ISBN 0797-6410

Course-of-values Recursion in Martin-Löf's Type Theory

Patricia Peratto

MASTER THESIS

Technical Report INCO-91-04
December 1991

Programa de Desarrollo de Ciencias Básicas
PEDECIBA Informática

In.Co., Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

Course-of-values Recursion in Martin-Löf's Type Theory

Patricia Peratto
peratto@incouy.edu.uy

Programa de Desarrollo de Ciencias Básicas
PEDECIBA Informática

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay

December 1991

Abstract

To facilitate reasoning about programs in Martin-Löf's Type Theory, the introduction of general recursion operators has been studied. This allows a more conventional programming style, being possible to separate the termination proof of the program from the proof of other properties.

This work experiments with this idea studying Course-of-Values induction for Natural Numbers. An induction rule is derived from natural induction, obtaining an expression for a recursion operator. A propositional equality that expresses the behavior of the obtained operator is proved. This equality will be usefull when proving properties of programs.

ALF is used as editor of Martin-Löf's monomorphic Set Theory and checker of all the proofs.

1 Introduction

In recent years several formalisms for program construction have been introduced. Programming logics becomes an important area of study. A programming logic is a conceptual framework which allows to express specifications and programs and to verify their correspondence. Editors and automatic tools for using Programming Logics have been developed. These tools allow us to automatically check if one program satisfies its specification.

Martin-Löf's Type Theory [Löf84] (TT) was developed as a work in mathematics and philosophy more than a work in programming theory. The intention was to clarify the meaning of constructive mathematics. The identification of propositions and sets allows the view of Martin-Löf's Type Theory as a logic for Programming: a proposition expresses a specification and a proof of the proposition is a program satisfying the specification.

Martin-Löf's Type Theory as a theory of programs, is a theory for total correctness. The type structure is very rich, being possible to completely express the task of a program. The programming language of TT is like typed lambda calculus and all well-typed programs are normalizable.

There are no explicit operators of general recursion in TT. Recursion is included into the language by means of primitive constants expressing primitive recursion.

To facilitate reasoning about programs in TT the introduction of general recursion operators has been studied. This allows a more conventional programming style, being possible to separate the termination proof of the program from the proof of other properties.
In [Smi83] a course-of-values induction rule for Lists is proved. Quicksort is defined in type theory using this rule. Paulson [Pau86] formalizes induction and recursion over a wide class of well-founded relations. Some well-founded relations are constructed from simpler ones, using rules that preserve the well-foundness property. Nordström [Nor88] describes a way to allow a general recursion operator in a version of type theory extended with propositions and subsets. He presents a proof rule for a course-of-values recursion operator and shows how to generalize the idea to an arbitrary set well-founded by a relation.

This work experiments with this idea studying Course-of-Values induction for Natural Numbers. An induction rule is derived from natural induction, obtaining an expression for a recursion operator. A propositional equality that expresses the behavior of the obtained operator is proved. This equality will be usefull when proving properties of programs. ALF [ACN90] is used as editor of Martin-Löf's monomorphic Set Theory and checker of all the proofs.

The rest of the paper is organized as follows:

Section 2 presents basic concepts concerning primitive recursion and course-of-values recursion.

In section 3 notational conventions and basic sets needed in the rest of the work are presented.

Section 4 describes the introduction of course-of-values recursion into type theory. Induction rules and recursion rules for the schema are presented and some applications of the operator to programming are showed.

In section 5 a course-of-values induction rule is derived in type theory, giving an expression for a recursion operator. A propositional equality expressing a correspondence between the obtained expression and the recursion rule previously defined is showed.

Section 6 comments the work, describing the alternatives followed in its development.

In the appendix, the complete formalization of these results is given, as it is seen when using ALF.

2 Primitive Recursion and Course of Values Recursion

In the first part of this section, Primitive Recursion and Course of Values recursion are briefly described. Then course-of-values recursion is defined in terms of primitive recursion. The functions used in this section range over Natural Numbers. In the definitions, the usual order of generation of the Natural Numbers is considered. For a more detailled explanation see [Her65, Hen77].

2.1 Primitive Recursion

In a definition by primitive recursion, the value of a function at one argument is specified in terms of its predecessor and its functional value. The definition takes the form of:

$$\begin{aligned}&F(0) = a\\&F(s(x)) = H(x, F(x))\end{aligned}$$

From now on, a function like H will be called a *step function.*

The more relevant characteristics of Primitive Recursion are:

- the defined function is computable and total.
- the schema of definition corresponds closely to the induction principle for natural numbers, easing to reason about the functions defined.

2.2 Course of Values Recursion

In a course-of-values definition, the value of a function at one argument is specified in terms of one or more preceding values. A typical example is the definition of the Fibonacci function *Fib*:

$$\begin{aligned}&Fib(0) = 1\\&Fib(1) = 1\\&Fib(s(x)) = Fib(x) + Fib(pred(x)) \qquad \text{where } x \geq 1\end{aligned}$$

In arithmetics, we can talk about finite sequences of natural numbers via suitable codings. One of such codings is Gödel Numbering, that gives the schema:

$$\begin{array}{l} F(0) = d \\ F(s(x)) = e(x, \psi(x)) \end{array}$$

where

$$\psi(x) = \prod_{i=0}^{x} \quad prime(i)^{F(i)}$$

prime is a function that defines the prime numbers, being $prime(0) = 2$.
$\psi(x)$ computes the Gödel number for the values $F(0), \ldots, F(x)$;

Gödel Numbering characteristics are:

- ψ can be defined by primitive recursion.
- an *extraction function* E to decode Gödel numbers can be defined. This is a function that given i, $0 \leq i \leq x$, obtains the exponent of $prime(i)$ in $\psi(x)$.
- the extraction function E can be defined by primitive recursion.

2.3 Course of Values definitions are Primitive Recursive

Any course-of-values definition can be expressed with primitive recursion. A primitive recursive definition of the function ψ in terms of d and e is:

$$\begin{array}{l} \psi(0) = 2^d \\ \psi(s(x)) = \psi(x) * prime(s(x))^{e(x, \psi(x))} \end{array}$$

To compute a function defined like F, it is sufficient to decode the function ψ obtaining the exponent of the prime number corresponding to the argument. Then F can be defined as:

$$F(x) = E(x, \psi(x))$$

This result permits to generalize the properties verified by primitive recursive definitions to course-of-values definitions. In particular:

- any function defined by course of values recursion will be computable and total.
- since the schema corresponds closely to the complete induction principle, it can be used to reason about functions.
- this increases the clarity and expressivity of function definitions.

3 Notation and Basic Sets

In this section, we present some notational conventions of Martin-Löf's Type Theory. Refer to [NPS90] for a complete explanation of Martin-Löf's Type Theory.

3.1 Notational Conventions

In intuitionistic predicate logic, a proposition is identified with the set of its proofs. The judgements $A\ prop$ and $A\ true$ are used when regarding a set as a proposition. In the last case, the proof object is omitted.

The notation used for expressions is as follows: if e is an expression and x a variable, then $[x]e$ is used for abstraction of x from e. If e and t are expressions then $e(t)$ is used for the application of e to t.

To introduce definitions, the notation $a \equiv_{def} b$ is used.

Assumptions are of the form $[x \in A]$, where x is a variable and A is a set.

The rules are formulated in a natural deduction style

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C}$$

where the premises $P_1, P_2, \ldots, P_n$ and the conclusion C are (possibly hypothetical) judgements. In a rule, only the assumptions that are discharged are written. The premises that are obvious for the context are omitted (for example, if $a \in A$ is a premise, A set is excluded).

The proposition $Id(A, a, b)$ is written also as $a =_A b$. $\neg(A)$ denotes $A \Rightarrow \perp$.

The notation used for types, constants and rules is the same of [NPS90] polymorphic version of TT. In the appendix these results are formalized in the monomorphic ST.

3.2 Basic sets and Relations

Besides the basic sets of type theory, some new inductively defined sets are used in this work. The order for Natural Numbers is needed in the formalization of the course-of-values recursion rule. In [Sza91] definitions for $<$ and $\leq$ are introduced and their properties proved. I will use these definitions for $<$ and $\leq$.

The following properties of $=_A$, $<$ and $\leq$ are used in the proofs and examples:

Assume $A\ set$, $P(x)\ set\ [x \in A]$, $B\ set$, $f(x) \in B\ [x \in A]$, a, b $\in$ A.

$a =_A a$ (reflexivity)

$a =_A b \supset b =_A a$ (symmetry)

$(a =_A b \;\&\; b =_A c) \supset a =_A c$ (transitivity)

$(a =_A b \;\&\; P(a)) \supset P(b)$ (substitutivity)

$a =_A b \supset f(a) =_B f(b)$ (congruence)

Assume m, n, $p \in N$.

$\neg(n < 0)$

$n < s(n)$

$n < s(m) \supset n \leq m$

$n \leq m \supset (n < m \vee n =_N m)$

$n \leq m \vee m < n$

$(n < m \wedge m < p) \supset n < p$

$(n =_N m \wedge m < p) \supset n < p$

In the applications, the following properties for expressions containing the operators $+$, $-$ and $*$ are also used. These properties, are interpretations into the theory of axioms and theorems of formal number theories and primitive recursive arithmetics.

Let k, m, n, $p \in N$.

$n + 0 =_N n$

$n + m =_N m + n$

$n =_N m \supset n + p =_N m + p$

$n =_N m \supset p + n =_N p + m$

$(n =_N m \wedge k =_N p) \supset n + k =_N m + p$

$n + m + p =_N n + (m + p)$

$s(n) \leq m \supset m - s(n) < m$

$n \leq m \supset (m - n) + n =_N m$

$n * 0 =_N 0$

$n * m =_N m * n$

$(n =_N m) \supset n * p =_N m * p$

$(n =_N m) \supset p * n =_N p * m$

4 Course of Values Recursion in Type Theory

In this section, a course-of-values induction rule and a computation rule for a course-of-values operator are introduced to TT.

