# Reasoning About CBV Functional Programs in Isabelle/HOL* **

John Longley and Randy Pollack

Edinburgh University, U.K. {rap,jrl}@inf.ed.ac.uk

## 1 Introduction

We consider the old problem of proving that a computer program meets some specification. By proving, we mean machine checked proof in some formal logic. The programming language we choose to work with is a call by value functional language, essentially the functional core of Standard ML (SML). In future work we hope to add exceptions, then references and I/O to the language.

The full SML language has a formal definition in terms of a big-step operational semantics [MTHM97]. While such a definition may support formal reasoning about meta-theoretical properties of the language, it is too low-level for convenient reasoning about programs [Sym94,GV94]. Our approach stands in an alternative tradition of high-level, axiomatic *program logics* [GMW79,Pau87], and allows programmers to reason relatively directly at a level they are familiar with. In these respects, our work has roots in the logic of the LAMBDA 3.2 theorem prover and the ideas of Fourman and Phoa [PF92].

In contrast to some approaches, where the programming language is embedded in a first order logic [Tho95,HT95,Stä03], we have chosen to use higher order logic (HOL) as a meta language in order to have a rich set-theoretic language for writing program specifications. For example, we will discuss a program for sorting lists. The specification involves mathematical definitions of being an ordered list and being a permutation of another list, which are expressed in HOL using inductively defined relations.

A key feature of our approach is that the meaning of the logic can be explained in terms of purely operational concepts, such as syntactic definability and observational equivalence. Thus the logic will be intelligible to SML programmers. On the other hand, the *soundness* of our logic with respect to this interpretation can be justified by a denotational semantics; indeed, in designing our logic we have relied on well-understood denotational models for guidance.

It is clear that non-trivial proofs about programs require powerful proof automation facilities combined with flexible user interaction. The Isabelle/HOL system [NPW02] provides a ready made proof environment with these features. Using a higher order abstract syntax (HOAS) presentation in Isabelle/HOL,

* Research supported by EPSRC grant GR/N64571: "A proof system for correct program development".

** Version of April 27, 2004 (955)

we have done pragmatic experiments without developing syntactical and logical foundations from scratch. However, we have not found it possible to give a completely faithful encoding of our logic in Isabelle/HOL (see sections 2.1 and 3.3), so our work should be regarded as an experimental prototype rather than a finished tool for reasoning about programs. We have in mind an approach that would fix these problems (section 3.3), but building a system that implements this approach is left as a possibility for future work.

The Isabelle/HOL/Isar source files of our work are available from URL http://homepages.inf.ed.ac.uk/rap/mlProgLog.tgz.

**Related work** In addition to related work mentioned above, our work is very close in spirit to Extended ML [KST97,KS98]. That work takes specification and reasoning about official SML programs much further than we do, including SML program modules, and deep study of modularity for specifications. However, our approach differs from that of Extended ML in its use of insights from denotational semantics, which has enabled us to design a clean and soundly-based logic, without the explosion in complexity that beset the Extended ML project.

A foundational development of domain theory, also in Isabelle/HOL, is described in [MNvOS99]. This work is not an operational program logic, but provides a HOL type of continuous functions, and the tools to reason about them. It also goes further in uniform definition of datatypes than we have yet. However, the need to reason foundationally limits its pragmatic convenience. Furthermore, we believe our presentation, based on a logically fully abstract model (section 3.2), can be soundly extended to prove more observational equivalences than the system of [MNvOS99].

An embedding of the Ocaml language into Nuprl Type theory is reported in [Kre04]. Since Nuprl is extensional, fixpoints can be directly represented using the $Y$ combinator. However, these fixpoints can only be typed in a total setting, so this approach cannot reason about non-terminating functions, but only about functions total on a specified domain. E.g. our proof (section 5.1) that removing all zeros from a stream of naturals returns a stream without any zeros cannot be developed in the Nuprl approach.

**Structure of the paper.** In section 2 we present the syntax of the core programming language, and axioms of our logic for this core. In section 3 we explain the operational interpretation we have in mind for the logic, and outline the denotational semantics that underpins and justifies it. In section 4 we add datatypes for natural numbers and polymorphic lists to our language, and describe some case studies in reasoning about programs on these datatypes. In fact, reasoning about programs on these well-founded datatypes is not so different from reasoniong in logics of total functions, like HOL iteself. Thus in section 5 we consider the recursive datatype of streams, and set a problem for ourselves that cannot be treated in a coinductive system of total functions. Indeed, we need one more general axiom to reason about recursive datatypes. The example of streams points the way towards a uniform treatment of all positive recursive datatypes.

# 2 The core language

## 2.1 Syntax of the programming language

See figure 1. There is a typeclass, `SML`, to contain programming language types, and a subclass, `SMLeq`, for SML equality types. Types in other typeclasses retain their purely logical meaning in HOL. Variables in typeclass `SML` range over syntactic programs.

