Programming with Variable Functions

Martin Odersky

University of South Australia
odersky@cis.unisa.edu.au

Abstract

What is a good method to specify and derive imperative programs? This paper argues that a new form of functional programming fits the bill, where *variable functions* can be updated at specified points in their domain.

Traditional algebraic specification and functional programming are a powerful pair of tools for specifying and implementing domains of discourse and operations on them. Recent work on evolving algebras has introduced the function update in algebraic specifications, and has applied it with good success in the modelling of reactive systems. We show that similar concepts allow one to derive efficient programs in a systematic way from functional specifications. The final outcome of such a derivation can be made as efficient as a traditional imperative program with pointers, but can still be reasoned about at a high level.

Variable functions can also play an important role in the structuring of large systems. They can subsume object-oriented programming languages, without incurring the latter's problems with pointer aliasing and modularity.

1 Introduction

In his paper "Hints on Programming Language Design"[Hoa73, HJ89], which was given as a keynote address at the first ACM Symposium on Principles of Programming Languages in 1973, C.A.R. Hoare made the following, quite prophetic, remark:

> "In a high-level language, the programmer is deprived of the dangerous power to update his own program while it is running. Even more valuable, he has the power to split his machine into a number of separate variables, arrays, files, etc.; when he wishes to update any of these he must quote its name explicitly on the left of the assignment, so that the identity of the part of the machine subject to change is immediately apparent; and, finally, a high-level language can guarantee that all variables are disjoint, and that updating any one of them cannot possibly have any effect on any other.
>
> ...
>
> [In contrast,] references are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover."

Hoare's prophecy is still valid today. To be true, there have been programming paradigms that side-step the problem by eliminating one of its two constituents, reference passing and mutable state. Mainstream program verification usually does not consider reference passing, and uses copying instead. On the other hand, declarative programming sidesteps the problem by having references but not considering mutable state. Without mutable state, a reference is indistinguishable from the value it refers to, so references become simply an efficient implementation technique.

More recently, refinements of functional programming languages have emerged which re-introduce mutable state without re-introducing references as an observable concept. Programming languages based on linear type disciplines [Gir87, Wad90b, LM92, Bak92, Ode92, AP94, MOTW95, CH97] allow state to be updated destructively as long as it can be verified that only one reference to the state exists. In this case, updating state is indistinguishable from creating and modifying a copy. Programming languages based on monads [Mog89, Wad90a, Mog91, Wad92, HPW92] allow one to partition the program into a functional part which uses references and an imperative part which uses mutable state.

Appeared in Proc. ACM International Conference on Functional Programming, Baltimore, 1998.

While all of these approaches are useful, they seem to be not flexible enough to completely replace references. On the contrary, with the rise of object-oriented programming, references (disguised as objects) have been elevated to principal program building blocks. So it seems we are today further than ever away from a solution to Hoare's problem.

This work tries to tackle Hoare's problem by taking seriously a simple, well known equivalence:

$$x.f \quad \equiv \quad f(x) \ .$$

Seen as an equivalence between values, it states that field selection corresponds to application of a selector function. Extending the equivalence to left-hand sides of assignments, we get something more interesting:

$$x.f := y \quad \equiv \quad f(x) := y \ .$$

Whereas the left assignment is quite conventional, the right assignment is not. In essence, we postulate the existence of a *variable function* f, which gets updated by the assignment at one point in its domain. Most programming languages know only a restricted form of a variable function over an integer interval, i.e. an array.

The treatment of arrays as variable functions goes back to the early work on axiomatic program verification [Hoa71, Dij76] and has been stated explicitly by Reynolds [Rey79]. We propose to extend this concept to variable functions over arbitrary domains. In fact, since variable functions can model mutable fields by the above equivalences, we propose to do away with mutable fields altogether and treat the left hand forms of selection and update as syntactic sugar for the right hand forms.

In a sense this gives us a formalism completely dual to object-oriented languages and impure functional languages such as Lisp or Scheme. In those languages every variable or field can be mutable, whereas every function or method is immutable. We propose instead a formalism in which all data are immutable but there can be functions that are mutable.

Is this more than just a change in viewpoint? After all, every program with pointers can be transformed by the above equivalence into a program with variable functions, so how can the new notation give us better reasoning capabilities? Variable functions are certainly no "silver bullet" which makes opaque programs clear by a simple change in notation. However, variable functions do give the programmer increased flexibility which can be used to formulate programs that are clearer and better structured than their pointer-based counterparts. The increase in flexibility comes from two sources.

First, in a pointer-based program, the reference x in $x.f$ is an element of an unstructured type. Pointers are allocated from the heap and there is no a priory relationship between two pointers except their equality or disequality. On the other hand, the domain of a variable function can have structure. This is crucial for program proofs that proceed by case distinction or induction. Many examples of such proofs are found for programs that use arrays. The same principles apply for programs with variable functions over other structured types, e.g. algebraic data types. Section 3 will present an application of such reasoning principles in a proof of refinement for two implementations of queues.

Second, in a pointer-based program the field f in $x.f$ needs to be declared together with the declaration of x's type. Sometimes, but not always, this is an appropriate program structuring. On the other hand, in a variable function application $f(x)$ the function f need not be declared at the same place as x. In may cases, this freedom allows more natural modular decompositions. Section 4 will give examples.

