Partial Functions in a Total Setting

Simon Finn (simon@ahl.co.uk)[*]

Michael Fourman (M.Fourman@ed.ac.uk)[†*]

John Longley (jrl@lfcs.ed.ac.uk)[†]

January 19, 1996

Abstract

We discuss a scheme for defining and reasoning about partial recursive functions within a classical two-valued logic in which all terms denote. We show how a total extension of the partial function introduced by a recursive declaration may be axiomatised within a classical logic, and illustrate by an example the kind of reasoning that our scheme supports. By presenting a naive set-theoretic semantics, we show that the system we propose is logically consistent. We discuss some of the practical advantages and limitations of our approach in the context of mechanical theorem-proving.

1 Introduction

Partial functions appear ubiquitously in formal descriptions of computer systems. They arise, for example, in connection with non-terminating programs or programs that terminate exceptionally, and in situations where we are interested in the behaviour of a function only on some restricted set of argument values. Partial functions also arise frequently in specification formalisms such as the 'Z' notation [15]. Partial functions arise because we want to declare functions using general recursive definitions, which don't automatically guarantee totality. So an important problem is to find a logic in which we can reason about these functions conveniently but safely. This turns out to be surprisingly hard—if x denotes a value outside the intended domain of a function f, then the treatment of the term $f(x)$ is problematic. Several approaches to this problem have previously been proposed—for a useful survey see [3].

[*]Abstract Hardware Limited, Brunel Science Park, Uxbridge, Middx., UB8 3PQ.

[†]Laboratory for the Foundations of Computer Science, University of Edinburgh, EH9 3JZ. This research was partially supported by SERC grant GR/F31199 and SERC/DTI grant GR/F35890.

The choice of a logic is a crucial issue in the design of mechanical theorem-provers or proof assistants: we would like a logic that satisfied the competing requirements of logical soundness, expressive power and ease of reasoning. The logics used in existing theorem-provers exhibit various strengths and weaknesses, and display a range of attitudes towards the problem of partial functions. For example:

(i) We can restrict ourselves to reasoning about *total* functions, and so ignore the problem completely. In the HOL system [7], for instance, only primitive recursive definitions are allowed—a function declaration will thus be legal only if it can be shown by a simple syntactic check to define a total function. This certainly makes the logic simple, but we might hope for greater expressive power. Some other theorem-provers, such as the Boyer-Moore [2] and LEGO [11] systems, allow more general function definitions, but the user is required to prove theorems guaranteeing their totality.

(ii) We can incorporate *subtyping* into our type discipline—given a partial function on some type τ, we can regard its domain as a subtype σ of τ. Elements of σ may then be construed also as elements of τ. This is the approach taken in the EHDM proof environment [13], [12]. This approach offers considerable expressive power, though it has the disadvantage that our type system will become undecidable if the allowed function definitions are sufficiently general. In the EHDM system, for instance, the user sometimes has to discharge a proof obligation in order to show that a term is well-typed.

(iii) We can incorporate the notion of partiality into the logic itself, by using a logic that takes account of possibly non-denoting terms. Examples of such logics include

- Scott's Logic of Computable Functions, used as the basis of the LCF proof system [8];
- the three-valued Logic of Partial Functions, LPF [1], used in VDM [9] and implemented in the Mural system [10];
- the Logic of Partial Elements, LPE [14], used in earlier versions of the LAMBDA theorem-prover [4];
- the partial functions version, PF, of Church's simple type theory used in the IMPS proof system [6].

These logics are all very expressive, and they all have in common the notion that only certain well-formed terms or formulae are "defined". However, in each of them, some of the familiar logical rules are complicated by extra "definedness" premises. Much time and effort may be expended in propagating and discharging these, even when we are reasoning only about total functions. For instance, in LPE there is a special *existence predicate* E; and we need to prove that $\mathrm{E}\ t$ holds in order to deduce $P(t)$ from $\forall x.\, P(x)$.

In this article we discuss yet another approach, one that combines the practical benefits of a familiar logic, expressive flexibility, and decidable type-checking. The basic idea is as follows: we regard a function application $f(x)$ as *always defined*, but if x is outside the intended domain of f we will not be able to prove anything about its value. In this case we may think of $f(x)$ as an arbitrary or "random" value of the appropriate type. There is an analogy here with the Hilbert choice operator ϵ, familiar to users of HOL and LAMBDA as `@` and `any` respectively: if τ is any inhabited type, then $\epsilon x : \tau . \mathit{false}$ represents an arbitrary value of type τ. Similarly, in our treatment, `hd []`, (where `hd` takes the head of a list), represents an arbitrary value of the appropriate type. However, this expression has the same logical status as any other expression—hence, for example, the theorem

$$\vdash \texttt{hd []} = 5 \;\vee\; \texttt{hd []} \neq 5$$

is provable, even though neither $\vdash \texttt{hd []} = 5$ nor $\vdash \texttt{hd []} \neq 5$ is provable.

This approach to the semantics of partiality has been proposed previously by Abrial, in the context of 'Z' (see [15], Section 4.3). Our goal in this paper is to develop this idea to give a semantics for recursively defined functions, and to show how such functions can be axiomatised within a conventional classical logic. Our motivation is pragmatic rather than philosophical—our aim is not to offer the most satisfactory "ontology" of partial terms, but to address some of the issues that arise in the implementation and use of theorem-provers. We believe our approach has several practical advantages over the others outlined above.

The rest of the article is structured as follows. In Section 2 we introduce the basic idea with the help of examples, and illustrate the kind of reasoning that our scheme supports. This section presents the scheme as it would appear to the user of a theorem-prover. Such a user should also be aware of the limitations highlighted in Section 5. In Section 3 we give the formal details of how, given a general (partial) function definition, we may syntactically generate logical rules to axiomatise a total extension of the function. (From a formalist point of view, such axioms specify the "meaning" of the function definition in that they determine precisely what can and cannot be proved about the function.) This section contains what the builder of a theorem-prover would need to know in order to implement our scheme, (which has been implemented in LAMBDA theorem-prover [5]). In Section 4 we present a naive and intuitive set-theoretic semantics for our function definitions and axioms. This section is addressed to the logician who seeks assurance that the scheme is logically consistent. Furthermore, such a semantics helps to clarify the "intended meaning" of the function definitions, and provides reassurance that the axioms say the things one would expect. In Section 5 we mention some advantages, subtleties, and potential pitfalls associated with our scheme.

The authors would like to thank Matt Fairtlough and Wesley Phoa for their valuable input during discussions on this work, and Cliff Jones for his perceptive and helpful comments.

2 Examples

We begin with some concrete examples. These should convey the basic ideas, and should help to motivate the more technical discussions of Sections 3 and 4.

