mathbb{V}$ is a function prescription.*

Definition 4.1.8 *A function definition has one of the following three forms:*

1. *Ordinary definition:*

$$\texttt{fun}\quad F\texttt{=}G$$

   Where G is a function prescription.

2. *Definition using abstraction:*

$$\texttt{fun}\quad F\texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)=}T$$

   Here F an identifier, $v_1,\dots,v_n$ are variables and T is a term for which $\mathrm{FV}(T)\subseteq\{v_1,\dots,v_n\}$.

3. *Definition using primitive recursion:*

$$\begin{aligned}\texttt{fun}\quad & F\texttt{(0,}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)=}T_1\\ & F\texttt{(}w\texttt{+1,}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)=}T_2\end{aligned}$$

   Here F is an identifier, $v_1,\dots,v_n$ and w are variables and T_1 and T_2 are terms for which $\mathrm{FV}(T_1)\subseteq\{v_1,\dots,v_n\}$ *and* $\mathrm{FV}(T_2)\subseteq\{v_1,\dots,v_n,w\}$.

Definition 4.1.9 *Valid contexts of* PRIMF^+*-function definitions are defined as follows:*

1. *The definition of the set of used identifiers* $\mathrm{id}(F)$ *in a* PRIMF^+*-function prescription F is a straightforward extension of the definition of the set of used identifiers in a* PRIMF*-function prescription.*

2. *A context of* PRIMF^+*-function definitions is a finite sequence of* PRIMF^+*-function definitions.*

3. *The set of used identifiers in a context of function definitions Δ is defined as* $\mathrm{id}(\Delta)=\{\nu\mid\nu$ *is defined in some* $\delta\in\Delta\}$

4. *A valid context of* PRIMF^+*-function definitions is defined inductively as:*

   (a) $\langle\rangle$ *is valid.*

   (b) *If Δ is a valid context then so is the concatenation of Δ with a function definition δ provided that* $\mathrm{id}(\delta)\subseteq\mathrm{id}(\Delta)$. *If δ is a function definition by primitive recursion, cf. definition 4.1.8 item 3 the term* $F\texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{,}w\texttt{)}$ *may occur in T_2.*

4.1.3 The language PRIMP

We can now construct the language of primitive recursive predicate prescriptions:

Definition 4.1.10 *Formulas (the elements of* FORM*) are constructed as follows:*

1. *If T_1 and T_2 are terms from* TERM, *then* $T_1\texttt{<}T_2$, $T_1\texttt{=}T_2$ *and* $T_1\texttt{>}T_2$ *are formulas.*

2. *If X is a formula, then so are* `~` X *and* `(`X`)`.

3. *If X and Y are formulas , then so are* X `&` Y, X `v` Y *and* X `->` Y.

4. *If X is a formula, m a variable and T a term such that $m \notin \mathrm{FV}(T)$, then* `Ex(`m `<` T`)`X *and* `All(`m `<` T`)`X *are formulas.*

Definition 4.1.11 *Let X be a formula. The set of free variables* $\mathrm{FV}(X)$ *of X is constructed as follows:*

$$\mathrm{FV}(X) = \begin{cases} \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) & \textit{if } X \textit{ is one of } T_1 \texttt{<} T_2,\ T_1 \texttt{=} T_2,\ T_1 \texttt{>} T_2 \\ \mathrm{FV}(Y) & \textit{if } X = \texttt{~}Y \\ \mathrm{FV}(Y) \cup \mathrm{FV}(Z) & \textit{if } X = Y \texttt{ \& } Z \\ \mathrm{FV}(Y) \cup \mathrm{FV}(Z) & \textit{if } X = Y \texttt{ v } Z \\ \mathrm{FV}(Y) \cup \mathrm{FV}(Z) & \textit{if } X = Y \texttt{ -> } Z \\ \mathrm{FV}(T) \cup \mathrm{FV}(Y) \setminus \{m\} & \textit{if } X = \texttt{Ex(}m \texttt{<} T\texttt{)}Y \\ \mathrm{FV}(T) \cup \mathrm{FV}(Y) \setminus \{m\} & \textit{if } X = \texttt{All(}m \texttt{<} T\texttt{)}Y \end{cases}$$

Definition 4.1.12 *If X is a formula and $v_1, \ldots, v_n$ are variables such that* $\mathrm{FV}(X) \subseteq \{v_1, \ldots, v_n\}$ *then* `(`v_1`,`$\ldots$`,`v_n`)`X *is a predicate prescription.*

Definition 4.1.13 *Predicate definitions look like:*

1. *Ordinary definition:*

   `pred` P`=`G

   where G is a predicate prescription.

2. *Definition using abstraction:*

   `pred` P`(`v_1`, ..., `v_n`)=`X

   where P is an identifier, $v_1, \ldots, v_n$ are variables and X is a formula for which it holds that $\mathrm{FV}(X) \subseteq \{v_1, \ldots, v_n\}$.

Definition 4.1.14 *A valid context of predicate definitions is a finite sequence of predicate definitions which is valid in the same sense as the valid contexts defined in 4.1.2.*

Example 4.1.15 *Check whether the input is even, odd or prime respectively, expressed in* PRIMP*:*

```
pred Divides(n,m) = Ex(d<S(m)) mult(n,d)=m
pred Even(n) = Divides(2,n)
pred Odd(n) = ~Even(n)
pred Prime(n) = n>1 & All(d<n) (Divides(d,n) -> d=1)
```

4.2 Semantics

It is not difficult to provide semantics for the function- and predicate prescriptions. Obviously, the function prescriptions would be mapped to primitive recursive functions while the predicate prescriptions would be mapped to primitive recursive predicates. We will not give such a map here. The reader can infer the meaning of the languages from the translations defined in the next section and the semantics of the system LEGO.

4.3 Translations

We now define maps to translate expressions from the various languages we defined in this chapter to the lambda calculus of LEGO. In LEGO the relations for equality and less-than are usually defined as follows.

Definition 4.3.1 *In* LEGO *define the following expressions:*

$$\begin{array}{lcl} \mathsf{Q} & \equiv & \lambda x:\mathbf{N}.\lambda y:\mathbf{N}.\forall P:\mathbf{N}\to *.(px)\to(py) \\ \mathsf{lt} & \equiv & \lambda x:\mathbf{N}.\lambda y:\mathbf{N}.\exists z:\mathbf{N}.\,\mathsf{Q}(\mathsf{Add}\,x(\mathbf{S}\,z))y \end{array}$$

We have used identifiers as function- and predicate prescriptions to allow sharing of code. We want the user to be able to express complex algorithms.

When theoretical computer scientists define formal languages they usually do not allow identifiers to play this role. Instead they use meta-variables (not part of the object language) as an abstraction mechanism. In an implementation of such a language this is of course not possible. We have to describe the language on an object level. This is why we defined valid contexts.

When we try to translate prescriptions, we are forced to tow along the context in which a prescription occurs. Instead we expand the context, i.e. we substitute all occurrences of identifiers by their definitions and define the translations on expanded prescriptions. This is done here in order to describe the translation to the pure lambda calculus of LEGO but in our implementation we translate the identifiers in our languages to meta-variables in the proof-checker LEGO.

