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: Datatypes

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

Datatypes: Motivation

Last lecture: Construction of natural numbers.

How can we build “datatypes” as conservative extension?

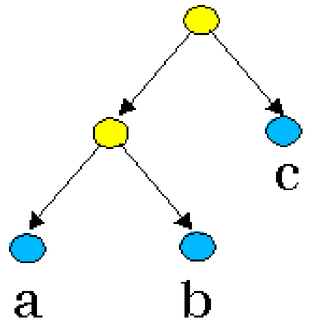
Can we generalize the natural number construction to arbitrary datatypes?

- Well, yes — by Gödelization . . .
- . . . and by an S-expressions-like tree data structure and inductive definitions.

Caveat: We will simplify! See Datatype_Universe.thy and [Wen99] for Details.

S-Expressions

Idea: We build an "ancestor"-datatype of binary trees $\alpha\ dtree$ (LISP-like S-expressions).

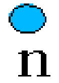


This is encoded as a set of "nodes" (defined by their path from the root and a value in the leaves), e.g.:

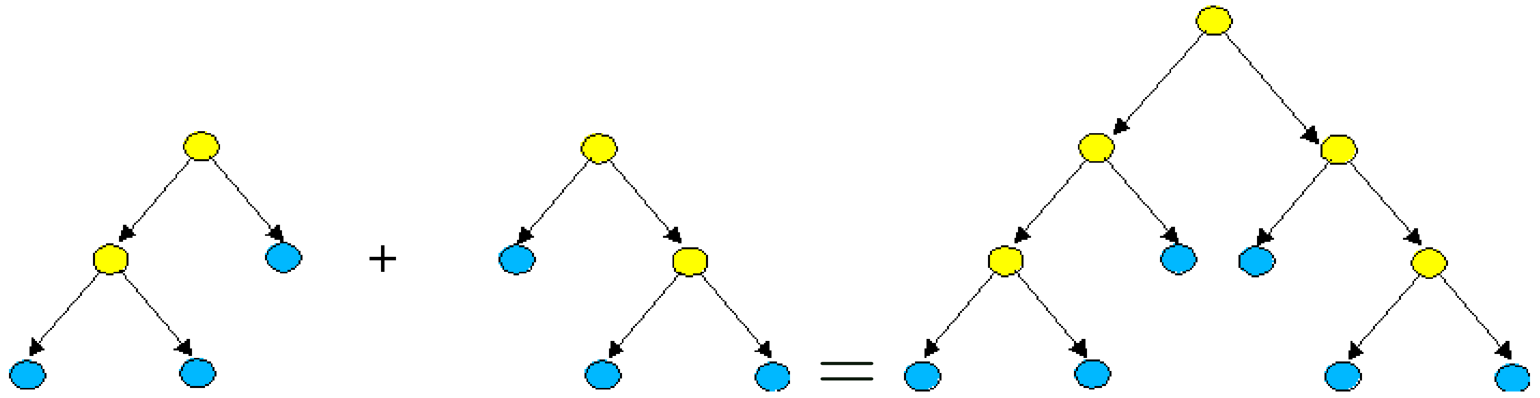
$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

Building Trees

- $\mathtt{Atom}(n)$



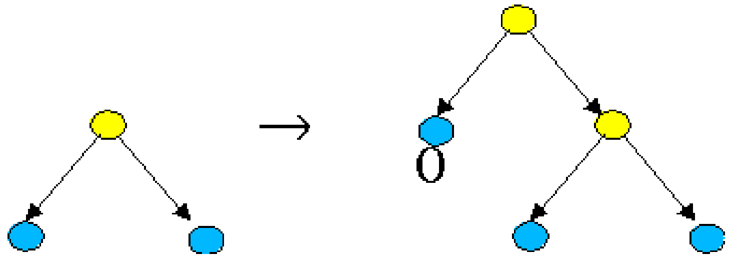
- $\mathtt{Scons}\ X\ Y$



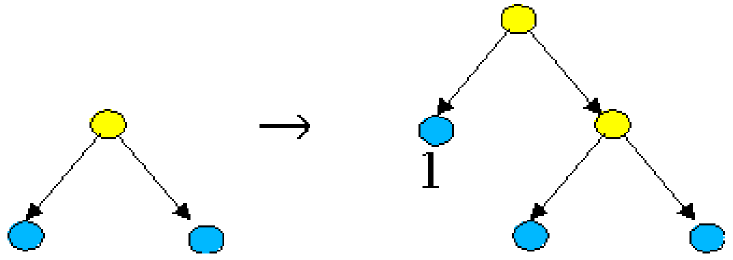
Tagging Trees

We want to tag an S-expression by either 0 or 1. This can be done by "Scons"-ing it with an S-expression consisting of an administration label. By convention, the tag is to the left.