Our goal is to show how general function definitions can be incorporated into a theorem-prover for classical higher-order logic. For the kind of theorem-prover we have in mind, a user may enter a function declaration,[1] and the system will respond by introducing logical theorems or rules axiomatising its behaviour. Primitive recursive functions are unproblematic; we can introduce a function constant satisfying the recursion equations.

Consider the following definition of the sorting function *quicksort*:

```
fun quicksort []        = []
  | quicksort (h :: t)  =
      case partition h t of
        (before, after) =>
          quicksort before @ [h] @ quicksort after;
```

Here we are assuming that we have already defined a suitable function `partition` (this is primitive recursive and hence total), and that we have managed to prove the following theorem:

```
partition h t = (before, after)  |-
   length before <= length t  /\  length after <= length t
```

Given the `quicksort` definition, the system will introduce a constant `quicksort` to denote the function defined, *plus* an auxiliary predicate `DOM'quicksort`. The expression `DOM'quicksort x` will be used to mean that `x` lies within the intended domain of `quicksort` – the declaration determines the value of `quicksort x` for such `x`, or (more loosely) the function application `quicksort x` is guaranteed to "terminate". The system will then generate axioms stipulating that, for arguments satisfying `DOM'quicksort`, the function `quicksort` obeys its defining clauses. For arguments not satisfying `DOM'quicksort` we say nothing about the result.

Each clause of the definition will in fact generate one axiom for `quicksort` and one for `DOM'quicksort`. The first clause will generate the two axioms:

```
DOM'quicksort([])  |-  quicksort [] = []

|-  DOM'quicksort([]) = TRUE
```

and the second clause will generate the axioms:

[1]Throughout this paper we use an ML-style syntax compatible with LAMBDA v 4.2.1.

```
DOM'quicksort(h :: t)  |-
    quicksort(h :: t)  =
      case partition h t of
        (before, after) =>
          quicksort before @ [h] @ quicksort after

|-  DOM'quicksort(h :: t) =
      forall before, after.
        (before, after) = partition h t ->>
          DOM'quicksort(before) /\ DOM'quicksort(after)
```

In this particular case it so happens that the function we have defined is *total*—the quicksort algorithm always terminates. The above axioms (together with the theorem for `partition`) are enough to allow a user to prove the following theorem:

```
|-  forall k. DOM'quicksort(k)
```

Once we have proved this theorem, we can use it to "prune" the above axioms, so that we need never refer to the `DOM` predicate again. We could also prove theorems asserting that the list returned by `quicksort` is indeed sorted, or that the function `quicksort` is identical to the function defined by some other sorting algorithm:

```
|-  forall k. sorted (quicksort k)

|-  forall k. quicksort k = othersort k
```

Of course, we do not expect the machine to be able to deduce this automatically from the syntactic form of the declaration – although there is scope for tactics to automate such reasoning for common patterns of recursion.

Our scheme also allows us to declare genuinely partial functions in exactly the same way. For example, declaring the difference function

```
fun diff x y = if x = y then 0 else 1 + diff (x + 1) y;
```

generates the axioms

```
DOM'diff#(x,y) |-
    diff x y = if x = y then 0 else 1 + diff (x + 1) y

|- DOM'diff#(x,y) = (x = y = false ->> DOM'diff#(x + 1,y))
```

We can then prove that

```
|-  DOM'diff#(x,y) = x <= y
```

and that

```
x <= y |- x + diff x y = y
```

Of course, there are simpler ways to tackle this example. A more experienced proof hacker might define `diff` as follows:

```
fun diff 0 y         = y
  | diff (S x) (S y) = diff x y
```

Our scheme then gives the more immediately useful axioms

```
|- diff 0 y = y

|- diff (S x) (S y) = diff x y
```

Because we use primitive recursion, all `DOM` predicates can be optimised away automatically.

The `DOM` predicate does duty for the `E` predicate of LPE (for instance), and in the case of a genuinely partial function we would still expect to have to spend time propagating and discharging `DOM` conditions. Notice the gains, however: for total functions we can prove totality once for all and then forget about the `DOM` predicates (as above); and even for partial functions we only need the `DOM` predicates for reasoning that depends specifically on the *value* of a function application. Thus, for instance, we can prove the theorem

```
forall x. P(x)  |-  P(f 0)
```

without mentioning the predicate `DOM'f` at all. Moreover, we have retained the familiarity and simplicity of (two-valued) classical logic. (For a discussion of some possible drawbacks of our approach, see Section 5.3.)

3 Axiomatising partial functions

In this section we describe in detail the general procedure whereby the axiomatisation of a function can be derived syntactically from its declaration (we want the theorem-prover to be able to do this automatically). We here assume that, as in HOL or LAMBDA, we are reasoning only about non-empty *well-founded* datatypes, which may be modelled semantically as free algebras – thus we exclude from consideration certain ML types such as

```
datatype d = D of d -> d
```

Suppose the general form of a legal function declaration to be

$$
\begin{array}{l}
\texttt{fun f}\ pat_{11}\ \ldots\ pat_{1n}\ =\ exp_1 \\
\quad\texttt{| f}\ pat_{21}\ \ldots\ pat_{2n}\ =\ exp_2 \\
\quad\ \ldots \\
\quad\texttt{| f}\ pat_{m1}\ \ldots\ pat_{mn} =\ exp_m
\end{array}
$$

where the pat_{ij} are ML-style patterns (which may contain variables), and the exp_i are expressions of higher-order logic. There is no notion of sequential execution here, so the order of the clauses should not matter; hence we stipulate that no two clauses may possess overlapping patterns. However, we do *not* require that the patterns should be exhaustive. Moreover, there is no restriction on the ways in which `f` can occur in the exp_i. (This is in contrast to primitive recursion, for instance, where `f` may only take arguments that are syntactically "smaller" than the corresponding LHS patterns.) We permit all partial recursive function definitions.

Given a declaration of this form, we introduce an auxiliary predicate `DOM'f`. We then use each of the m clauses of the definition to produce one axiom for `f` and one axiom for `DOM'f`. We can think of `DOM'f` $x_1 \ldots x_n$ as meaning (very roughly)[2] that the function application `f` $x_1 \ldots x_n$ is guaranteed to terminate. A better intuition is that the `DOM` predicate characterises those arguments for which the value of the function is determined. Our axiomatisation of `f` is incomplete in that we say nothing about the values of the function for other arguments. Providing the predicate `DOM'f` holds, we can reason about the value of the function call by using the corresponding clause of the definition. By contrast, if the function call does *not* terminate then it is not possible to reason about the value; our semantics assigns an arbitrary value to the result. This is how we avoid paradox in the case of functions such as

```
fun g x = g x + 1
```

The axioms for `f` simply say that the function obeys its defining clause on all arguments for which it is known that the function call terminates: for each $1 \leq i \leq m$ we have an axiom

$$\texttt{DOM'f}(pat_{i1},\ldots,pat_{in})\quad \texttt{|-}\quad \texttt{f}\ pat_{i1}\ \ldots\ pat_{in}\ \texttt{=}\ exp_i \qquad (\dagger)$$

