J. Functional Programming **9** (2): 147–166, March 1999. Printed in the United Kingdom
© 1999 Cambridge University Press

Implementing theorem provers in a purely functional style

KEITH HANNA
Computing Laboratory, University of Kent, UK
(*e-mail:* `fkh@ukc.ac.uk`)

Abstract

This paper discusses the principles of implementing an LCF-style proof assistant using a purely functional metalanguage. Two approaches are described; in both, signatures are treated as ordinary values, rather than as mutable components within an abstract datatype. The first approach treats the object logic as a partial algebra and represents it as a partial datatype, that is, a datatype in which the domains of the constructors are restricted by predicate functions. This results in a compact, executable specification of the logic. The second approach uses an abstract type to allow an efficient representation of the logic, whilst keeping the same interface. A case study describes how these principles were put into practice in implementing a fairly complex dependently-sorted logic.

1 Introduction

The *Edinburgh LCF* approach (Gordon *et al*., 1979) to implementing a logic is well known and widely used. It is usually programmed in a functional programming language that provides imperative features (such as the *reference types* of ML). In this paper we describe a way of implementing a logic in a similar style but using a purely functional programming approach. We will show that adopting such an approach offers significant benefits and few drawbacks.

This paper is organised as follows. In section 2, we overview the essential features of an *imperative* style implementation of a language. In sections 3 and 4, we describe an alternative, *algebraic* approach, based on representing a logic as a partial algebra, that allows a purely functional implementation and serves well as an executable specification for a logic. In section 5, we describe a modified form of the approach which allows for an efficient implementation, and in section 6 we discuss experience gained with a functional implementation of a moderately complex dependently-sorted logic.

2 Principles of the imperative approach

In the LCF approach, the *object-level* formalism (typically a logic) is represented by an abstract datatype (ADT) in a computational *metalanguage*. The signature of the object logic is represented as a mutable component of the state of the ADT, and terms and theorems are represented as values defined relative to this signature.

σ	$::=$	$prop \mid nat \mid \sigma \to \sigma' \mid (\sigma)$	sorts
v			names
δ	$::=$	$v : \sigma$	declarations
Σ	$::=$	$empty \mid \delta \ \{;\ \delta\}^*$	signatures
τ	$::=$	$v \mid \tau\ \tau' \mid \tau = \tau' \mid \boldsymbol{\lambda}\ \delta\ .\ \tau \mid (\ \tau\ )$	terms

Fig. 1. Definition of the *Simple* language.

```
newSig          :: String -> ()
newParent       :: String -> ()
newConst        :: (String, Sort) -> ()

propSort        :: Sort
natSort         :: Sort
mkFnSort        :: (Sort, Sort) -> Sort

mkVar           :: (String, Sort) -> Term
mkConst         :: String -> Term
mkApp           :: (Term, Term) -> Term
mkEq            :: (Term, Term) -> Term
mkAbs           :: (Term, Term) -> Term
```

Fig. 2. Constants exported by an LCF-style implementation of *Simple*. (The corresponding destructor and test functions are also exported.)

We can illustrate the essential features of the imperative approach (and, afterward, of the algebraic approach) by constructing terms of a simple language, the typed λ-calculus named *Simple* defined in figure 1. A typical *Simple* signature is

$$\begin{aligned}&0 : nat\,;\\&plus : nat \to nat \to nat\,;\\&forall : (nat \to prop) \to prop\end{aligned}$$

and a typical term on this signature is $forall\ (\lambda m : nat\,.\ plus\ m\ 0 = m)$.

Within the metalanguage, *Simple* is represented by an ADT that:

- Holds a representation of the *current signature* (i.e. a set of declarations) of the object language.
- Exports the abstract types `Sort` and `Term`, whose values represent *Simple* sorts and terms.
- Exports the set of constants defined[1] in figure 2 that allows *Simple* entities to be constructed, with the ADT guaranteeing their well-formedness. (Although several of these constants have a functional type, they are not pure functions; in addition to returning a value some of them also modify the state of the ADT.)

[1] For uniformity, we have adopted a Haskell-like syntax throughout, even for programs that make use of ML-like imperative features.

As an example, consider the construction of the *Simple* term $\lambda m{:}nat.\ plus\ m\ 0$ on the *Simple* signature shown above.

- First, a new signature (the *current* signature) is established. Its name (`demo`) is specified by evaluating:

  ```
  newSig "demo"
  ```

  and then the three constants are added to it by evaluating:

  ```
  newConst ("0", natSort)

  let st = mkFnSort (natSort, mkFnSort (natSort, natSort))
  in newConst("plus", st)

  let st = mkFnSort (mkFnSort(natSort, propSort), propSort)
  in newConst("forall", st)
  ```

  The evaluation of the above expression causes (as a side-effect) the signature component of the state of the ADT to be updated with the new signature.
- Next, the desired term, $\lambda m{:}nat.\ plus\ m\ 0$, is constructed. First, values representing the variable, *m*, and the body, *plus m* 0, of the abstraction are formed by evaluating:

  ```
  tm1 = mkVar("m", natSort)

  tm2 = let c1 = mkConst("0")
            c2 = mkConst("plus")
        in mkApp(mkApp(c2, tm1), c1)
  ```

  and then the abstraction is formed:

  ```
  tm3 = mkAbs(tm1, tm2)
  ```

In practice, parser functions (for example, `parseTm :: String -> Term`) are provided to provide a convenient user-interface to the system. Such functions (sometimes built into the concrete syntax of the metalanguage) parse the string wrt the current signature.

