On the Most Suitable Axiomatization of Signed Integers

Hubert Garavel

To cite this version:

Hubert Garavel. On the Most Suitable Axiomatization of Signed Integers. 23th International Workshop on Algebraic Development Techniques (WADT), Sep 2017, Gregynog, Wales, UK, United Kingdom. pp.120-134, 10.1007/978-3-319-72044-9_9 . hal-01667321

HAL Id: hal-01667321
https://hal.inria.fr/hal-01667321

Submitted on 19 Dec 2017

On the Most Suitable Axiomatization of Signed Integers

Hubert Garavel

INRIA, Grenoble, France
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
Saarland University, Saarbrücken, Germany
E-mail: hubert.garavel@inria.fr
http://convecs.inria.fr

Abstract

The standard mathematical definition of signed integers, based on set theory, is not well-adapted to the needs of computer science. For this reason, many formal specification languages and theorem provers have designed alternative definitions of signed integers based on term algebras, by extending the Peano-style construction of unsigned naturals using "zero" and "succ" to the case of signed integers. We compare the various approaches used in CADP, CASL, Coq, Isabelle/HOL, KIV, Maude, mCRL2, PSF, SMT-LIB, TLA+, etc. according to objective criteria and suggest an "optimal" definition of signed integers.

Keywords: abstract data type, algebraic specification, computer language, formal method, integer number, natural number, number theory, semantics, term rewrite system, theorem proving

1 Introduction

It took a few millennia to properly formalize number theory but, at present, mathematics has sound and well-established concepts for numbers.

In computer science, the situation is different. Following a tradition initiated by Fortran and Algol 60, most programming languages rely on a set of predefined data types, among which numerical types (integers, reals, etc.) have finite domains and usually map to the machine words provided by the underlying hardware. In many cases, the semantics of these types is not defined formally, as it depends on the implementation. Even languages with strong semantic foundations may be incompletely defined if they import predefined types; this is the case, for instance, with the synchronous languages Lustre [31, Sect. 3.1 and 3.2], Esterel [5, Sect. 4.3.1], and Signal [13, Sect. 2.3], which assume the existence of predefined signed integers and floating-point reals, presumably imported from the C language.

Some specification languages use a similar approach, by assuming the existence of numerical types rather than defining them formally. Because specification languages are expected to be more abstract and higher-level than programming languages, such predefined numerical types are usually infinite and the mapping to hardware implementation is often left aside. These are four examples of specification languages in which numbers are taken for granted[1]:

- The VDM language [15] [19, Sect. 3.1.2] has five predefined numerical types, real numbers being the most general one ($\texttt{nat1} \subset \texttt{nat} \subset \texttt{int} \subset \texttt{rat} \subset \texttt{real}$).
- The predefined library of PVS [25, Chap. 7] assumes the existence of a universal `number` type, of which the usual numerical types are subsets ($\texttt{naturalnumber} \subset \texttt{integer} \subset \texttt{rational} \subset \texttt{real} \subset \texttt{number}$). Actually, PVS defines many more types (e.g., `posnat`, `nonneg_int`, `nzrat`, etc.) that form a lattice, the elements of which are related by PVS judgments and properties.
- The Z notation [16, Sect. 11.2 and B.7] assumes the existence of an unspecified number type $\mathbb{A}$ ("arithmos") that contains naturals and integers ($\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{A}$). These sets and their operations are defined using high-level statements such as: "*multiplication on integers is characterised by the unique operation under which the integers become a commutative ring with identity element 1*" [16, Sect. B.7.11].
- The B language [20, Sect. 3.4, 5.3, 5.6, and 7.25.2] assumes the existence of the set $\mathbb{Z}$, of which $\mathbb{N}_1$ and $\mathbb{N}$ are subsets ($\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z}$). The language also defines a set `INT` of "concrete integers" that belong to a finite range `MININT`...`MAXINT`, the bounds of which are implementation-dependent, e.g., $(-2^{31})...(2^{31}-1)$, as well as two subsets $\texttt{NAT}_1$ and `NAT` of `INT` without negative values ($\texttt{NAT}_1 \subset \texttt{NAT} \subset \texttt{INT} \subset \mathbb{Z} \wedge \texttt{NAT}_1 \subset \mathbb{N}_1 \wedge \texttt{NAT} \subset \mathbb{N}$).

Relying on undefined or externally-defined numerical types makes these languages closer to programming languages than formal methods. Their semantics is not entirely defined and properties involving numbers cannot be proven within these languages, unless some specific theories are imported. We believe that a unified semantic definition that encompasses numbers is highly preferable.

The present article addresses the following problem: *what is the best way to define the set $\mathbb{Z}$ of integers formally?* An ideal definition should be mathematically elegant and compatible with the needs of computer-aided verification.

Our motivation for this question arose in 1996 when applying the LOTOS language [14] to an industrial case study that required signed integers: the predefined library of LOTOS, based on ACT-ONE [9] abstract data types, provided only natural numbers, but no signed integers. At that time, no obvious solution could be found in the literature. During the past decades, various approaches have been devised for this problem and implemented in mainstream specification languages and theorem provers. The present article reviews and compares

[1] In the present article, we distinguish between *naturals* (also: *natural numbers*), which are the elements of $\mathbb{N}$ (i.e., unsigned), and *integers*, which are the elements of $\mathbb{Z}$ (i.e., signed). We assume that $0 \in \mathbb{N}$ and we write $\mathbb{N}_1 =_{def} \mathbb{N} \setminus \{0\}$.

these approaches. We restrict our study to arbitrary large naturals and integers (i.e., the mathematical sets $\mathbb{N}$ and $\mathbb{Z}$), thus excluding finite subranges of $\mathbb{N}$ and $\mathbb{Z}$, especially machine numbers (e.g., 32-bit integers) and modular arithmetic.

2 Definitions of $\mathbb{N}$

In mathematics, natural numbers can be constructed in two main ways:

- *Construction based on set theory*: assuming that the concept of set preexists, we know from the works of Zermelo, Fraenkel, and Von Neumann that natural numbers can be defined as the following sequence of sets:

  $$\begin{array}{lll} 0 & =_{def} & \varnothing \\ 1 & =_{def} & 0 \cup \{0\} = \{0\} = \{\varnothing\} \\ 2 & =_{def} & 1 \cup \{1\} = \{0, 1\} = \{\varnothing, \{\varnothing\}\} \\ 3 & =_{def} & 2 \cup \{2\} = \{0, 1, 2\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\} \end{array}$$

  and so on, where number $n+1$ is defined as the set $n \cup \{n\}$, i.e., $\{0, 1, ..., n\}$. We are not aware of any computer language that uses this approach to define natural numbers. Even formal methods based on set theory, such as VDM, Z, B, or TLA+ [18] do not define their naturals this way.