The axioms for `DOM'f` give conditions on $x_1, \ldots, x_n$ that are sufficient to guarantee that `f` $x_1\ \ldots\ x_n$ will terminate. To specify how the form of these axioms is determined, we need to give a syntactic way of transforming an expression exp into a formula $\ll exp \gg$ – informally, we think of $\ll exp \gg$ as meaning "the evaluation of exp is guaranteed to terminate". We then take as our axioms for `DOM'f` the equations

[2]This informal interpretation of `DOM'f` is reasonable only when all calls to previously-defined functions also terminate. The semantics of Section 4 gives a precise interpretation for `DOM'f`. See Section 5.3 for further discussion of the differences between our `DOM` predicates and termination in Standard ML.

$$\texttt{|-}\quad \texttt{DOM'f(}pat_{i1},\ldots,pat_{in}\texttt{)} \texttt{ = } \ll exp_i \gg \qquad (\ddagger)$$

for $1 \leq i \leq m$. [3] We choose to use equality here, rather than the implication

$$\texttt{|-}\quad \ll exp_i \gg \;\rightarrow\; \texttt{DOM'f(}pat_{i1},\ldots,pat_{in}\texttt{)}$$

for pragmatic reasons: we want to use rewriting to eliminate `DOM` predicates.

There are many different ways in which $\ll \_ \gg$ could be defined, depending on how we wished to interpret the phrase "is guaranteed to terminate". Firstly, whether a given call terminates might depend on semantic choices—for instance, on whether we are thinking of call-by-name or call-by-value semantics. Secondly, we must decide how to interpret "termination" for non-executable logical constructs such as quantifiers. Finally, we do not insist that the `DOM` predicate represents a *necessary* condition for termination—we thus have some freedom in deciding how much of the termination behaviour we want to capture. (We discuss this point more fully in Section 5.3.) The definition below illustrates one way of responding to these issues. However, the same idea applies to other variants, and indeed to logical constructs other than those we consider here.

We define the operator $\ll \_ \gg$ inductively on terms of our logic as follows:[4]

(i) Atomic expressions other than occurrences of `f` always terminate. Thus if x is any atom other than `f`:

$$\ll x \gg = \texttt{TRUE}$$

(ii) For each occurrence of `f`, we generate a call to `DOM'f`. If the occurrence of `f` is not "fully applied", then we introduce extra, universally quantified, variables for the missing arguments. Thus for $0 \leq k \leq n$:

$$\begin{aligned}&\ll \texttt{f}\ e_1\ \ldots\ e_k \gg = \\ &\qquad \ll e_1 \gg \ \texttt{/\textbackslash} \ldots \texttt{/\textbackslash}\ \ll e_k \gg \ \texttt{/\textbackslash} \\ &\qquad \texttt{forall}\, x_{k+1},\ldots,x_n.\ \texttt{DOM'f(}e_1,\ldots,e_k,x_{k+1},\ldots,x_n\texttt{)}\end{aligned}$$

(iii) Other function applications are treated "structurally".[5] Thus, for e not of the form `f` $e_1\ \ldots\ e_k$ for $k < n$:

$$\ll e\ e' \gg = \ll e \gg \ \texttt{/\textbackslash}\ \ll e' \gg$$

[3] In both these sets of axioms, all atoms occurring in the pat_{ij} are considered as free variables. For simplicity, we assume that the theorem-prover is able to distinguish these from *constants* which may have the same name.

[4] In applying these clauses to generate the axioms for the examples in Section 2, we have taken the liberty of "optimising out" a few redundant conjuncts.

[5] One case deserves special mention: if e' is `f`, the condition we obtain here is stricter than, say, termination of the corresponding ML program.

(iv) Case expressions are treated in a way which takes account of the "flow of control" within the expression. (Here, for simplicity, the patterns in a `case` construct are assumed to be non-overlapping and exhaustive.[6])

$$\begin{aligned}
&\ll \texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ldots\texttt{|}\ p_k\ \texttt{=>}\ e_k \gg\ = \\
&\quad \ll e \gg\ \texttt{/\textbackslash} \\
&\quad (\texttt{forall}\,\overline{x_1}.\ p_1\ =\ e\ \texttt{->>}\ \ll e_1 \gg)\ \texttt{/\textbackslash} \\
&\quad \ldots \\
&\quad (\texttt{forall}\,\overline{x_k}.\ p_k\ =\ e\ \texttt{->>}\ \ll e_k \gg)
\end{aligned}$$

where $\overline{x_i}$ stands for all the variables occurring in the pattern p_i.

(v) A variable-binding expression is considered to terminate iff the body of the expression terminates for all values of the bound variable. Thus:

$$\ll Q\,x.\ e \gg\ =\ \texttt{forall}\ x.\ \ll e \gg$$

where Q is any binder (i.e. `forall`, `exists`, `fn`, `any`). (Note that this corresponds to a very strict interpretation of the binders. More generous versions are also possible, see Section 5.1.)

(vi) Other expressions are treated structurally—we simply conjoin together the `DOM` calls from the various subexpressions. For example:

$$\ll \texttt{(}e\texttt{,}e'\texttt{)} \gg\ =\ \ll e \gg\ \texttt{/\textbackslash}\ \ll e' \gg$$

Note that clauses (ii) and (iii) embody the choice of *applicative-order* (call-by-value) semantics. The above scheme can be extended in an obvious way to cover *mutually* recursive functions—each mutually recursive function call generates a call to the corresponding `DOM` predicate.

4 Set-theoretic semantics

We now outline a semantics for our system. As mentioned earlier, there are at least two reasons for wanting to do this—to establish the consistency of our scheme, and to provide a framework for thinking about our functions and axioms in an intuitive way. The consistency is by no means obvious—there certainly are some logical principles that do apply to non-terminating function calls, and we might be worried that these might allow us to deduce more than we really intend in the case of some pathological function definition. We prove that our system actually is consistent by interpreting our function definitions in some model, and checking

[6]Exhaustiveness is not essential and is not required in the LAMBDA implementation. We impose it since it simplifies our presentation of the semantics.

that the corresponding axioms are valid under this interpretation (and that our rules of inference are sound).

Our model is very elementary and purely set-theoretic—it allows us to think of a type simply as a set, and a term of that type as a member of the set. A function definition is then interpreted by just an ordinary function between sets—the `DOM` predicate is modelled by (the characteristic function of) a subset of its domain. In our view, it is a virtue of our logic that it can be modelled without using any domain theory or category theory. Our contention is that such a logic will be more easily grasped by a non-specialist than a logic for which the models are necessarily mathematically abstruse.

4.1 Interpretation of types

The interpretation $[\![\_]\!]$ for the types of our logic is defined inductively as follows.