The current signature of the system can be enriched at any stage by using `newConst` to add further constants or by using `newParent` to merge the signature with an already existing (disjoint) signature. It is not, however, possible to regress to an earlier signature since allowing such an operation would invalidate the integrity of any term that incorporated constants present only in the current signature.

An extension of the same principles allows a logic, rather than just a language, to be represented. For this, the ADT also exports:

- an abstract type, `Thm`, for representing theorems;
- a function, `newAxiom`, that allows axioms to be added to the signature (which may now be termed a *theory presentation*);
- a set of functions, each one corresponding to an inference rule, that allow values of type `Thm` to be constructed.

3 Algebraic approach

The *algebraic* approach, which we now describe, is an alternative way of representing a formalism. Its distinguishing features are: (i), that signatures, like terms, are represented by first-class *values* of ordinary variables rather than as a mutable component of the state of an ADT, and (ii) that the object formalism is represented as a partial algebra.

This approach relies upon an extension to the type structure of the metalanguage[2], which we now describe.

3.1 Partial datatypes

An ordinary datatype represents a free algebra ('free' in the sense that a constructed value may be taken apart to yield, uniquely, the component values from which it was constructed). A *partial datatype* is one which represents a partial free algebra, that is, one where the domains of the operations are not total.

Datatypes are often used to represent abstract types. When doing this, there are two reason why it may often be useful (for programming in general) to be able to restrict them to partial datatypes:

- to ensure that each abstract value has a *unique* (or canonical) concrete representation; and
- to exclude the representation of improper abstract values.

For instance, in representing a rational by a pair of integers, one might wish to exclude both non-canonical representations (where the two integers have a common factor) and improper ones (where the second integer is zero).

We represent a partial datatype by allowing *restrictor predicates* to be associated with the constructors of a datatype according to the syntax:

$$\textit{constructor} \quad \textit{atype}_1 \ \ldots \ \textit{atype}_n \quad . \quad \textit{predicate}$$

Each such predicate, which has the same type as its associated constructor, delimits the domain of its constructor. An attempt to apply a constructor outside the subset defined by its predicate will fail.

As an example, a partial datatype that represents the set of *balanced* binary trees is shown in figure 3. The predicate `isBalanced` has the same type as the constructor `Node` and is true only when its two arguments represent subtrees of equal weight.

3.2 Algebraic approach for a language

Taking *Simple* as an example, we now describe how a language may be represented by a partial datatype that closely parallels the usual schematic-style definition of

[2] A prototype implementation of a functional language with this extension was carried out. An alternative, equally valid, approach would be to extend a functional language so as to allow pattern-matching on abstract types, along the lines described in Burton and Cameron (1993).

```
data BTree = Tip Int
           | Node BTree BTree . isBalanced

weight (Tip i)      = i
weight (Node t1 t2) = weight t1 + weight t2

isBalanced t1 t2    = (weight t1 == weight t2)
```

Fig. 3. A partial datatype for balanced trees.

```
data Sort       = PropSort
                | NatSort
                | FnSort Sort Sort

data Dec        = Dec Token Sort . isDec

data Sig        = Empty
                | Ext Dec Sig

data Term       = Sym Int Sig . isSym
                | App Term Term . isApp
                | Eq Term Term . isEq
                | Abs Term . isAbs
```

Fig. 4. A partial datatype representation of *Simple*.

its formation rules. Such a representation may be seen as providing an *executable specification* of the language.

The partial datatype that represents *Simple* is shown in figure 4.

3.2.1 Sorts

The sorts of *Simple* are represented by the datatype `Sort`. The constructors of this datatype are used to form sorts in exactly the same way as the constructing functions of the abstract type in the imperative implementation were used. Since there are no restrictions in the way that function sorts can be built, there are no restrictor predicates associated with this datatype.

3.2.2 Declarations

Declarations are formed by associating a name with a sort. They are represented by the datatype `Dec`. It is convenient to restrict the names to non-empty alphameric sequences and so a restrictor predicate, `isDec`, is associated with the constructor to enforce this.

3.2.3 Signatures

Signatures are sequences (rather than sets) of declarations. There are no restrictions associated with the construction of signatures.

There are three auxiliary functions associated with signatures that will be needed for defining the remaining restrictor predicates. These are

```
len            :: Sig -> Int
lookupSort     :: Sig -> Int -> Sort
equivSig       :: Sig -> Sig -> Bool
```

These functions yield the length of a signature, extract the sort component of the i^{th} declaration in the signature (the most recently added declaration having index 0) and determine whether two signatures are alpha-equivalent, i.e. whether, ignoring their names, they are identical.

3.2.4 Terms

Terms are defined in such a way that they are context free; i.e. they do not rely upon referencing a global state. Rather, each term incorporates information from which a unique signature, called its *support signature*, can be derived and with respect to which its free variables are defined.

Terms are represented by the datatype `Term`. Associated with this type are the functions

```
sigOf  :: Term -> Sig
sortOf :: Term -> Sort
```

The first yields the support signature of a term, the second, its sort.