4.1 Course of Values Recursion and Complete Induction

To express in TT a course-of-values recursive definition, a representation of all the functional values for arguments less than a given one must be found. A convenient way to do this, is to define a function that *stores* all the functional values for arguments less than the given argument [Nor88]. This is a function like:

$$y \in \prod z \in N.(z \leq x \Rightarrow C(z))$$

The *step function e(x)*, will take as argument y to give the solution for $s(x)$. Then $e(x)$ is a function element with type:

$$(\prod z \in N.(z \leq x \Rightarrow C(z))) \Rightarrow C(s(x))$$

The course-of-values induction rule can be stated as:

Course-of-Values Induction rule - 1

$$\frac{\begin{array}{l} p \in N \\ C(v)\ set\ [v \in N] \\ d \in C(0) \\ apply(e(x), y) \in C(s(x))\ [x \in N,\ y \in \prod z \in N.(z \leq x \Rightarrow C(z))] \end{array}}{covrec(p, d, e) \in C(p)}$$

The operator *covrec* makes it possible, course-of-values recursive definitions. The value of $covrec(p, d, e)$ is obtained in the following way:

Evaluate p to canonical form.

1. If the value of p is 0 then the value of $covrec(p, d, e)$ is the value of d.

2. If the value of p is $s(a)$ then the value of $covrec(p, d, e)$ is the value of $apply(e(a), \lambda z.\lambda q.covrec(z, d, e))$.

The justification of the rule is based on the semantics of TT and the computation rule for *covrec*:

That $p \in N$ means that when computed, p gives as value 0 or $s(a)$ where $a \in N$.

1. If the value of p is 0 then the value of $covrec(p, d, e)$ is the value of d which solves the problem $C(0)$.

2. If the value of p is $s(a)$ then the value of $covrec(p, d, e)$ is the value of $apply(e(a), \lambda z.\lambda q.covrec(z, d, e))$ which solves the problem $C(s(a))$, if $\lambda z.\lambda q.covrec(z, d, e) \in \prod z \in N.(z \leq a \Rightarrow C(z))$.

   To know this, we have to know that $covrec(z, d, e) \in C(z)\ [z \in N,\ q \in z \leq a]$ and to know that $covrec(z, d, e) \in C(z)\ [z \in N,\ q \in z \leq a]$, we have to know that $covrec(n, d, e) \in C(n)\ [q \in n \leq a]$ for an arbitrary $n \in N$.

   So,

   - If the value of n is 0, reasoning as in (1) we prove that the value of $covrec(n, d, e)$ solves the problem $C(0)$.
   - If the value of n is $s(b)$ we proceed as in (2) to show that the value of $apply(e(b), \lambda z.\lambda q.covrec(z, d, e))$ solves the problem $C(s(b))$.

   Notice the importance of the restriction imposed by $q \in n \leq a$. Although q's value is not used in computations, is its existence that guarantee that at most in a steps this method terminate.

Another schema for course-of-values induction, is given by a step function which takes a solution of all arguments strictly smaller than x to a solution of $C(x)$. The corresponding rule is:

Course-of-Values Induction rule - 2

$$\frac{\begin{array}{l} p \in N \\ C(v)\ set\ [v \in N] \\ apply(e(x), y) \in C(x)\ [x \in N,\ y \in \prod z \in N.(z < x \Rightarrow C(z))] \end{array}}{covrec(p, e) \in C(p)}$$

where the value of $covrec(p, e)$ is the value of $apply(e(p), \lambda z.\lambda q.covrec(z, e))$.

There is nothing in this rule which is particular to the set of Natural Numbers. The rule reflects that the natural numbers are well-ordered by $<$. So, we can generalize the rule to an arbitrary set A which is well-founded by a relation $\prec_A$ [Nor88]. The computation rule remains the same.

If some of the constructions in the rule are omitted, we obtain the rule for complete induction:

Complete Induction rule

$$\frac{\begin{array}{l} p \in N \\ C(v)\ prop\ [v \in N] \\ C(x)\ true\ [x \in N,\ C(z)\ true\ [z \in N,\ z < x\ true\ ]\ ] \end{array}}{C(p)\ true}$$

The justification of this rule comes from Course-of-Values Induction rule-2.

4.2 Applications

To easy notation the following abbreviations will be used in this subsection:

let $x,\ z \in N,\ q \in z < x,$
$\quad y \in \prod z \in N.(z < x) \Rightarrow C(z),$
$\quad e(x) \in (\prod z \in N.(z \leq x \Rightarrow C(z))) \Rightarrow C(s(x))$

we will write $e(x, y)$ instead of $apply(e(x), y)$ and $y(z, q)$ instead of $apply(apply(y, z), q)$.

4.2.1 natrec in terms of covrec

The introduction of covrec, permits to replace the operator for primitive recursion natrec with an operator for pattern matching, allowing a programming style much more similar to the conventional in functional programming languages. So, we will define a primitive constant natcases expressing pattern-matching over Natural Numbers, and show that the primitive recursion operator natrec can be reduced to natcases and covrec [Nor88].

The primitive constant *natcases* expresses pattern-matching over natural numbers as follows:

The value of $natcases(p, d, e)$ is computed by first computing the value of p.

1. If the value of p is 0 then the value of $natcases(p, d, e)$ is the value of d.

2. If the value of p is $s(a)$ then the value of $natcases(p, d, e)$ is the value of $e(a)$.

Now, *natrec* can be expressed in terms of *natcases* and *covrec* as:

$natrec(p, d, e) \equiv_{def} covrec(p, [x]natcases(x,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u)))))$

where $lesssucc(u) \in u < s(u)\ [u \in N]$.

If we define *natrec* in this way, the expression $natrec(p, d, e)$ will be computed in the same way that the traditional primitive recursion operator. This can be shown as follows:

The program $natrec(p, d, e)$ is definitionally equal to:

$$covrec(p, [x]natcases(x,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u)))))$$

This expression is computed by computing the value of:

$apply(natcases(p,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u)))),$
$\quad \lambda z.\lambda q.covrec(z, [x]natcases(x,\ \lambda\ y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u)))))$

To compute the natcases expression we compute p.

1. If the value of p is 0 we compute:

   $$\begin{aligned}&apply(\ \lambda y.d,\\ &\qquad \lambda z.\lambda q.covrec(z, [x]natcases(x,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u))))))\end{aligned}$$

   which by the computation rule for apply yields d.

2. If the value of p is $s(a)$ we compute the value of the program:

   $$\begin{aligned}&apply(\ \lambda y.e(a, y(a, lesssucc(a)))),\\ &\qquad \lambda z.\lambda q.covrec(z, [x]natcases(x,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u))))))\end{aligned}$$

   which by the computation rule for apply yields

   $$e(a, covrec(a, [x]natcases(x,\ \lambda y.d,\ [u]\ \lambda y.e(u, y(u, lesssucc(u))))))$$

   which by the given definition, is definitionally equal to

   $$e(a, natrec(a, d, e)).$$

4.2.2 The Fibonacci function

The Fibonacci function can be defined as:

```
Fib(n) ≡_def covrec(n,
                  [x]natcases(x,
                              λy.1,
                              [a]natcases(a,
                                          λy.1,
                                          [b] λy. y(s(b),lesssucc(s(b)))
                                                        +
                                                  y(b, lesstrans(lesssucc(b), lesssucc(s(b)))))))
```

where $lesstrans(q, q') \in n < p\ [n, m, p \in N,\ q \in n < m,\ q' \in m < p]$.

This example shows the application of covrec and natcases together. The program reflects the traditional structure of this function's definition.

4.2.3 Division

We illustrate the application of course-of-values induction to programming and reasoning over programs, by defining functions associated with division.

Assume $n, m \in N$. Let $order(n, m)$ denote a proof of $n \leq m \vee m < n$.

Quotient

The quotient of dividing n by m, can be defined as:

```
quo(n,m) ≡
        natcases(m, 0
                [u]covrec(n,
                        [a]λy.when(order(s(u), a)
                                [l] s(y(a − s(u), lessproof(l)))
                                [r] 0)))
```

where $lessproof(l) \in a - s(u) < a \quad [a, u \in N,\ l \in s(u) \leq a]$.

The termination of this program follows from the fact that it is an element in N. Now, we can establish the following properties:

Idquo1

$quo(n, 0) \ =_N \ 0 \ \ true \ \ [n \in N]$

Proof. by *Id*-introduction.

Idquo2

$quo(n, s(u)) \ =_N \ s(quo(n - s(u), s(u))) \ \ true \ \ [n, u \in N,\ s(u) \leq n \ true]$

Proof. apply $\vee$-elimination to $order(s(u), n)$:

1. case $[s(u) \leq n]$ follows by *Id*-congruence with s.
2. case $[n < s(u)]$ obtain a proof of $s(u) < s(u)$ from $s(u) \leq n$ and $n < s(u)$ apply $\perp$-elimination to this proof.

Idquo3

$quo(n, s(u)) \ =_N \ 0 \ \ true \ \ [n, u \in N,\ n < s(u) \ true]$

Proof. apply $\vee$-elimination to $order(s(u), n)$:

1. case $[s(u) \leq n]$ $\perp$-elimination to a proof of $s(u) < s(u)$.
2. case $[n < s(u)]$ follows by *Id*-introduction.

Remainder

Similarly, the remainder obtained dividing n by m, can be defined as:

```
rem(n,m) ≡
        natcases(m, n
                [u]covrec(n,
                        [a]λy.when(order(s(u), a)
                                [l] y(a − s(u), lessproof(l))
                                [r] a)))
```

and we can prove:

Idrem1

$rem(n,0) =_N n \ true \ [n \in N]$

Proof. by *Id*-introduction.

Idrem2

$rem(n, s(u)) =_N rem(n - s(u), s(u)) \ true \ [n, u \in N, \ s(u) \leq n \ true]$

Proof. apply $\vee$-elimination to $order(s(u), n)$:

1. case $[s(u) \leq n]$ follows by *Id*-introduction.
2. case $[n < s(u)]$ $\perp$-elimination to a proof of $s(u) < s(u)$.