- In0_def $\quad In0(X) \equiv Scons\ Atom(Inr(0))(X)$



- In1_def $\quad In1(X) \equiv Scons\ Atom(Inr(1))(X)$



Products and Sums on Sets of S-Expressions

Product of two sets A and B of S-expressions: All `Scons`-trees where left subtree from A, right subtree from B.

$$\texttt{uprod\_def} \quad \mathit{uprod}\,A\,B \equiv \bigcup_{x\in A}\bigcup_{y\in B}\{(\mathit{Scons}\,x\,y)\}$$

Sum of two sets A and B of S-expressions: union of A and B after S-expressions in A have been tagged 0 and S-expressions in B have been tagged 1, so that one can tell where they come from.

$$\texttt{usum\_def} \quad \mathit{usum}\,A\,B \equiv \mathit{In0}\,`\,A \cup \mathit{In1}\,`\,B$$

Some Properties of Trees and Tree Sets

- `Atom`, `In0`, `In1`, `Scons` are injective.
- `Atom` and `Scons` are pairwise distinct. `In0` are `In1` pairwise distinct.
- Tree sets represent a universe that is closed under products and sums: `usum`, `uprod` have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$.
- `uprod` and `usum` are monotone.
- Tree sets represent a universe that is closed under products and sums combined with arbitrary applications of lfp.

Reminder: we simplified!

Lists in Isabelle

Now we define inductively a subset of S-expressions having the “structure of lists”. As a pre-requisite, we define “raw constructors”:

```
constdefs
  NIL :: 'a dtree
  "NIL ≡In0(Atom(Inr(0)))"
  CONS :: ['a dtree, 'a dtree] ⇒ 'a dtree
  "CONS M N ≡In1(Scons M N)"
```

Lists as S-Expressions: Intuition

Examples of how lists would be represented as S-expressions:

Nil	$[]$
$\quad In0(Atom(Inr(0)))$	
$Cons(7, Nil)$	$[7]$
$\quad CONS\ (Atom(Inl\,7))\ In0(Atom(Inr(0)))$	
$Cons(5, Cons(7, Nil))$	$[5, 7]$
$\quad CONS\ (Atom(Inl\,5))$	
$\qquad (CONS\ (Atom(Inl\,7))\ In0(Atom(Inr(0))))$	

Now let's construct the S-expressions having this form.

Lists as S-Expressions: Inductive Construction

Based on the “raw constructors”, we define the inductive set of S-expressions:

```
 list         :: "'a dtree set ⇒ 'a dtree set"
inductive " list (A)"
  intrs
   NIL_I    NIL ∈ list (A)
   CONS_I ⟦ a ∈ A; M ∈ list (A) ⟧
           ⟹ CONS a M ∈list(A)"
```

See `src/HOL/Induct/SList.thy` in the Isabelle distribution for details!

Defining the “Real” List Type

We apply a type definition in order to define the type list by the inductive subset list (A).

```
typedef (List)
   'a  list  =
    " list (range (Atom o Inl))  ::  'a dtree  set"
  by  ...
```

Choosing A as $range\,(Atom \circ Inl)$ together with the explicit type declaration forces A to be the set containing all $Atom\,(Inl\,t)$, for each $t :: \alpha$.

This is an example of a definition of a polymorphic type.

List Constructors

We define the real constructor names for lists:

```
 Nil_def   " Nil ::' a  list  ≡ Abs_list (NIL)"
Cons_def "x#(xs::'a  list ) ≡
   Abs_list (CONS (Atom(Inl(x))) (Rep_list  xs))"
```

. . . derive the induction scheme and forget about NIL and $CONS$.

Summary

Similar to primitive recursion, compilers for datatype definitions can be provided that preform a conservative construction behind the scene

datatype 'a list = Nil | Cons 'a ('a list)

In particular, this automates the proofs of:

- the induction theorem;
- distinctness;
- injectivity of constructors.