3.2.5 Symbols

Symbol formation is in accordance with the *de Bruijn* scheme (for example, see Barendregt (1984)):

$$\frac{v_{n-1}:\sigma_{n-1};\ \ldots\ v_i:\sigma_i;\ \ldots\ v_0:\sigma_0 \quad \vdash \quad i}{v_i:\sigma_i}$$

That is, a symbol is represented by an integer, *i*, specifying the position of its declaration in the signature. The use of the de Bruijn scheme has much to recommend it; alpha-equivalence is automatic and problems with capture do not occur. Notice that there is no sharp distinction between constants and variables; the former are simply symbols bound early on in a signature, the latter, ones bound nearer its end.

Using the algebraic representation, a symbol is represented by

```
data Term = Sym Int Sig . isSym
          | . . .
```

The restrictor predicate has only to check that the index is within bounds:

```
isSym i sg = (i >= 0) && (i < len sg)
```

The two auxiliary functions associated with terms have equally trivial definitions:

```
sigOf  (Sym i sg)  = sg
sortOf (Sym i sg ) = lookupSort sg i
```

3.2.6 Applications and equalities

The formation rules for applications and equalities are:

$$\frac{\tau : \sigma' \to \sigma \qquad \tau' : \sigma'}{\tau\ \tau' : \sigma} \quad \text{and} \quad \frac{\tau : \sigma \qquad \tau' : \sigma}{(\tau = \tau') : \mathit{prop}}$$

and their datatype representations (see figure 4) are straightforward. The restrictor predicate, `isApp`, associated with applications has to check that the support signatures of the two terms are equal (or, an alternative which may be preferred, that they are alpha-equivalent) and that their sorts are compatible

```
isApp tm1 tm2 =
    ((sigOf tm1) == (sigOf tm2)) && (st1a == sortOf tm2)
    where FnSort st1a st1b = sortOf tm1
```

The restrictor predicate, `isEq`, associated with equalities checks that the two support signatures are equal and that the two sorts are equal.

The auxiliary functions associated with applications and equalities are simple. For example, for applications:

```
sigOf  (App tm1 tm2) = sigOf tm1
sortOf (App tm1 tm2) = st1a
       where FnSort st1a st1b = sortof tm1
```

3.2.7 Abstractions

The formation rule for abstraction is:

$$\frac{\Sigma;\ \delta\ \vdash\ \tau}{\Sigma\ \vdash\ \lambda\delta.\ \tau}$$

That is, the most recent declaration is taken from the support signature and bound within the abstraction. Since, in the algebraic representation, the support signature is an integral part of the term, the constructor takes only a *single* argument:

```
data Term = . . .
          | Abs Term . isAbs
```

(contrast this with the *binary* constructor, `mkAbs :: (Term, Term) -> Term` in the imperative approach). The restrictor predicate for abstractions has only to check that the support signature of the body of the abstraction is non-empty:

```
isAbs tm = ((sigOf tm) /= Empty)
```

The definitions of the two auxiliary functions for abstractions are straightforward:

```
sigOf  (Abs tm) = sg where Ext dc sg = sigOf tm

sortOf (Abs tm) = FnSort st1 (sortOf tm)
                  where Ext (Dec tk st1) sg = sigOf tm
```

3.3 Example

As an example of the algebraic approach, consider the construction[3] of the same *Simple* signature and term that were used as illustrations in section 2.

- First, the new signature, `sg :: Sig`, is constructed:

  ```
  sg1 = Ext (Dec "0" NatSort) Empty

  sg2 = let st = FnSort NatSort (FnSort NatSort NatSort)
        in Ext (Dec "plus" st) sg1

  sg  = let st = FnSort (FnSort NatSort PropSort) PropSort
        in Ext (Dec "forall" st) sg2
  ```

- Next, the desired term, $\lambda m\!:\!nat.\ plus\ m\ 0$, is constructed. This is done by first extending the signature with the declaration of the bound variable:

  ```
  sg' = Ext (Dec "m" NatSort) sg
  ```

  and then constructing the body of the abstraction on this extended signature:

  ```
  tm  = let c1 = Sym 3 sg'         -- the constant "0"
            c2 = Sym 2 sg'         -- the constant "plus"
            v  = Sym 0 sg'         -- the variable "m"
        in App (App c2 v) c1
  ```

  and then forming the abstraction:

  ```
  tm1 = Abs tm
  ```

3.4 Parsing and unparsing

With the algebraic approach, a parsing function takes a signature as one of its arguments and parses the entity with respect to that signature. For example, a term parser is a function of type

```
parseTerm :: Sig -> String -> Term
```

Such a function is easily written. Names representing symbols are looked up on the signature, yielding their index. If a binder (such as a λ-abstraction) is encountered, the signature is extended with the new binding and the body of the term is parsed

[3] For clarity, we present the construction in separate stages although, of course, they can be merged to yield a single term.

relative to the extended signature. If desired, provision can be made for primed names to denote hidden instances of multiply bound names, as in a term like $\lambda x{:}nat \rightarrow nat.\ \lambda x{:}nat.\ x'\ x$.

The signature provides a natural location to store syntactic attributes of names. For instance, variant forms of declaration can allow for:

- *anonymous* symbols, i.e. symbols that are not given a name and are therefore denoted by their index number;
- *eponymous ordinary names*, i.e. symbols given an ordinary alphameric name;
- *eponymous operators*, i.e. symbols given operator status, along with a fixity, precedence and direction of associativity;
- *eponymous polymorphic operators*, i.e. dependently-sorted operators that (as described in section 6) require elided sorts to be inferred.

