Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Higher-Order Logic: Arithmetic

Burkhart Wolff

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the Isabelle/HOL library.

- Orders
- Sets
- Functions
- (Least) fixpoints and induction
- (Well-founded) recursion
- Arithmetic
- Datatypes

Motivation

Current stage of our course:

- On the basis of conservative embeddings, set theory can be built safely.
- Inductive sets can be defined using least fixpoints and suitably supported by Isabelle.
- Well-founded orderings can be defined without referring to infinity. Recursive functions can be based on these. Needs inductive sets though. Support by Isabelle provided.

Next important topic: arithmetic.

Which Approach to Take?

- Purely definitional?

  Not possible with eight basic rules (cannot enforce infinity of HOL model)!

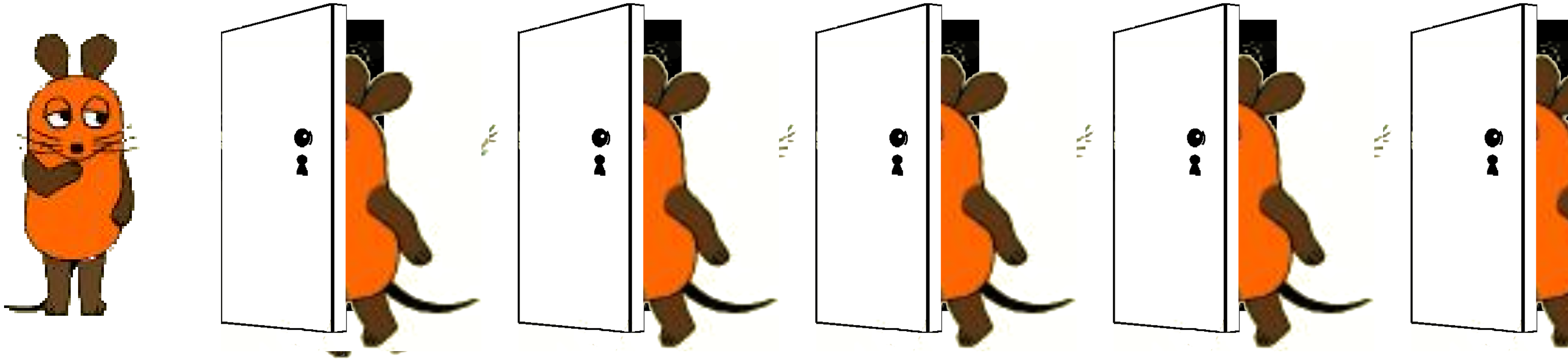
- Heavily axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

  Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as inductive subset?

  Yes. Finally use infinity axiom.

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many guests in his rooms if the receptionist can do the following procedure: A new guest arrives. The receptionist tells all guests to move one room. They move one room forward, the new guest takes the first room, and all are home and dry !

Axiom of Infinity

The axiomatic core of numbers:

axioms infinity : "∃ f :: ind ⇒ ind . inj f ∧ ¬ surj f"

where injective and surjective are:

inj f ≡ ∀ x. ∀ y. f(x)=f(y) ⟶ x=y
surj f ≡ ∀ y. ∃ x. y=f(x)

The axiom forces ind to be the "infinite type" (called "I" in [Chu40]).

Natural Numbers: Nat.thy

Based on the axiom of inifinity, a *proto*-Zero and a *proto*-Suc can be introduced by type specification:

consts
  ZERO :: ind
  SUC :: ind $\Rightarrow$ ind

specification (SUC)
  SUC_charn: inj SUC $\wedge\neg$ surj SUC
      by (*rule* infinity)
specification (ZERO)
  ZERO_charn: ZERO $\neq$SUC X
      by (*insert* SUC_charn, auto simp: surj_def)

The proofs show that witnesses satisfy the required properties of the constants.