Idrem3

$rem(n, s(u)) =_N n \ true \ [n, u \in N, \ n < s(u) \ true]$

Proof. apply $\vee$-elimination to $order(s(u), n)$:

1. case $[s(u) \leq n]$ $\perp$-elimination to a proof of $s(u) < s(u)$.
2. case $[n < s(u)]$ follows by *Id*-introduction.

Lessrem

$rem(n, s(u)) < s(u) \ true \ [n, u \in N]$

Proof. By Complete Induction. Assume

$n, u \in N, \ rem(z, s(u)) < s(u) \ true \ [z \in N, \ z < n \ true]$

apply $\vee$-elimination to $order(s(u), n)$:

1. case $[s(u) \leq n]$

   by Idrem2 we have

   $$rem(n, s(u)) =_N rem(n - s(u), s(u)) \ true$$

   from the inductive assumption, $n - s(u) \in N$ and $n - s(u) < n \ true$ we get

   $$rem(n - s(u), s(u)) < s(u) \ true$$

So, applying the corresponding proof of $(n =_N m \wedge m < p) \supset n < p \ \mathit{true}\ [n, m, p \in N]$ we prove

$$rem(n, s(u)) < \ s(u) \ \ \mathit{true}$$

2. case $[n < s(u)]$

   by Idrem3 we have

$$rem(n, s(u)) =_N n \ \ \mathit{true}$$

   Then, apply the corresponding proof of $(n =_N m \wedge m < p) \supset n < p \ \mathit{true}\ [n, m, p \in N]$ to get

$$rem(n, s(u)) < \ s(u) \ \ \mathit{true}$$

Finally, as one would expect, the following proposition are also true:

Proposition.

$$m \ * \ quo(n, m) \ + \ rem(n, m) \ =_N \ n \ \ \mathit{true}\, [n, m \in N]$$

Proof.

By Natural Induction on m.

If $m = 0$, we have to prove

$0 \ * \ quo(n, 0) \ + \ rem(n, 0) \ =_N \ n \ \ \mathit{true} \ \ [n \in N]$

This follows from Idrem1, properties of $*$ and $+$.

If $m = s(u)$, we have to prove

$s(u) \ * \ quo(n, s(u)) \ + \ rem(n, s(u) \ =_N \ n \ \ \mathit{true} \ \ [n, u \in N]$

which we prove by Complete Induction on n.

Assume

$n, u \in N$,
$s(u) \ * \ quo(z, s(u)) \ + \ rem(z, s(u)) \ =_N \ z\, \mathit{true}\, [z \in N, \ z < n\, \mathit{true}]$

Apply $\vee$-elimination to $order(s(u), n)$.

1. case $s(u) \leq n$. Successively applying Id-transitivity we have:

$s(u) * quo(n,s(u)) + rem(n, s(u))$

$=_N$ by Idquo2

$s(u) * s(quo(n - s(u), s(u))) + rem(n, s(u))$

$=_N$ by Idrem2

$s(u) * s(quo(n - s(u), s(u))) + rem(n - s(u), s(u))$

$=_N$ by properties of $*$ and $+$

$s(u) * quo(n - s(u), s(u)) + rem(n - s(u), s(u)) + s(u)$

$=_N$ from the inductive assumption, $n - s(u) \in N$ and $n - s(u) < n$ *tr*

$n - s(u) + s(u)$

$=_N$ by properties of $-, +$ with $s(u) \leq n$

n

2. case $n < s(u)$. Successively applying Id-transitivity we have:

$s(u) * quo(n,s(u)) + rem(n, s(u))$

$=_N$ by Idquo3

$s(u) * 0 + rem(n, s(u))$

$=_N$ by properties of $*$ and $+$

$rem(n, s(u))$

$=_N$ by Idrem3

n

5 Course of Values recursion as a derived rule

A formulation of course-of-values recursion will be expressed into TT. A course-of-values induction rule is proved, obtaining as the corresponding proof term an expression for a course-of-values operator.

5.1 Course-of-Values Recursion rule as a derived rule

Consider the following rule:

$$\frac{\begin{array}{l} p \in N \\ C(v)\ set\ [v \in N] \\ apply(e(x), y) \in C(x)\ [x \in N, y \in \prod z \in N.(z < x \Rightarrow C(z))] \end{array}}{covrec(p, e) \in C(p)}$$

Assume the rule premises. To prove the rule, we must find an element in $C(p)$ where $p \in N$. Considering that the solution for any value can be obtained from the previous solutions by applying the step function, we define:

$$covrec(p, e) \equiv_{def} apply(e(p), Approxfun(e, p))$$

where $Approxfun(e, p) \in \prod z \in N.(z < p \Rightarrow C(z))$.

$Approxfun$ is defined by Natural Induction on p as follows:

$$\begin{array}{lll} Approxfun(e, p) \equiv_{def} natrec(& p, & \\ & G_0, & \\ & [u, v]G_{succ}(e, u, v)) & \text{where} \end{array}$$

$G_0 \in \prod z \in N.(z < 0 \Rightarrow C(z))$ and

$$G_{succ}(e, u, v) \in \prod z \in N.(z < s(u) \Rightarrow C(z)) \quad \left[\begin{array}{l} u \in N \\ v \in \prod z \in N.(z < u \Rightarrow C(z)) \end{array} \right]$$

To define G_0, is to define an *empty* function. This can be done as follows:

assume $z \in N,\ q \in z < 0$

let $notlesszero(z)$ denote an element in $\neg(z < 0)\ [z \in N]$

by $\Rightarrow$-elimination we get

$$apply(notlesszero(z), q) \in \perp \ [z \in N,\ q \in z < 0]$$

now, by $\perp$-elimination

$$\perp\ elim(apply(notlesszero(z), q)) \in C(z)\ [z \in N,\ q \in z < 0]$$

and by $\Rightarrow$-introduction and $\prod$-introduction

$$G_0 \equiv_{def} \lambda z.\lambda q.bottomE(apply(notlesszero(z), q)) \in \prod z \in N.(z < 0 \Rightarrow C(z))$$

$G_{succ}(e, u, v)$, is defined as follows:

assume $z \in N,\ q \in z < s(u)$

let $lesssucctoleq(q)$ denote an element in $z \leq u\ [z, u \in N,\ q \in z < s(u)]$

from this element, we can construct a proof of $z < u$ or a proof of $z =_N u$.

(a) assume we have constructed $ls \in z < u$,

by the inductive assumption and $\prod$-elimination we get

$$apply(v, z) \in z < u \Rightarrow C(z) \left[\begin{array}{l} z, u \in N \\ v \in \prod z \in N.(z < u \Rightarrow C(z)) \end{array} \right]$$

and by $\Rightarrow$-elimination

$$apply(apply(v, z), ls) \in C(z) \left[\begin{array}{l} z, u \in N \\ ls \in z < u \\ v \in \prod z \in N.(z < u \Rightarrow C(z)) \end{array} \right]$$

(b) assume we have constructed $eq \in z =_N u$,

by the third premise is

$$apply(e(u), v) \in C(u))\ [u \in N,\ v \in \prod z \in N.(z < u \Rightarrow C(z))]$$

by using a proof of $(z =_N u\ \&\ C(u)) \supset C(z)$ we can define an element in $C(z)$. It will be denoted as

$$Idsubst(eq, apply(e(u), v)) \in C(z)) \left[\begin{array}{l} z, u \in N \\ eq \in z =_N u \\ v \in \prod z \in N.(z < u \Rightarrow C(z)) \end{array} \right]$$

Now, by $\vee$-elimination is

$$\begin{aligned}&when\,(\;lesssucctoleq(q),\\ &\qquad [\,ls\,]\;apply(apply(v,z),ls),\\ &\qquad [\,eq\,]\;Idsubst(eq, apply(e(u),v))\,)\;\in\; C(z))\\ &\qquad [z,u \in N,\; q \in z < s(u),\; v \in\; \prod z \in N.(z < u \Rightarrow C(z))]\end{aligned}$$

and by $\Rightarrow$-introduction and $\prod$-introduction the following term is obtained:

$$\begin{aligned}G_{succ}(e,u,v) \equiv_{def}\; \lambda\, z\, \lambda\, q.\, when\,(\;&lesssucctoleq(q),\\ &[\,ls\,]\;apply(apply(v,z),ls),\\ &[\,eq\,]\;Idsubst(eq, apply(e(u),v))\,)\end{aligned}$$

$$\in \prod z \in N.(z < s(u) \Rightarrow C(z)) \;\left[\begin{array}{l} u \in N \\ v \in \prod z \in N.(z < u \Rightarrow C(z)) \end{array}\right]$$

Informally, we can explain *Approxfun* definition as follows: from an initial *empty* function G_0, a family of functions is constructed. This functions are *approximations* to the function being defined. The $s(u) - th$ function, $G_{succ}(e,u,v)$, will be defined for all the values less than $s(u)$, and for all the values less than u is exactly the same as the $u - th$ function v. This has some similarity with the way in which is solved a recursive specification in Domain Theory, but is not the same. The main difference is that these approximations are constructed *from* the first function which is defined for the given argument. We are knowing that we want the $p - th$ approximation, and this is what we compute.

5.2 A result about equality

In subsection 4.1, covrec was introduced as a primitive constant and the computation rules for the operator were given. In subsection 5.1 an expression for the operator in terms of primitive recursion was found, giving the possibility of introduce course-of-values recursion as an abbreviation.