Sometimes, *antiquotation* (which allows metalinguistic expressions to be incorporated in a string that is being parsed) is useful in a parser. This can be simulated by a modified version of the above parsing function, of type

```
parseTerm :: Sig -> String -> [Term] -> Term
```

that takes, as an additional argument, a list of terms and inserts the i^{th} one wherever a marker of the form `@i` is encountered in the string.

Unparsing functions (pretty-printers) are likewise easily written; the signature they require can be obtained by the `sigOf` function so it does not need to be passed as a separate argument.

4 Algebraic approach for a logic

We now show how the algebraic approach developed so far can be extended to allow the representation of a logic. For this, we exploit the *propositions-as-types* principle (Howard, 1969) that treats *proofs* as explicit objects and draws the following analogies:

Proof	—	Term
Theorem (established by a proof)	—	Sort (of a term)
Proof constructor (or inference rule):	—	Term constructor:
axiom	—	symbol
specialisation	—	application
generalisation	—	λ-abstraction

We illustrate the approach using the *Simple logic*, an extension of the *Simple* language introduced earlier. The concrete syntax of the logic is shown in Fig. 5; it contains the following new features (explained in more detail in section 6.3):

- Identification of a class ϕ of *formulae*, i.e. terms of sort *prop*.
- A new kind of declaration, called a *proclamation*, of the form $v{:}\phi$.
- A new class of term, *universal quantifications*, of the form $\forall\delta.\ \phi$.
- A class π of *proofs*, containing a representative (but very limited!) selection of kinds of proof: v (appeal to an axiom); $\diamond\,\pi$ (symmetry of equality); $\pi\ \tau$ (specialisation) and $\gamma\delta.\ \pi$ (generalisation).

σ	$::=$	$prop \mid nat \mid \sigma \to \sigma' \mid (\sigma)$	sorts
v			names
δ	$::=$	$v : \sigma \mid v : \phi$	declarations
Σ	$::=$	$empty \mid \delta \ \{; \ \delta\}^*$	signatures
τ	$::=$	$v \mid \tau \ \tau' \mid \tau = \tau' \mid \boldsymbol{\lambda} \ \delta \ . \ \tau \mid \forall \ \delta \ . \ \phi \mid (\ \tau \ )$	terms
ϕ			terms of sort *prop* (i.e., formulae)
π	$::=$	$v \mid \diamond \ \pi \mid \pi \ \tau \mid \ \gamma \ \delta \ . \ \pi \mid (\pi)$	proofs (γ is a binder)

Fig. 5. Definition of the *Simple* logic.

A typical *Simple* signature (or, as it is now more properly called, *theory presentation*) is:

$$\begin{aligned}
&0 \quad : nat\,; \\
&suc : nat \to nat\,; \\
&(+) : nat \to nat \to nat\,; \\
&a1 \ \ : \forall n\!:\!nat.\ n + 0 = n; \\
&a2 \ \ : \forall n\!:\!nat.\ \forall m\!:\!nat.\ n + suc\ \ m = suc(n + m)
\end{aligned}$$

This signature has declarations for three symbols (0, *suc* and +) and proclamations for two axioms (*a1* and *a2*).

A typical proof on this signature is $\gamma m\!:\!nat.\ \diamond (a1\ \ (suc\ \ m))$, which establishes the theorem $\forall m\!:\!nat.\ suc\ \ m = suc\ \ m + 0$. The component subproofs of a proof can be read as a set of instructions for establishing intermediate theorems. For example:

Proof		**Establishes the theorem**
$a1$	(axiom)	$\vdash \forall n\!:\!nat.\ n + 0 = n$
$a1\ \ (suc\ \ m)$	(specialise)	$\vdash suc\ \ m + 0 = suc\ \ m$
$\diamond(a1\ \ (suc\ \ m))$	(symm. of eq.)	$\vdash suc\ \ m = suc\ \ m + 0$
$\gamma m\!:\!nat.\ \diamond (a1\ \ (suc\ \ m))$	(generalise)	$\vdash \forall m\!:\!nat\ \ suc\ \ m = suc\ \ m + 0$

4.1 Simple logic as a partial datatype

The partial datatype that represents the abstract syntax of the *Simple* logic is shown in figure 6. Compared with the datatype for the *Simple* language, this one has extra constructors for proclamations (`Proc`) and for universal quantification (`ForAll`), and a new datatype, `Proof`, for proofs.

4.1.1 Proclamations

We introduce the term *proclamation* to describe the assertion of an axiom. Proclamations are formed by associating a name with a formula. They behave analogously to declarations of symbols.

4.1.2 Proofs

Proofs are analogous to terms. Like terms, they incorporate their own support signature and hence are context free. Functions analogous to the `sigOf` and `sortOf`

```
data Sort       = PropSort
                | NatSort
                | FnSort Sort Sort

data Dec        = Dec  Token Sort . isDec
                | Proc Token Term . isProc

data Sig        = Empty
                | Ext Dec Sig

data Term       = Sym Int Sig . isSym
                | App Term Term . isApp
                | Eq Term Term . isEq
                | Abs Term . isAbs
                | ForAll Term . isForAll

data Proof      = Axiom Int Sig . isAxiom
                | Symmetry Term Term
                | Spec Proof Term . isSpec
                | Gen Proof . isGen
```

Fig. 6. A partial datatype representation of the *Simple* logic.

functions for terms are defined for proofs in an analogous way. The latter function, `theoremOf :: Proof -> Term`, yields the theorem a proof establishes. Just as with terms, a function for parsing proofs may be defined:

```
parseProof :: Sig -> String -> Proof
```

(In practice, parsing tends to be useful only for relatively small proofs; larger ones are generally constructed algorithmically.)

