FUNCTIONAL PROGRAMMING WITH SIDE-EFFECTS

Mark B. JOSEPHS

Oxford University Programming Research Group, Oxford OX1 3QD, United Kingdom

Communicated by M. Sintzoff
Received June 1985
Revised January 1986

Abstract. Functional and logic programming languages are combined into a new applicative language. The ultimate aim is the development of more efficient programs than would otherwise be possible. This paper introduces the key idea and gives examples of its use to solve various programming problems. On implementations that support parallel evaluation of expressions, further interesting possibilities arise.

1. Introduction

In this paper a proposal is put forward for a new programming feature in an otherwise purely functional language. Unlike a conventional functional language, an applicative expression may contain occurrences of free variables. During the process of evaluating such expressions, it is possible for these variables to become bound to further expressions.

Consider, for example, the expression

\[ \text{cons} \, \text{assign} \, x = 1 + 2; \, 7 \, x \]

in which \( x \) is a free variable. Now it is our intention that evaluation of the tail, \( x \), of this expression should 'suspend' until \( x \) gets instantiated, i.e. bound to some expression. Thus, if the tail alone is demanded, we have deadlock. If instead it is the head of the expression that is to be evaluated, program execution will involve \( x \) being bound to \( 1 + 2 \) (the side-effect), while the head reduces to 7. Note that \( 1 + 2 \) only gets evaluated if both the head and tail of the expression are demanded, in which case the expression reduces to

\[ \text{cons} \, 7 \, 3 \]

So, we hope to provide the functional programmer with a new tool and intend to demonstrate in this paper that it is both useful and easy to use. We assume that our functional language is implemented on a graph reduction machine and note that the new feature can be accommodated without modifying the machine. (This possibility of binding variables by graph reduction was recognized in [4], viz. "The
logic based programming languages are supported by a facility that permits a packet to be treated as a variable by allowing a reduction to have the side-effect of assigning new contents to any argument packet.") The appendix to this paper shows that programming with side-effects is made safe by systematic program transformation for introducing side-effects.

One way of viewing the proposed extension is that Prolog's logical variables have been made available for use, in a restricted manner, in functional programs. Unlike Prolog, however, we do not allow backtracking.

For example, in Prolog if we want to express the fact that each $y_i$, $1 \leq i \leq n$, is functional in $x_1, \ldots, x_m$, we can form a suitably defined relation of the $m+n$ arguments $x_1, \ldots, x_m, y_1, \ldots, y_n$. In a functional language we would have to define separate functions for each $y_i$ or, perhaps, define a function of $x_1, \ldots, x_m$ that evaluated to the tuple $[y_1, \ldots, y_n]$. In our extended language we can choose one of the $y_i$'s, say $y_k$, and define a function $f$ of $x_1, \ldots, x_m, y_1, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n$ that evaluates to $y_k$. The other $y_i$'s can be determined by evaluating the expressions that are assigned to them as a result of $f$ being applied to its arguments.

Our extension also provides a way of synchronizing the parallel execution of functional programs:

In a purely functional language the only way to represent configurations of processes of the 'producer/consumer' variety is to apply the consumer (as a function) to the producer (an expression that reduces to the value being communicated). This gives a 'tight coupling' of the producer and consumer processes. The ability to introduce variables to act as channels between processes (expressions) [3] is also present in our language; a variable is assigned to during the reduction of one of the expressions, so implementing a loosely-coupled network.

2. The syntax of the extended language

In this section we give an informal description of the syntax of a functional language that incorporates the extension.

As in KRC [8], a program will consist of a set of function definitions and the expression that is to be evaluated. Function definitions can be viewed as recursion equations or rewrite rules, and take the form

$$\text{(function name)}(\text{pattern}^*) = \text{(expression)}, \text{(guard)}$$

Here, a pattern can be formed from variables and basic values (e.g. numbers) by means of certain constructors (e.g. cons, fork). Any variables appearing in patterns have the right-hand side of the equation as their scope. The guard (which can be omitted) is simply an expression, but should evaluate to a boolean. In addition to these user-declared functions, we will assume the existence of some primitive functions, e.g. $\text{add}$. 


The category of expressions is defined to include variables, basic values, function names and function applications, as well as the forms

\[ \langle \text{side-effect} \rangle ; \langle \text{expression} \rangle \]

\[ \langle \text{declaration} \rangle ; \langle \text{expression} \rangle \]

where

\[ \langle \text{side-effect} \rangle ::= \text{assign} \ (\text{variable}) = (\text{expression}) \]

\[ \langle \text{declaration} \rangle ::= \text{var} \ (\text{variable}) \]

Note that for \((\text{var} \ x; E)\), the scope of \(x\) is \(E\). Of course, variables can only appear within their scope; so for example,

\[ \text{foo} \ x \ (a : y) = \text{var} \ z; \text{assign} \ y = \text{tail} \ z; f x z \]

is a well-formed definition provided \(f\) is the name of some function.

KRC conventions that have been adopted include (i) ':' and '++' as infix forms of \textit{cons} and \textit{append}, respectively, and (ii) a notation based on set abstraction, so that, for example, \(\{b \leftarrow x; a < b\}\) denotes the list of numbers drawn from \(x\) that are greater than \(a\).

3. Semantics

The language is based on lazy evaluation, as is the case for many functional languages. Furthermore, although its \texttt{var} and \texttt{assign} constructs have their counterparts in imperative languages, e.g. Pascal, there are some subtle differences between \texttt{assign} and a standard assignment statement. We can give an informal semantics to this extension as follows.

A. Evaluation of \(\text{var} \ x; E\)
   1. A new location in the store is allocated to \(x\) and marked as unset. (We call \(x\) a free variable.)
   2. \(E\) is evaluated.

B. Evaluation of \(\text{assign} \ x = E_1; E_2\)
   1. If \(x\) is not free, an error is reported for improper assignment.
   2. \(E_1\) is stored in unevaluated form at \(x\), i.e. \(x\) is bound to \(E_1\).
   3. \(E_2\) is evaluated.
   
