When is a functional program not a functional program?

John Longley

Abstract

In an impure functional language, there are programs whose behaviour is completely functional (in that they behave extensionally on inputs), but the functions they compute cannot be written in the purely functional fragment of the language. That is, the class of programs with functional behaviour is more expressive than the usual class of pure functional programs. In this paper we introduce this extended class of "functional" programs by means of examples in Standard ML, and explore what they might have to offer to programmers and language implementors.

After reviewing some theoretical background, we present some examples of functions of the above kind, and discuss how they may be implemented. We then consider two possible programming applications for these functions: the implementation of a search algorithm, and an algorithm for exact real-number integration. We discuss the advantages and limitations of this style of programming relative to other approaches. We also consider the increased scope for compiler optimizations that these functions would offer.

0 Introduction

This paper is concerned with a rather surprising fact. In an impure functional language such as Standard ML [6], there exist programs whose behaviour is completely *functional*—in the sense that they can be modelled simply by a mathematical function from input-values to output-values—but the functions they compute cannot be written in the purely functional fragment of the language.

Let us explain more carefully what we mean by "programs whose behaviour is functional". For simplicity we will consider only simply typed ML terms over the basic types `unit`, `bool` and `int`. We will model these basic types semantically by the evident sets of values (writing $*$ for the value `()`):

$$\begin{array}{ll} [\![\texttt{unit}]\!] & = \{*\} \\ [\![\texttt{bool}]\!] & = \{\texttt{true}, \texttt{false}\} \\ [\![\texttt{int}]\!] & = \{\ldots\,\tilde{}2, \tilde{}1, 0, 1, 2, \ldots\} \end{array}$$

Given $[\![\sigma]\!]$ and $[\![\tau]\!]$, we will model the function type σ`->`τ by a certain set $[\![\sigma\texttt{->}\tau]\!]$ of partial functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$ (see below). We define inductively what it means for an ML term $M:\sigma$ to *denote* an element of $[\![\sigma]\!]$. For basic types γ, we say that M denotes $v \in [\![\gamma]\!]$ if M evaluates to v, say in the initial dynamic environment. For function types σ`->`τ we say that M denotes a partial function $f : [\![\sigma]\!] \rightharpoonup [\![\tau]\!]$ if whenever $N:\sigma$ denotes $x \in [\![\sigma]\!]$,

- if $f(x) = y$ then MN denotes y, and
- if $f(x)$ is undefined then MN diverges.

In fact, we define the denotation relations and the sets $[\![\sigma]\!]$ by simultaneous recursion on types, taking $[\![\sigma\texttt{->}\tau]\!]$ to be the set of all partial functions $[\![\sigma]\!] \rightharpoonup [\![\tau]\!]$ that are denoted by some term M.

We will now say a term $M:\sigma$ is *functional* (in behaviour) if it denotes some element $f \in [\![\sigma]\!]$. Intuitively, such a term behaves extensionally on its inputs—provided, of course, that these inputs are themselves functional in the same sense—so its behaviour on such inputs is captured completely by the mathematical function f. We will reserve the term *pure* for terms written in the usual pure functional fragment of ML.

We now give a simple example to show that the class of functional programs defined above is more expressive than the class of pure functional programs. First note that the set $[\![\texttt{unit->unit}]\!]$ has just two elements $\top, \bot$:

$$\begin{array}{ll} \top(*) & = * \\ \bot(*) & \text{is undefined} \end{array}$$

Next, the set $[\![\texttt{(unit->unit)->unit}]\!]$ contains just three elements *top*, *mid*, *bot*;[1] they may be defined in ML respectively by

```
fun top x = ()
fun mid x = x ()
fun bot x = bot x
```

Notice that $\mathit{bot} \leq \mathit{mid} \leq \mathit{top}$ with respect to the pointwise ordering.