- *Construction based on algebraic terms:* we know from Peano axioms that natural numbers can be represented as algebraic terms built with two constructors [`zero` : $\rightarrow \mathbb{N}$] and [`succ` : $\mathbb{N} \rightarrow \mathbb{N}$]. Because this approach lends itself well to machine execution and automated reasoning, it has been adopted by most computer languages that formally define their numbers.

  The simplicity of this approach fits theoretical needs well but faces practical limitations: Peano-style naturals are encoded in base one (*unary* representation), which is cumbersome when writing large numbers, and inefficient when directly executing algebraic specifications by giving them as input to some rewrite engine or encoding them into some language (e.g., a functional programming language) that supports inductive types and pattern matching; in practice, unary representation often causes stack overflow for naturals larger than 10^6.

  For this reason, refined approaches have been proposed to represent naturals more compactly and reduce the amount of rewrite steps; this is usually done by encoding numbers in a base different from one, e.g., two [2] [4], three [8], four [7], ten [32] [17] [2] [4] [29], sixteen [2], or in some arbitrary base [33, Systems DA and JP]. In this article, we stick to the unary representation, since it is used by a majority of specification languages and software tools.

The choice between set theory and algebraic terms for defining unsigned naturals can also be found in the construction of signed integers. In the sequel, we examine each approach in turn.

3 Approach 1: Definition of $\mathbb{Z}$ Using Set Theory

In mathematical textbooks, the set of integers is defined as $\mathbb{Z} = (\mathbb{N} \times \mathbb{N}) / \sim$, where × is the cartesian product and ∼ the equivalence relation such that $(x, y) \sim (x', y') \iff x + y' = x' + y$.

This approach has been retained for defining integers in CASL [23, Sect. V:2, page 381] and Isabelle/HOL [28, Sect. 52 page 586]. It has the merit of exhibiting a nice analogy with rational numbers, whose set can be similarly defined as $\mathbb{Q} = (\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})) / \sim$, where ∼ is another equivalence relation such that $(x, y) \sim (x', y') \iff x \cdot y' = x' \cdot y$ — but the analogy does not hold beyond this point, as the set $\mathbb{R}$ of real numbers cannot be defined as a quotient set of $\mathbb{Q} \times \mathbb{Q}$.

Such a set-theoretic definition of integers has various drawbacks: (i) it relies on cartesian product and quotient set, which are involved notions, compared to the simplicity of Peano axioms; (ii) it makes the definition of integers very different from that of naturals; (iii) it goes against the intuition as it builds a two-dimensional surface where a half line towards negative integers would be sufficient; (iv) it does not support reasoning by induction on integers, as pointed out, e.g., in [24, Sect. 8.4 page 165]; (v) it is not optimal computationally, neither in memory (the cartesian product suggests that it takes two naturals to build one integer) nor in time (the quotient operation requires two additions and an equality test to compare two integers).

Having mentioned this approach, we now consider, in the remainder of this article, alternative definitions based upon algebraic terms rather than set theory.

4 Formal definitions

4.1 Syntactic notations

Following the established terminology of algebraic specification, we define a *sort*[2] to be a collection of algebraic terms. These terms should be well-typed, meaning all the usual constraints arising from the arity of operations, the sorts of operation arguments, and the sorts of operation results must be taken into account.

If t is a term of sort S, if f is a unary operation $[f : S \to S]$, and if n is a natural number, let $f^n(t)$ be the term defined inductively by $f^0(t) =_{def} t$ and $f^{n+1}(t) =_{def} f(f^n(t))$. For instance, $\texttt{succ}^3(\texttt{zero}) = \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))$.

As often as possible, we try to split the set of operations into *constructors*, which determine the set of possible values of a sort, and *non-constructors*, which are defined functions specified using algebraic equations or term rewrite rules. We define the *constructors of a sort S* to be all constructors that return a result of sort S. For instance, the constructors of natural numbers defined in the Peano style are `zero` and `succ`; all other operations also returning a natural number (e.g., addition, subtraction, multiplication, etc.) are seen as non-constructors.

We define a *ground normal form* to be any well-typed term built using constructors only; consequently, a ground normal form cannot contain free vari-

[2] We prefer using *sort* rather than *type*, which is often given a different meaning.

ables or non-constructors. We define the *domain* of a sort S to be the set, noted $dom(S)$, of all ground normal forms of sort S. We define the *image* of a constructor f to be the set, noted $img(f)$, of all ground normal forms whose top-level constructor is f. For instance, if `nat` is the sort defined by the two constructors [`zero` : → `nat`] and [`succ` : `nat` → `nat`], its domain is $dom(S) = \{\mathtt{succ}^n(\mathtt{zero}) \mid n \in \mathbb{N}\}$ and the images of its constructors are $img(\mathtt{zero}) = \{\mathtt{zero}\}$ and $img(\mathtt{succ}) = \{\mathtt{succ}^n(\mathtt{zero}) \mid n \in \mathbb{N} \setminus \{0\}\}$.

4.2 Semantic denotations

Our goal is to study how mathematical integers can be conveniently represented by sorts, constructors, and algebraic terms. We thus carefully distinguish between *notations* (i.e., terms) and *denotations* (i.e., numbers from $\mathbb{Z}$).

If f is a constructor, we write $[\![f]\!]$ its intended denotation, which we conveniently formulate as a λ-expression, in which all the trivial conversions between sorts or types (i.e., from Peano-like terms to $\mathbb{N}$, or from $\mathbb{N}$ to $\mathbb{Z}$) are implicitly performed. For instance, $[\![\mathtt{zero}]\!] = 0$ and $[\![\mathtt{succ}]\!] = \lambda n.(n+1)$.