4.1.3 Axioms

The `Axiom` constructor is analogous to the `Sym` constructor for terms. It takes an index identifying a proclamation in its support signature and yields an atomic proof establishing the theorem the proclamation asserts.

4.1.4 Symmetry of equality

`Symmetry` is a typical example of a proof constructor. It corresponds to the inference rule that describes the symmetry of equality:

$$\frac{\pi:(\tau = \tau')}{(\diamond\ \pi):(\tau' = \tau)} \quad \text{Symmetry}$$

That is, it takes a proof of a theorem of the form $\tau = \tau'$ and yields a proof of the theorem $\tau' = \tau$.

4.1.5 Specialisation

The `Spec` constructor is analogous to the `App` (application) constructor for terms. It corresponds to the inference rule:

$$\frac{\pi : (\forall v : \sigma.\ \phi) \qquad \tau : \sigma}{(\pi\ \tau) : \phi[\tau/v]} \quad \forall\text{-Elim}$$

(More precisely, it corresponds to the more general case of the application of a *dependently-sorted* function where the sort of the result is dependent upon the term to which the function is applied.)

4.1.6 Generalisation

The `Gen` constructor is analogous to the `Abs` (λ-abstraction) constructor for terms. It corresponds to the inference rule:

$$\frac{\Sigma;\ v : \sigma\ \vdash\ \pi : \phi}{\Sigma\ \vdash (\gamma v : \sigma.\ \pi) : (\forall v : \sigma.\ \phi)} \quad \forall\text{-Intro}$$

5 Efficient representation

5.1 Sources of inefficiency

Whilst the representation of a logic as a partial datatype is ideal as an executable specification, it is too inefficient for practical use, for the following reasons:

1. The signature of a term is stored, redundantly, within each symbol of a term. (In practice, these multiple signatures are likely to be stored, internally, as references to a *single* instance of the data structure, but this is still inefficient.)
2. Operations that are known to preserve the integrity of a term (such as substitution of an appropriately-sorted subterm for a free variable inside a term) will involve redundant evaluation of the restrictor predicates.
3. Every time an application is formed, the sorts of both subterms have to be computed.
4. Proofs, which are usually generated either entirely automatically or by using high-level guidance (such as a tree of tactics), tend to be immensely large. Since, in practice, low-level proofs are scarcely ever required (it is only their *existence* which is of interest), generating explicit representations of them is wasteful.
5. Shifting a term onto an extended instance of its support signature requires rebuilding the term with the indices of its free symbols appropriately offset. This shifting operation is usually required whenever substitution, a frequently used operation, is carried out. For example, consider the following two terms (shown with their symbols annotated with their indices relative to the signature $[0; suc; (+); a1\,; a2]$ that was defined earlier):

$$\begin{aligned} \tau_1 &= \lambda n{:}nat.\ \lambda m{:}nat.\ n_1\ +_4\ suc_5\ m_0 \\ \tau_2 &= \lambda x{:}nat.\ suc_4\ (suc_4\ x_0) \end{aligned}$$

and consider the term resulting from the substitution of τ_2 for the symbol *suc* in τ_1:

$$\tau_1[\tau_2/suc] \quad = \quad \lambda n\!:\!nat.\ \lambda m\!:\!nat.\ n_1\ +_4\ (\lambda x\!:\!nat.\ suc_6\ (suc_6\ x_0))\ m_0$$

The effect is that the indices of the free symbols in τ_2 have had to be offset by 2 to allow for the two extra levels of binding surrounding the context into which it was substituted.

5.2 *Efficient representation*

The various inefficiencies listed above can be ameliorated by adopting an ADT representation for terms and proofs in place of the concrete one using partial data types. By doing this:

1. The multiple copies of the support signature occurring in a term or proof can be factored out and replaced by a single instance of the signature along with single instances of the signature extensions (i.e. declarations or proclamations) required for λ-abstractions, generalisations, etc.
2. Operations, such as substitution, that are known to preserve the integrity of a term, can be implemented *within* the ADT and hence can bypass the restrictor predicates (`isSym`, etc).
3. The sort of each term can be memoised within the adt. For ordinary sorts, this tends not to produce significant efficiency gains since the sorts of terms can usually be computed from their immediate constituents. However, for dependently-sorted languages, such memoisation can be very beneficial.
4. The concrete representation of proofs can be eliminated and just the sort of each proof (i.e. the theorem it establishes) stored.
5. The inefficiency of substitution can be reduced by adopting the *generalised de Bruijn* scheme explained below.

With these changes, terms and proofs for the Simple logic are now implemented as shown in figure 7. In this scheme:

- A term (type `Term`) now consists of a preterm (which represents a term less its support signature), a memoised sort and the support signature for the term.
- A proof (type `Proof`) now consists only of the sort of the proof (i.e. the theorem the proof establishes) and its support signature; the proof object itself is omitted.
- A preterm (type `Preterm`) is represented by an ordinary datatype. A preterm is only meaningful with respect to a particular support signature.
- The datatype constructors (`Sym`, etc.) of the earlier partial datatype are now replaced by (LCF-like) constructor functions (`makeSym`, etc.), along with the corresponding destructor and test functions[4].