Now let $F : [\![\texttt{(unit->unit)->unit)}]\!] \rightharpoonup [\![\texttt{bool}]\!]$ be the partial function specified by

$$\begin{array}{ll} F(\mathit{top}) & = \texttt{false} \\ F(\mathit{mid}) & = \texttt{true} \\ F(\mathit{bot}) & \text{is undefined.} \end{array}$$

Intuitively, given $g \in [\![\texttt{(unit->unit)->unit}]\!]$, the result of Fg (if defined) tells us whether g needs to "look at" its argument in order to return a result. Clearly `F` is a mathematically well-defined partial function, but it cannot be written

[1] There is also a fourth partial function $h : [\![\texttt{unit->unit}]\!] \rightharpoonup [\![\texttt{unit}]\!]$ such that $h(\top)$ is undefined but $h(\bot) = *$; however, it follows from the undecidability of the halting problem that this function is not denotable even in full ML.

in the pure functional fragment of ML, since it is not monotonic with respect to the pointwise ordering. However, F *can* be implemented in impure ML, for example using references:

```
fun F g =
   let val r = ref false in
      (g (fn ()=>(r:=true;())) ; !r)
   end
```

Other implementations of F using exceptions or continuations are also possible—see Section 1.3 below.

The idea is that programmers could make use of functions such as F and still believe they were doing functional programming. The semantics of local references in ML ensures that even nested calls to F behave in the functionally correct way.

Here is an example to show how, in principle, a function like F might be used to advantage. Suppose we want a program which takes a function g from integers to integers and computes the sum $\sum_{i=1}^{1,000,000} g(i)$. Suppose moreover that, for some reason, the arguments of g are to be given *lazily*—that is, g is to be represented by an ML function g:(unit->int)->int. Let

```
Sum1: ((unit->int)->int) -> int
```

be a pure ML function that computes the sum in the obvious way. Now observe that if g happens never to evaluate its argument, it must be a *constant* function. So by using F to detect whether this is the case, we get a spectacular gain in efficiency when g is constant:

```
fun test a = fn ()=>(a();0)
fun Sum2 g =
   if F (fn a=>(g(test a);())) then Sum1 g
   else g(fn ()=>1) * 1000000
```

This example is only half-serious, but it suggests that it is worth exploring what can be done using "functional" programs in this extended sense.

There is a natural mathematical class of computable functions—the *sequentially realizable* (SR) functions—which intuitively contains the above function F and "all things like it". This class of functions was studied extensively from a theoretical point of view in [4]. The purpose of the present paper is to consider these functions from a more practical standpoint, and to explore what they might have to offer to programmers and language implementors.

The paper is structured as follows. In Section 1 we briefly mention some of the relevant theoretical background, introduce some important examples of SR functions, and consider how they can be implemented. In Section 2 we ask what interesting kinds of program can be written in this extended functional style. Here we discuss two possible application areas: the implementation of generic search algorithms, and an algorithm for exact real-number integration. In Section 3 we ask what we might stand to gain from knowing that these programs are "functional"—in particular, the scope for compiler optimization of such programs. Finally, in section 4 we draw some tentative conclusions, and mention some questions for further consideration.

Although in this paper we have used Standard ML for our examples, we believe that the main ideas would apply equally well to lazy functional languages such as Haskell [3] (at least if one allows non-standard extensions; see e.g. [8]). An ML source file containing the examples in this paper is available electronically via the author's home page [5].

Acknowledgements. I am grateful to Simon Peyton Jones for providing me with the impetus to write this paper. I have benefited from discussions with David Aspinall, Mike Fourman, Andrew Kennedy and Eike Ritter. The comments from the anonymous referees were also very helpful. This research was funded by EPSRC Research Grant GR/L89532: "Notions of computability for general datatypes".

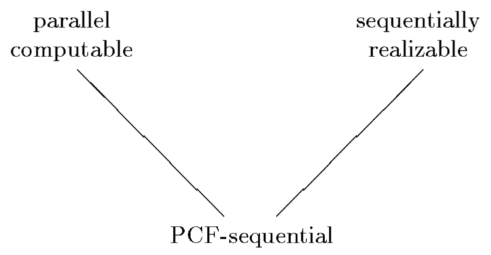
1 The sequentially realizable functionals

1.1 Conceptual background

We begin with a brief sketch of the theoretical picture that forms the background to the present paper.