   Thus, \texttt{assign} is a lazy single-assignment construct.

C. Evaluation of a free variable is suspended until an expression is assigned to it.

Notes. (1)

\[ (\text{var} \ x; \text{var} \ y; E) = (\text{var} \ y; \text{var} \ x; E) \]
since distinct new locations are allocated to \( x \) and \( y \) in each case; we write such expressions as simply \((\text{var } x, y; E)\).

(2) \((\text{assign } x = E_1; \text{assign } y = E_2; E) = (\text{assign } y = E_2; \text{assign } x = E_1; E)\)
since the expressions \( E_1 \) and \( E_2 \) are stored unevaluated. Again we adopt the abbreviated notation \((\text{assign } x = E_1, y = E_2; E)\).

(3) Some functional languages have a construct such as letrec for making local recursive definitions. Then

\[(\text{var } x; \text{assign } x = E'; E) = (\text{letrec } x = E' \text{ in } E)\]
since in both cases all occurrences of \( x \) in \( E \) and \( E' \) refer to \( E' \).

(4) Order of evaluation of expressions is important, e.g. addition only remains a commutative operator if it evaluates its operands in parallel! (Suppose instead that it evaluates its left operand before its right; then

\[\text{var } x; ((\text{assign } x = 3; 2) + x)\]

\[= 2 + 3\]

\[= 5\]

whereas, evaluation of \(\text{var } x; (x + (\text{assign } x = 3; 2))\) deadlocks.) We shall try to take the order of evaluation into account so as to improve the efficiency of programs.

4. Examples of programming with side-effects

The first two problems are taken from [2]. Bird uses transformation techniques to develop efficient programs from inefficient programs that act as specifications. Tupling is used to improve efficiency by avoiding repeated traversal of a data structure, but as a result the programs lose their clarity. We claim that, given some familiarity with our extended language, our solutions to the problems are no less efficient and not too difficult to develop.

For all problems for which a formal specification, i.e. a functional program, has been given, the reader is referred to the appendix, where the solutions are systematically derived by means of transformational programming.

Example 1. We wish to change a given tree into a second tree identical in shape to the first. However, each tip value should be replaced by the minimum tip value of the tree.

\[
data \text{ tree } = \text{tip } \text{int } | \text{fork } \text{tree } \text{tree} \\
\text{transform } t = \text{replace } t \ (\text{tmin } t) \\
\text{replace } (\text{tip } n) \ m = \text{tip } m \\
\text{replace } (\text{fork } L \ R) \ m = \text{fork } (\text{replace } L \ m) \ (\text{replace } R \ m) \\
\text{tmin } (\text{tip } n) = n \\
\text{tmin } (\text{fork } L \ R) = \text{min } (\text{tmin } L) \ (\text{tmin } R)\]
**Method.** We introduce a local variable \( v \) to hold the minimum tip value, \( t_{\text{min}} t \), for a given tree \( t \). Thus, for \( t = \text{tip} n \), we wish to assign \( v = n \), and for \( t = \text{fork} L R \), we wish to assign \( v = \min y z \), where we have introduced \( y \) and \( z \) to stand for \( t_{\text{min}} L \) and \( t_{\text{min}} R \), respectively.

Suppose also that \( m \) stands for the minimum tip value of the tree to be transformed. Of course, for this tree \( v = m \). We only have to realize one more thing: namely, a tip is replaced by a tip, and a fork by a fork, and we are ready to formulate our solution.

**Solution 1.**

\[
\text{transform } t = \text{var } m; \ \text{replace } t \ m \ m \\
\text{replace } (\text{tip } n) m \ v = \text{assign } v = n; \ \text{tip } m \\
\text{replace } (\text{fork } L R) m \ v = \text{var } y, z; \ \text{assign } v = \min y z; \\
\text{fork } (\text{replace } L m y) \ (\text{replace } R m z)
\]

A logical reading can be given to this program as follows:

\[
\forall x, t. \ x = \text{transform } t \iff \exists m. \ x = \text{replace } t \ m \ m \\
\forall x, n, m, v. \ x = \text{replace } (\text{tip } n) m \ v \iff v = n \land x = \text{tip } m \\
\forall x, L, R, m, v. \ x = \text{replace } (\text{fork } L R) m \ v \\
\iff \exists y, z. \ v = \min y z \land x = \text{fork } (\text{replace } L m y) \ (\text{replace } R m z)
\]

From this we can infer that if \( \text{transform} \) is applied to a given tree and evaluates to some new tree, then this resulting tree will indeed meet our specification.

What this logical reading does not tell us is how much, if any, of the transformed tree can be computed. For this operational semantics of the program, an understanding of demand-driven evaluation is required.

Returning to our solution, we can deduce that all the forks and tips can be computed in response to demands. However, a demand for a tip value will be suspended until all the tips have been computed.

This means that our program is unsuitable—it would deadlock—if, say, we wanted to determine the minimum tip value by sending demands only along some selected branch of the tree. For example, \( \text{findval} (\text{transform} (\text{fork} (\text{tip} 2) (\text{tip} 1))) \) will deadlock rather than evaluate to 1, for \( \text{findval} \) defined as

\[
\text{findval} (\text{tip } n) = n \\
\text{findval} (\text{fork } L R) = \text{findval } L
\]

Thus, our program has a different operational behaviour from that given by Bird: his program is deadlock-free.