Defining the Set Nat

Now we define inductively a set generated by ZERO and SUC:

consts NAT :: ind set

inductive NAT
  intros
  ZERO_I: ZERO∈NAT
  SUC_I : ⟦ x∈NAT ⟧⟹SUC x∈NAT

(Recall that Isabelle converts this in:

$$Nat = lfp\,(\lambda X.\{Zero\_Rep\} \cup (Suc\_Rep\,`\,X))$$

and derives an induction scheme)

Defining the Type nat

The inductive set Nat is now abstracted via type definition to the type nat:

typedef (Nat)
  nat = "Nat" **by** (...)

Constants in nat

Moreover, we define 0 and Suc via their corresponding values in Nat :

consts

Suc :: nat ⇒ nat

pred_nat :: (nat × nat) set

defs

Zero_nat_def : 0 ≡ Abs_Nat Zero_Rep

Suc_def : Suc ≡ (λn. Abs_Nat (Suc_Rep (Rep_Nat n)))

pred_nat_def : pred_nat ≡ {(m, n). n = Suc m}

Some Theorems in Nat

From the induction inherited from Nat, we derive:

nat_induct $\llbracket\ P\ 0;\ \bigwedge n.\, P\ n \Longrightarrow P\ (Suc\ n)\ \rrbracket \Longrightarrow P\ n$

diff_induct $\llbracket\ \bigwedge x.\ \ P\ x\ 0;\ \bigwedge y.\ \ P\ 0\ (Suc\ y);$
$\bigwedge x\ y.\, P\ x\ y \Longrightarrow P\ (Suc\ x)(Suc\ y)\rrbracket$
$\Longrightarrow P\ m\ n$

Moreover, we have as pre-requisite for wf-induction:

$$wf(pred\_nat)$$

These are the main weapons for proving theorems in basic number theory.

Nat.thy and Well-Founded Orders

Definition of orders:

$m < n \equiv (m, n) \in pred\_nat\hat{}+$

$m \le (n :: nat) \equiv \neg (n < m)$

have the properties:

$m \le m$

$[\![x \le y;\ y \le z]\!] \Longrightarrow x \le z$

$[\![x \le y;\ y \le x]\!] \Longrightarrow x = y$

$x < y \lor y < x \lor x = y$

Using Primitive Recursion

Nat.thy defines rich theory on nat. Uses **primrec** syntax for defining recursive functions, and case construct.

primrec

add_0 $0 + n = n$

add_Suc $\mathsf{Suc}\ m + n = \mathsf{Suc}(m + n)$

primrec

diff_0 $m - 0 = m$

diff_Suc $m - \mathsf{Suc}\ n = (\mathsf{case}\ m - n\ \textbf{of}\ 0 => 0 \mid \mathsf{Suc}\ k => k)$

primrec

mult_0 $0 * n = 0$

mult_Suc $\mathsf{Suc}\ m * n = n + (m * n)$

Some Theorems in Nat.thy

add_0_right	$m + 0 = m$
add_ac	$m + n + k = m + (n + k)$
	$m + n = n + m$
	$x + (y + z) = y + (x + z)$
mult_ac	$m * n * k = m * (n * k)$
	$m * n = n * m$
	$x * (y * z) = y * (x * z)$

Note third part of add_ac, mult_ac, respectively.

Technically, add_ac and mult_ac are lists of `thm`'s.

Proof of add_0_right

$$\dfrac{\dfrac{\text{add\_0}}{0+0=0} \qquad \dfrac{\dfrac{\dfrac{\text{add\_suc}}{Suc\,m+n = Suc(m+n)}}{Suc(m+n) = Suc\,m+n}\,\textit{sym} \qquad \dfrac{[n+0=n]^1}{Suc(n+0) = Suc\,n}\,\textit{fun\_cong}}{Suc\,n+0 = Suc\,n}\,\textit{subst}}{m+0=m}\,nat.induct^1 \;\; add\_0\_right$$

Integers

The integers $..., -2, -1, 0, 1, 2, ...$ are identified with equivalence classes over nat × nat (thought as “differences” $0-1$,$1-2$,$3-4$,...).