We can look at the computation rule for the primitive constant, as *specifying* the expected behavior of the derived expression. In TT, this can be expressed as:

$$covrec(p,e) \;=_{C(p)}\; apply(e(p),\; \lambda\, z.\lambda\, q.covrec(z,e))\;\; true \qquad \text{(I)}$$

where

$p \in N$,
$C(x)\; Set\,[x \in N]$,
$apply(e(x),y) \in C(x)\,[x \in N,\; y \;\in\; \prod z \in N.(z < x \Rightarrow C(z))\,]$

We try to prove this equality as follows :

since

$$covrec(p, e) =_{C(p)} apply(e(p), Approxfun(e, p))\ true$$

is enought to show that

$$Approxfun(e, p) =_{\prod z \in N.(z<p \Rightarrow C(z))} \lambda z.\lambda q.covrec(z, e)\ true$$

applying natural induction, for $p = 0$ we have to show:

$$G_0 =_{\prod z \in N.(z<0 \Rightarrow C(z))} \lambda z.\lambda q.covrec(z, e)\ true$$

at this point, the intentional equality of two function elements must be proved. Since a function is a canonical element (it is just evaluated), there is no way to prove this last equation.

Instead of (I), the Eq-equality of both expressions is proved. This proof has an intensional part, that proves that these functions yields equal values when applied to equal elements in the domain.

Lemma

Let

$p \in N$
$C(x)\ Set\ [x \in N]$,
$apply(e(x), y) \in C(x)\ [x \in N,\ y \in \prod z \in N.(z < x \Rightarrow C(z))]$

then

$$apply(apply(Approxfun(e, p), z), q) =_{C(z)} covrec(z, e)\ true\ \ [z \in N,\ q \in z < p]$$

Proof.

The proof is done by Natural Induction on p. Since $q \in z < p$, depends on p, we will look for a proof of

$$\forall q \in z < p.\ Id(C(z),\ apply(apply(Approxfun(e, p), z), q),\ covrec(z, e))\ true\ \ [z \in N]$$

Applying $\forall$-elimination to this proof and $q \in z < p$, we get the desired result.

Natural Induction on p.

(a) if $p = 0$, we must prove

$\forall q \in z < 0.\, Id(C(z),\ apply(apply(Approxfun(e,0),z),q),\ covrec(z,e))\ true\ [z \in N]$

which follows applying $\forall$-introduction and $\perp$-elimination from $q \in z < 0$ and $\neg(z < 0)$.

(b) if $p = s(u)$, we must prove

$\forall q \in z < s(u).\, Id(C(z),\ apply(apply(Approxfun(e,s(u)),z),q),\ covrec(z,e))\ true$

under the assumptions

$$\left[\begin{array}{l} z, u \in N, \\ \forall ls \in z < u.\, Id(C(z),\ apply(apply(Approxfun(e,u),z),ls),\ covrec(z,e))\ \ true \end{array}\right]$$

Assume $q \in z < s(u)$.
Unfolding $Approxfun$ and applying $\prod$-elimination and $\Rightarrow$-elimination,
we see that it is enought to prove

$$\begin{array}{l} when\,(\ lesssucctoleq(q), \\ \qquad [\,ls\,]\ apply(apply(Approxfun(e,u),z),ls) \\ \qquad [\,eq\,]\ Idsubst(eq, apply(e(u), Approxfun(e,u))) \quad =_{C(z)}\ covrec(z,e)\ \ true \end{array}$$

under the assumptions

$$\left[\begin{array}{l} z, u \in N,\ q \in z < s(u) \\ \forall ls \in z < u.\, Id(C(z),\ apply(apply(Approxfun(e,u),z),ls),\ covrec(z,e))\ \ true \end{array}\right]$$

This equality follows applying $\vee$-elimination to $lesssucctoleq(q)$:

(a) assume we have $ls \in z < u$, then by $\forall$-elimination from the inductive assumption we get

$$apply(apply(Approxfun(e,u),z),ls)\ =_N\ covrec(z,e))\ \ true$$

(b) assume we have $eq \in z =_N u$, applying Id-elimination to this proof, we get

$$Idsubst(eq, apply(e(u),\ Approxfun(e,u))) =_{C(z)} covrec(z,e)\ \ true$$

Now, applying $\forall$-introduction to this proof and the assumption $q \in z < s(u)$ we get the proof of the inductive step.

Theorem

In type theory with Extensional Equality, we can prove:

$$Eq(C(p), covrec(p,e), apply(e(p),\ \lambda\, z.\lambda\, q.covrec(z,e)))\ \ true$$

where

$p \in N$,
$C(x)\ Set\, [x \in N]$,
$apply(e(x), y) \in C(x)\, [x \in N,\ y \in \prod z \in N.(z < x \Rightarrow C(z))\,]$

Proof.

A similar proof for Eq instead of Id in Lemma, gives a term in

$$Eq(C(z), apply(apply(Approxfun(e,p), z), q),\ covrec(z,e))\ \ true\ [z \in N,\ q \in z < p]$$

By strong Eq-elimination we get

$$apply(apply(Approxfun(e,p), z), q) = covrec(z,e) \in C(z)\ \ [z \in N,\ q \in z < p]$$

by $\Rightarrow$-introduction and Eq-introduction it follows

$$Eq(z < p \Rightarrow C(z), \lambda\, q.apply(apply(Approxfun(e,p), z), q), \lambda\, q.covrec(z,e))\ \ true\ \ [z \in N]$$

and by η-equality and transitivity of Eq

$$Eq(z < p \Rightarrow C(z), apply(Approxfun(e,p), z),\ \lambda\, q.covrec(z,e))\ \ true\ \ [z \in N]$$

applying strong Eq-elimination one more time, we get

$$apply(Approxfun(e,p), z) = \lambda\, q.covrec(z,e) \in\ z < p \Rightarrow C(z)\ \ [z \in N]$$

by $\prod$-introduction and Eq-introduction it follows

$$Eq(\prod z \in N.(z < p \Rightarrow C(z)),$$
$$\lambda\, z.(apply(Approxfun(e,p), z), \lambda\, z.\lambda\, q.covrec(z,e))\ \ true$$

and by η-equality and transitivity of Eq

$$Eq(\prod z \in N.(z < p \Rightarrow C(z)), Approxfun(e,p), \lambda\, z.\lambda\, q.covrec(z,e))\ \ true$$

Now, apply Eq-congruence to prove

$$Eq(C(p), apply(e(p), Approxfun(e,p)), apply(e(p), \lambda\, z.\lambda\, q.covrec(z,e)))\ true$$

which by the given definition for *covrec*, is the same that

$$Eq(C(p), covrec(p,e), apply(e(p), \lambda\, z.\lambda\, q.covrec(z,e)))\ true$$

The proof is complete.

This proof can be seen as stating the well-foundness of the relation $<$ on N.

6 Comments on the work

In this section we will describe the steps followed when formalizing the proofs and applications. We will describe the alternative implementations and mention the problems we have to face in each case.

6.1 From the polymorphic to the monomorphic

To formalize the proofs in ALF [ACN90], we use an implementation of Martin-Löf's monomorphic set theory. The major difference is that all constants contains explicit information about which sets its arguments belong to. Beside this, the constants are currified and the selectors takes their main argument in the rightmost place [NPS90]. As an instance, *covrec* is declared as:

$$covrec : (C : (N)Type; e : (x : N)(\prod z \in N.z < p \Rightarrow C(z)) \Rightarrow C(x); p : N)\, C(p)$$

There is a fundamental difference between both theories. There are derivable judgements in the polymorphic theory which can not be proved in the monomorphic theory. In particular, the extensional equality *Eq* does not fit in the monomorphic theory, then the previous **Theorem** will not have a corresponding proof in the formalization.

6.2 About the induction rule

The first rule we tried to formalize is:

$$\frac{\begin{array}{l} p \in N \\ C(v)\ set\ [v \in N] \\ d \in C(0) \\ e(x,y) \in C(s(x))\ [x \in N,\ y(z,q) \in C(z)\ [z \in N, q \in z \leq x] \end{array}}{covrec(p,d,e) \in C(p)}$$

where the step function *e*, is represented by an abstraction and the inductive structure of the natural numbers is reflected in the rule premises. Once defined the corresponding ALF constant, we tried to write a program for the Fibonacci function. When doing the proof, we arrive at the following situation:

Conjeture : $Fib(n : N) : N$

Current term : $[n : N]covrec(n, 1, [x, y]covrec(x, 1, [x', y']y(x, leqreflex(x)) + y(x', ?1)))$

Subgoals : $?1 : x' \leq x$

Local context :

$n : N$
$x : N$
$y : (z : N)(q : z \leq x)N$
$x' : N$
$y' : (z : N)(q : z \leq x')N$

In this proof term, *leqreflex* represents a proof of $x \leq x$. To finish this definition, we need a proof element for $x' \leq x$, but this element can not be defined since x is a constant when looked from **?1**.

From this application, the following aspects of this *covrec* definition becomes clear:

- To use this operator, we must *compute explicitly* the previous arguments, by using operations that allow the proof of the order property. As an instance, Fibonacci function can be defined applying *predecessor* as:

  $$[n : N]covrec(n,1, \\ [x, y]covrec(x, 1, [x', y']y(x, leqreflex(x)) + y(pred(x), leqpred(x))))$$

  where $leqpred(x)$ is a proof of $pred(x) \leq x$.
  This limitation, is a consequence of the representation of the step function by an abstraction. Since we prefer to define an operator that allows the use of case analysis over its major argument, we decide to represent the step function as a function element.

- We have used *covrec* two times in this expression. The second time, without any particular reason (we could have used *natrec* in the same way). Really, we have no interest in recursion in this place, and what we want to do is to *analize* x value.
  If we remember quotient and remainder examples, we have analized the proposition $(n \leq m) \vee (m < n)$ in order to define them, and the definition was made, knowing before hand that m value is of the form $s(u)$ and independently of n value. Then, we think that there is really no reason to consider the inductive structure of natural numbers as part of *covrec* definition.

6.3 About the equality

When introducing a primitive constant, its definition can be reflected at the type level using Id sets. As a consequence, useful properties of programs can be proved. This also, was showed in the previously presented examples.

If we introduce an operator as a definition, we had to face the problem of proving this equality properties. It is not enought to give the definition, and something more must be constructed that allows the proof of equality properties.

The intentional part of the equality presented above, **(Lemma)** seems to be useful for this.