Definition 4.3.2 *We define the maps inductively and dependent of each other:*

1. *The map* $[\![\ ]\!]:\mathrm{TERM}\to\Lambda_L$ *is defined inductively as follows:*

$$\begin{array}{rcl} [\![v]\!] & = & v \\ [\![\underline{n}]\!] & = & \mathbf{S}^n\,\mathbf{0} \\ [\![F(T_1,\ldots,T_n)]\!] & = & [\![F]\!][\![T_1]\!]\cdots[\![T_n]\!] \end{array}$$

2. *The map* $[\![\ ]\!]:\mathrm{PRIMF}^+\to\Lambda_L$ *is defined inductively as follows:*

$$\begin{array}{rcl} [\![\mathsf{Z}]\!] & = & \lambda x:\mathbf{N}.\mathbf{0} \\ [\![\mathsf{S}]\!] & = & \mathbf{S} \end{array}$$

$$
\begin{array}{rcl}
[\![\texttt{P[0,0]}]\!] & = & \mathbf{0} \\
[\![\texttt{P[}n\texttt{,}k\texttt{]}]\!] & = & \lambda\vec{x} : \mathbf{N}^n.x_k \\
[\![\texttt{Comp[}F\texttt{,}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]}]\!] & = & \lambda\vec{x} : \mathbf{N}^k.[\![F]\!]([\![G_1]\!]\vec{x})\cdots([\![G_n]\!]\vec{x}) \\
 & & \textit{where } k = \mathrm{arity}(G_i) \\
[\![\texttt{Prim[}F\texttt{,}G\texttt{]}]\!] & = & \lambda y : \mathbf{N}.\lambda\vec{x} : \mathbf{N}^k.\ \mathbf{R_N} \\
 & & ([\![F]\!]\vec{x}) \\
 & & (\lambda r : \mathbf{N}.\lambda z : \mathbf{N}.([\![G]\!]rz\vec{x})) \\
 & & \textit{where } k = \mathrm{arity}(F) \\
[\![\texttt{(}\vec{x}\texttt{)}T]\!] & = & \lambda\vec{x} : \mathbf{N}^k.[\![T]\!]
\end{array}
$$

3. The map $[\![\ ]\!] : \mathrm{FORM} \to \Lambda_L$ *is defined inductively as follows:*

$$
\begin{array}{rcl}
[\![T_1 \texttt{<} T_2]\!] & = & \mathsf{lt}\,[\![T_1]\!]\,[\![T_2]\!] \\
[\![T_1 \texttt{=} T_2]\!] & = & \mathsf{Q}\,[\![T_1]\!]\,[\![T_2]\!] \\
[\![T_1 \texttt{>} T_2]\!] & = & \mathsf{lt}\,[\![T_2]\!]\,[\![T_1]\!] \\
[\![\texttt{\~{}}\,X]\!] & = & \neg[\![X]\!] \\
[\![X \,\texttt{\&}\, Y]\!] & = & [\![X]\!]\ \&\ [\![Y]\!] \\
[\![X \,\texttt{v}\, Y]\!] & = & [\![X]\!] \vee [\![Y]\!] \\
[\![X \,\texttt{->}\, Y]\!] & = & [\![X]\!] \to [\![Y]\!] \\
[\![P(T_1,\ldots,T_n)]\!] & = & [\![P]\!][\![T_1]\!]\cdots[\![T_n]\!] \\
[\![\texttt{All(}x \texttt{<} T\texttt{)}X]\!] & = & \forall x : \mathbf{N}.(\mathsf{lt}\,x\,T) \to X \\
[\![\texttt{Ex(}x \texttt{<} T\texttt{)}X]\!] & = & \exists x : \mathbf{N}.(\mathsf{lt}\,x\,T)\ \&\ X
\end{array}
$$

4. The map $[\![\ ]\!] : \mathrm{PRIMP} \to \Lambda_L$ *is defined inductively as follows:*

$$
[\![\texttt{(}\vec{x}\texttt{)}X]\!] \quad = \quad \lambda\vec{x} : \mathbf{N}^k.[\![X]\!]
$$

4.4 Proof-objects

This section describes the automatic generation of characteristic terms, and proof-objects for correctness statements. (The correctness of the characteristic terms, with respect to the formulas from which they are generated.) The correctness proof can be used to prove propositions of the form $P(\underline{\vec{n}})$, where P is a primitive recursive predicate, as described in chapter 3.

4.4.1 A proof-object generator

First, some notation is introduced. We define a LEGO-expression for the notion of characteristic term of a formula.

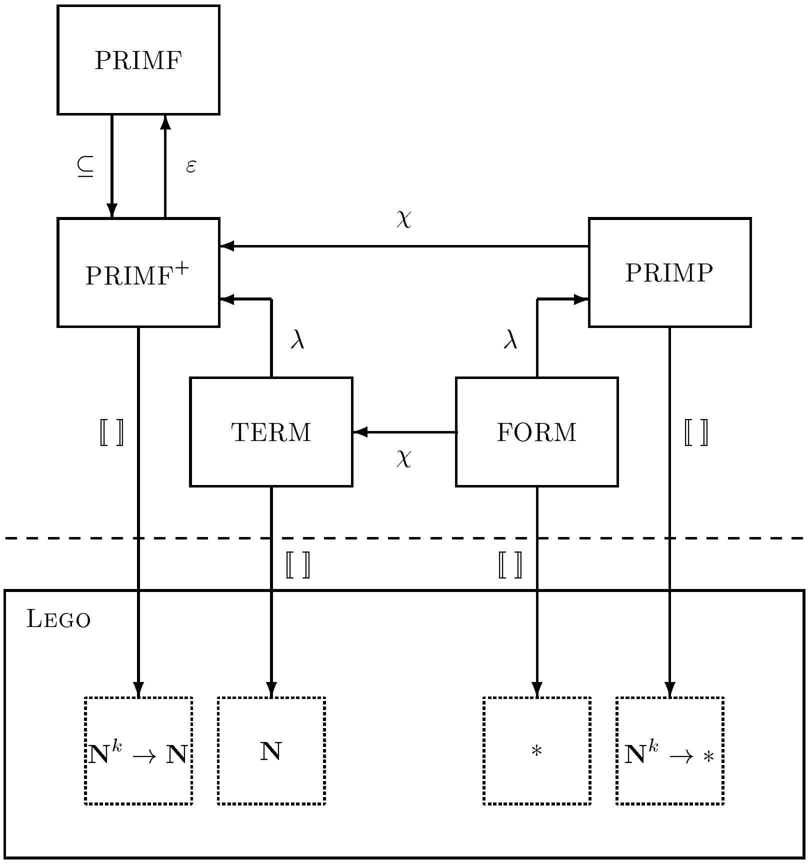


Figure 4.1: Languages and their relations. Arrows marked with $[\![\ ]\!]$ are the translations defined in this section. Arrows marked with λ indicate that the language pointed to is obtained by abstraction. Arrows marked with χ indicate the generation of a characteristic function or term.

Definition 4.4.1 *In* LEGO *define the following expressions:*

$$\begin{array}{rcl} \mathsf{istrue} & \equiv & \lambda x : \mathbf{N}.\exists y : \mathbf{N}.(\mathsf{Q}\, x\, (\mathbf{S}\, y)) \\ \mathsf{codes} & \equiv & \lambda x : \mathbf{N}.\lambda p : *.(\mathsf{istrue}\, x \to p)\ \&\ (p \to \mathsf{istrue}\, x) \end{array}$$

If, for some formula φ, it holds that $\vdash \mathsf{codes}[\![\chi_\varphi]\!]\,[\![\varphi]\!]$, then this means that χ_φ is the characteristic term for the formula φ. We can now express the correctness of the two basic predicates.

Lemma 4.4.2 *There exist primitive recursive function prescriptions* $\chi_<$*,* $\chi_=$ *and* LEGO*-expressions* $\mathsf{ok}_<$*,* $\mathsf{ok}_=$ *such that in* LEGO*:*

$$\begin{array}{l} \vdash \mathsf{ok}_< : \forall x : \mathbf{N}.\forall y : \mathbf{N}.\, \mathsf{codes}([\![\chi_<]\!]\, x\, y)(\mathsf{lt}\, x\, y) \\ \vdash \mathsf{ok}_= : \forall x : \mathbf{N}.\forall y : \mathbf{N}.\, \mathsf{codes}([\![\chi_=]\!]\, x\, y)(\mathsf{Q}\, x\, y) \end{array}$$

Proof:

Take

$$\begin{array}{rcl} \texttt{nor} & = & \texttt{Prim[sgbar,Comp[Z,P[3,3]]]} \\ \chi_< & = & \texttt{Comp[sg,rtbus]} \\ \chi_= & = & \texttt{Comp[nor,}\chi_<\texttt{, Comp[}\chi_<\texttt{,P[2,2],P[2,1]]]} \end{array}$$

The proof-objects are obtained by generating induction proof-objects (using $\mathbf{R}_*$*) on* x*.*

□

The following two lemmas will be used in the construction of characteristic terms and their correctness proof-objects for compound formulas.

