to appear in Springer Lecture Notes in Computer Science 1022:1-13, 1995.

Elementary Strong Functional Programming

D.A.Turner

University of Kent
Canterbury CT2 7NF, England

Abstract

Functional programming is a good idea, but we haven't got it quite right yet. What we have been doing up to now is weak (or partial) functional programming. What we should be doing is strong (or total) functional programming - in which all computations terminate. We propose an elementary discipline of strong functional programming. A key feature of the discipline is that we introduce a type distinction between *data*, which is known to be finite, and *codata*, which is (potentially) infinite.

1 What is Functional Programming?

It is widely agreed that functional programming languages make excellent introductory teaching vehicles for the basic concepts of computing. The wide range of topics covered in this symposium is evidence for that. But what is functional programming?

Well, it is programming with functions, that much seems clear. But this really is not specific enough. The methods of denotational semantics show us how almost any programming language construction, no matter how opaque and side-effect ridden, can be construed functionally if we are willing to introduce complicated enough spaces for the functions to work on.

It is somewhat difficult to pin down with complete precision, but what we conventionally mean by functional programming involves the idea that the functions are transparent in the notation we actually write, rather than having to be teased out by some complex process of interpretation. For example if I write, in Miranda[†] or Haskell (actually neither language has nat as a distinct type, but that's an oversight)

```
>       fac :: nat->nat
>       fac 0 = 1
>       fac (n+1) = (n+1) * fac n
```

then the semantics I intend has that `fac` really is a function from natural numbers to natural numbers, not something else, such as a function from $nat \times store$

to $nat \times store$, as I would have to say in a language with side effects, or a transformation over *nat*-demanding continuations, which is what I would have to say in a language with jumps as well as side effects.

Further, the equations which I have written as the definition of `fac` are actually true, and are everything I need to know about it. From them I can infer not only, e.g.

```
        fac 3 = 6
```

but also more general properties of `fac`, by using induction principles and algebraic reasoning.

In a functional language things are what we say they are, not something much more complicated in disguise. This is particularly apparent in the notational style represented by such languages as Miranda [9], Haskell [4], and the functional subset of Standard ML [3]. We have

(i) strong typing: the domain and codomain of each function is either stated in or inferable from the program text, and there is a syntactic discipline which prevents a function from being applied to an inappropriate argument.

(ii) functions are defined by equations - typically involving case analysis by pattern matching - and we can do equational reasoning in a familiar algebraic way.

(iii) expressions can be evaluated by treating the program equations as rewrite rules, so computation is a special case of equational reasoning - and the final result will be independent of the order in which the rewrite rules are applied.

(iv) there are simple induction rules associated with the various (non-function) data types - and new types are introduced in a way that enables a corresponding induction principle to be readily inferred.

We can sum this up by saying that functional programming is programming with functions in a style that supports equational reasoning and proof by induction.

Those of us who have become converted are convinced that this is an excellent way to teach programming.

THE BAD NEWS. Unfortunately, none of the properties I have ascribed to functional languages above is actually quite true of any of our present languages. There is a pathology, connected with the possibility of run-time errors and non-terminating computations, which runs right through everything, and messes up all the details.

For a discussion of the complexities that can arise in reasoning about Miranda programs see Thompson [7]. Similar complications arise for any of the functional languages in current use, the details depending on such matters as whether the language is strict or lazy.

The thesis of this paper is as follows. Functional programming is a very good idea, but we haven't got it quite right yet. What we have been doing up to now is weak functional programming. What we should be doing is strong functional programming.

The remaining sections of the paper are organised as follows. Section 2 introduces the idea of strong functional programming. In section 3 we outline an elementary language for strong functional programming over finite data. In section 4 we show how the concept of codata can be added, to bring back the possibility of programming with infinite streams etc. In section 5 we make some closing remarks.

2 Strong Functional Programming

Conventional functional programming may be called *weak.* What is the difference between weak and strong?

In a weak functional language if we have an expression, say

```
e :: int
```

we know that if evaluation of e terminates successfully, the result will be an integer - but evaluation of e might fail to terminate, or might result in an error condition.