[4] Or, better, with a *Views*-like extension to the metalanguage, the same role can continue to be performed using pattern-matching.

```
newtype Term     =  Term (Preterm, Sort, Sig)

newtype Proof    =  Proof (Term, Sig)

data Preterm     =  Sym Int
                 |  App Preterm Preterm
                 |  Eq Preterm Preterm
                 |  Abs Dec Preterm
                 |  ForAll Dec Preterm
                 |  Offset Int Int Preterm

makeSym          :: Int -> Sig -> Term
makeSym i sg     =  if isSym i sg then Term (Sym i, lookupSort sg i, sg)
                              else error "Bad symbol"

isSym            :: Int -> Sig -> Bool
isSym i sg       =  (i >= 0) && (i < len sg)
```

Fig. 7. Abstract datatype representation for terms and proofs.

5.2.1 *Generalised* de Bruijn *representation*

The de Bruijn scheme for representing symbols is ideal from all points of view, save only the overhead involved in having to rebuild terms whenever they have to be shifted onto an extension of their original support signature. This inefficiency can be ameliorated by using what we term a *generalised de Bruijn* scheme. This involves introducing an extra term constructor, $\uparrow_n^m$, which has the effect of applying an *offset* to the free symbols of a subterm. The integer m defines the amount by which the free symbols are offset and the integer n defines the position of the boundary between free and bound symbols (since the latter, of course, should not be shifted).

Let $[i]$ denote an instance of a symbol with a de Bruijn index of i, and let $m, n \geqslant 0$. Then the meaning of an offset term is recursively defined by:

$$\begin{array}{lcl} \uparrow_n^m\ [i] & \equiv & [\textbf{if } i \geqslant n \textbf{ then } i+m \textbf{ else } i] \\ \uparrow_n^m\ (\tau\ \tau') & \equiv & (\uparrow_n^m\ \tau)\ (\uparrow_n^m\ \tau') \\ \uparrow_n^m\ (\lambda\delta.\ \tau) & \equiv & \lambda\delta.\ (\uparrow_{n+1}^m\ \tau) \end{array}$$

(with equalities and quantifications similarly defined).

For instance, returning to the earlier example involving the substitution of the term τ_2 for the symbol *suc* in τ_1, the resultant term can now be expressed by using an offset applied to τ_2 (instead of having to rebuild τ_2 on an extended signature) as

$$\lambda n.\ \lambda m.\ n_1 +_4 \uparrow_0^2 (\lambda x.\ suc_4\ (suc_4\ x_0))\ m_0$$

When scanning this term (for example, when testing it for α-equivalence with another term), the offset subterm $\uparrow_0^2\ (\lambda x.\ suc_4\ (suc_4\ x_0))$ is treated as equivalent to $\lambda x.\ \uparrow_1^2\ (suc_4\ (suc_4\ x_0))$ which, in turn, is equivalent to $\lambda x.\ suc_6\ (suc_6\ x_0)$.

The presence of a small number of offsets in a term will not significantly affect the costs of typical term manipulation operations. If the number of offsets rises to a level where it were to become significant, then, and only then, need the term be rebuilt.

It is also possible to use a negative offset to shift a term onto a *shorter* signature but this, however, is a potentially unsafe operation since it may cause some symbols to go out of scope. A typical operation where it can be guaranteed to be safe is β-reduction. When a term of the form $(\lambda v.\ \tau)\ \tau'$ is reduced to $\tau[\tau'/v]$, the free symbols in the original term, τ, need to be offset by -1 to compensate for the shortening of its support signature due to the loss of the λ binding. The operation is safe since no instances of the symbol v will remain in the final term.

Within the ADT, the offset constructor, $\uparrow_n^m$, can be implemented by an `Offset` constructor in the preterm datatype (see figure 7). For instance, the two preterms

```
Abs (Dec "n" NatSort) (App (Sym 4) (Sym 0))
```

and

```
Offset 2 0  (Abs (Dec "n" NatSort) (App (Sym 2) (Sym 0)))
```

are equivalent; both, interpreted relative to the signature $[0; suc; (+); a1; a2]$, represent the term $\lambda n.\ suc\ n$.

6 Case study: the Veritas logic

The Veritas logic[5] is a higher-order, dependently-sorted logic that was developed for exploring the use of dependent sorts in reasoning about digital systems. The logic (in the form of an LCF-style proof assistant) has been implemented in both the functional subset of SML and, experimentally, in Haskell (some 13K lines). The decision to work within a functional framework was a purely pragmatic one, motivated by the desire to achieve a demonstrably sound implementation, even at the possible loss of speed of user interaction. In retrospect, this decision turned out to be fully vindicated.

Many aspects of the implementation follow conventional practice (as, for instance, described in Gordon and Melham (1993)); here we focus on those aspects that were particularly influenced by adopting a purely functional approach and/or by treating signatures as values.

6.1 Signatures

Veritas signatures are implemented as values along the lines described in section 5. In addition to ordinary declarations and proclamations, a Veritas signature can also include *definitions* (a pair consisting of a name and a defining term) and declarations for algebraic *datasorts* (trees, etc.).

