A case study in machine-assisted proofs: The Integers form an Integral Domain

Gustavo Betarte
Department of Computing Science
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: gustun@cs.chalmers.se

Abstract

We present a formalization of the set Z of integers using Martin-Löf's type theory. In particular we focus on the task of proving that this set with the operations + and $*$ form an Integral Domain. The proofs are developed for an inductive definition of Z, but we also discuss what kind of proofs could be obtained for a formulation where the set is defined as a quotient. The differences between both approaches when one is interested in regarding the computational meaning of proofs are pointed out.

In order to better reason about the proofs of the properties following from the postulates of an integral domain, an abstract formalization of this algebraic system is also proposed. With this, we aimed at not just being able to formally reflect the derivation of the properties independently of the concrete representation we were interested in, but also to translate these results to every algebraic structure satisfying those postulates.

Keywords and phrases: integers, type theory, integral domain, formal proofs.

Contents

Part I
Introduction

We will present in this paper some results obtained in an attempt to formalize the theory of integers in Martin-Löf's theory of sets with the help of the proof-assistant Alf [10]. In particular we focus on the task of proving that the set Z of integers with the operations $+$ and $*$ form an integral domain.

Although we follow closely the presentation of these topics in Birkhoff [3], our work is based on a substantially different approach. It is not a relevant problem for the authors of [3], for instance, to define Z , it is just assumed to exist. We will grasp what it means to be an integer following the general explanation in type theory (described in Martin-Löf [12], for instance) of what it means to be a set. There, a set is defined by prescribing how its canonical elements are formed as well as how two equal canonical elements of the set are formed. We will also follow the pattern of introducing a set by means of the formation, introduction, elimination and equality rules associated to it.

The former says that we can form a certain set from other certain sets or families of sets. With the introduction rules all the possible ways of constructing the canonical elements are made explicit. Thus, every set is inductively defined. The elimination rule states what one has to know in order to prove properties of (construct functions over) the elements of the set. It can be regarded as a kind of structural induction rule. The introduction and elimination rules are linked by means of the equality rules showing how the constructions defined in terms of the latter operate on the elements of the set generated by the introduction rules. Reading these equalities as conversions, one can think of the elimination rule as an operator defined for the set which can be naturally given a computational semantics. For example, the set N of natural numbers would be defined as:

N-formation

$$N\ set$$

N-introduction$_1$

$$0 \in N$$

N-introduction$_2$

$$\frac{n \in N}{succ(n) \in N}$$

N-elimination

$$\frac{\begin{array}{l} C(x)\ set\ [x \in N] \\ b \in C(0) \\ e(x,y) \in C(succ(x))\ [x \in N; y \in C(x)] \\ n \in N \end{array}}{natrec(b,e,n) \in C(n)}$$

where C is a family of sets indexed by N. Let us look at the explanation of *natrec*:

- first execute n getting a canonical element of N.

- if $n = 0$ then execute b which yields a canonical element $f \,\epsilon\, C(0)$. By the substitutivity rule for families of sets $f \,\epsilon\, C(n)$.
- otherwise $n = succ(m)$, with $m \,\epsilon\, N$. Then execute $e(m, natrec(b, e, m))$. If m has not the value 0 continue as in the second case until this value is reached.

The evident equality rules that follow from this explanation are:

$$\begin{array}{lcl} natrec(d, e, 0) & = & d \;\epsilon\; C(0) \\ natrec(d, e, succ(m)) & = & e(m, natrec(d, e, m)) \;\epsilon\; C(succ(m)) \end{array}$$

It can be noted that looking at these equalities as the definition of a function, it fits the schema of definition for primitive recursion:

$$\begin{array}{lcl} f(x_1, \ldots, x_n, 0) & = & d(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, succ(m)) & = & g(m, x_1, \ldots, x_n, f(x_1, \ldots, x_n, m)) \end{array}$$

So, every function defined in terms of *natrec* will be computable. The addition operation $+$ for N, for instance, can be defined as:

$$a \;+\; b \;\equiv\; natrec(a, (u, v)succ(v), b)$$

That is, apply b times $succ$ to a.

In type theory propositions are identified with sets, then proofs of propositions are identified with elements of sets. A proposition is defined by prescribing how we are allowed to prove it, and it is true if it is possible to exhibit a proof of it. So, we can think of a family $C(x)$ $(x \,\epsilon\; S)$ of sets as a propositional function (or a property) defined over the elements of the set S. If the proofs (constructions) in the second and third premisses and in the conclusion of the rule introducing *natrec* are suppressed, we can read that rule as a formulation of the principle of mathematical induction:

$$\frac{\begin{array}{l} C(x)\, prop\; [x \,\epsilon\, N] \\ C(0)\; true \\ C(succ(x))\; true [x \,\epsilon\, N; C(x)\; true] \\ n \,\epsilon\, N \end{array}}{C(n)\; true}$$

So, as remarked in [12], when propositions are identified with sets recursion and induction turn out to be the same concept. The usual properties of the addition $a + b$ (which can be easily proved inductively) are then formalized as functions (noncanonical constants) defined in terms of *natrec*. Thence proofs objects can be regarded as computational procedures.

With the intention of obtaining similar features for our formulation of the arithmetic of integers we are interested in working with an inductive definition of the set. Then, the disjoint union $(-N) + \{0_Z\} + N$ is the formalization we propose of Z. However, the elimination rule associated to this definition of the set $(Zcases)$ works just as a case operator. When trying to define the basic arithmetical operations for the set ($+$ and $*$, for instance) and prove some of their elementary properties we realized that the operator was not suitable enough to work with. For that reason the formalization of an induction

principle for the set is proposed and shown to be derivable in the theory. The recursion operator obtained (expressed in terms of *Zcases* and *natrec*) allows to define primitive recursive functions over the set.

An alternative approach for defining noncanonical constructions which dismisses the use of elimination rules is introduced in Coquand [5]. That approach aims at providing the possibility of defining functions in Martin-Löf's logical framework using pattern matching. It is argued (and illustrated with some examples) that the discipline of introducing new constants just by means of the elimination rules has some drawbacks. Let us look for instance how the definition of + above could be reformulated:

$$\begin{array}{lcl} +(a, 0) & = & a \\ +(a, succ(m)) & = & succ(+(a, m)) \end{array}$$

The readability of the definition is improved and the intuition of the computational behaviour of the constant is better displayed. With the aim of achieving mathematical constructions closer to programs we chose this approach to develop our proofs. It is known that at least for some subtle proofs this pattern-matching discipline is more powerful than the discipline of using elimination constants. However, that does not happen in the kind of proofs that we are interested in. All that we proved could be done using elimination rules.

In Martin-Löf [11] the character of (functional) programming language of type theory is emphasized and shown to be a suitable framework to reflect the close relation between constructive mathematics and programming. We use it as a programming logic, identifying specifications with propositions and sets, thus the proofs of the propositions can be interpreted as methods to construct elements in sets, or programs fitting a specification. The algorithmic nature of the proofs of the properties is then formally reflected as a construction in the language.

Nevertheless, we are also interested in gaining knowledge about the task of formalizing mathematics in type theory. For that reason we also present some basic proofs developed for a formalization of the set of integers following a more traditional approach. That is, we will regard Z as the quotient of the set of pairs of natural numbers by an appropriate equivalence relation. We briefly discuss what kind of proofs are obtained when working with this approach and, point out some drawbacks when one is interested in associating a computational meaning to the proofs.

As with every algebraic structure, from the postulates of an integral domain many properties can be derived by equational reasoning. Having formalized different concrete integral domains (the two representations of Z with the operations satisfying the postulates) one could think that it would be desirable to have the possibility of working with some formalization of the abstract notion of this algebraic system. Thus, we could reason about the derivations of the proofs independently of the concrete algebras we are working with. Moreover, it would also be interesting to be able to translate the results obtained from the abstract formalization to the concrete cases. This work is also concerned with a proposal of formalization of algebraic structures oriented to fulfill these requirements. We will regard algebraic systems as lists of hypotheses, where each clause will correspond to one of the components of the structure (carrier sets, function symbols or axioms). We use for this the notion of *context*. Then, for instance we could introduce the notion of *semigroup* as the context:

$$[S : Set; op : (S; S)S; assoc : (x, y, z : S)Id(S, op(x, op(y, z)), op(op(x, y), z))]$$

where Id is the propositional equality for the set S.

The proofs of the properties derived from the postulates of those systems will be formalized as scheme of proofs developed under the corresponding hypotheses. In that sense our proofs will be close to the style of algebra texts. This is naturally reflected in the framework where all the forms of judgements are relativized with respect to a context.

We will also show that with this approach we can not only formalize algebraic systems and their derived properties but we will also be able to represent particular instances of them. These features are achieved using the notion of *substitution*. We will represent a concrete algebra by providing a substitution fitting the context which describes the corresponding abstract algebra. For the example presented above we could define the particular semigroup formed by $\langle N, + \rangle$ as the substitution:

$$\{S := N; op := +; assoc := +_{assoc}\}$$

where $+_{assoc} : (a, b, c : N)Id(N, +(a, +(b, c)), +(+(a, b), c))$ is the constant associated to the proof that $+$ is associative.

The formalization of algebraic constructions will be presented in part II of this work. The inductive representation of the set Z, the discussion on inductive definitions and proofs, and the quotient version of the set are presented in part III. Finally, we discuss some drawbacks we think the approach of using contexts to formalize algebraic notions has.

Part II
Algebraic Structures as Contexts

1 Introduction

The definition of what an *algebraic system* is can be found in most books on modern algebra. Following MacLane [8], we will say that an algebraic system consists of :

1. A nonempty set S, called the *domain* of the algebra.
2. A set of *relations* on S.
3. A set of *operations* over S.
4. A set of *postulates* (or axioms) that are basic rules which the operations satisfy.

An algebraic system $\mathbf{S}$, whose domain is S and whose (finite) set of operations (or operation symbols) is $\{f_1, \ldots, f_k\}$, is denoted by $\mathbf{S} = \langle S, f_1, \ldots, f_k \rangle$.

Well known examples of algebraic systems are *semigroups* and *monoids* which are defined as :

A *semigroup* $\langle S, + \rangle$ is a set S together with a binary operation $+ : S * S \to S$ which is associative. A *monoid* $\langle M, +, 0 \rangle$ is a semigroup $\langle M, + \rangle$ with an element 0 of M which is a unit for $+$, i.e., the following identities hold for every x, y, z in M :

- M1. $(x+y)+z \;=\; x+(y+z)$
- M2. $x+0 \;=\; 0+x = x$

2 Using contexts to formalize algebraic systems

As mentioned in the introduction we will regard an algebraic system as a list of hypotheses where each clause (type assignment) will be associated to a component of the system. Then it will be needed to assume sets and function symbols ranging over those sets as well as formally express (as types) the axioms that describe the system. All these requirements will be achieved using Martin-Löf's logical framework (which is implemented in Alf) as the setting to formalize those notions. The introduction of a hypothetical set S is formulated as the type declaration $S : Set$, being Set a primitive type of the framework. A binary function symbol f defined over S, for instance, is introduced as $f : (x : S; y : S)S$, where $(x_1 : A_1; \ldots; x_n : A_n)B$ is the notation for dependent function types. As well as sets are extended to families of sets in type theory, families of types extend types in this theory. This will allow to express the axioms as types depending on the variables and parameters which are involved in them. Each assumption associated to an axiom will have the form of a variable whose type is the dependent product of the type expressing the axiom indexed by the variables acting in that type. Lists of type declarations are formalized as contexts, constructions which are governed by the following rules:

$$[\,] : Context \qquad \frac{\Gamma : Context \quad \alpha : type \quad [\Gamma]}{[\Gamma; x : \alpha] : Context}$$

where x does not occur free in Γ and $[\Gamma; x : \alpha]$ is the extension of the context Γ with the clause $x : \alpha$. Thus, a context is either empty or an extension of a proper context.

However, when formalizing an algebraic system its carrier will not just be assumed to be a set as intended in type theory, but a pair $< S, R >$, with R an equivalence relation on S. This requirement follows from the fact that we do not want to restrict the notion of equality —which plays a decisive role in the formulation of the postulates and derivation of new properties— to that of propositional equality which is the primitive notion of equality in Martin-Löf's set theory. Since we also want to work with the formalized systems as equational logics, every time a new operation is defined on the set the proof of the congruence of the relation R with respect to this operation must be provided. When a predicate over the set S is introduced we will also ask for the proof of the substitutivity property of the predicate w.r.t. the relation R. An example of what we achieve with these conditions is that we will be able to prove that different formalizations of a concrete algebra satisfy the conditions required to be a specific algebraic system. This feature will be illustrated when we show that two different formalizations of the set Z of integers, an inductive one with the equality taken as the propositional equality on the set, and a quotient version with a provided congruence relation, satisfy the postulates of an *integral domain*.

Let us see how the notion of set introduced above could be defined using contexts:

```
SET  is   [S:Set; R:(S;S)Set;
           refl:(x:S)R(x,x);
           symm:(x:S;y:S;R(x,y))R(y,x);
           trans:(x:S;y:S;z:S;R(x,y);R(y,z))R(x,z)]
```

The operator **is** is used to introduce abbreviations of contexts.

Another feature we want to reflect is that of defining new algebraic systems as extensions of previously defined ones. Suppose that we want to introduce a binary operation defined over the set S together with the proof that the relation R is congruent with respect to this operation. In order to formalize this we could define the following structure:

```
Grupoid   is   SET   +   [op:(S;S)S;
                          opcong:(x:S;y:S;z:S;w:S;R(x,y);R(z,w))
                                                  R(op(x,z),op(y,w))]
```

The operator + is also provided by the system; it denotes iterated extension of contexts. Note that we can refer explicitly to the components of the context `SET` (the set `S` and the relation `R`) when defining this new structure.

2.1 Concrete algebras as substitutions

As mentioned above we are also interested in formalizing the fact that some particular representation is an instance of one abstract system. For that, we will use the notion of *substitution*, or more precisely of a substitution for the variables that belong to a given context. This notion is carefully explained in Tasistro [19] and we give here the basic

explanations. To say that γ is a substitution fitting the context Δ means that γ is an assignment of objects of appropriate types to the variables of the context Δ. This situation is generalized to the case where the objects may depend on variables of another context, say Γ, written as $\gamma : \Delta \quad [\Gamma]$. Suppose now that we know that α is a *type* under the context Δ. Then we can perform the substitution γ on α to obtain the type $\alpha\gamma$ under the context Γ. The rules governing the construction of substitutions are listed below :

$$\{\ \} : [\ ] \quad [\Gamma] \qquad\qquad \frac{\gamma : \Delta \quad [\Gamma] \quad \alpha : type \quad [\Delta] \quad a : \alpha\gamma \quad [\Gamma]}{\{\gamma; x := a\} : [\Delta; x : \alpha] \quad [\Gamma]}$$

where $\{\gamma; x := a\}$ is the extension of the substitution γ with the assignment of the object a to the fresh variable x. Then we will assert that some concrete algebraic structure γ is classified as a particular case of an abstract algebraic structure Δ if $\gamma : \Delta\ [\ ]$. To illustrate this, assume that we have:

- defined the set N of natural numbers,
- the binary operation $+$ on N,
- the set $IdN(m, n) \ = \ Id(N, m, n)$ for $m, n \in N$.
- constructed the proof (named *addcong*) that IdN is congruent with respect to $+$.

We could then define the substitutions `Neq` and `N+` as:

```
Neq  is   {S:=N; R:=IdN; refl:=Idrefl;
           symm:=Idsymm; trans:=Idtrans} : SET []

N+  is   {Neq;op := +; cong := addcong} : Groupoid []
```

2.2 A further example: groups and the concrete group Z_2

First, we will introduce the definition of `Semigroup` as an extension of `Groupoid`:

```
Semigroup is  Groupoid + [assoc:(x:S;y:S;z:S)R(op(x,op(y,z)),
                                               op(op(x,y),z))]
```

That is, `assoc` is the assumption that `op` is associative on `S`. Then we define `Monoid` as an extension of `Semigroup`:

```
Monoid    is   Semigroup   +   [unit:S;
                                ident:(x:S)and(R(op(x,unit),x),
                                               R(op(unit,x),x))]
```

That is, `unit` will be a distinguished element of the set `S` and `ident` states the axiom that `unit` is the identity element for `op`. Finally, `Group` is defined as :

```
Group     is   Monoid    +    [inv:(S)S;
                               opinv:(x:S)and(R(op(x,inv(x)),unit),
                                              R(op(inv(x),x),unit))]
```

That is, `inv` denotes a unary operation on `S` and `opinv` characterizes the inversion property of `inv`.

A familiar algebraic structure is Z_2, which is formed by the set with two different elements (ze, one) , together with a binary operation $xor : (Z_2; Z_2)Z_2$, which is defined as follows:

```
xor(ze,ze)    =  ze
xor(one,ze)   =  one
xor(ze,one)   =  one
xor(one,one)  =  ze
```

If we take the equality relation to be the propositional equality over Z_2 (IdZ_2) and the inverse as the identity function on Z_2 (i_2), we can prove that:

- IdZ_2 is congruent with respect to xor ($xorcong$)
- xor is associative ($xorassoc$)
- ze is the identity element for xor ($xorunit$)
- the result of applying xor to x and the inverse of x yields ze ($xorinv$)

Then, we can define the following substitution:

```
GroupZ_two is {S:= Z_two; R:=IdZ_two; refl:=Idrefl(Z_two);
               symm:=Idsymm(Z_two); trans:=Idtrans(Z_two); op:=xor;
               cong:=xorcong; assoc:=xorassoc; unit:=ze;
               ident:=xorunit; inv:=i_two; opinv:=xorinv} : Group []
```

It can be interpreted as the verification that Z_2 is a group.