In a strong functional language, if we have an expression

```
e :: int
```

we know that evaluation of e will terminate successfully with an integer result . In strong functional programming there are no non-terminating computations, and no run-time errors.

In the semantics of weak functional programming each type T contains an extra element $\bot_T$ to denote errors and non-terminations.

In strong functional programming $\bot$ does not exist. The data types are those of standard mathematics.

What are the advantages of strong functional programming? There are three principle ones:

1) The proof theory is much more straightforward.
2) The implementor has greater freedom of action.
3) Language design issues are greatly simplified (no strict versus non-strict).

2.1 Simpler Proof Theory

One of the things we say about functional programming is that it's easy to prove things, because there are no side effects. But in Miranda or Haskell - or indeed SML - the rules are not those of standard mathematics. For example if `e` is of type `nat`, we cannot assume

```
e - e = 0
```

because `e` might have for its value $\bot_{nat}$.

Similarly we cannot rely on usual principle of induction for nats

$$\frac{P(0) \qquad \forall n.P(n) \Rightarrow P(n+1)}{\forall n.P(n)}$$

without taking precautions to deal with the case $n = \bot$.

These problems arise, in different ways, in both strict and lazy languages. In strong functional programming these problems go away because there is no $\perp$ to worry about. We are back in the familiar world of sets.

2.2 Flexibility of Implementation

In strong functional programming reduction is *strongly Church-Rosser.* Note the distinction between
(A) Church-Rosser Property:
If E can be reduced in two different ways, then if they both produce normal forms, these will be the same
(B) Strong Church-Rosser Property:
Every reduction sequence leads to a normal form and normal forms are unique.
The ordinary Church-Rosser property gives a form of confluence, with strong Church-Rosser we have this plus strong normalisability - so we can evaluate in any order. This gives much greater freedom for implementor to choose an efficient strategy, perhaps to improve space behaviour, or to get more parallelism. The choice of eager or lazy evaluation becomes a matter for the implementor, and cannot affect the semantics.

2.3 Simpler Language Design

In weak functional programming languages we have many extra design decisions to make, because of strict versus non-strict. Consider for example the `&` operation on `bool`, defined by

```
True & True = True
True & False = False
False & True = False
False & False = False
```

but there are more cases to consider:

```
⊥ & y = ?
x & ⊥ = ?
```

Considering the possible values for these (which are constrained by monotonicity) gives us a total of four different possible kinds of `&` namely
(i) doubly strict `&`
(ii) left-strict `&`
(iii) right-strict `&`
(iv) doubly non-strict (parallel) `&`
Should we provide them all? Only one? How shall we decide?

In strong functional programming these semantic choices go away. Only one & operation exists, and it is defined by its actions on `True`, `False` alone.

2.4 Disadvantages

What are the disadvantages of strong functional programming? There are two obvious ones

1) Programming language is no longer Turing complete!

2) If all programs terminate, how do we write an operating system?

Can we live with 1? We will return to this in the closing section, so let us postpone discussion for now.

The answer to 2 is that we need *codata* as well as data. (But unlike in weak functional programming, the two will be kept separate. We will have finite data and infinite codata, but no partial data.)

There already exists a theory of strong functional programming which has been extensively studied. This is the constructive type theory of Per Martin-Löf (of which there are several different versions). This is a very complex theory which includes:

- Dependent types (types indexed over values)
- Second order types
- An isomorphism between types and propositions, that enables programs to encode proof information.

This is a powerful and interesting theory, but it not suitable as a vehicle for first year teaching - it seems unlikely to replace PASCAL as the introductory programming language.

We need something simpler.

3 Elementary strong functional programming

What I propose is something much more modest than constructive type theory, namely an *elementary* discipline of strong functional programming.

Elementary here means

1) Type structure no more complicated than Hindley/Milner, or one of its simple variants. So we will have types like $int \rightarrow int$, and polymorphic types like $\alpha \rightarrow \alpha$, but nothing worse.