As well as being able to construct a signature by *extending* an existing signature with a declaration, it is also possible to construct one by *combining* or by *sharing*

[5] An overview of the logic can be found in Hanna *et al.* (1990), a detailed account in Hanna and Daeche (1992b) and typical applications in Hanna and Daeche (1992a, 1993).

existing signatures[6]. The various auxiliary functions (for example, `lookupSort`), when applied to a signature of the form `Combine sg1 sg2` treat it as if it were the signature `sg1` appended to the signature `sg2`.

The use of the de Bruijn representation for symbols has the advantage that any name clash occurring when signatures are combined is unimportant, but it has the disadvantage that any common component of a pair of combined signatures will end up duplicated. (For example, if both signatures are extensions of a signature for natural numbers, then one would end up with two independent and incompatible copies of this subsignature.)

Related to signatures, a further type of value, for representing *signature morphisms*, is also provided. A signature morphism is constructed from two signatures and a sort-preserving map between the two. It allows a term or theorem constructed on the first signature to be mapped to its analogue on the second signature. (Signature morphisms are invaluable both for making use of generic theories and for the more mundane task of transferring terms or theorems between differing presentations of what is essentially the same theory.)

6.2 Terms

The Veritas logic provides dependent sorts (Π and Σ), datasorts and subsorts, and treats sorts as terms (themselves being of sort $U0$). The sorts of terms are not unique; rather, a term-sort combination (a judgment) is well formed if the term can be inferred to have the specified sort. Since this relation is not decidable, it is now necessary (previously, it was optional) that the sort of a term be stored explicitly within the representation of a term.

Atomic terms can be either ordinary symbols or can be constructors of datasorts. The latter are represented in a similar way to symbols, but with two integers, the first identifying the datasort declaration in the signature, the second identifying the constructor in the datasort.

The adoption of the de Bruijn representation for symbols was found in practice to allow the complex set of auxiliary functions associated with the term, signature and signature morphism abstract data types to be programmed ‘right-first-time’. By contrast, it was felt that the adoption of a name-based representation in the implementation would have been both inefficient and opaque; programming errors would have been not only inevitable but also would have been difficult to identify and to correct.

In only one respect was the presence of imperative features in the metalanguage missed. The term parser allows the sort with which a dependently-sorted operator is specialised to be omitted provided it can be inferred by unification. For instance, if the functional composition operator is declared as having the dependent sort

$$(\circ) : [r, s, t : U0] \to (s \to t) \to (r \to s) \to (r \to t)$$

[6] The combine, share and signature morphism operations were motivated by the Clear specification language (Burstall and Goguen, 1978).

then the parser will accept an abbreviated term like $odd \circ suc$ in place of $(\circ)\ (nat, nat, prop)\ odd\ \ suc$. An efficient functional implementation of unification was not found: fortunately, since the terms involved tend to be small, the absence of imperative assignment in this context did not give rise to significant inefficiency.

6.3 Proofs

Values of type proof (i.e. theorems) are usually constructed using LCF-style tactics. The user interacts with the system via a proof/term editor (since proofs are sometimes required for constructing terms and since simple proofs can simply be typed in and parsed, the two types are handled in a virtually identical way). The proof/term editor, and other tools, are largely driven by point-and-click operations in a multi-window X-Windows display. The purely functional implementation of the proof/term editor is interfaced to the operating system and the X Library routines using stream I/O in conjunction with a simple front-end interpreter written in C. The handling of user errors within the editor (for instance, selecting an inappropriate tactic or typing in a badly-sorted term) follows conventional practice and uses `Maybe` types.

The regime of lazy evaluation within Haskell (the metalanguage) is an advantage in that it facilitates purely-functional I/O but, in some other respects, it is a disadvantage. For instance, within the implementation of the editor it is necessary to force complete evaluation of all results, whether or not they are needed for display, simply in order to allow any user errors to be identified at the earliest opportunity.

When using a goal-directed approach, the actual *construction* of terms and proofs tends to take very little time; the process is generally perceived by the user as almost instantaneous. What can take time, however, is the process of *discovering* how to construct the object. This often involves an open-ended search within a tactic, as, for example, in searching for a sequence of inferences to simplify a term or establish a theorem. The constraint of being limited to using only purely functional algorithms here is potentially a burden. In many cases, any inefficiency can be, at least partially, offset by using the (insecure, context-sensitive) preterm type in search algorithms rather than the more expensive term type.

6.4 Concurrent implementation

A much remarked advantage of a purely functional implementation is that it allows concurrent evaluation on a multiprocessor machine. One high-level way of controlling the deployment of the processors is by inserting annotations into the functional program to indicate subexpressions that are candidates for concurrent evaluation (if resources allow). The gains that might be realisable in practice from using this approach to concurrent evaluation for the Veritas system were explored using the Haskell implementation. Two levels of granularity at which concurrency might usefully be employed in this program were identified: *fine grained* (within the abstract data type that represents Veritas terms) and *coarse grained* (within tactics).

Within the `Term` ADT, there is much scope for concurrency; any operation that is applied independently to the subterms of a term (substitution being a good example) can be undertaken concurrently. In order to avoid the overheads of initiating concurrent evaluation from outweighing the gains, it is necessary to be selective and avoid initiating trivial tasks. This was done by taking account both of the complexity of subterms (a count of the number of constructors in all subterms being memoised) and by preferring to allow concurrency at shallower levels of recursion than at deeper ones.