Our work on the SR functions forms part of a general programme of research addressing the question "What are the natural and interesting notions of *computable function* at higher types?" For first-order types such as int->int, we all know what we mean by the "computable" functions: they are just the Turing-computable or *partial recursive* functions. At higher types, however, it is less immediately clear what "computable" should mean.

In fact, it now appears that there are at least three candidates for a reasonable notion of higher-type computability, each with good mathematical credentials, and each yielding a different class of "computable functions". Firstly, there are the sequentially computable functions embodied by the prototypical functional language PCF—this corresponds essentially to the pure functional (and simply-typed) fragment of ML or Haskell. Secondly, there are the parallel-computable functions, embodied by an extension of PCF incorporating *parallel-or* and *exists* functions. These parallel operations are in some sense computable, but they are not implementable even in full ML. Both of the above notions have been widely known since the early work of Scott [10] and Plotkin [9]. The third class of computable functions—the SR functions[2]—has emerged more recently from work of Bucciarelli and Ehrhard [1, 2], van Oosten [7] and the present author [4]. The SR functions form a larger class than the PCF-computable functions, but they are nonetheless all expressible in an impure sequential language such as ML. They admit a wide variety of mathematical characterizations, and have many pleasing theoretical properties (see [4]). Since the PCF-computable functions are (in a suitable sense) contained in each of the other two classes, we have the following picture:



The class of SR functions is in some sense the *maximal* class of higher-type functions expressible in a sequential programming language: indeed, the class of ML-denotable functions defined as in the Introduction turns out to coincide

[2]Since in this paper we are only interested in computable functions, we will use the term "SR functions" for what are called the *effective* SR functionals in [4].

exactly with the SR functionals. Thus, the SR functionals represent the limit of how far one can travel with the power of exceptions or references but without sacrificing the extensionality (i.e. functional behaviour) of programs.

One aspect of the SR functions which may seem puzzling is that they are not all *monotonic* in the usual sense. A partial explanation for this is that although not all the SR functions are monotonic with respect to the usual pointwise order on functions, they *are* monotonic with respect to the *stable* order (see e.g. [1]), which turns out to be the relevant ordering to consider in connection with the SR functionals.

1.2 Examples of SR functionals

The function `F` given earlier is one example of an SR function that is not PCF-computable. We now mention a few further examples, in order to illustrate the kind of power that the SR functions provide. (Implementations of our functions will be discussed in the next subsection.)

The main example we wish to introduce is the *modulus* function `Mod`. Although relatively simple, this seems to be sufficiently powerful for most of the plausible programming applications of SR functions. First, suppose we are given an ML function `G:(int->int)->int` and a function `h:int->int`, and let us assume that the application `G h` terminates. Informally, in the course of the computing `G h`, `G` can learn information about `h` by calling it with various arguments. For example, `G` might ask "what is `h 5`?", and `h` might reply "3"; then `G` could ask for `h 1`, and `h` could reply with 2; then `G` might decide it knows enough about `h`, and return the final result. By the time the computation finishes, what `G` has learnt about `h` can be represented by a finite set of ordered pairs—in this case, the set $\{(1,2),(5,3)\}$. This set is called the *modulus* of `G` at `h`; it is the graph of the unique smallest subfunction `h'` of `h` such that `G h'` terminates.

The function `Mod` takes as arguments any functions `G` and `h` as above, and—if `G h` terminates—returns the modulus of `G` at `h`, e.g. as a list of pairs. The type of `Mod` in ML might therefore be

```
((int->int)->int) -> (int->int) ->
(int*int)list
```

It is crucial here that `Mod` must return only the *set* of ordered pairs, and must not give away any information about the *order* in which the calls to `h` were made. Otherwise, the behaviour of `Mod` would not be functional, since it would allow us to distinguish between different implementations of the same function `G`, such as

```
fun G1 h = h 5 + h 1
fun G2 h = h 1 + h 5
```

To prevent this, we can sanitize the result of `Mod` by *sorting* the list of pairs and removing duplicates, or else by using an abstract type of sets in place of lists.

