J. Functional Programming 6 (2): 365–373, March 1996 © 1996 Cambridge University Press

FUNCTIONAL PEARL

Deduction for functional programmers

J. J. LEIFER[†]

Cambridge University Computing Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

AND B. A. SUFRIN

Oxford University Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

1 Introduction

We investigate how formal logic can be introduced to students who are familiar with functional programming in a way that takes advantage of their familiarity with higher order functions, free data-types, homomorphisms, and induction principles. In our experience, students often struggle with formal logic because they are unclear about the distinction between theorems and metatheorems, the distinction between syntactic constructors and semantic operators (and hence the meaning of models and valuations), and the induction and recursion principles over proofs. Using a functional programming notation as a metalanguage clears up these ambiguities because of the imposed type discipline: theorems and metatheorems have distinctive types, so are easily distinguished; when operators are overloaded (for example, when they are used both in syntax and semantics) their different types can be written out; and the recursion and induction principles over proofs become straightforward because the data-type for proofs is explicitly described. As an added benefit, proofs – naturally tree-structured – need not be arbitrarily linearized just so that natural number recursion and induction can be performed on them.

We present a Gofer (Jones, 1991) functional program script that defines data-types for representing well-formed formulas, proofs, and sequents in the propositional logic. We then discuss the implementation of theorem and inference schemas and illustrate the latter by defining a function that provides a constructive proof of the Deduction Theorem. Finally, we compare our approach to prior work and conclude by remarking on other research we have done in this area.

2 Wffs, proofs and sequents

In the Hilbert propositional logic, well-formed formulas (wffs) are constructed from propositional variables, binary implication, and unary negation:

[†] Corresponding author. E-mail: `James.Leifer@cl.cam.ac.uk`

```
data Wff  =  Var String
          |  Not Wff
          |  Wff :=> Wff
```

A proof is formed from four axiom schemas (*K*, *S*, *CP*, *Hyp*) and application of inference rule Modus Ponens (*MP*) to two subproofs:

```
data Prf  =  K   Wff Wff
          |  S   Wff Wff Wff
          |  CP  Wff Wff
          |  Hyp Wff
          |  MP  Prf Prf
```

Sequents are pairs containing the hypotheses and the conclusion of a proof. Function *seq* extracts a sequent from a proof by instantiating each schema with its specified parameters (and, in the process reveals the definition of each axiom schema and of Modus Ponens):

```
data Seq        =  [Wff] :|- Wff

seq            ::  Prf -> Seq
seq (K x y)     =  []    :|-  x :=> (y :=> x)
seq (S x y z)   =  []    :|-  (x :=> (y :=> z))
                              :=> ((x :=> y) :=> (x :=> z))
seq (CP x y)    =  []    :|-  (Not x :=> Not y) :=> (y :=> x)
seq (Hyp x)     =  [x]   :|-  x
seq (MP p q)    =  ws 'union' hs  :|-  y,          if w == x
                   where      ws  :|-  w        =  seq p
                              hs  :|-  x :=> y  =  seq q
```

In the above, *union* calculates the union of two sets represented as lists; it is defined below along with *without* – for removing a singleton from a list – and *subset* – for comparing two lists by the set-inclusion relation:

```
xs 'union' ys   =  nub (xs ++ ys)
xs 'without' x  =  filter (/= x) xs
xs 'subset' ys  =  all ('elem' ys) xs
```

Function *con* extracts the conclusion of a proof and *hyps* extracts the hypotheses:

```
con    ::  Prf -> Wff
con p   =  x   where _  :|- x  =  seq p
hyps   ::  Prf -> [Wff]
hyps p  =  hs  where hs :|- _  =  seq p
```

3 Good proofs and theorem schemas

Function *seq* is partial because its *MP* case is not guarded exhaustively: *seq* causes an error if applied to a proof involving an incorrect use of *MP*. We define a total function *gp* – for 'good proof' – that tests a proof for internal consistency by checking that each instance of an application of *MP* involves two subproofs that correctly match:

```
gp               ::  Prf -> Bool
gp (MP p q)       =  gp p  &&  gp q  &&  isImp cq  &&  cp == limp cq
                     where (cp, cq) = (con p, con q)
gp _              =  True

isImp            ::  Wff -> Bool
isImp (_ :=> _)   =  True
isImp _           =  False

limp, rimp       ::  Wff -> Wff
limp (x :=> _)    =  x
rimp (_ :=> x)    =  x
```

Given this notion of a good proof, we introduce the total function *just* for checking if a proof justifies a sequent, i.e. if the proof has the same conclusion but possibly fewer hypotheses than the sequent:

```
just                :: Prf -> Seq -> Bool
p `just` (hs :|- x) = gp p  &&  con p == x  &&  hyps p `subset` hs
```