2) Programs and proofs will be kept separate, as in conventional programming. What we are looking for is essentially a strongly terminating subset of Miranda or Haskell (or for that matter SML, since the difference between strict and lazy goes away in a strong functional language)

First, we must be able to define data types.

```
>       data day = Mon | Tue | Wed | Thur | Fri | Sat | Sun

>       data nat = Zero | Suc nat
>       data list a = Nil | Cons a (list a)

>       data tree = Nilt | Node nat tree tree

>       data array a = Array (nat->a)
```

As is usual some types - `nat`, `list` for example - will be built in, with special syntax, for convenience. So we can write e.g. `3` instead of `Suc(Suc(Suc Zero))`.

There are three essential restrictions.

RULE 1) All primitive operations must be total. This will involve a some non-standard decisions - for example we will have

```
        0 / 0 = 0
```

Runciman [6] gives a useful and interesting discussion of how to make natural arithmetic closed. He argues that the basic arithmetic type in a functional language should be `nat` and not `int` and I am persuaded by his arguments.

Making all basic operations total of course requires some attention at types other than `nat` - for example we have to decide what to do about `hd[]`. There are various possible solutions - making `hd` return an element of a disjoint union, or giving it an extra argument, which is the value to be returned on `[]`, are the two obvious possibilities. It will require a period of experiment to find the best style. Notice that because `hd` is polymorphic we cannot simply assign a conventional value to `hd[]`, for with the abolition of $\perp$ we no longer have any values of type α.

RULE 2) Type recursion must be *covariant.* That is type recursion through the left hand side of $\rightarrow$ is not permitted. For example

```
>       data silly = Silly (silly->nat) ||not allowed!
```

Contravariant types like `silly` allow $\perp$ to sneak back in, and are therefore banned.

Finally, it should be clear that we also need some restriction on recursive function definitions. Allowing unrestricted general recursion would bring back $\perp$.

First note that to define functions we introduce the usual style of equational definition, using pattern matching over data types. Eg

```
>       size :: tree a -> nat
>       size Nilt = 0
>       size (Node n x y) = n + size x + size y
```

To avoid non-termination, we must restrict ourselves to *well-founded recursion.* How should we do this? If we were to allow arbitrary well-founded recursion, we would have to submit a proof that each recursive call descends on some well-founded ordering, which the compiler would have to check. We might also have to supply a proof that the ordering in question really is well-founded, if it is not a standard one.

This contradicts our requirement for an *elementary language,* in which programs and proofs can be kept separate. We need a purely syntactic criterion, by which the compiler can enforce well-foundedness. I propose the following rule

RULE 3) Each recursive function call must be on a syntactic subcomponent of its formal parameter (the exact rule is slightly more elaborate, to take account of pattern matching over several arguments simultaneously - this is so as to allow “nested” structural recursion, as in Ackermann’s function - the extension adds no power, because what it does can be desugared using higher order functions, but is syntactically convenient).

The classic example of what this allows is recursion of the form

```
>       f :: nat->thing
>       f 0 = something
>       f (n+1) = ...f n...
```

except that we generalise the paradigm to multiple arguments and to syntactic descent on the constructors of any data type, not just `nat`.

The rule effectively restricts us to primitive recursion, which is guaranteed to terminate. But isn’t primitive recursion quite weak? For example is it not the case that Ackermann’s function fails to be primitive recursive? NO, that’s a first order result - it does not apply to a language with higher order functions.

IMPORTANT FACT: we are here working in a higher order language, so what we actually have are the primitive recursive functionals of finite type, as studied by Gödel [2] in his *System T.*

These are known to include every recursive function whose totality can be proved in first order logic (starting from the usual axioms for the elementary data types, eg the Peano axioms for nat). So Ackermann is there, and much, much else. Indeed, we have more than system T, because we can define data structures with functional components, giving us infinitarily branching trees. Depending on the exact rules for typechecking polymorphic functions, it is possible to enlarge the set of definable functions to all those which can be proved total in *second order* arithmetic.