The point of `Mod` is that it provides information about how much of `h` is actually looked at by `G`. In this respect it has a similar feel to `F`, and indeed it is easy to see how `F` can be defined from `Mod` in pure functional ML. What is more surprising is that `Mod` is definable from `F`,[3] although this is not the most efficient way to implement `Mod`.

[3]This result is due to Alex Simpson; see [4, Proposition 9.13] for the construction.

Many minor variations on `Mod` are possible. For example, the types used in the above example are clearly not the most general possible. Also, in practice one is likely to want a function `Mod'` which, given `G` and `h`, returns both the result of `G h` and the modulus of `G` at `h`. Alternatively, one might not require the whole modulus, but only wish to know, say, the *largest* argument (if any) with which `h` was called; this could be computed by a simpler function `ModMax`. Finally, one could consider analogues of the modulus function at higher types (computing, for example, the modulus of a third-order function at a second-order one).

Although `Mod` is powerful enough for many purposes, there are many SR functions that are not definable from `Mod` in pure functional ML. The bizarre kinds of finite SR function discussed in [4, Section 9.4] provide one class of examples, but it would be surprising if one could find a practical use for these. A more promising example is a function `E` of type

```
(int stream->int) -> (int->int) -> int
```

(where `'a stream` is the familiar recursive type for streams). Here we give just an informal description of `E`; for further details see [5].

Intuitively, any sequential algorithm for a function of type `int stream->int` can be represented in an obvious way by a (possibly infinite) *decision tree*; and the type `int->int` can be used to encode a strategy for exploring such a decision tree and returning information about some finite part of it. The crucial observation which makes `E` functional is that for any function of type `int stream->int` there is a *canonical* (smallest) decision tree that computes it, so information about this tree is really information about the function itself.

Finally, we mention a function even more powerful than `E`: the *universal* SR function `H` defined in [4, Section 7]. One of the main results of [4] is that every SR function is definable from `H` in pure PCF. This means that if one were to extend the pure functional fragment of ML by adding `H` (implemented using non-functional features), one would have a language in which one could express all the SR functions and only them. Informally, the idea behind `H` is somewhat similar to `E`, but in place of `int stream->int` we have the more complex type `(int->int)->int`. The difficulty is that for functions of this type, there is no longer a *canonical* decision tree—this means that once `H` has obtained a result from one decision tree, in order to remain functional it has to check that all other decision trees for the same function would have given the same result. This entails a factorial-size search, which makes `H` less than appealing from a practical point of view!

1.3 Implementations of SR functions

All of the functions we have mentioned can be implemented in Standard ML, in several different ways. We have already seen how to implement `F` using references, but it can also be written using exceptions:

```
fun F g =
   let exception e in
      (g (fn _=>raise e) ; false)
      handle e => (g (fn ()=>()) ; true)
   end
```

Alternatively, in ML of New Jersey, `F` can be implemented using `callcc`:

```
fun F g =
   callcc (fn con =>
      (g (fn ()=>
         throw con (g (fn ()=>()) ; true)) ;
      false))
```

However, both these implementations suffer from an inefficiency which makes the reference implementation preferable in practice. In both of the above, once we have discovered that g evaluates its argument, we still need to check that `g(fn ()=>())` terminates, so the first part of the computation performed by g is done twice. This illustrates what seems to be a common phenomenon for SR functions: one sometimes has to do stupid extra work in order to behave functionally.

The exceptions implementation of `F` also suffers from another drawback:[4] it does not behave functionally on arguments involving wildcard exception handlers, for example:

```
fun id' x = x() handle _ => ()
```

Thus, this implementation is functional only with respect to the fragment of ML without wildcard handlers.

