On the Most Suitable Axiomatization of Signed Integers

Hubert Garavel

Inria Grenoble – LIG

Université Grenoble Alpes

Saarland University

http://convecs.inria.fr

Motivation

Taking numbers for granted

- Some formal methods do not define numbers
- They assume numbers pre-exist as "basic types"
  - B
    assumes $\mathbb{Z} \supset \mathbb{N} \supset \mathbb{N}_1$ (with MININT and MAXINT)
  - PVS
    assumes number $\supset$ real $\supset$ rational $\supset$ integer $\supset$ natural
  - VDM
    assumes real $\supset$ rat $\supset$ int $\supset$ nat $\supset$ nat1
  - Z
    assumes $\mathbb{Z} \supset \mathbb{N} \supset \mathbb{N}_1$

What is wrong with this?

- Actually, these formal methods are closer to programming languages:
  - they borrow the FORTRAN concept of basic types
  - closer to programming than to mathematics

- Properties involving numbers cannot be proven within these formal methods:
  - number-specific theories need to be imported
  - a unified framework including numbers is preferable

This talk

- Focus on formal methods that define numbers
- Focus on $\mathbb{N}$ and $\mathbb{Z}$
  - we consider arbitrary large numbers
  - this excludes "machine integers" and "modular arithmetics"
- Defining $\mathbb{N}$ is easy (note: $0 \in \mathbb{N}$)
- Defining $\mathbb{Z}$ is not trivial:
  - we compare the techniques used to define $\mathbb{Z}$
  - we suggest a "most suitable" approach
  - criteria: elegance, implementability, concisess

Building ℕ: two approaches

Approach 1: set theory

- Zermelo-Fraenkel and Von Neumann construct $\mathbb{N}$ as follows:
  - $0 := \emptyset$
  - $1 := 0 \cup \{0\} = \{0\} = \{\emptyset\}$
  - $2 := 1 \cup \{1\} = \{0, 1\} = \{\emptyset, \{\emptyset\}\}$
  - $3 := 2 \cup \{2\} = \{0, 1, 2\} = \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}$
  - etc. -- $n{+}1 := n \cup \{n\} = \{0, 1, \ldots, n\}$
- No formal method seems to use this approach (even those based on set theory)

Approach 2: algebraic terms

- Peano defines natural numbers using 5 axioms

- This amounts to having 2 constructors:

$$0 : \rightarrow \text{Nat}$$
$$\text{succ} : \text{Nat} \rightarrow \text{Nat}$$

- Many formal methods define $\mathbb{N}$ this way:

  CASL, Coq, Isabelle/HOL,
  LOTOS, Maude, mCRL2, etc.

Term rewrite systems: Bool

SORTS

bool

CONS

false : -> bool

true : -> bool

OPNS

not : bool -> bool

and : bool bool -> bool

VARS

b : bool

RULES

not (true) -> false

not (false) -> true

and (false, b) -> false

and (true, b) -> b

Term rewrite systems: Nat (1/3)

SORTS

nat

CONS

zero : -> nat

succ : nat -> nat

OPNS

even : nat -> bool

odd : nat -> bool

eq : nat nat -> bool

lt : nat nat -> bool

pred : nat -> nat

add : nat nat -> nat

sub : nat nat -> nat

mult : nat nat -> nat

div : nat nat -> nat

mod : nat nat -> nat

VARS

m n p q : nat

RULES

% pred (zero) is undefined

pred (succ (n)) -> n

Term rewrite systems: Nat (2/3)

even (zero) -> true

even (succ (n)) -> odd (n)

eq (zero, zero) -> true

eq (zero, succ (n)) -> false

eq (succ (n), zero) -> false

eq (succ (m), succ (n)) -> eq (m, n)

add (m, zero) -> m

add (m, succ (n)) ->
add (succ (m), n)

odd (zero) -> false

odd (succ (n)) -> even (n)

lt (zero, zero) -> false

lt (zero, succ (n)) -> true

lt (succ (n), zero) -> false

lt (succ (m), succ (n)) -> lt (m, n)

% sub (m, n) *undefined if* lt (m, n)

sub (m, zero) -> m

sub (succ (m), succ (n)) ->
sub (m, n)

Term rewrite systems: Nat (3/3)

mult (m, zero) -> zero

mult (m, succ (n)) -> add (m, mult (m, n))