3 Proofs over algebraic systems

We now turn to how to deal with proving the properties that can be derived from the postulates of such a system. Furthermore, suppose that a proof schema for, say, a property P which is a consequence of the postulates of a system A is constructed and, that a concrete representation A_{ins} of A is provided. Then, we will show how to obtain the proof term P_{ins} which will represent the instantiation of the property P to the particular algebra A_{ins}.

3.1 Proof scheme

We will construct the derivation of one property that is satisfied for any group G, the *left cancellation law*, which is formulated as:

For all $a, b, c \in G_{Set}$, if $op(c, a) \cong op(c, b)$ holds, then $a \cong b$

where G_{Set} denotes the carrier set of the group (S above) and $\cong$ is a more usual notation for R.

Proof

If $a, b, c \in G_{Set}$, then

i)	$op(c,a)$	$\cong$	$op(c,b)$	*(by hypothesis)*
ii)	$op(inv(c),op(c,a))$	$\cong$	$op(inv(c),op(c,b))$	*(by i and cong)*
iii)	$op(op(inv(c),c),a)$	$\cong$	$op(op(inv(c),c),b)$	*(by ii and assoc)*
iv)	$op(unit,a)$	$\cong$	$op(unit,b)$	*(by iii, opinv and cong)*
v)	a	$\cong$	b	*(by iv and ident)*

In each deduction step of this proof a simple but useful property of the equivalence relation $\cong$ is used:

If $x \cong y$, $x \cong z$ and $y \cong w$ hold, then $z \cong w$ with $x, y, z, w \in S$.

This is proved using the symmetric and transitive properties of $\cong$. It will be referred below as `simrepl`.

In the case of this kind of derivations, the task of constructing the proof term is straightforward. With the help of the proof assistant, the proof construction process is strictly top-down, one only has to do sequential refinements of the goal using the proof terms associated to each deduction step. Each of these terms is formalized as a functional expression which for given values of the types corresponding to the premisses yields a value of the type corresponding to the conclusion. The entire derivation will also be a functional expression built up from the combination of those functions. As an example, here is the formalization of the derivation of the cancellation property,

```
cancelleft =
  [a,b,c,h]
  simrepl(op(unit,a),op(unit,b),
          a,b,
          simrepl(op(op(inv(c),c),a),op(op(inv(c),c),b),
                  op(unit,a),op(unit,b),
                  simrepl(op(inv(c),op(c,a)),op(inv(c),op(c,b)),
                          op(op(inv(c),c),a),op(op(inv(c),c),b),
                          cong(inv(c),inv(c),
                               op(c,a),op(c,b),refl(inv(c)),h),
                          assoc(inv(c),c,a),
                          assoc(inv(c),c,b)
                         ),
                  cong(op(inv(c),c),unit,a,a,snd(opinv(c)),refl(a)),
                  cong(op(inv(c),c),unit,b,b,snd(opinv(c)),refl(b))
                 ),
          fst(ident(a)),
          fst(ident(b))
         ) : (a,b,c:S;h:R(op(c,a),op(c,b)))R(a,b) Group
```

The notation $[x_1, \ldots, x_n]e$ is used to express the abstraction of the variables $x_1, \ldots, x_n$ in the expression e. The outermost application of `simrepl` is associated to the deduction step from iv) to v); the next inner to the step from iii) to iv), and so on.

3.2 Proofs as instances

In section 2 we showed how to define particular representations of an algebraic system using substitutions. We will now present how to obtain also instantiations of properties

derived from the postulates of the system for those representations.

We defined $GroupZ_2$ to be the substitution that establishes that Z_2 is a group. Then, we can construct the proof that the cancellation property holds for the operation *xor* in this way:

```
xorcancel = cancelleft{GroupZ_two}:(x,y,z:Z_two,
                                    IdZ_two(xor(z,x),xor(z,y))
                                   )IdZ_two(x,y)
```

and now we get a term which represents the proof of the cancellation law for the operation of Z_2. These kind of definitions are justified by means of the following rules of substitution:

$$\frac{\gamma : \Delta \quad [\Gamma] \quad \alpha : type \quad [\Delta]}{\alpha\gamma : type \quad [\Gamma]} \qquad \frac{\gamma : \Delta \quad [\Gamma] \quad a : \alpha \quad [\Delta]}{a\gamma : \alpha\gamma \quad [\Gamma]}$$

They are explained as:" if we know that γ is a substitution for the variables of the context Δ depending on the variables of the context Γ and α is type (resp. a is an element of type α) under the context Δ, then α (resp. a) with γ is a type (resp. an element of α with γ) under the context Γ.

3.3 Using derived properties to define concrete algebras

All the examples above are concerned with definition of abstract algebras and properties that can be derived from the postulates, and their respective instantiations. Now we will show another way of obtaining instances of algebraic systems. There is a well known theorem in the theory of groups which says:

If G is a group, and for all $x \in G_{Set}$, $x^2 \cong unit$, then G is an *abelian group*

where x^2 is a more convenient way of writing $op(x, x)$.

To assert that a group is abelian we have to provide the proof that the binary operation *op* is commutative. Then, assume that for all $x \in G_{Set}$, $x^2 \cong unit$ and the lemmas below which follow (easily) from the axioms of group:

1. For all $x, y \in G_{Set}$, $op(x, y)^{-1} \cong op(y^{-1}, x^{-1})$.
2. For all $x \in G_{Set}$, if $x^2 \cong unit$ holds then $x \cong x^{-1}$

reading x^{-1} as $inv(x)$. The commutativity of the operation *op* can be derived as follows: For all $x, y \in G_{Set}$,

i)	$op(x, y)$	$\cong$	$op(x, y)^{-1}$	*(by lemma 2 and hypothesis)*
ii)	$op(x, y)^{-1}$	$\cong$	$op(y^{-1}, x^{-1})$	*(by lemma 1)*
iii)	$op(x, y)$	$\cong$	$op(y^{-1}, x^{-1})$	*(by i,ii and trans)*
iv)	$op(y^{-1}, x^{-1})$	$\cong$	$op(y, x)$	*(by lemma 2 and cong)*
v)	$op(x, y)$	$\cong$	$op(y, x)$	*(by iii,iv and trans)*

As done with `cancelleft`, we can introduce the definition:

```
commop = <proof-code>:(x:S)R(op(x,x),unit)(y,z:S)R(op(y,z),op(z,y)) Group
```

It can be noted that this property holds for Z_2, because:

- $xor(ze, ze) \cong ze$,
- $xor(one, one) \cong ze$,
- ze is the identity element for Z_2.

Then if $xorsqid$ is the name of the proof that for all $x \in Z_2$ $xor(x, x) \cong ze$, we can define:

```
commxor = commop{GroupZ_two}(xorsqid) : (x,y:Z_two)IdZ_two(xor(x,y),
                                                          xor(y,x)) []
```

Then `commxor` is the name associated to the proof that *xor* is commutative on Z_2.

Now we extend the structure `Group` with the proof that `op` is commutative to define the structure abelian group as:

```
AbGroup is Group + [comm :  (x,y:S)R(op(x,y),op(y,x)]
```

We could then introduce this new concrete structure as a substitution:

```
AbGroupZ_two  is  {GroupZ_two;comm := commxor} : AbGroup
```

to state that Z_2 is an abelian group.

4 Integral Domains

It is very common to find the set of integers presented as a particular case of an algebraic system. In [3] the authors assume that the set Z together with the binary operations $+$ and $*$ satisfies the postulates required to be an *integral domain*. We list below these postulates assuming that S is a set and $+$ and $*$ are defined over S:

(i) Commutative laws. For all a and b in S, $a + b = b + a$ and $a * b = b * a$;

(ii) Associative laws. For all a, b, and c in S, $a + (b + c) = (a + b) + c$ and $a * (b * c) = (a * b) * c$;

(iii) Distributive law (left). For all a, b, and c in S, $a * (b + c) = a * b + a * c$;

(iv) Zero. S contains an element 0 such that $a + 0 = a$ for all a in S;

(v) Unity. S contains an element $1 \neq 0$ such that $a * 1 = a$ for all a in S;

(vi) Additive inverse. For each a in S, the equation $a + x = 0$ has a solution x in S;

(vii) Cancellation law. If $c \neq 0$ and $c * a = c * b$, then $a = b$

In more concise terms, S is a commutative ring with unit and a left cancellation law on $*$.

From these postulates, and using that $=$ is an equivalence relation, many well-known properties involving $+$ and $*$ can be deduced using equational reasoning. On this fact a meaningful reflexion is added by the authors *"... one must, however, take some care in deducing consequences from postulates in this way in order to be sure that the proofs use only the postulates listed and rules of logic"*. With the formalization we have done of algebraic systems together with the congruence and substitutivity requirements and the proof-assistant control we achieve the required safeness.

4.1 Formalization of integral domains

We extend the definition of abelian group to define ring as:

```
Ring is AbGroup + [mult:(S;S)S;
                   multcong:(x,y,z,w:S;R(x,y);R(z,w))
                                    R(mult(x,z),mult(y,w));
                   munit:S; diff:not(R(munit,unit));
                   massoc:(x,y,z:S)R(mult(x,mult(y,z)),
                                     mult(mult(x,y),z));
                   mident:(x:S)R(mult(x,munit),x);
                   distlft:(x,y,z:S)R(mult(x,op(y,z)),
                                      op(mult(x,y),mult(x,z)))]
```

Then we define commutative ring as:

```
CommRing is  Ring + [*comm:(x,y:S)R(*(x,y),*(y,x))]
```

to finally introduce the definition of Integral Domain:

```
IntDom is CommRing + [*cancel:(x,y,z:S;
                               not(R(z,unit));
                               R(*(z,x),*(z,y))R(x,y)]
```

4.2 Derived properties

This is an example of formal proof of a property valid in any *integral domain* which can be derived from the postulates and the fact that $\simeq$ is an equivalence relation.

For S, the law is formulated in this way:

Distributive law (right). For all a,b, and c in S, $\quad (a+b)*c \simeq a*c+b*c$

Proof.

Assume a, b, c in S ; then:

i)	$(a+b)*c$	$\simeq$	$c*(a+b)$	*(by commutative law for $*$)*
ii)	$c*(a+b)$	$\simeq$	$c*a+c*b$	*(by left distributive law)*
iii)	$(a+b)*c$	$\simeq$	$c*a+c*b$	*(by i, ii and transitivity of $\simeq$)*
iv)	$c*a$	$\simeq$	$a*c$	*(by commutative law for $*$)*
v)	$c*b$	$\simeq$	$b*c$	*(by commutative law for $*$)*
vi)	$c*a+c*b$	$\simeq$	$a*c+b*c$	*(by iv, v and substitutivity of $+$)*
vii)	$(a+b)*c$	$\simeq$	$a*c+b*c$	*(by iii, vi and transitivity of $\simeq$)*

The Alf term representing this derivation is an explicit definition which combines the proof terms associated to each postulate and the properties of $\simeq$ involved in the deduction. It can be found in Appendix A.

Other derived properties holding for any particular integral domain that we have also formally proved are:

1. If z in S has the property that $a+z \simeq a$ for all a in S, then $z \simeq unit$.

2. For a and b in S, there is one and only one x in S such that $a + x \simeq b$.
3. If u has the property that $a * u \simeq a$, for all a in S, then $u \simeq *unit$.
4. For all a in S, $a * unit \simeq unit \simeq unit * a$.
5. For all a and b in S, $a^{-1} * b^{-1} \simeq (a * b)^{-1}$.
6. If $a * b \simeq unit$ and $a \neq unit$ then $b \simeq unit$.

The formal proofs of some of these properties can be found in appendix A.

Part III
Formalization of the Integers in Martin-Löf's Theory of Sets

1 Introduction

We mainly devote this part to present how recursion over Z can be formalized. In section 2 we present the definition of the integers. In section 3 two possible inductive principles that can be represented by recursion operators derived from the basic definitions of the theory are discussed. In this section we also introduce as an alternative the possibility of defining noncanonical constants by computation rules that are pattern matching equations (see Coquand [5]). This latter feature is provided by Alf [10].

We illustrate with one example (associativity of addition) the kind of proofs we had to develop in order to formally prove that our representation of Z satisfies the postulates of an integral domain.

An alternative way of formalizing the integers —regarding them as equivalence classes of pairs of natural numbers— is also presented. Although we did not formalize all the proofs of the properties for the quotient version of the set, some of them are displayed in order to point out the differences between both approaches.

Finally, using the concrete representation of the set, the operations and their properties, a substitution fitting the algebraic system `IntDom` is defined.

2 Inductive definition

The set of integers will be regarded as the set $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, which can be formalized as the disjoint union $(-N) + \{0_Z\} + N$. Here are our rules:

Z-formation

$$Z \; set$$

Z-introduction$_1$

$$0_Z \, \epsilon \, Z$$

Z-introduction$_2$

$$\frac{n \, \epsilon \, N}{pos(n) \, \epsilon \, Z}$$

Z-introduction$_3$

$$\frac{n \, \epsilon \, N}{neg(n) \, \epsilon \, Z}$$

where, if $n \, \epsilon \, N$, $pos(n)$ will represent the integer $n + 1$ and $neg(n)$ the integer $-(n + 1)$. The elimination rule will be just a case analysis over these three possibilities, that is:

Z-elimination

$$\frac{\begin{array}{l} C(x)\ set\ [x \epsilon Z] \\ b \epsilon C(0_Z) \\ d(n) \epsilon C(pos(n))\ [n \epsilon N] \\ e(n) \epsilon C(neg(n))\ [n \epsilon N] \\ z \epsilon Z \end{array}}{Zcases(b, d, e, z) \epsilon C(z)}$$

and the equality rules are defined as one would expect. We will follow the notation of Nordström et al. [14].

When one works in type theory, the only implicit functions that one is allowed to introduce are the elimination constants defined for each set. Every other definition must be given in terms of them. As we were interested in extending the above definition of the set Z with the usual arithmetical operations, and above all to prove some properties of those operations, a good formulation for them was the first goal to achieve.

In Szasz [16] it is shown that if you want to define addition ($+_Z$) over the integers in terms of *Zcases*, one possibility is to use addition and subtraction over N as well as propositional equality $=_N$ and the order relation $<$. Moreover, it is also necessary to prove associativity and commutativity of natural addition, that $=_N$ is an equivalence relation, and transitivity and trichotomy of $<$. In such case, the intuition behind the expression obtained for $+_Z$ is not easy to grasp, and looking at it as a program, it is computationally inefficient. Another possibility could be to treat the complicated cases $pos(m) + neg(n)$ and $neg(m) + pos(n)$ doing double induction over m and n, but that would not improve the readability of the expression.

A more elegant solution is proposed below.

3 Recursion on Z

In this section we will discuss alternative ways of introducing definitions and construct proofs of properties inductively over our formalization of the set.

3.1 Two different recursion operators for Z

We will present two possible versions of an induction principle for our formulation of Z that behave similarly to the one corresponding to N. Both recursion operators are derived inside the theory and expressed in terms of *Zcases* and *natrec*.

The main difficulty when defining the operator is that the usual order over Z is not well-founded. The way this problem is treated is what makes the difference between the two approaches.

Using sign predicates

This version was also presented in [16]. Assuming that P is a propositional function over Z an induction principle can be formulated as follows:

$$\frac{P(0_Z) \quad P(succ_Z(x))\ [x \epsilon Z, nonneg(x), P(x)] \quad P(pred_Z(x))\ [x \epsilon Z, nonpos(x), P(x)] \quad z \epsilon Z}{P(z)}$$

The functions $succ_Z$ and $pred_Z$ are the successor and predecessor in Z respectively. The predicates *nonpos* and *nonneg* state the non-positivity and non-negativity of their arguments respectively. As we identify propositions with sets, the definition of *nonpos* and *nonneg* have to be functions that when applied to an integer return a set. Furthermore, the intended meaning of these predicates is to classify the canonical elements of the set (a similar problem as to prove that $0 \neq succ(n)$, for all $n \in N$). In [14] it is shown how to deal with this kind of problems introducing the set U of small sets (or the first universe). As we have constructed our proofs in Martin-Löf's logical framework, we can avoid using the universe by defining this special elimination rule:

```
ZcasesU  :  (S:Set;pe:(N)Set;ne:(N)Set;z:Z)Set
```

which follows the schema proposed in Smith [15] (note that the result of applying ZcasesU yields a *Set*, not an element in a set $C(z)$ as in *Zcases*). The computation rules are defined as:

```
ZcasesU(S,pe,ne,0_Z)      =  S : Set
ZcasesU(S,pe,ne,pos(n))   =  pe(n) :  Set
ZcasesU(S,pe,ne,neg(n))   =  ne(n) :  Set
```

with `S:Set, pe:(N)Set, ne:(N)Set, n:N`.

Now, we can define the predicate *nonneg* in this way:

```
noneg(z) = ZcasesU(⊤,[n]⊤,[n]{},z) :  Set [z ∈ Z]
```

where ⊤ represents the one-element set (the true proposition), and {} the empty set (absurdity). The predicate *nonpos* can be defined in a similar way.