Having defined the denotation $[\![f]\!]$ of a constructor f, we extend this notion to define the denotation $[\![t]\!]$ of a ground normal form t by induction on the syntax of terms: $[\![f(t_1, ..., t_n)]\!] =_{def} [\![f]\!]([\![t_1]\!], ..., [\![t_n]\!])$. For instance, $[\![\mathtt{succ(zero)}]\!] = [\![\mathtt{succ}]\!]([\![\mathtt{zero}]\!]) = (\lambda n.(n+1))(0) = 1$.

A necessary condition for a sort S to represent $\mathbb{Z}$ is that $[\![.]\!]$ is a surjection from $dom(\mathrm{S})$ to $\mathbb{Z}$, i.e., $(\forall n \in \mathbb{Z})\ (\exists\, t \in dom(S) \mid [\![t]\!] = n)$, meaning that each integer can be denoted by at least one ground normal form of sort S.

Additionally, we say that the constructors of sort S are *free* if $[\![.]\!]$ is an injection[3] from $dom(\mathrm{S})$ to $\mathbb{Z}$, i.e., $(\forall t_1, t_2 \in dom(S))\ ([\![t_1]\!] = [\![t_2]\!] \Rightarrow t_1 = t_2)$, where $t_1 = t_2$ means the syntactic identity of terms t_1 and t_2. Thus, if the constructors are free, each integer is denoted by exactly one ground normal form of S. This definition remains compatible with the usual, more general definition stating that constructors are free if any two syntactically different ground normal forms cannot be proven equal (or be rewritten one into the other, or both into a common third term).

Let S be a sort intended to represent $\mathbb{Z}$, and $f_1, ..., f_n$ its constructors (with $n \geq 1$). It follows from the above definitions that these constructors are free iff all functions $[\![f_i]\!]$ are injective and the sets of denotations of the images $img(f_1)$, ..., $img(f_n)$ form a partition[4] of $\mathbb{Z}$, i.e., $\mathbb{Z} = \{[\![t]\!] \mid t \in img(f_1)\} \uplus ... \uplus \{[\![t]\!] \mid t \in img(f_n)\}$, where $\uplus$ denotes the disjoint union. Notice that if a sort has a single constructor, this constructor is not necessarily free, as it may be non-injective; various examples will be seen below.

To compare the various definitions of $\mathbb{Z}$ based on algebraic terms, we quantify the complexity of each approach using two natural numbers (m, n) defined as follows: given a sort S that represents $\mathbb{Z}$, n is the number of constructors (noted f_1, ..., f_n) of S, and m is the number of different sorts that occur in the arguments of f_1, ..., f_n; in particular, m is equal to one if sort S is defined directly, without referencing another sort. The number n of constructors adversely im-

[3] and, hence, a bijection.

[4] in particular, are pairwise disjoint.

pacts the conciseness of non-constructor operations having arguments of sort S: a unary function is likely to require n equations or rewrite rules; a binary function is likely to require n^2 equations or rewrite rules, etc. We write $\mathbf{NF}_n^m$ (resp. $\mathbf{F}_n^m$) an approach based on m sorts and n non-free (resp. free) constructors.

5 Definitions of $\mathbb{Z}$ Using Non-Free Constructors

In this section, we review four approaches that quickly come to mind when trying to define integers using algebraic terms. However, these approaches lead to non-free constructors which, we believe, are not optimal. The merit of an approach is inversely proportional to its number of *collisions*, i.e., the maximal number of distinct ground normal forms that can denote the same integer value.

5.1 Approach 2: $\mathbf{NF}_1^2$ – Two Sorts and One Non-Free Constructor

Interestingly, the set-theoretic approach of Sect. 3 can be reformulated in algebraic style by defining a sort `Int` built upon the preexisting sort `Nat` using a single constructor [$\texttt{pair} : \texttt{Nat}, \texttt{Nat} \rightarrow \texttt{Int}$] such that $[\![\texttt{pair}]\!] = \lambda m.\lambda n.(m - n)$. Clearly, this unique constructor is not free: for instance, $[\![\texttt{pair}(\texttt{succ}(\texttt{zero}), \texttt{zero})]\!] = [\![\texttt{pair}(\texttt{succ}(\texttt{succ}(\texttt{zero})), \texttt{succ}(\texttt{zero})]\!] = 1$. More generally, each integer $n \geq 0$ can be represented by an infinite number of terms $\{\texttt{pair}(\texttt{succ}^{k+n}(\texttt{zero}), \texttt{succ}^{k}(\texttt{zero})) \mid k \in \mathbb{N}\}$ and each integer $n \leq 0$ can be represented by an infinite number of terms $\{\texttt{pair}(\texttt{succ}^{k}(\texttt{zero}), \texttt{succ}^{k-n}(\texttt{zero})) \mid k \in \mathbb{N}\}$; the number of collisions is thus $\aleph_0$, which is a most undesirable property.

5.2 Approach 3: $\mathbf{NF}_3^1$ – One Sort and Three Non-Free Constructors

To define a sort `Int` representing $\mathbb{Z}$, an intuitive idea, used in the library[5] of the KIV tool [10, p. 8 and Sect. 5.1 page 42], is to take the Peano system [$\texttt{zero} :\rightarrow \texttt{Int}$] and [$\texttt{succ} : \texttt{Int} \rightarrow \texttt{Int}$] and extend it with a third constructor [$\texttt{pred} : \texttt{Int} \rightarrow \texttt{Int}$] such that $[\![\texttt{pred}]\!] = \lambda n.(n - 1)$, while `zero` and `succ` keep their usual denotations: $[\![\texttt{zero}]\!] = 0$ and $[\![\texttt{succ}]\!] = \lambda n.(n + 1)$. Definitions of integers involving such a predecessor operation have been studied in [33, System SP] [29], and also in [4] [3], where the predecessor operation is a non-constructor. The fact that $[\![\texttt{pred}]\!] = [\![\texttt{succ}]\!]^{-1}$ gives an appealing symmetry, as `pred` and `succ` progress in opposite directions.

Unfortunately, these constructors are not free, the simplest counterexample being $[\![\texttt{pred}(\texttt{succ}(\texttt{zero}))]\!] = [\![\texttt{succ}(\texttt{pred}(\texttt{zero}))]\!] = 0$. Each integer $n \geq 0$ can be represented by an infinite number of terms, e.g., $\{\texttt{pred}^{k}(\texttt{succ}^{n+k}(\texttt{zero})) \mid k \in \mathbb{N}\}$ or, more generally, any combination (in any order) of k applications of `pred` mixed with $(k + n)$ applications of `succ` — a dual remark holds if $n \leq 0$. The number of collisions is thus $\aleph_0$.