Within many tactics, there is likewise much scope for concurrency. In particular, the ‘or-else’ tactical (which is used in trying one tactic and then, if it fails, an alternative one) can be replaced by a version which tries both tactics concurrently. Such a tactical, which can be nested to arbitrary depth, provides the means to write highly concurrent tactics, such as simplifiers which try a variety of techniques concurrently to achieve a goal.

The tests of the effectiveness with which concurrency along the above lines could, in practice, be deployed, were undertaken using a specialised compiler (the HBC-PP system (Runciman and Wakeling, 1995b)) that takes a Haskell program with concurrency annotations and yields instrumented code that runs in quasi-parallel on a single-processor machine. The outcome of these tests was reported in Hanna and Howells (1995). In brief, it was found that while effective exploitation of fine-grained concurrency was difficult to achieve, control of coarse-grained concurrency was comparatively easy to achieve and could result in as much as a tenfold degree of useful processor parallelism being realised for some tasks.

7 Concluding remarks

In the LCF approach to the implementation of a proof assistant, a signature of the object logic is treated as a mutable component of the state of the system and a term or theorem is treated as a value defined relative to this state. In this paper we have proposed treating signatures as ordinary values and incorporating signatures in terms and theorems so that they become context-free ordinary values. These changes bring a number of advantages. They mean that a purely functional implementation can be adopted (with the increase in clarity and likelihood of correct implementation that this implies as well as the possibility of concurrent evaluation). They also mean that Clear-like theory structuring operations (combine, share, signature morphism) can be incorporated and that the user is not limited to working within a single theory at at time.

We have explored two approaches to implementing a logic in this style. In the first, the logic, viewed as a partial algebra, is represented by a partial datatype. The syntactic categories of the object logic (signatures, terms, etc.) are implemented as types, the formation rules (for terms and proofs) as constructors, and the conditions for well-formedness of the rules (sort checking, etc.) as restrictor predicates. The form of the datatype is directly related to the usual schematic presentation of the formation rules for a logic and the program serves well as a formal specification of the logic. It is, however, too inefficient for practical use. The second implementation is obtained from the first by a series of informal transformations (we speculate that it may be possible to formalise this process and derive it by means of a series of formal, correctness-preserving transformations). It results in an ADT which encapsulates a reasonably efficient representation (principally, one in which redundant signatures and explicit proofs are eliminated, and sorts are memoised) while presenting a similar interface to the algebraic definition. It has the disadvantage that pattern-matching can no longer be used to split syntactic objects into their constituents; instead, explicit destructor functions and test functions have to be provided.

Experience gained with the Veritas logic has shown that this approach is viable for implementing a reasonably complex logic in a relatively transparent manner. In the future, it is possible that large-scale theorem-proving will a cooperative activity carried out between geographically dispersed, trusted implementations operating over a network. In such a setting, the representation of signatures, terms and theorems as context-free values, and the ability to use signature morphisms to reshape results built on variant signatures would be an undoubted advantage.

Acknowledgements

Special acknowledgments are due to Neil Daeche and Gareth Howells who worked with the author in developing the Veritas system and exploring its concurrent implementation. Thanks also to an anonymous referee for many helpful suggestions.

References

Barendregt, H. P. (1984) *The Lambda Calculus.* North-Holland.

Burstall, R. M. and Goguen, J. A. (1978) Putting theories together to make specifications. *Proc 5th IJCAI*, pp. 1045–1058.

Burton, F. W. and Cameron, R. D. (1993) Pattern matching with abstract data types. *J. Functional Programming*, **3**(2), 171–190.

Gordon, M. J., Milner, R. and Wadsworth, C. P. (1979) *A Mechanised Logic of Computation: Lecture Notes in Computer Science 78.* Springer-Verlag.

Gordon, M. J. C. and Melham, T. F. (eds). (1993) *Introduction to HOL.* Cambridge University Press.

Hanna, F. K. and Daeche, N. (1992a) Dependent Types and Formal Synthesis. *Phil. Trans. Royal Soc.*, **339**, 121–135.

Hanna, F. K. and Daeche, N. (1992b) *Guide to the Veritas design logic (90pp).* Technical Report, University of Kent.

Hanna, F. K. and Howells, W. G. J. (1995) Parallel theorem proving. In: Runciman, C. and Wakeling, D. (eds.), *Applications of Functional Programming.* UCL Press.

Hanna, F. K., Daeche, N. and Longley, M. (1990) Specification and Verification using Dependent Types. *IEEE Trans. Software Eng.*, **16**(9), 949–964.

Hanna, K. and Daeche, N. (1993) Strongly-Typed Theory of Structures and Behaviours. *Correct hardware design and verification methods; IFIP Trans WG10.2*, pp. 39–54. Springer-Verlag.

Howard, W. A. (1969) The formulae-as-types notion of construction. In: Seldin, J. P. and Hindley, J. R. (eds.), *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pp. 479–490. Academic Press.

Runciman, C. and Wakeling, D. (eds). (1995a) *Applications of Functional Programming*. UCL Press.

Runciman, C. and Wakeling, D. (1995b) A quasi-parallel evaluator. In: Runciman, C. and Wakeling, D. (eds.), *Applications of Functional Programming*. UCL Press.