As an example, consider the quotient program. When proving its equality properties, *covrec* definition was implicitly used in the introduction of the *Id* elements. These properties can be proved using the derived *covrec* as follows:

The program for quotient was defined as

```
quo(n, m) ≡_def
    natcases(m, 0
             [u]covrec(n,
                       [a]λ y.when(order(s(u), a),
                                   [l] s(y(a − s(u), lessproof(l))),
                                   [r] 0)))
```

Properties

Idquo1

$quo(n, 0) \ =_N \ 0 \ \ true \ \ [n \in N]$

the proof remains the same.

Idquo2

$quo(n, s(u)) \ =_N \ s(quo(n - s(u), s(u))) \ \ true \ \ [n, u \in N, \ s(u) \leq n \ true]$

by unfolding *covrec* definition, $quo(n, s(u))$ is the same that

```
when( order(s(u), n),
    [l] s( apply(apply(Approxfun([a]λ y.when(order(s(u), a)
                                              [l₁] s(y(a − s(u), lessproof(l₁)))
                                              [r₁] 0, ))
                                                                         , n )
                     n − s(u)),
               lessproof(l)))
    [r] 0)
```

the equality between this program and $s(quo(n-s(u),s(u)))$ is proved by $\vee$-elimination to $order(s(u),n)$:

1. case $[s(u) \leq n]$ by Lemma we have:

$$\begin{array}{lll} apply(apply(Approxfun([a]\lambda\, y.when(order(s(u),a) & & \\ & [\,l_1\,]\; s(y(a-s(u),lessproof(l_1))) & \\ & [\,r_1\,]\; 0,)), & ,n) \\ \quad n-s(u)), & & \\ \quad lessproof(l)) & & \\ & =_N quo(n-s(u),s(u)) \quad true & \end{array}$$

   then, its enought to apply Id-congruence with s.

2. case $[n < s(u)]$ follows from $\perp$-elimination to a proof of $s(u) < s(u)$.

Idquo3

$quo(n,s(u)) =_N 0 \;\; true \;\; [n,u \in N,\; n < s(u)\, true]$

the proof remains the same.

A detailed version of these proofs are in the appendix.

Acknowledgements

I want to thanks my supervisor J.J.Cabezas by his valuable suggestions along the development of this work.
I want to thank specially Bengt Nordström by suggesting me to work on this subject on his stay in Montevideo and by his comments to this work.
I would also like to thanks Alvaro Tasistro by his suggestions and comments while doing this work, and to Nora Szasz by her suggestions about the processing of the appendix.
I would also want to thanks Bror Bjerner by his comments. Thanks to Alberto Pardo by answering all my questions concerning LaTeX and also to Fernando Carpani for solving all my problems concerning ALF installation.

References

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In G. Huet and G. Plotkin, editors, *Proceedings of First Workshop on Logical Frameworks*, pages 39–42. Esprit Basic Research Action 3245, 1990.

[Hen77] F. Hennie. *Introduction to Computability.* Addison-Wesley Co., 1977.

[Her65] H. Hermes. *Enumerability, Decidability, Computability.* Springer-Verlag, Berlin, 1965.

[Löf84] M. Löf. Intuitionistic Type Theory. Bibliopolis, 1984.

[Nor88] B. Nordström. Terminating General Recursion. *Bit*, 28:605–619, 1988.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory, An Introduction.* Oxford University Press, 1990.

[Pau86] L. C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[Smi83] Jan M. Smith. The identification of propositions and types in Martin-Löf's Type Theory: A programming example. *Foundations of Computation Theory, Lecture Notes in Computer Science*, 158, 1983.

[Sza91] Nora Szasz. A Machine Chequed Proof that Ackermann's Function is not Primitive Recursive, 1991. Department of Computer Science. Chalmers University of Technology and University of Göteborg.

APPENDIX

ALF is a system for editing proofs and theories, based on a combination of a general type system and Martin-Lõf's logical framework. When representing Martin-Lõf's monomorphic Set Theory in ALF, we associate to each set an ALF type and to elements in sets objects in types. In the monomorphic ST, propositions are identified with sets and proofs with elements in the corresponding set, so there are also represented by ALF types and objects.
Definitions are grouped in theories. To define a theory, you define a list of typings and definitions of constants. There are primitive constants that corresponds to constructors (it has only a type), and defined constants (it has a type and a definition). A defined constant can be implicitely defined (defined recursively) or explicitely defined (abreviattion). When implementing Martin-Lõf's monomorphic Set Theory in ALF, we define a theory for each set forming operation and basic set of the theory. ALF constants can be seen as representing rules of the theory. In this case formation and introduction rules are identified with primitive constants, elimination and equality rules with implicitely defined constants, and derived rules with defined constants.
Some constants are defined with hidden parameters, which is indicated with the symbol ↓ in front of them. In this case, the parameter is not visible but the information is there. In this form, we can get a polymorphic "vision" of the system by hidding type information. In some cases, we hide other parameters that are redundant and can be inferred from the context.

Pi Sets

(∀)(A: Type; B:(A)Type): Type

Λ(↓A: Type; ↓B:(A)Type; b:(x:A)B(x)):(∀x∈A.B(x))

```
∀elim (↓A: Type; ↓B:(A)Type; ↓C:((∀x∈A.B(x)))Type ; e:(y:(x:A)B(x))C(Λ(y)) ; f:(∀x∈A.B(x)) ): C(f) ≡
        case f of (Λ(A, B, b) => e(b))
```

∀apply (↓A: Type; ↓B:(A)Type; f:(∀x∈A.B(x)); a:A): B(a) ≡ ∀elim([z]z(a), f)

Functions

(⇒)(A,B: Type): Type ≡ (∀h∈A.B)

λ(↓A,B: Type; b: (A)B): A⇒B ≡ Λ(b)

apply(↓A,B: Type; f: A⇒B; a: A): B ≡ ∀elim([y]y(a), f)

Disjoint Union

(∨)(Type;Type):Type

inl(↓A,B:Type;A):A∨B;

inr(↓A,B:Type;B):A∨B;

```
when(↓A,B:Type; ↓C:(A∨B)Type; e:(x:A)C(inl(x)); f:(y:B)C(inr(y));p:A∨B):C(p) ≡
        case p of (inl(A,B,a) => e(a) |
                   inr(A,B,b) => f(b))
```

Exists Proposition

(∃)(A: Type; B:(A)Type): Type

∃intro(↓A: Type; ↓B:(A)Type; a: A; b: B(a)): (∃x∈A.B(x))

∃elim(↓A: Type; ↓B:(A)Type; ↓C: ((∃x∈A.B(x)))Type; d: (a:A)(b:B(a))C(∃intro(a, b))); p: (∃x∈A.B(x))): C
    case p of (∃intro(A, B, a, b) => d(a, b))

fst(↓A: Type; ↓B:(A)Type; p: (∃x∈A.B(x))): A ≡ ∃elim([a,b]a, p)

snd(↓A: Type; ↓B:(A)Type; p: (∃x∈A.B(x))): B(fst(p)) ≡ ∃elim([a,b]b, p)

Bottom

⊥:Type

⊥elim(↓C:(⊥)Type; b:⊥):C(b)

¬(A:Type):Type ≡ A⇒⊥

Id sets

Id(↓A:Type; a,b:A): Type

id(↓A:Type; x:A): Id(x,x)

IdE(↓A:Type; ↓C:(x:A)(y:A)(e:Id(x y))Type; a,b:A; e:Id(a,b); d:(x:A)C(x,x,id(x))): C(a,b,e) ≡
    case e of (id(A,a) => d(a))

Properties of Id

Idrefl(↓A:Type; a:A): Id(a,a) ≡ id(a)

Idsymm(↓A:Type; ↓a,b:A; i:Id(a,b)): Id(b,a) ≡ IdE(a,b,i,[x]id(x))

Idtrans(↓A:Type; ↓a:A; b:A; ↓c:A; p:Id(a,b); q:Id(b,c)):Id(a,c) ≡ apply(IdE(a,b,p,[y]λ([z]z)),q)

Idsubst(↓A:Type; P:(A)Type; a:A; ↓b:A; i:Id(a, b); p: P(a)): P(b) ≡ apply(IdE(a, b, i,[x]λ([w]w)), p)

Idcongr(↓A,B:Type; f:(A)B; ↓a,b:A; i:Id(a,b)): Id(f(a),f(b)) ≡ Idsubst([h]Id(f(a),f(h)),a,i,id(f(a)))

Natural Numbers

```
N: Type

0: N

s(n:N): N

natcases(↓C:(N)Type; d:C(0); e:(u:N)C(s(u)); p:N): C(p) ≡ case p of (0 => d | s(u) => e(u))

natrec(↓C: (N)Type; d: C(0); e: (u:N)(v:C(u))C(s(u)); n: N): C(n)≡
        case n of (0 => d | s(u) => e(u, natrec(C, d, e, u)))
```

Operations on N

```
pred(n: N): N ≡ natrec(0, [u,v]u, n)

(+)(x,y: N): N ≡ natrec(y, [u,v]s(v), x)

(−)(x,y: N): N ≡ natrec(x, [u,v]pred(v), y)

(*)(x,y: N): N ≡ natrec(0, [u,v](v+y), x)

1: N ≡ s(0)

2: N ≡ s(1)

3: N ≡ s(2)

4: N ≡ s(3)
```

Double Recursion

This constant defines a double recursion schema into the theory. The parameters where defined to represent double induction, as it is done usually in Arithmetics. The definition needs the use of higher order functions in order to be able to make recursion over one argument when using the other as parameter.

```
doubrec(↓C:(N)(N)Type; d:(x:N)C(x, 0); e:(y:N)C(0, s(y)); f:(x:N)(y:N)(C(x, y))C(s(x),s(y)); a,b:N): C(a,b) ≡
        ∀apply(natrec(Λ([k]d(k)),
                      [u,v]Λ([k]natrec(e(u),[w,z]f(w, u, ∀apply(v, w)), k)),
                      b), a)
```

Id on N

```
IdN:(x,y:N)Type ≡ Id(N)

Idsucc(↓x,y:N; i:IdN(x, y)): IdN(s(x), s(y)) ≡ Idcongr(s,i)
```

The next terms looks equal, but there are different. The differences are not visibles since corresponds to hidden parameters.