The derived induction rule associated to the induction principle for Z presented above is:

$$Zrec_U \quad \frac{\begin{array}{l} P(x)\ set\ [x \in Z] \\ b \in P(0_Z) \\ hp(x, nn, hx) \in P(succ_Z(x))\ [x \in Z,\ nn \in nonneg(x), hx \in P(x)] \\ hn(y, np, hy) \in P(pred_Z(y))\ [y \in Z,\ np \in nonpos(y), hy \in P(y)] \\ z \in Z \end{array}}{Zrec_U(b, hp, hn, z) \in P(z)}$$

and $Zrec_U$ defined as:

$$\begin{array}{rl} Zrec_U(b, hp, hn, z) = & Zcases(b, \\ & [n]natrec(hp(0_Z, tt, b), [u, v]hp(pos(u), tt, v), n), \\ & [n]natrec(hn(0_Z, tt, b), [u, v]hn(neg(u), tt, v), n), \\ & z) \end{array}$$

where $tt \in \mathsf{T}$.

A different formulation

In the definition of the rule above, *nonneg* and *nonpos* are needed to enforce that the inductive steps *hp* and *hn* are applied starting from 0_Z along the positive and negative branches respectively. Now, we will present another formulation of the induction principle —suggested by Björn von Sydow— where the predicates are not needed at all.

First, let us define $nonnegpred(n) \epsilon Z$ (predecessor of $pos(n)$) and $nonpossucc(n) \epsilon Z$ (successor of $neg(n)$), with $n \epsilon N$ as:

$$\begin{aligned} nonnegpred(n) &\equiv natrec(0_Z, [u,v]pos(u), n) \epsilon Z\ [n \epsilon N] \\ nonpossucc(n) &\equiv natrec(0_Z, [u,v]neg(u), n) \epsilon Z\ [n \epsilon N] \end{aligned}$$

Now we can define this alternative induction principle:

$$\frac{P(0_Z) \quad P(pos(x))[x \epsilon N, P(nonnegpred(x))] \quad P(neg(x))[x \epsilon N, P(nonpossucc(x))] \quad z \epsilon Z}{P(z)}$$

where $P(nonnegpred(x))$ and $P(nonpossucc(x))$ state that the property P holds for the predecessor of $pos(x)$ and the successor of $neg(x)$ respectively. Note that by the definitions of *nonpossucc* and *nonnegpred*, for the cases $pos(0)$ and $neg(0)$ those hypotheses become $P(0_Z)$. This induction rule can be derived from the earlier definitions.
Assume:

- $P(x)\ set\ [x \epsilon Z]$,
- $b \epsilon P(0_Z)$,
- $ps(x, hx) \epsilon P(pos(x))\ [x \epsilon N, hx \epsilon P(nonnegpred(x))]$,
- $ns(y, hy) \epsilon P(neg(y))\ [y \epsilon N, hy \epsilon P(nonpossucc(y))]$, and
- $z \epsilon Z$.

Then from these assumptions we will construct an element in $P(z)$ for all z in Z. For the case $z = 0_Z$ we take b, for the cases $z = pos(n)$ and $z = neg(n)$ we proceed by induction over n.

- Let $n \epsilon N$
  - For $n = 0$ we have that $nonnegpred(0) = 0_Z \epsilon Z$ (by definition of *nonnegpred*) and that $b \epsilon P(0_Z)$.
    So, $ps(0, b) \epsilon P(pos(0))$.
  - Now, assume $u \epsilon N$ and $v \epsilon P(pos(u))$.
    We have that $nonnegpred(succ(u)) = pos(u) \epsilon Z$ (by definition of *nonnegpred*)
    Then $ps(succ(u), v) \epsilon P(pos(succ(u)))$.
- In similar way, for any $n \epsilon N$, we can find an element in $P(neg(n))$.

From this proof it is straightforward to define a operator in terms of *Zcases* and *natrec*:

$$\begin{aligned} Zrec(b, ps, ns, z) = & Zcases(b, \\ & [n]natrec(ps(0_Z, b), [u,v]ps(succ(u), v), n), \\ & [n]natrec(ns(0_Z, b), [u,v]ns(succ(u), v), n), \\ & z) \end{aligned}$$

Thus, *Zrec* will work as an operator for defining functions over Z by primitive recursion.

Defining functions with Zrec

Let us look how the definitions of the arithmetical operations for Z are expressed in terms of *Zrec*.

First, we introduce the successor and predecessor functions for Z:

$$\begin{array}{rcl} succ_Z(z) & \equiv & Zcases(pos(0), [n]pos(succ(n)), [n]nonpossucc(n), z) \in Z \ [z \in Z] \\ pred_Z(z) & \equiv & Zcases(neg(0), [n]nonnegpred(n), [n]neg(succ(n)), z) \in Z \ [z \in Z] \end{array}$$

Now we can define the addition, subtraction and multiplication for Z in this way,

$$\begin{array}{rcl} a +_Z b & \equiv & Zrec(b, [n,h]succ_Z(h), [n,h]pred_Z(h), a) \in Z \ \ [a, b \in Z] \\ a -_Z b & \equiv & Zrec(a, [n,h]pred_Z(h), [n,h]succ_Z(h), b) \in Z \ \ [a, b \in Z] \\ a *_Z b & \equiv & Zrec(0_Z, [n,h]h +_Z b, [n,h]h -_Z b, a) \in Z \ \ [a, b \in Z] \end{array}$$

which look very similar to the definitions of the corresponding operations over N.

3.2 Pattern Matching

A suitable recursion operator for Z should satisfy two requirements: it should of course be sound and the proofs constructed with it should be natural and readable.

The former requirement is satisfied by the definition of $+_Z$ in terms of *Zcases* we commented above, but surely we can agree that the one line definition expressed using *Zrec* (which is defined in terms of *Zcases* and the elimination constant *natrec*) should be easier to grasp. With this recursion operator, then, we could define functions over Z like $+_Z$,$-_Z$ and $*_Z$ in the way one is used to do in Martin-Löf's Type Theory.

In Coquand [5], a different perspective is proposed introducing the possibility to define implicit constants in Martin-Löf's logical framework whose computation rules are defined using pattern matching over the arguments. To ensure correctness of the definitions two conditions are required: the equations must be well-founded and patterns must not overlap and must cover all cases. If these requirements hold, a sufficient condition stating the totality of the function defined is satisfied.

Another definition of $+_Z$

A definition of this function in Alf using pattern matching could be:

```
zadd(0_Z,b)       =  b
zadd(pos(n),b)    =  zs(zadd(nonnegpred(n),b))
zadd(neg(n),b)    =  zp(zadd(nonpossucc(n),b))
```

The functions `zs` and `zp` denote the codification of $succ_Z$ and $pred_Z$ respectively.
Let us write again the expression obtained using *Zrec*:

$$a +_Z b \equiv Zrec(b, [n,h]succ_Z(h), [n,h]pred_Z(h), a) \in Z \ [a, b \in Z]$$

There is a clear relation between the right hand side of each equation in the definition of `zadd` and the computation that is performed when the expression in terms of *Zrec* is applied to the same pair of arguments. But there is one important difference, too. In the case of the definition in terms of *Zrec*, we have a direct argument that the computation

of $+_Z$ when applied to every pair of integers will always yield a value, because *Zrec* is a recursive operator which allows us to define this function as a primitive recursive one.

The condition proposed in [5] to ensure that the equations with recursive calls are well-founded is split into two requirements: nested occurrences of the constant f being defined are forbidden in the recursive calls $f(v_1, \ldots, v_n)$ and there must exist one v_i *structurally smaller* than the argument in the same position in the left hand side of the equation. The definition of the relation involved in the second requirement can be found in the same paper.

The first requirement is satisfied by our definition above, but not the second. Although we know that *nonnegpred*(resp. *nonpossucc*) is a total function over N and intuitively one can see that the chain of the successive calls of *zadd* will stop at $zadd(0_Z, b)$, the expression $nonnegpred(n)$ (resp. $nonpossucc(n)$) does not match the definition of being structurally smaller than $pos(n)$ (resp. $neg(n)$). We can refine the pattern n in the two last equations above, and define now *zadd* as:

```
zadd(0Z,b)          =  b
zadd(pos(0),b)      =  zs(b)
zadd(pos(s(m)),b)   =  zs(zadd(pos(m),b))
zadd(neg(0),b)      =  zp(b)
zadd(neg(s(m)),b)   =  zp(zadd(neg(m),b))
```

However, $pos(m)$ (resp. $neg(m)$) do not yet match the definition of being structurally smaller than $pos(s(m))$ (resp. $neg(s(m))$). But it seems sensible to extend the definition of this relation to consider cases like this here.

4 Proving the postulates

The binary operator $*_Z$ was also defined by pattern matching and the equality relation used in the postulates taken to be the propositional equality over the set *Z*.

With the example below we want to illustrate how the postulates were proved using inductive reasoning.

4.1 An inductive proof

Associativity of $+_Z$. For all x,y, and z in Z, $\quad (x +_Z y) +_Z z =_Z x +_Z (y +_Z z)$

Proof.

Assume that for all x, y in Z we have already proved the following properties :

(szadd) $succ_Z(x +_Z y) = succ_Z(x) +_Z y$
(pzadd) $pred_Z(x +_Z y) = pred_Z(x) +_Z y$

Let now $x \in Z$, then

- For $x = 0_Z$ we have that
  - $0_Z +_Z (y +_Z z) = y +_Z z$
  - $(0_Z +_Z y) +_Z z = y +_Z z$ *(both by definition of $+_Z$).*

- $0_Z +_Z (y +_Z z) = (0_Z +_Z y) +_Z z$ *(by symmetry and transitivity of =).*

- For $x = pos(n)$, with $n \ \epsilon N$

  - If $n = 0$, then
    - $(pos(0) +_Z y) +_Z z = succ_Z(y) +_Z z$.
    - $pos(0) +_Z (y +_Z z) = succ_Z(y +_Z z)$ *(both by definition of* $+_Z$*).*
    - $(pos(0) +_Z y) +_Z z = pos(0) +_Z (y +_Z z)$ *(by* **szadd***, symmetry and transitivity of =).*
  - Now, assume $n = succ(u)$ and $(pos(u) +_Z y) +_Z z = pos(u) +_Z (y +_Z z)$, then
    - $(pos(succ(u)) +_Z y) +_Z z = (succ_Z(pos(u) +_Z y)) +_Z z)$ *(by definition of* $+_Z$*).*
    - $(succ_Z(pos(u) +_Z y) +_Z z) = succ_Z((pos(u) +_Z y) +_Z z)$ *(by* **szadd***).*
    - $(pos(succ(u)) +_Z y) +_Z z = succ_Z(pos(u) +_Z (y +_Z z))$ *(by substitutivity and transitivity of* $=$ *and induction hypothesis).*
    - $succ_Z(pos(u) +_Z (y +_Z z)) = pos(succ(u)) +_Z (y +_Z z)$ *(by* **szadd** *and definition of* $succ_Z$*).*
    - $(pos(succ(u)) +_Z y) +_Z z = pos(succ(u)) +_Z (y +_Z z)$ *(by transitivity of =).*

- Then by induction over N we get the proof for every $n \ \epsilon \ N$, hence, for every positive integer.

- In a similar way it is proved that the property holds for $x = neg(n)$ (using **pzadd** instead of **szadd**).

For each postulate the same schema of proof as above is followed, that is, doing case analysis on Z and induction over N in the positive and negative case. This is very well reflected by the Alf proof terms when all the power of the pattern matching definition is used. We will have one equation for each item described above and for the cases $pos(succ(u))$ and $neg(succ(u))$ the right hand side of the equation will include a recursive call on $pos(u)$ and $neg(u)$. The formalization of this proof, together with those of the rest of the postulates, can be found in Appendix B.

5 Integers as a quotient set

We remarked earlier that we were interested in developing the formalization of the set of integers as an inductively defined set. In this way the algorithmic nature of many constructions defined over the set could be better reflected than when representing Z as a quotient set. We will now present the formalization following the more traditional definition of the set of integers, that is, as the quotient of the set of pairs of natural numbers.

There is no primitive construction in type theory which allows to introduce such kind of sets. We will define Z_p as an abbreviation for $N \times N$ and then introduce the definition of the relation $\cong_{Zp}$ on Z_p together with the proofs that it is an equivalence relation. We define $< m, n > \cong_{Zp} < p, q >$ to mean that $m +_N q =_N p +_N n$, where $=_N$ and $+_N$ denote

the propositional equality and the addition operation over N. Thus, $\cong_{Zp}$ is a decidable relation. This is formalized as :

$$Z_p \;=\; N \times N \,\epsilon\, Set \qquad \frac{m, n \,\epsilon\, N}{int(m,n) \,\epsilon\, Z} \qquad \frac{m, n, p, q \,\epsilon\, N \quad m +_N q =_N p +_N n}{int(m,n) \cong_{Zp} int(p,q)}$$

The proofs that $\cong_{Zp}$ is an equivalence relation are strongly based on the fact that $=_N$ is an equivalence relation on N and congruent w.r.t. $+_N$. Associativity and commutativity of $+_N$ are also needed to prove transitivity of $\cong_{Zp}$.
Now we define the addition operation $+_Z$ for Z_p as :

$$int(m,n) +_Z int(p,q) = int(m +_N p, n +_N q)$$

In order to guarantee that the operation behaves uniformly over elements belonging to the same equivalence class we proved first that :

$$\frac{a, b, c \,\epsilon\, Z_p \quad a \cong_{Zp} b}{a +_Z c \cong_{Zp} b +_Z c} \qquad \frac{a, b, c \,\epsilon\, Z_p \quad a \cong_{Zp} b}{c +_Z a \cong_{Zp} c +_Z b}$$

to finally prove that $\cong_{Zp}$ is congruent w.r.t. $+_Z$, that is :

$$\frac{a, b, c, d \,\epsilon\, Z_p \quad a \cong_{Zp} b \quad c \cong_{Zp} d}{a +_Z c \cong_{Zp} b +_Z d}$$

We were also interested in defining the subtraction and multiplication operations and proving the postulates as done with the inductive formalization of the set. As expected, in most cases the proofs of the properties rely mainly on the proofs developed for the arithmetical operations defined over N. For instance, to prove that $+_Z$ is associative, we can proceed as follows:

Assuming that $a = int(m,n)$, $b = int(p,q)$ and $c = int(r,s)$,

$$\begin{array}{lll}
i) & (a +_Z b) +_Z c & = \; int(m +_N p, n +_N q) +_Z int(r,s) \\
ii) & & = \; int((m +_N p) +_N r, (n +_N q) +_N s) \\
iii) & & = \; int(m +_N (p +_N r), n +_N (q +_N s)) \\
iv) & & = \; int(m,n) +_Z int(p +_N r, q +_N s) \\
v) & & = \; a +_Z (b +_Z c)
\end{array}$$

In the deduction step from $ii)$ to $iii)$ we use the proof that this same property is satisfied by $+_N$.

Comparing this proof with that presented above for the inductive definition of the set it seems that to formalize the theory of integers using this approach would be simpler. One can realize that most of the postulates of an integral domain are also satisfied by the set N of natural numbers. Their proofs have already been formalized in type theory and implemented in the first version of Alf [2] (see for example Szasz [17] and von Sydow [20]). Those proofs can be seen as solving the positive case of the postulates for Z and in general, once you find the way of proving the positive case the negative one is solved in a similar way. But in spite of having all this work already done we could not use it to construct the proofs for the inductive version of Z. This was a consequence of the way in which $+_Z$ and $*_Z$ were defined, that is, independently of $+_N$ and $*_N$.

However, we are not only interested in constructing the proofs of the properties that allow classifying Z as an integral domain, that is the axiomatic part of the theory of

integers. When we started with this work we also aimed at setting the basic kernel needed to develop the formalization of "computational" components of the theory, such as the division and euclidian algorithm, i.e. to construct procedures that perform those algorithms and to provide also the proof of their correctness.

As earlier mentioned we use type theory as a programming logic. The expressions of this theory are variables, canonical (constructors, data) and noncanonical(selectors, programs) elements. The engine used to implement this powerful programming language is the framework (the logical theory of types) implemented in Alf. The computations are expressed defining functions by means of the (definitional) equality of the framework. This equality is explained by saying that every constant which is an abbreviation of an expression of type α when evaluated will take as value one of the canonical values introduced for α.

When we defined the inductive version of the set we introduced its elements as 0_Z or formed from the application of the functions pos and neg to a natural number. We implicitly also defined the notion of equality on Z which can be explicitly formulated as :

$$\frac{m = n : N}{pos(m) = pos(n) : Z} \qquad \frac{m = n : N}{neg(m) = neg(n) : Z}$$

They are justified by the rules of the framework that state the reflexivity of $=$ and the extensionality of functional symbols. This notion of equality of canonical objects is structural, that is two irreducible expressions are equal if they are syntactically the same expression. Then, by the confluence and normalization properties of the framework we forced integer expressions to have as value one and only one of these canonical forms above.

When a set is defined as a quotient as presented above, we lose the notion of normal form and the equality of the elements of the set is provided from outside the system. Then, we cannot expect that the equality of two integers can be automatically checked. Suppose, for instance, that we want to prove that for all integers z, $z +_Z 0_Z =_Z z$. In the case of the inductive version of Z the proof is very simple:

$$id(Z, z) \ : \ IdZ(z +_Z 0_Z, z) \ \ [z : Z]$$

The expression $z +_Z 0_Z$ is computed to z. Thus by introduction rule of IdZ the judgement holds. The substitutivity of IdZ is then automatic. This can never be the case when dealing with the quotient version of the set. Every time we introduce a new function (property) defined over (on) the elements of the set we must also provide the proof of the congruence (substitutivity) of $\cong_{Zp}$ with respect to the function (property). Those proofs, when needed, will always appear explicitly in the proof terms, but they are computationally irrelevant. So, we will obtain constructions that when regarded as programs will contain non interesting information. Of course there are many proofs using the propositional equality IdZ where its congruence and substitutivity properties have to be made explicit. However, those properties are defined once and for all function and property defined over the elements of Z.