Variable functions have come up in a number of contexts, ranging from program logics to specification languages, to programming. Dynamic logic [Har79] makes extensive use of variable functions called arrays (but being not restricted to integer domains). Evolving algebras[Gur95], now called abstract state machines, use a combination of algebraic data types and variable functions together with transition rules for system modelling [BBD+96] as well as programming language semantics [GH93, BR94, BS94].

Variable functions have also been a part of the SETL programming language [SDDS86], where they are called maps. Maps in their full generality are more powerful than variable functions in that they support enumeration over the function's domain. Also similar is the table construct of Hermes [SBL+91].

Interestingly, the syntax of variable functions goes back to at least Algol-W [WH66], but their use in this language is restricted, so that the concept is in fact equivalent to records.

The present work explores variable functions as a link between specification and programming. It is the first to look at variable functions in the context of a polymorphically typed, higher-order language.

There are several other approaches to the semantics, derivation and verification of pointer-based programs. The axiomatic definition of Pascal [HW73] treats the whole program heap as a collection of variable functions, one per record type declared in a program. The domain of these functions is unstructured, so that this approach does not lend itself immediately to program verification without additional techniques to address pointer aliasing. Burstall [Bur72] uses a similar approach for program verification. He handles de-aliasing by the introduction of "changed" sets. Bijlsma [Bij89] and Möller [Möl93, Möl97] refine this approach by using maps or relational algebra to express the topology of pointer structures. In all these treatments, the topological information about a pointer structure is external to the program be reasoned about. By going from unstructured to structured domains of variable functions we can internalize most of this information in the program itself. This should significantly improve documentation, remove many proof obligations (in particular where a pointer structure can be modelled directly as an algebraic type) and simplify the proofs for those obligations that remain.

An alternative way to verify programs with pointers works by axiomatising a notion of program equivalence for a language with references [Mas88, MT91, ORH93, SF93]. It is not obvious how to extend these equational theories so that they can be applied in proofs of refinement.

The rest of this paper is organized as follows. Section 2 sketches an example language to express variable functions. Section 3 exemplifies how variable functions are employed in program derivation. Section 4 illustrates how variable functions help in program decomposition. Section 5 presents an operational semantics for a core language with variable functions. Section 6 concludes.

2 A Notation for Variable Functions

This section gives an overview of *Fun!*, a language that supports variable functions over algebraic types. The syntax

$$\textbf{var}\ f\ (x:S):T := E$$

declares a variable function $f : S \to T$ whose initial value is E. The initial value may depend on the parameter, x. The initializer E is optional; if none is given the variable function is initially everywhere undefined. The variable function can be accessed like a normal function with syntax $f(E)$. It can also be updated with syntax $f(E_1) := E_2$. This will update the function at the value of E_1 with the value of E_2. Subsequent accesses to $f(E_1)$ will yield the value of E_2. The value of an update statement is the value being assigned.

For convenience, we also admit single point variables

$$\textbf{var}\ v:T := E\ .$$

Such variables can be thought of being defined over the unit domain (). Every using occurrence of a single point variable is implicitly augmented with a () argument.

Examples:

```
var snowfall(month: int): float
var root: BinTree := Empty
var prev (l: LinkedList): LinkedList := Nil
var h(t: BinTree): int := 0;
```

```
fun height(t: BinTree): int = t.
  case Empty ⇒
    0
  case Branch(l, x, r) ⇒
    if h(t) = 0 then
      h(t) := max(h(left(t)), h(right(t))) + 1
    h(t)
```

The last definition is a bit more involved than the others. It introduces a lazy memo function [Hug85], height. The function computes the height of its tree parameter t from the height of t's subtrees, and updates a map h with that value. Subsequent accesses to height(t) then take the value directly from h instead of recomputing it.

Field selection $x.f$ has the same meaning as function application $f(x)$, except that the name f is resolved in the module that declares the type of x, rather than in the current module. Hence, if both f and the type of x are declared in some module M, we may write $x.f$ instead of $M.f(x)$, which would be more conventional in functional languages. That way, we achieve conciseness on a par with object-oriented programming for those programs which cluster functions with their first argument.

The function height is implemented by pattern matching on the value of its t parameter. Pattern matching is expressed by applying the anonymous function

```
{ case Empty ⇒ 0
  case Branch(l, x, r) ⇒ ... }
```

to the argument t. The function application is written in postfix syntax $x.f$.

Implementation How can variable functions be represented? There is a range of possibilities. Most straightforward would be to implement a variable function such as height as a hash table or tree. Or one could implement height as an array, to be indexed by integer fields that are implicitly allocated in BinTree nodes. Or program analysis might determine that only one copy of height can exist at any one time, in which case one might implement height by distributing it over the nodes in its domain, in fact going back to the standard mutable field implementation.

In practice one has to be careful about avoiding storage leaks. We would expect that the storage for an association from a key x to a value $f(x)$ is reclaimed when either x or f individually becomes garbage. If one used a standard hashtable or tree, there would be a reference from the table to the key x, so x could never become garbage as long as the whole table was reachable. Better storage reclamation behavior requires some cooperation from the garbage collector. For instance, one can use weak references, which are not followed by the collector, as building blocks for weak hashtables.