% div (m, zero) *and* mod (m, zero) *are undefined*

if and (not (eq (n, zero)), lt (m, n)) -> true **then**

div (m, n) -> zero

mod (m, n) -> m

if and (not (eq (n, zero)), not (lt (m, n))) -> true **then**

div (m, n) -> succ (div (sub (m, n), n))

mod (m, n) -> mod (sub (m, n), n)

Building $\mathbb{Z}$ using set theory

Approach #1: Set-theoretic definition of $\mathbb{Z}$

- In mathematical textbooks, $\mathbb{Z}$ is defined as:

  $$\mathbb{Z} = \mathbb{N} \times \mathbb{N} \,/ \sim$$

  where $\sim$ is the equivalence relation such that:

  $$(m, n) \sim (m', n') \Leftrightarrow m + n' = m' + n$$

- Advantages:
  - standard approach in mathematics
  - similar to the definition of rational numbers:
    $\mathbb{Q} = \mathbb{Z} \times \mathbb{Z}^*/\sim$ where $(m, n) \sim (m', n') \Leftrightarrow m.n' = m'.n$
- Approach adopted by CASL and Isabelle/HOL

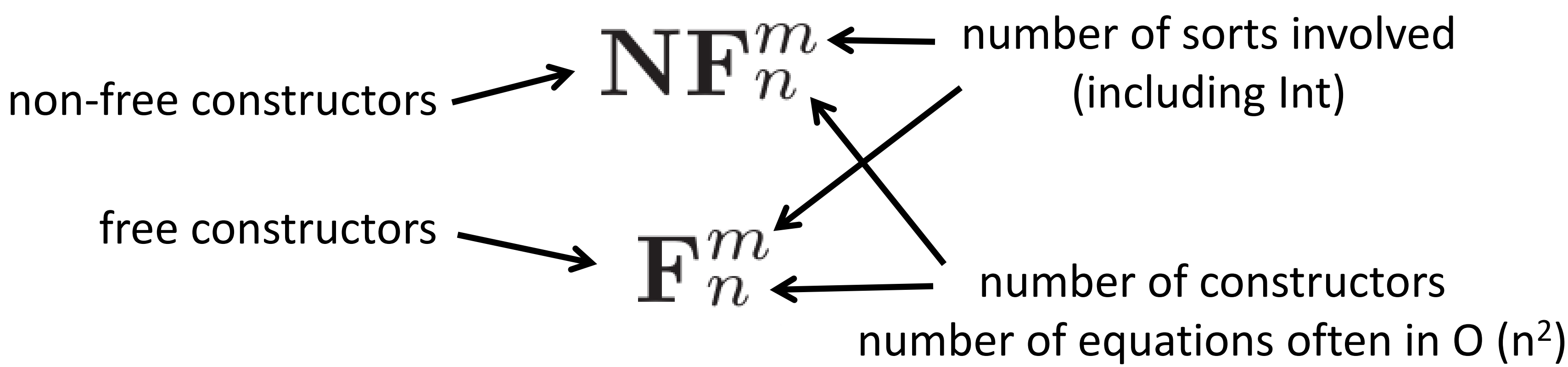
5 drawbacks (wrt computer science)

- Heavy concepts: cartesian product and quotient set
- Integers defined very differently from naturals
- Against the intuition:
  - one needs a half-line towards $-\infty$
  - instead; one builds a surface that is then projected
- Forbids induction proofs over integers [cf. Isabelle]
- Computationally expensive:
  - waste of memory bits: each Int costs two Nats
  - waste of CPU time: no structural equality $\Rightarrow$ comparing two Ints costs more than comparing pairs of Nats (normalization of terms may be suitable)

Building $\mathbb{Z}$ using algebraic terms

Goal - Proposed taxonomy

- Can we find Peano-like definitions for $\mathbb{Z}$?
  - using only basic notions (sorts, operations, equations)
- How to compare these various definitions?
  - By the freeness/non-freeness of their constructors
  - By the number *m* of their sorts
  - By the number *n* of their constructors



Terms and denotations

- Let [.] be the denotation: algebraic term $\rightarrow \mathbb{Z}$

- Clearly, [.] should be **surjective**
  so that all integers can be denoted by some term

- Do we want [.] to be **injective** too (ie., bijective)?
  - if [.] **not injective** : **non-free constructors**
    there exist at least two ground terms that are syntactically different but can be proven equal
  - if [.] **injective** : **free constructors**

Free vs non-free constructors

- Software tools usually prefer free constructors (few tools implement non-free constructors)