However, a minor modification to our program will enable it to construct the (complete) transformed tree in response to an initial demand. We do this by
annotating the definition of \texttt{replace} with the \texttt{val} combinator \cite{Hughes93} (the use of which makes functions \textit{strict}), so that

\[
\text{replace} (\text{fork} \, L \, R) \, m \, v = \text{var} \, y, \, z; \text{assign} \, v = \text{min} \, y \, z; \\
\text{val} (\text{val} \, \text{fork} \, (\text{replace} \, L \, m \, y)) \, (\text{replace} \, R \, m \, z)
\]

Note that for \((\text{val} \, f \, x)\), \(x\) gets evaluated before \(f\) is (evaluated and) applied to it: this corresponds to call-by-value. The simple expression \((f \, x)\) gives call-by-need, i.e. \(x\) gets evaluated only when first required. So, by annotating the program with \texttt{val}, the minimum tip value is computed in response to a demand for any tip's new value. Our program is no less efficient than Bird's and is now deadlock-free.

On implementations that support parallelism, it would be possible to use Hughes' \texttt{par} combinator \cite{Hughes93} in place of \texttt{val}: evaluation of \((\text{par} \, f \, x)\) involves the concurrent reduction of \(x\) and \((f \, x)\), its value being that of \((f \, x)\).

\textbf{Example 2.} This tree transformation problem is similar to Example 1. Again the transformed tree is to have the same shape as the original tree, but this time the original tip values must be sorted into increasing order and then allocated to the new tips from left to right.

\[
\text{transform} \, t = \text{replace} \, t \, (\text{sort} \, (\text{tips} \, t)) \\
\text{replace} \, (\text{tip} \, n) \, u = \text{tip} \, (\text{head} \, u) \\
\text{replace} \, (\text{fork} \, L \, R) \, u = \text{fork} \, (\text{replace} \, L \, u) \, (\text{replace} \, R \, (\text{drop} \, (\text{size} \, L) \, u)) \\
\text{tips} \, (\text{tip} \, n) = [n] \\
\text{tips} \, (\text{fork} \, L \, R) = \text{tips} \, L \, ++ \, \text{tips} \, R \\
\text{size} \, (\text{tip} \, n) = 1 \\
\text{size} \, (\text{fork} \, L \, R) = \text{size} \, L \, + \, \text{size} \, R
\]

(Note that this specification is slightly simpler than that given by Bird.)

\textbf{Method.} Suppose some subtree is to be replaced. It will suffice to have access to that part of the sorted list of tip values remaining after values for all tips on subtrees-to-the-left have been removed. We store this sublist in the variable \(u\). Having allocated the initial values on \(u\) to the tips in the subtree, the rest can be passed on via some shared variable \(w\), say. Furthermore, a list \(v\) of the tip values in the subtree can be produced.

So, at the root of the tree, \(v\) is a list of all the tip values in the tree and \(u\) is a sorted version of \(v\). (Note also that \(w\) will be instantiated to \([\ ]\) during execution.)

\textbf{Solution 2.}

\[
\text{transform} \, t = \text{var} \, v, \, w; \, \text{replace} \, t \, (\text{sort} \, v) \, v \, w \\
\text{replace} \, (\text{tip} \, n) \, u \, v \, w = \text{assign} \, w = \text{tail} \, u, \, v = [n]; \, \text{tip} \, (\text{head} \, u) \\
\text{replace} \, (\text{fork} \, L \, R) \, u \, v \, w = \text{var} \, x, \, y, \, z; \, \text{assign} \, v = x++y; \\
\text{fork} \, (\text{replace} \, L \, u \, x \, z) \, (\text{replace} \, R \, z \, y \, w)
\]

for some suitably defined function \texttt{sort}. Since \texttt{sort} is strict, an attempt to determine a new tip value will deadlock if all the tips do not eventually get demanded. As in Example 1, this can be avoided by using \texttt{val} or \texttt{par} (in an identical way).
We note that the list of tip values is created by appending sublists together. It should be possible to formulate a more efficient solution that avoids the use of ++. We revise our method as follows.

**Alternative method.** We would like an expression-graph such as

![Expression-graph 1](image1)

...to reduce to

![Expression-graph 2](image2)

A program that achieved this reduction with minimal overheads might be regarded as an 'optimal' solution to the problem. We devise such a program by modifying Solution 2.
An extra argument $r$ is used by $\texttt{replace}$ to keep a list of tip values on subtrees-to-the-right. $v$ is now used to store, for a given subtree, the list of its tip values appended on to $r$. Thus, at the root of the tree to be transformed, $r=[\ ]$ and $v$ is a list of all tip values (as before).

Solution 2(a).

$$\texttt{transform } t = \texttt{var } v, w; \texttt{replace } t \texttt{(sort } v) v w \texttt{[ ]}$$

$$\texttt{replace } (\texttt{tip } n) u v w r = \texttt{assign } w = \texttt{tail } u, v = n : r; \texttt{tip } (\texttt{head } u)$$

$$\texttt{replace } (\texttt{fork } L R) u v w r$$

$$= \texttt{var } y, z; \texttt{fork } (\texttt{replace } L u v z y) \texttt{(replace } R z y w r)$$

In Examples 1 and 2 we have used algorithms that would probably be favoured by a Prolog programmer. The cost of our efficient graph reduction mechanism is unexpected deadlock for the naive programmer.

Example 3. No set of examples of functional programs would be complete without inclusion of $\texttt{quicksort}$.

$$\texttt{quicksort } [\ ]=[\ ]$$

$$\texttt{quicksort } (a : x) = \texttt{quicksort } \{ b \leftarrow x; b < a \} ++ a : \texttt{quicksort } \{ b \leftarrow x; b \geq a \}$$

So, here is an efficient (lazy) version. It is also deadlock-free.