[5] `https://swt.informatik.uni-augsburg.de/swt/projects/lib/basic/specs/int-basic1/export/unit.xml`

5.3 Approach 4: $\mathbf{NF}_1^3$ – Three Sorts and One Non-Free Constructor

Another approach, used in the type library of PSF [30, Sect. 8] [21, Sect. 2.5], is based on the idea that an integer is a natural with a sign. Assuming the existence of a sort `Sign` with two free constructors $[\texttt{plus} :\to \texttt{Sign}]$ and $[\texttt{minus} :\to \texttt{Sign}]$, the sort `Int` can be defined using a constructor $[\texttt{pair} : \texttt{Sign}, \texttt{Nat} \to \texttt{Int}]$ such that $[\![\texttt{pair}]\!] = \lambda s.\lambda n.(\text{if } s = \texttt{plus} \text{ then } n \text{ else } - n)$.

This unique constructor is not free, because of the collision $[\![\texttt{pair}(\texttt{plus}, \texttt{zero})]\!] = [\![\texttt{pair}(\texttt{minus}, \texttt{zero})]\!] = 0$, a situation that we will refer to as the *double-zero issue*[6]. This is the only collision in this approach.

5.4 Approach 5: $\mathbf{NF}_2^2$ – Two Sorts and Two Non-Free Constructors

A variant of the previous approach $\mathbf{NF}_1^3$ of Sect. 5.3 does not rely on the sort `Sign` but uses instead two constructors $[\texttt{pos} : \texttt{Nat} \to \texttt{Int}]$ and $[\texttt{neg} : \texttt{Nat} \to \texttt{Int}]$ such that $[\![\texttt{pos}]\!] = \lambda n.n$ and $[\![\texttt{neg}]\!] = \lambda n.(-n)$. This approach is used in TLA+ [18, Sect. 18.4 page 347], where *Int* is defined as $\mathit{Nat} \cup \{\mathit{Zero} - n \mid n \in \mathit{Nat}\}$. Other variants in which `neg` is the unary-minus operator $[\texttt{neg} : \texttt{Int} \to \texttt{Int}]$ have been studied in, e.g., [7] [32] [17] [34] [4] [3].

Again, these constructors are not free, since $[\![\texttt{pos}(\texttt{zero})]\!] = [\![\texttt{neg}(\texttt{zero})]\!] = 0$. Some tool designers are aware of this double-zero issue and propose to address it in various ways:

- *Dependent types:* One may wish to rule out undesirable terms such as $\texttt{pair}(\texttt{minus}, \texttt{zero})$ or $\texttt{neg}(\texttt{zero})$. This is the approach followed in the SMT-LIB standard [1, Fig. 3.3 page 34, and page 35], which states: "*The set of values for the* `Int` *sort consists of all numerals and all terms of the form* $(-n)$ *where* n *is a numeral other than 0*". Prohibiting the algebraically-closed term (-0) can be done trivially, at the level of syntax, but things are far less easy with general terms of the form $(-e)$, where e is an expression containing free variables and/or arbitrary user-defined functions. Deciding whether such terms belong to `Int` is equivalent to answer the question $e = 0$, which is undecidable in the general case and would anyway require involved proofs in each particular case. Alternative approaches with efficient decision procedures are thus preferable.

- *Equations relating constructors:* Other approaches use equations or rewrite rules in order to formalize the relations that may exist between constructors. For instance, [4] and [3] handle the double-zero issue by adding two equations $-0 = 0$ and $- - n = n$, which eliminate unwanted terms by bringing them under a suitable normal form. However, this approach weakens the difference between constructors and non-constructors and often raises termination issues. There exists a classical technique (see [27, Sect. 3], [11, Sect. 3.3], etc.) to eliminate non-free constructors by replacing each of them with two distinct operations: a free constructor

[6] Notice that the IEEE 754 standard for floating-point numbers also defines two zeros, -0.0 and +0.0, which are equivalent in most cases but can still be distinguished, e.g., using the `signbit` primitive.

and a non-constructor. Unfortunately, such a dissociation of roles does not give exploitable results when applied to the construction of $\mathbb{Z}$.

- *Subtypes:* Another approach relies on subtyping and operator overloading to avoid the double-zero issue. For instance, the predefined type library of Maude [6, pages 45–46, 116, and 248–251] makes plain use of order-sorted specifications to define integers as follows: (i) it first defines the sort `Nat` of naturals, together with the subsort `NzNat` of `Nat`, which contains all naturals different from zero; (ii) it then defines the sort `Int` of integers, together with the subsort `NzInt` of `Int`, which contains all integers different from zero and is also a supersort of `NzNat`; (iii) a "unary-minus" constructor $[- : \texttt{NzNat} \rightarrow \texttt{NzInt}]$ is defined, such that $[\![-]\!] = \lambda n.(-n)$; (iv) this constructor is extended to integer sorts by introducing two subsort-overloaded non-constructors $[- : \texttt{NzInt} \rightarrow \texttt{NzInt}]$ and $[- : \texttt{Int} \rightarrow \texttt{Int}]$; (v) these non-constructors are defined by adding two equations $-0 = 0$ and $- - n = n$. This approach is the only one in which values of sort `Nat` are also values of sort `Int`; such implicit type conversion closely reflects the mathematical fact that $\mathbb{N} \subset \mathbb{Z}$, but might become a nuisance when turning an algebraic specification with arbitrary large numbers into a concrete implementation written in a programming language with bounded numbers (e.g., the C language with its `int` and `unsigned int` types): to avoid silent, yet unsafe numeric overflows, the programmer will have to detect all implicit type conversions in the formal specification and make them explicit in the program.

These various approaches, even if correct, are heavy. Lighter approaches using only the same basic concepts as for the Peano-style definition of naturals are desirable. All in one, we believe that definitions based on non-free constructors are sub-optimal. Therefore, in the next section, we investigate simpler approaches genuinely based on free constructors.