Lemma 4.4.3 *There exist function prescriptions* $f_\neg$*,*$f_\&$*,*$f_\vee$*,*$f_\to$ *and corresponding* LEGO*-expressions* $\mathsf{ok}_\neg$*,*$\mathsf{ok}_\&$*,*$\mathsf{ok}_\vee$*,*$\mathsf{ok}_\to$ *such that in* LEGO*:*

$$\begin{array}{l} \vdash \mathsf{ok}_\neg : \forall s : \mathbf{N}.\forall p : *.(\mathsf{codes}\, sp) \to (\mathsf{codes}([\![f_\neg]\!] s)(\neg p) \\ \vdash \mathsf{ok}_\& : \forall s,t : \mathbf{N}.\forall p,q : *.(\mathsf{codes}\, sp) \to (\mathsf{codes}\, tq) \to (\mathsf{codes}([\![f_\&]\!] st)(p\ \&\ q)) \\ \vdash \mathsf{ok}_\vee : \forall s,t : \mathbf{N}.\forall p,q : *.(\mathsf{codes}\, sp) \to (\mathsf{codes}\, tq) \to (\mathsf{codes}([\![f_\vee]\!] st)(p \vee q)) \\ \vdash \mathsf{ok}_\to : \forall s,t : \mathbf{N}.\forall p,q : *.(\mathsf{codes}\, sp) \to (\mathsf{codes}\, tq) \to (\mathsf{codes}([\![f_\to]\!] st)(p \to q)) \end{array}$$

Proof:

Take

$$\begin{array}{rcl} f_\neg & = & \texttt{Prim[Comp[S,P[0,0]],Comp[Z,P[2,2]]]} \\ f_\& & = & \texttt{Prim[Z,P[3,3]]} \\ f_\vee & = & \texttt{Prim[P[1,1],Comp[S,Comp[Z,P[3,3]]]]} \\ f_\to & = & \texttt{Prim[Comp[S,Z],P[3,3]]} \end{array}$$

The proof-objects are obtained by generating induction proof-objects (using $\mathbf{R}_*$*) on* s *and* t*.*

□

Lemma 4.4.4 *There exist* LEGO*-expressions* $\mathsf{ok}_{\forall 0}$,$\mathsf{ok}_{\forall 1}$,$\mathsf{ok}_{\exists 0}$ *and* $\mathsf{ok}_{\exists 1}$ *such that we can deduce in* LEGO*:*

$$\begin{array}{ll}
\vdash & \mathsf{ok}_{\forall 0} : \forall p : \mathbf{N} \to *.\,\mathsf{codes}(\mathbf{S}\,\mathbf{0})(\forall y : \mathbf{N}.(\mathsf{lt}\,y\mathbf{0}) \to (py)) \\
\vdash & \mathsf{ok}_{\forall 1} : \forall b : \mathbf{N}.\forall p : \mathbf{N} \to *.\forall t : \mathbf{N}.(\mathsf{codes}\,t(pb)\ \&\ \forall x : \mathbf{N}.(\mathsf{lt}\,xb) \to (px)) \\
& \to (\mathsf{codes}\,t(\forall x : \mathbf{N}.(\mathsf{lt}\,x(\mathbf{S}\,b)) \to (px))) \\
\vdash & \mathsf{ok}_{\exists 0} : \forall p : \mathbf{N} \to *.\,\mathsf{codes}\,\mathbf{0}(\exists y : \mathbf{N}.(\mathsf{lt}\,y\mathbf{0})\ \&\ (py)) \\
\vdash & \mathsf{ok}_{\exists 1} : \forall b : \mathbf{N}.\forall p : \mathbf{N} \to *.\forall t : \mathbf{N}.(\mathsf{codes}\,t(pb) \vee \exists x : \mathbf{N}.(\mathsf{lt}\,xb)\ \&\ (px)) \\
& \to (\mathsf{codes}\,t(\exists x : \mathbf{N}.(\mathsf{lt}\,x(\mathbf{S}\,b))\ \&\ (px)))
\end{array}$$

Proof:

1. *Finding* $\mathsf{ok}_{\forall 0}$ *and* $\mathsf{ok}_{\exists 0}$ *is easy.*
2. *We can find an inhabitant of the term*

$$\forall x : \mathbf{N}.\forall y : \mathbf{N}.(\mathsf{lt}\,x\,(\mathbf{S}\,y)) \to ((\mathsf{Q}\,x\,y) \vee (\mathsf{lt}\,x\,y))$$

Use this to find $\mathsf{ok}_{\forall 1}$ *and* $\mathsf{ok}_{\exists 1}$*.*

□

Lemmas 4.4.2, 4.4.3 and 4.4.4 enable us to perform induction on the structure of formulas. We get the following theorem, which tells us how to construct a correctness proof-object for any formula φ.

Theorem 4.4.5 *Let* $\varphi \in \mathrm{FORM}$ *and* $\Gamma = \{(x : \mathbf{N}) \mid x \in \mathrm{FV}(\varphi)\}$*. There exists a term* χ_φ *and a proof-object* ok_φ *such that in* LEGO*:*

$$\Gamma \vdash \mathsf{ok}_\varphi : \mathsf{codes}[\![\chi_\varphi]\!][\![\varphi]\!]$$

Proof:

The term χ_φ *is constructed inductively as follows:*

$$\chi_\varphi = \begin{cases}
\chi_<(T_1, T_2) & \textit{if } \varphi = T_1 \texttt{<} T_2 \\
\chi_=(T_1, T_2) & \textit{if } \varphi = T_1 \texttt{=} T_2 \\
f_\neg(\chi_\psi) & \textit{if } \varphi = \texttt{\~{}}\,\psi \\
f_\&(\chi_\psi, \chi_\xi) & \textit{if } \varphi = \psi\ \texttt{\&}\ \xi \\
f_\vee(\chi_\psi, \chi_\xi) & \textit{if } \varphi = \psi\ \texttt{v}\ \xi \\
f_\to(\chi_\psi, \chi_\xi) & \textit{if } \varphi = \psi\ \texttt{->}\ \xi \\
\mathrm{sg}(\prod_{x<T} \chi_\psi) & \textit{if } \varphi = \texttt{All(}x \texttt{<} T\texttt{)}\,\psi \\
\mathrm{sg}(\sum_{x<T} \chi_\psi) & \textit{if } \varphi = \texttt{Ex(}x \texttt{<} T\texttt{)}\,\psi
\end{cases}$$

Use induction on the structure of φ*:*

1. *Use lemma 4.4.2 in case* φ *is a basic formula. The term we are looking for is obtained by application of the appropriate function expression* ($\chi_<$ *or* $\chi_=$) *to the terms. The proof-object is constructed by applying the corresponding proof-object* ($\mathsf{ok}_<$ *or* $\mathsf{ok}_=$) *to the terms.*

2. *Use lemma 4.4.3 in case φ is a compound formula built from sub-formulas using one of the connectives $\neg$, &,$\vee$ or $\rightarrow$. The term we are looking for is obtained by application of the appropriate function expression ($f_\neg$,$f_\&$,$f_\vee$, or $f_\rightarrow$) to the terms we get from the induction hypothesis. The proof-object is constructed by application of the corresponding proof-object ($\mathsf{ok}_\neg$, $\mathsf{ok}_\&$, $\mathsf{ok}_\vee$ or $\mathsf{ok}_\rightarrow$) to the terms we get from the induction hypothesis.*
3. *If φ is a quantified formula we construct a function prescription to calculate the sum or product we saw above. Note that it would take a higher order function prescription to handle this case in the same way as the simple connectives. Such functions cannot be expressed in* PRIMF$^+$*. We construct a proof-object for a somewhat generalized proposition. Change the quantifier-bound by a fresh variable x and construct an induction-proof-object (using $\mathbf{R}_*$) on x. The two cases can be solved by using the proof-objects of lemma 4.4.4.*
4. *In case φ is an application of a predicate prescription P to some terms T_1,...,T_n we use the fact that a predicate prescriptions is an abstraction of a formula. So we have an application of an abstraction. By contracting this redex we get a less complex formula that is equivalent to φ on which we can apply our induction hypothesis.*

□

Corollary 4.4.6 *Let P be a predicate prescription in* PRIMP *with* $\mathrm{arity}(P) = k$. *There exists a function prescription f in* PRIMF$^+$ *and a proof-object* ok_P *such that in* LEGO*:*

$$\vdash \mathsf{ok}_P : \forall \vec{x} \in \mathbf{N}^k . \mathsf{codes}([\![f]\!]\vec{x})([\![P]\!]\vec{x})$$

Proof:

Take $\varphi = P(\vec{x})$ in theorem 4.4.5 and generalize.

