Journal of Universal Computer Science, vol. 10, no. 7 (2004), 751-768
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © *J.UCS*

Total Functional Programming

D.A.Turner
(Middlesex University, UK)
d.a.turner@mdx.ac.uk

Abstract: The driving idea of functional programming is to make programming more closely related to mathematics. A program in a functional language such as Haskell or Miranda consists of equations which are both computation rules and a basis for simple algebraic reasoning about the functions and data structures they define. The existing model of functional programming, although elegant and powerful, is compromised to a greater extent than is commonly recognised by the presence of partial functions. We consider a simple discipline of *total functional programming* designed to exclude the possibility of non-termination. Among other things this requires a type distinction between data, which is finite, and codata, which is potentially infinite.
Key Words: functional programming
Category: D.1.1

1 Introduction

In a typical modern algebra text the concept of **function** is defined as follows: *a function f with domain A and codomain B assigns to each element of x of A a unique element $f\ x$ of B.* Note that the function isn't given unless we are told its domain and codomain.

This is reflected very directly in modern strongly typed functional programming languages, such as Miranda† [Turner 1986], Haskell [Hudak et al. 1992], or the functional subset of Standard ML [Harper et el. 1986]. The domain and codomain of each function is either stated in or inferable from the program text and the functions are defined by equations, typically involving case analysis by pattern-matching. Thus we might define `fib` a function to compute the n'th member of the fibonacci series as follows[1]

```
>       fib :: Nat->Nat
>       fib 0 = 0
>       fib 1 = 1
>       fib (n+2) = fib (n+1) + fib (n+2)
```

The three equations uniquely define the assignment of values, `fib x` for each `x`. From these equations we can prove various theorems about fibonacci numbers, by using algebraic reasoning and induction. Among the theorems we can deduce

[1] The notation used for examples in this paper is an eclectic mixture of Miranda and Haskell. I have also taken the liberty of assuming a built-in type Nat of non-negative integers. Program fragments are indicated throughout by a leading >

are the values of `fib x` for specific `x`. For example

```
fib 20 = 6765
```

by using the equations from left to right, as *reduction rules*. The above functional program is thus both a mathematical definition of fib and at the same time an algorithm for computing it. One of the enduring myths about functional programming languages is that they are somehow non-algorithmic. On the contrary, the idea of functional programming is to present algorithms in a more transparent form, uncluttered by housekeeping details.

Unless we do something clever, like memoizing the function, the above is not an efficient algorithm for `fib` as it takes time exponential[2] in `n` to compute `fib n`. A more efficient program for `fib` is the tail-recursive one:

```
>       fib' n = f n 0 1
>       f 0 a b = a
>       f (n+1) a b = f n b (a+b)
```

We would like to confirm that

$\forall n \epsilon$`Nat.fib' n = fib n`

The key to this is to prove for arbitrary `p`

Theorem $\forall n \epsilon$`Nat.f n (fib p) (fib (p+1)) = fib (p+n)`
Proof by induction on `n`
Straightforward using the program equations for `f` *and* `fib` *and an induction step.*

The seemingly close fit between program text and mathematical reasoning accounts for a large part of the appeal of functional languages (together with their conciseness and expressive power) especially in a pedagogical context. But there is a serpent lurking in the garden (in fact a whole nest of them). Consider the following, perfectly legal, well-typed, program.

```
>       loop :: Int->Int
>       loop n = 1 + loop n
```

From this we have

$$loop\ 0 = 1 + loop\ 0$$

subtracting $loop\ 0$ from both sides and using $x - x = 0$ and associativity we get

$$0 = 1$$

From which we can infer anything. What went wrong?

Despite being of type Int the value of $loop\ 0$ is not an integer of the familiar kind. It is $\perp_{Int}$ the undefined integer. Because we allow unrestricted recursion

[2] Actually to compute $fib\ n$ takes time proportional to $fib\ n$ but this tends asymptotically to an exponential in n.

we are programming with *partial functions* not functions in the standard mathematical sense.

The thesis of this paper is that functional programming is a good idea, but we haven't got it quite right yet. What we have been doing is partial functional programming. What we should be doing is total functional programming.

The remaining sections of the paper are organised as follows. Section 2 introduces the idea of total functional programming. In section 3 we outline an elementary language for total 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 discuss some extensions and in section 6 make some closing remarks.

