Calculating Functional Programs

Jeremy Gibbons
School of Computing and Mathematical Sciences
Oxford Brookes University
jgibbons@brookes.ac.uk

November 21, 1997

Abstract

A good way of developing a correct program is to *calculate* it from its specification. Functional programming languages are especially suitable for this, because their referential transparency greatly helps calculation. We discuss the ideas behind program calculation, and illustrate with an example (the *maximum segment sum* problem). We show that calculations are driven by *promotion*, and that promotion properties arise from *universal properties* of the data types involved.

1 Context

The history of computing is a story of two contrasting trends. On the one hand, the cost and cost/performance ratio of computer hardware plummets; on the other, computer software is over-complex, unreliable and almost inevitably over budget. Clearly, we have learnt how to build computers, but not yet how to program them.

It is now widely accepted that ad-hoc approaches to constructing software break down as projects get more ambitious. A more formal approach, based on sound mathematical foundations, is required.

1.1 Formal approaches to constructing software

One such more formal approach is *program verification*, whereby a program constructed by traditional methods is mathematically proved correct. Although this approach helps with quality *assurance*, it provides no help with

[0]Presented at the ISRG/SERG Research Colloquium, School of Computing and Mathematical Sciences, Oxford Brookes University, 5th November 1997. An earlier version of this paper appeared as *An Introduction to the Bird-Meertens Formalism*, in *Proceedings of the New Zealand Formal Program Development Colloquium*, edited by Steve Reeves, Hamilton, New Zealand, November 1994 [6].

constructing a high-quality program in the first place. A better approach is *program derivation*, whereby the correctness requirements of a program are gradually transformed, by the application of meaning-preserving transformations, into a program that satisfies them. Provided that the transformations are correct and corectly applied, the resulting program necessarily satisfies its requirements. The problem then reduces to formulating and applying a useful body of meaning-preserving transformations.

1.2 Functional languages

Program derivation has enjoyed some considerable interest over the past two decades. Early work concentrated on deriving programs in imperative languages, such as Dijkstra's Guarded Command Language. More recently, it has been realized that functional languages offer to the program deriver a number of advantages over imperative languages. The essential advantage is that functional languages are higher-level, and so more abstract. This has the following consequences:

- functional programs express fewer implementation details than imperative programs, so programs are shorter and easier to reason about;
- functional languages are referentially transparent, so reasoning in them is closer to normal mathematics;
- functional languages can often be used to express the specification as well as the solution, so the derivation can be carried out in a single formalism—with imperative languages, the derivation typically alternates between the programming language and predicate calculus.

Therefore, interest over the past decade or so has been more on deriving programs in functional languages than in imperative languages.

1.3 Program calculation

We have seen that one advantage of functional languages is that they are often sufficiently abstract to express both the specification and the resulting program. This allows the derivation to proceed directly in a single formalism, rather than indirectly via the predicate calculus. We use the term *program calculation*, as opposed to simply program derivation, for such a direct derivation. It is often possible to present the derivation in the form

$$
\begin{array}{ll}
 & \text{original specification} \\
= & \quad \{ \text{ justification for first step } \} \\
 & \text{intermediate version} \\
= & \quad \{ \text{ justification for next step } \} \\
 & \qquad \vdots \\
= & \quad \{ \text{ justification for last step } \} \\
 & \text{final program}
\end{array}
$$

We call such a calculation *equational reasoning*, and we aim to present as much of the derivation as possible in this calculational style. This is for expository reasons: it clarifies the distinction between the routine parts of the development (those that consist of straightforward steps, using 'current technology') and the creative parts (those that we do not yet know how to calculate, but that require invention instead). In communicating the *essence* of a development, the calculational parts can be elided to just their first and last lines,

$$
\begin{array}{ll}
 & \text{original specification} \\
= & \quad \{ \text{ routine calculation } \} \\
 & \text{final program}
\end{array}
$$

allowing proper emphasis to be placed on the remaining parts, the *design decisions*.

Put another way, we are interested in extending what can be calculated precisely because we are not that interested in the calculations themselves. Developing techniques that allow more of a derivation to be calculated enables us to focus on the more important parts instead.

1.4 A calculus

Such an equational calculational style naturally entails a powerful collection of notations and theorems—a *calculus* for program construction—to provide the individual steps. Moreover, these theorems should be expressed as far as is possible as equations with just a few simple applicability conditions, so that the flow of the calculation need not be interrupted. Inductive proofs are eschewed as far as possible. The calculus that we have in mind, the so-called *Bird-Meertens Formalism* [4, 11], uses ideas from universal algebra and category theory to provide such a body of theorems. In particular, the theorems arise as properties of common patterns of computation over the data structures concerned. In most cases, the datatype definition is all that needs to be stated; everything else follows automatically from it.