The function `Mod` is also most easily implemented using references. To compute `Mod G h`, we maintain a log of all calls made to `h`, and apply `G` to a version of `h` that updates the log whenever it is called.

```
fun Mod G h =
   let val log = ref []
       fun h' x =
          let val y = h x in
             (log := insert(x,y)(!log) ; y)
          end
   in (G h' ; !log)
   end
```

Here `insert` is a suitable function for inserting a pair into a sorted list. One could of course achieve much greater efficiency by using, say, balanced trees to maintain sorted lists, and this might be acceptable for many purposes, but even so, one might resent having to do the sorting just for the sake of being a functional program. However, it is plausible that for many applications we would like the result to be sorted anyway. Note also that to implement the simpler function `ModMax` no sorting is required.

The function `E` requires a little more effort to implement (here it seems best to use a combination of references and exceptions), but it is reasonably efficient and there is not too much of an overhead arising from the need to be functional. By contrast, the function `H` is an extreme example of a function for which extra work is required in order to stay functional. (We do not know whether this inefficiency would necessarily be fatal in practical applications, however, because we do not have any practical applications for `H`.)

2 Some programming applications

2.1 Search algorithms

We now suggest some programming applications for SR functions. Our first application is the implementation of a "generic" search algorithm. Suppose (for example) that we wish to search all permutations of the integers $0, \ldots, n-1$ for those satisfying a property P, where P is to be supplied as a parameter. In principle this is a search of size $n!$, but we can often cut down the search as follows. Suppose we test a permutation $\sigma = (\sigma_0, \ldots, \sigma_{n-1})$. If we can already see that $P(\sigma)$ does not hold just by looking at $\sigma_0, \ldots, \sigma_{r-1}$ for some $r < n$, we can save ourselves a subsearch of size $(n-r)!$. If permutations are represented by functions of type `int->int`, and P by a function of type `(int->int)->bool`, we can use a function like `ModMax` to discover the value of r here. When cycling through all the permutations in some order, this number can be used to tell us the next permutation that we need to consider. In this way we can construct a function

```
search : int -> ((int->int)->bool) ->
         (int->int)list
```

such that `search n P` returns a list of the permutations of `0,...,n-1` satisfying `P`. (See [5] for the easy implementation details.)

By supplying different functions `P` to `search`, we can (for example) construct magic squares or solve the n queens problem with reasonable efficiency. Of course, our solution is no more efficient than the usual solutions to either of these problems, but there is a gain in *modularity*: we have separated out some general-purpose machinery for searching through permutations from the problem-specific details of the property we are interested in. Obviously this could be achieved with ordinary pure functional programming if we required `P` to return explicit modulus information together with the boolean result, but by not requiring this we have relieved the implementors of functions `P` of this (small) burden. Our solution therefore seems appealing if we wish to do many different searches on the same search space.

There is an important point here that we would like to emphasize. In general, programmers try to keep the interfaces between program modules as simple as possible, in order to keep the overall complexity of a system within the limits of understanding. Very often there is a trade-off between simplicity of interfaces and efficiency, since a simple interface may prevent different modules from exchanging useful information. As the above example shows, SR functions can sometimes be used to pass intensional information across module boundaries—to make black boxes less opaque—but without complicating the interfaces.

We should mention a limitation of the above approach. Our search algorithm will work most efficiently on arguments `P` which, when given a permutation `sigma`, call `sigma` with smaller arguments first, and then call `sigma` with larger arguments only if this is still necessary. However, there may be properties `P` for which some other evaluation order is more convenient, and so we would prefer an implementation of `search` in which the order in which we cycled through the permutations was not fixed in advance, but was driven dynamically by the evaluation behaviour of `P`. Such an implementation is indeed possible if we make use of the *list* of all the arguments with which `P` calls `sigma`, rather than just the maximum. Unfortunately, though, the modulus function does not suffice, because here we need to know the *order* in which these calls are made. Thus, this enhanced search algorithm, although an attractive example of general higher-order programming, goes beyond what can be done with SR functionals.[5]

[4] This was pointed out to me by Nick Benton and Andrew Kennedy.