□

We can now generate characteristic terms for any formula. Furthermore, we can generate correctness proof-objects for such terms. This enables us to prove any valid formula from FORM. Of course, we need to be able to prove that the characteristic term of such a formula actually has the property istrue. This is easy enough, however, because we constructed characteristic terms in such a way that they reduce to 1 if the formula is valid.

Corollary 4.4.7 *Let φ be a formula with* $\mathrm{FV}(\varphi) = \emptyset$ *such that* $\models \varphi$. *There exists a* LEGO*-expression p such that*

$$\vdash p : [\![\varphi]\!]$$

Proof:

Use theorem 4.4.5 to get

$$\vdash \mathsf{ok}_\varphi : \mathsf{codes}[\![\chi_\varphi]\!]\,[\![\varphi]\!]$$

From $\mathsf{codes}[\![\chi_\varphi]\!]\,[\![\varphi]\!]$ *we deduce* $(\mathsf{istrue}[\![\chi_\varphi]\!]) \to [\![\varphi]\!]$.
So we have to prove $\mathsf{istrue}[\![\chi_\varphi]\!]$. *Since* $\models \varphi$ *we have that*

$$[\![\chi_\varphi]\!] \twoheadrightarrow_{\beta\iota} \mathbf{S\,0}$$

This means that

$$\mathsf{Q}\,[\![\chi_\varphi]\!]\,(\mathbf{S\,0}) \twoheadrightarrow_{\beta\iota} \mathsf{Q}\,(\mathbf{S\,0})\,(\mathbf{S\,0})$$

We have a proof-object refl *such that*

$$\vdash \mathsf{refl} : \mathsf{Q}\,(\mathbf{S\,0})\,(\mathbf{S\,0})$$

since Leibniz' equality is reflexive. From the conversion-*rule it now follows that*

$$\vdash \mathsf{refl} : \mathsf{Q}\,[\![\chi_\varphi]\!]\,(\mathbf{S\,0})$$

which proves $\mathsf{istrue}[\![\chi_\varphi]\!]$.

□

Note that the size of the proof-object p we find does not depend on any numerical constants in the formula φ. It only depends on the structure of φ. In particular, if $\varphi = P(\underline{\vec{n}})$ like in corollary 4.4.6, then the size of p is not dependent on $\vec{n}$.

4.4.2 A proof-object generator generator

The proof-objects ok_φ we generate in 4.4.1 grow larger as φ grows more complex. Since the proof-objects are generated canonically (we have a computer program that generates them) we can construct a LEGO-expression that does the job for us. Such a term would be a translation of our program in Lambda calculus.

Theorem 4.4.8 *Let* $\#$: FORM $\to \mathbb{N}$ *be a suitable encoding function for* PRIMP. *Let* $\varphi \in$ FORM *and* $\Gamma = \{(x : \mathbf{N}) \mid x \in \mathrm{FV}(\varphi)\}$. *There exists a* LEGO*-term* ok *such that in* LEGO*:*

$$\Gamma \vdash \mathsf{ok}[\![\#\varphi]\!] : \mathsf{codes}[\![\chi_\varphi]\!][\![\varphi]\!]$$

By "a suitable encoding function" is meant that $\#$ should be an injection and that there should exist terms in LEGO which can interpret and decode a code-number. If we were to construct the term ok in the proof-checker LEGO we would surely *not* use natural numbers to encode FORM. Using an inductive type would be easier and more elegant.

This strong version of the theorem prover was not implemented. The implementation of the weaker version is described in the next chapter.

Chapter 5

Implementation aspects

5.1 Clean

The program described in chapter 4 was implemented using the functional language CLEAN (see [9]). In this chapter we will describe the different modules of the program and illustrate the implementation process by small fragments of code. It is not our intention to show all the details of the implementation.

5.2 Modules

The prover consists of six important modules:

1. *Lex*, the lexical analyzer.
2. *Parse*, the parser.
3. *CharFun*, compile characteristic function (term) for a predicate (formula).
4. *Proof*, generate proof-object.
5. *ToLego*, translate terms, functions, predicates and formulas to LEGO-strings.
6. *Types*, definitions for the type of terms, functions, predicates and formulas.

The flow of information through the modules can be seen in figure 5.1. Note that the module called *Types* does not occur in the figure. This module provides the definitions of the different types and some definitions of functions on them.

5.3 Types

Algebraic data-types in CLEAN are a convenient way to describe the inductively defined class of primitive recursive function prescriptions.

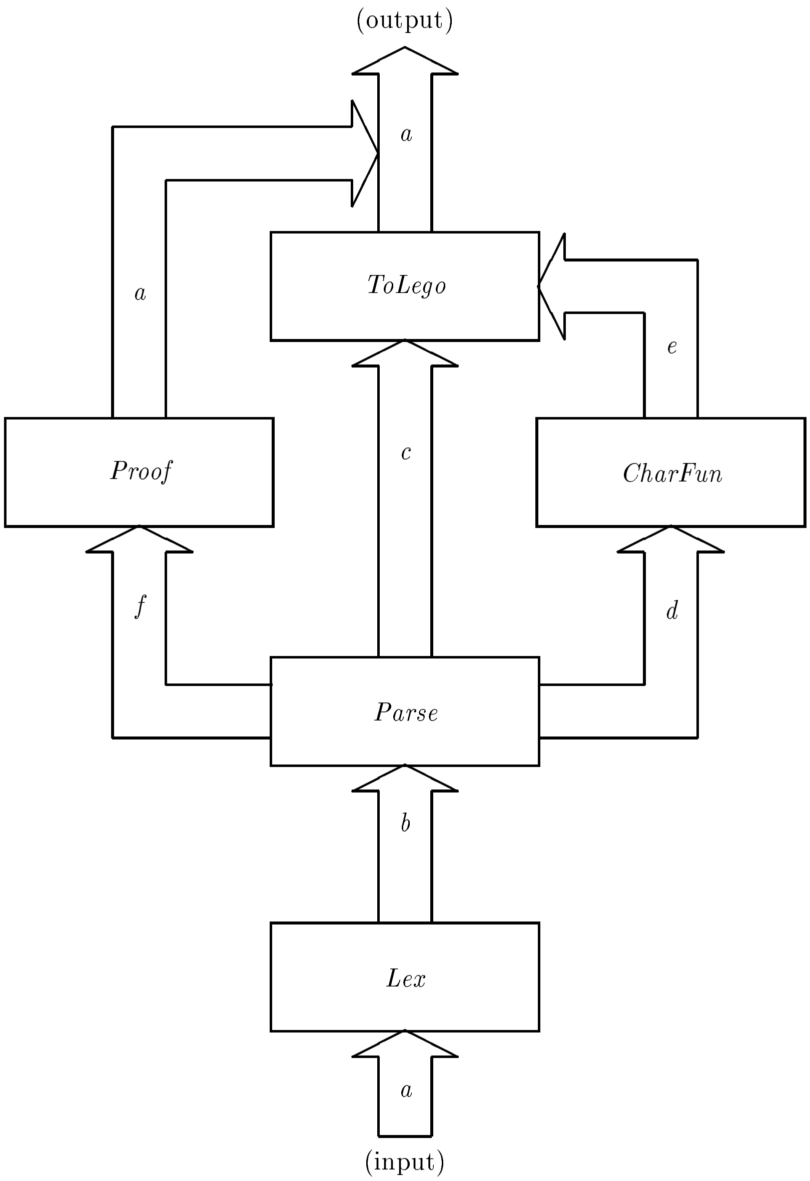


Figure 5.1: Flow of information through the different modules. *a*. Stream of characters. *b*. Stream of tokens. *c*. Stream of definitions (both function and predicate). *d*. Stream of predicate definitions. *e*. Stream of function definitions. *f*. The goal formula.

```
:: PRF  = Zero | Succ | Proj Int Int | FIdent String Int |
          Prim PRF PRF | Comp PRF [PRF] | FAbs [String] Term
:: Term = Const Int | Var String | FApp PRF [Term]
```

In the same manner the inductively defined class of primitive recursive predicates can be described.

```
:: PRP  = PIdent String Int | PAbs [String] Form
:: Form = Ls Term Term | Eq Term Term | Gr Term Term |
          Neg Form |
          Disj Form Form | Conj Form Form | Imp Form Form |
          All [(String,Term)] Form | Ex [(String,Term)] Form |
          PApp PRP [Term]
```