(i) The basic types are interpreted as sets in the obvious way, e.g.

$$\begin{array}{lcl} [\![\texttt{bool}]\!] & = & \{\mathit{true}, \mathit{false}\} \\ [\![\texttt{nat}]\!] & = & \{0, 1, 2, \ldots\} \end{array}$$

(ii) Function types are interpreted by set-theoretic *total* function spaces:

$$[\![\, t \texttt{->} t' \,]\!] \;=\; [\![\, t \,]\!] \to [\![\, t' \,]\!]$$

(iii) Product types are interpreted by cartesian products:

$$[\![\, t \texttt{*} t' \,]\!] \;=\; [\![\, t \,]\!] \times [\![\, t' \,]\!]$$

Similarly for other ways of forming new non-empty types (sums, quotients, non-empty subtypes etc.)

(iv) Inductive datatypes are interpreted by sets constructed as free algebras—we omit the easy details. (In fact, we can allow any ways of defining types that have natural interpretations as non-empty sets, e.g. inductive types modulo equational constraints.)

Henceforth we will identify the booleans with the truth-values in our (classical) meta-logic, so that for instance we write $(u = v)$ in place of the expression

$$\begin{cases} \mathit{true} & \text{if } u = v \\ \mathit{false} & \text{otherwise} \end{cases}$$

4.2 Interpretation of terms

We will define the interpretation of a term e in an environment E (that is, a finite function that assigns interpretations to all the constants and free variables appearing in e). For convenience, we assume here that our logic is *explicitly typed*—all constants and free variables have some specific type, and all variable-bindings carry explicit type annotations. We say that an environment E is *well-typed* if for all $x \in \mathrm{dom}\ E$ we have $E(x) \in [\![\, t \,]\!]$, where t is the type of x.

In fact, for the interpretation of function definitions, we need both an environment E and a *function-environment* F, whose role is explained in Section 4.3. We use the notation

$$E\,;F \vdash e \Rightarrow v$$

to mean "v is the interpretation of e in environment E and function-environment F". This notation allows us to present the semantics in an "operational" style by means of inference rules. The following rules define interpretations for all well-typed expressions other than those occurring as part of a function definition. (Note that F plays no active role in any of these rules.)

(i) $E\,;F \vdash \overline{n} \Rightarrow n \qquad (n = 0, 1, 2, \ldots)$

where $\overline{n}$ is the numeral for n. Similarly for the other basic types.

(ii) $E\,;F \vdash x \Rightarrow E(x) \qquad (x \in \mathrm{dom}\ E;\ x \notin \mathrm{dom}\ F)$

(iii) $$\frac{E\,;F \vdash e \Rightarrow v \qquad E\,;F \vdash e' \Rightarrow v'}{E\,;F \vdash \texttt{(}e\texttt{,}e'\texttt{)} \Rightarrow (v, v')}$$

and similarly for the term constructors for other type-forming operations.

(iv) $$\frac{E\,;F \vdash e \Rightarrow u \qquad E\,;F \vdash e' \Rightarrow v}{E\,;F \vdash e\ e' \Rightarrow u(v)} \quad (v \in \mathrm{dom}\ u)$$

(we will see later that the side-condition is redundant)

(v) $$\frac{E\,;F \vdash e \Rightarrow u \quad E'\,;F \vdash p_i \Rightarrow u \quad EE'\,;F \vdash e_i \Rightarrow v}{E\,;F \vdash \texttt{case}\ e\ \texttt{of}\ p_1\texttt{=>}e_1\texttt{|}\ldots\texttt{|}p_k\texttt{=>}e_k \Rightarrow v} \quad (\mathrm{dom}\, E' = FV(p_i))$$

where $1 \leq i \leq k$, and EE' means the environment in which the bindings in E are enriched (and possibly overwritten) by those in E'.

(vi) $$\frac{E\,;F \vdash e \Rightarrow v \qquad E\,;F \vdash e' \Rightarrow v'}{E\,;F \vdash e\texttt{=}e' \Rightarrow (v = v')}$$

(vii) $$\frac{E\,;F \vdash e \Rightarrow v \qquad E\,;F \vdash e' \Rightarrow v'}{E\,;F \vdash e\,\texttt{/\\}\,e' \Rightarrow v \wedge v'}$$

and similarly for the other propositional connectives.

(viii) $$\frac{E(x,v)\,;F \;\vdash\; e \;\Rightarrow\; u(v) \qquad \text{for each } v \in [\![\, t \,]\!]}{E\,;F \;\vdash\; \texttt{fn}\ x\texttt{:}t\texttt{=>}e \;\Rightarrow\; u} \quad (\mathrm{dom}\ u = [\![\, t \,]\!])$$

(ix) $$\frac{E(x,v)\,;F \;\vdash\; e \;\Rightarrow\; u(v) \qquad \text{for each } v \in [\![\, t \,]\!]}{E\,;F \;\vdash\; \texttt{any}\ x\texttt{:}t.\ e \;\Rightarrow\; \mathit{choose}_t\ u} \quad (\mathrm{dom}\ u = [\![\, t \,]\!])$$

where for each type t we have a semantic choice function choose_t satisfying $u(\mathit{choose}_t\ u) = \mathit{true}$ for all $u \neq \lambda v.\mathit{false}$.

(x) $$\frac{E(x,v)\,;F \;\vdash\; e \;\Rightarrow\; u(v) \qquad \text{for each } v \in [\![\, t \,]\!]}{E\,;F \;\vdash\; \texttt{forall}\ x\texttt{:}t.\ e \;\Rightarrow\; (u = \lambda v.\mathit{true})} \quad (\mathrm{dom}\ u = [\![\, t \,]\!])$$

and dually for `exists`.

Note that in general (viii), (ix) and (x) will be *infinitary* rules, so our derivation trees may be infinite. (However, they will always have finite depth, by induction on the size of e.) The following are now easy observations.

Proposition 1
(i) Given any well-typed environment E, function-environment F and expression e, where dom E *contains all constants and free variables appearing in e, there is a unique value v such that $E\,;F \vdash e \Rightarrow v$ is derivable using the above rules.*
(ii) If $E\,;F \vdash e \Rightarrow v$ is derivable and e has type t, then $v \in [\![\, t \,]\!]$.

Proof. By simultaneous induction on the structure of e. If e is a `case` expression, then rule (v) applies for some unique value of i, since the patterns are exhaustive and non-overlapping. If e is an application, then the side-condition in rule (iv) automatically holds since, by the induction hypothesis, u belongs to the appropriate total function space. □

Definition 2 *If $E\,;F \vdash e \Rightarrow v$ is derivable, we write $[\![\, e \,]\!]_{E;F} = v$. We also write $[\![\, e \,]\!]_E$ in place of $[\![\, e \,]\!]_{E;\langle\rangle}$, and $[\![\, e \,]\!]$ in place of $[\![\, e \,]\!]_{\langle\rangle;\langle\rangle}$.*

Note that, by Proposition 1*(i)*, this defines $[\![\, e \,]\!]$ for all *closed* terms e.