6 Definitions of $\mathbb{Z}$ Using Free Constructors

Free constructors obviously lead to simpler mathematics. On the practical side too, free constructors are desirable, for at least three reasons: (i) terms defined using free constructors have a unique representation, so that, in principle, no memory bit is wasted due to the existence of multiple representations of the same value; (ii) because each term has a unique representation, comparison of values relies upon syntactic identity, which can be efficiently implemented using bit-string comparison and/or hashing; no extra computation is required to compare multiple representations of the same value or to bring terms under a canonical form first; (iii) increasingly many computer languages (e.g., functional, object-oriented, etc.) are supporting pattern matching with free constructors, whereas the number of tools (e.g., term rewrite engines, tools for algebraic specifications, etc.) that can handle non-free constructors is shrinking, as many tools are no longer maintained[7].

[7] See `http://rewriting.loria.fr/systems.html` to learn about such halted tools.

6.1 Approach 6: $\mathbf{F}_2^3$ – Three Sorts and Two Free Constructors

The approaches $\mathbf{NF}_1^3$ of Sect. 5.3 and $\mathbf{NF}_2^2$ of Sect. 5.4 (i.e., defining an integer as the combination of a sign and a natural) can be reused and adapted to a free-constructor setting. This requires to introduce a new sort and to break the symmetry between the negative and positive cases, so as to forbid, by means of statically-decidable type checking, one of the two zero values.

For instance, mCRL2 [12, Appendices B.2, B.3, B.4, and D.5] has three predefined sorts: `Pos`, which denotes $\mathbb{N} \setminus \{0\}$ and is encoded in binary form, `Nat`, which denotes $\mathbb{N}$ and is defined in Peano style using two free constructors [`@c0` $:\rightarrow$ `Nat`] and [`@cNat` : `Pos` $\rightarrow$ `Nat`], and `Int`, which denotes $\mathbb{Z}$ and is also defined using two free constructors [`@cInt` : `Nat` $\rightarrow$ `Int`] and [`@cNeg` : `Pos` $\rightarrow$ `Int`] such that $[\![\texttt{@cInt}]\!] = \lambda n.n$ and $[\![\texttt{@cNeg}]\!] = \lambda n.(-n)$. The elements of $\mathbb{Z}$ are thus encoded as follows: $n \geq 0 \mapsto \texttt{@cInt}(n)$ and $n < 0 \mapsto \texttt{@cNeg}(-n)$.

As with Maude, the double-zero issue is avoided by a deliberate dissymmetry between both constructors `@cInt` and `@cNeg`, whose arguments have sorts `Nat` and `Pos`, respectively. But, contrary to Maude, `Nat` and `Pos` are distinct sorts, not subtypes; this requires explicit type conversions and may create confusion for users, who must carefully distinguish between natural and positive values.

6.2 Approach 7: $\mathbf{F}_3^2$ – Two Sorts and Three Free Constructors

A different approach can be found in the standard library of the Coq theorem prover[8] (see [26] for confirmation). To construct natural numbers, this library defines a sort[9] `nat` built in Peano style using two constructors [`0` $:\rightarrow$ `nat`] and [`S` : `nat` $\rightarrow$ `nat`]. In an alternative approach, the Coq library also contains a sort `positive` that represents $\mathbb{N} \setminus \{0\}$, i.e., all natural numbers greater or equal to one — such numbers are encoded in binary form, as unbounded strings of bits inductively defined by three free constructors [`xH` $:\rightarrow$ `positive`], [`xO` : `positive` $\rightarrow$ `positive`], and [`xI` : `positive` $\rightarrow$ `positive`]. Finally, Coq defines a sort `N` (also intended to represent $\mathbb{N}$) as the union of all `positive` values and of a constant `N0` denoting 0; this is done using two constructors [`Npos` : `positive` $\rightarrow$ `N`] and [`N0` $:\rightarrow$ `N`].

To construct integer numbers, the Coq library defines a sort `Z` that only uses the sort `positive`, but neither `nat` nor `N`. The sort `Z` is built using three free constructors [`Z0` $:\rightarrow$ `Z`], [`ZPos` : `positive` $\rightarrow$ `Z`], and [`ZNeg` : `positive` $\rightarrow$ `Z`] such that $[\![\texttt{Z0}]\!] = 0$, $[\![\texttt{ZPos}]\!] = \lambda n.n$, and $[\![\texttt{ZNeg}]\!] = \lambda n.(-n)$. The elements of $\mathbb{Z}$ are thus encoded as follows: $0 \mapsto \texttt{Z0}$, $n > 0 \mapsto \texttt{ZPos}(n)$, and $n < 0 \mapsto \texttt{ZNeg}(-n)$.

Notice that this approach could be slightly adapted to define `Z` using sort `N` (or `nat`) rather than `positive`. In such case, the three constructors would become [`Z0` $:\rightarrow$ `Z`], [`ZPos` : `N` $\rightarrow$ `Z`], and [`ZNeg` : `N` $\rightarrow$ `Z`] such that $[\![\texttt{Z0}]\!] = 0$, $[\![\texttt{ZPos}]\!] = \lambda n.(n{+}1)$, and $[\![\texttt{ZNeg}]\!] = \lambda n.(-n{-}1)$. The elements of $\mathbb{Z}$ would be thus encoded as follows: $0 \mapsto \texttt{Z0}$, $n > 0 \mapsto \texttt{ZPos}(n-1)$, and $n < 0 \mapsto \texttt{ZNeg}(-n-1)$.

Contrary to some aforementioned approaches, the Coq library handles negative and positive numbers symmetrically. Even if this approach is modified (as

[8] `http://coq.inria.fr/library`

[9] i.e., a *datatype* in the terminology of Coq.

explained in the previous paragraph) to use only two sorts, it still relies upon three constructors, which increases the length of definitions and proofs for most operations on integers; in particular, the usual binary operators are likely to require nine equations (rather than four when only two constructors are used).

6.3 Approach 8: $\mathbf{F}_2^2$ – Two Sorts and Two Free Constructors

We have seen so far two definitions of $\mathbb{Z}$ based on free constructors: one with three sorts and two constructors (i.e., $\mathbf{F}_2^3$ of Sect. 6.1), another one with two sorts and three constructors (i.e., $\mathbf{F}_3^2$ of Sect. 6.2). At this point, a natural question is: *is there a simpler definition with two sorts and two constructors only?*

The answer is: yes. Such a solution was found in 1996 when trying to extend the standard library of LOTOS with signed integers, and it has been distributed as part of the CADP toolbox since February 1997.