Solution 3.

$$\texttt{quicksort } [\ ]=[\ ]$$

$$\texttt{quicksort } (a : x) = \texttt{var } \texttt{rest}; \texttt{quicksort } (\texttt{partition } a x \texttt{rest}) ++$$

$$a : \texttt{quicksort } \texttt{rest}$$

$$\texttt{partition } a [\ ] \texttt{rest} = \texttt{assign } \texttt{rest} = [\ ]; [\ ]$$

$$\texttt{partition } a (b : x) \texttt{rest} = b : \texttt{partition } a x \texttt{rest}, b < a$$

$$= \texttt{var } \texttt{rest'}; \texttt{assign } \texttt{rest} = b : \texttt{rest'}; \texttt{partition } a x \texttt{rest'}$$

Example 4. In [6] an analysis is presented of a function $\texttt{split}$ which takes a list $l$ of characters and returns a pair comprising the first line of characters and the rest of the list. If the list $l$ is built up lazily, in many circumstances its members can be garbage collected soon after they have been produced. For example, the split may be performed just so as to determine the first character of the first and second lines of $l$. However, Hughes has shown that there are situations in which a ‘sequential evaluation’ strategy makes it impossible to obtain an expected constant space solution, no matter how we define the function $\texttt{split}$. Therefore he devises new primitives—$\texttt{par}$ and $\texttt{synch}$—with which he annotates $\texttt{split}$ in order to allow for the possibility of execution in constant space.

We now present a version of $\texttt{split}$ that can be run in constant space even though only sequential evaluation is required.
Solution 4.

\[\text{split } (nl : x) \ y = \text{assign } y = x; [ ]\]
\[\text{split } (ch : x) \ y = ch : \text{split } x y\]

Here, \(\text{split } l \ r\), where \(r\) is a free variable, will evaluate lazily to the first line of \(l\). \(r\) then becomes bound to the remainder of \(l\).

Hughes gives a program that uses \(\text{split}\) in order to compute the length of the first line of a list and the length of the remainder of the list. We might write this as

\[\text{program } l = \text{var } \text{rest}; \text{val cons } (\text{length } (\text{split } l \text{ rest})) [\text{length } \text{rest}]\]

Now the evaluation of either \(\text{length}\) (or both) requires only constant space, given a suitable garbage collector and assuming we use the space efficient definition of \(\text{length}\) formulated by Hughes.

Why have we used \(\text{val}\) in the definition of \(\text{program } l\)? This is because \((\text{length } \text{rest})\) can never be determined without parsing the first line of \(l\). This might have been inadvertently attempted, resulting in deadlock, had we written

\[\text{program } l = \text{var } \text{rest}; [\text{length } (\text{split } l \text{ rest}), \text{length } \text{rest}]\]

We mention here that Hughes' \(\text{synch}\) no longer has to be treated as a primitive. He gives the following description of it:

"\(\text{synch } e = [e, e]\)"

However, the two copies of \(e\) which are returned are actually different: call them \(e_1\) and \(e_2\). No demand is propagated from \(e_1\) or \(e_2\) to \(e\) until both have been demanded."

The following definition of \(\text{synch}\) achieves the same effect:

\[\text{synch } e = \text{var } e_1, e_2; [\text{assign } e_2 = e; e_1, \text{assign } e_1 = e; e_2]\]

Example 5. The Fibonacci function can be defined as follows:

\[\text{fib } 0 = 1\]
\[\text{fib } 1 = 1\]
\[\text{fib } n = \text{fib } (n - 1) + \text{fib } (n - 2)\]

Treating the above as a (KRC) program rather than simply as a specification, it is obviously a very inefficient way to determine the \(n\)th Fibonacci number: it gives rise to many repeated computations.

Method. The program that we are aiming at will have some way of 'remembering' the value of \(\text{fib } (n - 1)\) when it goes about evaluating \(\text{fib } n\). This can be achieved
(using a recursive WHERE construct) by tupling, as follows:

\[
\begin{align*}
\text{fib} \ 0 &= 1 \\
\text{fib} \ n &= \text{fst} (\text{fibpair} \ n) \\
\text{fibpair} \ 1 &= [1, 1] \\
\text{fibpair} \ n &= [x + y, x]
\end{align*}
\]

WHERE \([x, y] = \text{fibpair} \ (n - 1)\)

So, \(\text{fibpair} \ n = [\text{fib} \ n, \text{fib} \ (n - 1)]\), for \(n > 0\).

The above solution necessitates the construction and subsequent destruction of tuples. We believe this to be a source of inefficiency, and consider the following solution (using side-effects) to be an improvement.

**Solution 5.**

\[
\begin{align*}
\text{fib} \ 0 &= 1 \\
\text{fib} \ n &= \text{var} \ x; \ \text{fib'} \ n \ x \\
\text{fib'} \ 1 \ x &= \text{assign} \ x = 1; \ 1 \\
\text{fib'} \ n \ x &= \text{var} \ y; \ \text{assign} \ x = \text{fib'} \ (n - 1) \ y; \ x + y
\end{align*}
\]

Here, \(z = \text{fib'} \ n \ x \leftrightarrow z = \text{fib} \ n \wedge x = \text{fib} \ (n - 1)\), for \(n > 0\).

**Example 6.** Our final example is taken from [7]. Shapiro uses the example to illustrate the suitability of Concurrent Prolog for object-oriented programming. He presents a new programming paradigm called **incomplete messages**: our enhanced functional language supports this technique.

Our version of the problem can be described as follows:

We wish to define a function \textit{interpret} that consumes a stream of commands and produces a stream of values. The commands \textit{clear}, \textit{up} and \textit{down} are intended to effect the internal state of a ‘counter’. The command \textit{show} should result in the output of the current value of the counter’s state.

**Method.** We decompose interpret into two objects, \textit{use-counter} and \textit{counter}. The objects communicate via the shared variable \textit{chan}. Thus, we are able to resolve separately the problems of handling input/output and of updating the counter.