Then, the turnstile symbol '⊢' for making judgments about wffs is defined as follows: For $w : \mathit{Wff}, ws : [\mathit{Wff}]$,

$$ws \vdash w \iff (\exists p{:}\mathit{Prf} \;\bullet\; p \;\text{`}\mathit{just}\text{`}\; ws \;{:}{\mid}{-}\; w)$$

As an example of a judgment made with ⊢, consider the assertion

$$(\forall x : \mathit{Wff} \;\bullet\; [\,] \vdash x :=> x).$$

It is justified constructively by writing a total function *reflex* satisfying the following specification:

$$\begin{aligned} &\mathit{reflex} : \mathit{Wff} \rightarrow \mathit{Prf} \\ &\mathit{reflex}\; x \;\text{`}\mathit{just}\text{`}\; [\,] \;{:}{\mid}{-}\; x :=> x \end{aligned}$$

Here is such a function:

```
reflex   ::  Wff -> Prf
reflex x  =  MP (K x x) (MP (K x (x :=> x))
                        (S x (x :=> x) x))
```

Its proof of its correctness is simple (involving only normal order reductions) and is constructed automatically by a system such as Jape or Boyer-Moore. Figure 1 shows an example of the use of *reflex* and the pretty-printer *showPrf* : *Prf* → *String* whose definition is omitted but is available with all the other definitions in this article from Leifer (1995). Function *showPrf* linearizes a proof and decorates each of the nodes with the sequent justified at that place in the proof. (The sequents are printed in a concrete syntax similar to the standard notation used by logicians.)

Notice that '⊢', '⟺', '∀', and '∃' are not constructors of objects of type *Wff* – as one can see by looking at the definition of *Wff* – but are part of our specification

```
0    |- x => (x => x)                            K x, x
1    |- x => ((x => x) => x)                     K x, x => x
2    |- (x => ((x => x) => x))
        => ((x => (x => x)) => (x => x))         S x, x => x, x
3    |- (x => (x => x)) => (x => x)              MP 1, 2
4    |- x => x                                   MP 0, 3
```

Fig. 1. `showPrf (reflex (Var "x"))`.

language for functional programs. But *gp* and *just* are functional programs, not predicates in the specification language, so, to be absolutely correct, we should write, for example, $gp\ p = True$ – where *True* is a constant of built-in Gofer type *Bool* – instead of *gp p*, though this verges on the pedantic. Clearly, if *gp* or *just* were not total, we would have to be much more careful.

Function *reflex* is an example of a *theorem schema*, which we define as a value of type

$$\underbrace{Wff \rightarrow \cdots \rightarrow}_{\text{zero or more}} Prf$$

whose result satisfies *gp*. By this definition, *K*, *S*, *CP*, and *Hyp* are all theorem schemas.

4 The deduction theorem and inference schemas

As seen in the definition of *reflex* (and in Figure 1) constructing proofs of properties of :=> is tedious. The following inference rule, called the Deduction Theorem, simplifies the process considerably. It is stated as follows:

$$ws\ \text{`}union\text{`}\ [h] \vdash w \quad \Longrightarrow \quad ws \vdash h :=> w$$

(Note: '$\Longrightarrow$' is the implication symbol of the specification language and has *nothing* to do with :=> and objects of type *Wff*.) We will examine a stronger statement:

$$ws \vdash w \quad \Longrightarrow \quad ws\ \text{`}without\text{`}\ h \vdash h :=> w$$

which is justified constructively by writing a total function *ded* that satisfies the following specification:

$$ded : Wff \rightarrow Prf \rightarrow Prf$$
$$gp\ p \quad \Longrightarrow \quad ded\ h\ p\ \ \text{`}just\text{`}\ \ (hyps\ p\ \text{`}without\text{`}\ h\ \ :|\text{-}\ \ h :=> con\ p)$$

Given the task of constructing a proof of $h :=> w$, this specification shows that it is sufficient to construct a proof of *w* using *h* as an added hypothesis and then apply *ded* – in general, a far easier task. As an example, the problem of constructing a proof of $x :=> ((x :=> y) :=> y)$ is reduced to constructing a simpler 3 line proof (see Figure 3) followed by two applications of *ded* (see Figures 4 and 5), resulting in a 23 line proof.