We use a higher order syntax embedding of the programming language into Isabelle/HOL: an ML function type constructor, `->` (which applies only to `SML` types, and will be axiomatized as strict), is distinguished from the logical HOL function type, `=>` (which applies to all HOL types, and is non-strict, even on `SML` types). As usual, there are constants

```
lam :: "('a => 'b) => 'a->'b"                (binder "fn " 30)
APP :: "['a->'b,'a] => 'b"                   (infixl "$" 55)
```

relating object and meta function types. In these declarations, `'a` and `'b` are inferred to be in typeclass `SML`, and `$` is infix application for `SML` functions. Isabelle binder syntax allows to write `fn x. F x` for `lam F`, where `F` has HOL functional type. We have polymorphic constants `Fix` (a fixpoint operator) and `bot` (a non-terminating program), which are definable in official SML.

`UNIT`, `BOOL`, `**` (product) and `++` (sum) types are given atomically, with their constructors (e.g. `tt` and `ff`) and destructors (e.g. `BCASE`). From the declared type of `BCASE` you can see that it is non-strict in its branches, which is correct for an SML case statement.

Using Isabelle syntax translations, we can improve our syntax somewhat (see bottom of figure 1), but Isabelle parsing is so complex that we prefer not to steer too close to the wind with overloading and syntax translations.

Isabelle/HOL typechecking over typeclass `SML` serves to typecheck programs. This is very convenient for both developing and using our tool, but not quite faithful to the SML definition, as HOL polymorphism is not the same as SML polymorphism. For example, ML *let polymorphism* is not captured in our encoding. Different representations are possible, with explicit typing judgements, that would overcome this problem, but these are significantly more complicated.

## 2.2 Logic for the core programming language

HOL equality over types in the SML typeclass represents observational equivalence in the SML semantics, i.e. indistinguishability in any context. Equal programs may be intensionally different. For example, a naive Fibonacci program is equal to an efficient one, although they have different complexities. This is part of our approach: prove contextual properties of a simple, but inefficient program, prove that an efficient program is equal to the simple one, and conclude that the efficient program has the same contextual properties.

In figure 2 we define a judgement of definedness (i.e. termination), `dfd`, and syntax `udfd` for its negation. The defined constant `mlIter` will be explained below.

```
classes
  SML < type          --{* a class of programming language types *}
  SMLeq < SML         --{* a subclass for equality types         *}

defaultsort SML

typedecl  UNIT
typedecl  BOOL
typedecl  ('a,'b) "->"       (infixr 80)  --{* functions *}
typedecl  ('a,'b) "++"       (infixr 85)  --{* sums *}
typedecl  ('a,'b) "**"       (infixr 90)  --{* products *}

arities
  UNIT :: SMLeq
  BOOL :: SMLeq
  "->" :: (SML,SML)SML
  "++" :: (SML,SML)SML
  "++" :: (SMLeq,SMLeq)SMLeq
  "**" :: (SML,SML)SML
  "**" :: (SMLeq,SMLeq)SMLeq

consts   --{* bottom, unit and bool *}
  bot   :: "'a::SML"                          --{* polymorphic bottom *}
  UN    :: UNIT                             ("<>")
  tt    :: BOOL
  ff    :: BOOL
  EQ    :: "('a::SMLeq) -> 'a -> BOOL"
  BCASE :: "['a,'a] => (BOOL ->'a)"                  --{* non-strict *}
consts  --{* product: one constructor *}
  PAIR  :: "'a -> 'b -> 'a ** 'b"
  PCASE :: "('a -> 'b -> 'c) => ('a ** 'b -> 'c)"    --{* non-strict *}
consts  --{* sum: two constructors *}
  inl     :: "'a -> 'a ++ 'b"
  inr     :: "'b -> 'a ++ 'b"
  SumCASE :: "['a->'c, 'b->'c] => ('a ++ 'b -> 'c)"  --{* non-strict *}
consts  --{* functions and recursion *}
  lam   :: "('a => 'b) => 'a->'b"               (binder "fn " 30)
  APP   :: "['a->'b,'a] => 'b"                  (infixl "$" 55)
  Fix   :: "(('a->'b) -> 'a->'b) -> 'a->'b"

syntax  --{* some syntactic sugar *}
  IF    :: "[BOOL, 'a, 'a] => 'a"
  "[,]" :: "'a => 'b => 'a ** 'b"          (infixr 30)
  "[=]" :: "[('a::SMLeq), 'a] => BOOL"     (infixl 34)
translations
  "IF b x y" == "(BCASE x y) $ b"  --{* x and y are non-strict *}
  "x[,]y" == "PAIR $ x $ y"        --{* pairing strict in both args *}
  "x[=]y" == "EQ $ x $ y"          --{* EQ strict in both args *}
```