**Solution 6.**

\[
\begin{align*}
\text{interpret} \ \text{cmds} &= \text{var} \ \text{chan}; \\
\text{counter} \ chan \ 0 &\ // \ \text{use-counter} \ \text{cmds} \ \text{chan} \\
\text{use-counter} \ (\text{show} : \ \text{rest}) \ \text{chan} &= \text{var} \ x, \ \text{chan}'; \ \text{assign} \ \text{chan} = \text{show} : x : \ \text{chan}'; \\
\text{val} \ \text{cons} \ x \ \text{(use-counter} \ \text{rest} \ \text{chan}') \\
\text{use-counter} \ (\text{cmd} : \ \text{rest}) \ \text{chan} &= \text{var} \ \text{chan}'; \ \text{assign} \ \text{chan} = \text{cmd} : \ \text{chan}'; \\
\text{use-counter} \ \text{rest} \ \text{chan}'
\end{align*}
\]
use-counter [ ] chan = assign chan = [ ]; [ ]
counter (clear : rest) state = counter rest 0
counter (up : rest) state = val (counter rest) (state + 1)
counter (down : rest) state = val (counter rest) (state - 1)
counter (show : x : rest) state = assign x = state; counter rest state
counter [ ] state = true

Our combinator // (subordination) has its counterpart in Hoare's CSP [5]. It is intended that $E_1 // E_2$ reduces to the value of $E_2$, where $E_1$ and $E_2$ get evaluated in parallel [6]. We assume here that $E_1$ cannot be garbage collected while it is being evaluated.

Note that when use-counter sends the command show to the counter, a new variable, $x$, is also communicated: this is an example of an incomplete message. On receiving this, the counter will instantiate $x$ with the current value of its state. Furthermore, we have used val (in the show clause of use-counter) to delay the production of the output stream until this instantiation has occurred.

5. Conclusion

We have extended a purely functional language and so made a new style of programming possible, which we have dubbed 'functional programming with side-effects'.

Our language differs from other functional/logic hybrids, e.g. FPL [1], in that execution is solely by demand-driven graph reduction. However, our decision not to use unification as the evaluation mechanism has resulted in some programs deadlocking, which may prove undesirable.

On the plus side, a graph reduction based implementation of a functional language should be able to accommodate our new feature without requiring extensive modification.

Our extension enables some efficient programs to be developed that would otherwise be unavailable to the functional programmer. (We hope this has been well illustrated by our worked examples.) Furthermore, Hughes' ideas about synchronization appear to have been subsumed. We have also shown that our enhanced language may prove suitable for object-oriented programming.

Postscript. A functional programming language with side-effects has now been implemented and early results appear promising. The necessary modifications (to Wadler's MODULA-2 interpreter for his functional language Orwell [9]) were indeed trivial. The improved performance predicted in Example 5 was quite marked: the side-effects solution, compared with the tupling solution, took 20% fewer reductions and claimed 30% fewer locations in the store.
Acknowledgment

I wish to thank Tony Hoare, Richard Bird and Phil Wadler for their helpful and encouraging comments on earlier drafts of this paper. I am also grateful to the referees for suggesting improvements. I acknowledge the financial support of the British Science and Engineering Research Council.

Appendix. Transformational programming from a purely functional language to a functional language with side-effects

In this appendix we rework many of the examples from the main paper as transformation problems. We use the same fold/unfold method as Burstall/Darlington with two new rules:

introduction of suitably assigned variables . . . AV

and

move-in of assignments . . . MI

Here, AV is similar to WHERE abstraction. We can transform an expression E into

(var x; assign x = E'; E)

for some expression E' and variable x, provided E was not already in the scope of x; just as we might transform E into (letrec x = E' in E).

Note also that MI only preserves weak equivalence, i.e. it may introduce deadlock. However, if evaluation of an expression forces the evaluation of a subexpression, then move-in to this subexpression maintains strong equivalence. There are three general cases:

- move-in from assign x = E; (E_1; E_2) to
  (i) assign x = E; E_1; E_2
  (ii) E_1 (assign x = E; E_2)
- move-in from assign x = E, y = E_1; E_2 to
  (iii) assign y = (assign x = E; E_1); E_2.

Because of normal order reduction (i) cannot result in deadlock.

It is probably also worth mentioning that the properties of val and par ensure that the following strong equivalences hold:

(assign x = E; (val E_1 E_2)) = (val E_1 (assign x = E; E_2))
(assign x = E; (par E_1 E_2)) = (par (assign x = E; E_1) E_2)
  = (par E_1 (assign x = E; E_2))

Example 1. We first annotate the original specification with val to give an 'eager' rather than 'lazy' version of fork. (This is in preparation for an MI step.) Thus, our
specification becomes

\[
\text{transform } t = \text{replace } t \left( \text{tmin } t \right) \\
\text{replace \ (tip } n) \ m = \text{tip } m \\
\text{replace \ (fork } L \ R) \ m = \text{val} \left( \text{val } \text{fork } \left( \text{replace } L \ m \right) \right) \ \left( \text{replace } R \ m \right) \\
\text{tmin } \left( \text{tip } n \right) = n \\
\text{tmin } \left( \text{fork } L \ R \right) = \text{min } \left( \text{tmin } L \right) \ \left( \text{tmin } R \right)
\]

**Transformation.** Define \( \text{replace}' \ t \ m \ v = \text{assign } v = \text{tmin } t; \ \text{replace } t \ m \). Synthesize a new definition of \( \text{replace}' \).