2 Total Functional Programming

In conventional functional programming if we have an expression e of type `Int` say, we know that if evaluation of e terminates successfully the result will be an integer – but the evaluation might fail to terminate, or might result in an error condition.

In total functional programming if we have a well-typed expression e of type `Int` we know that evaluation of e will terminate with an integer result. There are no run-time errors and everything terminates.

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

In total functional programming $\perp$ does not exist. The data types are those of ordinary discrete mathematics. This has three main advantages, which we now briefly consider in turn.

(A note on correct usage: the term *function* already implies totality so the term "total function" should not be used. In partial fp we program with (perhaps) partial functions, in total fp we program only with functions.)

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 $\perp_{Nat}$.

Similarly we cannot rely on usual principle of induction for Nats

$$\frac{\begin{array}{l}P(0)\\ \forall n.P(n) \Rightarrow P(n+1)\end{array}}{\forall n.P(n)}$$

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

These problems arise, in different ways, in both strict and lazy languages. In total 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 Simpler Language Design

In partial functional programming we have a fundamental language design choice forced on us at an early stage — whether to make functional application strict in the argument. That is, is it a rule of the language that for any function `f`

$$\texttt{f}\ \perp = \perp$$

SML says yes to this, as does Scheme, while Miranda and Haskell embrace non-strictness and thus lazy evaluation as the norm, leading to a far-reaching differences in programming style.

There are many other decisions to make because of the presence of $\perp$. For example should the product space $A \times B$ be lifted or non-lifted? Haskell has the first, Miranda the second — this affects the behaviour of pattern-matching. In Miranda

```
>         f (x, y) = ...
```

is *irrefutable*, that is the match cannot fail, because at type $A \times B$ we have

$$(\perp_A, \perp_B) = \perp_{A\times B}$$

whereas Haskell's product type has an extra $\perp$ below $(\perp, \perp)$ and therefore for it the pattern (x, y) doesn't match $\perp$.

These seemingly trivial decisions can interact in unexpected ways and cause innocent looking programs which work in one system to fail in another (even moving between two languages which are both lazy).

For another example take the `&` operation on `Bool`, defined by

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

but there are more cases to be defined:

```
⊥ & y = ?
x & ⊥ = ?
```

Considering the possible values for these (which are constrained by monotonicity) gives us a total of four different possible versions of `&` namely

(i) doubly strict `&`

(ii) left-strict `&`

(iii) right-strict `&`

(iv) doubly non-strict (parallel) `&`

Most current programming languages opt for (ii), the left to right version but this is somewhat arbitrary and breaks the symmetry which `&` has in logic.

In total functional programming these semantic choices go away. There is only one possible definition of the product type $A \times B$; only one & operation exists, defined by its actions on `True`, `False` alone, and so on. We no longer have a split between strict and non-strict languages — in a well-typed program every subexpression must have a proper value and the choice between normal order and applicative order evaluation won't affect the outcome.

2.3 Flexibility of Implementation

In total functional programming reduction is *strongly Church-Rosser.* Note the distinction between

(A) Church-Rosser Property[3]: *If E can be reduced in two different ways and 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 says normal forms are unique - but the normal form need not exist and where it does, not every reduction sequence will find it; with strong Church-Rosser we have uniqueness of normal form plus strong normalisability - normal forms always exist and 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 evaluation order becomes a matter for the implementor, and cannot affect the semantics.

An alternative name for total functional programming, used in [Turner 1995], and inspired by the strong Church-Rosser property, is *strong* functional programming by contrast with which conventional fp may be called *weak.*

2.4 Disadvantages

There are two obvious disadvantages of total functional programming

1. Our programming language is no longer Turing complete!
2. If all programs terminate, how do we write an operating system?

[3] More exactly the Church Rosser property of a relation $\Rightarrow$ is that if $E_1 \Rightarrow E_2 \wedge E_1 \Rightarrow E_3$ there exists E_4 such that $E_2 \Rightarrow E_4 \wedge E_3 \Rightarrow E_4$. Property A is a consequence of this.

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 powerful theory of total 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 includes:

– Dependent types (types indexed over values)

– Second order types

– An isomorphism between types and propositions, that enables programs to express proof information.

The theory was developed as a foundational language for constructive mathematics. It is certainly possible to program in it, see for example [Nordstrom et al. 1990], but it would hardly be suitable as an introductory language to teach programming.