This also works for mutually and indirectly recursive datatype definitions.

More Detailed Explanations

Gödelization

In computation theory, Gödelization is the process of encoding data structures (words, trees, . . .) into natural numbers.

Assume that we want to encode “binary tree’s” over natural numbers. These tree’s could be defined as solutions of the following set equation:

$$S(Nat) = Nat + (S(Nat) \times S(Nat))$$

or by the corresponding free data type:

```
datatype S = Atom nat
           | Node [S, S] => S
```

This notation implies that

1. Atom and Node produce distinct values,
2. Atom and Node are injective

3. all values in S are inductively generated over Atom and Node.

Can we have a model satisfying these requirements so far?

The answer is yes: Consider $Atom\,x = 2^x$ and $Node\,x\,y = 3^x * 5^y$. Distinctness holds obviously, and due to uniqueness of prime number factorization, both functions are injective. Building an inductive subset of nat generated by Atom and Node will also give the induction principle.

Unfortunately, the construction is not polymorphic as the presented S-expression-construction.

S-Expressions Explained

The datastructure we have in mind here consists of binary trees where the inner nodes are not labeled, and the leaves are labeled

- either with a term of arbitrary type, in which case the leaf would be an actual "piece of content" in the datastructure,
- or with a natural number, in which case the leaf serves special purposes for organizing our datastructure, as we will see later.

I.e., such binary trees have a type parametrized by a type variable α, the type of the latter kind of leaves. Let us call the type of such trees $\alpha\ dtree$.

As always with parametric polymorphism, when we consider how the datastructure as such works, we are not interested in what the values in the former kind of leaves are. This is just like the type and values of list elements are irrelevant for concatenating two lists. Of course, α could, by coincidence, be instantiated to type `nat`.

Think of a label of the first kind as content label and a label of the second kind as administration label.

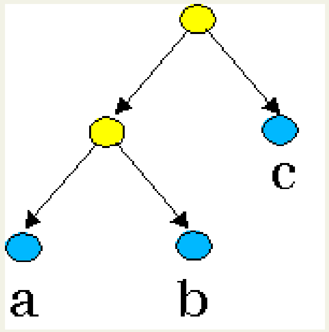
Technically, if something is either of this type or of that type, we are talking about a sum type. So a leaf label has type $\alpha + \mathtt{nat}$ (written $(\alpha, \mathtt{nat})\ \mathtt{sum}$ before), and it has the form either $\mathtt{Inl}(a)$ for some $a :: \alpha$, or $\mathtt{Inr}(n)$ for some $n :: \mathtt{nat}$.

Path Sets Explained

The set

$$\{(\langle 0,0\rangle, a), (\langle 0,1\rangle, b), (\langle 1\rangle, c)\}$$

represents the tree



The path $\langle 0,0\rangle$ means: from the root take left subtree, then again left subtree. The path $\langle 1\rangle$ means: take right subtree.

How can a path $\langle p_0, \ldots, p_n\rangle$ be represented? One idea is to use the function f :: $\mathtt{nat} \Rightarrow \mathtt{nat}$ defined by

$$f\ i = \begin{cases} p_i & \text{if } i \leq n \\ 2 & \text{otherwise} \end{cases}$$

as representation of $\langle p_0, \ldots, p_n \rangle$.