(i) \( \text{replace}' \ (\text{tip } n) \ m \ v = \text{assign } v = \text{tmin } (\text{tip } n); \ \text{replace } (\text{tip } n) \ m \)

\[\begin{aligned}
\text{instantiating } \text{tip } n \text{ for } t \\
= \text{assign } v = n; \ \text{tip } m
\end{aligned}\]

% unfolding definitions of \( \text{tmin} \) and \( \text{replace} \)

(ii) \( \text{replace}' \ (\text{fork } L \ R) \ m \ v = \text{assign } v = \text{tmin } (\text{fork } L \ R); \ \text{replace } (\text{fork } L \ R) \ m \)

\[\begin{aligned}
\text{instantiating } \text{fork } L \ R \text{ for } t \\
= \text{assign } v = \text{min } (\text{tmin } L) \ \left( \text{tmin } R \right); \\
\text{val} \left( \text{val } \text{fork } \left( \text{replace } L \ m \right) \right) \ \left( \text{replace } R \ m \right)
\end{aligned}\]

% unfolding definitions of \( \text{tmin} \) and \( \text{replace} \)

\[\begin{aligned}
= \text{var } y, z; \ \text{assign } v = \text{min } y \ z, y = \text{tmin } L, z = \text{tmin } R; \\
\text{val} \left( \text{val } \text{fork } \left( \text{replace } L \ m \right) \right) \ \left( \text{replace } R \ m \right)
\end{aligned}\]

% AV

\[\begin{aligned}
= \text{var } y, z; \ \text{assign } v = \text{min } y \ z; \\
\left( \text{assign } z = \text{tmin } R; \ \text{replace } R \ m \right)
\end{aligned}\]

% MI

\[\begin{aligned}
= \text{var } y, z; \ \text{assign } v = \text{min } y \ z; \\
\text{val} \left( \text{val } \text{fork } \left( \text{replace}' \ L \ m \ y \right) \right) \ \left( \text{replace}' \ R \ m \ z \right)
\end{aligned}\]

% folding definition of \( \text{replace}' \)

(A proof that the MI step maintains strong equivalence can be done by structural induction on \( R \) and relies on \( \text{tip} \) being non-strict. If \( \text{par} \) had been used in place of \( \text{val} \), its properties would make such a proof unnecessary.)

Synthesize a new definition of \( \text{transform} \).

\[
\text{transform } t = \text{replace } t \left( \text{tmin } t \right) \\
= \text{var } m; \ \text{assign } m = \text{tmin } t; \ \text{replace } t \ m \ % \text{AV} \\
= \text{var } m; \ \text{replace}' \ t \ m \ m \ % \text{folding definition of } \text{replace}'
\]

**Solution 1.**

\[
\text{transform } t = \text{var } m; \ \text{replace}' \ t \ m \ m \\
\text{replace}' \ (\text{tip } n) \ m \ v = \text{assign } v = n; \ \text{tip } m \\
\text{replace}' \ (\text{fork } L \ R) \ m \ v = \text{var } y, z; \ \text{assign } v = \text{min } y \ z; \\
\text{val} \left( \text{val } \text{fork } \left( \text{replace}' \ L \ m \ y \right) \right) \ \left( \text{replace}' \ R \ m \ z \right)
\]
Example 2.

\[ \text{transform } t = \text{replace } t \left( \text{sort } \left( \text{tips } t \right) \right) \]
\[ \text{replace } \left( \text{tip } n \right) u = \text{tip } \left( \text{head } u \right) \]
\[ \text{replace } \left( \text{fork } L \ R \right) u = \text{val } \left( \text{val fork } \left( \text{replace } L \ u \right) \right) \]
\[ \left( \text{replace } R \left( \text{drop } \left( \text{size } L \right) \ u \right) \right) \]
\[ \text{tips } \left( \text{tip } n \right) = \left[ n \right] \]
\[ \text{tips } \left( \text{fork } L \ R \right) = \text{tips } L ++ \text{tips } R \]
\[ \text{size } \left( \text{tip } n \right) = 1 \]
\[ \text{size } \left( \text{fork } L \ R \right) = \text{size } L + \text{size } R \]

Transformation. Define \( \text{replace}^\prime \) \( t \ u \ v \ w = \text{assign } v = \text{tips } t, w = \text{drop } \left( \text{size } t \right) \ u; \text{replace } t \ u \). Synthesize a new definition of \( \text{replace}^\prime \).

(i) \( \text{replace}^\prime \left( \text{tip } n \right) u v w \)
\[ = \text{assign } v = \text{tips } \left( \text{tip } n \right), w = \text{drop } \left( \text{size } \left( \text{tip } n \right) \right) \ u; \]
\[ \text{replace } \left( \text{tip } n \right) u \ % \text{instantiating } \text{tip } n \text{ for } t \]
\[ = \text{assign } v = \left[ n \right], w = \text{drop } 1 \ u; \text{tip } \left( \text{head } u \right) \]
\[ % \text{unfolding definitions of } \text{tips}, \text{size} \text{ and } \text{replace} \]
\[ = \text{assign } v = \left[ n \right], w = \text{tail } u; \text{tip } \left( \text{head } u \right) \]
\[ % \text{property of } \text{drop} \]

(ii) \( \text{replace}^\prime \left( \text{fork } L \ R \right) u v w \)
\[ = \text{assign } v = \text{tips } \left( \text{fork } L \ R \right), \]
\[ w = \text{drop } \left( \text{size } \left( \text{fork } L \ R \right) \right) \ u; \]
\[ \text{replace } \left( \text{fork } L \ R \right) u \ % \text{instantiating } \text{fork } L \ R \text{ for } t \]
\[ = \text{assign } v = \text{tips } L ++ \text{tips } R, \]
\[ w = \text{drop } \left( \text{size } L + \text{size } R \right) \ u; \]
\[ \text{val } \left( \text{val fork } \left( \text{replace } L \ u \right) \right) \]
\[ \left( \text{replace } R \left( \text{drop } \left( \text{size } L \right) \ u \right) \right) \]
\[ % \text{unfolding definitions of } \text{tips}, \text{size} \text{ and } \text{replace} \]
\[ = \text{assign } v = \text{tips } L ++ \text{tips } R, \]
\[ w = \text{drop } \left( \text{size } R \right) \left( \text{drop } \left( \text{size } L \right) \ u \right); \]
\[ \text{val } \left( \text{val fork } \left( \text{replace } L \ u \right) \right) \]
\[ \left( \text{replace } R \left( \text{drop } \left( \text{size } L \right) \ u \right) \right) \]
\[ % \text{property of } \text{drop} \]
\[ = \text{var } x, y, z; \text{assign } v = x ++ y, w = \text{drop } \left( \text{size } R \right) z, \]
\[ x = \text{tips } L, y = \text{tips } R, z = \text{drop } \left( \text{size } L \right) u; \]
\[ \text{val } \left( \text{val fork } \left( \text{replace } L \ u \right) \right) \left( \text{replace } R \ z \right) \ % \text{AV} \]
\[ = \text{var } x, y, z; \text{assign } v = x ++ y; \]
\[ \text{val } \left( \text{val fork } \left( \text{assign } x = \text{tips } L, z = \text{drop } \left( \text{size } L \right) u; \text{replace } L \ u \right) \right) \]
\[ \left( \text{assign } y = \text{tips } R, w = \text{drop } \left( \text{size } R \right) z; \text{replace } R \ z \right) \ % \text{MI} \]
\[ = \text{var } x, y, z; \text{assign } v = x ++ y; \]
\[ \text{val } \left( \text{val fork } \left( \text{replace}^\prime \left( L \ u \ x \ z \right) \right) \left( \text{replace}^\prime \ R \ z \ y \ w \right) \right) \]
Functional programming with side-effects