**Fig. 1.** Language

```
constdefs    --{* definedness defined in terms of bot *}
  dfd :: "'a => bool"
  "dfd x == x ~= bot"
translations
  "udfd x" == "~ dfd x"

--{* we will use HOL naturals to talk about least fixed point *}
consts       --{* usual iteration on HOL naturals *}
  iter :: "nat => ('a::type) => ('a => 'a) => 'a"
constdefs    --{* special iterator for use in Fix_min axiom *}
  mlIter :: "nat => ('a->'b) => (('a->'b) -> 'a->'b) => 'a ->'b"
  "mlIter n b F == iter n b (%h. fn x. F $ h $ x)"
```

**Fig. 2.** Logical preliminaries

Axioms for the core are given in figure 3. (One more general axiom will be introduced in section 5.1.) Application is strict (see axiom `beta_rule`); any expression `lam F` is defined. There is an extensionality rule, `fn_ext`, for SML functions. Eta follows from extensionality.

Unlike [Stä03], we do not use a notion of *value* in formulating our axioms, but the notion, `dfd`, of definedness, since observational equivalence (equality in the logic) preserves definedness, but not "valueness".

`UNIT`, `BOOL`, `**` and `++` types are treated as if inductively defined. Their constructors and destructors are `dfd`, and their computation rules are axiomatized (e.g. `if_true` and `if_false`). As mentioned, the `CASE` constants are non-strict in their branches: when applied to a value, only the chosen branch is evaluated. Each of the type (constructors) `UNIT`, `BOOL`, `**` and `++` also have an induction principle.

*Least fixpoint axiom* We want an axiom to say that `Fix` is the *least fixpoint* operator. First, assuming `F` is defined, from axiom `Fix_rule` we have

$$\texttt{Fix \$ F = fn x. F \$ (Fix \$ F) \$ x.}$$

Informally, `Fix $ F` should be the "limit" of approximations

$$\begin{aligned} h_0 &= \bot \\ h_{n+1} &= \texttt{fn x. F \$ } h_n \texttt{ \$ x} \end{aligned}$$

Rewriting this using `iter`, the iteration constant over HOL naturals (figure 2), we have

$$h_n = \texttt{iter } n \perp \texttt{ (\%h. fn x. F \$ h \$ x).}$$

Abstracting this equation by n, `F`, and $\perp$[1], we get the definition of `mlIter` in figure 2.

[1] For technical reasons it is convenient to parameterise `mlIter` by the base case.

```
axioms
        --{* application *}
  bot_ap[simp]:  "udfd f ==> udfd (f $ x)"
  ap_bot[simp]:  "udfd x ==> udfd (f $ x)"         --{* strict *}

        --{* function types *}
  beta_rule[simp]: "dfd x ==> (lam F) $ x = F x"   --{* call-by-value *}
  fn_ext:   "[| dfd f; dfd g; !!x. dfd x ==> (f$x) = (g$x) |] ==> f = g"
  fn_dfd[simp]:    "dfd (lam F)"

        --{* least fixpoints *}
  Fix_rule: "Fix = (fn F x. F $ (Fix $ F) $ x)"
  Fix_min:  "[| dfd F; dfd (C (Fix $ F)) |] ==>
                          EX k. dfd (C (mlIter k bot F))"

        --{* UNIT type *}
  unit_Induct    : "[| P <>; dfd x |] ==> P x"
  unit_dfd [simp]: "dfd <>"

        --{* BOOL type *}
  dfd_BCASE[simp]:  "dfd (BCASE f g)"
  boolInduct:       "[| P tt; P ff; dfd x |] ==> P x"
  if_true [simp]:   "IF tt x y = x"
  if_false [simp]:  "IF ff x y = y"
  eq_dfd [simp]:    "[| dfd x; dfd y |] ==> dfd (x [=] y)"
  eq_reflection:    "((x [=] y) = tt)  =  (dfd x & x = y)"

        --{* product types *}
  dfd_PCASE[simp]: "dfd (PCASE f)"
  pair_induct: "[|!!x y. [|dfd x; dfd y |]==> P(x[,]y); dfd z |]==> P z"
  pair_dfd[simp]:  "[| dfd x; dfd y |] ==> dfd (x[,]y)"
  split[simp]:     "PCASE c $ (x[,]y) = c $ x $ y"

        --{* sum types *}
  dfd_SumCASE[simp]: "dfd (SumCASE f g)"
  dfd_inl[simp]:     "dfd inl"
  dfd_inr[simp]:     "dfd inr"
  SumCASE_inl[simp]: "SumCASE f g $ (inl $ x) = f $ x"
  SumCASE_inr[simp]: "SumCASE f g $ (inr $ y) = g $ y"
  Sum_induct:        "[| !!x. dfd x ==> P (inl $ x);
                         !!y. dfd y ==> P (inr $ y); dfd z |] ==> P z"
```

**Fig. 3.** Core language axioms

To state that `Fix $ F` is the least fixpoint of `F` we say that if `Fix $ F` is defined in any context `C::('a->'b)=>'c`, then some finite unfolding, $h_n$, is already defined in that context:

$$\texttt{dfd (C (Fix \$ F))} \Longrightarrow \exists n.\ \texttt{dfd (C } h_n\texttt{)}.$$

Using `mlIter` for $h_n$ in this equation, we get the axiom `Fix_min` of figure 3.

