Formulating Haskell*

Simon Thompson
Computing Laboratory, University of Kent
Canterbury, CT2 7NF, U.K.
E-mail: `sjt@ukc.ac.uk`

Abstract

The functional programming language Haskell is examined from the point of view of proving programs correct. Particular features explored include the data type definition facilities, classes, the behaviour of patterns and guards and the monad approach to IO in the Glasgow Haskell compiler.

1 Introduction

Haskell [Hudak *et al.*, 1992, Hudak and Fasel, 1992] is a lazy functional programming language which is likely to become a *de facto* as well as a *de jure* standard academically and commercially. It is often said that a crucial part of the appeal of functional languages is the ease with which proofs concerning functional programs can be written. It is also widely accepted that if proof is to be taken seriously, it needs to be formalised, and to be checked by machine[1].

The aim of this paper is to initiate discussion on the form that a logic for Haskell might take. It is based to some degree on the author's previous work on devising a logic for Miranda, [Thompson, 1989], which in turn builds on earlier work in the area. Implementation of the logic for Miranda is in progress at the University of Kent (funded by SERC grant GR/F 29134), and it is expected that the system will be freely available within the next year.

The paper begins with a discussion of the carrier logic for the axiomatisation, and the relation between the semantics of the language and the logic. Next types and their operators, including equality, are axiomatised. Classes are unique to Haskell, and their effect on verification is discussed in Section 4. The logical form of definitions is described next, and in particular the unfortunate interaction between pattern matching and guards is explored. IO in the Glasgow Haskell compiler is defined in terms of an IO monad – we show how the monadic approach is amenable to formal treatment in section 6. The paper concludes with a number of miscellaneous points.

One issue addressed throughout the paper is the choice of which parts of the language are amenable to formal treatment. For example, only some of the numeric types are logically tractable.

*To appear in J. Launchbury & P.M. Sansom (eds.), *Functional Programming, Glasgow 1992*, Springer Verlag, Workshops in Computing, 1992.

[1]This paradigm is different from that prevailing in mathematics, where proofs are subject to social scrutiny, but the nature of proofs about programs seems to be sufficiently different to make this sort of check highly unlikely.

2 Logic

The basic intuition behind a functional program is that expressions denote values and expressions of ground type are evaluated to give printable values or results. It seems sensible for the logic to be one of *equations* between expressions of the language. These will be written

```
  e ≡ f
```

where `e` and `f` are Haskell expressions. The symbol $\equiv$ is used in preference to `=` and `==` for reasons which will become clear. The relation $\equiv$ is an equivalence relation, and obeys Leibniz's law: equals may be substituted for equals in terms and formulas.

Since the language is typed, so should the logic itself, making an equation of the form above invalid unless `e` and `f` have the same (or unifiable) types. (Logicians call this a *many-sorted* logic.)

Expressions in Haskell need not denote a defined value – evaluation of an expression may loop indefinitely, for instance. This phenomenon may be rendered logically in a number of ways: expressions can be permitted not to take values, giving a partial logic, or their value may be the undefined value, or 'bottom', $\bot$. The latter 'LCF' approach is adopted here, it is both simple and sufficiently expressible; further discussion of the issue can be found in [Thompson, 1989].

What should be the logic in which these equations are embedded? Lacking arguments to the contrary, the simplest option of first-order (many-sorted) predicate calculus seems appropriate. It is open whether it should be classical or constructive; note however that for simple $\forall\exists$ (or Π^0_2) statements, such as the assertion that a function from `Int` to `Int` is total, their strength is the same.

As a part of the standardisation exercise, it seems that a formal semantics for the language will emerge. This allows the possibility that the soundness of a proposed logic can be verified: every assertion or theorem of the system can be checked to see whether it is indeed valid in the formal interpretation. Soundness is clearly a minimal requirement, but can be problematic depending upon the form of semantics adopted: issues of full abstraction and the case of parallel `or` come to mind.

Again given the semantics, the converse to soundness can be examined: is every assertion validated by the semantics derivable in the logic? Such a result is unlikely in view of Gödel's incompleteness results, but *relative* completeness results have been established for Hoare logics of imperative languages.

3 Types

This section examines the way that types and operations over them are to be axiomatised.