In the same direction, since we identify propositions with sets, when predicates ranging over $Z_p/\cong_{Zp}$ are defined in terms of the components of the elements of this set ,different propositions could be associated to equal elements of $Z_p/\cong_{Zp}$. Suppose that we define the predicate $nonneg$ as :

$$nonneg(int(m, n)) = Id(N, n, 0) \ : \ Set \ \ [m, n \ : \ N]$$

Then we could not prove that a nonnegative integer that is not normalized satisfies this predicate, even though its equivalent normalized expression does. Moreover, we could prove that the negation of the predicate holds for every nonnegative integer of that form.

One possible solution to these problems could be to extend the framework with a primitive schema for introducing quotient sets internalizing the equality of its elements. However, we do not really know what a sound formulation of that construction could be. Suppose that the following schematic rules were available in the framework:

$$\frac{A\ Set\ \ eq : (A; A)Bool}{A/eq\ Set} \qquad \frac{a : A}{a : A/eq} \qquad \frac{a, b : A\ \ eq(a, b) =_{Bool} true}{a = b : A/eq}$$

where eq is a decidable equality defined over A. The first and second rules are the formation and introduction rules of the quotient set respectively. The third rule could be read as the internalization of the equality (which indeed must be an equivalence relation) over the quotient set as an equality judgement in the theory. Then we could define the integers as a quotient set using the rules above where A is Z_p and eq is the boolean formulation of $\cong_{Zp}$. But still it would be problematic to deal with dependent sets indexed by elements of Z_p/eq. Suppose that we want to define the order relation $<_Z$ as :

$$int(m, n) <_Z int(p, q)\ =\ (m +_N q)\ <_N\ (p +_N n)\ \epsilon\ Set$$

Assume that $int(m, n) = int(p, q) : Z_p/eq$ and that $int(r, s) : Z_p/eq$, then it would be impossible to check that :

$$int(m, n) <_Z int(r, s) = int(p, q) <_Z int(r, s)\ \epsilon\ Set$$

for this entails to check that

$$(m +_N s)\ <_N\ (r +_N n)\ =\ (p +_N s)\ <_N\ (r +_N q)\ \epsilon\ Set$$

In particular, it has to be checked that $m +_N s\ =\ p +_N s\, \epsilon\, N$ which does not necessarily have to hold. That is, equality of sets in the framework is structural, then when trying to check the latter equality using the substitution rule it would be required of m and p to be the same natural number. This is not the general case due to the definition of Z_p/eq.

This example is strongly inspired by the discussion in [4] on how to deal with types depending on objects of quotient types in NuPrl. A primitive rule for defining quotient types similar to that presented above is provided in this proof-assistant. Squashed types is the solution proposed in order to deal with the problem of the dependencies described above. That means to throw away the computational content of the type and just keep the notion of inhabitance. However, is not clear for us whether this would be a sensible solution when one is interested in regarding proofs as computational constructions.

6 A concrete Integral Domain

Now, we will declare explicitly that our representation of the set of integers is an integral domain. To do that we will use the proofs of the postulates that we have developed for Z to define a substitution `IntdomZ` that fits the context `Intdom` :

```
IntdomZ  is {S := Z;R := IdZ;
             refl := idreflZ;symm := idsymmZ;trans := idtransZ;
             op := zadd;opcong := zaddcong;unit := zz;assoc := Szaddssoc;
             ident := zaddident; inv := ~;opinv := zaddinv;
             comm := zaddcommut;
             mult := ztimes;multcong := ztimescong;
             munit := one;diff := onenotzero;massoc := Sztimesassoc;
             mident := ztimunit;distlft := ztimesdistL;
             mcomm := ztimescomm;mcancel := ztimcanlft} : Intdom  []
```

The proofs terms to which the constants assigned to the variables of the context `Intdom` are associated can be found in appendix B and C.

In section 4 we listed some of the properties derived from the postulates of an integral domain for which we constructed the corresponding proof terms. Their development depends on the assumption that we can use hypothetical proofs of the postulates characterizing an integral domain as well as the properties of the equivalence relation defined over the set. We also showed in section 3.2 that we can combine substitutions with expressions of the framework in order to define new expressions. Hence, we could now obtain the particular instances of those proofs for the concrete integral domain represented by `IntdomZ` (see appendix C). It is not necessary, for instance, to construct the proof of the right distributivity law for the inductive representation of the set Z we have presented. As illustrated above, we just need to introduce the following definition :

```
distRZ = distright{IntdomZ}:(x,y,z:Z)IdZ(*(+(x,y),z),+(*(x,z),*(y,z)))
```

where $+$ and $*$ denote *zadd* and *ztimes* respectively.

Furthermore, we could also obtain the proofs of these properties for the quotient version of the set, Z_p, by defining another substitution, say `IntdomZp`. The values assigned to the variables of the context `IntDom` would be then the set Z_p, the operations and the proofs of the postulates developed for this different representation of the set Z.

Part IV
Discussion

We were interested in formalizing the arithmetics of integer numbers in Martin-Löf's type theory with the help of the proof assistant Alf. What we developed at last amounts to the formalization of the set of integers, the arithmetical operations and the properties establishing that this concrete algebra is an integral domain. On top of what has been exposed in this paper, the order relation $<$ and the modulo function have also been implemented. This constitutes the basic kernel needed to develop the theory of divisibility, the notion of prime number, the division and g.c.d. algorithms and the fundamental theorem of arithmetics.

As we have argued above, we wanted to work with an inductive definition of the set and the complete development of the proofs is done within this approach. However, to think about the definition of Z as a quotient set, to develop some basic proofs and compare both formalizations provided interesting insights about the task of formalizing mathematics in type theory. It also motivated the attempt to formulate an abstract notion of integral domain which could be used to reason about the properties satisfied by the algebra of integers independently of the chosen representation, and thereby allowed to naturally translate the results to the two different formalizations of this algebra.

To formalize the notion of what an algebraic structure is, whose components are sets and n-ary operations on those sets which satisfy specified axioms, we chose the notion of context.

The features of our approach that we have illustrated in this paper are mostly concerned with definition of abstract algebras and how to deal with derivations of properties from the postulates that characterize such algebras. We have also shown that we can formalize the notion of instantiation of those structures and derivations and how these constructions can be combined to define new ones. The way in which we defined the abstract notion of algebraic systems provides also the possibility of applying the abstract theory of particular algebraic structures to concrete examples.

However, we are aware that the approach we have presented to formalize algebraic systems has many drawbacks. Since algebra is often considered as the study of the properties of algebras which are invariant under isomorphism (*algebraic properties*), we should be able to reason about constructions like morphisms between structures, homomorphic images and quotient structures. The formalization of some of these constructions above using contexts can be really unpleasant, and some others cannot even be defined.

Suppose we want to define the notion of morphism between groups:

$$
\begin{array}{l}
Group_X \ is\ [X : Set; R_X : (X; X)Set; \ldots; op_X : (X; X)X; \ldots] \\
Group_Y \ is\ [Y : Set; R_Y : (Y; Y)Set; \ldots; op_Y : (Y; Y)Y; \ldots] \\
Morph_{XY} \ is\ Group_X + Group_Y + [f : (X)Y; (x, y : X)R_Y(f(op_X(x, y)), op_Y(f(x), f(y)))]
\end{array}
$$

where $Group_X$ and $Group_Y$ are defined as groups as shown above.
Then, first we note that we must introduce $Group_X$ and $Group_Y$ as abbreviations of two "different" contexts, that difference being only the names of the variables. This is a clear consequence of our choice of regarding algebraic systems as list of named hypotheses. We are not thinking of those systems as classifying collections of mathematical objects. Hence,

it is not possible to assume variables ranging over them. This also entails that we cannot even define a function (or better a functor) which given an algebraic structure returns a structure formed by components of the former (for instance, one that given an abelian group yields the corresponding group). Besides, we must be careful with the naming of the variables used in the third context (X, Y, op_X and op_Y in this case) in order to make sure that we are expressing the property we want over the components of $Group_X$ and $Group_Y$ we mean to.

So, we are searching here for the possibility of formalizing algebraic structures in type theory as entities describing mathematical constructions in such a way that we can refer to and define operations on those constructions. But, what kind of entities? Should they be defined as sets or as types? By type here we mean a category (in the sense of [12]) whose definition does not require to know how its objects are constructed but just to grasp what it means to be an object of such category. In this latter case, is the theory of types (the framework implemented in Alf) powerful or suitable enough to express the fundamental algebraic constructions?

Along these last years some works dealing with formalization of algebraic concepts (directly or not) have been presented. To our knowledge the solutions have always followed the approach of using either Σ-sets or Σ-types in the case of working with constructive type theories (see for instance [14], [4], [7], [1],) or predicates involving logical constants and predefined equalities as presented in [6].

Algebraic structures as sets

Since the early '80s the formalization of abstract data types has been an important topic of discussion in the programming language area. In Mitchell [13] the authors present a functional language which incorporates existential types of the form $\exists t.\sigma(t)$, where t is a type variable which may occur free in the type expression $\sigma(t)$. Values of such types are also introduced and are intended to model abstract data types. They are called *data algebras*. The intuitionistic explanation of the existential quantifier and what it means to be a proof of an existential proposition together with the Curry-Howard correspondence of propositions with types lies behind this formalization of abstract data types.

In Nordström et al. [14] a similar approach is used in order to propose a methodology of module specification in type theory. A module is there understood as a tuple $\langle A_1, A_2, \ldots, A_n \rangle$, where some A_i are sets and some are elements and functions defined on these sets. An example of the application of these notions to formalize algebra could be the definition of group as the tuple:

$$\langle M, *, u, inv, P_{ass}, P_{unit}, P_{inv} \rangle$$

where M is a set, $* \in M \times M \to M$,$u \in M \to M$ and P_{ass},P_{unit} and P_{inv} express the group properties.

The natural way of expressing specification of modules in type theory is using general sums, written $\Sigma x \in A.B(x)$, which corresponds to the disjoint union of the family of sets $B(x)$ with x ranging over the set A. The elements of this kind of sets are pairs where the first component determines the type of the second. We refer to Martin-Löf [12] for theoretical explanations.

So, going back to the example above, let us look now how the notion of group could be defined in terms of Σ-sets:

$$
\begin{array}{l}
(\Sigma M \in U) \\
\quad (\Sigma * \in M \times M \to M) \\
\quad (\Sigma u \in M) \\
\quad (\Sigma inv \in M \to M) \\
\qquad (\Pi x, y, z \in M) \\
\qquad\quad [*(x, *(y, z)) =_M *(*(x, y), z)] \times \\
\qquad\quad [*(x, u) =_M x] \times \\
\qquad\quad [*(x, inv(x)) =_M u]
\end{array}
$$

where U is the name for the set of the small sets (the first universe). Since in type theory the sets have to be inductively defined, it is impossible to have the notion of the set of all sets. But, as in our case, it is usually necessary to talk about sets whose elements (or their components) are sets. In order to deal with this situation, the notion of universe, intended as the least set closed under specific set forming operations, is introduced. This process could be iterated, obtaining then a hierarchical sequence of universes $U = U_0 \in U_1 \in \ldots \in U_n \in \ldots$.

Now, turning back to the case which motivated this discussion the notion of morphism between two groups could be defined as:

$$
\begin{array}{rcl}
Morphism & \equiv & (\Sigma X \in Group)(\Sigma Y \in Group) \\
 & & (\Sigma f \in (X_{set})Y_{set})(\Pi x, y \in X_{set})(f(*_X(x, y)) =_Y *_Y(f(x), f(y)))
\end{array}
$$

because now Group is a set (which belongs to the universe U_1) and we can assume variables ranging over that set. The operations $X_{set}, Y_{set}, *_X, *_Y$ and $=_Y$ are defined in terms of the projection functions. The functor we mentioned above taking abelian groups to their corresponding groups could also be defined without problems. Moreover, we could define a special universe, say U_{eq}, as:

$$
U_{eq} \equiv (\Sigma X \in U)(\Sigma eq \in (X, X)U)(Ref_{eq} \times Symm_{eq} \times Trans_{eq})
$$

which would be the universe of the sets with an equivalence relation defined over them.

So, it seems that, if one wants to grasp algebraic structures as sets, their formalization using Σ-sets and universes forms an expressive and suitable alternative.

However, in the mechanical formalization it could be hard to deal with the codification. The set of natural numbers, for instance, would be represented by two different symbols: N representing the set itself and N_U the element of the universe U. That is, in order to work with universes we are compelled to deal with the codes we use to reflect the set structure at the level of objects and the corresponding decodification functions. And this is rather cumbersome.

Another consequence of working with universes is that the way of forming sets remains fixed because the canonical members of the universes are codings of a fixed number of set constructing operations. Nevertheless, is not at all clear, at least for us, that this presupposes a serious restriction when trying to formalize algebraic notions.

Algebraic structures as types

As an alternative to the approach presented above, we could try to grasp algebraic structures as types. The immediate question is then... what kind of types are they?

In Luo [7] and MacQueen [9] a higher order calculus (ΣCC) and a language with ramified dependent types (DL) are presented. The language includes a Σ-type constructor together with the corresponding projection operations. In both works the adequacy of Σ-types as a basic mechanism to express abstract structures is analysed as well as a safe methodology to deal with the manipulation of independently developed theories. Moreover, in [7] the power of the calculus to express mathematical problems is illustrated with many examples of formalization of algebraic notions. All the limitations presented by the contexts can be solved using this approach in a similar way as done with Σ-sets above. But the restriction that the domains of quantification must be closed does not appear. In such systems the notion of set remains open.

This same understanding of sets is proposed by Martin-Löf in the formulation of his logical framework. A primitive type *Set* is introduced. It classifies inductively defined sets but there is no longer the requirement of providing all the possible ways of forming those sets. However, in order to define Σ structures whose domains are types (the type *Set*, for instance) it would be necessary to reflect this notion at the level of types, too. One (among many) possible way of achieving this could be extending the type level with rules for *dependent pairs*:

$$\frac{\alpha\ type \quad \beta\ type \quad [x:\alpha]}{(x:\alpha;\beta)\ type} \qquad \frac{\alpha\ type \quad \beta\ type \quad [x:\alpha] \quad a:\alpha \quad b:\beta(a)}{(a,b):(x:\alpha;\beta)}$$

$$\frac{p:(x:\alpha;\beta)}{\pi_1(p):\alpha} \qquad \frac{p:(x:\alpha;\beta)}{\pi_2(p):\beta(\pi_1(p))}$$

The two first rules introduce the type and object constructors for dependent pairs respectively and π_1 and π_2 are the corresponding projections. But it is not clear for us whether it is really necessary to abandon the setting provided by the theory of sets, where it seems possible to formalize - even with some apparent restrictions - most of the basic algebraic notions.

In another direction, we have also the doubt whether Σ(dependent pairs)-types are the best tool to express algebraic notions. In a recent paper (Tasistro [18]) an extension to Martin-Löf's logical framework has been presented where the type level is enriched with labelled record types. After having done some examples using this kind of types to formalize algebraic notions we think that there are some features of this approach that seem to be useful when formalizing algebra. We will illustrate this point with the following examples. Suppose first that we define monoid as the following record type:

$$monoid \equiv \langle X : Set; R : (X;X)Set, \ldots, op : (X;X)X, \ldots, pri : P_{id}, pra : P_{ass} \rangle$$

where $\langle \ldots \rangle$ is the notation for the record types constructor. Now we could define group and abelian monoid as the extensions:

$$\begin{aligned} group &\equiv \langle M : monoid; inv : (X)X; prinv : P_{inv} \rangle \\ abmonoid &\equiv \langle M : monoid; pco : P_{comm} \rangle \end{aligned}$$

to finally define two "different" formalizations of abelian group:

$$\begin{array}{lcl} abgroup_1 & \equiv & \langle G : group; pco : P_{comm} \rangle \\ abgroup_2 & \equiv & \langle Am : abmonoid; inv : (X)X; prinv : P_{inv} \rangle \end{array}$$

It can be noted that $abgroup_1$ and $abgroup_2$ only differ in the order of their components. The explanation proposed in [18] of what it means to be a record type allows to justify a subtyping relation between records. This relation could be used for instance to show that these two definitions above are equivalent. Furthermore, it is rather simple to prove that every record object of the form $abgroup_1$ (resp. $abgroup_2$) it is also an object of the form *group* (resp. *abmonoid*) and *monoid*. Moreover, it could also be proved that given a record object of the form $abgroup_1$ (which extends *group*) it is also an object of the form *abmonoid*. Similarly for an object of the form $abgroup_2$ (which extends *abmonoid*) with respect to *group*. So, relations between algebraic structures are naturally formalized by their definitions as records extending other ones. This is a clear difference with the contexts and Σ approaches. In the former there is no way of relating independently defined contexts, in the latter the order of the components is in the very nature of the explanation of the type.

Another interesting feature of records arises in the following case: As shown above with contexts we could define a proof, say $cancel_{rec} : group \to P_{cancel}$, that for every group the cancellation law holds. Now, let us suppose that we want to construct a new proof, say $agpr : abgroup \to P_{agpr}$, which uses $cancel_{rec}$. By the subtyping relation for the function types it can be inferred that $cancel_{rec}$ has also type $abgroup \to P_{cancel}$. Then it can be applied directly to the argument of *agpr* in the body of the definition. This kind of inclusion polymorphism is very useful when one thinks of an incremental development of algebraic structures. All that has been proved for one structure can naturally be applied for those extending it. This can also be reflected using the context approach. Every proof constructed under certain context Γ, say, can be used in a proof developed under a context Δ which extends Γ. However using Σ-types(sets), in the example above for instance, we should first apply the forgetful functor from abelian groups to groups that we have earlier mentioned in order to be able to use $cancel_{rec}$.