4.3 Interpretation of functions

We now give the interpretation of `fun` definitions. As in Section 2 we consider just a single declaration of the form

$$\begin{array}{lllcl}
\texttt{fun f} & pat_{11} & \ldots\ pat_{1n} & = & exp_1 \\
\quad\texttt{| f} & pat_{21} & \ldots\ pat_{2n} & = & exp_2 \\
\quad\ \ldots & & & & \\
\quad\texttt{| f} & pat_{m1} & \ldots\ pat_{mn} & = & exp_m
\end{array}$$

(mutual recursion can be dealt with similarly). We will find it convenient to distinguish notationally between this declaration and the function `f` it declares—we thus use Δ to denote the piece of syntax constituting the above declaration.

Our goal is to give definitions of $[\![\, \mathtt{f} \,]\!]$ and $[\![\, \mathtt{DOM'f} \,]\!]$. We regard the "meaning" of Δ as being given by these two components: $[\![\, \Delta \,]\!] = ([\![\, \mathtt{f} \,]\!], [\![\, \mathtt{DOM'f} \,]\!])$. Suppose that the pat_{ij} are patterns of type s_j and the exp_i are expressions of type t. Then we expect to obtain

$$\begin{array}{lll} [\![\, \mathtt{f} \,]\!] & \in & [\![\, s_1 \,]\!] \to \ldots \to [\![\, s_n \,]\!] \to [\![\, t \,]\!] \\ [\![\, \mathtt{DOM'f} \,]\!] & \in & ([\![\, s_1 \,]\!] \times \ldots \times [\![\, s_n \,]\!]) \to \mathit{bool} \end{array}$$

The first step in defining these is to transform the declaration syntactically into a form that we call *first-order*. We say the declaration Δ is first-order provided every occurrence of $\mathtt{f}$ in the exp_i is fully applied (i.e. occurs at the head of some subexpression $\mathtt{f}\, e_1 \ldots e_n$). A general declaration Δ may be translated into a first-order one by eta-expanding occurrences of $\mathtt{f}$ in the exp_i, i.e. replacing every subexpression $\mathtt{f}\, e_1 \ldots e_k$ $(0 \leq k < n)$ that is not itself an applicand by

$$\mathtt{(fn}\ x_{k+1}, \ldots, x_n\ \mathtt{=>}\ \mathtt{f}\ e_1\ \ldots\ e_k\ x_{k+1}\ \ldots\ x_n\mathtt{)}$$

We write $\hat{\Delta}$ to denote the transformed declaration—we also write $\widehat{exp}_i$ for the translations of the exp_i. (Note that these depend implicitly on the atom $\mathtt{f}$ and the value of n). We define the meaning of a general declaration to be the same as the meaning of its first-order expansion: $[\![\, \Delta \,]\!] = [\![\, \hat{\Delta} \,]\!]$.[7] For the rest of this subsection we will therefore assume that Δ is first-order.

We now explain the role of the function-environment F introduced earlier. Let us write $S = [\![\, s_1 \,]\!] \times \ldots \times [\![\, s_n \,]\!]$ and $T = [\![\, t \,]\!]$; suppose now that we are given a set-theoretic *partial* function $\theta : S \rightharpoonup T$. We can regard θ as an "approximation" to (an uncurried form of) $\mathtt{f}$. We express this intention by putting the pair $(\mathtt{f}, \theta)$ in the function-environment. A function-environment is thus a finite map taking function-names to partial functions between the appropriate sets.[8] The idea is now to "interpret" $\mathtt{f}$ by θ. To do this, we augment the inference rules of Section 4.2 with the rule

$$(\star) \quad \frac{E\,;F\ \vdash\ e_1\ \Rightarrow\ v_1 \quad \cdots \quad E\,;F\ \vdash\ e_n\ \Rightarrow\ v_n}{E\,;F\ \vdash\ \mathtt{f}\ e_1\ \ldots\ e_n\ \Rightarrow\ \theta(v_1, \ldots, v_n)} \quad (\mathtt{f}, \theta) \in F;\ \ (v_1, \ldots, v_n) \in \operatorname{dom} \theta$$

This is the only rule that "introduces" partiality into the semantics—it assigns meanings to certain fully applied occurrences of $\mathtt{f}$ but not to others. There is no clash here with rule 4.2(iv) for application, since there is no value v such that $E\,;F\ \vdash\ \mathtt{f}\ e_1\ \ldots\ e_{n-1}\ \Rightarrow\ v$ because we have not assigned any independent meaning to $\mathtt{f}$. We can now prove the following properties:

[7] Thus, our semantics is as strict for `fn`-expressions as it is for their eta-reducts. This is not the case for the semantics of Standard ML, for instance.

[8] In fact, since here we are not handling mutual recursion, we never require F to contain more than one entry.

Proposition 3

(i) Given any well-typed environment E, function-environment F, and expression e (possibly containing occurrences of $\mathtt{f}$), there is at most one *value v such that $E\,; F \vdash e \Rightarrow v$ is derivable using the rules of 4.2 together with the rule $(\star)$.*

(ii) If $E\,; F \vdash e \Rightarrow v$ is derivable in this system, and e has type t, then $v \in [\![\, t \,]\!]$.

(iii) If $E\,; F(f, \theta) \vdash e \Rightarrow v$ is derivable and $\theta' : S \rightharpoonup T$ extends θ, then also $E\,; F(f, \theta') \vdash e \Rightarrow v$ is derivable.

Proof. *(i)* and *(ii)* are proved by simultaneous induction, as in Proposition 1. For *(iii)*, just observe that any valid derivation remains valid when θ is replaced everywhere by θ'. □

We now show how, given an approximation θ to $\mathtt{f}$, we can construct a "next approximation" $\mathcal{F}\theta$. We suppose here that D is an environment giving interpretations for all the constants already in scope at the point of declaration of $\mathtt{f}$.

Definition 4 *Let $\mathcal{F} : (S \rightharpoonup T) \to (S \rightharpoonup T)$ be defined as follows. Given $\theta : S \rightharpoonup T$ and $(v_1, \ldots v_n) \in S$, define $\mathcal{F}\theta(v_1, \ldots, v_n) = w$ iff, for some clause,*

$$\mathtt{f}pat_{i1} \;\ldots\; pat_{in} \;=\; exp_i$$

in Δ, where $1 \leq i \leq m$,
and some environment E with dom $E = FV(pat_{i1}, \ldots, pat_{in})$*, we have*

(a) $[\![\, pat_{ij} \,]\!]_E = v_j$ for each $1 \leq j \leq n$; and moreover

(b) $DE\,; (\mathtt{f}, \theta) \vdash exp_i \Rightarrow w$.