3.1 Algebraic Types

Algebraic types can be treated in a uniform way, as they come with constructors and definition by pattern matching, but no predefined operators.

The `Bool`ean type is a typical example of a simple algebraic type: it is an enumerated type. Its elements are `True`, `False` and $\bot$, and therefore theorems valid in Boolean algebra, like

`x || not x` $\equiv$ `True`

will *fail* to be valid, since

$\bot$ `|| not` $\bot \equiv \bot$

This was taken as one justification for introducing the quantifier $\forall_{\mathtt{def}}$ in the logic for Miranda. Familiar theorems are rendered thus

$(\forall_{\mathtt{def}}\ \mathtt{x :: Bool})(\mathtt{x\ ||\ not\ x} \equiv \mathtt{True})$

In a similar way, the induction principle (or elimination rule) for the type can take two forms

$$\frac{\mathtt{P(True)}\quad \mathtt{P(False)}}{(\forall_{\mathtt{def}}\ \mathtt{x :: Bool}).\mathtt{P(x)}}$$

and

$$\frac{\mathtt{P(True)}\quad \mathtt{P(False)}\quad \mathtt{P}(\bot)}{(\forall\ \mathtt{x :: Bool}).\mathtt{P(x)}}$$

It is a matter of general principle that from the latter rule can be derived the rule of exhaustion:

$(\forall\ \mathtt{x :: Bool})(\mathtt{x} \equiv \mathtt{True} \vee \mathtt{x} \equiv \mathtt{False} \vee \mathtt{x} \equiv \bot)$

On the other hand, an axiom to assert the distinctness of the constructors is required:

$(\mathtt{False} \not\equiv \mathtt{True} \wedge \bot \not\equiv \mathtt{False} \wedge \bot \not\equiv \mathtt{True})$

It is logically sufficient to assert one such axiom, as from this those at other types can be derived.

A general algebraic type like that of lists is rather more complex. The type is inhabited by finite lists like `[2,3,4]`, but also by partial lists, such as $\mathtt{[2,}\bot\mathtt{,3]}$ and $\mathtt{4:}\bot\mathtt{:3:}\bot$ and by infinite lists like `[1,3,..]`. Some theorems for lists are valid over the whole type

`map (f.g)` $\equiv$ `map f . map g`

whilst others are only valid for finite lists (possibly containing $\bot$)

`reverse (reverse x)` $\equiv$ `x`

and still others are restricted to finite lists of *defined* elements

`product x == 0` $\equiv$ `elem 0 x`

Properly to reflect these differences, in [Thompson, 1989] restricted quantifiers and induction rules were introduced. Typical of such a rule is

$$\frac{\mathtt{P([])}\quad (\forall_{\mathtt{def}}\ \mathtt{x :: a})(\forall_{\mathtt{def}}\ \mathtt{l :: [a]}).\mathtt{P(l)} \Rightarrow \mathtt{P(x:l)}}{(\forall_{\mathtt{def}}\ \mathtt{l :: [a]}).\mathtt{P(l)}}$$

This characterises the finite lists of defined elements. Note that often the proof of the second hypothesis will be a consequence of

$$(\forall_{\mathtt{def}}\ \mathtt{x :: a})(\forall\ \mathtt{l :: [a]}).\mathtt{P(l)} \Rightarrow \mathtt{P(x:l)}$$

in which the list `l` will simply be an arbitrary list.

The *infinite* lists are described quite differently. They can be seen as members of a *greatest* fixed point, with equality over them characterised by a bisimulation-like co-induction principle [Pitts, 1992]. Pitts characterises the infinite elements as a greatest fixed point of an inductive definition, and defines the equality relation over the set of elements in a similar way. For lists the principle states that two lists `l` and `m` are equal if $\mathtt{l} \asymp \mathtt{m}$ for some *pre-equality* relation $\asymp$. The relation $\asymp$ is a pre-equality relation if and only if for all `l` and `m`,

$$\mathtt{l} \asymp \mathtt{m} \Rightarrow (\mathtt{l} \equiv [\,] \equiv \mathtt{m}) \vee (\mathtt{l} \equiv \mathtt{a} : \mathtt{l}' \wedge \mathtt{m} \equiv \mathtt{b} : \mathtt{m}' \wedge \mathtt{a} \equiv \mathtt{b} \wedge \mathtt{l}' \asymp \mathtt{m}')$$

The attractive feature of his approach is that it works for all algebraic types, such as solutions of

```
   data LambdaModel = Fun (LambdaModel -> LambdaModel)
```

giving a characterisation of equality of these elements of a model of the untyped λ-calculus. The disadvantage of his approach is that the principle requires a *second-order* logic for its formulation, since two elements are equal if *for some pre-equality relation* .... A similar problem presents itself with induction principles, of course, and the usual expedient is to replace the characterisation by a *schema*, restricting the pre-equality relations to those which are definable in the logic.