A Refinement The distributed implementation in terms of mutable node fields incurs the overhead that every node in the domain of the variable function has to carry a field, whether the function has been updated at that node or not. If the function is only updated at a few points in its domain this might waste a lot of space. On the other hand, the distributed implementation is clearly superior in cases where function updates are common. In general, the tradeoff between distributed and non-distributed implementation will be difficult to make automatically. It therefore seems reasonable to also give the programmer a choice between the two implementations.

We achieve this by introducing a second declaration form for variable functions, where the parameter comes first:

$$\textbf{var}\ (x : S)\ f : T := E$$

The new form is intentionally analogous to the field access syntax $x.f$ where the leading parameter section contains the `this` parameter

This form of declaration can be used only in a global context, not for a local variable in a function. Its meaning is the same as the standard variable function declaration **var** $f\ (x : S) : T := E$. However, the graph of f will now be represented as a field in every S node. Since f and S might be defined in different program files, this means that the final layout of objects can be determined only at link time.

Except for their implementation aspects, the two variable declaration forms **var** $(x : S)\ f$ and **var** $f\ (x : S)$ are completely equivalent. In particular, they can be freely mixed with the two forms of use $f(x)$ and $x.f$; it is not required to use field access syntax only with field declarations and function application syntax only with function declarations.

More details of our notation will be explained in the context of the examples in the following sections.

3 Example: Queues

This section derives systematically an efficient imperative implementation of mutable queues from a high-level functional specification. This problem is generally believed to be hard. In previous work [ORH93], we gave a proof that a monadic queue implementation satisfies algebraic laws on queues which were derived from the functional specification. Compared to this work, the present treatment is more systematic (since it uses a standard data refinement approach [Hoa72]), complete (since it directly relates to the functional specification), and concise.

As a first step, here is the definition of an abstract type for *immutable* queues.

```
module Queue
   type Queue =
      case Empty
      case Append(prev: Queue, elem: A)
   fun isEmpty(Empty) = true
      | isEmpty(Append(_, _) = false
   fun first(Append(Empty, x)) = x
      | first(Append(Append(q, x), y)
         = first(Append(q, x))

   fun rest(Append(Empty, x)) = Empty
      | rest(Append(Append(q, x), y)
         = Append(rest(Append(q, x), y)
```

This program defines a module Queue which contains a type of queues of the same name as well as operations on queues. For the time being, we let the type of queue elements be an arbitrary but fixed type, A. The Queue type is an algebraic data type with two constructors, Empty and Append.

The module also defines functions isEmpty, first and rest operating on values of Queue type. All functions operate by pattern matching on their parameter. They implement in a straightforward way the standard algebraic axioms for queues.

So far, we have a purely functional program. As a next step, let's define a module that encapsulates a single mutable queue variable. The initial definition of this module is quite simple:

```
module QueueVar
   var q: Queue := Empty;
   fun isEmpty()      =  Queue.isEmpty(q)
   fun append(x: A)   =  q := Append(q, x)
   fun front()        =  first(q)
   fun skip()         =  q := rest(q)
```

The module contains a mutable variable q of type Queue as well as operations isEmpty, append, front, and skip, which simply apply the corresponding functional operations to the encapsulated variable.

This module is fine as a specification or high-level design, but it's not a an efficient program since the front and skip operations take time linear in the length of the queue. There exist several purely functional implementations of queues with constant time cost for these operations [HM81, Bur82, Oka97]. Rather than go down that path, we will develop a refinement for queue variables that closely resembles the usual imperative algorithm.

The new algorithm keeps a reference to an entry immediately before the head of the queue, so that front can be implemented efficiently. Further, there is a mapping next that maps each queue element to its successor, so that skip can be implemented efficiently.

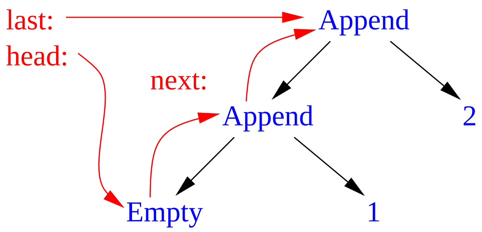

Figure 1: Relationship between `next` and `Append`

module QueueVar1

var next(q: Queue): Queue
var head: Queue := Empty
var last: Queue := Empty

fun isEmpty() = head = last

fun append(x: A) =
last := next(last) := Append(last, x)

fun front() **where not** isEmpty() =
elem(next(head))

fun skip() **where not** isEmpty() =
head := next(head)

It's worth noting that the definition of front uses function elem as a selector of the elem field in Queue. The relationship between next and Append is depicted in Figure 1.

We still have to prove that the new efficient implementation is a refinement of the original specification. This task is very hard for a conventional pointer-based implementation. The reason is that, in the absence of additional knowledge about aliases, a pointer assignment $p.x := y$ will invalidate any dereference $q.x$ as long as we can't prove somehow that p and q are either definitely the same or definitely different. This alias analysis is known to be difficult in general. We will show that variable functions over algebraic types give the problem sufficient structure to allow an easy proof.