- One can always eliminate non-free constructors by splitting each of them into a pair (constructor, non-constructor)
  - but, for signed integers, such elimination does not give elegant results

Building $\mathbb{Z}$ using algebraic terms — with non-free constructors

Approach #2: $\mathbf{NF}_1^2$

- The set-theoretic definition ($\mathbb{Z} = \mathbb{N} \times \mathbb{N}/\sim$) implicitly defines a constructor:

  pair : nat $\times$ nat $\rightarrow$ int

- This is a non-free constructor:

  pair (zero, succ (zero)) $\sim$
  pair (succ (zero), succ (succ (zero))

  Equivalence classes are even infinite:

  ($\forall$p, q, k) pair (p, q) $\sim$ pair (p + k, q + k)

Approach #3: $\mathbf{NF}^1_3$ (used in KIV)

- Very intuitive idea:
  - nat is defined with 2 constructors **zero** and **succ**
  - int can be defined with 3 constructors:

    zero : $\rightarrow$ int

    succ : int $\rightarrow$ int

    pred : int $\rightarrow$ int *symmetric of* succ
- These are non-free constructors:

  pred (succ (zero)) = succ (pred (zero)) = zero

  More generally:

  ($\forall$x) pred (succ (x)) = succ (pred (x)) = x

Approach #4: $\mathbf{NF}_1^3$ (used in PSF)

- Another intuitive idea: define $\mathbb{Z}$ as $\{+, -\} \times \mathbb{N}$

- This is done using a single constructor:

  pair : sign × nat → int

- This constructor is not free:

  pair (+, zero) = pair (–, zero)

Approach #5: $\mathbf{NF}_2^2$ (SMTlib, Maude)

- Variant on the previous version
- No introduction of a "sign" sort
- But two constructors:

  $+$: nat $\rightarrow$ int

  $-$: nat $\rightarrow$ int
- Again, these constructors are not free:

  $+$ (zero) = $-$ (zero)
- How do SMTlib and Maude handle this issue?

The SMTlib solution

- The authors of SMTlib (v2.5, 2015) are aware of this issue. They write:

"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".

- Syntactic prohibition of (– 0) is easy
- But what about (– x) where x = 0?
  - this sounds like *dependent* types

The Maude solution

- Use of "higher-level" features:
  - 0) assume that Nat is defined
  - 1) define a new sort NzNat as a **subsort** of Nat
  - 2) define a new sort Int such that Nat **subsort** of Int
  - 3) define a sort NzInt such that
    (NzNat **subsort** of NzInt) and (NzInt **subsort** of Int)
  - 4) define an operation "–" : NzNat $\rightarrow$ NzInt
  - 5) **extend** "–" to Int such that – 0 = 0 and – – x = x
- Correct, but heavy machinery
- Is there a lighter solution?

Building $\mathbb{Z}$ using algebraic terms — with free constructors

Approach #6: $\mathbf{F}_2^3$ (used in mCRL2)

- Sources: the recent book by Groote & Mousavi and http://www.mcrl2.org/dev/user_manual/language_reference/data.html

- Three sorts:
  - Pos: non zero natural numbers (constructors for a binary representation)
  - Nat: natural numbers

    @c0 : → Nat   @cNat : Pos → Nat
  - Int: integers

    @cInt : Nat → Int   @cNeg : Pos → Int

The mCRL2 solution

- Intuitively:
  - @cInt (m:Nat) corresponds to +m
  - @cNeg (m:Pos) corresponds to −m

- There is an intended **dissymetry** Nat / Pos
  - duplication of zero is avoided
    @cNeg (zero) would not type check

Approach #7: $\mathbf{F}_3^2$ (used in Coq)

- Two possible definitions: Coq library vs. Christine Paulin's tutorial (LASER school, Elba Island, 2011)

Z0 : Z	Z0 : Z
| Zpos : nat $\rightarrow$ Z	| Zpos : positive $\rightarrow$ Z
| Zneg : nat $\rightarrow$ Z	| Zneg : positive $\rightarrow$ Z
where:	where:
(Zpos n) stands for n+1	(Zpos n) stands for n
(Zneg n) stands for –n–1	(Zneg n) stands for –n

Can we do better?

- mCRL2:
  - 3 sorts: Nat, Pos, Int
  - 2 constructors: @cInt, @cNeg

- Coq:
  - 2 sorts: nat (or Positive), Z
  - 3 constructors: Z0, Zneg, Zpos

- Is there a (2 sorts, 2 constructors) solution?