So it seems the restriction to primitive recursion does not deprive us of any functions that we need, BUT we may have to code things in an unfamiliar way - and it is an open question whether it gives us all the *algorithms* we need (this is a different issue, as it relates to complexity and not just computability). I have been studying various examples, and find the discipline surprisingly convenient.

An example.

Quicksort is not primitive recursive. However Treesort is primitive recursive (we descend on the subtrees) and for each version of Quicksort there is a Treesort which performs exactly the same comparisons and has the same complexity, so we haven't lost anything.

Another example - fast exponentiation.

```
>       pow :: nat->nat->nat
>       pow x n = 1,                      if n == 0
>               = x * pow (x * x) (n/2), if odd n
>               = pow (x * x) (n/2),     otherwise
```

(An aside - note that the last guard of a guard set must be `otherwise`.) This definition is not primitive recursive - it descends from n to n/2. Primitive recursion on nats descends from (n+1) to n.

However, we can recode by introducing an intermediate data type `[bit]`, (i.e. list-of-bit), and assuming a built in function that gives us access to the binary representation of a number.

```
>      data bit = On | Off

>      bits :: nat->[bit] ||built in

>      pow x n = pow1 x (bits n)
>      pow1 x Nil = 1
>      pow1 x (On : y) = x * pow1 (x * x) y
>      pow1 x (Off : y) = pow1 (x * x) y
```

Summary of programming situation:

Expressive power - we can write any function which can be proved total in the first order theory of the (relevant) data types. (FACT, DUE TO GÖDEL)

Efficiency - I find that around 80% of the algorithms we ordinarily write are already primitive recursive. Many of the others can be reexpressed as primitive recursive, with same computational complexity, by introducing an intermediate data structure. (MY CONJECTURE: with more practice we will find this is always true.)

I believe it would not be at all difficult to learn to program in this discipline, but you do have to make some changes to your programming style. More research is needed (for example Euclid's algorithm for *gcd* is difficult to express in a natural way).

It is worth remarking that there is a sledge-hammer approach that can be used to rewrite as primitive recursive any algorithm for which we can compute an upper bound on its complexity. We add an additional parameter, which is a natural number initialised to the complexity bound, and count down on that argument while recursing. This wins no prizes for elegance, but it is an existence proof that makes more plausible my conjecture above.

3.1 PROOFS

Proving things about programs written in this discipline is very straightforward. Equational reasoning, starting from the program equations as axioms about the functions they define.

For each data type we have a principle of structural induction, which can be read off from the type definition, eg

```
>       data nat = Zero | Suc nat
```

this gives us, for any property P over `nat`

$$\frac{\begin{array}{l} P(\texttt{Zero}) \\ \forall n.P(n) \Rightarrow P(\texttt{Suc}\ n) \end{array}}{\forall n.P(n)}$$

We have no $\perp$ and no domain theory to worry about. We are in standard (set theoretic) mathematics.

4 CODATA

What we have sketched so far would make a nice teaching language but is not enough for production programming. Let us return to the issue of writing an operating system.

An operating system can be considered as a function from a stream of requests to a stream of responses. To program things like this functionally we need infinite lists - or something equivalent to infinite lists.

In making everything well-founded and terminating we have seemingly removed the possibility of defining infinite data structures. To get them back we introduce *codata type definitions:*

```
>       codata colist a = Conil | a <> colist a
```

Codata definitions are equations over types that produce final algebras, instead of the initial algebras we get for data definitions. So the type colist contains all the infinite lists as well as finite ones - to get the infinite ones alone we would omit the `Conil` alternative. Note that infix `<>` is the coconstructor for colists.

The rule for coprimitive corecursion on codata is the dual to that for primitive recursion on data. Instead of descending on the argument, we ascend on the result. Like this

```
>       f :: something->colist nat        ||example
>       f args = RHS (f args')
```

where the leading operator of RHS *must be a coconstructor.* There is no constraint on the form of `args'`.

Notice that corecursion *creates* (potentially infinite) codata, whereas ordinary recursion *analyses* (necessarily finite) data. Ordinary recursion is not legal over codata, because it might not terminate. Conversely corecursion is not legal if the result type is data, because data must be finite.