The notion of a function being total in one argument is defined:

```
tot1 :: "('a -> 'b) => bool"
"tot1 f == ALL x. dfd x --> dfd (f $ x)"
```

This is used in examples below.

### 2.3 Observational order

We define *observational order*, *observational equivalence* (syntax `x <o= y` and `x =o= y` resp.) and *observational limit*.

```
obsLeq :: "'a => 'a => bool"            (infixl "<o=" 18)
"a <o= b == ALL (C::'a => UNIT). dfd (C a) --> dfd (C b)"
obsEq :: "'a => 'a => bool"             (infixl "=o=" 18)
"a =o= b == (a <o= b) & (b <o= a)"
obsLim :: "'a => (nat => 'a) => bool"
"obsLim y x ==
  ALL (C::'a => UNIT). dfd (C y) = (EX (n::nat). dfd (C (x n)))"
```

`<o=` is in fact a partial order, preserved by every context. `bot` is the `<o=`-least element of every `SML` type. It is worth noting that the following are equivalent

- `a =o= b`
- `ALL (C::'a => UNIT).  C a = C b`
- `ALL (C::'a => UNIT).  dfd (C a) = dfd (C b)`
- `EX x. obsLim a x & obsLim b x`

The lemma we need for later proofs is:

```
Fix_lim_iter: "dfd F ==> obsLim (Fix $ F) (%n. mlIter n bot F)"
```

saying that `Fix $ F` is the observational limit of finite iterations of `F`.

## 3 Interpretations of our logic

We outline two kinds of semantic interpretations for our logic: one in terms of purely operational concepts, and one in terms of a denotational model for the programming language. The former is what we expect the programmer to have in mind, while the latter is used to justify the soundness of our logic, and also to inspire the design of the logic in the first place. The agreement between these two interpretations is a property known as *logical full abstraction* [LP97].

The language presented in section 2 is intended to provide an extensible core for more realistic programming languages, so we formulate our interpretations in a general setting. To begin with, let us merely assume that we have

- A programming language $\mathcal{L}$ consisting of types of typeclass `SML`, and of terms of such types, extending the language defined by figure 1.
- An intended operational semantics for $\mathcal{L}$. It suffices to give a relation $M \Downarrow v$ between closed monomorphic terms $M$ of $\mathcal{L}$ and certain "observable values" $v$, whose precise nature we need not specify.[2]
- A logical language $K(\mathcal{L})$, whose formulae are constructed from terms of $\mathcal{L}$ by means of the usual logical operators `=`, `/\`, `\/`, `~`, `ALL`, `EX`.

In the logic presented in Section 2, there are many formulae not in $K(\mathcal{L})$, since for instance we may mix types of $\mathcal{L}$ with HOL types such as `nat`. However, in order to give the idea behind the operational interpretation, it is simplest to concentrate on $K(\mathcal{L})$.

### 3.1 Operational interpretation

We now give a simple way of reading formulae of $K(\mathcal{L})$ in terms of operational concepts, by defining what it means for a formula to be *operationally true*. For closed monomorphic formulae `P` (i.e. those containing no free term or type variables), operational truth is defined by structural induction:

- A formula $M$=$N$ is operationally true if $M$ and $N$ are observationally equivalent: i.e., for all contexts $C(-)$ of $\mathcal{L}$ and all observable values $v$ we have

$$C(M) \Downarrow v \quad \text{iff} \quad C(N) \Downarrow v \tag{1}$$

  The programming intuition is that `M` may be replaced by `N` in any larger program without affecting the result.[3]
- A formula $P$`/\`$Q$ is operationally true if both $P$ and $Q$ are operationally true; similarly for `\/` and `~`. Thus, the propositional connectives have their familiar classical reading.
- A formula `ALL(`$x$`::`$t$`).`$P$ is operationally true if, for all closed terms $M$`::`$t$ of $\mathcal{L}$, the formula $P[M/x]$ is operationally true. Similarly for `EX`. The important point is that variables range over syntactically definable programs, rather than elements of some independent mathematical structure.

If two terms are observationally equivalent, they will satisfy exactly the same predicates; i.e. substitutivity of equality is sound for this interpretation. Thus, the above is the usual classical interpretation of first order logic (with a separate ground sort for each type), where a type $t$ is interpreted as the set of closed monomorphic terms of type $t$ modulo observational equivalence.

We now extend our interpretation to open and polymorphic formulae:

[2] Typically the observable values would be printable values of ground types such as integers and booleans, plus a dummy value used to indicate termination for programs of higher type.

[3] For how this relates to the formal definition of observational equivalence given in section 2, see section 5.1 below.