Note that, in the above definition, the value of i is necessarily unique since the patterns are non-overlapping. We think of E here as the environment generated by the pattern-matching—pattern-matching process determines E uniquely for the free algebras we use to model datatypes. Given this, it follows from Proposition 3*(i)* that w, if it exists, is unique.

Proposition 5

(i) If $\theta \subseteq \theta'$ then $\mathcal{F}\theta \subseteq \mathcal{F}\theta'$, where $\subseteq$ is the inclusion ordering on $S \rightharpoonup T$.

(ii) $\mathcal{F}$ has a least fixed point $\theta_\infty = \mathcal{F}\theta_\infty$.

Proof. *(i)* follows immediately from Definition 4 and Proposition 3*(iii)*. *(ii)* is then an application of Tarski's fixed point theorem. Note that since $S \rightharpoonup T$ is not a complete lattice, we need to use transfinite iteration to construct the least fixed point. Note also that iteration up to ω will not in general suffice, because of the presence of the infinitary rules. □

Definition 6 *Suppose the declaration Δ is first-order. Then, with θ_∞ as above,*

(i) let $[\![\, \mathtt{f} \,]\!]$ be (the curried form of) any total function $S \to T$ extending θ_∞;

(ii) let $[\![\, \mathtt{DOM'f} \,]\!] : S \to bool$ be the characteristic function of dom θ_∞.

Note that we have used only the graph ordering on $S \rightharpoonup T$ rather than any sophisticated domain ordering. Furthermore, we have not used any notion of *continuity*—indeed the operator $\mathcal{F}$ will not in general be continuous!

4.4 Validity of axioms

We must now check that the axioms discussed in Section 3 are valid under the interpretation we have given. Recall that for each $1 \leq i \leq m$ we have an axiom (†) for `f` and an axiom (‡) for `DOM'f`. We sketch the proofs that these are valid, omitting some of the details. We use identifiers such as Δ and θ_∞ with the same meanings as in Section 4.3. Throughout this section, F will denote the function-environment $\langle(\mathtt{f}, \theta_\infty)\rangle$, and D will denote the "current environment" at the point of declaration.

We begin by restricting our attention to first-order declarations—later we show that the results for the general case may be reduced to those for the first-order case. The axioms for `f` are easily shown to be valid:

Proposition 7 *Suppose Δ is first-order. Then the axioms*

$$\mathtt{DOM'f(}pat_{i1}, \ldots, pat_{in}\mathtt{)} \quad \mathtt{|-} \quad \mathtt{f}\ pat_{i1}\ \ldots\ pat_{in}\ \mathtt{=}\ exp_i \qquad (\dagger)$$

for `f` are valid under the given interpretations of `f` and `DOM'f`.

Proof. It is easily seen from the definitions of $[\![\,\mathtt{f}\,]\!]$ and $[\![\,\mathtt{DOM'f}\,]\!]$ that (†) is valid iff, for any environment E giving interpretations to the pattern-variables, we have

$$\begin{aligned}&([\![\,pat_{i1}\,]\!]_E, \ldots, [\![\,pat_{in}\,]\!]_E) \in \operatorname{dom} \theta_\infty \implies \\ &\qquad \theta_\infty\,([\![\,pat_{i1}\,]\!]_E, \ldots, [\![\,pat_{in}\,]\!]_E) = [\![\,exp_i\,]\!]_{DE;F}\end{aligned}$$

But this assertion is indeed true in our model, since from Definition 4 (and the remark following it) it is easy to see that

$$[\![\,exp_i\,]\!]_{DE;F} = \mathcal{F}\theta_\infty\,([\![\,pat_{i1}\,]\!]_E, \ldots, [\![\,pat_{in}\,]\!]_E)$$

(where if either side is defined then so is the other). □

The corresponding theorem for the `DOM'f` axioms is somewhat harder. Let us write $E\,; F \vdash e\downarrow$ to mean "$[\![\,e\,]\!]_{E;F}$ is defined" (i.e. $E\,; F \vdash e \Rightarrow v$ for some v). The following lemma says that this property behaves formally like the operator $\ll \_ \gg$ defined in Section 2.

Lemma 8 *Let e be any first-order expression (relative to `f` and n), and suppose E is any environment giving interpretations to all the constants and free variables of e (other than `f`). Then $E\,; F \vdash e\downarrow$ iff $[\![\ll e \gg]\!]_{E;F} = true$.*

Proof. By induction on the structure of e. We consider only a selection of the cases.

- If e is an atom other than $\mathtt{f}$, then $[\![\ll e \gg]\!]_{E;F} = \mathit{true}$ always; likewise $E\,;F \vdash e\downarrow$ always since $[\![e]\!]_{E;F} = E(e)$.

- If e is of the form $\mathtt{f}\ e_1\ \ldots\ e_k$ with $k \leq n$, then $k = n$ by the assumption that e is first-order. But now

$$\begin{array}{rll} E\,;F \vdash \mathtt{f}\ e_1\ \ldots\ e_n\downarrow & \text{iff} & E\,;F \vdash e_i\downarrow \quad \text{for } 1 \leq i \leq n \quad \text{and} \\ & & ([\![e_1]\!]_{E;F},\ \ldots\ ,[\![e_n]\!]_{E;F}) \in \text{dom}\ \theta_\infty \\ & \text{iff} & [\![\ll e_i \gg]\!] = \mathit{true} \quad \text{for } 1 \leq i \leq n \quad \text{and} \\ & & [\![\mathtt{DOM'f}(e_1, \ldots, e_n)]\!] = \mathit{true} \\ & \text{iff} & [\![\ll \mathtt{f}\ e_1\ \ldots\ e_n \gg]\!] = \mathit{true} \\ & & \qquad \text{by definition of } \ll \_ \gg \end{array}$$

- If e is of the form $e'\ e''$, where e' is not of the form $\mathtt{f}\ e_1\ \ldots\ e_k$ with $k < n$, then by rule (iv) of Section 4.2 we have

$$\begin{array}{rll} E\,;F \vdash e'\ e''\downarrow & \text{iff} & E\,;F \vdash e'\downarrow \quad \mathit{and} \quad E\,;F \vdash e''\downarrow \\ & \text{iff} & [\![\ll e' \gg]\!] = [\![\ll e'' \gg]\!] = \mathit{true} \\ & \text{iff} & [\![\ll e'\ e'' \gg]\!] = \mathit{true} \end{array}$$

- If e is of the form $Q\,x.\ e'$ (Q a binder), then by rules (viii) and (ix) of Section 4.2 we have

$$\begin{array}{rll} E\,;F \vdash Q\,x : t.\ e'\downarrow & \text{iff} & E(x,v)\,;F \vdash e'\downarrow \quad \text{for all } v \in [\![t]\!] \\ & \text{iff} & [\![e']\!]_{E(x,v);F} = \mathit{true} \quad \text{for all } v \in [\![t]\!] \\ & \text{iff} & [\![\ll Q\,x : t.\ e' \gg]\!] = \mathit{true} \end{array}$$