Idfromsucc(↓x,y:N; i:IdN(s(x),s(y))): IdN(x, y) ≡ Idcongr(pred,i)

Idpred(↓x,y:N; i:IdN(x, y)): IdN(pred(x),pred(y)) ≡ Idcongr(pred,i)

Peano4

Natrec(d: Type; e: (u:N)(T:Type)Type; n: N): Type ≡ case n of (0 => d | s(u) => e(u, Natrec(d, e, u)))

Iszero(n: N): Type ≡ Natrec(N, [u,T]⊥, n)

peano4(n: N): ¬(IdN(0, s(n))) ≡ λ([h]Idsubst(Iszero,0,h,0))

The following properties state equalities between objects in N constructed by the use of the operations +, − and ∗. These equalities correspond to axioms and theorems of formal number theories and primitive recursive arithmetics and are basics for the rest of the work. They are needed for proof of properties of the order on N, and when defining the applications.

Properties of +

addzero(n:N): IdN(n+0, n) ≡ natrec(id(0),[u,v]Idsucc(v), n)

Saddzero(n:N): IdN(n, n+0) ≡ Idsymm(addzero(n))

addsucc(n,m: N): IdN(s(n+m), n+s(m)) ≡ natrec(id(s(0+m)), [u,v]Idsucc(v), n)

Saddsucc(n,m: N): IdN(n+s(m), s(n+m)) ≡ Idsymm(addsucc(n,m))

addsubstL(↓a,b,y:N; p:IdN(a,b)): IdN(a+y, b+y) ≡ Idcongr([h](h+y),p)

addsubstR(↓a,b,x:N; p:IdN(a,b)): IdN(x+a, x+b) ≡ Idcongr([h](x+h),p)

addsubst(↓x,y,w,z:N; p:IdN(x,y); q:IdN(z,w)): IdN(x+z, y+w) ≡ Idtrans(y+z,addsubstL(p), addsubstR(q))

addassocR(n,m,k: N): IdN(n+m+k, n+(m+k)) ≡ natrec(id(m+k), [u,v]Idsucc(v), n)

addassocL(n,m,k: N): IdN(n+(m+k), n+m+k) ≡ Idsymm(addassocR(n, m, k))

addcommut(n,m: N): IdN(m+n, n+m) ≡ natrec(Saddzero(n), [u,v]Idtrans(s(n+u),Idsucc(v), addsucc(n, u)), m)

Properties of −

zerominus(n:N): IdN(0−n, 0) ≡ natrec(id(0),[u,v]Idpred(v)),n)

predminus(n,m:N): IdN(pred(n)−m, pred(n−m)) ≡
natrec(Idpred(id(n)), [u,v]Idtrans(pred(pred(n−u)), Idpred(v), Idpred(id(n−s(u)))),m)

Spredminus(n,m:N): IdN(pred(n−m), pred(n)−m) ≡ Idsymm(predminus(n,m)) minusdiff(x,y: N): IdN(s(x)−s(y), x−y) ≡ Spredminus(s(x),y)

xminusx(x: N): IdN(x−x,0) ≡ natrec(id(0), [u,v]Idtrans(u−u, minusdiff(u,u), v), x)

minussubstL(↓a,b,y:N; p:IdN(a, b)): IdN(a−y, b−y) ≡ Idcongr([h](h−y),p)

minussubstR(↓a,b,x: N; p:IdN(a, b)): IdN(x−a, x−b) ≡ Idcongr([h](x−h),p)

minussubst(↓x,y,w,z:N; p:IdN(x, y); q: IdN(z, w)): IdN(x−z, y−w) ≡ Idtrans(y−z,minussubstL(p), minussubstR(q))

Properties of *

timeszero(n:N): IdN(n*0, 0) ≡ natrec(id(0), [u,v]Idtrans(u*0,addzero(u*0), v), n)

Stimeszero(n:N): IdN(0, n*0) ≡ Idsymm(timeszero(n))

```
timessucc(n,m: N): IdN(n*s(m), n*m+n) ≡
          natrec(id(0),
                 [u,v]Idtrans(u*m+(u+s(m)),
                              Idtrans(u*m+u+s(m), addsubstL(v), addassocR(u*m, u, s(m))),
                              Idtrans(u*m+(m+s(u)),
                                      addsubstR(Idtrans(s(u)+m, Idsymm(addsucc(u, m)), addcommut(m, s(u)))),
                                      addassocL(u*m, m, s(u)))),
                 n)
```

Stimessucc(n,m: N): IdN(n*m+n, n*s(m)) ≡ Idsymm(timessucc(n, m))

timessubstL(↓a,b,y: N; p: IdN(a, b)): IdN(a*y, b*y) ≡ Idcongr([h]h*y,b, p)

timessubstR(↓a,b,x: N; p: IdN(a, b)): IdN(x*a, x*b) ≡ Idcongr([h]x*h,b, p)

timessubst(↓x,y,w,z: N; p: IdN(x, y); q: IdN(z, w)): IdN(x*z, y*w) ≡ Idtrans(y*z, timessubstL(p), timessubstR(q))

timescommut(n,m:N): IdN(m*n, n*m) ≡ natrec(Stimeszero(n), [u,v]Idtrans(n*u+n, addsubstL(v), Stimessucc(n, u)), m)

Less

In order to formalize the course-of-values recursion rule, we have to define the order for natural numbers. To do this, we define an inductive family of sets following [Sza91].

(<)(N;N): Type

zeroless(x: N): 0<s(x)

success(↓x,y:N; p:x<y): s(x)<s(y)

```
lessE(↓C:(x:N)(y:N)(x<y)Type; zz:(x:N)C(0,s(x),zeroless(x));
      ss:(x:N)(y:N)(p:x<y)(u:C(x, y, p))C(s(x),s(y),success(p));
      ↓n,m:N; p:n<m): C(n, m, p) = case p of (zeroless(x) => zz(x) |
                                              success(x, y, q) => ss(x, y, q, lessE(C, zz, ss, x, y, q)))
```

Properties of Less

```
lesssucc(x: N):x<s(x) ≡ natrec(zeroless(0),[u,v]success(v), x)

lessIdtoless(↓x:N; y:N; ↓z:N; lxy:x<y; iyz:IdN(y,z)): x<z ≡ Idsubst ([z1](x<z1), y, iyz, lxy)

Idlesstoless(↓x:N; y:N; ↓z:N; ixy:IdN(x,y); lyz:y<z): x<z ≡ Idsubst([x1](x1<z), y, Idsymm(ixy), lyz)
```

The following proofs shows the equivalence between n<m and (∃k∈N.IdN(s(k)+n, m)). This equivalence is used in the transitivity proof for <.

```
lesstoexists(↓n,m: N; p:n<m): (∃k∈N.IdN(s(k)+n, m)) ≡
            lessE([x]∃intro(x, addzero(s(x))),
                  [x,y,q,u]∃intro(fst(u), Idtrans(s(s(fst(u))+x),Saddsucc(s(fst(u)), x), Idsucc(snd(u)))),
                  p))

existstolessL0:(∀m∈N.(∃k∈N.IdN(s(k)+0, m))⇒0<m) ≡
            Λ([m]λ([k]lessIdtoless(s(fst(k))+0,
                             lessIdtoless(s(fst(k)), zeroless(fst(k)), Saddzero(s(fst(k)))),
                             snd(k)))))

existstolessLsucc(u:N; v:(∀m∈N.(∃k∈N.IdN(s(k)+u, m))⇒u<m): (∀m∈N.(∃k∈N.IdN(s(k)+s(u), m))⇒s(u)<m)≡
            Λ([m]λ([k]lessIdtoless(s(s(fst(k))+u),
                             success( apply( ∀apply(v, s(fst(k))+u), ∃intro(fst(k), id(s(fst(k))+u)))),
                             Idtrans(s(fst(k))+s(u), addsucc(s(fst(k)), u), snd(k)))))

existstoless(↓n,m:N; p:(∃k∈N.IdN(s(k)+n, m))):n<m ≡
            apply(∀apply(natrec(existstolessL0, existstolessLsucc, n), m), p)
```

Using the previous equivalences, the following constants can be defined. From this constants, transitivity and asymmetry of < is proved.

```
lessbydiff(↓x,y:N: k: N; d: IdN(s(k)+x, y)): x<y ≡ existstoless(∃intro(k, d))

lessdiff(↓x,y: N; lxy: x<y): N ≡ fst(lesstoexists(lxy))

lessdiffproof(↓x,y: N; lxy: x<y): IdN(s(lessdiff(lxy))+x, y) ≡ snd(lesstoexists(lxy))

lesstrans(↓x:N; y:N; ↓z:N; lxy: x<y; lyz: y<z): x<z ≡
        lessbydiff(lessdiff(lyz)+s(lessdiff(lxy),
                Idtrans(s(lessdiff(lyz))+s(lessdiff(lxy)+x),
                        addassocR(s(lessdiff(lyz)), s(lessdiff(lxy)), x),
                        Idtrans(s(lessdiff(lyz))+y, addsubstR(lessdiffproof(lxy)), lessdiffproof(lyz))))

lesstranssucc(↓x:N; y:N; ↓z:N; lxy: x<y; lyz: y<z): s(x)<z ≡
lessbydiff(lessdiff(lyz)+lessdiff(lxy),
        Idtrans(s(lessdiff(lyz))+(lessdiff(lxy)+s(x)),
                addassocR(s(lessdiff(lyz)), lessdiff(lxy), s(x)),
                Idtrans(s(lessdiff(lyz))+y,
                        addsubstR(Idtrans(s(lessdiff(lxy))+x, Saddsucc(lessdiff(lxy),x), lessdiffproof(lxy))),
                        lessdiffproof(lyz))))
```

```
lessfromsucc(↓x,y:N; q:s(x)<s(y)): x<y ≡
               lessbydiff(lessdiff(q),
                          Idfromsucc(Idtrans(s(lessdiff(q)+s(x),addsucc(s(lessdiff(q)), x),lessdiffproof(q))))

asymmadd(x,k: N; h: IdN(s(k)+x, x)): ⊥≡
          apply(natrec(λ([h₁]apply(peano4(k), Idtrans(s(k)+zero, Idsymm( h₁),addzero(s(k))))),
                       [u,v]λ([h₁]apply(v, Idfromsucc(Idtrans(s(k)+s(u), addsucc(s(k), u), h₁)))),
                       x),
                h)

lessasymm(x:N): ¬(x<x) ≡ λ([h]asymmadd(x,lessdiff(h), lessdiffproof(h)))
```