- An open monomorphic formula $P$ (with free variables $x_1, \ldots, x_n$) is operationally true if all of its closed instances $P[M_1/x_1, \ldots, M_n/x_n]$ are operationally true, where the $M_i$ are closed terms of appropriate types.
- A polymorphic formula $P$ (with type variables $\alpha_1, \ldots, \alpha_m$) is operationally true if all its monomorphic instances $P[t_1/\alpha_1, \ldots, t_m/\alpha_m]$ are operationally true, where the $t_i$ are monomorphic types of $\mathcal{L}$.

Is it convincing, on purely operational grounds, that the axioms of figure 3 are operationally true? In principle this might depend on the language $\mathcal{L}$, but in fact most of our axioms have been formulated to be true for a wide range of languages, even including non-functional fragments of SML.

Glossing over details, the only axioms that raise interesting questions are `fn_ext` and `Fix_min`. The axiom `fn_ext` (function extensionality) is the only one of our axioms that is specific to functional languages, and corresponds to what is known as the *context lemma*: if two programs are applicatively equivalent then they are observationally equivalent. The idea behind axiom `Fix_min` is that any "experiment" `C` which yields a value when performed on a term `Fix $ F` can only unroll the recursion operator a finite number of times, so that the same experiment must succeed when performed on `mlIter k bot F` for some `k`.

`Fix_min` plays more or less the same role as the familiar *Scott induction* principle in program logics such as LCF [Sco93]. We prefer the `Fix_min` axiom partly because it avoids the reference to inclusive predicates, and partly because it is not dependent on an order relation $\sqsubseteq$ in the style of domain theory. If we introduced such a relation as primitive, we would be obliged to axiomatize it, which is problematic since the appropriate order relation may vary from one language $\mathcal{L}$ to another.[4]

### 3.2 Denotational interpretation

Whilst our axioms can (with hindsight) be justified on purely operational grounds, it is better to achieve this by showing that they hold in some denotational model which agrees with our operational one in a suitable sense. The use of a denotational semantics has several advantages. Firstly, our understanding of the model can be used to suggest what the axioms ought to be in the first place. Secondly, the verifications that the axioms hold in the model tend to involve more abstract reasoning than the corresponding operational verifications, and to be more easily transferable from one language to another. Thirdly, a denotational semantics can be used to show soundness (and hence consistency) for the whole logic, not just the fragment $K(\mathcal{L})$, whereas it is unclear how the operational interpretation could be extended to cover types such as `UNIT->nat`.

Without going into technical details, the model we have in mind is a *presheaf category* $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$, where $\mathcal{C}$ is some denotational model of $\mathcal{L}$. Types of our logic will be interpreted by objects $X$ in the presheaf category, and closed terms by

[4] There are even "functional" languages for which the order relation is not defined extensionally, see e.g. [Lon99].

morphisms $1 \to X$. We then interpret our logic in the ordinary classical way over the homsets $\mathrm{Hom}(1, X)$.

The category $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ has two important full subcategories, corresponding to **Set** and to $\mathcal{C}$ itself. We use objects of **Set** to interpret pure HOL types such as `nat` or `nat->nat`, and objects of $\mathcal{C}$ to interpret SML types. Thus, $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ offers a model in which the ordinary mathematical universe of sets lives side-by-side with the computational universe of SML types and programs.

Moreover, we can choose the category $\mathcal{C}$ to be a model of $\mathcal{L}$ that is both *fully abstract* (observationally equivalent programs have the same denotation) and *universal* (every element of the relevant object is the denotation of some program). From these facts it is not hard to see that, when restricted to $K(\mathcal{L})$, our interpretation agrees precisely with the operational one given earlier. This goodness-of-fit property is known as *logical full abstraction*.

In future work we will extend our logic to deal with some non-functional fragments of SML including exceptions, references and I/O. An overview of the denotational ideas underpinning our approach is given in [Lon03].

### 3.3 A serious problem

An attempt to give a denotational semantics in this way for the whole of our logic, as currently formalized, shows up a significant problem. In Isabelle/HOL the definite description operator (written `THE`) is available for all types. Consider the following "program":

```
UNIT_swap :: "UNIT -> UNIT"
"UNIT_swap == fn x. (THE y. ~y=x)"
```

This claims to be a function that swaps `bot` and `<>`. Such a function cannot be definable in SML (with it, one could solve the halting problem), violating our operational requirement that terms of an `SML` type are SML definable. In fact, one can derive a contradiction using the axiom `Fix_rule`, since `UNIT_swap` clearly does not have a fixed point.

The definite description operator is pragmatically essential in pure HOL, and useful in specifications of programs, but its use must be controlled, as the above example shows. Our proposed solution involves introducing a typeclass `mathtype` for "pure mathematical types", a subclass of `type`, analogous to `SML`. We then insist that free variables in the body of a definite description are restricted to be of class `mathtype`. Unfortunately, this proposal can not be implemented without significant re-engineering of Isabelle/HOL, or building our own system from scratch.