Using *pattern-matching* we can elegantly specify functions over the algebraic data-types. The function name `Arity` is overloaded. This can be done in CLEAN by defining it as a *class* and specifying instances for the different types:

```
class Arity a :: !a -> Int

instance Arity PRF
where
   Arity Zero         = 1
   Arity Succ         = 1
   Arity (Proj n k)   = n
   Arity (FIdent s n) = n
   Arity (FAbs xs t)  = length xs
   Arity (Prim h g)   = (Arity h) + 1
   Arity (Comp f gs)  = Arity gs
```

A combination of *infix operators* and overloaded function names leads to code that closely resembles the definitions from the specification (compare definitions 4.1.6 and 4.1.11):

```
class FV a :: !a -> [String]

instance FV Term
where
   FV (Const n)   = []
   FV (Var x)     = [x]
   FV (FApp f ts) = FV ts

instance FV Form
where
   FV (Ls t1 t2)         = (FV t1) u (FV t2)
   FV (Eq t1 t2)         = (FV t1) u (FV t2)
   FV (Gr t1 t2)         = (FV t1) u (FV t2)
   FV (Neg p)            = FV p
```

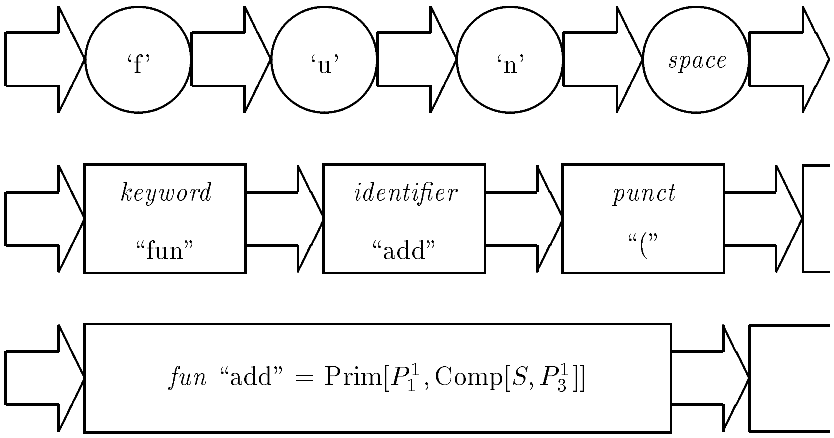


Figure 5.2: Lexical analysis and parsing of function definitions.

```
FV (Disj p1 p2)       = (FV p1) u (FV p2)
FV (Conj p1 p2)       = (FV p1) u (FV p2)
FV (Imp p1 p2)        = (FV p1) u (FV p2)
FV (All [] p)         = FV p
FV (All [(x,b):xs] p) = ((FV (All xs p)) u (FV b)) -- [x]
FV (Ex [] p)          = FV p
FV (Ex [(x,b):xs] p)  = ((FV (Ex xs p)) u (FV b)) -- [x]
FV (PApp p ts)        = FV ts
```

5.4 Parsing

The input is parsed in two stages: The lexical analysis and the actual parsing phase. Although the modules concerned with parsing are the largest modules in the application, their interfaces towards the other modules consist of only two functions.

```
Tokenize:: !*File -> (!*[Token],!*File)
Parse:: [Token] -> ([Def],Form)
```

During the lexical analysis the input consists of a stream of characters which is transformed into a stream of tokens. Tokens are the basic building blocks of our language. Example tokens are keywords like: `fun` and `Prim` or symbols like `<` but also identifiers like `add` or `prime`. The task of the lexical analyzer is to identify the tokens and to divide them into different categories (keyword, identifier, numeral, connective etc.). The lexical analysis phase consists itself of two phases. First multi-character tokens are assembled. Then they are assigned a category.

During the parsing phase the input is a stream of tokens which is transformed into a stream of definitions (a valid context of function and predicate definitions) followed by a formula (the goal to prove). The parser is a simple top-down parser with a *look-ahead* of at most three tokens. Top-down means that the parser will start from the start-symbol of a grammar of the language and tries to derive the current input. The look-ahead tokens are used in this process to decide which alternative to take. Here is the code to parse a function prescription, for example:

```
parsefunexp:: [Token] String Term [Def] -> (PRF,[Token])
parsefunexp []     f rec defs = abort "Unexpected eof\n"
parsefunexp [t:ts] f rec defs
   | isZero t   = (Zero,ts)
   | isSucc t   = (Succ,ts)
   | isProj t   = parseproj [t:ts]
   | isPrim t   = parseprim [t:ts] f rec defs
   | isComp t   = parsecomp [t:ts] f rec defs
   | isIdent t  = (FIdent f n, ts)
   | isBracel t = parseabstrfunexp [t:ts] f rec defs
                = abort "Not a function expression\n"
     where
        f = toString t
        n = lookupArity f defs
```

The input (a stream of tokens called `[t:ts]`) is supposed to denote a function prescription, so the first token `t` should be either `Z`, `S`, `P`, `Prim`, `Comp`, an identifier, or a left-brace (indicating an abstraction).

Writing a top-down parser for a formal language is very easy. All we have to do is write a good (affix-) grammar. The non-terminals in the grammar are translated to Clean -functions. Decisions about which alternative to take are based on look-ahead tokens.

5.5 Characteristic terms

The module *CharFun* exports two important functions.

```
toPRF:: !PRP [Def] -> PRF
toTerm:: !Form [Def] -> Term
```

The function `toPRF` translates a predicate prescription to a function prescription. Since predicate prescriptions are defined as abstractions of formulas, `toPRF` is very easy. It calls `toTerm` to translate the formula and then uses abstraction to obtain a function.

The function `toTerm` translates a formula to its corresponding characteristic term. This is done as described in chapter 4.

```
toTerm:: !Form [Def] -> Term
toTerm (Ls t1 t2)  ds = FApp ch_ls [t1,t2]
toTerm (Eq t1 t2)  ds = FApp f_nor [FApp ch_ls [t1,t2],
```

```
                                    FApp ch_ls [t2,t1]]
toTerm (Gr t1 t2)  ds = FApp ch_ls [t2,t1]
toTerm (Neg p)     ds = FApp sgbar [toTerm p ds]
toTerm (Conj p q)  ds = FApp f_and [toTerm p ds,toTerm q ds]
toTerm (Disj p q)  ds = FApp f_or  [toTerm p ds,toTerm q ds]
toTerm (Imp p q)   ds = FApp f_arr [toTerm p ds,toTerm q ds]
toTerm (PApp p ts) ds = FApp (toPRF p ds) ts
toTerm (All xs p)  ds = Product xs (toTerm p ds)
toTerm (Ex xs p)   ds = Sum     xs (toTerm p ds)
```

To translate quantified formulas we use `Product` and `Sum`. These functions construct a primitive recursive function prescription for the bounded product and sum over the characteristic term of the sub-formula of the quantified formula.

```
Product:: ![(String,Term)] Term -> Term
Product [] t          = t
Product [(x,b):xs] t = FApp (Prim s1 s2) [b:ys1]
where
   s1  = ConstantFun 1 n
   s2  = FAbs ["rec",x:ys]
         (FApp f_and [Product xs t, Var "rec"])
   n   = length ys
   ys1 = map Var ys
   ys  = (FV t) -- [x]
```

5.6 Proofs

In the module *Proof* the proof-object generating is done. This module exports only one function:

```
Proof:: !Form [String] [Def] -> String
```

which takes as input a formula, a list of the used variables so far and the current context of definitions. The output of this function are LEGO-tactics which instruct LEGO to build a proof-object for a correctness proof for the characteristic term of this formula as described in theorem 4.4.5. This is done by recursion on the algebraicly defined type of formulas. On basic formulas the appropriate lemma of 4.4.2 is called. On negations, conjunctions, disjunctions and implications we can use one of the lemmas of 4.4.3 applied to the characteristic terms of the sub formulas. After this, `Proof` is applied recursively on the sub formulas. The results of these are appended. Proof objects for formulas which have as main connective a quantifier are a little more difficult to generate. As in theorem 4.4.5 we replace the bound term by a fresh variable (not in the list of used variables) and generate an induction proof using lemma 4.4.4.