The following proof states that N is well-founded by <. It is used as "basis" in covrec definition and in the proof of equality.

```
notlesszero(x:N):¬(x<0) ≡
          λ([h]apply(peano4(lessdiff(h)+x),
                     Idtrans(s(lessdiff(h))+x,
                             Idsymm(lessdiffproof(h)),
                             Idtrans(lessdiff(h)+s(x), addsucc(lessdiff(h), x), Saddsucc(lessdiff(h), x)))))
```

Less or Equal

The ≤ relation is defined in terms of < and IdN as:

(≤)(x,y: N): Type ≡ (x<y)∨IdN(x, y)

Properties of Less or Equal

lesstoleq(↓x,y:N; lxy:x<y): x≤y ≡ inl(lxy)

Idtoleq(↓x,y:N; ixy:IdN(x, y)): x≤y ≡ inr(ixy)

zeroleq(x: N): 0≤x ≡ natrec(Idtoleq(id(0)), [u,v]lesstoleq(zeroless(u)), x)

succleq(↓x,y:N; qxy:x≤y): s(x)≤s(y) ≡ when([lxy]lesstoleq(succless(lxy)), [ixy]Idtoleq(Idsucc(ixy)), qxy)

leqreflex(x:N): x≤x ≡ Idtoleq(id(x))

leqsucc(x:N): x≤s(x) ≡ lesstoleq(lesssucc(x))

leqpred(x:N): pred(x)≤x ≡ natrec(leqreflex(0), [u,v]leqsucc(u), x)

The following properties expresses transitivities holding between <, ≤ and IdN.

lessleqtoless(↓x:N; y:N; ↓z:N; lxy:x<y; qyz: y≤z): x<z ≡ when([lyz]lesstrans(y,lxy,lyz), [iyz]lessIdtoless(y,lxy,iyz), qyz)

```
leqlesstoless(↓x:N; y:N; ↓z:N; qxy:x≤y; lyz:y<z): x<z ≡ when([lxy]lesstrans(y,lxy,lyz), [ixy]Idlesstoless(y,ixy,lyz), qxy)

leqIdtoleq(↓x:N; y:N; ↓z:N; qxy:x≤y; iyz:IdN(y, z)): x≤z ≡ Idsubst([z₁](x≤z₁), y, iyz, qxy)

Idleqtoleq(↓x:N; y:N; ↓z:N; ixy:IdN(x, y); qyz:y≤z): x≤z ≡ Idsubst([x₁](x₁≤z), y, Idsymm(ixy), qyz)

leqtrans(↓x:N; y:N; ↓z:N; qxy:x≤y; qyz:y≤z): x≤z ≡
          when([lxy]lesstoleq(lessleqtoless (y, lxy, qyz)),[ixy]Idleqtoleq(y, ixy, qyz), qxy)

lessleqtoleq(↓x:N; y:N; ↓z:N; lxy:x<y; qyz:y≤z): x≤z ≡ leqtrans(y,lesstoleq(lxy), qyz)

leqlesstoleq(↓x:N; y:N; ↓z:N; qxy:x≤y; lyz:y<z): x≤z ≡ leqtrans(y,qxy,lesstoleq(lyz))

leqtoexists(↓x,y:N; qxy:x≤y): (∃k∈N.IdN(k+x, y)) ≡
            when([lxy]∃intro(s(lessdiff(lxy)), lessdiffproof(lxy))), [ixy]∃intro(0, ixy), qxy)

existstoleq(↓x,y:N; p:(∃k∈N.IdN(k+x, y))): x≤y ≡
            apply( apply(natrec(λ([h]λ([ixy]Idtoleq(ixy))), [u,v]λ([h]λ([lxy]lesstoleq(lessbydiff(u,lxy)))), fst(p)),
                          id(fst(p))),
            snd(p))

leqbydiff(↓x,y:N; k:N; d:IdN(k+x, y)): x≤y ≡ existstoleq(∃intro(k, d))

leqdiff(↓x,y:N; qxy:x≤y):N ≡ fst(leqtoexists(qxy))

leqdiffproof(↓x,y:N; qxy:x≤y): IdN(leqdiff(qxy)+x, y) ≡ snd(leqtoexists(qxy))

leqfromsucc(↓x,y:N; q: s(x)≤s(y)): x≤y ≡ when([lxy]lesstoleq(lessfromsucc(lxy)), [ixy]Idtoleq(Idfromsucc(ixy)), q)

lesstoleqsucc(↓x,y:N; lxy:x<y): s(x)≤y ≡ leqbydiff(lessdiff(lxy), Idtrans(s(lessdiff(lxy))+x, Saddsucc(lessdiff(lxy), x),
                                                                                    lessdiffproof(lxy)))

leqsucctoless(↓x,y:N; qxy:s(x)≤y):x<y ≡ lessfromsucc(leqlesstoless(y,qxy,lesssucc(y)))

leqtolesssuc(↓x,y:N; qxy:x≤y): x<s(y) ≡ leqlesstoless(y, qxy, lesssucc(y))

lesssucctoleq(↓x,y:N; l:x<s(y)): x≤y ≡ leqfromsucc(lesstoleqsucc(l))

leqzerotoId(↓x:N; q:x≤0): IdN(x, 0) ≡ when([l]⊥elim(apply(notlesszero(x), l)), [i]i, q)
```

The next are properties of < and ≤ with respect to −.

```
lessminus(x,y: N): s(x)−s(y)<s(x) ≡
          Idlesstoless(x−y, minusdiff(x, y), natrec(lesssucc(x−0), [u,v]leqlesstoless(x−u, leqpred(x−u), v), y)

lessminusfun(a,b:N): s(b)≤a⇒a−s(b)<a ≡ natrec(λ([q]when([l]bottomE(apply(notlesszero(s(b)), l)),
                                                          [i]bottomE(apply(peano4(b), Idsymm(i))), q)),
                                            [u,v]λ([q]lessminus(u, b)),
                                            a)

lessproof(a,b:N; q:s(b)≤a): a−s(b)<a ≡ apply(lessminusfun(a, b), q)
```

The following proof is done by using the double recursion schema.

```
Idminusadd(a,b: N): a≤b⇒IdN((b−a)+a, b) ≡
            doubrec([x]λ([q]addsubst(zerominus(x),leqzerotoId(q))),
                    [y]λ([q]addzero(s(y)−0)),
                    [x,y,f]λ([q]Idtrans(y−x+s(x),addsubstL(minusdiff(y, x)),
                                                Idtrans( s(y−x+x),Saddsucc(y−x, x),
                                                                Idsucc( apply( f, leqfromsucc(q)))))),
                    a, b)
```

The next constant is very useful when defining applications. This allows to "decide" the order relation (we are looking for a proof of (x≤y)∨¬ (x≤y)).

```
order(x,y: N): x≤y∨y<x ≡
      natrec(inl(zeroleq(y)),
            [u₁,v₁]when([l₁]when([luy]inl(lesstoleqsucc(luy)),
                                  [iuy]inr(Idlesstoless(u₁,Idsymm(iuy),lesssucc(u₁))),
                                  l₁),
                        [r₁]inr(lesstrans(u₁, r₁, lesssucc(u₁))),
                        v₁),
            x)
```

Another definition for order, can be done by using the scheme of double recursion defined earlier. This example shows a nice application of this operator to programming.

```
order(x,y: N): x≤y∨y<x ≡
      doubrec([x₁]natcases(inl(zeroleq(0)),[u]inr(zeroless(u)), x₁),
              [y₁]inl(lesstoleq(zeroless(y₁))),
              [x₁,y₁,f]when([lxy]inl(succleq(lxy)),
                            [lyx]inr(succless(lyx)),
                            f),
              x,
              y)
```

covrec definition

Now, we present the definition for the course-of-values operator. The proof follows the same steps presented earlier in the paper. We introduce also an abbreviation for the type of the functions.

```
approxtype(↓C:(N)Type; p:N): Type ≡ (∀z∈N.z<p⇒C(z))

gzero(↓C:(N)Type): approxtype(0) ≡ Λ([z]λ([q]⊥elim(apply(notlesszero(z), q))))

gsucc(↓C:(N)Type; e:(x:N)(approxtype(x)⇒C(x)) ; u: N; v: approxtype(u)): approxtype(s(u)) ≡
      Λ([z]λ([q] when([ls]apply(∀apply(v, z), ls),
                      [eq]Idsubst(C, u, Idsymm(eq), apply(e(u), v)),
                      lesssucctoleq(q))))
```

```
approxfun(↓C:(N)Type; e:(x:N)(approxtype(x)⇒C(x)) ; p: N): approxtype(p) ≡
              natrec(gzero, [u,v]gsucc(e, u, v), p)

covrec(↓C:(N)Type; e:(x:N)(approxtype(x)⇒C(x)); p:N):C(p) ≡ apply(e(p), approxfun(e, p))
```

Some applications

We have defined natcases as an implicitly defined constant. Now, natrec is defined as an abbreviation by using natcases and covrec. In the paper we had justified these definition by refering to the computational behavior, here on the other side, we present a derivation of the operator into the system.

natrec in terms of covrec

```
natrec'(↓C: (N)Type; d:C(0); e:(u:N)(v:C(u))C(s(u)); p:N): C(p) ≡
          covrec([x]natcases(λ([y]d)) ,
                             [u]λ([y]e(u, apply(∀apply(y, u), lesssucc(u)))),
                             x),
              p)
```

Fibonacci function

```
Fib(n:N):N ≡ covrec([x]natcases(λ([y]1),
                            [a]natcases(λ([y]1),
                                       [b]λ([y](apply(∀apply(y, s(b)),lesssucc(s(b))))
                                                                          +
                                               (apply(∀apply(y, b),lesstrans(s(b),lesssucc(b),lesssucc(s(b))))))
                                       ,a)
                            ,x)
              ,n)
```

A result about equality

What follows is the proof of the Lemma already presented. The proof follows the same steps as before. Since some terms are too long, we prove some auxiliary lemmas and combine them to obtain the proof.