## 4 Inductive datatypes

In section 5 we indicate, using the example of streams, how all positive recursive datatypes can be uniformly constructed from the types of their constructors. In

```
typedecl  NAT
arities   NAT :: SMLeq
consts      --{* datatype of natural numbers *}
  ZZ    :: NAT
  SS    :: "NAT -> NAT"
  NCASE :: "'a => (NAT -> 'a) => (NAT -> 'a)" --{* case is non-strict *}
axioms     --{* nat as a datatype *}
  ZZ_dfd [simp]:    "dfd ZZ"
  SS_tot1 [simp]:   "tot1 SS"
  dfd_NCASE [simp]: "dfd (NCASE f g)"
  nat_Induct:
      "[| P ZZ; !!y. [| dfd y; P y |] ==> P (SS $ y); dfd x |] ==> P x"
  NCASE_ZZ [simp]:  "NCASE ZZ x y = x"
  NCASE_SS [simp]:  "NCASE (SS $ n) x y = (y $ n)"
```

**Fig. 4.** A datatype of natural numbers

this section we simply axiomatize inductively defined (well founded) datatypes NAT and LIST as examples, to show we can reason about them straightforwardly.

Modulo a good deal of detailed work, reasoning about total programs over well founded datatypes is not so different than reasoning about systems of total functions, such as type theory or HOL itself. For example, uniform iteration and recursion functions are defined (primitive recursion over NAT, fold over LIST, ...), and their properties proved. These can be used to define other programs whose totality (on defined inputs) follows easily.

### 4.1 Natural numbers

The formalization of NAT is shown in figure 4. There are constants for the constructors ZZ and SS, and for the eliminator NCASE. These are axiomatized to be dfd and tot1. There are axioms for the computation of NCASE. Only the induction axiom, nat_Induct, while natural, needs serious semantic justification.

By primitive recursion over HOL natural numbers (nat) there is an injection from nat onto the defined NATs. By this means we can convert many properties of NAT into properties of nat, which may be automatically proved by Isabelle's tactics.

By HOL inductive definition we define order relations on NAT, e.g. the less-than relation (syntax x[<]y).

```
inductive NATLT intros
NATLT_Z: "dfd x ==> ZZ [<] SS $ x"
NATLT_S: "m [<] n ==> SS $ m [<]  SS $ n"
```

For specification, this relation is more convenient than the BOOLean valued program that computes less-than. Complete induction can be derived, and from this a least number principle and well founded induction for NAT-valued measures. As an example, we have defined a naive Fibonacci program, and a fast

Fibonacci program, and proved they are equal. The naive Fibonacci program is easily seen to satisfy the Fibonacci recursion equations, hence so does the fast Fibonacci program. Moreover, every program satisfying the Fibonacci recursion equations is equal to the the naive Fibonacci program.

### 4.2 Lists

Polymorphic `LIST` is axiomatised analagously with `NAT`. We use infix `[::]` for `CONS`; the list eliminator is `LCASE`. Basic functions like map, append, flatten and reverse are easily defined from the uniform fold operator. Their correctness follows from showing they have the expected recursion equations, usually by a few steps of computation. Many basic properties follow by easy induction: map distributes over composition, append is associative, reverse is involutive, .... We give an efficient reverse program, and show it is equal to naive reverse.

After defining a length function, we derive a length induction principle for lists from the wellfounded measure induction over `NAT` (section 4.1). With this we prove a more challenging example: theorem 16 from Paulson's textbook [Pau91]

```
aop $ y $ (foldleft $ aop $ e $ xs) = foldleft $ aop $ y $ xs
```

where `aop` is associative and `e`, a right identity of `aop`, is `dfd`.

*Sorting* The examples mentioned above are trivial in one sense: correctness is expressed in terms of some recursion equations. Our stated reason for axiomatising ML in HOL, instead of FOL, is to have a richer language for program specifications. The specification for sorting involves abstract properties *ordered* and *permutation*. For example, *permutation* (syntax `xs ~ ys`) is given as a HOL inductive definition:

```
consts perm :: "('a LIST * 'a LIST) set"
inductive perm   intros
perm_trn:  "[| xs ~ ys; ys ~ zs |] ==> xs ~ zs"
perm_NIL:  "NIL ~ NIL"
perm_CONS: "[| dfd x; xs ~ ys |] ==> x[::]xs ~ x[::]ys"
perm_hd:"[|dfd x; dfd y; dfd zs |]==> x[::]y[::]zs ~ y[::]x[::]zs"
```

We give a program for insertion sort (figure 5), and prove it is correct. The sort program itself is a polymorphic function taking in a `BOOL`ean valued order function, `le`, and a list, and returning a sorted list. We use an Isabelle *locale* to specify the properties the order function must have, and prove in that locale that `isort` returns an ordered permutation of the input list. Thus for any instantiation of that locale (e.g. with the order function `LE` over `NAT`) `isort` is a correct sort program.