[5] An intermediate algorithm making use of the unordered *set* of arguments called might also be possible, but it seems that the control

This illustrates an interesting general point about the SR functionals: in order to be functional, one sometimes deliberately throws away information that might actually be of interest to the programmer.

2.2 Exact integration

Our second application is in the area of exact real number computation, and concerns an algorithm for integrating a given real function to any desired precision. Let us represent real numbers in the interval $I = [-1, 1]$ (non-uniquely) by streams of extended binary digits, and real functions by functions on such streams:

```
datatype Digit = One | Zero | MinusOne
datatype Real  = Real of Digit * (unit -> Real)
type RealFun   = Real -> Real
```

We wish to write a program which, when given an integer k and a total function $f : I \to I$ represented by a value `F:RealFun`, computes $\int_{-1}^{1} f$ to within 2^{-k}. (Once we have done this, we could in principle compute the integral as a value of type `Real`.) To do this, it is enough to know the value of $f(x)$ to within $\epsilon = 2^{-(k+1)}$ for every $x \in I$.

Our algorithm proceeds as follows. First we compute $f(-1)$ to within ϵ, by applying `F` to the stream $\mathtt{MinusOne}^{\infty}$ (using an obvious notation); suppose the result is the dyadic rational y_0. Using a variant of the `ModMax` function, we can detect how many digits of the input stream were used by `F` to obtain y_0. Suppose j input digits were required, and let $\delta_0 = 2^{-(j-1)}$; then we know that for any $x \in [-1, -1 + \delta_0]$ we have $|f(x) - y_0| \leq \epsilon$. We therefore have that $\int_{-1}^{-1+\delta_0} f$ is $\delta_0 y_0$ to within $2^{-(j+k)}$. Next we compute $f(-1 + \delta_0)$, where the argument is now represented by the stream $\mathtt{MinusOne}^{j-1}; \mathtt{One}; \mathtt{MinusOne}^{\infty}$. As before we obtain some y_1 and δ_1 such that $|f(x) - y_1| \leq \epsilon$ for all $x \in [-1 + \delta_0, -1 + \delta_0 + \delta_1]$. We continue creeping along the x-axis in this way until we reach 1 (see below). When we finish, we will have approximated f everywhere to within ϵ by pieces of constant functions. By adding up the areas of the corresponding rectangles as we go, we obtain an approximation to the desired integral. Since the δ_i add up to 2, the total error in this approximation is at most $2\epsilon = 2^{-k}$.

It remains to show that the algorithm terminates, i.e., that we do eventually reach $x = 1$. Suppose for contradiction that the sequence $x_1, x_2, \ldots$ of x-values generated by the algorithm converges to some $x_\infty < 1$. Then clearly the streams $S_1, S_2, \ldots$ representing these x-values themselves converge (in an obvious sense) to some stream S_∞ representing x_∞. But since f is assumed to be a total function, the application $F(S_\infty)$ must yield some real number $f(x_\infty)$. In particular, the computation of $f(x_\infty)$ to within ϵ must use some finite number j of digits of S_∞. Now for some sufficiently large n, the stream S_n will agree with S_∞ on the first j digits; and in this case, upon reaching S_n our algorithm would detect the modulus j, and so at the next iteration would generate a stream $S_{n+1} \geq S_\infty$. Thus $S_{n+2} > S_\infty$, a contradiction.[6]

structure would become much more complicated, unless we are willing to accept the risk of testing some permutations more than once.

[6]Interestingly, the stream S_∞ need not be *computable*, since we may not have any way of approximating x_∞ from above. Thus, we need to assume that `F` in some sense represents a total function on the classical reals and not just on the computable ones, and we ought to clarify what we mean by $F(S_\infty)$ in this case. However, we will gloss over these points here, since they make a difference only for highly pathological functions.