I am interested in finding something simpler.

3 Elementary total functional programming

What I propose is something much more modest than constructive type theory, namely an *elementary* discipline of total 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 $Nat \to Nat$, and polymorphic types like $\alpha \to \alpha$, 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)

3.1 Rules for elementary total fp

First, we must be able to define data types.

```
>       data Bool = False | True
>       data Nat = Zero | Suc Nat
>       data List a = Nil | Cons a (List a)
>       data Tree = Nilt | Node Nat Tree Tree
>       data Array a = Bounds Nat Nat (Nat->a)
```

and so on.

As is usual some types - `Nat` and `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))` and correspondingly, some primitive operations such as $+$, $-$ and $>$ on Nat will be built in for efficiency, although they could easily be defined.

We define functions by the usual style of equational definition using pattern matching over data types. Eg

```
>   size :: Tree -> Nat
>   size Nilt = 0
>   size (Node n x y) = 1 + size x + size y
>
>   filter :: (a->Bool) -> List a -> List a
>   filter f Nil = Nil
>   filter f (Cons a x) = Cons a (filter f x), if f a
>                       = filter f x,          otherwise
```

In using guard syntax: `if` and `otherwise`, we assume the presence of type Bool.

There are three essential restrictions to maintain totality.

RULE 1) All case analysis must be complete. So where a function is defined by pattern matching, every constructor of the argument type must be covered and in a set of guarded alternatives, the terminating 'otherwise' case must be present.

In the same spirit any built in operations must be total. This will involve a some non-standard decisions - for example we will have

```
        0 / 0 = 0
```

Runciman [Runciman 1989] gives a useful and interesting discussion of how to make natural arithmetic closed. He argues rather presuasively that the basic arithmetic type in a functional language should be `Nat` rather than `Int`. In a total language it is certainly desirable to have `Nat` as a staticly recognised type, even if `Int` is also provided, since there are functions that have no sensible value on negative integers – factorial for example.

Making all operations total of course requires some attention at types other than `Nat` - for example we have to decide what to do about `hd`. This is more troublesome.

```
>        hd :: List a -> a
>        hd (Cons a x) = a
>        hd Nil = ...???...
```

Because `hd` is polymorphic we cannot simply assign a conventional value to `hd Nil`, for with the abolition of $\perp$ we no longer have any values of type α. The same problem will arise with any selector function on a sum type. Two simple solutions are

– Supply an extra argument to `hd`, which is the value to be returned if the list is empty.

– Don't use `hd`. Instead always do a case analysis, using pattern matching and including a case for the empty list.

Both of these are workable and force you to pay attention to exactly those boundary cases which are likely to cause trouble. The first incurs a modest overhead in passing extra arguments to various functions. The second avoids this but at the cost of a more substantial rewrite of the program. A more elegant solution would be to somehow modify the type system to admit subtypes – such as *non-empty-List,* on which `hd` is well-defined.

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 = Very (Silly->X) ||not allowed!
```

Here `X` is an arbitrary type. Contravariant types like `Silly` allow $\perp$ to sneak back in, and are therefore banned. We show how the damage arises:

```
>       bad :: Silly -> X
>       bad (Very f) = f (Very f)
>       foo :: X
>       foo = bad (Very bad)
```

We have obtained a value, `foo`, of type X, with no normal form — using the equation for `bad` to rewrite `foo` gets back the same term. The construction will work for any X, for example `Nat`. So we have an expression of type `Nat` which does not reduce to a numeral, like `loop 0` of our Introduction. The restrictions on recursion which we introduce next (RULE 3 below) will not prevent this, since the definitions of `bad` and `foo` above are not recursive. A modification of the above scheme gives a fixpoint operator, equivalent to having general recursion.

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

RULE 3) Each recursive function call must be on a syntactic subcomponent of its formal parameter. This form of recursion, often called *structural recursion,* sits naturally with function definition by pattern matching. A typical example of what this allows is recursion of the form

```
>       f :: Nat->Thing
>       f 0 = something
>       f (n+1) = ...f n...
```

which is primitive recursion, but we may recurse via pattern matching on the subcomponents of any data type, including lists and arbitrary trees, not just on `Nat`. In the case of a function of multiple arguments we also permit "nested" structural recursion as in Ackermann's function