## 5 Recursive datatypes: streams

The examples in preceding sections, over inductive datatypes, could be carried out in logics of total functions. In this section we address an example that cannot

```
insrt :: "('a -> 'a -> BOOL) -> 'a -> 'a LIST -> 'a LIST"
"insrt == fn le x. Fix $ (fn f. LCASE (unl x)
                                  (fn y ys. IF (le $ x $ y)
                                               (x[::](y[::]ys))
                                               (y[::](f $ ys))))"
isort :: "('a -> 'a -> BOOL) -> 'a LIST -> 'a LIST"
"isort == fn le. Fix $
  (fn f xs. LCASE NIL (fn y ys. insrt $ le $ y $ (f $ ys)) $ xs)"
```

**Fig. 5.** Insertion sort program

```
pre_rmZZs :: "(NAT SEQ -> NAT SEQ) -> NAT SEQ -> NAT SEQ"
"pre_rmZZs == fn F. SCASE (fn p.
                IF (Fst$p [=] ZZ)
                   (F $ (Snd$p $ <>))
                   (Fst$p[:::](fn z. F $ (Snd$p $ <>))))"
rmZZs :: "NAT SEQ -> NAT SEQ"
"rmZZs == Fix $ pre_rmZZs"
```

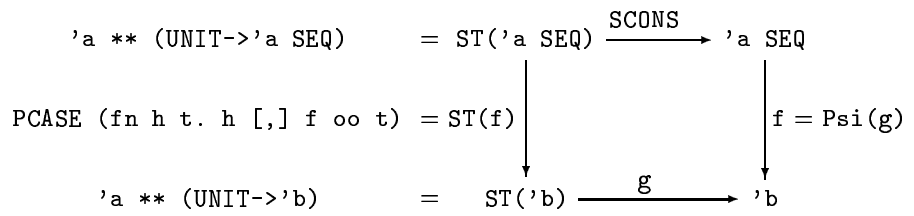
**Fig. 6.** A function to remove all zeros from a `NAT SEQ`.

be treated in logics of totality. Consider the type of polymorphic streams (`SEQ` for *sequence*), that would be defined in SML by:

```
datatype 'a SEQ = SCONS of 'a * (unit -> 'a SEQ)
```

We represent this datatype using the well known characterization that (`'a SEQ`, `SCONS`) is the initial algebra of the functor `ST X = 'a ** (UNIT->X)`. As an example over `SEQ`, consider the function, `rmZZs`, that recurses through a `NAT SEQ` removing all the zeros (figure 6). A datatype analogous to `SEQ` is definable using coinduction in HOL, Coq, and Nuprl, but the function `rmZZs` could only be definable in a complex way, with a restricted domain.

Streams are formalised (figure 7) with two constants and three axioms. The constructor, `SCONS` (infix `[:::]`) is `dfd` and `tot1`. The other constant, `Psi`, canonically completes the initial algebra property. This is expressed by axiom `seq_init`, which states that if `g::ST('b)->'b` is `dfd`, then `Psi $ g` is the unique `dfd` function `f` making the diagram commute:




From this we define the categorical destructor

```
typedecl  'a SEQ
arities   SEQ :: (SML)SML   --{* no SMLeq arity *}

--{* covariant functor characterises 'a Seq" *}
types ('a, 'b) ST = "'a ** (UNIT -> 'b)"  --{* object part of functor *}
constdefs                                 --{* arrow part of functor *}
  ST :: "('c->'d) -> ('a,'c)ST -> ('a,'d)ST"
  "ST == fn f. PCASE (fn (a::'a) (h::UNIT->'c). (a [,] (f oo h)))"
consts  --{* sequences (lazy lists) *}
  SCONS :: "('a,'a SEQ)ST -> 'a SEQ"      --{* constructor *}
  Psi :: "(('a,'b)ST -> 'b) -> 'a SEQ -> 'b"
constdefs     --{* the initial algebra property *}
  SEQ_Init_sq :: "(('a,'b)ST -> 'b) => ('a SEQ -> 'b) => bool"
  "SEQ_Init_sq g f == (dfd f) & ((f oo SCONS) = (g oo (ST $ f)))"
axioms --{* stream as a datatype *}
  dfd_SCONS[simp]:  "dfd SCONS"
  tot1_SCONS[simp]: "tot1 SCONS"
  seq_init:         "dfd g ==> SEQ_Init_sq g f = (f = Psi $ g)"
```

**Syntax note:** `oo` is program composition, i.e. `f oo g = fn x. f $ (g $ x)`.

**Fig. 7.** Streams as an initial algebra

```
  SDESTR :: "'a SEQ -> ('a ** (UNIT->'a SEQ))"
  "SDESTR == Psi $ (ST $ SCONS)"
```

and prove that `SCONS` and `SDESTR` are inverse isomorphisms. The SML case eliminator for streams is defined, and its computation rule proved:

```
  SCASE :: "('b -> (UNIT -> 'b SEQ) -> 'a) => ('b SEQ -> 'a)"
  "SCASE f == PCASE f oo SDESTR"
lemma SCASE_SCONS: "SCASE y $ (a[:::]as) = y $ a $ as"
```