Nevertheless, we do not claim that these advantages could enforce the decision of chosing record types to formalize algebra. The same questions as for Σ-types hold. We think that, at least for us, it is necessary to better understand which is the appropriate theoretical setting to formalize algebra in type theory.

Other works with integers in proof-assistants

The type of integers *int* is built into NuPRL (Constable et al. [4]). The canonical elements of this type and the operations $+, -, *$ are also built into the system. There is a noncanonical form associated to this type which provides a mechanism for definition and proof by induction to which the recursive operators presented in this paper are very close. The properties of the operations are provided by the system. We do not know works on formalization of algebraic notions using this system, but in [4] Σ-types are suggested as a possible methodology to represent algebraic structures.

Valerie Menissier, at INRIA, has formalized the set Z together with the proofs that it is an integral domain in Coq. She has the set inductively defined and uses inductive reasoning to construct the proofs of the properties. There is also a great coincidence with

our work in the sequence and kind of lemmas needed in the development of the main proofs.

Integers have been formalized in LEGO, a proof assistant developed at the LCF of Edinburgh, as equivalence classes of pairs of natural numbers. A work in progress on formalization of Galois theory has been presented in Aczel [1]. There, some basic algebraic notions (sets, mappings and algebraic structures) are formalized in terms of Σ-types using the implementation of Luo's extended calculus of constructions provided by LEGO.

Another formalization of the integers as equivalence classes of pairs of naturals but using HOL is presented in Gunter [6]. There, it is also presented a method for representing algebraic notions which is strongly based in the use of higher order predicates to describe the structures as well as constructions over them. The basic logic of HOL is an extended version of the simple theory of types, where propositions are interpreted classically.

Acknowledgements

First, I would like to warmly thank my supervisor Björn von Sydow. He spent many hours of his time on discussions concerned with this work, and on reading and commenting on different versions of this paper.

I am greatly indebted to Alvaro Tasistro and Nora Szasz, for careful reading of this paper, many interesting discussions and their permanent support.

I also received helpful comments from Bengt Nordström, Kent Petersson and Jan Smith on an earlier version of this paper and from John Hughes who read a draft version of the part on integers.

Finally, I want to thank my former supervisor Juan José Cabezas for all his support and encouragement along all these years.

References

[1] P. Aczel. Galois : A Theory Development Project. A report on work in progress for the Turin meeting on the representation of mathematics in Logical frameworks, 1993.

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

[3] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra.* Macmillan, 1953.

[4] R. Constable et al. *Implementing mathematics with the Nuprl development sys tem.* Prentice-Hall, 1986.

[5] Th. Coquand. Pattern Matching with Dependent Types. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[6] E. Gunter. Doing Algebra in Simple Type Theory. Unpublished paper provided by the HOL system installation.

[7] Z. Luo. A Higher-order Calculus and Theory of Abstractions. Technical report, LFCS, Department of Computer Science, University of Edinbu rgh, 1988.

[8] S. MacLane and G. Birkhoff. *Algebra.* MacMillan, 1967.

[9] D. MacQueen. Using Dependent Types to Express Modular Structures. In *Proceedings of the 13th POPL*, 1986.

[10] L. Magnusson. The new implementation of Alf. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[11] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[12] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[13] John Mitchell and Gordon Plotkin. Abstract types have existential type. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, New York, 1985.

[14] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.An Introduction.* Oxford University Press, 1989.

[15] J. M. Smith. Propositional Functions and Families of Types. In *Workshop on Programming Logic*, pages 140–159, 1989.

[16] N. Szasz. Are Integer Numbers with addition a Group? Winter Meeting 1992.Department of Computer Science. Chalmers University of Technology and University of Göteborg.

[17] N. Szasz. A Machine Checked Proof that Ackermann's Function is not Primitive Recursive. In G. Huet and G. Plotkin, editors, *Proceedings of the Second Workshop on Logical Frameworks*. Esprit Basic Research Action 3245, Cambridge University Press, 1992.

[18] A. Tasistro. Extension of Martin-Löf's Theory of Types with Record Types and Subtyping, 1993. Privately circulated notes.

[19] A. Tasistro. Formulation of Martin-Löf's theory of types with explicit substitution, 1993. Licenciate thesis.Programming Methodology Group, Dept. of Computer Science, University of Göteborg and Chalmers University of Technology.

[20] B. von Sydow. A Machine-assisted Proof of the Fundamental Theorem of Arithmetic. Technical report, Programming Methodology Group,Dept. of Computer Science, University of Göteborg and Chalmers University of Technology, 1992.

Part V
Appendices

The following appendices contain the complete definitions and proofs to which we have referred in the paper.

Appendix A includes the incremental definition of the algebraic structures from SET to Integral Domain and the proofs of the derived properties following from the postulates of the latter. In appendix B all the definitions, lemmas and proofs required to prove that the set Z of integers is an integral domain are listed. The definition of the substitution fitting the context `Intdom` and the instantiation of the derived properties for Z are listed in appendix C.

All the proofs have been developed using the formalization of Martin-Löf's set theory provided by the Alf library. For the proofs of the integers we also used the formalization of the set N of natural numbers with its corresponding properties.

Appart from the definitions of contexts and substitutions, three different forms of definitions will be found :

1. $c : \alpha\ [\Gamma]$ `C`
2. $c : \alpha\ [\Gamma]$ `I`
3. $c = e : \alpha\ [\Gamma]$

The letter `C` at the end of a definition indicates that c is introduced as a canonical constant. With the letter `I` it is indicated that c is an implicit constant defined using pattern matching over (some) of its arguments. Following such definition, the equations associated to each case of pattern are stated. In the third case, c is just an abbreviation (explicit constant).

A Formalization of integral domain and derived properties

```
SET       is             [S:Set; R:(S;S)Set;
                          refl:(x:S)R(x,x);
                          symm:(x:S;y:S;R(x,y))R(y,x);
                          trans:(x:S;y:S;z:S;R(x,y);R(y,z))R(x,z)]


Grupoid   is   SET   +   [op:(S;S)S;
                          opcong:(x:S;y:S;z:S;w:S;R(x,y);R(z,w))
                                                  R(op(x,z),op(y,w))]

Semigroup is  Groupoid + [assoc:(x:S;y:S;z:S)R(op(x,op(y,z)),
                                               op(op(x,y),z))]


Monoid    is   Semigroup   +   [unit:S;
                                ident:(x:S)and(R(op(x,unit),x),
                                               R(op(unit,x),x))]

Group     is   Monoid    +    [inv:(S)S;
                                opinv:(x:S)and(R(op(x,inv(x)),unit),
                                               R(op(inv(x),x),unit))]


AbGroup   is   Group   +   [comm:(x:S;y:S)R(op(x,y),op(y,x))]


Ring      is   AbGroup   +   [mult:(S;S)S;
                              multcong:(x,y,z,w:S;R(x,y);R(z,w))
                                                 R(mult(x,z),mult(y,w));
                              munit:S; diff:not(R(munit,unit));
                              massoc:(x,y,z:S)R(mult(x,mult(y,z)),
                                                mult(mult(x,y),z));
                              mident:(x:S)R(mult(x,munit),x);
                              distlft:(x,y,z:S)R(mult(x,op(y,z)),
                                                 op(mult(x,y),mult(x,z)))]


CommRing  is   Ring   +   [mcomm:(x,y:S)R(mult(x,y),mult(y,x))]


Intdom    is   CommRing   +   [mcancel:(x,y,z:S;not(R(z,unit));
                                               R(mult(x,z),mult(y,z)))R(x,y)]
```

These are the proof terms of some of the properties derived from the postulates of an integral domain that are presented in [3].

Unicity of the additive inverse.

```
uniqinv = [a,b,c,h,h1]
           cancelleft(a,b,c,
                      trans(op(a,b),
                            unit,
                            op(a,c),
                            h,
                            symm(op(a,c),unit,h1))
                      ) : (a:S;b:S;c:S;
                           R(op(a,b),unit);
                           R(op(a,c),unit))
                                     R(b,c)     Intdom
```

Unicity of unit for op.

```
uniqunit = [a,p]
           trans(a,
                 op(unit,a),
                 unit,
                 symm(op(unit,a),a,snd(R(op(a,unit),a),
                                       R(op(unit,a),a),ident(a))),
                 p(unit)
                ) : (a:S;p:(x:S)R(op(x,a),x))R(a,unit)     Intdom
```

Non-zero divisor.

```
nzdiv = [a,b,h1,h3]
        mcancel(b,unit,a,h3,
                trans(mult(b,a),
                      mult(a,b),
                      mult(unit,a),
                      mcomm(b,a),
                      trans(mult(a,b),
                            unit,
                            mult(unit,a),
                            h1,
                            trans(unit,
                                  mult(a,unit),
                                  mult(unit,a),
                                  symm(mult(a,unit),unit,timeszero(a)),
                                  mcomm(a,unit))))
                ) : (a:S;b:S;
                     R(mult(a,b),unit);
                     not(R(a,unit)))
                           R(b,unit)  Intdom
```

Zero of multiplication.

```
timeszero = [x]
            cancelleft(mult(x,x),
                       mult(x,unit),
                       unit,
                       trans(op(mult(x,x),mult(x,unit)),
                             mult(x,op(x,unit)),
                             op(mult(x,x),unit),
                             symm(mult(x,op(x,unit)),
                                  op(mult(x,x),mult(x,unit)),
                                  distlft(x,x,unit)),
                             trans(mult(x,op(x,unit)),
                                   mult(x,x),
                                   op(mult(x,x),unit),
                                   multcong(x,x,op(x,unit),x,
                                            refl(x),
                                            fst(R(op(x,unit),x),
                                                R(op(unit,x),x),
                                                ident(x))),
                                   symm(op(mult(x,x),unit),
                                        mult(x,x),
                                        fst(R(op(mult(x,x),unit),
                                              mult(x,x)),
                                            R(op(unit,mult(x,x)),
                                              mult(x,x)),
                                            ident(mult(x,x))))))
                     ) : (x:S)R(mult(x,unit),unit)     Intdom
```

Right distributivity.

```
distr = [x,y,z]
        trans(mult(op(x,y),z),
                   mult(z,op(x,y)),
                   op(mult(x,z),mult(y,z)),
                   mcomm(op(x,y),z),
                   trans(mult(z,op(x,y)),
                         op(mult(z,x),mult(z,y)),
                         op(mult(x,z),mult(y,z)),
                         distlft(z,x,y),
                         cong(mult(z,x),
                              mult(x,z),
                              mult(z,y),
                              mult(y,z),
                              mcomm(z,x),
                              mcomm(z,y)
                             )
                         )
             ) : (x:S;y:S;z:S)
                 R(mult(op(x,y),z),op(mult(x,z),mult(y,z))) Intdom
```

B Integers

Definition of the set Z

```
Z : Set     [] C

  zz : Z     [] C
  pos : (n:N)Z     [] C
  neg : (n:N)Z     [] C

one = pos(0) : Z []

Zcases : (C:(Z)Set;a:C(zz);
          b:(n:N)C(pos(n));
          c:(n:N)C(neg(n));
          z:Z)C(z)     [] I

  Zcases(C,a,b,c,zz) = a
  Zcases(C,a,b,c,pos(n)) = b(n)
  Zcases(C,a,b,c,neg(n)) = c(n)

zs : (Z)Z     [] I

  zs(zz) = pos(0)
  zs(pos(n)) = pos(s(n))
  zs(neg(0)) = zz
  zs(neg(s(h))) = neg(h)

zp : (Z)Z     [] I

  zp(zz) = neg(0)
  zp(pos(0)) = zz
  zp(pos(s(h))) = pos(h)
  zp(neg(n)) = neg(s(n))
```

Propositional equality (IdZ)

```
IdZ = Id(Z) : (x:Z;y:Z)Set     []

Idzs = [x,y,p]
       idcongr(Z,Z,zs,x,y,p) : (x:Z;y:Z;p:IdZ(x,y))IdZ(zs(x),zs(y))     []

Idzp = [x,y,p]
       idcongr(Z,Z,zp,x,y,p) : (x:Z;y:Z;p:IdZ(x,y))IdZ(zp(x),zp(y))     []

multsubs = [x,y,z,w,h,h1,h2]
           idtrans(Z,z,y,w,
                   idtrans(Z,z,x,y,idsymm(Z,x,z,h1),h),
                   h2) : (x:Z;y:Z;z:Z;w:Z;IdZ(x,y);
                          IdZ(x,z);IdZ(y,w))IdZ(z,w)     []
```

Discrimination of canonical elements

```
Isnotzz : (x:Z)Set [] I

  Isnotzz(zz) = Empty
  Isnotzz(pos(n)) = N1
  Isnotzz(neg(n)) = N1

posnotzz : (n:N;IdZ(pos(n),zz))Empty     [] I

  posnotzz(n,h) = case h of
  end

negnotzz : (n:N;IdZ(neg(n),zz))Empty     [] I

  negnotzz(n,h) = case h of
  end
```

Properties relating successor and predeccessor (zs and zp)

```
invzp : (z:Z)IdZ(zs(zp(z)),z)     [] I

  invzp(zz) = id(Z,zz)
  invzp(pos(0)) = id(Z,pos(0))
  invzp(pos(s(h))) = id(Z,pos(s(h)))
  invzp(neg(0)) = id(Z,neg(0))
  invzp(neg(s(h))) = id(Z,neg(s(h)))

invzs : (z:Z)IdZ(zp(zs(z)),z)     [] I

  invzs(zz) = id(Z,zz)
  invzs(pos(0)) = id(Z,pos(0))
  invzs(pos(s(h))) = id(Z,pos(s(h)))
  invzs(neg(0)) = id(Z,neg(0))
  invzs(neg(s(h))) = id(Z,neg(s(h)))

invzsp = [x]idtrans(Z,
                    zs(zp(x)),
                    x,
                    zp(zs(x)),
                    invzp(x),
                    idsymm(Z,
                           zp(zs(x)),
                           x,
                           invzs(x))) : (x:Z)IdZ(zs(zp(x)),
                                                 zp(zs(x)))     []
```