% folding definition of replace’

Synthesize a new definition of transform.
transform t = replace t (sort (tips t))
= var v, w; assign v = tips t, w = drop (size t) (sort v);
   replace t (sort v) % AV
= var v, w; replace’ t (sort v) v w
   % folding definition of replace’

Solution 2

transform t = var v, w; replace’ t (sort v) v w
replace’ (tip n) u v w = assign v = [n], w = tail u; tip (head u)
replace’ (fork L R) u v w = var x, y, z; assign v = x ++ y;
   val (val fork (replace’ L u x z))(replace’ R z y w)

Alternative transformation. Define replace” t u v w r = assign v = tips t ++ r, w = drop (size L) u; replace t u. Synthesize a new definition of replace”.

(i) replace” (tip n) u v w r = assign v = tips (tip n) ++ r,
   w = drop (size (tip n)) u;
   replace (tip n) u % instantiating tip n for t
= assign v = [n] ++ r,
   w = drop 1 u; tip (head u)
   % unfolding definitions of tips, size and replace
= assign v = n : r, w = tail u; tip (head u)
   % properties of ++ and drop

(ii) replace” (fork L R) u v w r
   = assign v = tips (fork L R) ++ r,
   w = drop (size (fork L R)) u;
   replace (fork L R) u % instantiating fork L R for t
   = assign v = (tips L ++ tips R) ++ r,
   w = drop (size L + size R) u;
   val (val fork (replace L u)) (replace R (drop (size L) u))
   % unfolding definitions of tips, size and replace
   = assign v = tips L ++ (tips R ++ r),
   w = drop (size R) (drop (size L) u);
   val (val fork (replace L u)) (replace R (drop (size L) u))
   % properties of ++ and drop
   = var y, z; assign v = tips L ++ y, y = tips R ++ r,
   w = drop (size R) z,
   z = drop (size L) u;
   val (val fork (replace L u)) (replace R z) % AV
= var y, z; val (val fork (assign v = tips L ++ y, 
z = drop (size L) u; replace L u))
    (assign y = tips R ++ r, w = drop (size R) z; replace R z)  % MI
= var y, z; val (val fork (replace" L u v z y)) (replace" R z y w r)
    % folding definition of replace"

Synthesize a new definition of transform.

\[
\text{transform } t = \text{replace } t \left( \text{sort } (\text{tips } t) \right) \\
\begin{align*}
&= \text{var } v, w; \text{assign } v = \text{tips } t, w = \text{drop} \left( \text{size } t \right) (\text{sort } v); \\
&\quad \text{replace } t \left( \text{sort } v \right) \quad \% \text{AV} \\
&= \text{var } v, w; \text{assign } v = \text{tips } t \text{++ [ ], } w = \text{drop} \left( \text{size } t \right) (\text{sort } v); \\
&\quad \text{replace } t \left( \text{sort } v \right) \quad \% \text{property of ++} \\
&= \text{var } v, w; \text{replace" } t \left( \text{sort } v \right) v w [ ] \\
&\quad \% \text{folding definition of replace"}
\end{align*}
\]

Solution 2(a).

\[
\begin{align*}
\text{transform } t &= \text{var } v, w; \text{replace" } t \left( \text{sort } v \right) v w [ ] \\
\text{replace" } (\text{tip } n) u v w r &= \text{assign } v = n : r, w = \text{tail } u; \text{tip } (\text{head } u) \\
\text{replace" } (\text{fork } L R) u v w r &= \text{var } y, z; \\
&\quad \text{val } (\text{val } \text{fork } (\text{replace" } L u v z y)) \\
&\quad \quad (\text{replace" } R z y w r)
\end{align*}
\]

Example 3.

\[
\begin{align*}
\text{quicksort } [ ] &= [ ] \\
\text{quicksort } (a : x) &= \text{quicksort } \{ b \leftarrow x; b < a \} \text{++ } a : \text{quicksort } \{ b \leftarrow x; b \geq a \}
\end{align*}
\]

Transformation. Define \( \text{partition } a \times \text{rest } = \text{assign } \text{rest } = \{ b \leftarrow x; b \geq a \}; \{ b \leftarrow x; b < a \} \). Synthesize a new definition of \( \text{partition} \).