Approach #8: $\mathbf{F}_2^2$ (used in CADP)

- 2 sorts: Nat (reused) and Int (defined)
- 2 constructors for Int:
  - Pos : Nat $\rightarrow$ Int Pos (n) denotes n
  - Neg : Nat $\rightarrow$ Int Neg (n) denotes $-$ n $-$ 1
- The classical idempotence identities:

  $(\forall n) \quad +(+\, n) = n \quad \wedge \quad -(-\, n) = n$

  get a counterpart:

  $(\forall n) \quad \text{Pos}(\text{Pos}(n)) = n \quad \wedge \quad \text{Neg}(\text{Neg}(n)) = n$

There is no "simpler" solution

- Search for functions F (similar to + and –) that are
  - involutive : $F(F(n)) = n$
  - simple (i.e., affine) : $F(n) = an + b$
- Solutions:
  - $F(n) = n$ $\Rightarrow$ F corresponds to our Pos operator
  - $F_b(x) = -x + b$
    There is a infinite family of solutions indexed by b
    but only $b = -1$ satisfies $F(\mathbb{N}) \cup F_b(\mathbb{N}) = \mathbb{Z}$

  $\Rightarrow$ F_{-1} corresponds to our Neg operator

  $\Rightarrow$ the pair (Pos, Neg) is unique

What about derived functions?

- Claim: using the Pos and Neg constructors, the usual operators on Int can be defined simply
  - as simply as their Nat equivalents using 0 and succ
- In the next slides: [thanks to R. Mateescu and M. Sighireanu]
  - succ: Int $\rightarrow$ Int   succ is no longer a constructor for Int
  - pred: Int $\rightarrow$ Int
  - eq , < : Int, Int $\rightarrow$ Bool   equality and order relations
  - + : Int, Int $\rightarrow$ Int
  - – : Int $\rightarrow$ Int and – : Int, Int $\rightarrow$ Int   unary and binary
  - *, div, mod, rem : Int, Int $\rightarrow$ Int

Operators "succ" and "pred"

(in green : existing operators defined on $\mathbb{N}$)

(in blue : operators to be defined on $\mathbb{Z}$)

succ (Pos (n)) = Pos (succ (n))

succ (Neg (0)) = Pos (0)

succ (Neg (succ (n))) = Neg (n)

pred (Pos (0)) = Neg (0)

pred (Pos (succ (n))) = Pos (n)

pred (Neg (n)) = Neg (succ (n))

Operators "eq" and "<"

Pos (m) eq Pos (n) = m eq n

Pos (m) eq Neg (n) = false

Neg (m) eq Pos (n) = false

Neg (m) eq Neg (n) = m eq n

Pos (m) < Pos (n) = m < n

Pos (m) < Neg (n) = false

Neg (m) < Pos (n) = true

Neg (m) < Neg (n) = n < m

Operators "abs", "odd", and "even"

abs (Pos (n)) = Pos (n)

abs (Neg (n)) = Pos (succ (n))

odd (Pos (n)) = odd (n)

odd (Neg (n)) = even (n)

even (Pos (n)) = even (n)

even (Neg (n)) = odd (n)

Operators "+" and (unary, binary) "-"

Pos (0) + x = x

Pos (succ (n)) + x = Pos (n) + succ (x)

Neg (0) + x = pred (x)

Neg (succ (n)) + x = Neg (n) + pred (x)

– (Pos (0)) = Pos (0)

– (Pos (succ (n))) = Neg (n)

– (Neg (n)) = Pos (succ (n))

x – y = x + (– (y))

Operator "*" (two definitions)

Pos (0) * x = Pos (0)

Pos (succ (n)) * x = (Pos (n) * x) + x

Neg (0) * x = – (x)

Neg (succ (n)) * x = (Neg (n) * x) – x

Pos (m) * Pos (n) = Pos (m * n)

Pos (m) * Neg (n) = succ (Neg (m * succ (n)))

Neg (m) * Pos (n) = succ (Neg (succ (m) * n))

Neg (m) * Neg (n) = Pos (succ (m) * succ (n))

Operator "div"

Pos (m) div Pos (n) = Pos (m div n)

Pos (m) div Neg (n) = – (Pos (m div succ (n)))

Neg (m) div Pos (n) = – (Pos (succ (m) div n))

Neg (m) div Neg (n) = Pos (succ (m) div succ (n))