This approach uses two sorts `Nat`, which denotes $\mathbb{N}$, and `Int`, which denotes $\mathbb{Z}$. The sort `Int` is built from `Nat`, without the need for a third sort, using two free constructors $[\texttt{pos} : \texttt{Nat} \to \texttt{Int}]$ and $[\texttt{neg} : \texttt{Nat} \to \texttt{Int}]$ such that $[\![\texttt{pos}]\!] = \lambda n.n$ and $[\![\texttt{neg}]\!] = \lambda n.(-n-1)$. The elements of $\mathbb{Z}$ are thus encoded as follows: $n \geq 0 \mapsto \texttt{pos}(n)$ and $n < 0 \mapsto \texttt{neg}(-n-1)$.

Although this approach does not handle negative and positive numbers symmetrically, it enjoys a nice symmetry property, as the denotations of constructors are both involutive functions, i.e., $[\![\texttt{pos}]\!] = [\![\texttt{pos}]\!]^{-1}$ and $[\![\texttt{neg}]\!] = [\![\texttt{neg}]\!]^{-1}$, or also $[\![\texttt{pos}]\!]([\![\texttt{pos}]\!](n)) = n$ and $[\![\texttt{neg}]\!]([\![\texttt{neg}]\!](n)) = n$, which seems a counterpart of the algebraical identities $+(+n) = n$ and $-(-n) = n$.

The constructor pair (`pos`, `neg`) is unique in this respect, and there is no simpler solution: consider the set Φ of involutive functions φ over $\mathbb{Z}$, i.e., $(\forall n)$ $(\varphi(\varphi(n)) = n)$; consider the "simple" elements of Φ, namely affine functions such that $(\forall n)$ $(\varphi(n) =_{def} an + b)$, where a and b are constants; the "simple" involutive solutions are either $\lambda n.n$ or all functions of the form $\lambda n.(-n+b)$; among the restrictions to $\mathbb{N}$ of these solutions, the only pair of free constructors is `pos` (which corresponds to $a = 1$ and $b = 0$) and `neg` (which corresponds to $a = -1$ and $b = -1$), thus ensuring that $\mathbb{Z} = \{[\![\texttt{pos}]\!](n) \mid n \in \mathbb{N}\} \uplus \{[\![\texttt{neg}]\!](n) \mid n \in \mathbb{N}\}$.

It is worth noting that this approach supports straightforward induction on integers, which is lacking in some other approaches — see, e.g., [24, Sect. 8.4 and 8.4.3] for a discussion concerning Isabelle/HOL. Using the `pos` and `neg` constructors, induction on integers can be achieved by two inductions on naturals: firstly, one proves that the property P hold for $\texttt{pos}(0)$ and that, if P holds for $\texttt{pos}(n)$, it also holds for $\texttt{pos}(n+1)$; secondly, one proves that P holds for $\texttt{neg}(0)$ and that, if P holds for $\texttt{neg}(n)$, it also holds for $\texttt{neg}(n+1)$.

6.4 Approach 9: $\mathbf{F}_1^2$ – Two Sorts and One Constructor

Given that $\mathbb{Z}$ can be defined as $\mathbb{Z} =_{def} (\mathbb{N} \times \mathbb{N}) / \sim$ and that there exist bijections from $\mathbb{N}^2$ to $\mathbb{N}$ (e.g., diagonal enumeration), it is possible to define the sort `int` using a single constructor $[\texttt{map} : \texttt{nat} \to \texttt{int}]$ that is a bijection from $\mathbb{N}$ to $\mathbb{Z}$. There are various choices for `map`; the simplest definition is likely to be the following one: $[\![\texttt{map}]\!] =_{def} \lambda n.(\text{if } \texttt{even}(n) \text{ then } n/2 \text{ else } -(n+1)/2)$, where $[\texttt{even} : \texttt{nat} \rightarrow \texttt{bool}]$ is the predicate characterizing even natural numbers. This definition gives, as n increases, the following sequence of values for $\texttt{map}(n)$: 0, -1, 1, -2, 2, -3, 3, -4, 4, etc. The elements of $\mathbb{Z}$ are thus encoded as follows: $n \geq 0 \mapsto \texttt{map}(2n)$ and $n < 0 \mapsto \texttt{map}(-2n-1)$.

This approach $\mathbf{F}_1^2$ is related to the approach $\mathbf{F}_2^2$ of Sect. 6.3 by two identities: $\texttt{pos}(n) = \texttt{map}(2n)$ and $\texttt{neg}(n) = \texttt{map}(2n+1)$. But, even if having a single constructor map is a form of minimality, it does not make the definitions of the usual non-constructors (+, <, etc.) more concise than using the two constructors of approach $\mathbf{F}_2^2$. Indeed, most definitions still need to distinguish two cases, depending whether some argument is odd or even; approach $\mathbf{F}_1^2$ checks this using conditional equations, whereas approach $\mathbf{F}_2^2$ uses pattern matching. For instance, the incrementation function $[\texttt{incr} : \texttt{int} \rightarrow \texttt{int}]$ requires two conditional equations:

```
incr (map (x)) = map (x + 2)    if even (x) = true
incr (map (x)) = map (x - 2)    if even (x) = false
```

Such a systematic reliance on conditional equations, parity tests, and divisions by two has, at least, three drawbacks: (i) the definitions are neither elegant nor intuitive; (ii) reasoning by induction is not easy; (iii) direct implementation in algebraic or functional languages is not efficient, since parity tests cost $O(n)$ in time (before deciding if a number is odd or even, one needs to visit all its cons subterms).

6.5 Approach 10: $\mathbf{F}_3^1$ – One Sort and Three Free Constructors

After the WADT'2016 presentation in Gregynog, Lutz Schröder suggested to the author yet another approach, in which the sort int is defined using three free constructors: $[\texttt{zero} :\rightarrow \texttt{int}]$, $[\texttt{nego} :\rightarrow \texttt{int}]$[10], and $[\texttt{succ} : \texttt{int} \rightarrow \texttt{int}]$ such that $[\![\texttt{zero}]\!] = 0$, $[\![\texttt{nego}]\!] = -1$, and $[\![\texttt{succ}]\!] = \lambda n.(\text{if } n \geq 0 \text{ then } n+1 \text{ else } n-1)$. The elements of $\mathbb{Z}$ are thus encoded as follows: $0 \mapsto \texttt{zero}$, $n > 0 \mapsto \texttt{succ}^n(\texttt{zero})$, $-1 \mapsto \texttt{nego}$, and $n < -1 \mapsto \texttt{succ}^{-n-1}(\texttt{nego})$.