Let's first make precise what we mean when we say that QueueVar1 is a refinement of QueueVar. We do so by defining an *abstraction function* from QueueVar1 states to QueueVar states. The function is defined as follows:

abs(h: Queue, l: Queue): Queue
= Append(... Append(Empty, x_1), ..., x_n)
where
l = Append(... Append(h, x_1), .., x_n)
$n \geq 0$

The equalities in this definition are meant to be structural, no pointer identity is assumed. We then need to show the following refinement laws.

$$\begin{aligned}
\text{isEmpty()} &= \text{isEmpty}(abs(\text{head},\text{last}))\\
\text{front()} &= \text{first}(abs(\text{head},\text{last}))\\
\text{append(x)};\ abs(\text{head},\text{last}) &= \text{Append}(abs(\text{head},\text{last}),\ \text{x})\\
\text{skip()};\ abs(\text{head},\text{last}) &= \text{rest}(abs(\text{head},\text{last})).
\end{aligned}$$

These laws show that a *simulation relation* exists between modules QueueVar1 and QueueVar. The simulation relation is depicted in Figure 2. State changes in QueueVar1 correspond one-to-one to state changes in QueueVar, after applying the abstraction mapping abs. Analogously, access functions in QueueVar1 return the same result as access functions in QueueVar.

To prove these refinement laws, we first establish two invariants for the QueueVar1 module. Say a queue q_1 is a *prefix* of a queue q_2 if there exist elements x_1, ..., x_n so that

$$q_2 = \text{Append}(\ \dots\ \text{Append}(q_1,\ x_1),\ \dots,\ x_n)$$

We write in this case $q_1 \ll q_2$. Then we always have:

$$\text{head} \ll \text{last}$$

and

$$\begin{aligned}&\forall\ \text{q,q'}:\ \text{q} \ll \text{last} \wedge \text{q} = \text{Append(q', x)}\\ &\qquad \Rightarrow\ \text{next(q')} = \text{q}\ .\end{aligned}$$

Both invariants are clearly true initially, and a simple application of predicate transformers shows that they are maintained by each operation in QueueVar1. With the invariants, it is then straightforward to prove the refinement laws. Two of the laws, namely

$$\begin{aligned}
\text{isEmpty()} &= \text{isEmpty}(abs(\text{head},\text{last}))\\
\text{append(x)};\ abs(\text{head},\text{last}) &= \text{Append}(abs(\text{head},\text{last}),\ \text{x})
\end{aligned}$$

follow immediately by expanding out the definitions on both sides. Instead of showing

$$\text{front()} = \text{first}(abs(\text{head},\text{last}))$$

directly, we prove a slightly stronger statement:

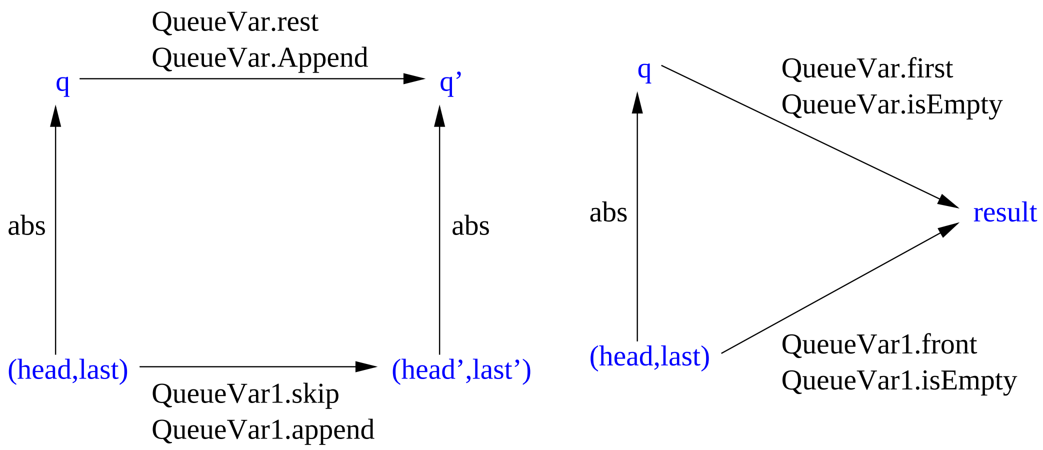

Figure 2: Simulation of `QueueVar` by `QueueVar1`

$$
\begin{aligned}
&\forall \mathsf{q}\colon \mathsf{head} \ll \mathsf{q} \wedge \mathsf{q} \ll \mathsf{last} \wedge \mathsf{q} \neq \mathsf{head} \\
&\qquad \Rightarrow \mathsf{front}() = \mathsf{first}(\mathit{abs}(\mathsf{head},\mathsf{q}))\ .
\end{aligned}
$$

The proof is by an induction on the number n of elements that are in q but not in head. The case where $n = 0$ is excluded by the precondition of front. Hence the base case is $n = 1$. This case is shown by direct computation.

$$
\begin{aligned}
& \mathsf{q} = \mathsf{Append}(\mathsf{head},\ \mathsf{x}) \\
\Rightarrow\ & \text{(by the invariant)} \\
& \mathsf{next}(\mathsf{head}) = \mathsf{q} \\
\Rightarrow\ & \text{(by the two lines above)} \\
& \mathsf{elem}(\mathsf{next}(\mathsf{head})) = \mathsf{x}\ .
\end{aligned}
$$