Alternatively, mathematical induction may be used to define equality over types which appear only in the range position of a function-space constructor on the right-hand side of their definitions. For lists we have:

$\mathtt{l} \equiv \mathtt{m}$ if and only if $(\forall_{\mathtt{def}}\ \mathtt{n :: Nat})(\mathtt{take\ n\ l} \equiv \mathtt{take\ n\ m})$
where `take` is defined in the standard prelude.

This fragmentation of the principles of induction over the type of lists seems to be unavoidable: the general principle of induction for lazy lists is too weak to include the others as special cases, since it is restricted to *admissible* predicates. [Paulson, 1987] contains details of the general rule as well as providing a good background reference to the LCF approach.

3.2 Built-in Types

Built-in types are akin to abstract types: no direct access is given to the (machine) representation, rather manipulation is through predefined operators. Axioms for these types therefore have to reflect as much of the structure of the type as is thought necessary. In the case of floating-point and complex numbers, it seems highly unlikely that any satisfactory (but sound!) axiomatisation exists, and we would argue that this part of the language be omitted from the logic[2]

It is therefore appropriate to restrict attention to the integral and rational types. Even then, giving a sufficiently abstract presentation of the fixed-precision integers, `Int`, is difficult, so we restrict attention to the full integers, `Integer`. In the Miranda logic a subtype of the integers, `Nat`, is introduced, allowing theorems on properties of natural number-functions to be expressed directly. This means that Miranda definitions have to be read as being overloaded in a limited way, but it presents no theoretical difficulties.

How are the integers and operations over them axiomatised? The system will include the *graphs* of the operations, giving their values at each argument sequence. This will not be enough to axiomatise the primitive operations, `primQuotRem` and the like, which will also be specified by their primitive recursive definitions. From these definitions can be derived the usual theorems such as associativity of `(+)`, or indeed the results themselves may be included

[2]The IEEE characterisation of floating point operations seems to be too low level to be usable in verification. It is perfect for the specification of a floating-point unit, for instance.

as primitives. (From the point of view of a logical characterisation, any built-in operation adds some uncertainty as to its precise behaviour.)

Function types are characterised by function composition, which is a defined operation, and (logical) equality between functions. Basic to functional programming is that functions are characterised by their behaviour, meaning the values they return, and that programs have the same behaviour when equals are substituted for equals. Equality on functions is given by the *extensionality rule*:

$$\frac{(\forall \mathtt{x} :: \mathtt{a})(\ \mathtt{f\ x} \equiv \mathtt{g\ x}\ )}{\mathtt{f} \equiv \mathtt{g}}$$

This rule is adhered to by Haskell – it is a matter of some delicacy in language design to ensure that this is the case. Tuples do *not* obey the extensionality condition. This is because

```
fst (⊥,⊥) ≡ ⊥          fst ⊥ ≡ ⊥
snd (⊥,⊥) ≡ ⊥          snd ⊥ ≡ ⊥
```

giving $(\perp,\perp)$ and $\perp$ the same components, but because of the behaviour of pattern matching over pairs, the function

```
test (x,y) = 27
```

returns 27 on (⊥,⊥) and ⊥ on ⊥. This adds a slight complication to the characterisation of equality. Any advantage of such a definition is at the *implementation* level: to check whether a member of a product type is defined (i.e. unequal to $\perp$) simply requires a check that it is a pair; if $\perp$ and $(\perp,\perp)$ are identified, the pair (e_1, e_2) is defined if and only if one of the expressions e_1, e_2 is defined, so a parallel evaluation of the expressions e_1 and e_2 is required. More is said about pattern matching in general in Section 5.