```
>       ack :: Nat->Nat->Nat
>       ack 0 n = n+1
>       ack (m+1) 0 = ack m 1
>       ack (m+1) (n+1) = ack m (ack (m+1) n)
```

the extension to multiple arguments adds no power, because what it does can be desugared using higher order functions, but is syntactically convenient.

The rule to allow recursion only by *syntactic descent* on data constructors 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 1958] 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.

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
```

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 `List Bit`, and assuming a built in function that gives us access to the binary representation of a number.

```
>      data bit = On | Off

>      bits :: Nat->List Bit  ||built in

>      pow x n = pow1 x (bits n)
>      pow1 x Nil = 1
>      pow1 x (Cons On y) = x * pow1 (x * x) y
>      pow1 x (Cons 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 - it is a readily observed that three quarters or more 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. And it is sometimes quite inconvenient – for example Euclid's algorithm for *gcd* is difficult to express in a natural way). We return to this in section 5 below.

There is a sledge-hammer approach that can be used to rewrite as primitive recursive any algorithm for which we can compute a prinitive recursive 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.

The problem of writing a decision procedure to recognise structural recursion in a typed lambda calculus with case-expressions and recursive, sum and product types is solved in the thesis of Andreas Abel [Abel 1999]. Adapting it cope with a richer type system and a more equational style of function definition would be non-trivial but probably no harder than things that functional language compilers already do.

3.2 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.

4.1 Programming with Codata

The rule for *primitive 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 the context `RHS(-)` must be a *coconstructor,* with the corecursive call to `f` as one of its arguments. 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 total 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 strong normalisability? No! But you have to have the right definition of normal form. Every expression whose principle operator is a coconstructor is in normal form. (To get confluence as well as strong normalisability requires a little more care. Each corecursive definition is translated into a closed term and an explicit *unwind* operation introduced – see [Telford and Turner 1997] for details. The scheme in [Wadler et al. 1998] for translating lazy definitions into a strict language is related.)

4.2 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 1994]. 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. We give one example of a proof by coinduction.

The following theorem about the standard functions `map`, `iterate`, is from [Bird and Wadler 1988]. We have changed the name of `map` to `comap` because for us it is a different function when it acts on colists.

```
>       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 Bird and Wadler 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.

The absence of a base case in this form of induction is at first sight puzzling. It is important to note that coinduction is valid only for the proof of *equations* over infinite structures, not of arbitrary properties of the data structure as with ordinary induction.

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 1982] and the take-lemma [Bird and Wadler 1988].

The framework seems simpler than previous accounts of coinduction - see for example [Pitts 1994], 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 an unmitigated blessing - not only when reasoning about finite data, but perhaps even more so in the case of infinite data.

5 Beyond structural recursion?

The restriction to structural recursion is sometimes frustrating. If the compiler can understand that $(n-1)$ is smaller than n (for positive integer n) why can it not see that $n/2$ also descends from n (again for positive n)? This would enable the straightforward definition of fast exponentiation to be accepted, without our having to introduce the intermediate data type `List Bit`. A similar consideration applies to partitioning a list into two non-empty parts as in Quicksort.

A significant result in this area is the paper [Arkoudas and McAllester 1996] defining a decision procedure for ***Walther recursion***, a generalisation of primitive recursion. By a "reducer-conserver" analysis of the program the properties of descending in size from its argument and conserving the size of its argument (in a sense of "size" appropriate to the data type) is transmitted from one function to another. For example from knowing that $(a - b)$ descends from a (for a, b positive integers and '$-$' natural subtraction) and examining the definition of integer division as repeated subtraction it is inferred that $n/2$ descends.

Arkoudos & McAllester argue that Walther recursion adds no power because the recursions it accepts can be translated into primitive recursion but it adds convenience. Their system will recognise as well-founded Quicksort, *gcd* by Euclid's algorithm and many similar examples. The programs are expressed in a first order, monomorphic, functional language (essentially a simple subset of LISP).

Their system permits (and requires) the identification of simple subtypes, such as non-empty list or non-zero natural number, that are relevant to the analysis. For example natural subtraction is a conserver over natural numbers but a reducer over positive numbers and (perhaps rather inconveniently) has to be defined separately for these two types.

Generalising the definition and decision procedure for Walther recursion to a higher order polymorphic language is an important and (as far as I know) unsolved challenge. This would certainly make elementary total fp a more attractive proposition by admitting a wider and more natural class of recursive definitions.