2 An example

To illustrate the concept of calculating functional programs, we present an example, that of computing the *maximum segment sum* of a list of numbers. This is a classical problem in programming texts, largely because it is a simple problem with a very elegant yet not at all obvious linear-time solution. Bentley [2] uses it as an argument for the benefits of algorithm design, developing in turn cubic, quadratic, $O(n \log n)$ and linear algorithms. Kaldewaij [9], among others, use it as we do, as an illustration of program calculation; in particular, this development is inspired by [5].

We use the notation of Gofer [8], but we introduce the necessary notation as we go along.

2.1 The problem

Given a list of numbers, the *maximum segment sum* problem is to compute the maximum of the sums of the contiguous *segments* of that list. For example, when the list consists of the numbers

$$31 \quad -41 \quad 59 \quad 26 \quad -53 \quad 58 \quad 97 \quad -93 \quad -23 \quad 24$$

the greatest sum possible, 187, is the sum of the segment that omits the first two and the last three elements of the list.

2.2 Lists

A list is either empty (written `[]`), or is constructed from *consing* a new element `a` onto a shorter list `as` (written `a : as`). One basic operation on lists is `map`, which applies a function to every element of the list:

```
map f [] = []
map f (a:as) = f a : map f as
```

2.3 Folds

Basic as `map` is, the most fundamental operation on lists is the `foldr`, which essentially encapsulates the natural pattern of recursion over lists. It is defined by

```
foldr f e [] = e
foldr f e (a:as) = f a (foldr f e as)
```

This 'natural pattern of recursion' over lists is determined completely by the definition of lists as a datatype. To see that folds are more fundamental than maps, note that all maps are folds,

```
map f e = foldr fm []  where  fm a bs = f a : bs
```

but that some folds are not maps (we see some examples below).

Some other common examples of folds are:

- the function `sum`, which sums the elements of a list:

```
sum = foldr (+) 0
```

- the function `maximum`, which finds the largest element of a list:

  ```
  maximum = foldr max minfinity
  ```

  where `minfinity` represents $-\infty$, the 'largest element' of the empty list (in this paper, we do not actually need the maximum of the empty list—it is safe for `maximum []` to be undefined);

- the operator `++`, which concatenates two lists:

  ```
  x ++ y = foldr (:) y x
  ```

- the function `concat`, which concatenates a list of lists into one long list:

  ```
  concat = foldr (++) []
  ```

- the function `inits`, which returns all initial segments of a list:

  ```
  inits = foldr fi [[]]  where  fi a xss = [] : map (a:) xss
  ```

  (where `(a:)` is a *sectioned* operator, representing the function that takes `as` to `a:as`);

- the function `tails`, which returns all tail segments of a list:

  ```
  tails = foldr ft [[]]  where  ft a xss = (a : head xss) : xss
  ```

  (where the function `head` satisfies

  ```
  head (a:as) = a
  ```

  and is defined only for non-empty lists);

- the function `scanr`, which returns all partial results computed during a `foldr`:

  ```
  scanr f e = foldr fs [e]  where  fs a bs = f a (head bs) : bs
  ```

  (so that `tails` is `scanr (:) []`).

Using these components, we can define the functions `segs`, which returns all contiguous segments of a list:

```
segs = concat . map inits . tails
```

(that is, the segments of a list are the initial segments of the tail segments of that list), and hence formally define the problem:

```
mss = maximum . map sum . segs
```

This program takes cubic time (there are quadratically many segments, and finding the sum of each takes linear time); the problem is to find a linear-time algorithm for `mss`.

2.4 Universal properties

The main reason that programming with `foldr` is so attractive is the existence of a *universal property*, stating that the equations in `h`

```
h [] = e
h (a:as) = f a (h as)
```

have a unique solution, namely `foldr f e`. Thus, to show that some complicated expression `h` is equal to some function `foldr f e`, it suffices to show that `h` satisfies the above equations. The universal property is a kind of 'canned induction'. Like the fold itself, the universal property too arises for free from the datatype definition.

As an illustration of the universal property, we present the so-called *scan lemma*,

```
map (foldr f e) . tails = scanr f e
```

which can be thought of as an alternative characterization of `scanr`. Now, `scanr` is an instance of `foldr`, by definition:

```
scanr f e = foldr fs [e]  where  fs a bs = f a (head bs) : bs
```

so it suffices to show that

```
map (foldr f e) (tails []) = [e]
```

and

```
map (foldr f e) (tails (a:as)) = fs a (map (foldr f e) (tails as))
```

The first is easily verified; for the second, we have

```
     map (foldr f e) (tails (a:as))
=        { tails }
     map (foldr f e) ((a : head (tails as)) : tails as)
=        { map }
     foldr f e (a : head (tails as)) : map (foldr f e) (tails as)
=        { foldr }
     f a (foldr f e (head (tails as))) : map (foldr f e) (tails as)
=        { h (head xss) = head (map h xss) }
     f a (head bs) : bs where bs = map (foldr f e) (tails as)
=        { fs }
     fs a (map (foldr f e) (tails as))
```

2.5 Promotion

Program calculation of kind we are interested in is driven largely by *promotion properties*, which change the order in which certain computations take place. One example is the following, stating that two maps can be combined into one:

```
map f . map g = map (f . g)
```

—read from right to left, the map is promoted through the function composition. Since the right-hand side is a fold, this promotion property can be proved using the universal property.