The other cases are handled similarly using the appropriate clauses in the definition of $\ll \_ \gg$. □

Proposition 9 *Suppose Δ is first-order. Then the axioms*

$$\texttt{|-}\ \ \mathtt{DOM'f}(\mathit{pat}_{i1}, \ldots, \mathit{pat}_{in}) \ \texttt{=}\ \ll \mathit{exp}_i \gg \qquad (\ddagger)$$

for `DOM'f` *are valid under the given interpretations of* `f` *and* `DOM'f`*.*

Proof. Clearly (‡) is valid provided, for any environment E giving interpretations to the pattern-variables, we have

$$([\![\mathit{pat}_{i1}]\!]_E, \ldots, [\![\mathit{pat}_{in}]\!]_E) \ \in\ \text{dom}\ \theta_\infty \quad \text{iff} \quad [\![\ll \mathit{exp}_i \gg]\!]_{DE;F} = \mathit{true}$$

But this holds in our model, since

$$\begin{array}{lll} [\![\ll exp_i \gg]\!]_{DE;F} = true & \text{iff} & DE\,;F \vdash exp_i \downarrow \quad \text{by Lemma 8} \\ & \text{iff} & ([\![pat_{i1}]\!]_E, \ldots, [\![pat_{in}]\!]_E) \in \text{dom } \mathcal{F}\theta_\infty \\ & & \qquad \text{by definition of } \mathcal{F} \end{array}$$

and dom $\mathcal{F}\theta_\infty$ = dom θ_∞. □

Finally we need the following proposition, which says that the general case may be reduced to the first-order case:

Proposition 10 *Let Δ be a general declaration. If the axioms generated from $\hat{\Delta}$ are valid under the given interpretation for $[\![\texttt{f}]\!]$ and $[\![\texttt{DOM'f}]\!]$, then the axioms generated from Δ are likewise valid.*

Proof. Note first that Δ and $\hat{\Delta}$ yield the same values for $[\![\texttt{f}]\!]$ and $[\![\texttt{DOM'f}]\!]$ by definition. For the axioms (†) for `f`, we need the fact that

$$[\![\widehat{exp_i}]\!]_{D'} = [\![exp_i]\!]_{D'}$$

where D' is the "updated" environment $D\,(\texttt{f}, [\![\texttt{f}]\!])$. But this follows easily by induction from the fact that the eta-rule is valid in our model:

$$[\![\texttt{f}\ e_1 \ \ldots\ e_k]\!]_{D'} = [\![\texttt{fn}\ x_{k+1}, \ldots, x_n\ \texttt{=>}\ \texttt{f}\ e_1 \ \ldots\ e_k\ x_{k+1}\ \ldots\ x_n]\!]_{D'}$$

For the axioms (‡), it suffices to note that the predicates $\ll exp_i \gg$ and $\ll \widehat{exp_i} \gg$ are syntactically identical—this follows by induction from the fact that

$$\ll \texttt{f}\ e_1 \ \ldots\ e_k \gg = \ll \texttt{fn}\ x_{k+1}, \ldots, x_n\ \texttt{=>}\ \texttt{f}\ e_1\ \ldots\ e_k\ x_{k+1}\ \ldots\ x_n \gg \qquad \square$$

This completes the proof that our axioms are valid. In order to show that our system was consistent, we would also need to check that the inference rules of our logic were *sound* for our model. For any reasonable formulation of higher-order logic, this would be completely routine.

5 Further remarks

We conclude with some miscellaneous remarks concerning the strengths, subtleties and weaknesses of our approach. We also mention some areas where there is scope for further work.

5.1 Quantifiers and `any`

Our treatment of the binding operators in Section 3(v) deserves further comment. The behaviour of `any` in combination with recursion is especially subtle.

For the quantifiers `forall` and `exists` we could have given a more generous interpretation—for instance, in Section 3 we could have defined

$$\begin{aligned}\ll \texttt{exists}\ x.\ e \gg \ &=\ (\texttt{forall}\ x.\ \ll e \gg)\ \texttt{\textbackslash/} \\ &\quad\ (\texttt{exists}\ x.\ \ll e \gg \texttt{/\textbackslash}\ e\,)\end{aligned}$$

(with corresponding changes to the semantic clauses in Section 4.2).

However, in the case of `any` we are forced to take a very strict interpretation. If in the semantics of Section 4 we allowed $[\![\, \texttt{any}\ x.e \,]\!]_{E;F}$ to have a value when not all $[\![\, e \,]\!]_{E(x,v);F}$ were defined, then this value might change as the recursion unwound and new values of $[\![\, e \,]\!]$ "arrived". This would cause the operator $\mathcal{F}$ to be non-monotonic and we would not be able to obtain a fixed point.

Hence, although the following function definition is legal in our scheme, we cannot prove it defines the identity function (even though the identity is the only total function satisfying its two clauses):

```
fun f 0     = 0
  | f (S n) = (any x. f x = n) + 1
```

There are other uses of `any` in function definitions that behave more as one would expect. For instance, the reader is invited to check that the following function *can* be shown to be the identity:

```
fun id 0     = 0
  | id (S n) = any x. (id n < x) = true /\ (x <= S n) = true
```

5.2 Advantages of our scheme

As mentioned in Section 2, one advantage of our approach is that it does not impose unnecessary proof obligations on users who wish to work only with total functions. Once we have proved a theorem asserting totality of a function, we can forget about the `DOM` predicates completely. Indeed, if our theorem-prover detects that some function definition is primitive recursive (for instance), the `DOM` predicates need not even be generated, and simpler forms of the axioms will suffice.[9]

Moreover, even when reasoning about non-total functions, it seems that proofs in our logic involve fewer definedness conditions than proofs in other systems (see Section 2). This suggests that proofs in our logic may be less time-consuming.

[9]The LAMBDA system incorporates this feature. The axioms generated in this case are similar to the theorems generated by a HOL package.

The gain stems from the fact that the `DOM` predicates do not feature at all in the rules of the pure logic—they are no more than particular object-level constants.

Of course, users will still need to prove `DOM` conditions sometimes. But the analogy with "termination" of the corresponding program suggests a way in which automated assistance for proving `DOM` conditions might be provided, at least for the "executable" fragment of the logic. A machine could try to prove `|- DOM'f` e by simulating the execution of `f` e—if this terminated then the course of computation would yield a proof of `DOM'f` e. Even if e is not a concrete value but an expression containing variables, it might still be possible to achieve a similar result by running the computation "symbolically". Of course, this would give us at best a *semi-decision* procedure, since such a proof tactic would diverge if applied to an unprovable goal. Also, there are limitations to the analogy between `DOM` predicates and termination (see below). Nevertheless, we feel that proof tactics along these lines might be of some benefit to users.