Hence,

$$
\begin{aligned}
& \mathsf{first}(\mathit{abs}(\mathsf{head},\mathsf{q})) \\
=\ & \mathsf{first}(\mathsf{Append}(\mathsf{Empty},\ \mathsf{x})) \\
=\ & \mathsf{x} \\
=\ & \mathsf{elem}(\mathsf{next}(\mathsf{head})) \\
=\ & \mathsf{front}()\ .
\end{aligned}
$$

For the inductive step, assume that

$$\mathsf{q} = \mathsf{Append}(\mathsf{Append}(\mathsf{q}',\ \mathsf{x}),\ \mathsf{y})\ ,$$

for some q' such that $\mathsf{head} \ll \mathsf{q}'$. Then, by definition of *abs*,

$$\mathit{abs}(\mathsf{head},\ \mathsf{q}) = \mathsf{Append}(\mathsf{Append}(\mathit{abs}(\mathsf{head},\ \mathsf{q}'),\ \mathsf{x}),\ \mathsf{y})\ ,$$

and we have:

$$
\begin{aligned}
& \mathsf{front}() \\
=\ & \text{(by the induction hypothesis)} \\
& \mathsf{first}(\mathit{abs}(\mathsf{head},\ \mathsf{Append}(\mathsf{q}',\ \mathsf{x}))) \\
=\ & \text{(by definition of } \mathit{abs}) \\
& \mathsf{first}(\mathsf{Append}(\mathit{abs}(\mathsf{head},\ \mathsf{q}'),\ \mathsf{x})) \\
=\ & \text{(by definition of } \mathit{first}) \\
& \mathsf{first}(\mathsf{Append}(\mathsf{Append}(\mathit{abs}(\mathsf{head},\ \mathsf{q}'),\ \mathsf{x}),\ \mathsf{y})) \\
=\ & \text{(by definition of } \mathit{abs}) \\
& \mathsf{first}(\mathit{abs}(\mathsf{head},\ \mathsf{Append}(\mathsf{Append}(\mathsf{q}',\ \mathsf{x}),\ \mathsf{y})))\ .
\end{aligned}
$$

The proof for the connection between skip and rest is quite analogous, and is omitted here.

We have thus shown that QueueVar1 implements the QueueVar specification. The proof relied crucially on variable functions over structured domains. If we had to deal with pointers directly, the abstraction function *abs* from QueueVar1 to QueueVar and with it the inductive proof of simulation between the two modules could not have been formulated in the way we presented it.

However, there is still something wrong with the QueueVar1 implementation: The data structure referenced from last will grow with each append, and none of the other operations will shrink its size. Hence, for programs that perform an unbounded number of append's, we will need an unbounded amount of memory.

Fortunately, this problem has a simple solution. Let's postulate that all variables in QueueVar1 and the Queue type itself are private to that module. Then the prev field of a queue, which leads from an Append node to its predecessor, is never accessed in the actual program code. We only need it for the correctness proof of QueueVar1. Hence, our program is unaffected if we do away with the prev fields in queues. We still have to make sure that all queues we create are pairwise different. This is achieved by switching from an algebraic data type with structural equality to a *generative* type with reference equality. The new definition of Queue is as follows.

```
type Queue
    case Empty
    case new Append(/*prev: Queue,*/ elem: A)
```

The keyword **new** in front of Append indicates that every application of **new** Append yields a fresh Append node, different from all Append nodes created before. The rest of the queue module is given below.

```
module QueueVar2

    var next (q: Queue): Queue
    var head: Queue := Empty
    var last: Queue := Empty

    fun isEmpty() =
        head = last

    fun append(x: A) =
        last := next(last) := new Append(/*last,*/ x)

    fun front() where not isEmpty() =
        elem(next(head))

    fun skip() where not isEmpty() =
        head := next(head)
```

The only changes compared to QueueVar1 are the generative definition of Append and, analogously, the generative constructor **new** Append(x) in the implementation of append. To point out the correspondence with QueueVar1, we left the code that was replaced as a comment. The correctness of QueueVar2 is easily established from the correctness of QueueVar1. It is sufficient to note that every application of Append in QueueVar1 will create a queue which is different from any queue created up to that point; hence, switching to a generative data type will not affect the topology of the graph spanned by head and next.

As a final step in the derivation, we can change the definition of next to enforce a field-based representation:

```
var (q: Queue) next: Queue
```

The rest of the module stays the same. We have thus systematically derived from a functional specification an imperative queue module which could have been written in this form by a competent programmer.

The final implementation is on a par with the best possible imperative implementations of queues – in fact it is probably more efficient than what most programmers would have written! The reason is that, if we translated our algorithm to a traditional record- or object-based language, the first queue element would look strange since it has an undefined elem field. Many programmers would tend to avoid this undefined field by initializing first and last to a null value. But then the append and rest operation would have to check for the special case of an empty list. These case distinctions are costly, both conceptually and in terms of run-time efficiency.