The following implementation of *ded* can, by structural induction on its second argument, be proved to meet its specification:

```
ded              ::  Wff -> Prf -> Prf
ded h (MP p q)   =   MP (ded h p) (MP (ded h q)
                                      (S h (con p) (rimp (con q))))
ded h (Hyp x)    =   reflex h,  if  h == x
ded h p          =   MP p (K (con p) h)
```

Function *ded* is an example of an *inference schema* — a value of type:

$$r : \underbrace{Wff \to \cdots \to}_{\text{zero or more}} \underbrace{Prf \to \cdots \to}_{\text{one or more}} Prf$$

whose specification is of the form:

$$\begin{array}{l}\big(\forall x_0, \ldots, x_n : Wff, p_0 \ldots p_m : Prf \\ \quad \mid gp\ p_0 \wedge \cdots \wedge gp\ p_m \ \wedge\ G\ x_0 \ldots x_n\ p_0 \ldots p_m \\ \quad \bullet\, gp(r\ x_0 \ldots x_n\ p_0 \ldots p_m) \\ \big)\end{array}$$

where

$$G : \underbrace{Wff \to \cdots \to}_{\text{zero or more}} \underbrace{Prf \to \cdots \to}_{\text{one or more}} Bool$$

is the *guard* of the schema. Notice that by these definitions, *MP* is an inference schema guarded by

$$G\ p\ q \quad = \quad isImp\ (con\ q)\ \wedge\ con\ p = limp\ (con\ q)$$

5 The LCF approach

The approach to logic outlined here differs from that taken in Gordon (1979). In LCF and its successors, wffs are represented by the abstract data type *Form* and theorems by *Thm*. The latter's constructors are the axioms and inference rules of the object logic. Proofs are not represented as data, but as programs that construct values of type *Thm*. Axiom schemas are functions of type [*Form*] → *Thm*, and proof rules are functions of type [*Thm*] → *Thm*. In LCF, the primitive *Thm* constructors can only generate sound theorems (each causing an exception if used incorrectly) with the advantage that if an object has type *Thm*, that is witness to its soundness. In our system, the soundness of a proof must be checked by *gp*. However, the burden of justifying that a derived inference rule meets its specification is equally hard in both system: in ours one verifies that the rule generates objects satisfying *gp* and in LCF one verifies that the rule calls each primitive (or derived) constructor correctly so as not to cause an exception.

In contrast to our explicit representation of a proof tree, LCF style systems represent the tree implicitly in the data used to implement the control structures of the metalanguage. Meta-level reasoning in our framework is conducted by induction over the structure of proofs, whereas in the LCF framework it is conducted by

```
p,q,r :: Prf

p       =  MP (Hyp (Var "x"))
              (Hyp (Var "x" :=> Var "y"))

q       =  ded (Var "x" :=> Var "y") p

r       =  ded (Var "x") q
```

Fig. 2. The definition of three proofs illustrating the use of *ded*. See Figures 3, 4, and 5.

```
0   x |- x              Hyp x
1   x => y |- x => y    Hyp x => y
2   x, x => y |- y      MP 0, 1
```

Fig. 3. The output from `showPrf p`.

```
0   x |- x                                  Hyp x
1   |- x => ((x => y) => x)                 K x, x => y
2   x |- (x => y) => x                      MP 0, 1
3   |- (x => y) => ((x => y) => (x => y))   K x => y, x => y
                :
                :
8   |- ((x => y) => (x => y))
          => (((x => y) => x) => ((x => y) => y))
                                            S x => y, x, y
9   |- ((x => y) => x) => ((x => y) => y)   MP 7, 8
10  x |- (x => y) => y                      MP 2, 9
```

Fig. 4. The output from `showPrf q` in abridged form.

```
0   |- x => (x => x)                          K x, x
1   |- x => ((x => x) => x)                   K x, x => x
2   |- (x => ((x => x) => x))
         => ((x => (x => x)) => (x => x))     S x, x => x, x
3   |- (x => (x => x)) => (x => x)            MP 1, 2
                 :
                 :
20  |- (x => (((x => y) => x) => ((x => y) => y)))
         => ((x => ((x => y) => x)) => (x => ((x => y) => y)))
                              S x, (x => y) => x, (x => y) => y
21  |- (x => ((x => y) => x))
         => (x => ((x => y) => y))            MP 19, 20
22  |- x => ((x => y) => y)                   MP 10, 21
```

Fig. 5. The output from `showPrf r` in abridged form.

induction on the computation that results in a theorem. Hence, recursion over the structure of a proof is not possible in LCF: an inference rule – such as *ded* – that is defined by case-analysis on the history of its argument proof cannot be written in LCF except as a primitive constructor

In some of the successors of LCF, for example Isabelle (Paulson, 1994), it is possible to formalise the rules of a logic in such a way that an explicit record of the proof tree is made as a theorem is constructed.