Eqzero(↓C: (N)Type; e:(x:N)(approxtype(x)⇒C(x)); z: N; q: z<0):
Id(apply(∀apply(approxfun(e,0), z),q), covrec(e,z)) ≡ ⊥elim(apply(notlesszero(z), q))

Eq_{ls}(↓C: (N)Type;e:(x:N)(approxtype(x)⇒C(x)); u, z: N;
v:(∀ls∈z<u.Id(apply(∀apply(approxfun(e,u), z), ls), covrec(e,z))) ; q: z<u):
Id(apply(∀apply(approxfun(e,u), z), q), covrec(e,z)) ≡ ∀apply(v, q)

Eq_{eq}(↓C: (N)Type; e:(x:N)(approxtype(x)⇒C(x)); u, z: N; eq: IdN(z, u)):
Id(Idsubst(C, u, Idsymm(eq), apply(e(u), approxfun(e, u))), covrec(e,z)) ≡
IdE(z, u, eq, [x]id(Idsubst(C, x, Idsymm(id(x)), apply(e(x), approxfun (e, x)))))

```
Eqsucc(↓C: (N)Type;e: (x:N)(approxtype(x)⇒C(x)); u, z: N;
        v:(∀ls∈z<u.Id(apply(∀apply(approxfun(e,u), z), ls), covrec(e,z))) ; q: z<s(u)):
                Id(apply(piapply(approxfun(e, s(u)), z), q), covrec(e,z)) ≡
        when([ls]Eq_ls(e, u, z, v, ls),
              [eq]Eq_eq(e, u, z, eq)),
              lesssucctoleq(q))

funEq(↓C:(N)Type; e: (x:N)(approxtype(x)⇒C(x)); p, z: N):
        (∀q∈z<p.Id(apply(∀apply(approxfun(e,p), z), q), covrec(e,z))) ≡
        natrec(Λ([q]Eqzero(e, z, q)), [u,v]Λ([q]Eqsucc(e, u, z, v, q)), p)

Equality(↓C:(N)Type; e: (x:N)(approxtype(x)⇒C(x)); p,z: N; q:z<p):
        Id(apply(∀apply(approxfun(e,p), z),q), covrec(e,z)) ≡ ∀apply(funEq(e, p, z), q)
```

Here we formalize the presented examples concerning to Division. The same programs are defined and properties proved. In the proof of equality properties of the programs, the result about equality is used.

Quotient definition

```
quo(n,m:N): N ≡ natcases(0,
                    [u]covrec([a]λ([y]when([l]s(apply(∀apply(y, a−s(u)),lessproof(a, u, l))),
                                            [r]0,
                                            order(s(u), a))),
                              n),
                    m)
```

Properties of Quotient definition

This constant represents the corresponding application of the Lemma. This term is used below in the proof of the properties of quo definition.

```
Eqquo(n,u:N; l: s(u)≤n):
      IdN(apply(∀apply(approxfun([a]λ([y]when([l1]s(apply(∀apply(y, a−s(u)), lessproof(a, u, l1))),
                                              [r1]0, order(s(u), a))), n)),
                n−s(u)),
            lessproof(n, u, l)),
        quo(n−s(u),s(u))) ≡
      Equality([a]λ([y]when([l1]s(apply(∀apply(y, a−s(u)), lessproof(a, u, l1))),
                            [r1]0,
                            order(s(u), a))),
           n,
           n−s(u),
           lessproof(n, u, l))

Idquo1(n: N): IdN(quo(n,0), 0) ≡ id(0)

Idquo2(n,u: N; q: s(u)≤n): IdN(quo(n, s(u)), s(quo(n−s(u), s(u)))) ≡
      when([l]Idcongr(s,Eqquo(n,u,l))
           [r]bottomE(apply(lessasymm(s(u)),leqlesstoless(n, q, r)))
           order(s(u), n))
```

```
Idquo3(n,u: N; q: n<s(u)): IdN(quo(n, s(u)), 0) ≡ when([l]bottomE(apply(lessasymm(s(u)),leqlesstoless(n, l, q))),
                                                   [r]id(0),
                                                   order(s(u), n))
```

Remainder definition

```
rem(n,m:N): N ≡ natcases(n,
                    [u]covrec([a]λ([y]when([l]apply(∀apply(y, a−s(u)),lessproof(a, u, l)),
                                           [r]a,
                                           order(s(u), a))),
                              n),
                    m)
```

Properties of Remainder definition

This constant represents the corresponding application of the Lemma. This term is used below in the proof of the properties of rem definition.

```
Eqrem(n,u:N; l: s(u)≤n):
     IdN(apply(∀apply(approxfun([a]λ([y]when([l_1]apply(∀apply(y, a−s(u)), lessproof(a, u, l_1)),
                                              [r_1]a, order(s(u), a))), n),
                      n−s(u))
               lessproof(n, u, l)),
         rem(n−s(u),s(u))) ≡
     Equality([a]λ([y]when([l_1]apply(∀apply(y, a−s(u)), lessproof(a, u, l_1)),
                           [r_1]a,
                           order(s(u), a))),
              n,
              n−s(u),
              lessproof(n, u, l))

Idrem1(n: N): IdN(rem(n, 0), n) ≡ id(n)

Idrem2(n,u: N; q: s(u)≤n): IdN(rem(n, s(u)), rem(n−s(u), s(u))) ≡
     when([l]Eqrem(n,u,l)
          [r]bottomE(apply(lessasymm(s(u)),leqlesstoless(n, q, r)))
          order(s(u), n))

Idrem3(n,u: N; q: n<s(u)): IdN(rem(n, s(u)), n) ≡ when([l]bottomE(apply(lessasymm(s(u)),leqlesstoless(n, l, q))),
                                                   [r]id(n),
                                                   order(s(u), n))

Lessrem(n,u: N): rem(n, s(u))<s(u) ≡
     covrec([a]λ([y]when([l]Idlesstoless(rem(a−s(u), s(u)),
                                          Idrem2(a, u, l),
                                          apply(∀apply(y, a−s(u)), lessproof(a, u, l))),
                         [r]Idlesstoless(a, Idrem3(a, u, r), r),
                         order(s(u), a))), n)
```

division

The proposition concerning division, presented earlier in the main text is proved. The proof follows the same steps. Some parts of the proof are too long, so they are separated in lemmas.

```
div1(n:N): IdN(0*quo(n,0)+rem(n,0), n) ≡
    Idtrans( 0+n, addsubst(Idtrans(quo(n,0)*0, timescommut(quo(n,0),0), timeszero(quo(n,0))),
                        Idrem1(n)),
                        Idtrans(n+0, addcommut(n,0), addzero(n)))


div2a(n,u: N;q:s(u)≤n):
    IdN(s(u)*quo(n, s(u))+rem(n, s(u)),s(u)*quo(n-s(u), s(u))+(s(u)+rem(n-s(u), s(u)))) ≡
        Idtrans(s(u)*s(quo(n-s(u), s(u)))+rem(n-s(u), s(u)),
                addsubst(timessubstR(Idquo2(n, u, q)), Idrem2(n, u, q)),
                Idtrans(s(u)*quo(n-s(u), s(u))+s(u)+rem(n-s(u), s(u)),
                        addsubstL(timessucc(s(u),quo(n-s(u), s(u))))
                        addassocR(s(u)*quo(n-s(u), s(u)), s(u), rem(n-s(u), s(u)))))

div2b(n,u: N; q:s(u)≤n; v: IdN(s(u)*quo(n-s(u), s(u)) +rem(n-s(u), s(u)),n-s(u))):
    IdN(s(u)*quo(n-s(u), s(u))+(s(u)+rem(n-s(u), s(u))), n) ≡
        Idtrans(s(u)*quo(n-s(u), s(u))+(rem(n-s(u), s(u))+ s(u)),
                addsubstR(addcommut(rem(n-s(u), s(u)), s(u))),
                Idtrans(s(u)*quo(n-s(u), s(u))+rem(n-s(u), s(u))+s(u),
                        addassocL(s(u)*quo(n-s(u), s(u)) , rem(n-s(u), s(u)), s(u)),
                        Idtrans(n-s(u)+s(u),
                                addsubstL(v),
                                apply(Idminusadd(s(u), n), q))))))

div2(n,u: N;q:s(u)≤n; v: IdN(s(u)*quo(n-s(u), s(u)) +rem(n-s(u), s(u)),n-s(u))):
    IdN(s(u)*quo(n, s(u))+rem(n, s(u)), n) ≡
        Idtrans(s(u)*quo(n-s(u), s(u))+(s(u)+rem(n-s(u), s(u))), div2a(n,u,q), div2b(n,u,q,v))


div3(n,u: N; q:n<s(u)): IdN(s(u)*quo(n, s(u))+rem(n, s(u)), n) ≡
    Idtrans(0+n, addsubst(Idtrans(s(u)*0, timessubstR(Idquo3(n, u, q)), timeszero(s(u))), Idrem3(n, u, q)),
                Idtrans(n+0, addcommut(n, 0), addzero(n)))


div(n,m: N): IdN(m*quo(n, m)+rem(n, m), n) ≡
    natcases(div1(n),
            [u]covrec([a]λ([y]when([l]div2(a,u,l,apply(piapply(y, a-s(u)), lessproof(a, u, l))),
                                   [r]div3(a, u, r),
                                   order(s(u), a))),
                      n),
            m)
```