4 Classes

Classes give a general treatment of *overloading* or *ad hoc* polymorphism. Functions which are polymorphic in the usual sense of parametric polymorphism are amenable to uniform treatment. At each type the same defining equation is used, with the same logical characterisation. How much will this be true of type classes; in other words, how much can logical structure be built on top of the class structure?

There seem to be two distinct cases. On the one hand, classes like `Text` and all the numeric classes will in general *fail* to share any significant properties. In the first case this is plain, but for numeric classes, it is an unfortunate truth that, for example, addition on `Int` and `Integer` behave in fundamentally different ways, even though the two types inhabit the same classes.

More optimistically, for the classes `Eq` and `Ord` the intention is that operations of equality and ordering are defined on their members. Even if the implementations are different, all equality relations should be (partial) equivalence relations on their domains; all orderings should be pseudo-partial orderings. (They will not be total since they will in general fail to be reflexive.) In the logic this could be reflected by a *logical class*.

```
logical class (Eq a) => Equality a
where
symm  is (∀ x,y :: a)( x==y ≡ y==x )
trans is (∀ x,y,z :: a)
          ( x==y ≡ True ∧ y==z ≡ True ⇒ x==z ≡ True)
```
For a type to inhabit this logical class, *proofs* of the theorems `symm` and `trans` have to be given.

The method of `derived` instantiations could be extended to the logic. A symmetrical and transitive relation on type `a` will be extended to a similar relation on type `[a]` by the standard definition of equality, for instance.

A similar treatment of ordering is possible.
```
class (Eq a) => DefEq a
where
defined :: a -> Bool
defined x = x==x

logical class (Ord a) => Ordering a
where
asymm is (∀ x,y :: a)
          ( x<=y ≡ True ∧ y<=x ≡ True ⇒ x==y ≡ True)
total is (∀def x,y :: a)( x<=y ≡ True ∨ y<=x ≡ True )
trans is ...
```
In these examples, the same name is given to occurrences of the same formula at different type instances: the polymorphism in the formula is parametric. The *proofs* of the formulae are defined differently at different types, an *ad hoc* overloading. It is also possible to give an *ad hoc* overloading to names with, for instance, `exhaustion`, used to name the appropriate axiom of exhaustion at each type.
```
exhaust.Bool is (∀ x :: Bool)(x ≡ True ∨ x ≡ False ∨ x ≡ ⊥)
exhaust.Nat is
     (∀ x :: Nat)(x ≡ ⊥ ∨ x ≡ 0 ∨ (∃ df y :: Nat)(x ≡ y+1))
```
Whether this mechanism has other than mnemonic value remains to be seen.

5 Definitions

Haskell definitions have the form of equations, so it is plausible that the `=` symbol of the language can simply be replaced by the ≡ of the logic. In a simple definition of the form
```
f x y = x+y*y
```
this is the case, but the addition of pattern matching, guards and scopes (in the form of `where` and `let`) makes the situation substantially more complicated. A function will be defined by a sequence of equations, and the order of these will be significant. Take the case of
```
g [13]    = 27
g (a:b:x) = 32
```
Given the argument `[bot,2,3]`, when `bot` is defined by
```
bot = bot
```
the result of evaluating `g [bot,2,3]` is undefined (at least with Gofer and the Glasgow prototype – `hbi` gives a result!). Re-ordering the equations gives

the result 32 on the same argument. A thorough analysis of the sequential nature of pattern matching both within and between clauses of a definition is required to give a full rendering of a definition by pattern matching. Details of the transformation are explained in [Thompson, 1989], with the example of g above giving

```
    g [13] ≡ 27  ∧  (a==a ≡ True ⇒ g (a:b:x) ≡ 32)
```

The rules mentioned cover pattern matching and guards within function definitions.