At first sight, this approach looks truly beautiful, as it is the simplest possible extension of the Peano system, to which only one constructor nego of null arity is added. Morever, the definition of sort int is self-contained and does not rely on the existence of a sort nat. A similar approach appears in [8, Sect. 4], where integers are written in base 2 starting from two constants 0 and -1.

However, the definitions of non-constructors can be algorithmically involved with this approach, due to the dual nature of succ (which means either incrementation on positive numbers or decrementation on negative numbers, and might thus complicate induction proofs) and because the sign of a number cannot be immediately determined without traversing all succ constructors until a terminal constant (zero or nego) is reached. Some operations are nevertheless easy to express; this is the case, for instance, of the two predicates $[\texttt{isneg} : \texttt{int} \rightarrow \texttt{bool}]$ and $[\texttt{ispos} : \texttt{int} \rightarrow \texttt{bool}]$ that check whether an integer is strictly negative or positive-or-null:

[10] We name so this constructor, as a shorthand for *NEGative One*.

```
isneg (zero) = false                ispos (zero) = true
isneg (nego) = true                 ispos (nego) = false
isneg (succ (x)) = isneg (x)        ispos (succ (x)) = ispos (x)
```

But other operations, even the simplest ones, are much less intuitive. For instance, incrementation [$\texttt{incr} : \texttt{int} \to \texttt{int}$] is tricky because it requires either to insert one `succ` if the argument is positive or to delete one `succ` if the argument is negative. We believe that this cannot be done without introducing an auxiliary operation [$\texttt{buff} : \texttt{int} \to \texttt{int}$] that keeps one `succ` in a virtual buffer until the terminal constant gets known, an information that is required to take an insertion-or-deletion decision:

```
incr (zero) = succ (zero)           buff (zero) = succ (succ (zero))
incr (nego) = zero                  buff (nego) = nego
incr (succ (x)) = buff (x)          buff (succ (x)) = succ (buff (x))
```

To avoid, such intricacies, one can resort to conditional equations, the premises of which use the aforementioned `isneg` and `ispos` predicates. This way, incrementation could be defined more concisely:

```
incr (x) = succ (x)    if ispos (x)
incr (nego) = zero
incr (succ (x)) = x    if isneg (x)
```

Such equations would be very similar to those of approach $\mathbf{F}_2^2$ of Sect. 6.3. Actually, both approaches are related by the two identities $\texttt{pos}(\texttt{succ}^n(\texttt{zero})) = \texttt{succ}^n(\texttt{zero}) = n$ and $\texttt{neg}(\texttt{succ}^n(\texttt{zero})) = \texttt{succ}^n(\texttt{nego}) = -n - 1$. The approach $\mathbf{F}_2^2$ is yet simpler: it uses normal equations rather than conditional ones, and determines the sign of a number in time $O(1)$ by pattern matching on the top-level constructor of the term, rather than in time $O(n)$ by invoking the predicates `isneg` and `ispos` that visit all subterms to reach an innermost terminal constant.

As a final remark, any sort S defined by $(k + 1)$ constructors consisting of k constants $[f_i :\to S]_{i \in \{0,...,k-1\}}$ and one successor function $[\texttt{succ} : S \to S]$[11] also represents $\mathbb{N}$, which can be seen by taking $(\forall i \in \{0, ..., k - 1\})$ $([\![f_i]\!] =_{\mathit{def}} i)$ and $[\![\texttt{succ}]\!] =_{\mathit{def}} \lambda n.(n + k)$; under these definitions, the constructors of S are free.

7 Conclusion

We have shown that the standard definition of signed integers found in mathematical textbooks (namely, $\mathbb{Z} =_{\mathit{def}} (\mathbb{N} \times \mathbb{N}) / \sim$) is not well-adapted to the needs of computer science. It has been argued, e.g. in [22, p. 86], that this standard definition, although less straightforward and natural than a term-algebra approach[12], should nevertheless be preferred because it leads to shorter definitions and proofs by avoiding subdivision into a large number of cases. This

[11] This definition generalizes the Peano system (for $k = 1$) and the approach presented in this section (for $k = 2$).

[12] [22] suggests here the approach $\mathbf{NF}_3^1$ of Sect. 5.2 based on three non-free constructors.

argument predates the computer era: it might have been valid when all proofs had to be done manually, but is less relevant today, as interactive or automated theorem provers play an ever-increasing role and handle case disjunctions much better than humans. Moreover, it is unsure that the definitions of arithmetic and relational operators are significantly longer when integers are defined in Peano style with only a few constructors.

In computer science, there is consensus to specify naturals using the Peano constructors `zero` and `succ`, but no consensus at all on how integers should be specified. Leaving aside languages that assume the existence of predefined integers, we reviewed ten different ways of defining integers formally: the set-theoretical approach, four approaches based on non-free constructors ($\mathbf{NF}_1^2$, $\mathbf{NF}_3^1$, $\mathbf{NF}_1^3$, and $\mathbf{NF}_2^2$), and five approaches based on free constructors ($\mathbf{F}_2^3$, $\mathbf{F}_3^2$, $\mathbf{F}_2^2$, $\mathbf{F}_1^2$, and $\mathbf{F}_3^1$).

Such a diversity is enjoyable, but too many diverging approaches can be a nuisance. Given the practical usefulness of integers, a common approach would be desirable, so as to ease tool interoperability and enable formal specifications and proofs to be reused. In this respect, the approach $\mathbf{F}_2^2$ of Sect. 6.3 would be the best candidate: it is simple, enjoys elegant mathematical properties, allows induction over integers, leads to concise definitions for the usual operations on integers, and has been implemented in the CADP toolbox since 1997.

Finally, considering the alternative approaches that propose more efficient representations for unsigned naturals than the Peano-style unary representation, $\mathbf{F}_2^2$ is orthogonal to some of these approaches, e.g., [2], and could be combined with them to provide efficient representations for signed integers as well.