It is worth comparing our approach with previous approaches to exact integration. In [11], Simpson gives a lazy functional algorithm for integration using Berger's *uniform modulus* functional. Simpson's algorithm is particularly remarkable in that, unlike ours, it can be coded in just PCF (*alias* pure functional ML or Haskell). However, our algorithm has a number of advantages. Firstly, it is more efficient in practice, since in Simpson's algorithm the modulus information is obtained in a rather inefficient way. (In principle, though, the two algorithms differ in runtime only by a constant factor.) Secondly, our algorithm seems easier to understand, since it does not rely on any clever use of higher-type recursion and the evaluation behaviour is easy to visualize. Thirdly, our approach using SR functions can be extended to compute integrals for certain classes of *discontinuous* functions, which cannot be integrated at all in pure PCF. For example, if $f : (I - \{1\}) \to I$ has no continuous extension to the whole of I, we can still compute the integral to within $2^{-(k-1)}$ using the above method: once we reach $x = 1 - \epsilon$ we can stop, since the unexplored part of f from $1 - \epsilon$ to 1 will contribute at most ϵ to the integral. One can also define, for example, an SR function that integrates any function $f : I \to I$ that is undefined on at most two points (though here the modulus function does not suffice—we need the function `E`).

Other approaches are possible if we allow ourselves to use a different type to represent real functions. For example, as with the search algorithm above, we could insist that our implementations of real functions return explicit modulus information, but this would seem to complicate the coding of functions significantly. More interestingly, one can represent real functions using the following datatype of "processes" operating on streams:

```
datatype RealFun' =
   Question of Digit -> RealFun'
 | Answer of Digit * (unit -> RealFun')
```

We should admit that, using this representation, one can give a purely functional implementation of (essentially) the above algorithm that is considerably more efficient than the one using SR functions. The price to pay is that we are forced to program our functions in a rather low-level style, making the supply and demand of digits very explicit. Meanwhile, our SR implementation is the best available if we wish to integrate functions of ML type `Real->Real`.

3 Advantages over impure functional programming

So far we have explored the possible benefits of programming with SR functionals as opposed to conventional pure functional programming. Given that SR functionals can be implemented in an impure language anyway, we should also ask the dual question: what are the advantages of SR-based programming over unrestricted impure functional programming? In other words, what can we gain from knowing that programs such as `Mod` behave functionally? One possible answer is that functional programs are often easier to reason about (both formally and informally) than non-functional ones, and so the SR functions seem attractive from the point of view of program verification. This possibility was briefly considered in [4, Section 12.4], where it was argued that the class of SR functions was easier to reason about even than the class of PCF-computable functions. (This is because the SR functions enjoy better decidability properties than the PCF ones.) Here, however, we concentrate on another possible answer: they offer increased scope for *compiler optimization.*

It is well known, and commonly exploited by compiler writers, that more optimizing transformations are legitimate for functional code than for code that may have side-effects. Since SR functions are functional in terms of their behaviour, one would expect that, in principle, a compiler should be able to take advantage of this fact.

For example, consider the ML function

```
fun G f =
   if f 0 = 0 then f 1 else f 1 + f 1
```

As part of the optimization phase, a compiler might consider eliminating the common subexpression `f 1`, and replacing the body of the above declaration by

```
let val x = f 1 in
   if f 0 = 0 then x else x + x
end
```

This avoids duplicating the evaluation of `f 1` in the case that `f 0` is not `0`. However, this transformation changes the evaluation order: `f 1` is now evaluated before `f 0`. This optimization is therefore valid provided that the argument `f` is functional. Whether this is so will depend, of course, on the context in which `G` is used; indeed, a compiler might generate two different versions of the object code for `G`, one for use in functional contexts and one for other contexts.