Now we can define infinite structures, such as

```
>       ones :: colist nat
>       ones = 1 <> ones

>       fibs :: colist nat
>       fibs = f 0 1
>              where
>              f a b = a <> f b (a+b)
```

and many other examples which every Miranda or Haskell programmer knows and loves.

NOTE THAT ALL OUR INFINITE STRUCTURES ARE TOTAL

As in the case of primitive recursion over data, the rule for coprimitive corecursion over codata requires us to rewrite some of our algorithms, to adhere to the discipline of strong functional programming. This is sometimes quite hard - for example rewriting the well known sieve of Eratosthenes program in this discipline involves coding in some bound on the distance from one prime to the next.

There is a (very nice) principle of coinduction, which we use to prove infinite structures equal. It can be read off from the definition of the codata type. We discuss this in the next subsection.

A question. Does the introduction of codata destroy the strong Church-Rosser property? No! (But you have to have the right definition of normal form. Every expression whose principle operator is a coconstructor is in normal form.)

4.1 Coinduction

First we give the definition of bisimilarity (on colists). We can characterise $\approx$ the bisimilarity relation as follows

$x \approx y \Rightarrow hd\ x = hd\ y \wedge tl\ x \approx tl\ y$

Actually this is itself a corecursive definition! To avoid a meaningless regress what one actually says is that anything obeying the above is a *bisimulation* and by bisimilarity we mean the largest such relation. For a fuller discussion see Pitts [5]. Taking as read this background understanding of how to avoid logical regress, we say that in general two pieces of codata are bisimilar if:

- their finite parts are equal, and
- their infinite parts are bisimilar.

The principle of coinduction may now be stated as follows: *Bisimilar objects are equal.*

One way to understand this principle is to take it as the definition of equality on infinite objects

We can package the definition of bisimilarity and the principle that bisimilar objects are equal in the following method of proof: *When proving the equality of two infinite structures we may assume the equality of recursive substructures of the same form.*

For colists we get — to prove

```
        g x1 ... xn  =  h x1 ... xn
```

It is sufficient to show

```
        g x1 ... xn  =  e <> g a1 ... an
        h x1 ... xn  =  e <> h a1 ... an
```

There is a similar rule for each codata type

A trivial example

```
>       x = 1 <> x
>       y = 1 <> y
```

How do we prove that x = y?

Theorem x = y
Proof by coinduction

```
        x
        =  1 <> x                       {x}
        =  1 <> y                       {ex hypothesi}
        =  y                            {y}
```

QED

Example: reflection on infinite trees

```
>       codata inftree = T nat inftree inftree
>       refl :: inftree -> inftree
>       refl (T a x y) = T a (refl y)(refl x)
```

Theorem refl (refl x) = x
Proof by coinduction

```
        refl (refl (T a y z)
        = refl (T a (refl z) (refl y))            {refl}
        = T a (refl (refl y)) (refl (refl z))     {refl}
        = T a y z                                 {ex hypothesi}
```

QED

Example: the (co)map-iterate theorem

The following theorem is from Bird & Wadler (see [1], page 184). We have changed the name of `map` to `comap` because for us they are different functions.

```
>       iterate f x = x <> iterate f (f x)
>       comap f (a <> x) = f a <> comap f x
```

Theorem iterate f (f x) = comap f (iterate f x)
Proof by coinduction

```
        iterate f (f x)
        = f x <> iterate f (f (f x))            {iterate}
        = f x <> comap f (iterate f (f x))      {ex hypothesi}
        = comap f (x <> iterate f (f x))        {comap}
        = comap f (iterate f x)                 {iterate}
```

QED

The proof given in [1] uses the `take`-lemma - it is longer than that given above and requires an auxiliary construction, involving the application of a `take` function to both sides of the equation, and an induction on the length of the take.

Summary

The "strong coinduction" principle illustrated here seems to give shorter proofs of equations over infinite lists than either of the proof methods for this which have been developed in the theory of weak functional programming - namely partial object induction (Turner [8]) and the take-lemma (Bird [1]).