Operator "mod"

The result is zero or has the same sign as the right operand

Consistent with modular arithmetic: (x+n) mod n = x mod n

$$
\begin{array}{lll}
y < 0 \;\wedge\; x <= y & \Rightarrow & x \bmod y = (x - y) \bmod y \\
y > 0 \;\wedge\; x >= y & \Rightarrow & x \bmod y = (x - y) \bmod y \\
y < 0 \;\wedge\; x > 0 & \Rightarrow & x \bmod y = (x + y) \bmod y \\
y > 0 \;\wedge\; x < 0 & \Rightarrow & x \bmod y = (x + y) \bmod y \\
\text{otherwise} & \Rightarrow & x \bmod y = x
\end{array}
$$

Operator "rem"

The result is zero or has the same sign as the left operand

Consistent with Euclidian division: x rem y = x – (y * (x div y))

$$x >= 0 \;\wedge\; y \neq 0 \Rightarrow x \text{ rem } y = x \text{ mod abs } (y)$$

$$x < 0 \;\wedge\; y \neq 0 \Rightarrow x \text{ rem } y = -((-x) \text{ mod abs } (y))$$

What about induction?

- Isabelle/HOL manuals complain that the set-theoretic definition of $\mathbb{Z}$ forbids inductive proofs

- Our definition of $\mathbb{Z}$ supports induction:
  - P holds for Pos (0)
  - P holds for Neg (0)
  - if P holds for Pos (n), then P holds for Pos (n+1)
  - if P holds for Neg (n), then P holds for Neg (n+1)

Funny "minimal" approaches (single constructor or single sort)

Approach #9: $\mathbf{F}_1^2$ (mapping $\mathbb{N}$ to $\mathbb{Z}$)

- $\mathbb{Z}$ can be defined as $\mathbb{N}\times\mathbb{N}/\sim$
- Bijections from $\mathbb{N}^2$ to $\mathbb{N}$ exist (diagonal enumeration)
- So, bijections from $\mathbb{N}$ to $\mathbb{Z}$ exist, e.g.:

  f (n) := **if** (n *is even*) **then** n/2 **else** – (n+1)/2

  i.e.,

  f ($\mathbb{N}$) = {0, -1, 1, -2, 2, -3, 3, -4, 4, ...}

- We can define $\mathbb{Z}$ with a **single** constructor f by using such a bijection

What about defined functions?

- Computationally expensive
  - sign tests must be implemented by O(n) parity tests

$$\text{even}\ (n) \Rightarrow \text{abs}\ (f\ (n)) = f\ (n)$$
$$\text{odd}\ (n) \Rightarrow \text{abs}\ (f\ (n)) = f\ (n+1)$$

- Strongly similar to the CADP approach:

$$\text{Pos}\ (n) = f\ (2n)$$
$$\text{Neg}\ (n) = f\ (2n+1)$$

  - the one-constructor approach uses Boolean premises whereas the CADP approach uses pattern matching on its two constructors Pos and Neg

Approach #10: $\mathbf{F}^1_3$

- Suggested by Lutz Schröder at WADT 2016

  int = 0 | -1 | succ : int -> int

  - +n is represented by $succ^n$ (0)
  - -n is represented by $succ^n$ (-1)
- Funnily: these constructeurs can also describe ℕ
  - if 0 means 0, -1 means 1, succ (n) means n+2
- Advantages:
  - single sort: int does not depend on any other sort
  - strict extension of Peano (by just adding -1)

What about defined functions?

- Drawbacks:
  - computationally expensive: sign tests costs O(n)
  - bizarre induction: succ means either incrementation or decrementation

- Quite similar to the CADP approach:

$$Pos\ (succ^n\ (0)) = succ^n\ (0)$$
$$Neg\ (succ^n\ (0)) = succ^n\ (-1)$$

Conclusion

Summary

- Consensus on $\mathbb{N}$, but no consensus on $\mathbb{Z}$
  - no definition: B, PVS, VDM, Z, synchronous languages
  - set product and quotient: CASL, Isabelle/HOL
  - non-free constructors: KIV, PSF, SMTlib (dependent types), Maude (subsorts and operation overloading)
  - free constructors: mCRL, Coq, CADP + funny solutions
- Approach $\mathbf{F}_2^2$ (CADP) seems the most suitable
- Unified definition of $\mathbb{Z}$:
  - better interoperability for tools
  - reuse/sharing of specifications and proofs