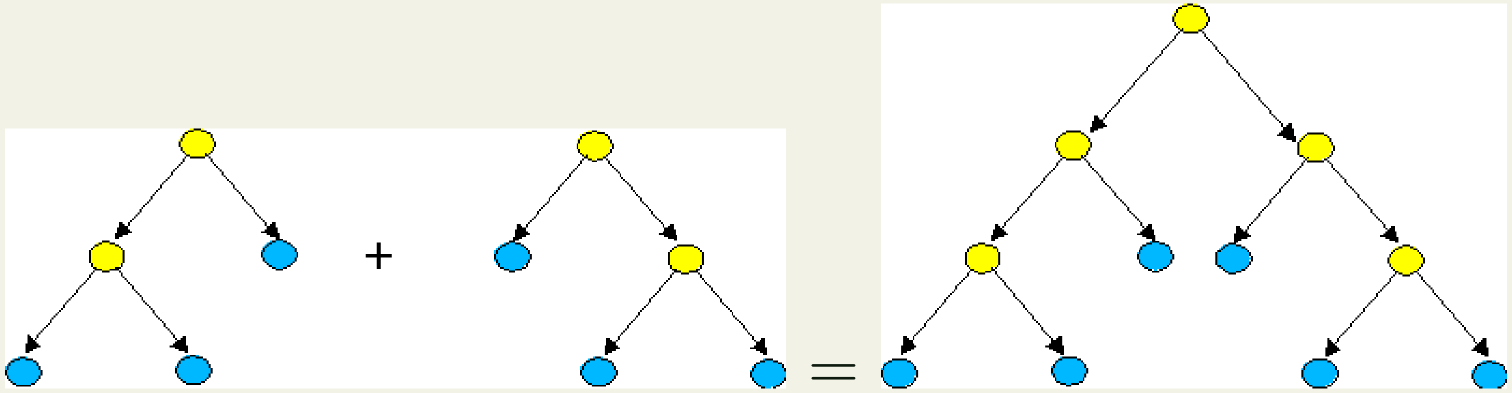
`Atom`

`Atom` takes a leaf label and turns it into a (simplest possible) S-expression (tree).

So it has type $\alpha + nat \Rightarrow \alpha\ dtree$.

Scons

`Scons` takes two S-expressions and creates a new S-expression as illustrated below:



So it has type $[\alpha\ dtree, \alpha\ dtree] \Rightarrow \alpha\ dtree$.

$In0$ ‘ . . ., $In1$ ‘ . . .

Recall that ‘ denotes the image of a function applied to a set.

Injective and Pairwise Distinct Functions

This means that any of $Atom$, $In0$, $In1$, $Scons$ applied to different S-expressions will return different S-expressions.

Moreover, a term with root $Scons$ is definitely different from a term with root $Atom$, and a term with root $In0$ is definitely different from a term with root $In1$.

Why is this important? It is an inherent characteristic of a datatype. A datatype consists of terms constructed using term constructors and is uinquely defined by what it is syntactically (one also says that terms are generated freely using the constructors). For example, injectivity of Suc and pairwise-distinctness of 0 and Suc mean for any two numbers m and n, the terms $\underbrace{Suc(\ldots Suc}_{m \text{ times}}(0)\ldots)$ and $\underbrace{Suc(\ldots Suc}_{n \text{ times}}(0)\ldots)$ are different.

Computing the Closure

Given a set T of trees (S-expressions), the closure of T under `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod` is the smallest set T' such that $T \subseteq T'$ and given any tree (or two trees, as applicable) from T', any tree constructable using `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod` is also contained in T'.

Remembering the construction of inductivelty defined sets, the closure is the least fixpoint of a monotone function adding trees to a tree set. This function must be constructed using `Atom`, `In0`, `In1`, `Scons`, `usum`, `uprod`. We do not go into the details, but note that it is crucial that `uprod` and `usum` are monotone, and note as well that slight complications arise from the fact that `usum` and `uprod` have type $[(\alpha\ dtree)\ set, (\alpha\ dtree)\ set] \Rightarrow (\alpha\ dtree)\ set$ rather than $(\alpha\ dtree)\ set \Rightarrow (\alpha\ dtree)\ set$.

The Expected Type of $Cons$

$Cons$ should have the polymorphic type $[\alpha, \alpha\ list] \Rightarrow \alpha\ list$. The important point is: the first argument is of different type than the second argument. If the first is of type τ, then the second must be of type $\tau\ list$.

In contrast, $CONS$ is of type $[(\alpha\ dtree), (\alpha\ dtree)] \Rightarrow \alpha\ dtree$.

In order to apply $CONS$ to a "list" (in fact an s-expression) and a "list element", we must first wrap the list element by $Atom \circ Inl$, so that it becomes an s-expression.

List Syntaxes

Nil, $\mathit{Cons}(7, \mathit{Nil})$, $\mathit{Cons}(5, \mathit{Cons}(7, \mathit{Nil}))$ are lists written according to what some programming languages introduce as the first, "official" syntax for lists.

For convenience, programming languages typically allow for the same lists to be written as $[]$, $[7]$, $[5, 7]$.

References

[Wen99] Markus Wenzel. Inductive datatypes in hol - lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, and and Laurent Théry C. Paulin, editors, *Proceedings of TPHOLs*, volume 1690 of *LNCS*, pages 19–36. Springer-Verlag, 1999.