5.7 Performance

Our goal to generate small proof-objects that are hard to do manually has now been reached. However, replacing proof obligations by calculations has a drawback. Although the calculation does not add to the size of the proof-object, it does add to the time the proof-checker needs to check the types. In fact, we have tested the primality example on 2, 3 and 5 and it worked fine. It took our computer a very long time to check the proof-object for 11. And a primality check for 61 was not feasible.

What causes the low performance? Three important reasons are considered:

- The proof-checker LEGO is not designed as a fast implementation of a functional language. It is an interpreter and not a compiler and it has no special tricks to avoid reduction like CLEAN has (rewriting graphs instead of terms, strictness annotations etc.).
- We use a simplistic algorithm. The primitive recursive primality-check algorithm we generate from the predicate checks all divisors of n instead of only those up to $\sqrt{n}$. Whether m is a divisor of n is checked by trying to multiply m with all numbers less than n. This is not a very efficient way to test primes.
- The unary implementation of natural numbers we use is very ancient. Representing numbers using $\mathbf{0}$ and $\mathbf{S}$ may be very useful for proving something about natural numbers by induction but it is very difficult to write efficient algorithms using this representation. For example the addition we use in LEGO has a linear complexity measured in the size of the operands. This means an exponential complexity if we consider the number of bit-operations. Since most algorithms that we can express in our language use addition, we are bound to get slow calculations.

5.8 Improvements

How can we improve the behavior of the programs we generate? It is not the intention of a system like the prototype proof development system LEGO to be a fast implementation for a programming language. Correctness of type checking is more important than efficiency of reductions. Remember that mathematicians have to be able to fully trust their proof-checker. They rely on the correctness of LEGO. Tricks to improve the performance under water may make this more difficult. So improvements on the first point are hard to make.

On the second point there is also not much hope. The generated programs are inherently slow because of the fact that we generate them canonically from predicates and logical formulas. We could allow the user of our program to write a faster algorithm, but then we would also have to force him to give a correctness-proof of his algorithm relative to the predicate. This was not the intention of our program.

However, there is room for improvements on the third point. If we can find a suitable implementation for the decimal representation $\mathbf{N}_{10}$ of natural numbers, we could speed-up our basic algorithms considerably. But not only do we need a better representation of our type $\mathbf{N}$ we also have to give proof-objects for correctness proofs of this representation and for each of the fast versions of our basic algorithms. By a correctness proof of our representation we mean that there should exists terms ρ, ρ^{-1}, ok_ρ and $\mathsf{ok}_{\rho^{-1}}$ such that:

$$
\begin{aligned}
&\rho : \mathbf{N} \to \mathbf{N}_{10} \\
&\rho^{-1} : \mathbf{N} \to \mathbf{N}_{10} \\
&\mathsf{ok}_\rho : \forall n : \mathbf{N}.\, \mathsf{Q}(\rho^{-1}(\rho n))n \\
&\mathsf{ok}_{\rho^{-1}} : \forall n : \mathbf{N}_{10}.\, \mathsf{Q}(\rho(\rho^{-1} n))n
\end{aligned}
$$

This allows us to switch to $\mathbf{N}$ if we want to prove something by induction and back to $\mathbf{N}_{10}$ if we want to do calculations.

Chapter 6

Conclusions

In this thesis we have studied the interaction of proof checking with calculations. Informal mathematical proofs often contain (symbolic) calculations. The result of our calculation is needed to prove a goal. Yet the steps of the calculation are generally not considered to be part of the proof.

Proof development systems, however, force the user to spell out all the steps in the calculation. These are not satisfied until they clearly see that the result of the calculation actually helps to prove the goal. If the system is based on type theory (typed lambda calculus) there is a better way to do the calculations. Type theory can be regarded as a language in which one can express propositions and proofs, but it can also be seen as a functional language in which we can express calculations (algorithms) and types. So calculations that are part of a proof should be translated to algorithms. The fact that the algorithm equals its result is justified by the so-called *conversion-rule*. The reduction itself is proof enough.

From the ideas given above, a theorem prover was developed for a small but useful class of propositions. It was implemented in the functional language CLEAN. The prover takes as input a series of primitive recursive predicates (actually syntactical prescriptions of such predicates). It then generates for each predicate a characteristic function (prescription) that will reduce to 1 in case it is applied to arguments for which the predicate holds. Finally the prover canonically generates proof-objects which prove the correctness of the characteristic functions (algorithms) relative to the predicates. Propositions built up out of these predicates can now be proved automatically since the characteristic term of such a proposition will reduce to 1.

The prover can be used to prove trivial propositions like "5 is a prime". Since we first give a correctness-proof for the primality checker (which was generated out of the syntax of the prime predicate) the proof-object of "61 is a prime" is of the same size as that of "5 is a prime". The proof-checker (LEGO) however takes much longer to calculate the result of the primality checker applied to 61. In fact, testing the primality of 61 turns out to be infeasible. One way to try to improve the (complexity) behavior of the generated checking algorithms is to use a representation of the natural numbers that can be better employed to write faster algorithms.

Sample proof-objects can be found in the appendix.

Appendix A

Example proof-objects

To give an indication of the size of the proof-objects involved, here are some proof-objects that prove correctness statements cf. theorem 4.4.5. The following input was given to the prover:

```
fun plus(0,y)   =  y
    plus(x+1,y) =  S(plus(x,y))

pred even(n)    =  Ex (m<S(n)) plus(m,m) = n

goal ~even(5)
```

This resulted in the generation of LEGO-definitions for the various functions and predicates and for LEGO-tactics that instructed LEGO to generate the following proof-object:

```
[oddfive_ok =
  not_ok (even five) (ch_even five) (nat_elim ([x0:nat]codes (([y:nat]nat_elim
  ([_:nat]nat) (([_:nat]zero) zero) ([y'3,rec:nat]([rec'5,m:nat]f_or (f_nor (([
  x1,x2:nat]([y'9:nat]nat_elim ([_:nat]nat) zero ([y'10,rec'11:nat]([x1'12,x2'1
  3:nat]([x1'14:nat]succ (([_:nat]zero) x1'14)) (([_,x:nat]x) x1'12 x2'13)) rec
  '11 y'10) y'9) (rtbus x1 x2)) (plus m m) five) (([x1,x2:nat]([y'9:nat]nat_eli
  m ([_:nat]nat) zero ([y'10,rec'11:nat]([x1'12,x2'13:nat]([x1'14:nat]succ (([_
  :nat]zero) x1'14)) (([_,x:nat]x) x1'12 x2'13)) rec'11 y'10) y'9) (rtbus x1 x2
  )) five (plus m m))) rec'5) rec y'3) y) x0) (Ex ([m:nat]and (lt m x0) (Q (plu
  s m m) five)))) (lemma_ex0 ([x1:nat]Q (plus x1 x1) five)) ([x0:nat][ih_x0:cod
  es (nat_elim ([_:nat]nat) zero ([y,rec:nat]f_or (f_nor (nat_elim ([_:nat]nat)
   zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) five ([_,rec'5:nat]predec r
  ec'5) (nat_elim ([_:nat]nat) y ([_,rec'5:nat]succ rec'5) y))) (nat_elim ([_:n
  at]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y'4,_:nat]y'4
  ) (nat_elim ([_:nat]nat) zero ([y'4,_:nat]y'4) (nat_elim ([_:nat]nat) zero ([
  y'4,_:nat]y'4) (nat_elim ([_:nat]nat) zero ([y'4,_:nat]y'4) (nat_elim ([_:nat
  ]nat) zero ([y'4,_:nat]y'4) (nat_elim ([_:nat]nat) y ([_,rec'5:nat]succ rec'5
  ) y)))))))) rec) x0) (Ex ([m:nat]and (lt m x0) (Q (plus m m) five)))]lemma_ex
  1 x0 ([x1:nat]Q (plus x1 x1) five) (f_or (f_nor (nat_elim ([_:nat]nat) zero (
  [_,_:nat]succ zero) (nat_elim ([_:nat]nat) five ([_,rec:nat]predec rec) (nat_
  elim ([_:nat]nat) x0 ([_,rec:nat]succ rec) x0))) (nat_elim ([_:nat]nat) zero
  ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y,_:nat]y) (nat_elim ([_:n
  at]nat) zero ([y,_:nat]y) (nat_elim ([_:nat]nat) zero ([y,_:nat]y) (nat_elim
  ([_:nat]nat) zero ([y,_:nat]y) (nat_elim ([_:nat]nat) zero ([y,_:nat]y) (nat_
  elim ([_:nat]nat) x0 ([_,rec:nat]succ rec) x0)))))))) (nat_elim ([_:nat]nat)
  zero ([y,rec:nat]f_or (f_nor (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero)
   (nat_elim ([_:nat]nat) five ([_,rec'6:nat]predec rec'6) (nat_elim ([_:nat]na
```

```
  t) y ([_,rec'6:nat]succ rec'6) y))) (nat_elim ([_:nat]nat) zero ([_,_:nat]suc
  c zero) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) z
  ero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim (
  [_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5
  ) (nat_elim ([_:nat]nat) y ([_,rec'6:nat]succ rec'6) y)))))))) rec) x0)) (or_
  ok (Q (plus x0 x0) five) (Ex ([x:nat]and (lt x x0) (Q (plus x x) five))) (f_n
  or (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) fi
  ve ([_,rec:nat]predec rec) (nat_elim ([_:nat]nat) x0 ([_,rec:nat]succ rec) x0
  ))) (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) z
  ero ([y,_:nat]y) (nat_elim ([_:nat]nat) zero ([y,_:nat]y) (nat_elim ([_:nat]n
  at) zero ([y,_:nat]y) (nat_elim ([_:nat]nat) zero ([y,_:nat]y) (nat_elim ([_:
  nat]nat) zero ([y,_:nat]y) (nat_elim ([_:nat]nat) x0 ([_,rec:nat]succ rec) x0
  )))))))) (nat_elim ([_:nat]nat) zero ([y,rec:nat]f_or (f_nor (nat_elim ([_:na
  t]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) five ([_,rec'6:nat]pr
  edec rec'6) (nat_elim ([_:nat]nat) y ([_,rec'6:nat]succ rec'6) y))) (nat_elim
   ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y'5,_:n
  at]y'5) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) z
  ero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim (
  [_:nat]nat) zero ([y'5,_:nat]y'5) (nat_elim ([_:nat]nat) y ([_,rec'6:nat]succ
   rec'6) y)))))))) rec) x0) (eq_ok (plus x0 x0) five) ih_x0)) (succ five)):cod
  es (sgbar (ch_even five)) (not (even five))
];
```