Two other examples that we will use are as follows. The first states that a map can be promoted through `concat`:

```
map f . concat = concat . map (map f)
```

The second states that a fold can be similarly promoted:

```
foldr f e . concat = foldr f e . map (foldr f e)
```

provided that `f` is associative with unit `e`.

It may not be immediately obvious how to prove these last two properties, because neither side of either equation is obviously a fold. However, there is a general theorem (itself another promotion property, and another consequence of the universal property) that a map can always be combined with a fold:

```
foldr f e . map g = foldr f' e  where f' a b = f (g a) b
```

This is the necessary catalyst, allowing our two promotion properties to be proved using the universal property.

2.6 The calculation

We are now ready to proceed with the calculation. We have:

```
     maximum . map sum . segs
=        { segs }
     maximum . map sum . concat . map inits . tails
=        { promotion through concat }
     maximum . concat . map (map sum) . map inits . tails
=        { promotion through concat again }
     maximum . map maximum . map (map sum) . map inits . tails
=        { promotion of map through composition }
     maximum . map (maximum . map sum . inits) . tails
=        { suppose maximum . map sum . inits = foldr f e }
     maximum . map (foldr f e) . tails
=        { scan lemma }
     maximum . scanr f e
```

This completes the derivation; provided that `f` and `e` can be computed in constant time, the whole takes linear time. All that remains is to figure out how to write `maximum . map sum . inits` as a fold `foldr f e`. Once again, the universal property comes into play; it suffices to find `f` and `e` such that

```
maximum (map sum (inits [])) = e
```

and

```
maximum (map sum (inits (a:as))) = f a (maximum (map sum (inits as)))
```

Simplifying the former shows that `e` must be `0`. For the latter, we have to show how to promote `maximum . map sum . inits` through cons; we have

```
     maximum (map sum (inits (a:as)))
=        { inits }
     maximum (map sum ([] : map (a:) (inits as)))
=        { map }
     maximum (sum [] : map (sum . (a:)) (inits as))
=        { sum [] = 0; sum (a:as) = a + sum as }
     maximum (0 : map ((a+) . sum) (inits as))
=        { maximum }
     max 0 (maximum (map (a+) (map sum (inits as))))
=        { distributivity: maximum . map (a+) = (a+) . maximum }
     max 0 (a + maximum (map sum (inits as)))
```

Thus,

```
maximum . map sum . inits = foldr f 0  where  f a b = max 0 (a + b)
```

and, with this `f`,

```
mss = maximum . scanr f 0
```

3 Looking forward

We have shown how folds are the fundamental pattern of recursion over lists. Malcolm [10] shows that the same applies to any suitable datatype: there is a 'fold' and a 'universal property', and these arise for free from the datatype definition. Moreover, Meijer *et al* [12] and others have shown that similar results hold for a *dual* of folds, called unfolds; instead of collapsing a data structure to a value, as folds do, unfolds *generate* a data structure from a value; they are just as fundamental a concept as folds.

In some situations, functions are not sufficiently general to express a natural specification. This is particularly the case when the specification is non-deterministic or under-determined; also, many problems are most naturally specified using inverses, which do not come naturally in the functional world. In answer to this, Backhouse *et al* [1] and Bird and de Moor [3] are working on calculi for calculating *relational* programs. It remains to be seen whether it is worth paying the extra complexity for the extra generality and power provided by relational languages.

It has also been shown that the same ideas can be applied to calculating parallel programs [13] and hardware designs [7].

References

[1] Roland Backhouse and Paul Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, pages 7–42. Springer-Verlag, 1993.

[2] Jon Bentley. *Programming Pearls.* Addison-Wesley, 1986.

[3] Richard Bird and Oege de Moor. *The Algebra of Programming.* Prentice-Hall, 1996.

[4] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

[5] Richard S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, April 1989.

[6] Jeremy Gibbons. An introduction to the Bird-Meertens Formalism. In Steve Reeves, editor, *Proceedings of the First New Zealand Formal Program Development Colloquium*, pages 1–12, Hamilton, November 1994.

[7] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design.* North-Holland, 1990.

[8] Mark P. Jones. *Gofer Manual.* Department of Computer Science, Yale University, 1991.

[9] Anne Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall, 1990.

[10] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[11] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra,

editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[12] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *LNCS 523: Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[13] David B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.