Properties of addition (zadd)

```
zadd : (Z;Z)Z     [] I

  zadd(zz,h1) = h1
  zadd(pos(0),h1) = zs(h1)
  zadd(pos(s(h)),h1) = zs(zadd(pos(h),h1))
  zadd(neg(0),h1) = zp(h1)
  zadd(neg(s(h)),h1) = zp(zadd(neg(h),h1))

zaddzs : (x:Z;y:Z)IdZ(zs(zadd(x,y)),zadd(zs(x),y))     [] I

  zaddzs(zz,y) = id(Z,zs(y))
  zaddzs(pos(n),y) = id(Z,zadd(zs(pos(n)),y))
  zaddzs(neg(0),y) = idtrans(Z,
                             zs(zadd(neg(0),y)),
                             y,
                             zadd(zs(neg(0)),y),
                             invzp(y),
                             id(Z,y))
  zaddzs(neg(s(h)),y) = idtrans(Z,
                                zs(zadd(neg(s(h)),y)),
                                zadd(neg(h),y),
                                zadd(zs(neg(s(h))),y),
                                invzp(zadd(neg(h),y)),
                                id(Z,zadd(neg(h),y)))

zaddzp : (x:Z;y:Z)IdZ(zp(zadd(x,y)),zadd(zp(x),y))     [] I

  zaddzp(zz,y) = id(Z,zp(y))
  zaddzp(pos(0),y) = idtrans(Z,zp(zs(y)),y,y,invzs(y),id(Z,y))
  zaddzp(pos(s(h)),y) = idtrans(Z,
                                zp(zadd(pos(s(h)),y)),
                                zadd(pos(h),y),zadd(zp(pos(s(h))),y),
                                invzs(zadd(pos(h),y)),
                                id(Z,zadd(pos(h),y)))
  zaddzp(neg(n),y) = id(Z,zp(zadd(neg(n),y)))

zadd0 : (x:Z)IdZ(zadd(x,zz),x)     [] I
  zadd0(zz) = id(Z,zz)
  zadd0(pos(0)) = id(Z,pos(0))
  zadd0(pos(s(h))) = Idzs(zadd(pos(h),zz),pos(h),zadd0(pos(h)))
  zadd0(neg(0)) = id(Z,neg(0))
  zadd0(neg(s(h))) = Idzp(zadd(neg(h),zz),neg(h),zadd0(neg(h)))

zaddsubstL = [a,b,y,p]idcongr(Z,Z,[h]zadd(h,y),a,b,p
                              ) : (a:Z;b:Z;y:Z;p:IdZ(a,b))
                                      IdZ(zadd(a,y),zadd(b,y))     []

zaddsubstR = [a,b,x,p]idcongr(Z,Z,[h]zadd(x,h),a,b,p
                              ) : (a:Z;b:Z;x:Z;p:IdZ(a,b))
                                      IdZ(zadd(x,a),zadd(x,b))     []
```

```
zaddcong = [x,y,z,w,p,q]idtrans(Z,zadd(x,z),zadd(y,z),zadd(y,w),
                               zaddsubstL(x,y,z,p),
                               zaddsubstR(z,w,y,q))
   : (x:Z;y:Z;z:Z;w:Z;p:IdZ(x,y);q:IdZ(z,w))IdZ(zadd(x,z),zadd(y,w))     []

zaddpos : (m:N;n:N)IdZ(zadd(pos(m),pos(n)),pos(s(plus(m,n))))     [] I

  zaddpos(0,n) = id(Z,pos(s(n)))
  zaddpos(s(n1),n) = Idzs(zadd(pos(n1),pos(n)),pos(s(plus(n1,n))),
                          zaddpos(n1,n))

zaddneg : (m:N;n:N)IdZ(zadd(neg(m),neg(n)),neg(s(plus(m,n))))     [] I

  zaddneg(0,n) = id(Z,neg(s(n)))
  zaddneg(s(n1),n) = idtrans(Z,
                             zadd(neg(s(n1)),neg(n)),
                             zp(neg(s(plus(n1,n)))),
                             neg(s(plus(s(n1),n))),
                             Idzp(zadd(neg(n1),neg(n)),neg(s(plus(n1,n))),
                                  zaddneg(n1,n)),id(Z,neg(s(plus(s(n1),n)))))

zaddssoc : (x:Z;y:Z;z:Z)IdZ(zadd(zadd(x,y),z),zadd(x,zadd(y,z)))     [] I

  zaddssoc(zz,y,z) = id(Z,zadd(y,z))
  zaddssoc(pos(0),y,z) = idsymm(Z,zadd(pos(0),zadd(y,z)),
                                zadd(zadd(pos(0),y),z),zaddzs(y,z))
  zaddssoc(pos(s(h)),y,z) = idtrans(Z,
                                    zadd(zadd(pos(s(h)),y),z),
                                    zs(zadd(zadd(pos(h),y),z)),
                                    zadd(pos(s(h)),zadd(y,z)),
                                    idsymm(Z,
                                           zs(zadd(zadd(pos(h),y),z)),
                                           zadd(zadd(pos(s(h)),y),z),
                                           zaddzs(zadd(pos(h),y),z)),
                                           Idzs(zadd(zadd(pos(h),y),z),
                                                zadd(pos(h),zadd(y,z)),
                                                zaddssoc(pos(h),y,z)))

  zaddssoc(neg(0),y,z) = idsymm(Z,zadd(neg(0),zadd(y,z)),
                                zadd(zadd(neg(0),y),z),zaddzp(y,z))
  zaddssoc(neg(s(h)),y,z) = idtrans(Z,
                                    zadd(zadd(neg(s(h)),y),z),
                                    zp(zadd(zadd(neg(h),y),z)),
                                    zadd(neg(s(h)),zadd(y,z)),
                                    idsymm(Z,
                                           zp(zadd(zadd(neg(h),y),z)),
                                           zadd(zadd(neg(s(h)),y),z),
                                    zaddzp(zadd(neg(h),y),z)),
                                    Idzp(zadd(zadd(neg(h),y),z),
                                         zadd(neg(h),zadd(y,z)),
                                         zaddssoc(neg(h),y,z)))
```

```
 Szaddssoc = [a,b1,c]idsymm(Z,zadd(zadd(a,b1),c),zadd(a,zadd(b1,c)),
                           zaddssoc(a,b1,c)
                          ) : (a:Z;b:Z;c:Z)IdZ(zadd(a,zadd(b,c)),
                                               zadd(zadd(a,b),c))     []

zsaddsnd : (x:Z;y:Z)IdZ(zs(zadd(x,y)),zadd(x,zs(y)))     [] I

  zsaddsnd(zz,y) = id(Z,zs(y))
  zsaddsnd(pos(0),y) = id(Z,zs(zs(y)))
  zsaddsnd(pos(s(h)),y) = Idzs(zs(zadd(pos(h),y)),zadd(pos(h),zs(y)),
                               zsaddsnd(pos(h),y))
  zsaddsnd(neg(0),y) = invzsp(y)
  zsaddsnd(neg(s(h)),y) = idtrans(Z,
                                  zs(zadd(neg(s(h)),y)),
                                  zp(zs(zadd(neg(h),y))),
                                  zadd(neg(s(h)),zs(y)),
                                  invzsp(zadd(neg(h),y)),
                                  Idzp(zs(zadd(neg(h),y)),
                                       zadd(neg(h),zs(y)),
                                       zsaddsnd(neg(h),y)))

zpaddsnd : (x:Z;y:Z)IdZ(zp(zadd(x,y)),zadd(x,zp(y)))     [] I

  zpaddsnd(zz,y) = id(Z,zp(y))
  zpaddsnd(pos(0),y) = idsymm(Z,zadd(pos(0),zp(y)),zp(zadd(pos(0),y)),
                              invzsp(y))
  zpaddsnd(pos(s(h)),y) = idtrans(Z,
                                  zp(zadd(pos(s(h)),y)),
                                  zs(zp(zadd(pos(h),y))),
                                  zadd(pos(s(h)),zp(y)),
                                  idsymm(Z,
                                         zs(zp(zadd(pos(h),y))),
                                         zp(zadd(pos(s(h)),y)),
                                         invzsp(zadd(pos(h),y))),
                                  Idzs(zp(zadd(pos(h),y)),zadd(pos(h),zp(y)),
                                       zpaddsnd(pos(h),y)))
  zpaddsnd(neg(0),y) = id(Z,zp(zp(y)))
  zpaddsnd(neg(s(h)),y) = Idzp(zp(zadd(neg(h),y)),zadd(neg(h),zp(y)),
                               zpaddsnd(neg(h),y))

zaddcommut : (x:Z;y:Z)IdZ(zadd(x,y),zadd(y,x))     [] I

  zaddcommut(zz,y) = idtrans(Z,
                             zadd(zz,y),
                             y,
                             zadd(y,zz),
                             id(Z,y),
                             idsymm(Z,zadd(y,zz),y,zadd0(y)))
```

```
  zaddcommut(pos(0),y) = idtrans(Z,
                                 zadd(pos(0),y),
                                 zs(zadd(y,zz)),
                                 zadd(y,pos(0)),
                                 Idzs(y,
                                      zadd(y,zz),
                                      idsymm(Z,zadd(y,zz),y,zadd0(y))),
                                 zsaddsnd(y,zz))
  zaddcommut(pos(s(h)),y) = idtrans(Z,
                                    zadd(pos(s(h)),y),
                                    zs(zadd(y,pos(h))),
                                    zadd(y,pos(s(h))),
                                    Idzs(zadd(pos(h),y),
                                         zadd(y,pos(h)),
                                         zaddcommut(pos(h),y)),
                                    zsaddsnd(y,pos(h)))
  zaddcommut(neg(0),y) = idtrans(Z,
                                 zadd(neg(0),y),
                                 zp(zadd(y,zz)),
                                 zadd(y,neg(0)),
                                 Idzp(y,
                                      zadd(y,zz),
                                      idsymm(Z,zadd(y,zz),y,zadd0(y))),
                                 zpaddsnd(y,zz))
  zaddcommut(neg(s(h)),y) = idtrans(Z,
                                    zadd(neg(s(h)),y),
                                    zp(zadd(y,neg(h))),
                                    zadd(y,neg(s(h))),
                                    Idzp(zadd(neg(h),y),
                                         zadd(y,neg(h)),
                                         zaddcommut(neg(h),y)),
                                    zpaddsnd(y,neg(h)))
```

Properties of the inverse function

```
~ : (Z)Z     [] I

  ~(zz) = zz
  ~(pos(n)) = neg(n)
  ~(neg(n)) = pos(n)

zs~ : (x:Z)IdZ(~(zs(x)),zp(~(x)))     [] I

  zs~(zz) = id(Z,neg(0))
  zs~(pos(n)) = id(Z,neg(s(n)))
  zs~(neg(0)) = id(Z,zz)
  zs~(neg(s(h))) = id(Z,pos(h))

Szs~(x) = [x]idsymm(Z,~(zs(x)),zp(~(x)),zs~(x)
                    ): (x:Z)IdZ(zp(~(x)),~(zs(x)))     []

zp~ : (x:Z)IdZ(~(zp(x)),zs(~(x)))     [] I
```

```
  zp~(zz) = id(Z,pos(0))
  zp~(pos(0)) = id(Z,zz)
  zp~(pos(s(h))) = id(Z,neg(h))
  zp~(neg(n)) = id(Z,pos(s(n)))

Szp~(x) = idsymm(Z,~(zp(x)),zs(~(x)),zp~(x)
               ) :(x:Z)IdZ(zs(~(x)),~(zp(x))) []

idem~ : (x:Z)IdZ(~(~(x)),x)     [] I

  idem~(zz) = id(Z,zz)
  idem~(pos(n)) = id(Z,pos(n))
  idem~(neg(n)) = id(Z,neg(n))

Sidem~ = [x]idsymm(Z,~(~(x)),x,idem~(x)) : (x:Z)IdZ(x,~(~(x)))     []

distadd~ : (x:Z;y:Z)IdZ(~(zadd(x,y)),zadd(~(x),~(y)))     [] I

  distadd~(zz,y) = id(Z,~(y))
  distadd~(pos(0),y) = zs~(y)
  distadd~(pos(s(n1)),y) = idtrans(Z,
                                   ~(zadd(pos(s(n1)),y)),
                                   zp(zadd(neg(n1),~(y))),
                                   zadd(~(pos(s(n1))),~(y)),
                                   idtrans(Z,
                                           ~(zadd(pos(s(n1)),y)),
                                           zp(~(zadd(pos(n1),y))),
                                           zp(zadd(neg(n1),~(y))),
                                           zs~(zadd(pos(n1),y)),
                                           Idzp(~(zadd(pos(n1),y)),
                                                zadd(neg(n1),~(y)),
                                                distadd~(pos(n1),y))),
                                   idtrans(Z,
                                           zp(zadd(neg(n1),~(y))),
                                           zadd(zp(neg(n1)),~(y)),
                                           zadd(~(pos(s(n1))),~(y)),
                                           zaddzp(neg(n1),~(y)),
                                           id(Z,zadd(~(pos(s(n1))),~(y)))))
  distadd~(neg(0),y) = zp~(y)
  distadd~(neg(s(n1)),y) = idtrans(Z,
                                   ~(zadd(neg(s(n1)),y)),
                                   zs(zadd(pos(n1),~(y))),
                                   zadd(~(neg(s(n1))),~(y)),
                                   idtrans(Z,
                                           ~(zadd(neg(s(n1)),y)),
                                           zs(~(zadd(neg(n1),y))),
                                           zs(zadd(pos(n1),~(y))),
                                           zp~(zadd(neg(n1),y)),
                                           Idzs(~(zadd(neg(n1),y)),
                                                zadd(pos(n1),~(y)),
                                                distadd~(neg(n1),y))),
                                   id(Z,zs(zadd(pos(n1),~(y)))))
```

Properties of subtraction (zminus)

```
zminus : (x:Z;y:Z)Z     [] I

  zminus(x,zz) = x
  zminus(x,pos(0)) = zp(x)
  zminus(x,pos(s(h))) = zp(zminus(x,pos(h)))
  zminus(x,neg(0)) = zs(x)
  zminus(x,neg(s(h))) = zs(zminus(x,neg(h)))

zmintoadd : (x:Z;y:Z)IdZ(zminus(x,y),zadd(x,~(y)))     [] I

  zmintoadd(x,zz) = idtrans(Z,zminus(x,zz),x,zadd(x,~(zz)),
                            id(Z,x),idsymm(Z,zadd(x,~(zz)),x,zadd0(x)))
  zmintoadd(x,pos(0)) = idtrans(Z,
                                zminus(x,pos(0)),
                                zp(x),
                                zadd(x,~(pos(0))),
                                id(Z,zp(x)),
                                idtrans(Z,zp(x),zadd(~(pos(0)),x),
                                        zadd(x,~(pos(0))),id(Z,zp(x)),
                                        zaddcommut(~(pos(0)),x)))
  zmintoadd(x,pos(s(h))) = idtrans(Z,
                                   zminus(x,pos(s(h))),
                                   zp(zadd(x,neg(h))),
                                   zadd(x,~(pos(s(h)))),
                                   Idzp(zminus(x,pos(h)),zadd(x,neg(h)),
                                        zmintoadd(x,pos(h))),
                                   zpaddsnd(x,neg(h)))
  zmintoadd(x,neg(0)) = idtrans(Z,
                                zminus(x,neg(0)),
                                zs(x),
                                zadd(x,~(neg(0))),
                                id(Z,zs(x)),
                                idtrans(Z,
                                        zs(x),
                                        zadd(~(neg(0)),x),
                                        zadd(x,~(neg(0))),
                                        id(Z,zs(x)),
                                        zaddcommut(~(neg(0)),x)))
  zmintoadd(x,neg(s(h))) = idtrans(Z,
                                   zminus(x,neg(s(h))),
                                   zs(zadd(x,pos(h))),
                                   zadd(x,~(neg(s(h)))),
                                   Idzs(zminus(x,neg(h)),
                                        zadd(x,pos(h)),
                                        zmintoadd(x,neg(h))),
                                   zsaddsnd(x,pos(h)))

Szmintoadd = [x,y]idsymm(Z,zminus(x,y),zadd(x,~(y)),
                         zmintoadd(x,y)) : (x:Z;y:Z)IdZ(zadd(x,~(y)),
                                                        zminus(x,y))     []
```

```
zzminus : (x:Z)IdZ(zminus(zz,x),~(x))     [] I

  zzminus(zz) = id(Z,zz)
  zzminus(pos(n)) = idtrans(Z,
                            zminus(zz,pos(n)),
                            zadd(zz,~(pos(n))),
                            ~(pos(n)),
                            zmintoadd(zz,pos(n)),
                            id(Z,neg(n)))
  zzminus(neg(n)) = idtrans(Z,
                            zminus(zz,neg(n)),
                            zadd(zz,pos(n)),
                            ~(neg(n)),
                            zmintoadd(zz,neg(n)),
                            id(Z,pos(n)))

Szzminus = [x]idsymm(Z,zminus(zz,x),~(x),zzminus(x)
                    ) : (x:Z)IdZ(~(x),zminus(zz,x))     []

zadditinvR : (x:Z)IdZ(zadd(x,~(x)),zz)     [] I

  zadditinvR(zz) = id(Z,zz)
  zadditinvR(pos(0)) = id(Z,zz)
  zadditinvR(pos(s(n1))) = idtrans(Z,
                                   zadd(pos(s(n1)),~(pos(s(n1)))),
                                   zs(zminus(pos(n1),pos(s(n1)))),
                                   zz,
                                   Idzs(zadd(pos(n1),~(pos(s(n1)))),
                                        zp(zminus(pos(n1),pos(n1))),
                                        Szmintoadd(pos(n1),pos(s(n1)))),
                                   idtrans(Z,
                                           zs(zminus(pos(n1),pos(s(n1)))),
                                           zs(zp(zadd(pos(n1),~(pos(n1))))),
                                           zz,
                                           idcongr(Z,
                                                   Z,
                                                   [x]zs(zp(x)),
                                                   zminus(pos(n1),pos(n1)),
                                                   zadd(pos(n1),~(pos(n1))),
                                                   zmintoadd(pos(n1),pos(n1))),
                                           idtrans(Z,
                                                   zs(zp(zadd(pos(n1),
                                                              ~(pos(n1))))),
                                                   zadd(pos(n1),~(pos(n1))),
                                                   zz,
                                                   invzp(zadd(pos(n1),
                                                              ~(pos(n1)))),
                                                   zadditinvR(pos(n1)))))
```

```
  zadditinvR(neg(0)) = id(Z,zz)
  zadditinvR(neg(s(n1))) = idtrans(Z,
                                   zadd(neg(s(n1)),~(neg(s(n1)))),
                                   zadd(neg(n1),zp(pos(s(n1)))),
                                   zz,
                                   zpaddsnd(neg(n1),~(neg(s(n1)))),
                                   zadditinvR(neg(n1)))

zadditinvL = [x]idtrans(Z,zadd(~(x),x),zadd(x,~(x)),zz,
                        zaddcommut(~(x),x),zadditinvR(x)
                       ) : (x:Z)IdZ(zadd(~(x),x),zz)     []

SzadditinvL = [x]idsymm(Z,zadd(~(x),x),zz,zadditinvL(x)
                        ) : (x:Z)IdZ(zz,zadd(~(x),x))     []

uniqinv = [x,y,h]zaddcancelft(x,y,~(y),
                              idtrans(Z,
                                      zadd(~(y),x),
                                      zadd(x,~(y)),
                                      zadd(~(y),y),
                                      zaddcommut(~(y),x),
                                      idtrans(Z,
                                              zadd(x,~(y)),
                                              zz,
                                              zadd(~(y),y),
                                              h,
                                      SzadditinvL(y)))
                             ) : (x:Z;y:Z;IdZ(zadd(x,~(y)),zz))IdZ(x,y)   []

~minus = [x,y]idtrans(Z,
                      ~(zminus(x,y)),
                      ~(zadd(x,~(y))),
                      zadd(~(x),y),
                      idcongr(Z,Z,
                              [x]~(x),
                              zminus(x,y),
                              zadd(x,~(y)),
                              zmintoadd(x,y)),
                      idtrans(Z,
                              ~(zadd(x,~(y))),
                              zadd(~(x),~(~(y))),
                              zadd(~(x),y),
                              distadd~(x,~(y)),
                              zaddsubstR(~(~(y)),
                                         y,
                                         ~(x),
                                         idem~(y)))
                      ): (x:Z;y:Z)IdZ(~(zminus(x,y)),zadd(~(x),y))     []
```

```
minus~ = [x,y]idtrans(Z,
                      zadd(x,y),
                      zadd(x,~(~(y))),
                      zminus(x,~(y)),
                      zaddsubstR(y,
                                 ~(~(y)),
                                 x,
                                 Sidem~(y)),
                      Szmintoadd(x,~(y))
                     ) : (x:Z;y:Z)IdZ(zadd(x,y),zminus(x,~(y)))     []
```