Now, let us return to our example of the primality checker. We provide the prover with the following input:

```
fun plus(0,y)      =  y
    plus(x+1,y)    =  S(plus(x,y))

fun times(0,y)     =  0
    times(x+1,y)   =  plus(times(x,y),y)

pred divides(n,m)  =  Ex (d<S(m)) times(d,n)=m

pred prime(n)      =  n>1 & (All (d<n) divides(d,n) -> d=1)

goal prime(n)
```

The prover will generate characteristic function prescriptions for `divides` and `prime`. An expanded version of the latter function prescription, expressed in PRIMF:

```
fun ch_prime =
   Comp[Prim[Z,P[3,3]],Comp[Comp[Prim[P[0,0],Comp[Comp[S,Z],P[2
   ,2]]],Prim[P[1,1],Comp[Prim[P[0,0],P[2,2]],P[3,1]]]],Comp[S,
   Z],P[1,1]],Comp[Prim[Comp[S,Z],Comp[Prim[Z,P[3,3]],Comp[Prim
   [Comp[S,Z],P[3,3]],Comp[Comp[Prim[Comp[Z,P[2,2]],Comp[Prim[P
   [1,1],Comp[Comp[S,Z],P[3,3]]],Comp[Prim[Prim[Comp[S,P[0,0]],
   Comp[Z,P[2,2]]],Comp[Z,P[3,3]]],Comp[Comp[Prim[P[0,0],Comp[C
   omp[S,Z],P[2,2]]],Prim[P[1,1],Comp[Prim[P[0,0],P[2,2]],P[3,1
   ]]]],Comp[Prim[Z,Comp[Prim[P[1,1],Comp[S,P[3,1]]],P[3,1],P[3
   ,3]]],P[4,2],P[4,4]],P[4,3]],Comp[Comp[Prim[P[0,0],Comp[Comp
   [S,Z],P[2,2]]],Prim[P[1,1],Comp[Prim[P[0,0],P[2,2]],P[3,1]]]
   ],P[4,3],Comp[Prim[Z,Comp[Prim[P[1,1],Comp[S,P[3,1]]],P[3,1]
   ,P[3,3]]],P[4,2],P[4,4]]]],P[4,1]]],Comp[S,P[2,2]],P[2,2],P[
   2,1]],P[3,2],P[3,3]],Comp[Prim[Prim[Comp[S,P[0,0]],Comp[Z,P[
```

```
2,2]]],Comp[Z,P[3,3]]],Comp[Comp[Prim[P[0,0],Comp[Comp[S,Z],
P[2,2]]],Prim[P[1,1],Comp[Prim[P[0,0],P[2,2]],P[3,1]]]],P[3,
2],Comp[S,Comp[Z,P[3,3]]]],Comp[Comp[Prim[P[0,0],Comp[Comp[S
,Z],P[2,2]]],Prim[P[1,1],Comp[Prim[P[0,0],P[2,2]],P[3,1]]]],
Comp[S,Comp[Z,P[3,3]]],P[3,2]]]],P[3,1]]],P[1,1],P[1,1]]]
```