Now suppose `G` is used in some context `Mod G f`, where `f` is known to be functional. The functional character of `Mod` means that it is in principle quite legitimate to use the optimized version of `G` here. This contrasts with what would happen for the non-functional version of `Mod` which retains the order of the calls to `f`: the two versions of `G` would yield different results.

In many modern compilers, the code optimizations are regulated by means of a type system carrying information about which pieces of code are functional (in various respects). It seems that optimizations of the above kind could be supported by such a compiler fairly easily, by implementing a standard library of SR functions and ascribing types to them that reflected their functional status (these would typically be better than the types that the compiler could infer for itself from the source code). This would give us a syntactically identifiable class of SR programs. (We cannot expect a compiler to recognize *all* SR programs as such, since it is undecidable in general whether a program is in the SR fragment.)

It appears that one could achieve a similar effect in a lazy language such as Haskell (in which code transformations are often crucial for efficiency). Functions like `Mod` could be implemented using a suitable side-effect monad; and some implementations of Haskell provide an operation `unsafePerformIO` for transforming this into a value that the compiler will treat as if purely functional (see [8]). Thus, the compiler disclaims all responsibility for differences in program behaviour resulting from changes in evaluation order; but with `Mod`, this will never be a problem.

4 Conclusion

In this paper we have tried to give a practically oriented introduction to the SR functions by means of examples, and to convey an impression of what can be done with them. In summary, SR functions can be used to obtain intensional information about a function, such as how much use it makes of this argument, but they make this information available in a completely extensional way.

We have illustrated some of the potential advantages of programming with SR functions, both in relation to pure functional programming in the usual sense, and in relation to unrestricted impure functional programming. On the one hand, we gain some interesting extra programming power over ordinary functional programs, but on the other hand, we retain the advantages of ease of reasoning and scope for compiler optimization.

We have also pointed out some of the limitations of SR programming: sometimes implementations of SR functions need to perform extra work in order to ensure that they behave functionally, and sometimes this extra work even deprives the programmer of information he might be interested in. On occasions, these disadvantages will clearly outweigh any benefit that might be gained from being "functional".

The purpose of this paper has been to advertise an elegant theoretical idea which might have useful applications in functional programming, both to programming and to language implementation. The particular examples we have discussed are rather tentative, and are given mainly for the purpose of illustration. One possible application area we have not investigated so far is the compile-time analysis of programs: here one is often interested in knowing which pieces of code are and are not "looked at" under various conditions, and it seems plausible that SR functions might be of some use here. In any case, we hope that other workers in functional programming, whose primary interests and experience are more practical than those of the present author, may be able to find other good uses for SR functions.

References

[1] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *Proc. 6th Annual Symposium on Logic in Computer Science*, pages 138–145. IEEE, 1991.

[2] T. Ehrhard. Projecting sequential algorithms on strongly stable functions. *Annals of Pure and Applied Logic*, 77:201–244, 1996.

[3] S. Peyton Jones and J. Hughes (eds.). Report on the programming language Haskell 98. Available electronically from `www.haskell.org`, February 1999.

[4] J.R. Longley. The sequentially realizable functionals. Technical Report ECS-LFCS-98-402, Department of Computer Science, University of Edinburgh, 1998. Submitted to *Annals of Pure and Applied Logic.*

[5] J.R. Longley. When is a functional program not a functional program?: a walkthrough introduction to the sequentially realizable functionals. ML source file, available from `http://www.dcs.ed.ac.uk/hom/jrl/`, 1998.

[6] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: revised 1997.* MIT Press, 1997.

[7] J. van Oosten. A combinatory algebra for sequential functionals of finite type. Technical Report 996, University of Utrecht, 1997. To appear in Proc. Logic Colloquium, Leeds.

[8] S. Peyton Jones and S. Marlow. Stretching the storage manager: weak pointers and stable names in Haskell. Draft paper, 1999.

[9] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[10] D.S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Comp. Sci.* , 121:411–440, 1993. First written in 1969 and widely circulated in unpublished form since then.

[11] A.K. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science*, pages 456–464. Springer LNCS 1450, 1998.