A method of abstract interpretation due to Alastair Telford captures much of the same ground as Walther recursion for a simple higher order (but still monomorphic) programming language, see [Telford and Turner 2000].

There is a version of Telford's abstract interpretation scheme for codata. Interestingly, this recognises a class of valid corecursive definitions which includes primitive corecursion but also allows other examples such as

```
>       evens = 2 <> comap (add 2) evens
```

which fails to be primitive corecursion because of the intervening call to `comap`. The definition is neverthless productive[4] (the dual concept to well-founded) because of a conservative property of comap. See [Telford and Turner 1997].

This is suggests there is a notion of *Walther corecursion* which works analogously to the way in which Walther recursion extends the scope of primitive

[4] productivity is of course, like well-foundation, undecidable in general

recursion. The whole area merits further investigation.

6 Observations and Concluding Remarks

I have outlined an elementary discipline of total functional programming, in which we have finite data and possibly-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 to ensure well-foundation for the former and productivity for the latter and simple proof rules for both data and codata.

Although the syntactic discipline proposed may be found too restrictive in the forms of recursion and corecursion it allows, I would argue that the distinction between data and codata is very helpful to a clean system for functional programming and is in fact necessary within a framework ensuring totality.

The attraction of an *elementary* total language in the sense defined at section 3 is primarily pedagogical. In the presence of dependent types the expressive power of structural recursion is greatly enhanced, see for example [McBride 2003] for a nice illustration of this.

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 total functional programming, L, an interpreter for L in L cannot be written. Does this really matter? I have two observations which suggest this might in fact be something to which we could accommodate ourselves quite easily.

1) We can 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 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) We can draw an analogy with the (closely related) issue of compile-time type systems. If we consider a complete computing system written in a typed high level language, including operating system, compilers, editors, loaders and so on, it seems that there will always be at least one place – in the loader for example – where we are obliged to escape from the type discipline. Nevertheless many of us are happy to do almost all of our programming in languages with

compile time type systems. One the rare occasions when we need to we can open an escape hatch, such as Haskell's *UnsafePerformIO*.

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).

Five decades ago, at the beginning of electronic computing, we chose (B). If it is the case, as seems likely, that we can have languages of type (A) which accomodate all the programs we need to write, bar a few special situations, it may be time to reconsider this decision.

Acknowledgements

This work was supported in part by EPSRC grant GR/L03279. An earlier version of this paper, including the coinduction method, was presented in [Turner 1995].

References

[Abel 1999] Andreas Abel "A Semantic Analysis of Structural Recursion", Diploma Dissertation, 50 pages, Ludwigs-Maximillians-University, Munich, February 1999.

[Arkoudas and McAllester 1996] Kostas Arkoudas, David McAllester "Walther Recursion", Proceedings CADE 13, Springer LNCS 1104:643-657, 1996.

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

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

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

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

[McBride 2003] Conor McBride "First Order Unification by Structural Recursion", Journal of Functional Programming 13(6):1061-1075 , November 2003.

[Nordstrom et al. 1990] B. Nordstrom, K. Petersson, J. M. Smith "Programming in Martin-Löf's Type Theory: An Introduction", Oxford Science Publications, 1990.

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

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

[Telford and Turner 1997] A.J.Telford, D.A.Turner "Ensuring Streams Flow", Johnson, ed, Algebraic Methodology and Software Technology - AMAST '97, Sydney, Australia, December 1997 (Springer LNCS vol 1349:509-523).

[Telford and Turner 2000] A.J.Telford, D.A.Turner "Ensuring Termination in ESFP", Journal of Universal Computer Science, 6(4):474-488, April 2000.

[Turner 1982] 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).

[Turner 1986] D. A. Turner "An Overview of Miranda", SIGPLAN Notices, 21(12):158–166, December 1986. *This can also be found at* http://miranda.org.uk.

[Turner 1995] D.A.Turner "Elementary Strong Functional Programming". In R.Plasmeijer, P.Hartel, eds, First International Symposium on Functional Programming Languages in Education, Nijmegen, NL, Dec 1995 (Springer LNCS, vol 1022:1-13).

[Wadler et al. 1998] Philip Wadler, Walid Taha, David McQueen "How to add laziness to a strict language without even being odd", 7 pages, Workshop on Standard ML, Baltimore, September 1998.

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