5.1 Patterns & Guards

One aspect of definitions is particularly complicated to explain; this is the interaction of pattern matching and guards. In definitions without guards, such as

```
    f  p1 =  e1
    f  p2 =  e2
     ...
```

if an argument matches more than one pattern, the first matching equation will be used. In the case that guards are added,

```
    f  p1 |  g11 =  e11
          |  g12 =  e12
             .....
    f  p2 |  g21 =  e21
          |  g22 =  e22
             .....
     ...
```

it may be possible for an argument to match p_1, but to *fail* the guards g_{11}, g_{12}, ...and so to 'fall through' to the subsequent equation. The 'entry conditions' for this equation are no longer being in the complement of the pattern p_1, since now there is the possibility of being in the unification of p_1 and p_2 combined with failing the conjunction of the guards g_{11}, g_{12}, ... In such a case the logical translation must treat the second equation in two different ways. The first gives a rewrite of the second equation to

```
    f σ( p2) | σ( g21) = σ( e21)
             | σ( g22) = σ( e22)
             .....
     ...
```

where σ is the most-general substitution unifying the patterns p_1 and p_2, guarded by the expression

```
    σ( g11) && σ( g12) && ...
```

whilst the second is given by unifying the complement of p_1 with p_2.

This problem is made worse still if the first equation has local definitions given by a `where` clause. In such an eventuality, the guards will use the identifiers defined locally, and it is not enough to use the same definitions redefined to use the pattern variables given by the substitution σ, since name clashes may result with definitions local to the second equation; renaming of local definitions will be necessary in general.

A simple way of removing this problem is to add a compulsory `Else` or `True` case,

```
f  p1 |  g11 =  e11
      |  g12 =  e12
         .....
      |  True =  e1k
```

so that once a pattern is matched by an argument, the function is committed to using this pattern. Many functions have this feature, and it is not clear that the extra power of avoiding it in certain circumstances is worth the extra effort required to understand function definitions. This undesirable feature is shared by Miranda, but not by Standard ML, in which guards are replaced by the

```
if ... then ... else ...
```

expression.

One feature differentiating Haskell from Miranda is the strength of the expression language. In Miranda, functions can only be defined in a series of equations: there is no explicit lambda, and in particular, case analyses are always in the form of a series of equational clauses. This restricts the programmer somewhat, but is an advantage when explaining the language in a logical form. The description of Haskell needs to include in some way or another how the `case` and lambda expression forms evaluate: these will need a collection of *axiom schemes* to cover all eventualities. In Miranda, by contrast, the explanation is given in the logical translation of the function definitions.

5.2 Irrefutable Patterns

Haskell introduces the notion of an irrefutable pattern: a pattern which is only matched on demand. All top-level pattern bindings are irrefutable by default, and it is useful to make irrefutable the argument patterns in interactive functions (for background discussion see [Thompson, 1990]). An example is

```
f ~(a:x) = a + f x
```

This can be described either by

```
f y = (head y) + f (tail y)
```

or by

```
f y = a + f x
      where
      (a:x) = y
```

The latter seems to be more in the spirit of the definitions than does the former.

The effect of a pattern binding can be a program error – what is the effect in the logic? If an axiom of the form

```
(a:x) = e
```

is introduced, and the expression `e` evaluates to `[]`, the effect is to give

```
(a:x) ≡ []
```

which contradicts the distinctness of the list constructors, and is indeed a logical contradiction. To safeguard against this in the logic, it is necessary to write

```
matches e == True ⇒ (a:x) ≡ e
```

where `matches` has the definition:

```
matches (a:x) = True
```

This has the effect of leaving `a` and `x` undefined when `matches e` is not `True`. As patterns are explained in a similar way, by the translation from

```
f p@q = blah
```

to

```
   f p = blah
         where
         q = p
```

5.3 Scopes

A local definition is given either by a `let`, giving a definition local to an expression, or a `where`, giving a definition local to the right hand side of a clause of a definition. The simple effect of a definition is a *conjunction* of the logical effect of the local and the global, with a restriction on the scope of the names involved. In other words, the logic will naturally inherit the scoping of Haskell.

The scope of some definitions in `where` clauses can be restricted to a subset of the right hand sides. Given the type

```
   data mo = Eenie Int | Meenie Bool
```

definitions like

```
   f :: mo -> ...
   f x | isEenie x  = ... a ...
       | isMeenie x = ... b ...
                      where
                      (Eenie a)  = x
                      (Meenie b) = x
```

are common. The scope of `a` is the first equation and of `b` the second. Thus `where` is eliminated in favour of `let`.

Similarly, modules and abstract data types provide control on the visibility of definitions. This can be reflected in the logic (for ADTs, say) by making visible outside the implementation scope only those theorems which do not refer to the underlying representation.