The framework seems simpler than previous accounts of coinduction - see for example Pitts [5], because we are not working with domain theory and partial objects, but with the simpler world of total objects.

Moral: Getting rid of partial objects seems to be an unmitigated blessing - not only when reasoning about finite data, but also, perhaps even more so, in the case of infinite data.

5 Observations and Concluding Remarks

I have outlined an elementary discipline of strong (or total) functional programming, in which we have both finite data and (potentially) infinite codata, which we keep separate from each other by a minor variant of the Hindley Milner type discipline. There are syntactic restrictions on recursion and corecursion which ensure well-foundation for the former, and finite progress for the latter, and simple proof rules for both data and codata.

Although the particular syntactic discipline proposed may be too restrictive (particularly in the forms of corecursion it permits - further research is required here) I would like to argue that the distinction between data and codata is very helpful to a clean discipline of functional programming, and gives us a better teaching vehicle, and perhaps a better vehicle for production programming also (because of the greater freedom of choice for the implementor).

A question we postponed from section 2 is whether we ought to be willing to give up Turing completeness. Anyone who has taken a course in theory of computation will be familiar with the following result, which is a corollary of the Halting Theorem.

Theorem: For any language in which all programs terminate, there are always-terminating programs which cannot be written in it - among these are the interpreter for the language itself.

So if we call our proposed language for strong functional programming, L, an interpreter for L in L cannot be written. Does this really matter? I can see two arguments which suggest this might in fact be something to which we could accommodate ourselves quite easily

1) We will have a hierarchy of languages, of ascending power, each of which can express the interpreters of those below it. For example if our language L has a first order type system, we can easily add some second order features to get a language L_2, in which we can write the interpreter for L, and so on up. Constructive type theory, with its hierarchy of universes, is like this, for example.

2) There is no such theoretical obstacle to our writing a compiler for L in L, which is of far greater practical importance.

Summary

There is a dichotomy in language design, because of the halting problem. For our programming discipline we are forced to choose between

A) Security - a language in which all programs are known to terminate.

B) Universality - a language in which we can write

(i) all terminating programs

(ii) silly programs which fail to terminate

and, given an arbitrary program we cannot in general say if it is (i) or (ii).

Four decades ago, at the beginning of electronic computing, we chose (B). It may be time to reconsider this decision.

Acknowledgements: This paper was presented at the First International Symposium on Functional Programming Languages in Education, Nijmegen, Netherlands, December 1995. An earlier version of the paper was presented at the A. J. Perlis Memorial Symposium on the Future of Programming Languages at Yale University in April 1993. I am grateful to the audience at that symposium for a number of comments.

References

[1] R. S. Bird, P. Wadler "Introduction to Functional Programming", Prentice Hall, 1988.

[2] K. Gödel "On a hitherto unutilized extension of the finitary standpoint", Dialectica 12, 280-287 (1958).

[3] R. Harper, D. MacQueen, R. Milner "Standard ML", University of Edinburgh LFCS Report 86-2, 1986.

[4] Paul Hudak et al. "Report on the Programming Language Haskell", SIGPLAN Notices, 27(5), May 1992..

[5] A. M. Pitts "A Co-induction Principle for Recursively Defined Domains", Theoretical Computer Science, 124(2):195-219, 1994.

[6] Colin Runciman "What about the Natural Numbers", Computer Languages, 14(3):181-191, 1989.

[7] S. J. Thompson "A Logic for Miranda", Formal Aspects of Computing, 1(4):339-365, 1989.

[8] D. A. Turner "Functional Programming and Proofs of Program Correctness" in Tools and Notions for Program Construction, pp 187-209, Cambridge University Press, 1982 (ed. Néel).

[9] D. A. Turner "Miranda: A non-strict functional language with polymorphic types" Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (LNCS, vol 201). *This and other papers about Miranda can be found at* http://miranda.org.uk.

[†]**Note** 'Miranda' is a trademark of Research Software Ltd.