Properties of multiplication (ztimes)

```
ztimes : (x:Z;y:Z)Z     [] I

  ztimes(zz,y) = zz
  ztimes(one,y) = y
  ztimes(pos(s(h)),y) = zadd(ztimes(pos(h),y),y)
  ztimes(~(one),y) = zminus(zz,y)
  ztimes(neg(s(h)),y) = zminus(ztimes(neg(h),y),y)

ztimessubstL = [a,b,y,p]idcongr(Z,Z,[h]ztimes(h,y),a,b,p
                                ) : (a:Z;b:Z;y:Z;p:IdZ(a,b))
                                        IdZ(ztimes(a,y),ztimes(b,y))    []

ztimessubstR = [a,b,x,p]idcongr(Z,Z,[h]ztimes(x,h),a,b,p
                               ) : (a:Z;b:Z;x:Z;p:IdZ(a,b))
                                      IdZ(ztimes(x,a),ztimes(x,b))     []

ztimescong = [x,y,w,z,p,q]idtrans(Z,
                                  ztimes(x,w),
                                  ztimes(y,w),
                                  ztimes(y,z),
                                  ztimessubstL(x,y,w,p),
                                  ztimessubstR(w,z,y,q)
                                 ) : (x:Z;y:Z;w:Z;z:Z;p:IdZ(x,y);q:IdZ(w,z))
                                         IdZ(ztimes(x,w),ztimes(y,z))     []

ztimeszz : (x:Z)IdZ(ztimes(x,zz),zz)     [] I

  ztimeszz(zz) = id(Z,zz)
  ztimeszz(one) = id(Z,zz)
  ztimeszz(pos(s(n1))) = idtrans(Z,
                                 ztimes(pos(s(n1)),zz),
                                 ztimes(pos(n1),zz),
                                 zz,
                                 zadd0(ztimes(pos(n1),zz)),
                                 ztimeszz(pos(n1)))
  ztimeszz(~(one)) = id(Z,zz)
```

```
  ztimeszz(neg(s(n1))) = idtrans(Z,
                                 ztimes(neg(s(n1)),zz),
                                 ztimes(neg(n1),zz),
                                 zz,
                                 id(Z,ztimes(neg(n1),zz)),
                                 ztimeszz(neg(n1)))

Sztimeszz = [x]idsymm(Z,ztimes(x,zz),zz,ztimeszz(x)
                    ) : (x:Z)IdZ(zz,ztimes(x,zz))     []

ztimunit : (x:Z)IdZ(ztimes(x,one),x)     [] I

  ztimunit(zz) = id(Z,zz)
  ztimunit(one) = id(Z,one)
  ztimunit(pos(s(n1))) = idtrans(Z,
                                 ztimes(pos(s(n1)),one),
                                 zadd(pos(n1),one),
                                 pos(s(n1)),
                                 zaddcong(ztimes(pos(n1),one),
                                          pos(n1),one,one,
                                          ztimunit(pos(n1)),
                                          id(Z,one)),
                                 idtrans(Z,
                                         zadd(pos(n1),one),
                                         zadd(one,pos(n1)),
                                         pos(s(n1)),
                                         zaddcommut(pos(n1),one),
                                         id(Z,pos(s(n1)))))

  ztimunit(~(one)) = id(Z,~(one))
  ztimunit(neg(s(n1))) = idtrans(Z,
                                 ztimes(neg(s(n1)),one),
                                 zp(neg(n1)),
                                 neg(s(n1)),
                                 idcongr(Z,Z,zp,
                                         ztimes(neg(n1),one),
                                         neg(n1),
                                         ztimunit(neg(n1))),
                                 id(Z,neg(s(n1))))

 Sztimunit = [x]idsymm(Z,ztimes(x,one),x,ztimunit(x)
                     ) : (x:Z)IdZ(x,ztimes(x,one))     []

negztimesL : (x:Z;y:Z)IdZ(~(ztimes(x,y)),ztimes(~(x),y))     [] I

  negztimesL(zz,y) = id(Z,zz)
  negztimesL(pos(0),y) = Szzminus(y))
```

```
  negztimesL(pos(s(n1)),y) = idtrans(Z,
                                     ~(ztimes(pos(s(n1)),y)),
                                     zadd(~(ztimes(pos(n1),y)),~(y)),
                                     ztimes(~(pos(s(n1))),y),
                                     distadd~(ztimes(pos(n1),y),y),
                                     idtrans(Z,
                                             zadd(~(ztimes(pos(n1),y)),~(y)),
                                             zadd(ztimes(neg(n1),y),~(y)),
                                             ztimes(~(pos(s(n1))),y),
                                             zaddsubstL(~(ztimes(pos(n1),y)),
                                                        ztimes(neg(n1),y),
                                                        ~(y),
                                                        negztimesL(pos(n1),y)),
                                             Szmintoadd(ztimes(neg(n1),y),y)))
  negztimesL(neg(0),y) = idtrans(Z,
                                 ~(ztimes(neg(0),y)),
                                 ~(~(y)),
                                 ztimes(~(neg(0)),y),
                                 idcongr(Z,Z,[x]~(x),
                                         zminus(zz,y),~(y),zzminus(y)),
                                         idem~(y))
  negztimesL(neg(s(n1)),y) = idtrans(Z,
                                     ~(ztimes(neg(s(n1)),y)),
                                     ~(zadd(ztimes(neg(n1),y),~(y))),
                                     ztimes(~(neg(s(n1))),y),
                                     idcongr(Z,Z,[x]~(x),
                                            zminus(ztimes(neg(n1),y),y),
                                            zadd(ztimes(neg(n1),y),~(y)),
                                            zmintoadd(ztimes(neg(n1),y),y)),
                                            idtrans(Z,
                                                    ~(zadd(ztimes(neg(n1),y),
                                                           ~(y))),
                                                    zadd(~(ztimes(neg(n1),y)),
                                                         ~(~(y))),
                                                    ztimes(~(neg(s(n1))),y),
                                                    distadd~(ztimes(neg(n1),y),
                                                             ~(y)),
                                                    zaddcong(~(ztimes(neg(n1),
                                                                      y)),
                                                             ztimes(pos(n1),y),
                                                                    ~(~(y)),
                                                             y,
                                                            negztimesL(neg(n1),
                                                                       y),
                                                             idem~(y))))

SnegztimesL = [x,y]idsymm(Z,~(ztimes(x,y)),ztimes(~(x),y),
                          negztimesL(x,y)
                         ) : (x:Z;y:Z)IdZ(ztimes(~(x),y),~(ztimes(x,y))) []

negztimesR : (x:Z;y:Z)IdZ(~(ztimes(x,y)),ztimes(x,~(y)))     [] I
```

```
  negztimesR(zz,y) = id(Z,zz)
  negztimesR(pos(0),y) = id(Z,~(y))
  negztimesR(pos(s(n1)),y) = idtrans(Z,
                                     ~(ztimes(pos(s(n1)),y)),
                                     zadd(~(ztimes(pos(n1),y)),~(y)),
                                     ztimes(pos(s(n1)),~(y)),
                                     distadd~(ztimes(pos(n1),y),y),
                                     zaddsubstL(~(ztimes(pos(n1),y)),
                                                ztimes(pos(n1),~(y)),
                                                ~(y),
                                                negztimesR(pos(n1),y)))
  negztimesR(neg(0),y) = idtrans(Z,
                                  ~(ztimes(neg(0),y)),
                                  ~(~(y)),
                                  ztimes(neg(0),~(y)),
                                  idcongr(Z,Z,[x]~(x),zminus(zz,y),~(y),
                                          zzminus(y)),Szzminus(~(y))))
  negztimesR(neg(s(n1)),y) = idtrans(Z,
                                     ~(ztimes(neg(s(n1)),y)),
                                     zadd(~(ztimes(neg(n1),y)),y),
                                     ztimes(neg(s(n1)),~(y)),
                                     ~minus(ztimes(neg(n1),y),y),
                                     idtrans(Z,
                                             zadd(~(ztimes(neg(n1),y)),y),
                                             zadd(ztimes(neg(n1),~(y)),y),
                                             ztimes(neg(s(n1)),~(y)),
                                             zaddsubstL(~(ztimes(neg(n1),y)),
                                                        ztimes(neg(n1),~(y)),
                                                        y,
                                                        negztimesR(neg(n1),y)),
                                             minus~(ztimes(neg(n1),~(y)),y)))

SnegztimesR = [x,y]idsymm(Z,~(ztimes(x,y)),ztimes(x,~(y)),
                          negztimesR(x,y)
                         ) : (x:Z;y:Z)IdZ(ztimes(x,~(y)),~(ztimes(x,y)))  []

negztimesLtoR = [x,y]idtrans(Z,
                             ztimes(~(x),y),
                             ~(ztimes(x,y)),
                             ztimes(x,~(y)),
                             idsymm(Z,~(ztimes(x,y)),
                                      ztimes(~(x),y),negztimesL(x,y)),
                             negztimesR(x,y)
                             ):(x:Z;y:Z)IdZ(ztimes(~(x),y),
                                            ztimes(x,~(y)))     []
```

```
timdoub~ = [x,y]idtrans(Z,
                        ztimes(~(x),~(y)),
                        ~(ztimes(x,~(y))),
                        ztimes(x,y),
                        idsymm(Z,~(ztimes(x,~(y))),ztimes(~(x),~(y)),
                                 negztimesL(x,~(y))),
                        idtrans(Z,
                                ~(ztimes(x,~(y))),
                                ~(~(ztimes(x,y))),
                                ztimes(x,y),
                                idsymm(Z,
                                       ~(~(ztimes(x,y))),
                                       ~(ztimes(x,~(y))),
                                       idcongr(Z,Z,[x]~(x),
                                               ~(ztimes(x,y)),
                                               ztimes(x,~(y)),
                                               negztimesR(x,y))),
                                idem~(ztimes(x,y)))
                       ) : (x:Z;y:Z)IdZ(ztimes(~(x),~(y)),ztimes(x,y))     []

ztiminone : (x:Z)IdZ(ztimes(x,~(one)),~(x))     [] I

  ztiminone(zz) = id(Z,zz)
  ztiminone(pos(n)) = idtrans(Z,
                              ztimes(pos(n),~(one)),
                              ~(ztimes(pos(n),one)),
                              ~(pos(n)),
                              idsymm(Z,
                                     ~(ztimes(pos(n),one)),
                                     ztimes(pos(n),~(one)),
                                     negztimesR(pos(n),one)),
                                     idcongr(Z,Z,[x]~(x),
                                             ztimes(pos(n),one),
                                             pos(n),
                                             ztimunit(pos(n))))
  ztiminone(neg(n)) = idtrans(Z,
                              ztimes(neg(n),~(one)),
                              ~(ztimes(neg(n),one)),
                              ~(neg(n)),
                              idsymm(Z,
                                     ~(ztimes(neg(n),one)),
                                     ztimes(neg(n),~(one)),
                                     negztimesR(neg(n),one)),
                              idcongr(Z,Z,[x]~(x),ztimes(neg(n),one),
                                      neg(n),ztimunit(neg(n))))

Sztiminone = [a]idsymm(Z,ztimes(a,neg(0)),~(a),ztiminone(a))
                     ) : (a:Z)IdZ(~(a),ztimes(a,neg(0))) []
```

```
lemmdistL1 = [a,b1,c,d1]
             idtrans(Z,
                     zadd(zadd(zadd(a,b1),c),d1),
                     zadd(zadd(a,zadd(b1,c)),d1),
                     zadd(zadd(a,c),zadd(b1,d1)),
                     zaddsubstL(zadd(zadd(a,b1),c),
                                zadd(a,zadd(b1,c)),d1,
                                zaddssoc(a,b1,c)),
                     idtrans(Z,zadd(zadd(a,zadd(b1,c)),d1),
                               zadd(zadd(a,zadd(c,b1)),d1),
                               zadd(zadd(a,c),zadd(b1,d1)),
                               zaddsubstL(zadd(a,zadd(b1,c)),
                                          zadd(a,zadd(c,b1)),d1,
                                          zaddsubstR(zadd(b1,c),
                                                     zadd(c,b1),a,
                                                     zaddcommut(b1,c))),
                               idtrans(Z,
                                       zadd(zadd(a,zadd(c,b1)),d1),
                                       zadd(zadd(zadd(a,c),b1),d1),
                                       zadd(zadd(a,c),zadd(b1,d1)),
                                       zaddsubstL(zadd(a,zadd(c,b1)),
                                                  zadd(zadd(a,c),b1),d1,
                                                  Szaddddssoc(a,c,b1))),
                                       zaddssoc(zadd(a,c),b1,d1)))
                     ) : (a:Z;b:Z;c:Z;d:Z)IdZ(zadd(zadd(zadd(a,b),c),d),
                                               zadd(zadd(a,c),zadd(b,d)))  []

lemdistL2 = [a,b1,c,d1]
            idtrans(Z,
                    zadd(zadd(a,b1),zadd(c,d1)),
                    zadd(zadd(zadd(a,b1),c),d1),
                    zadd(zadd(a,c),zadd(b1,d1)),
                    Szaddssoc(zadd(a,b1),c,d1),
                    lemmdistL1(a,b1,c,d1)
                    ) : (a:Z;b:Z;c:Z;d:Z)IdZ(zadd(zadd(a,b),zadd(c,d)),
                                             zadd(zadd(a,c),zadd(b,d)))   []
```

```
lemmdistL3 = [a,b,c,d]
             idtrans(Z,
                     zminus(zadd(a,b),zadd(c,d)),
                     zadd(zadd(a,b),~(zadd(c,d))),
                     zadd(zminus(a,c),zminus(b,d)),
                     zmintoadd(zadd(a,b),zadd(c,d)),
                               idtrans(Z,
                                       zadd(zadd(a,b),~(zadd(c,d))),
                                       zadd(zadd(a,b),zadd(~(c),~(d))),
                                       zadd(zminus(a,c),zminus(b,d)),
                                       zaddsubstR(~(zadd(c,d)),
                                                  zadd(~(c),~(d)),
                                                  zadd(a,b),
                                                distadd~(c,d)),
                                       idtrans(Z,
                                               zadd(zadd(a,b),
                                                    zadd(~(c),~(d))),
                                               zadd(zadd(a,~(c)),
                                                    zadd(b,~(d))),
                                               zadd(zminus(a,c),zminus(b,d)),
                                               lemmdistL2(a,b,~(c),~(d)),
                                               zaddcong(zadd(a,~(c)),
                                                        zminus(a,c),
                                                        zadd(b,~(d)),
                                                        zminus(b,d),
                                                        Szmintoadd(a,c),
                                                        Szmintoadd(b,d))))
                    ) : (a:Z;b:Z;c:Z;d:Z)IdZ(zminus(zadd(a,b),zadd(c,d)),
                                             zadd(zminus(a,c),zminus(b,d))) []

ztimesdistL : (a:Z;b:Z;c:Z)IdZ(ztimes(c,zadd(a,b)),
                               zadd(ztimes(c,a),ztimes(c,b)))     [] I

  ztimesdistL(a,b,zz) = id(Z,zz)
  ztimesdistL(a,b,pos(0)) = id(Z,zadd(a,b))
  ztimesdistL(a,b,pos(s(n1))) = idtrans(Z,
                                        ztimes(pos(s(n1)),zadd(a,b)),
                                        zadd(zadd(ztimes(pos(n1),a),
                                                  ztimes(pos(n1),b)),
                                             zadd(a,b)),
                                        zadd(ztimes(pos(s(n1)),a),
                                             ztimes(pos(s(n1)),b)),
                                        zaddsubstL(ztimes(pos(n1),zadd(a,b)),
                                                   zadd(ztimes(pos(n1),a),
                                                        ztimes(pos(n1),b)),
                                                   zadd(a,b),
                                                   ztimesdistL(a,b,pos(n1))),
                                        lemmdistL2(ztimes(pos(n1),a),
                                                   ztimes(pos(n1),b),a,b))
```

```
  ztimesdistL(a,b,neg(0)) = idtrans(Z,
                                    ztimes(neg(0),zadd(a,b)),
                                    ~(zadd(a,b)),
                                    zadd(ztimes(neg(0),a),ztimes(neg(0),b)),
                                    zzminus(zadd(a,b)),
                                    idtrans(Z,
                                            ~(zadd(a,b)),
                                            zadd(~(a),~(b)),
                                            zadd(ztimes(neg(0),a),
                                                 ztimes(neg(0),b)),
                                            distadd~(a,b),
                                            zaddcong(~(a),
                                                     zminus(zz,a),
                                                     ~(b),
                                                     zminus(zz,b),
                                                     Szzminus(a)),
                                                     Szzminus(b)))))

 ztimesdistL(a,b,neg(s(n1))) = idtrans(Z,
                                       ztimes(neg(s(n1)),zadd(a,b)),
                                       zminus(zadd(ztimes(neg(n1),a),
                                                   ztimes(neg(n1),b)),
                                              zadd(a,b)),
                                       zadd(ztimes(neg(s(n1)),a),
                                            ztimes(neg(s(n1)),b)),
                                       idcongr(Z,Z,
                                               [x]zminus(x,zadd(a,b)),
                                               ztimes(neg(n1),zadd(a,b)),
                                               zadd(ztimes(neg(n1),a),
                                                    ztimes(neg(n1),b)),
                                               ztimesdistL(a,b,neg(n1))),
                                       lemmdistL3(ztimes(neg(n1),a),
                                                  ztimes(neg(n1),b),a,b))

SztimesdistL = [a,b1,c]idsymm(Z,
                              ztimes(c,zadd(a,b1)),
                              zadd(ztimes(c,a),ztimes(c,b1)),
                              ztimesdistL(c,a,b1)
                             ) : (a:Z;b:Z;c:Z)IdZ(zadd(ztimes(c,a),
                                                       ztimes(c,b)),
                                                  ztimes(c,zadd(a,b)))  []
lemmcomm1 = [a,b]
            idtrans(Z,
                    zadd(ztimes(a,b),a),
                    zadd(ztimes(a,b),ztimes(a,one)),
                    ztimes(a,zadd(b,one)),
                    zaddsubstR(a,ztimes(a,one),ztimes(a,b),
                               Sztimunit(a)),
                    SztimesdistL(b,one,a)
                   ) : (a:Z;b:Z)IdZ(zadd(ztimes(a,b),a),
                                    ztimes(a,zadd(b,one)))     []
```

```
lemcomm2 = [a,b]
           idtrans(Z,
                   zminus(ztimes(a,b),a),
                   zadd(ztimes(a,b),~(a)),
                   ztimes(a,zminus(b,one)),
                   zmintoadd(ztimes(a,b),a),
                   idtrans(Z,
                           zadd(ztimes(a,b),~(a)),
                           zadd(ztimes(a,b),ztimes(a,neg(0))),
                           ztimes(a,zminus(b,one)),
                           zaddsubstR(~(a),
                                      ztimes(a,neg(0)),
                                      ztimes(a,b),
                                      Sztiminone(a)),
                           idtrans(Z,
                                   zadd(ztimes(a,b),ztimes(a,neg(0))),
                                   ztimes(a,zadd(b,neg(0))),
                                   ztimes(a,zminus(b,one)),
                                   SztimesdistL(b,neg(0),a)),
                                   ztimessubstR(zadd(b,neg(0)),
                                                zminus(b,one),a,
                                   Szmintoadd(b,pos(0))))
                   ) : (a:Z;b:Z)IdZ(zminus(ztimes(a,b),a),
                                    ztimes(a,zminus(b,one)))     []

ztimescomm : (a:Z;b:Z)IdZ(ztimes(a,b),ztimes(b,a))     []

 ztimescomm(zz,b) = idsymm(Z,ztimes(b,zz),zz,ztimeszz(b))
 ztimescomm(pos(0),b) = Sztimunit(b)
 ztimescomm(pos(s(n1)),b) = idtrans(Z,
                                    zadd(ztimes(pos(n1),b),b),
                                    zadd(ztimes(b,pos(n1)),b),
                                    ztimes(b,zadd(one,pos(n1))),
                                    zaddsubstL(ztimes(pos(n1),b),
                                               ztimes(b,pos(n1)),b,
                                               ztimescomm(pos(n1),b)),
                                    idtrans(Z,
                                            zadd(ztimes(b,pos(n1)),b),
                                            ztimes(b,zadd(pos(n1),one)),
                                            ztimes(b,zadd(one,pos(n1))),
                                            lemmcomm1(b,pos(n1)),
                                            ztimessubstR(zadd(pos(n1),one),
                                                         zadd(one,pos(n1)),
                                                         b,
                                                    zaddcommut(pos(n1),one))))
 ztimescomm(neg(0),b) = idtrans(Z,
                                zminus(zz,b),
                                ~(b),
                                ztimes(b,neg(0)),
                                zzminus(b),
                                Sztiminone(b)))
```

```
  ztimescomm(neg(s(n1)),b) = idtrans(Z,
                                     ztimes(neg(s(n1)),b),
                                     zminus(ztimes(b,neg(n1)),b),
                                     ztimes(b,neg(s(n1))),
                                     idcongr(Z,Z,
                                             [x]zminus(x,b),
                                             ztimes(neg(n1),b),
                                             ztimes(b,neg(n1)),
                                             ztimescomm(neg(n1),b)),
                                     lemcomm2(b,neg(n1)))
ztimesdistR = [a,b,c]
              idtrans(Z,
                      ztimes(zadd(a,b),c),
                      ztimes(c,zadd(a,b)),
                      zadd(ztimes(a,c),ztimes(b,c)),
                      ztimescomm(zadd(a,b),c),
                      idtrans(Z,
                              ztimes(c,zadd(a,b)),
                              zadd(ztimes(c,a),ztimes(c,b)),
                              zadd(ztimes(a,c),ztimes(b,c)),
                              ztimesdistL(c,a,b),
                              zaddcong(ztimes(c,a),ztimes(a,c),
                                       ztimes(c,b),ztimes(b,c),
                                       ztimescomm(c,a),
                                       ztimescomm(c,b)))
                    ) : (a:Z;b:Z;c:Z)IdZ(ztimes(zadd(a,b),c),
                                         zadd(ztimes(a,c),ztimes(b,c)))  []

SztimesdistR = [a,b,c]idsymm(Z,
                             ztimes(zadd(a,c),b),
                             zadd(ztimes(a,b),ztimes(c,b)),
                             ztimesdistR(a,c,b)
                             ) : (a:Z;b:Z;c:Z)Id(Z,zadd(ztimes(a,b),
                                                        ztimes(c,b)),
                                                 ztimes(zadd(a,c),b))  []

lemmassoc1 = [a,b]idtrans(Z,
                          zadd(ztimes(a,b),b),
                          zadd(ztimes(a,b),ztimes(pos(0),b)),
                          ztimes(zadd(a,pos(0)),b),
                          zaddsubstR(b,ztimes(pos(0),b),ztimes(a,b),
                                     idrefl(Z,b)),
                          SztimesdistR(a,b,pos(0))
                         ) : (a:Z;b:Z)IdZ(zadd(ztimes(a,b),b),
                                          ztimes(zadd(a,pos(0)),b))     []
```

```
lemmassneg = [a,b1,c]
             idtrans(Z,
                     ztimes(zminus(a,b1),c),
                     ztimes(zadd(a,~(b1)),c),
                     zminus(ztimes(a,c),ztimes(b1,c)),
                     ztimessubstL(zminus(a,b1),zadd(a,~(b1)),c,
                                  zmintoadd(a,b1)),
                     idtrans(Z,
                             ztimes(zadd(a,~(b1)),c),
                             zadd(ztimes(a,c),ztimes(~(b1),c)),
                             zminus(ztimes(a,c),ztimes(b1,c)),
                             ztimesdistR(a,~(b1),c),
                             idtrans(Z,
                                     zadd(ztimes(a,c),ztimes(~(b1),c)),
                                     zadd(ztimes(a,c),~(ztimes(b1,c))),
                                     zminus(ztimes(a,c),ztimes(b1,c)),
                                     zaddsubstR(ztimes(~(b1),c),
                                                ~(ztimes(b1,c)),
                                                ztimes(a,c),
                                                SnegztimesL(b1,c)),
                                     Szmintoadd(ztimes(a,c),ztimes(b1,c))))
                     ) : (a:Z;b:Z;c:Z)IdZ(ztimes(zminus(a,b),c),
                                          zminus(ztimes(a,c),ztimes(b,c))) []

ztimesassoc : (x,y,z:Z)IdZ(ztimes(ztimes(x,y),z),ztimes(x,ztimes(y,z))) [] I

 ztimesassoc(zz,y,z) = id(Z,zz)
 ztimesassoc(pos(0),y,z) = id(Z,ztimes(y,z))
 ztimesassoc(pos(s(n1)),y,z) = idtrans(Z,
                                       ztimes(zadd(ztimes(pos(n1),y),y),z),
                                       zadd(ztimes(ztimes(pos(n1),y),z),
                                            ztimes(y,z)),
                                       zadd(ztimes(pos(n1),ztimes(y,z)),
                                            ztimes(y,z)),
                                       ztimesdistR(ztimes(pos(n1),y),y,z),
                                       zaddsubstL(ztimes(ztimes(pos(n1),y),z),
                                                  ztimes(pos(n1),ztimes(y,z)),
                                                  ztimes(y,z),
                                                  ztimesassoc(pos(n1),y,z)))

 ztimesassoc(neg(0),y,z) = idtrans(Z,
                                   ztimes(zminus(zz,y),z),
                                   ztimes(~(y),z),
                                   zminus(zz,ztimes(y,z)),
                                   ztimessubstL(zminus(zz,y),~(y),z,
                                                zzminus(y)),
                                   idtrans(Z,
                                           ztimes(~(y),z),
                                           ~(ztimes(y,z)),
                                           zminus(zz,ztimes(y,z)),
                                           SnegztimesL(y,z),
                                           Szzminus(y,z)))
```

```
 ztimesassoc(neg(s(n1)),y,z) = idtrans(Z,
                                       ztimes(ztimes(neg(s(n1)),y),z),
                                       zminus(ztimes(ztimes(neg(n1),y),z),
                                                     ztimes(y,z)),
                                       ztimes(neg(s(n1)),ztimes(y,z)),
                                       lemmassneg(ztimes(neg(n1),y),y,z),
                                       idcongr(Z,Z,
                                               [x]zminus(x,ztimes(y,z)),
                                               ztimes(ztimes(neg(n1),y),z),
                                               ztimes(neg(n1),ztimes(y,z)),
                                               ztimesassoc(neg(n1),y,z)))


(* Lemmas for and proof of cancellation on ztimes *)

pos_times_pos : (m:N;n:N)IdZ(ztimes(pos(m),pos(n)),
                        pos(plus(mult(m,s(n)),n)))     [] I

pos_times_neg : (m:N;n:N)IdZ(ztimes(pos(m),neg(n)),
                        neg(plus(mult(m,s(n)),n)))     [] I

neg_times_pos = [m,n]idtrans(Z,
                        ztimes(neg(m),pos(n)),
                        ztimes(pos(n),neg(m)),
                        neg(plus(mult(n,s(m)),m)),
                        ztimescomm(neg(m),pos(n)),
                        pos_times_neg(n,m)
                       ) : (m:N;n:N)IdZ(ztimes(neg(m),pos(n)),
                                        neg(plus(mult(n,s(m)),m)))     []


neg_times_neg = [m,n]idtrans(Z,
                     ztimes(neg(m),neg(n)),
                     ztimes(pos(m),pos(n)),
                     pos(plus(mult(m,s(n)),n)),
                     timdoub~(pos(m),pos(n)),
                     pos_times_pos(m,n)
                    ) : (m:N;n:N)IdZ(ztimes(neg(m),neg(n)),
                                     pos(plus(mult(m,s(n)),n)))     []

nzeroptp = [m,n,h]posnotzz(plus(mult(m,s(n)),n),
                           idtrans(Z,
                                   pos(plus(mult(m,s(n)),n)),
                                   ztimes(pos(m),pos(n)),
                                   zz,
                                   idsymm(Z,
                                          ztimes(pos(m),pos(n)),
                                          pos(plus(mult(m,s(n)),n)),
                                          pos_times_pos(m,n)),
                                   h)
                          ) : (m:N;n:N;IdZ(ztimes(pos(m),pos(n),
                                           zz))Empty     []
```

```
nzeroptn = [m,n,h]negnotzz(plus(mult(m,s(n)),n),
                                idtrans(Z,
                                        neg(plus(mult(m,s(n)),n)),
                                        ztimes(pos(m),neg(n)),
                                        zz,
                                        idsymm(Z,
                                               ztimes(pos(m),neg(n)),
                                               neg(plus(mult(m,s(n)),n)),
                                               pos_times_neg(m,n)),
                                        h)
                               ) : (m:N;n:N;IdZ(ztimes(pos(m),neg(n)),
                                                zz))Empty     []

nzerontpz = [m,n,h]negnotzz(plus(mult(n,s(m)),m),
                                idtrans(Z,
                                        neg(plus(mult(n,s(m)),m)),
                                        ztimes(neg(m),pos(n)),
                                        zz,
                                        idsymm(Z,
                                               ztimes(neg(m),pos(n)),
                                               neg(plus(mult(n,s(m)),m)),
                                               neg_times_pos(m,n)),
                                        h)
                                ) : (m:N;n:N;IdZ(ztimes(neg(m),pos(n)),
                                                 zz))Empty     []

nzerontn = [m,n,h]posnotzz(plus(mult(m,s(n)),n),
                                idtrans(Z,
                                        pos(plus(mult(m,s(n)),n)),
                                            ztimes(neg(m),neg(n)),
                                            zz,
                                            idsymm(Z,
                                                   ztimes(neg(m),neg(n)),
                                                   pos(plus(mult(m,s(n)),n)),
                                                   neg_times_neg(m,n)),
                                        h)
                                ) : (m:N;n:N;IdZ(ztimes(neg(m),neg(n)),
                                                 zz))Empty     []
```

```
lemmcancel1 : (x:Z;y:Z;IdZ(ztimes(x,y),zz);Isnotzz(x))IdZ(y,zz)     [] I

  lemmcancel1(zz,y,h,h1) = case h1 of end
  lemmcancel1(pos(n),y,h,h1) =
  case y of
           neg(n1) => case nzeroptn(n,n1,h) of end
           pos(n1) => case nzeroptp(n,n1,h) of end
           zz => id(Z,zz)
  end
  lemmcancel1(neg(n),y,h,h1) =
  case y of
           neg(n1) => case nzerontn(n,n1,h) of end
           pos(n1) => case nzerontpz(n,n1,h) of end
           zz => id(Z,zz)
  end

lemmcancel2 = [a,b]idtrans(Z,
                       zadd(ztimes(a,~(b)),ztimes(a,b)),
                       zadd(~(ztimes(a,b)),ztimes(a,b)),
                       zz,
                       zaddsubstL(ztimes(a,~(b)),
                                  ~(ztimes(a,b)),
                                  ztimes(a,b),
                                  SnegztimesR(a,b)),
                       zadditinvL(ztimes(a,b))
                      ) : (a:Z;b:Z)IdZ(zadd(ztimes(a,~(b)),
                                            ztimes(a,b)),zz)     []

ztimcanlft = [a,b,c,h,h1]
             uniqinv(a,b,
                     lemmcancel1(c,
                                 zadd(a,~(b)),
                                 multsubs(zadd(ztimes(c,a),ztimes(c,~(b))),
                                          zadd(ztimes(c,b),ztimes(c,~(b))),
                                          ztimes(c,zadd(a,~(b))),
                                          zz,
                                          zaddsubstL(ztimes(c,a),
                                                     ztimes(c,b),
                                                     ztimes(c,~(b)),h),
                                                     SztimesdistL(a,~(b),c),
                                          idtrans(Z,
                                                  zadd(ztimes(c,b),
                                                       ztimes(c,~(b))),
                                                  zadd(ztimes(c,~(b)),
                                                       ztimes(c,b)),
                                                  zz,
                                                  zaddcommut(ztimes(c,b),
                                                             ztimes(c,~(b))),
                                                  lemmcancel2(c,b))),
                                  h1)
                      ) : (a:Z;b:Z;c:Z;
                           IdZ(ztimes(c,a),ztimes(c,b));Isnotzz(c)) IdZ(a,b) []
```