5.3 Limitations of our scheme

In Section 4 we gave a proof of the logical consistency of our system. However, in practice we are interested in more than this—we would like to know that there is a "good fit" between our formalism and the real-world situations that we are using it to describe. If this relationship has been misunderstood by the user, theorems generated using the theorem-prover may be misinterpreted as expressing false (or unproven) statements about the real world. In this subsection we seek to explain more fully the "intended use" of our logic, and to point out some of the possible pitfalls.

There are essentially two different ways in which partial function declarations can be understood: as definitions of genuinely partial functions between the relevant domains, or as loose specifications of *total* functions. Logics such as LPF [1], which has a three-valued semantics, fit well with the former interpretation, whereas our scheme fits more comfortably with the latter, as will be clear from the semantics we have given.

The distinction between these interpretations does not matter much to us if we adhere to the principle of writing function applications only in contexts where they are "guarded" by conditions ensuring their well-definedness. For instance, the predicate

```
(y > 0) = true  /\  x div y = 1
```

has the same truth-value in our scheme as in LPF for any assignment of values to `x` and `y`. It seems likely that there is a metatheorem asserting that the two interpretations agree for certain "guarded" formulae. In specification languages such as 'Z', writing such formulae would seem to be good practice in any case.

The "loose specification" interpretation may also be appropriate in the context of hardware verification. Suppose we wish to reason about a hardware component whose behaviour is only specified for certain input values. Typically, the design of such a component will ensure that it still exhibits *some* determinate behaviour even on "incorrect" inputs, though what this behaviour is may not be specified. In this case, our scheme would be particularly suitable—the conclusions derived in the logic would be correct for *any* component satisfying the specification.

However, there are problems if we try to use our logic to reason about the behaviour of programs. Although in Sections 2 and 3 we used program termination as an analogy to help motivate our scheme, this analogy should not be pressed too far. A non-terminating program is not the same as a program returning an unknown value, but our semantics is not capable of reflecting this distinction. Here LPF reflects the real-world situation perhaps more faithfully. We now give some examples of predicates that are misleading if interpreted as statements about Standard ML programs.

The first kind of discrepancy arises in connection with functions of higher type. Consider the following declarations:

```
fun apply f x = f x

fun id  0    = 0
  | id (S n) = S (apply id n)
```

In Standard ML, `id` will behave as an identity function. By contrast, in our scheme the theorem `|- id 1 = 1` is not provable (although `|- id 0 = 0` is). The problem is that we cannot, in general, keep track of how a higher-order function, such as `apply` will choose to apply its argument. To be safe, we take (in Section 3(ii)) the pessimistic view, and universally quantify the missing arguments. The resulting `id` is thus "under-defined" relative to ML. This example is artificial, but similar remarks apply to the following natural code for summing the nodes of a labelled tree:

```
fun sumNodes (Tree(n, ts)) = n + sum (map sumNodes ts);
```

We can strengthen such examples. Consider

```
fun hal  0       = 0
  | hal  1       = 1 + (hal 1)
  | hal (S(S n)) = S (apply hal n)
```

(`hal` is almost `half`). In ML, `hal` n will terminate for all even n. However, because, in our axiomatisation of `DOM` predicates (‡ in section 3), we chose to use equality rather than implication, we can prove that

```
|- DOM'hal(x) = (x = 0)
```

On the other hand, a function application may (provably) have some concrete value in our scheme when the corresponding ML program diverges. This is perhaps the more serious problem. Consider for instance

```
    fun f x = f x;
    fun g x = x * f x
```

The resulting function `g` is "over-defined" relative to ML – we can prove the theorem `|- g 0 = 0` even though `g 0` diverges in ML.

These examples illustrate the differences between regarding `f x` as having an unknown value and as having no value. In particular, the role of the `DOM` predicates in our system is purely auxiliary – we introduce them to provide a formal means for reasoning about the values of function-calls. It is tempting but misleading to think of them expressing "definedness" – `DOM'g x` really only means "the function `g` obeys its defining equation for the value `x`". In our final example, `|- g x = x * f x` independently of whether we can prove anything about `f x`.

We have described the details of our scheme as it has been implemented in the LAMBDA theorem-prover. However, as we discussed in Section 3, variants of the scheme are certainly possible – in particular, there is scope for variation in our definition of the syntactic operator $\ll \_ \gg$. It seems that there amay be other definitions that avoid some of the above problems, in particular the "over-definedness" phenomenon mentioned above. For such alternative definitions, we expect to obtain metatheorems guaranteeing certain relationships between our logic and the behaviour of programs in Standard ML. We have not yet worked out all the details, however; and it is not yet clear to us what the trade-offs are between competing definitions of $\ll \_ \gg$.

References

[1] H. Barringer, J.H. Cheng and C.B. Jones, because we chose, to use equality rather than implication in our axiomatisation of `DOM` predicates (‡ in section 3), A logic covering undefinedness in program proofs. *Acta Informatica* 21, 251–269, 1984.

[2] R.S. Boyer and J.S. Moore, *A Computational Logic.* ACM Monograph Series, Academic Press, 1979.

[3] J.H. Cheng and C.B. Jones, On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock (eds.), *Proceedings of the Third Refinement Workshop*, Springer-Verlag, 1990.

[4] S. Finn and M.P. Fourman, *Logic Manual for the Lambda System.* Abstract Hardware Limited, version 3.2, 1990.

[5] S. Finn and M.P. Fourman, *Logic Manual for the Lambda System.* Abstract Hardware Limited, version 4.2.1, 1992.

[6] W.M. Farmer, A Partial Functions Version of Church's Simple Theory of Types. *Journal of Symbolic Logic* 55, 3, 1269–1291, 1990.

[7] M.J.C. Gordon, HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam (eds.), *VLSI Specification, Verification and Synthesis,* 73–128, Kluwer Academic Publishers, 1987.

[8] M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science,* Springer-Verlag, 1979.

[9] C.B. Jones, *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[10] C.B. Jones et al., eds., *Mural: A Formal Development Support System.* Springer-Verlag, 1991.

[11] Z. Luo and R. Pollack, LEGO proof development system: User's manual. LFCS Report Series ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[12] S. Owre, J.Rushby, *et. al.*, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering* 21, 2, 107–125, 1995.

[13] J. Rushby, F. von Henke and S. Owre, An Introduction to Formal Specification and Verification using EHDM. Technical Report SRI-CSL-91-02, SRI International, 1991.

[14] D.S. Scott, Identity and existence in intuitionistic logic. In *Lecture Notes in Mathematics* Volume 735, 661–696, Springer-Verlag, 1979.

[15] J.M. Spivey, *Understanding Z—A Specification Language and its Formal Semantics.* Cambridge Tracts in Computer Science 3, CUP, 1988.