Acknowledgements

The author is grateful to Radu Mateescu and Mihaela Sighireanu who, in 1997 at INRIA Grenoble, helped writing LOTOS algebraic equations to show that the approach $\mathbf{F}_2^2$ was feasible. Acknowledgements are also due to Holger Hermanns and Gilles Nies (Saarland University), and to Jan Friso Groote (Technical University of Eindhoven) for their valuable remarks. Jan Bergstra and Alban Ponse (University of Amsterdam) provided the author with recent bibliographic references. This article benefited from the lively WADT'2016 discussions in Gregynog, especially with Alexander Knapp (Augsburg University), who mentioned KIV, with Narciso Marti-Oliet (Conplutence University of Madrid), who gave informed comments about Maude, and with Lutz Schröder (Friedrich-Alexander-Universität Erlangen-Nürnberg), who suggested the approach $\mathbf{F}_3^1$. Frédéric Lang, Wendelin Serwe, and Hugues Evrard (INRIA Grenoble) proofread earlier versions of this article.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard – Version 2.5. 93 pages, June 2015.

[2] Jan A. Bergstra. Four Complete Datatype Defining Rewrite Systems for an Abstract Datatype of Natural Numbers (version 3). Report TCS1407v3, University of Amsterdam (UvA), The Netherlands, June 2014.

[3] Jan A. Bergstra and Alban Ponse. Datatype Defining Rewrite Systems for the Ring of Integers, and for Natural and Integer Arithmetic in Unary View. *The Computing Research Repository (CoRR)*, abs/1608.06212, August 2016.

[4] Jan A. Bergstra and Alban Ponse. Three Datatype Defining Rewrite Systems for Datatypes of Integers Each Extending a Datatype of Naturals (version 3). Report TCS1409v3, University of Amsterdam (UvA), The Netherlands, February 2016. Also available at `http://arxiv.org/abs/1406.3280`.

[5] Gérard Berry. *The Esterel V5 Language Primer – Version v5_91*. INRIA Sophia-Antipolis, France, June 2000.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] Dave Cohen and Phil Watson. An Efficient Representation of Arithmetic for Term Rewriting. In Ronald V. Book, editor, *Proceedings of 4th International Conference on Rewriting Techniques and Applications (RTA'91), Como, Italy*, volume 488 of *LNCS*, pages 240–251. Springer, April 1991.

[8] Evelyne Contejean, Claude Marché, and Landy Rabehasaina. Rewrite Systems for Natural, Integral, and Rational Arithmetic. In Hubert Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications (RTA'97), Sitges, Spain*, volume 1232 of *LNCS*, pages 98–112. Springer, June 1997.

[9] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[10] Gidon Ernst, Dominik Haneberg, Jörg Pfähler, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Bogdan Tofan. *A Practical Course on KIV*. Germany, 2015.

[11] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[12] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[13] Bernard Houssais. *The Synchronous Programming Language SIGNAL – A Tutorial*. INRIA Rennes, France, September 2004.

[14] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[15] ISO/IEC. Vienna Development Method – Specification Language – Part 1: Base Language. Internation Standard 13817-1:1996, International Organization for Standardization – Programming Languages, their Environments and System Software Interfaces, Geneva, December 1995.

[16] ISO/IEC. Z Formal Specification Notation – Syntax, Type System and Semantics. International Standard 13568:2002, International Organization for Standardization – Information Technology, Geneva, July 2002.

[17] J. Richard Kennaway. Complete Term Rewrite Systems for Decimal Arithmetic and Other Total Recursive Functions. Proceedings of the 2nd International Workshop on Termination, La Bresse, France, May 1995.

[18] Leslie Lamport. *Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[19] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. VDM-10 Language Manual. Technical Report TR-2010-06, Overture – Open-source Tools for Formal Modelling, April 2010.

[20] Thierry Lecomte. *B Language Reference Manual (Version 1.8.7).* ClearSy System Engineering, Aix-en-Provence, France, 2010.

[21] Sjouke Mauw and J.C. Mulder. A PSF Library of Data Types. In Mark G. J. van den Brand, Arie van Deursen, T. B. Dinesh, Jasper Kamperman, and Eelco Visser, editors, *Proceedings of ASF+SDF'95: A Workshop on Generating Tools From Algebraic Specifications*, May 1995. Technical Report P9504-5, Programming Research Group, University of Amsterdam, The Netherlands.

[22] Elliott Mendelson. *Number Systems and the Foundations of Analysis.* Dover Books on Mathematics Series. Courier Corporation, 1973.

[23] Peter D. Mosses. *CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science.* Springer, 2004.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic. (Early Version: Lecture Notes in Computer Science, vol. 2283, Springer, 2002), February 2016.

[25] Sam Owre and Natarajan Shankar. The PVS Prelude Library. Technical Report CSL-03-01, SRI International, Computer Science Laboratory, March 2003.

[26] Christine Paulin-Mohring. Basics of COQ. Lecture Notes of the LASER 2011 Summer School, September 2011.

[27] Simon J. Thompson. Laws in Miranda. In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the ACM Conference on LISP and Functional Programming (LFP'86), Cambridge, Massachusetts, USA*, pages 1–12, 1986.

[28] University of Cambridge and Technische Universität München. Isabelle/HOL – Higher-Order Logic. February 2016.

[29] Luca van der Kamp. A Term Rewrite System for Decimal Integer Arithmetic. Bachelor Informatica, University of Amsterdam (UvA), The Netherlands, June 2016.

[30] Jos J. van Wamel. A Library for PSF. Report PR9301, Programming Research Group, University of Amsterdam, The Netherlands, 1993.

[31] Verimag. *Lustre Language Reference Manual – Versions 3, 3+, 4, 5*. Grenoble, France, September 1997.

[32] Humphrey Robert Walters. A Complete Term Rewriting System for Decimal Integer Arithmetic. Technical Report CS-R9435, CWI, Amsterdam, The Netherlands, 1994.

[33] Humphrey Robert Walters and Hans Zantema. Rewrite Systems for Integer Arithmetic. In Jieh Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA'95), Kaiserslautern, Germany*, volume 914 of *LNCS*, pages 324–338. Springer, April 1995.

[34] Hans Zantema. Basic Arithmetic by Rewriting and its Complexity. Technical University of Eindhoven, The Netherlands, December 2003.