Next the prover will generate a proof-object `prime_ok` for the correctness statement:

```
[prime_ok =
  [n:nat]and_ok (gt n one) ({d:nat}(lt d n)->(divides d n)->Q d one) (([x1,x2:n
  at]([y:nat]nat_elim ([_:nat]nat) zero ([y'5,rec:nat]([x1'7,x2'8:nat]([x1'9:na
  t]succ (([_:nat]zero) x1'9)) (([_,x:nat]x) x1'7 x2'8)) rec y'5) y) (rtbus x1
  x2)) one n) (([y,x1:nat]nat_elim ([_:nat]nat) (([x1'4:nat]succ (([_:nat]zero)
   x1'4)) x1) ([y'4,rec:nat]([rec'6,d,n'8:nat]f_and (f_arr (ch_divides d n'8) (
  f_nor (([x1'9,x2:nat]([y'11:nat]nat_elim ([_:nat]nat) zero ([y'12,rec'13:nat]
  ([x1'14,x2'15:nat]([x1'16:nat]succ (([_:nat]zero) x1'16)) (([_,x:nat]x) x1'14
   x2'15)) rec'13 y'12) y'11) (rtbus x1'9 x2)) d one) (([x1'9,x2:nat]([y'11:nat
  ]nat_elim ([_:nat]nat) zero ([y'12,rec'13:nat]([x1'14,x2'15:nat]([x1'16:nat]s
  ucc (([_:nat]zero) x1'16)) (([_,x:nat]x) x1'14 x2'15)) rec'13 y'12) y'11) (rt
  bus x1'9 x2)) one d))) rec'6) rec y'4 x1) y) n n) (gr_ok n one) (nat_elim ([x
  0:nat]codes (([y,x1:nat]nat_elim ([_:nat]nat) (([x1'5:nat]succ (([_:nat]zero)
   x1'5)) x1) ([y'5,rec:nat]([rec'7,d,n'9:nat]f_and (f_arr (ch_divides d n'9) (
  f_nor (([x1'10,x2:nat]([y'12:nat]nat_elim ([_:nat]nat) zero ([y'13,rec'14:nat
  ]([x1'15,x2'16:nat]([x1'17:nat]succ (([_:nat]zero) x1'17)) (([_,x:nat]x) x1'1
  5 x2'16)) rec'14 y'13) y'12) (rtbus x1'10 x2)) d one) (([x1'10,x2:nat]([y'12:
  nat]nat_elim ([_:nat]nat) zero ([y'13,rec'14:nat]([x1'15,x2'16:nat]([x1'17:na
  t]succ (([_:nat]zero) x1'17)) (([_,x:nat]x) x1'15 x2'16)) rec'14 y'13) y'12)
  (rtbus x1'10 x2)) one d))) rec'7) rec y'5 x1) y) x0 n) ({d:nat}(lt d x0)->(di
  vides d n)->Q d one)) (lemma_all0 ([x1:nat](divides x1 n)->Q x1 one)) ([x0:na
  t][ih_x0:codes (nat_elim ([_:nat]nat) (succ zero) ([y,rec:nat]f_and (f_arr (c
  h_divides y n) (f_nor (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_e
  lim ([_:nat]nat) one ([_,rec'6:nat]predec rec'6) y)) (nat_elim ([_:nat]nat) z
  ero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y'5,_:nat]y'5) y)))) r
  ec) x0) ({d:nat}(lt d x0)->(divides d n)->Q d one)]lemma_all1 x0 ([x1:nat](di
  vides x1 n)->Q x1 one) (f_and (f_arr (ch_divides x0 n) (f_nor (nat_elim ([_:n
  at]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) one ([_,rec:nat]pred
  ec rec) x0)) (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:n
  at]nat) zero ([y,_:nat]y) x0)))) (nat_elim ([_:nat]nat) (succ zero) ([y,rec:n
  at]f_and (f_arr (ch_divides y n) (f_nor (nat_elim ([_:nat]nat) zero ([_,_:nat
  ]succ zero) (nat_elim ([_:nat]nat) one ([_,rec'7:nat]predec rec'7) y)) (nat_e
  lim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y'6,
  _:nat]y'6) y)))) rec) x0)) (and_ok ((divides x0 n)->Q x0 one) ({x:nat}(lt x x
  0)->(divides x n)->Q x one) (f_arr (ch_divides x0 n) (f_nor (nat_elim ([_:nat
  ]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) one ([_,rec:nat]predec
   rec) x0)) (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat
  ]nat) zero ([y,_:nat]y) x0)))) (nat_elim ([_:nat]nat) (succ zero) ([y,rec:nat
  ]f_and (f_arr (ch_divides y n) (f_nor (nat_elim ([_:nat]nat) zero ([_,_:nat]s
  ucc zero) (nat_elim ([_:nat]nat) one ([_,rec'7:nat]predec rec'7) y)) (nat_eli
  m ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) zero ([y'6,_:
  nat]y'6) y)))) rec) x0) (arr_ok (divides x0 n) (Q x0 one) (ch_divides x0 n) (
  f_nor (([x1,x2:nat]([y:nat]nat_elim ([_:nat]nat) zero ([y'7,rec:nat]([x1'9,x2
  '10:nat]([x1'11:nat]succ (([_:nat]zero) x1'11)) (([_,x:nat]x) x1'9 x2'10)) re
  c y'7) y) (rtbus x1 x2)) x0 one) (([x1,x2:nat]([y:nat]nat_elim ([_:nat]nat) z
  ero ([y'7,rec:nat]([x1'9,x2'10:nat]([x1'11:nat]succ (([_:nat]zero) x1'11)) ((
  [_,x:nat]x) x1'9 x2'10)) rec y'7) y) (rtbus x1 x2)) one x0)) (nat_elim ([x1:n
  at]codes (([y,x1'6,x2:nat]nat_elim ([_:nat]nat) (([x1'8,x2'9:nat]([_:nat]zero
  ) (([_,x:nat]x) x1'8 x2'9)) x1'6 x2) ([y'8,rec:nat]([rec'10,d,n'12,x0'13:nat]
  f_or (f_nor (([x1'14,x2'15:nat]([y'16:nat]nat_elim ([_:nat]nat) zero ([y'17,r
  ec'18:nat]([x1'19,x2'20:nat]([x1'21:nat]succ (([_:nat]zero) x1'21)) (([_,x:na
  t]x) x1'19 x2'20)) rec'18 y'17) y'16) (rtbus x1'14 x2'15)) (times d x0'13) n'
  12) (([x1'14,x2'15:nat]([y'16:nat]nat_elim ([_:nat]nat) zero ([y'17,rec'18:na
  t]([x1'19,x2'20:nat]([x1'21:nat]succ (([_:nat]zero) x1'21)) (([_,x:nat]x) x1'
  19 x2'20)) rec'18 y'17) y'16) (rtbus x1'14 x2'15)) n'12 (times d x0'13))) rec
  '10) rec y'8 x1'6 x2) y) x1 n x0) (Ex ([d:nat]and (lt d x1) (Q (times d x0) n
  )))) (lemma_ex0 ([x3:nat]Q (times x3 x0) n)) ([x1:nat][ih_x1:codes (nat_elim
  ([_:nat]nat) zero ([y,rec:nat]f_or (f_nor (nat_elim ([_:nat]nat) zero ([_,_:n
```

```
  at]succ zero) (nat_elim ([_:nat]nat) n ([_,rec'8:nat]predec rec'8) (nat_elim
  ([_:nat]nat) zero ([_,rec'8:nat]plus rec'8 x0) y))) (nat_elim ([_:nat]nat) ze
  ro ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) (times y x0) ([_,rec'8:nat]pre
  dec rec'8) n))) rec) x1) (Ex ([d:nat]and (lt d x1) (Q (times d x0) n)))]lemma
  _ex1 x1 ([x3:nat]Q (times x3 x0) n) (f_or (f_nor (nat_elim ([_:nat]nat) zero
  ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) n ([_,rec:nat]predec rec) (nat_el
  im ([_:nat]nat) zero ([_,rec:nat]plus rec x0) x1))) (nat_elim ([_:nat]nat) ze
  ro ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) (times x1 x0) ([_,rec:nat]pred
  ec rec) n))) (nat_elim ([_:nat]nat) zero ([y,rec:nat]f_or (f_nor (nat_elim ([
  _:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) n ([_,rec'9:nat]p
  redec rec'9) (nat_elim ([_:nat]nat) zero ([_,rec'9:nat]plus rec'9 x0) y))) (n
  at_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) (times
  y x0) ([_,rec'9:nat]predec rec'9) n))) rec) x1)) (or_ok (Q (times x1 x0) n) (
  Ex ([x:nat]and (lt x x1) (Q (times x x0) n))) (f_nor (nat_elim ([_:nat]nat) z
  ero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) n ([_,rec:nat]predec rec) (na
  t_elim ([_:nat]nat) zero ([_,rec:nat]plus rec x0) x1))) (nat_elim ([_:nat]nat
  ) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) (times x1 x0) ([_,rec:nat]
  predec rec) n))) (nat_elim ([_:nat]nat) zero ([y,rec:nat]f_or (f_nor (nat_eli
  m ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) n ([_,rec'9:n
  at]predec rec'9) (nat_elim ([_:nat]nat) zero ([_,rec'9:nat]plus rec'9 x0) y))
  ) (nat_elim ([_:nat]nat) zero ([_,_:nat]succ zero) (nat_elim ([_:nat]nat) (ti
  mes y x0) ([_,rec'9:nat]predec rec'9) n))) rec) x1) (eq_ok (times x1 x0) n) i
  h_x1)) (succ n)) (eq_ok x0 one)) ih_x0)) n):{n:nat}codes (ch_prime n) (prime
  n)
];
```

References

[1] Henk Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, second, revised edition, North-Holland, Amsterdam, 1984

[2] Henk Barendregt, *Lambda Calculi with Types*, Handbook of Logic in Computer Science, Volume II, 1992

[3] Henk Barendregt, *The quest for correctness*, CSI Nijmegen University, Department of Software Technology CWI Amsterdam, 1995

[4] Erik Barendsen, *Representation of Logic, Data Types and Recursive Functions in Typed Lambda Calculi*, Masters Thesis, Faculteit der Wiskunde en Informatica, Katholieke Universiteit Nijmegen, 1989

[5] Jean-Yves Girard, Yves Lafont & Paul Taylor, *Proofs and Types*, Cambridge University Press, 1989

[6] Per Martin-Löf *Intuitionistic type theory*, Napoli: Bibliopolis, 1984

[7] Zhaohui Luo, *ECC, an Extended Calculus of Constructions*, Department of Computer Science, University of Edinburgh

[8] Zhaohui Luo, Robert Pollack, *LEGO Proof Development System: User's Manual*, Department of Computer Science, University of Edinburgh, 1992 (1993,1994)

[9] Rinus Plasmeijer & Marko van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Publishers Ltd., 1993

[10] Helmut Schwichtenberg, *Definierbare Funktionen im λ-Kulkül mit Typen*, Arch.math.Logik 17 1976, 113–114

[11] Wim Veldman, *Berekenbaar en Bewijsbaar (Recursietheorie)*, Collegedictaat Katholieke Universiteit Nijmegen