In contrast, if we regard next as a variable function over an algebraic queue data type, first and last are simply initialized to the empty queue. No special "trick" is involved. In fact, with this approach the more efficient solution is hard to avoid.

4 Use of Variable Functions in Program (De-)Composition

In the last section, we have presented a case study as an argument that variable functions are the right way to think about programs with references to mutable state. This section shows that this concept also helps in programming on a larger scale, where programs are composed from separate modules. The main benefit to be gained here is that, unlike fields of records or objects, variable functions can be declared in places unrelated to their argument type.

As a first example, consider the task of depth-first traversing an acyclic graph. The standard algorithm uses mark bits in the graph nodes to keep track of the fact that a node has already been visited. This is problematic for several reasons. First, it violates the principle of separation of concerns because when defining the graph structure we now have to anticipate the need for traversals when declaring the mark fields. Second, it makes the program more fragile since a local traversal function now uses global state. We have to worry about initializing that state, about preventing concurrent access to it, etc.

A solution with variable functions does not have these shortcomings. Let the graph be given by a type Node of nodes and a successor function from nodes to list of nodes.

```
type Node
fun succ (x: Node): List[Node]
```

Then the traversal function is formulated as follows.

```
fun traverse(root: Node, p: Node → void) =
    var visited(n: Node): boolean := false
    fun visit(n: Node) =
        if not visited(n) then
            visited(n) := true
            p(n)
            succ(n).forall(visit)
    visit(root)
```

The visited function is now declared in its logical place: the traversal function itself. On the other hand, accesses to visited would likely be slower than if visited was implemented as a field in the graph nodes themselves. We can get back the more efficient implementation by declaring visited as a global function:

```
var (n: Node) visited: Boolean := false
```

```
traverse(root: Node, p: Node → void) =
   /* as before */
```

With the new form, we have re-gained along with the efficiency of the old mark bit implementation one of its disadvantages: Mark bits are again global, so we have to worry about multiple uses. Nevertheless, we still have retained some of the benefits of the variable function approach. First, the visited function can be defined in the same module as the traversal function, which is not necessarily the module where graph nodes are defined. Second, the traverse function itself is as before; it does not need to be changed to reflect the change in implementation. This simplifies the transformation process from program designs to efficient implementations.

As another example, consider symbol tables in a compiler. Some attributes of symbols are universal; we would expect them to be present in any program that uses the symbol table. For instance, we might want to always store with a symbol its name, type, and owner. Other attributes would depend on the concrete application at hand. For instance, a backend might want to store an address with a variable or function symbol, whereas a browser might want to store a list of positions where the symbol is referenced. Some of this flexibility can be obtained by polymorphism, in either subtype or parametric form. Nevertheless such flexibility requires planning – the type of symbols has to be made instantiatable, and all program points that create symbols have to be supplied with a factory method [GHJV94] or something equivalent to generate the right kind of symbol. With variable functions, we can simply keep associations from symbols to additional data in the modules where they are maintained and used. Example:

```
module Symtab
   type Symbol =
      case Symbol ( name: String,
                    tp: Type,
                    owner: Symbol)
   ...
module Backend
   var (sym: Symbol) adr: int
   ...
module XRef
   var (sym: Symbol) uses: List[Position]
   ...
```

This use of variable functions in program composition can be seen a special case of type adaptation [Höl93], restricted to the case of instance fields. An extension to methods seems feasible, but is not part of the current paper.

5 A Calculus for Variable Functions

This section gives an operational semantics for a core language with variable functions and algebraic data types. Variable functions are higher-order, they can be applied and updated with other functions as arguments. The syntax of terms is given in Figure 3. There are three alphabets, one for pattern-bound variables x, one for **var**-bound variable functions f, and one for data constructors F. As operations we have function application $M\ N$, function update $f\ M := N$ and selection $M.C$, where C is a sequence of pattern matching clauses $P_1 \Rightarrow E_1 \parallel \ldots \parallel P_n \Rightarrow E_n \parallel$ **fail**. Pattern matching clauses are tried left to right. They always end in a **fail** case which represents a pattern matching failure. To avoid clutter we usually omit the **fail** case in program examples.

Functions are declared using **var** D **in** M where D is a set of (possibly recursive) pattern matching function definitions $f\ C$.

Function update requires an f variable in function position. Hence, the function being updated cannot be a parameter or the result of a computation. However, one can always simulate these cases by a pair of setter and getter functions. An example, which simulates a first-class reference cell, is shown below.

```
var
   cell () ⇒ 0
   setter x ⇒ cell () := x
   getter () ⇒ cell ()
in
   (setter, getter)
```

(For conciseness, we have omitted the $\parallel$ **fail** cases in pattern matches and have used tupling syntax instead of designated unary and binary tuple constructors.).

Each of the functions declared in a **var** declaration is updatable. There is no separate syntax to define immutable functions — there's no need for it, since the same effect can be achieved by encapsulation. For instance, the term

$$\textbf{var}\ a\ x \Rightarrow M\ \textbf{in}\ a$$

is operationally indistinguishable from the lambda-abstraction $\lambda x.M$. In particular, the function defined by that term cannot be updated, since updates are only allowed for f-named functions, and the name a of the function being defined is hidden in a local scope.