*Examples* Now we can define many standard functions on streams, and prove their usual properties: head and tail (`shd`, `stl`), $n$th element from a stream (`snth`), take or drop $n$ elements from the front of a stream (`sTAKE`, `sdrop`). For example:

```
  sdrop $ n $ (stl $ xs $ <>) = stl $ (sdrop $ n $ xs) $ <>
  snth $ n $ s = shd $ (sdrop $ n $ s)
```

Interesting from a semantic viewpoint, we show that every stream is the observational limit of its initial segments

```
lemma s_lim_sTAKEs: "obsLim s (%n. sTAKE n $ s)"
```

However, we do not yet seem able to prove *stream extensionality*

```
  (ALL n. snth $ n $ s = snth $ n $ t) ==> s = t
```

which is operationally true. Stream extensionality is equivalent to a characterization proposed in [Pit94]. Finally, we cannot prove that (`ST`, `SDESTR`) is a final coalgebra. Thus, another axiom seems needed.

### 5.1 Another general axiom

Our final general axiom reflects that `x =o= y` means `x` and `y` are indistinguishable in any context.

```
axioms   obs_eq: "x =o= y ==> x = y"
```

This can be seen as saying that Leibniz equality (e.g. observational equivalence) implies extensional equality. By a fact from section 2.3, this is equivalent to uniqueness of observational limits

```
obsLim a x ==> obsLim b x ==> a = b
```

From this second formulation it is clear that stream extensionality follows from `s_lim_sTAKEs`. Furthermore, from stream extensionality we conjecture we can prove that that (`ST`, `SDESTR`) is a final coalgebra.

Using stream extensionality, we have proved that the program `rmZZs` returns a sequence with no zeros!

## 6 Conclusion

Reasoning about programs is hard. Our high level, operationally inspired logic doesn't remove the need to reason about the details of a program. However Isabelle's automation proved very useful for routine details, such as the frequent need for case distinction between `dfd` and `udfd` arguments in our CBV language. There is plenty of scope for special purpose tactics to address other routine tasks. We found the use of HOL, with its inductive definition of properties, to be much better than first order (i.e. equational) specification, and were also able to convert some questions about `SML` datatypes into questions about HOL types that are easily solved in Isabelle/HOL.

## References


[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[GV94] E. Gunter and M. VanInwegen. HOL-ML. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*. Springer-Verlag, 1994.

[HT95] Steve Hill and Simon Thompson. Miranda in Isabelle. In Lawrence Paulson, editor, *Proceedings of the first Isabelle Users Workshop*, number 397 in University of Cambridge Computer Laboratory Tech. Report Series, 1995.

[Kre04] Christoph Kreitz. Building reliable, high-performance networks with the nuprl proof development system. *Journal of Functional Programming*, 14(1), January 1004.

[KS98] S. Kahrs and D. Sannella. Reflections on the design of a specification language. In *Proc. Intl. Colloq. on Fundamental Approaches to Software Engineering. ETAPS'98*, volume 1382 of *LNCS*. Springer-Verlag, 1998.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Comp. Sci.*, 173, 1997.

[Lon99] J.R. Longley. When is a functional program not a functional program? In *Proc. 4th International Conference on Functional Programming, Paris*, pages 1–7. ACM Press, 1999.

[Lon03] John Longley. Universal types and what they are good for. In GQ Zhang, J. Lawson, Y.M. Liu, and M.K. Luo, editors, *Domain Theory, Logic and Computation, Proc. 2nd Inter. Symp. on Domain Theory*. Kluwer, 2003.

[LP97] J. Longley and G. Plotkin. Logical full abstraction and PCF. In J. Ginzburg, editor, *Tbilisi Symposium on Language, Logic and Computation*. SiLLI/CSLI, 1997.

[MNvOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau87] L. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Camb. Univ. Press, 1987.

[Pau91] L. Paulson. *ML for the Working Programmer*. Camb. Univ. Press, 1991.

[PF92] W. Phoa and M. Fourman. A proposed categorical semantics for pure ML. In W. Kuich, editor, *Proceedings on Automata, Languages and Programming (ICALP '92)*, volume 623 of *LNCS*. Springer-Verlag, 1992.

[Pit94] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[Sco93] D.S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[Stä03] R. F. Stärk. Axioms for strict and lazy functional programs. *Annals of Pure and Applied Logic*, 2003. To appear.

[Sym94] D. Syme. Reasoning with the formal definition of Standard ML in HOL. *Lecture Notes in Computer Science*, 780:43–58, 1994.

[Tho95] Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7, 1995.