IntDef = Equiv + NatArith +

constdefs

```
intrel  ::  ((nat×nat) ×(nat ×nat)) set
intrel  ≡{p. ∃ x1 y1 x2 y2.
                     p=((x1::nat,y1),(x2,y2)) ∧
                     x1+y2 = x2+y1}
```

typedef (Integ)

```
int = UNIV//intrel  (...)
```

Injections of nat's into integers, negation, addition, multiplication were now defined in terms of "differences":

int :: nat => int
int m ≡Abs_Integ(intrel " {(m,0)})

minus_int_def :
− z ≡Abs_Integ (⋃(x,y)∈Rep_Integ z. intrel "{(y,x)})

add_int_def :
z + w ≡ ...

add_int_def : z ∗ w ≡ ...

Note that we use overloading here!!!

Some Theorems in IntArith

Some theorems on integers are:

zminus_zadd_distrib	$-(z + w) = -z + -w$
zminus_zminus	$-(-z) = z$
zadd_ac	$z1 + z2 + z3 = z1 + (z2 + z3)$
	$z + w = w + z$
	$x + (y + z) = y + (x + z)$
zmult_ac	$z1 * z2 * z3 = z1 * (z2 * z3)$
	$z * w = w * z$
	$z1 * (z2 * z3) = z2 * (z1 * z3)$

Compare to nat theorems.

Further Number Theories

- Binary Integers (Bin.thy, for fast computation)
- Rational Numbers (HOL-Complex/Rational.thy)
- Real Numbers (HOL-Complex/Real.thy: based on Dedekind-sections of positive rationals.
- Hyperreals (HOL-Complex/Hyperreal.thy for non-standard analysis)
- Machine numbers such as JavaIntegers [RW04] and floats [Har98, Har00] for Intel's PentiumIV

Conclusion on Arithmetic

Using conservative extensions in HOL, we can build

- the naturals (as type definition based on ind), and
- higher number theories (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Future: Analysis of hybrid systems.

The methodological overhead of the conservative method can be tackled by powerful mechanical support.

More Detailed Explanations

The Peano Axioms

The Peano axioms are:

- $0 \in nat$
- $\forall x. x \in nat \to Suc(x) \in nat$
- $\forall x. Suc(x) \neq 0$
- $\forall xy. Suc(x) = Suc(y) \to x = y$
- $\forall P.(P(0) \wedge \forall n.(P(n) \to P(Suc(n)))) \to \forall n.P(n)$

The latter formula is not an axiom in first-order logic, it is traditionally described as “axiom schema”.

However, it fit’s smoothely into HOL.

The case Statement for nat

The case statement for nat is a function of type
nat ⇒ nat ⇒ nat) ⇒ nat ⇒ nat. case z f n is defined as follows (using a common mathematical notation):

$$\texttt{case}\ z\ f\ n = \begin{cases} z & \text{if } n = 0 \\ f\ k & \text{if } n = Suc\ k \end{cases}$$

An ML-like pattern match construct in:

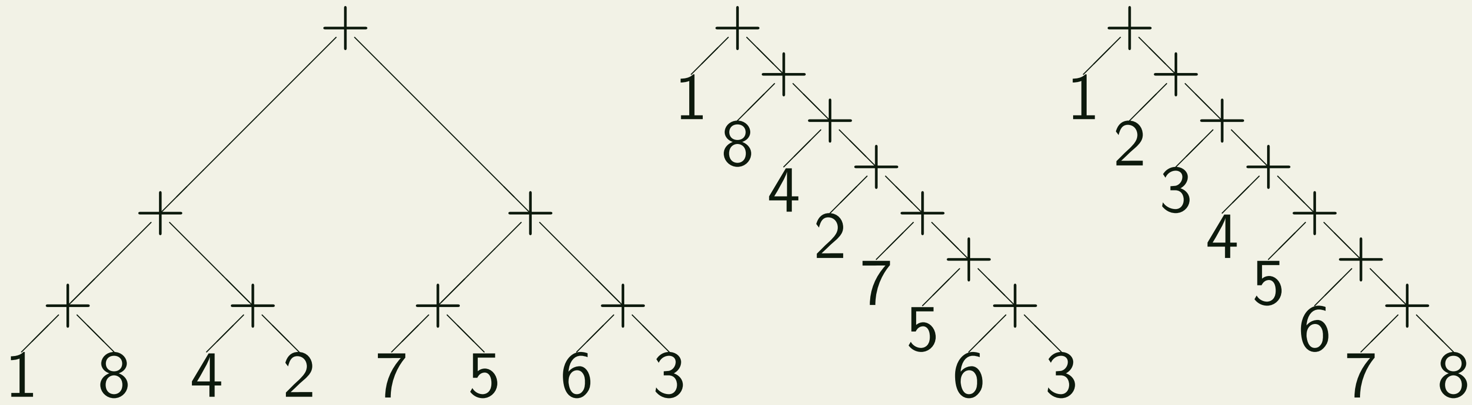
diff_Suc "m − Suc n = (case m − n **of** 0 => 0 | Suc k => k)"

uses a paraphrasing for case 0 (λ x.x) (n−m).

Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called left-commutation laws and are crucial for (ordered) rewriting.

Suppose we have the term shown below. Using associativity ($m + n + k = m + (n + k)$) this will be rewritten to the second term. Using left-commutation, this will be rewritten to the third term. This is a so-called AC-normal form, for an appropriately chosen term ordering.



Equivalence Classes

Recall the general concept of an equivalence relation. Generally, for a set S and an equivalence relation R defined on the set, one can define $S//R$, the quotient of S w.r.t. R.

$$S//R = \{A \mid A \subseteq S \wedge \forall x, y \in A.(x, y) \in R\}$$

That is, one partitions the set S into subsets such that each subset collects equivalent elements. This is a mathematical standard concept.

We explain it for integers in more detail. One can view a pair (n, m) of natural numbers as representation of the integer $n - m$. But then (n, m) and (n', m') represent the same integer if and only if $n - m = n' - m'$, or equivalently, $n + m' = n' + m$. In this case (n, m) and (n', m') are said to be equivalent. The set of equivalent elements is an equivalence class. The quotient maps therefore a set to a set of equivalence classes.

Reals According to Dedekind

The reals have been axiomatized by Dedekind by stating that a set R is partitioned into two sets A and B such that $R = A \cup B$ and for all $a \in A$ and $b \in B$, we have $a < b$. Now there is a number s such that $a \leq s \leq b$ for all $a \in A$ and $b \in B$. The irrational numbers are characterised by the fact that there exists exactly one such s. This axiomatization has been used as a basis for formalizing real numbers in Isabelle/HOL.

Hyperreals

In non-standard analysis, one works with sequences that are not necessarily converging. This is a relatively new field in mathematics and Isabelle/HOL has been successfully applied in it [FP98]. We just mention this here to say that Isabelle/HOL is used for "cutting-edge" mathematics and not just toy examples.

Hybrid Systems

Hybrid systems is a field in software engineering concerned with using finite automata for controlling physical systems such as ABS in cars etc.

References

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[FP98] Jacques D. Fleuriot and Lawrence C. Paulson. A combination of nonstandard analysis and geometry theorem proving, with application to newton's principia. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th CADE*, volume 1421 of *LNCS*, pages 3–16. Springer-Verlag, 1998.

[Har98] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[Har00] John Harrison. Formal verification of the ia/64 division algorithms. In Mark Aagaard and John Harrison, editors, *Proceedings of the 13th TPHOLs*, volume 1869 of *LNCS*, pages 233–251. Springer-Verlag, 2000.

[RW04] Nicole Rauch and Burkhart Wolff. Formalizing java's two's-complement integral type in isabelle/hol. Technical Report 458, ETH Zürich, 11 2004.