Figure 3 also defines a structural equivalence relation between terms that are taken to be identical. Besides α conversion we have associativity and commutativity of declarations in a **var** clause. The remaining two rules allow for scope-extrusion [MPW92] and merging/splitting of scopes. These rules are subject to the *hygiene condition* that bound variables in a term are pairwise different, and

Term Structure:

$$
\begin{array}{llcll}
\text{Terms} & M, N & ::= & x & \text{Pattern-bound variable} \\
 & & \mid & f & \text{Function} \\
 & & \mid & F\ M_1 \ldots M_n & \text{Algebraic data} \\
 & & \mid & M\ N & \text{Application} \\
 & & \mid & f\ M := N & \text{Assignment} \\
 & & \mid & M.C & \text{Selection} \\
 & & \mid & \mathbf{var}\ D\ \mathbf{in}\ M & \text{Definition} \\
\\
\text{Definitions} & D & ::= & f\ C \ \mid\ D, D & \\
\text{Matchings} & C & ::= & \mathbf{fail} \ \mid\ P \Rightarrow M \ [\!]\ C & \\
\text{Patterns} & P & ::= & x \ \mid\ f \ \mid\ F\ P_1 \ldots P_n & \\
\text{Values} & V, W & ::= & f \ \mid\ F\ V_1 \ldots V_n &
\end{array}
$$

Evaluation Contexts:

$$
\begin{array}{lcl}
E & ::= & [\,] \\
 & \mid & E\ N \ \mid\ V\ E \\
 & \mid & E.C \\
 & \mid & f\ E := N \ \mid\ f\ V := E \\
 & \mid & \mathbf{var}\ D\ \mathbf{in}\ E
\end{array}
$$

Equivalences:

α conversion for x, f

(,) in D is associative and commutative

$$
\begin{array}{lcl}
E[\mathbf{var}\ D\ \mathbf{in}\ M] & \equiv & \mathbf{var}\ D\ \mathbf{in}\ E[M] \\
\mathbf{var}\ D\ \mathbf{in}\ \mathbf{var}\ D'\ \mathbf{in}\ M & \equiv & \mathbf{var} D, D'\ \mathbf{in}\ M
\end{array}
$$

All equivalences apply to hygienic terms only.

Figure 3: Term structure of core Fun!

Application and Update

$$
\begin{array}{rcl}
\mathbf{var}\ D, fC\ \mathbf{in}\ E[fV] & \rightarrow & \mathbf{var}\ D, fC\ \mathbf{in}\ E[V.C] \\
\mathbf{var}\ D, fC\ \mathbf{in}\ E[fV := W] & \rightarrow & \mathbf{var}\ D, f(V \Rightarrow W\ [\!]\ C)\ \mathbf{in}\ E[W]
\end{array}
$$

Pattern Matching

$$
\begin{array}{rcll}
V\ .\ (x \Rightarrow M\ [\!]\ C) & \rightarrow & M[x := V] & \\
f\ .\ (f \Rightarrow M\ [\!]\ C) & \rightarrow & M & \\
(F\ V_1 \ldots V_n)\ .\ (F\ P_1 \ldots P_n \Rightarrow M\ [\!]\ C) & \rightarrow & \quad V_1\ .\ (P_1 \Rightarrow & \\
 & & \quad\quad \ldots & \\
 & & \quad\quad\quad V_n\ .\ (P_n \Rightarrow M & \\
 & & \quad\quad\quad\quad [\!]\ \mathbf{fail} \Rightarrow (F\ V_1 \ldots V_n).C) & \\
 & & \quad\quad\quad \ldots & \\
 & & \quad\quad\quad [\!]\ \mathbf{fail} \Rightarrow (F\ V_1 \ldots V_n).C) & \\
V\ .\ (P \Rightarrow M\ [\!]\ C) & \rightarrow & V\ .\ C & \text{in all other cases.}
\end{array}
$$

Figure 4: Reductions in core Fun!

---

are also different from free variables. We require that all terms and parts of terms we write satisfy that condition.

Figure 4 shows core Fun!'s reductions. The first two reductions in the figure have as their respective redex a function application and a function update. Application and update can only occur in hole position of an evaluation context relative to their function definition and their operands have to be values. Similar rules for single-point variables were put forward in [CF91].

Note that the two reduction rules for function application and function update are symmetrical. Function application transfers a pattern matching case clause from the environment where the function is defined into the evaluation context. Function update takes a binding $f V := W$, and prefixes the corresponding pattern matching clause $V \Rightarrow W$ to f's definition.

Crucial for this technique is the fact that in core Fun! every value is a pattern. In particular function names f are both values and patterns, which means that functions can be tested for equality. Here's an example of a core Fun! program fragment which declares together with a function g an equality test for g.

var
  g ...
  eq_g x ⇒ x. { g ⇒ true ⟦ y ⇒ false }

It's important to note that such an equality test is intensional; it will only yield true if the value of its operand x is the function name g. One might be concerned that unrestricted intensional equality tests between functions would invalidate too many program equivalences. But note that it is always possible to recover these program equivalences in a type system, by preventing function types in patterns. In delegating the issue of function equality to a type system we keep the basic calculus both simple and flexible.