C The integral domain formed by $\langle Z, zadd, ztimes\rangle$

```
idreflZ = idrefl(Z) :(x:Z)IdZ(x,x) []
idsymmZ = idsymm(Z) : (x,y:Z;IdZ(x,y))IdZ(y,x) []
idtransZ = idtrans(Z) : (x,y,z:Z;IdZ(x,y);IdZ(y,z))IdZ(x,z) []

onenotzero = lambda(IdZ(one,zz),Empty,posnotzz(0)) : not(IdZ(one,zz)) []

zaddident = [x]And_intro(IdZ(zadd(x,zz),x),
                         Id(Z,zadd(zz,x),x),
                         zadd0(x),
                         idrefl(Z,zadd(zz,x))
                        ) : (x:Z)And(IdZ(zadd(x,zz),x),IdZ(zadd(zz,x),x))  []

zaddinv = [x]And_intro(IdZ(zadd(x,~(x)),zz),
                       IdZ(zadd(~(x),x),zz),
                       zadditinvR(x),
                       zadditinvL(x)
                       ):(x:Z)And(IdZ(zadd(x,~(x)),zz),IdZ(zadd(~(x),x),zz)) []


IntdomZ  is {S := Z;R := IdZ;
             refl := idreflZ;symm := idsymmZ;trans := idtransZ;
             op := zadd;opcong := zaddcong;unit := zz;assoc := Szaddssoc;
             ident := zaddident; inv := ~;opinv := zaddinv;comm := zaddcommut;
             mult := ztimes;multcong := ztimescong;
             munit := one;diff := onenotzero;massoc := Sztimesassoc;
             mident := ztimunit;distlft := ztimesdistL;mcomm := ztimescomm;
             mcancel := ztimcanlft} : Intdom  []
```

The derived properties for Z

```
uniqinvZ = uniqinv{Intdomz} : (a,b,c:Z;
                               IdZ(zadd(a,b),zz);
                               IdZ(zadd(a,c),zz)
                              )IdZ(b,c) []

uniqunitZ = uniqunit{IntdomZ} : (a:Z;p:(x:Z)IdZ(zadd(x,a),x))IdZ(a,zz) []

timeszeroZ = timeszero{IntdomZ} : (x:Z)IdZ(ztimes(x,zz),zz) []

nzdivZ = nzdiv{IntdomZ} : (a,b:Z;
                           IdZ(ztimes(a,b),zz);
                           not(IdZ(a,zz))
                          )IdZ(b,zz) []

distrZ = distr{IntdomZ} : (x,y,z:Z)IdZ(ztimes(zadd(x,y),z),
                                       zadd(ztimes(x,z),ztimes(y,z))) []
```