6 Input/Output

Input/output in Haskell can be programmed by means of streams or continuations. Stream programs are simply lazy-list manipulating functions, and the methods of verifying them are inherited from lists. An advantage of the stream approach is that it introduces explicit notations for the values on input and output, but a major disadvantage is a consequence of the *absence* of interleaving information in the functions. Much work is needed to translate stream functions into descriptions of *traces* of input/output behaviour. A trace is a sequence of input and output actions of the form

```
   [ r c , w c ]
```

which describes the action of reading `c` followed by writing `c`. Details of how to translate stream programs into traces can be found in [Thompson, 1990].

The Glasgow Haskell compiler supports input/output by a third, primitive mechanism: *monads*, [Peyton Jones and Wadler, 1993, Wadler, 1992]. Preliminary work suggests that a trace description of monadic IO is quite straightforward. An interaction of type

```
   IO a
```

will have a trace of the form

$$([rc, wc, ...], x)$$

where `[ r c , w c , ... ]` is a trace of I/O actions and `x` is a value of type `a`. The basic operations of the monad can then be described by their traces. Getting a character, `getcIO`, which is of type `IO Char` will have traces of the form

```
   ( [ r ch ] , ch )
```

where `ch` is a character. `putcIO` is a function from `Char` to `IO ()`; `putcIO ch` will have the trace

```
   ( [ w ch ] , () )
```

The combination operator, `bindIO` has type

```
   IO a -> (a -> IO b) -> IO b
```

We can describe traces of `bindIO m f` in terms of traces of `m` and `f c` (with `x` in type `a`) thus. If `(s,x)` is a trace of `m` and `(t,y)` a trace of `f x` then

```
   ( s++t , y )
```

is a trace of `bindIO m f`.

These trace descriptions can be seen as primitive, or can be proved on the basis of the implementations of the operations in [Peyton Jones and Wadler, 1993], assuming a suitable axiomatisation of the underlying C compiler! The simple operation of the `bindIO` functional is due to the data dependencies evident in the underlying implementation, for instance.

Whether this approach can scale up to tackle real problems is open, as indeed is the field of verifying interactive programs itself.

7 Other Issues

In discussing the interpretation of definitions, such as

```
   fac x | x == 0 = 1
         | x > 0  = x * fac (x-1)
```

the equational rendering

```
   x == 0 ≡ True ⇒ fac x ≡ 1
   x > 0 ≡ True  ⇒ fac x ≡ x * fac (x-1)
```

implies that `fac` is a fixed point of the definition, but not necessarily the *least* one. It is open whether this needs to be incorporated – further discussion can be found in [Thompson, 1989].

8 Conclusion

The paper addresses the design of the Haskell programming language from the point of view of giving (formal) proofs of correctness of functional programs. It is evident that Haskell share the elegance and simplicity of other lazy languages, but that certain features cause difficulties for the verifier. The ability freely to combine pattern matching, guards and local definitions causes difficulties beyond the advantage gained. Classes come in two forms: ones like the numerical classes where the overloading is conventional or mnemonic, since the operations share little but name; and the others, like the equality class, in which the operations have a common axiomatisation. It is the latter form which the verifier can work with more effectively.

I am grateful to Gareth Howells and Mark Longley for discussions about

Haskell and functional program verification. The referees made useful suggestions about both presentation and content.

References

[Hudak and Fasel, 1992] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[Hudak *et al.*, 1992] Paul Hudak, Simon Peyton Jones, and Philip Wadler (Editors). Report on the Programming Language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[Paulson, 1987] Lawrence C. Paulson. *Logic and Computation — Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[Peyton Jones and Wadler, 1993] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth Annual Symposium on Principles of Programming Languages (POPL).* ACM, 1993.

[Pitts, 1992] Andrew M. Pitts. A co-induction principle for recursively defined domains. Preprint – Computer Laboratory, University of Cambridge, 1992.

[Thompson, 1989] Simon J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1, 1989.

[Thompson, 1990] Simon J. Thompson. Interactive functional programs: a method and a formal semantics. In David A. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[Wadler, 1992] Philip Wadler. The essence of functional programming. In *Nineteenth Annual Symposium on Principles of Programming Languages (POPL).* ACM, 1992.