Let $\rightarrow$ be the smallest relation that contains the reduction rules and is closed under arbitrary context formation. Let $\twoheadrightarrow$ denote the transitive closure of $\rightarrow$. Then the following propositions can be shown by arguments analogous to the ones in [CF91].

Proposition 5.1 $\rightarrow$ is confluent: If $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ then there exists a term M_3 such that $M_1 \twoheadrightarrow M_3$ and $M_2 \twoheadrightarrow M_3$.

Let $\mapsto$ be the smallest relation that contains the reduction rules and is closed under evaluation context formation (i.e. if $M \rightarrow N$ then also $E[M] \mapsto E[N]$). Then we have:

Proposition 5.2 $\mapsto$ is deterministic: If $M \mapsto M_1$ and $M \mapsto M_2$ then $M_1 \equiv M_2$.

Proposition 5.3 $\mapsto$ is a standard evaluation relation for $\rightarrow$: If $M \twoheadrightarrow V$ then $M \mapsto\!\!\!\!\rightarrow V$.

6 Conclusion

This paper has motivated and explained variable functions, and has emphasized their role in program verification. It should be seen as the first piece of a much larger puzzle, which aims to turn variable functions into a viable programming method.

If implemented, such a method should be beneficial in several areas. This paper has already presented applications to program reasoning and decomposition. But the flexibility afforded by variable functions promises to also be very useful in the task of gluing together together independently developed components. Hölzle [Höl93] has argued that the traditional object-oriented structuring method provides insufficient support for component composition. Variable functions promise to provide a better framework for this task, and their implementation fits well with techniques for binary component adaptation [KH98].

Acknowledgements. This work was motivated by Egon Börger's talks on evolving algebras given in Karlsruhe in 1994 and 1996, and by John Reynold's POPL keynote address in which he brought back into memory Tony Hoare's statement which is cited in the first part of the paper. Thanks also to Gary Leavens, John Maraist, David Stoutamire, Philip Wadler and the members and guests of IFIP working group 2.8 for their suggestions and constructive criticism.

References

[AP94] P. Achten and R. Plasmeijer. Towards distributed interactive programs in the functional programming language Clean. In J. Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Funtional Languages*, pages 28.1–28.16. University of East Anglia, Norwich, UK, 1994.

[Bak92] Henry G. Baker. Lively linear Lisp—'look ma, no garbage!'. *ACM SIGPLAN Notices*, 27(8):89–98, August 1992.

[BBD+96] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.

[Bij89] A. Bijslma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–205, September 1989.

[BR94] Egon D. Börger and Dean Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1994.

[BS94] Egon D. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP(R) programs. In Egon D. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[Bur72] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[Bur82] Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14:205–206, 1982.

[CF91] Erik Crank and Matthias Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 233–244, January 1991.

[CH97] Chih-Ping Chen and Paul Hudak. Rolling your own mutable adt – a connection between linear types and monads. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 54–67, January 1997.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, Springer LNCS 702, pages 274–309, 1993.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Har79] David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[HJ89] C. A. R. Hoare and Cliff B. Jones. *Essays in Computing Science*. Prentice-Hall International, London, 1989.

[HM81] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13:50–53, 1981.

[Hoa71] C.A.R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, 1971.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Stanford Artificial Intelligence Laboratory Memo AIM–224 or STAN–CS–73–403, Stanford University, Stanford, California, December 1973.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Object-Oriented Programming 7th European Conference ECOOP '93 Kaiserslautern, Germany, Proceedings*, volume 707 of *Springer Verlag, Lecture Notes in Computer Science*, pages 36–56, 1993.

[HPW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, March 1992.

[Hug85] John Hughes. Lazy memo-functions. In *Proceedings, Functional Programming Languages and Computer Architecture, Nancy, France*, pages 129–146. Springer-Verlag, September 1985. Lecture Notes in Computer Science 201.

[HW73] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proc. European Conference on Object-Oriented Programming*, Springer Lecture Notes in Computer Science, July 1998.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 235–246, Los Alamitos, California, June 1992. IEEE Computer Society Press.

[Mas88] I. A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10:177–210, 1988.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 1989 IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Möl93] Bernhard Möller. Towards pointer algebra. *Science of Computer Programming*, 21, 57–90 1993.

[Möl97] Bernhard Möller. Calculating with pointer structures. In *Proc. IFIP TC2/WG2.1 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, 1997.

[MOTW95] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. *Electronic Notes in Theoretical Computer Science*, 1(1), 1995.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–77, 1992.

[MT91] Ian Mason and Carolyn Talcott. Equivalence in functional languages with side effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[Ode92] Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

[Oka97] Chris Okasaki. Catenable double-ended queues. In *Proc. International Conference on Functional Programming*, pages 66–74. ACM, June 1997.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[Rey79] John C. Reynolds. Reasoning about arrays. *Communications of the ACM*, 22:290–299, 1979.

[SBL+91] R. Strom, D. Bacon, A. Lowry, A. Goldberg, D. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, February 1991. ISBN 0-13-389537-8.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[SF93] Amr Sabry and John Field. Reasoning about explicit and implicit representations of state. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 17–30, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[Wad90a] Philip Wadler. Comprehending monads. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 61–78, June 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Proc.19th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[WH66] Niklaus Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, 1966.