(i) \( \text{partition } a [ ] \text{rest } \)
\[
\begin{align*}
&= \text{assign } \text{rest } = \{ b \leftarrow [ ]; b \geq a \}; \{ b \leftarrow [ ]; b < a \} \quad \% \text{instantiating } [ ] \text{ for } x \\
&= \text{assign } \text{rest } = [ ]; [ ]; \quad \% \text{property of sets}
\end{align*}
\]

(ii) \( \text{partition } a \ (e : x) \text{rest } = \text{assign } \text{rest } = \{ b \leftarrow (e : x); b \geq a \}; \{ b \leftarrow (e : x); b < a \} \\
\quad \% \text{instantiating } (e : x) \text{ for } x
\]

Case \( e < a \):
\[
\begin{align*}
\text{partition } a \ (e : x) \text{rest } &= \text{assign } \text{rest } = \{ b \leftarrow x; b \geq a \}; \{ b \leftarrow x; b < a \} \\
\quad \% \text{property of sets} \\
&= e : \text{assign } \text{rest } = \{ b \leftarrow x; b \geq a \}; \{ b \leftarrow x; b < a \} \\
\quad \% \text{MI} \\
&= e : \text{partition } a \times \text{rest } \\
\quad \% \text{folding definition of } \text{partition}
\end{align*}
\]

N.B. The MI step above only preserves weak equivalence, since \( \text{cons} \) is non-strict in its arguments. However, if \( \text{partition} \) is considered as a function defined local to
quicksort, the fact that the tail of \((\text{partition } a \ (e : x) \ \text{rest})\) is always evaluated before \text{rest}, is sufficient to ensure freedom from deadlock. Alternatively, explicit annotation with \text{val} could be used to preserve strong equivalence.

Case \(e \geq a\):

\[
\text{partition } a \ (e : x) \ \text{rest} = \begin{cases} 
\text{assign rest} = e; \ \{b \leftarrow x; b \geq a\}; \ \{b \leftarrow x; b < a\} & \% \text{property of sets} \\
\text{var rest'}; \ \text{assign rest} = e; \ \text{rest'} = \{b \leftarrow x; b \geq a\}; \ \{b \leftarrow x; b < a\} & \% \text{AV} \\
\text{var rest'}; \ \text{assign rest} = e; \ \text{rest'}; \ \text{partition } a \ x \ \text{rest'} & \% \text{folding definition of partition}
\end{cases}
\]

Synthesize a new definition of quicksort.

\[
\text{quicksort} \ (a : x) = \begin{cases} 
\text{quicksort} \ \{b \leftarrow x; b < a\} \ ++ \ a: \ \text{quicksort} \ \{b \leftarrow x; b \geq a\} & \\
\text{var rest}; \ \text{assign rest} = \{b \leftarrow x; b \geq a\}; \ \\
\text{quicksort} \ \{b \leftarrow x; b < a\} \ ++ \ a: \ \text{quicksort rest} & \% \text{AV} \\
\text{var rest}; \ \text{quicksort} (\text{assign rest} = \{b \leftarrow x; b \geq a\}; \ \{b \leftarrow x; b < a\}) \ ++ \\
\text{a: qucksort rest} & \% \text{MI} \\
\text{var rest}; \ \text{quicksort} (\text{partition } a \ x \ \text{rest}) \ ++ \ a: \ \text{quicksort rest} & \% \text{folding definition of partition}
\end{cases}
\]

Solution 3.

\[
\text{quicksort} \ [ ] = [ ]
\text{quicksort} \ (a : x) = \text{var rest};
\text{quicksort} (\text{partition } a \ x \ \text{rest}) \ ++ \ a: \ \text{quicksort rest}
\text{partition} a \ [ ] \ \text{rest} = \text{assign rest} = [ ]; [ ]
\text{partition} a \ (e : x) \ \text{rest} = e : \text{partition} a \ x \ \text{rest}, \ e < a
\text{var rest'}; \ \text{assign rest} = e; \ \text{rest'}; \ \text{partition} a \ x \ \text{rest'}
\]

Example 5.

\[
\text{fib} \ 0 = 1
\text{fib} \ 1 = 1
\text{fib} \ n = \text{fib} \ (n - 1) + \text{fib} \ (n - 2)
\]

Transformation. Define \text{fib'} \ n \ x = \text{assign} x = \text{fib} \ (n - 1); \ \text{fib} \ n, \ n > 0. Synthesize a new
definition of $\text{fib}'$. 

(i) 
\[
\text{fib}' \ 1 \ x = \text{assign} \ x = \text{fib} \ 0; \ \text{fib} \ 1 \ \ %\text{instantiating 1 for } n \\
= \text{assign} \ x = 1; \ 1 \ \ %\text{unfolding definition of } \text{fib}
\]

(ii) 
\[
\text{fib}' \ n \ x = \text{assign} \ x = \text{fib} \ (n - 1); \ \text{fib} \ n, \ n > 1 \\
= \text{assign} \ x = \text{fib} \ (n - 1); \ x + \text{fib} \ (n - 2) \ \ %\text{unfolding definition of } \text{fib} \\
= \text{var} \ y; \ \text{assign} \ y = \text{fib} \ (n - 2), \ x = \text{fib} \ (n - 1); \ x + y \ %\text{AV} \\
= \text{var} \ y; \ \text{assign} \ x = (\text{assign} \ y = \text{fib}' \ (n - 2); \ \text{fib} \ (n - 1)); \ x + y \ %\text{MI} \\
= \text{var} \ y; \ \text{assign} \ x = \text{fib}' \ (n - 1) \ y; \ x + y \\
\] %folding definition of $\text{fib}'$

(The MI step preserves strong equivalence, assuming $x+y$ demands $x$ before, or in parallel, with $y$.)

Synthesize a new definition of $\text{fib}$.
\[
\text{fib} \ n = \text{var} \ x; \ \text{assign} \ x = \text{fib}' \ (n - 1); \ \text{fib} \ n, \ n > 0 \ %\text{AV} \\
= \text{var} \ x; \ \text{fib}' \ n \ x \ %\text{folding definition of } \text{fib}'
\]

Solution 5.
\[
\text{fib} \ 0 = 1 \\
\text{fib} \ n = \text{var} \ x; \ \text{fib}' \ n \ x \\
\text{fib}' \ 1 \ x = \text{assign} \ x = 1; \ 1 \\
\text{fib}' \ n \ x = \text{var} \ y; \ \text{assign} \ x = \text{fib}' \ (n - 1) \ y; \ x + y
\]

References