6 Thoughts on *ded* and the teaching of logic

The presentation of *ded* offers several distinct advantages:

- Our method of defining *ded* as a functional program and then proving it correct demonstrates a clear separation of concerns. The proof of correctness is a simple result and is done easily by mechanical means (Leifer, 1995). Many presentations of the Deduction Theorem in logic texts combine the definition and the proof together (see, for example, Hamilton, 1993), and this leads to a more complicated exposition than is provided by the five line definition of *ded*.
- We have replaced the notion of induction and recursion on the length of a proof as a textual object (as is done in Hamilton (1993), for example) with a simple induction and recursion over proof trees, where the cases are dictated directly by the type definition of *Prf* and no arbitrary linearization of the proof is necessary.
- Students can experiment with and test *ded* on the computer. This provides them with a concrete justification for the introduction of hypothetical proof techniques because they can see directly the effect of applying *ded* to a hypothetical proof in order to generate a purely axiomatic one, as is illustrated in Figures 3, 4 and 5.

This last point leads to the observation that the unadulterated Hilbert logic is almost impossible to use in practice for proving propositions since the only primitive inference rule, *MP*, is not suited to 'backward' (i.e. goal-directed) proof. So, *ded* could be considered a compiler from an enriched proof language (one that includes the discharging of hypotheses) to a more basic one. An example of such an enriched language is given in Figure 6 along with a function defined in terms of *ded* that maps a hypothetical proof to an axiomatic one.

Following this example, wffs could be enriched to include constructors for conjunction and disjunction. Proofs could be enriched to include many derived rules. A comfortable deductive logic then is built up in which proofs are written at as high a level as possible and can always be 'compiled' down to purely axiomatic ones. The proof of correctness of this compiler is simply the collection of proofs that each inference schema meets its specification.

The task of designing the pieces of the high-level proof system may be given to students who can build the functional programs that accomplish the 'compiling'.

```
data HPrf          =  HK Wff Wff
                   |  HS Wff Wff Wff
                   |  HCP Wff Wff
                   |  HMP HPrf HPrf
                   |  HHyp Wff
                   |  Ded Wff HPrf

pure              ::  HPrf -> Prf
pure (HK x y)      =  K x y
pure (HS x y z)    =  S x y z
pure (HCP x y)     =  CP x y
pure (HMP p q )    =  MP (pure p) (pure q)
pure (HHyp x)      =  Hyp x
pure (Ded h p)     =  ded h (pure p)
```

Fig. 6. Some preliminary definitions in the description of enriched, hypothetical proofs.

They can have the fun of seeing the compiler translate a high-level proof to a lower level one and then of comparing the lengths of the two.

To accommodate this plan, we have built data types for representing natural deduction-style proofs; translators from proofs in such an enriched proof system to pure *Prf*-style proofs; unification support functions for generating tactics from inference rules; and tactic combinator functions ('tacticals'). With these, the first author has implemented an automatic theorem prover (as the composition of several simple inference tactics) whose domain is exactly the tautological propositions, thus providing a constructive proof of completeness. In collaboration with C. A. R. Hoare, he has shown as a corollary to the completeness proof that in the presence of *ded* all uniform inference rules (i.e. rules that respect substitution of variables) have a normal form that involves no explicit recursion on the structure of their arguments, thus justifying that the logic described by systems such as LCF is not weakened by the restriction that proofs are represented as opaque objects with no primitive destructors.

Our work on *ded* has led us to believe that computing science – and, in particular, functional programming – can offer its students new ways of learning and mastering formal logic. (This thought seems to have inspired much of the excellent presentation in Ben-Ari, 1993.) The program script in this article, along with additional material, is available in Leifer (1995), and we encourage you to experiment with and enhance our results.

Acknowledgements

We wish to thank the following for their detailed comments and criticisms on the drafts of this paper: R. S. Bird, E. W. Dijkstra (in conjunction with the Austin Tuesday Afternoon Club), D. Gries, C. A. R. Hoare, J. M. Spivey, and P. Wadler. The paper has benefited greatly from their advice and aid.

References

Ben-Ari, M. (1993) *Mathematical Logic for Computer Science*. Prentice Hall.

Gordon, M. J. C., Milner, A. J. R. G. and Wadsworth, C. P. (1979) *Edinburgh LCF: Lecture Notes in Computer Science 78*. Springer-Verlag.

Hamilton, A. G. (1993) *Logic for Mathematicians*. Cambridge University Press.

Jones, M. P. (1991) *An Introduction to Gofer*. Available, with compiler source code, via anonymous ftp from `nebula.cs.yale.edu`.

Leifer, J. (1995) *Formal Logic via Function Programming*. Available via anonymous ftp from `ftp.comlab.ox.ac.uk` in `pub/Packages/JAPE/LogicViaFP/`.

Paulson, L. C. (1994) *Introduction to Isabelle*. Computer Laboratory, Cambridge University.