Implementing Functional Programs Using Mutable Abstract Data Types

Ganesh C. Gopalakrishnan,
M. K. Srivas (SUNY, Stony Brook),

Tech. Report. UUCS-87-019

Implementing Functional Programs Using Mutable Abstract Data Types

Ganesh C. Gopalakrishnan,
Department of Computer Science, Univ. of Utah, Salt Lake City, Utah 84112

and

Mandayam K. Srivas,
Department of Computer Science, SUNY, Stony Brook, NY 11794

We study the following problem in this paper. Suppose we have a purely functional program that uses a set of abstract data types by invoking their operations. Is there an order of evaluation of the operations in the program that preserves the applicative order of evaluation semantics of the program even when the abstract data types behave as mutable modules. An abstract data type is mutable if one of its operations destructively updates the object rather than returning a new object as a result. This problem is important for several reasons. It can help eliminate unnecessary copying of data structure states. It supports a methodology in which one can program in a purely functional notation for purposes of verification and clarity, and then automatically transform the program into one in an object oriented, imperative language, such as CLU, ADA, Smalltalk, etc., that supports abstract data types. It allows accruing both the benefits of using abstract data types in programming, and allows modularity and verifiability.

Keywords: Functional Program Implementation, Mutable Modules, Abstract Data Types, Syntactic Conditions.

Contents

List of Figures

1 Introduction

Suppose we have a purely functional [9] program P that uses a set of abstract data types [8,3] by invoking their operations. Suppose we view every abstract data type in P to be *mutable*, i.e., one in which some of the operations creates an instance of the module type by destructively updating the old instance, rather than creating a new copy. This will, in general, alter the meaning of P. P with mutable data types is sensitive to the order of evaluation of the data type operations in it. This poses the following interesting questions: Is there a way of correctly implementing a functional program employing in situ (in place) update operations so that no copying is necessary ? If so, under what conditions is this possible ? In this paper, we study a property, referred to as the *in situ evaluability property* of functional programs, which helps answer the above questions. .

We define a purely functional program P to be *in situ evaluable* if

- Some of the data types in P can be implemented using mutable modules;
- An evaluation order for the operations in P can be found such that the intended semantics of P is preserved. This evaluation order will be called the *in situ evaluation order.*

Our work analyses the conditions for in situ evaluation in the context of applicative order evaluation. We formulate syntactic conditions on functional programs, and show that they are sufficient to ensure in situ evaluation. These conditions are also necessary for expressions not containing conditionals or recursion. (For conditionals and recursive expressions, a set of necessary syntactic conditions seem to be impossible to formulate because of the undecidability of the halting problem.) The proof that our syntactic conditions are sufficient is constructive in that it defines the in situ evaluation order.

The procedure can be used to directly transform a functional program into an equivalent one in an imperative, object oriented language, such as CLU [14], ADA [17], Smalltalk [4]. We also show that in some cases it is possible to transform a functional program which is not in situ evaluable into one that is, using the algebraic axioms of the abstract data types used in the program.

As an example consider the following expression which denotes a computation on an object q belonging to a *Queue* data type. The operation *ins* returns a new queue obtained by adding a given element to q, and $front$ fetches the front element of q.

$$ins(ins(q, v), front(q)) \tag{1}$$

In a purely functional language, the arguments to the operations in expression (1) can be evaluated in any order. Suppose we assume that *ins* is destructive, i.e., it returns q after it actually modifies q by adding v. Then, in order for the expression to return the same result as before, *front* has to be evaluated before the (inner) *ins* operation unless there is a facility to save the state of q before evaluating the (inner) *ins*. Now consider the following expression:

$$ins(q, front(ins(q, v))) \tag{2}$$

For this expression there exists no order of invocations of the operations that would evaluate it consistent with the applicative order semantics if *ins* were destructive without saving

the original state of q. This is because the inner *ins* operation would modify q before it is needed by the outer *ins*. According to our definition, the first expression is in situ evaluable whereas the second one is not. However, the second expression is semantically equivalent to the following expression which is in situ evaluable:

$$if\ empty(q)\ then\ ins(q, v)\ else\ ins(q, front(q)) \tag{3}$$

1.1 Related Work

The closest related effort is that reported in [10]. In this work, the problem of updating arrays and similar contiguously allocated storage structures (*aggregates*) has been studied, with a view to detect situations where destructive updates can be performed on array locations without affecting the *call by need* (normal order) [9] semantics. In [12], safe procedural implementations of data types has been studied. Our work is distinguished in the following respects:

- We perform the analysis with respect to arbitrary abstract data types (not just Lists or Arrays as in [10].)
- Our technique does not involve abstract interpretation. It is simpler to implement than the analysis suggested in [10] which uses abstract interpretation [16].
- In [12], determining in situ evaluability by syntactic analysis has not been considered; the approach taken there is to combine Dijkstra's predicate transformer semantics [2] and the algebraic semantics to effect transformations.

1.2 Terminology, Assumptions, and Problem Statement

An *abstract data type* consists of a set (possibly infinite) of values, and a finite set of operations with the constraint that the operations are the only means of constructing, observing and manipulating the values. For the purposes of our examples, we group the data types into two kinds: *module type*, and *simple type*. Examples of simple types are: *integer*, *boolean*. Some times we refer to the values of a module type as "states" since they denote the states of an object instance of the module type. Every data type used in our examples other than the simple types listed above are assumed to be a module type.

We classify the operations of a data type into two groups: *constructors*, and *observers*. Every operation of a data type which returns as its result a value of that type is a constructor, eg., *ins* on *Queue*. Every operation which returns a value belonging to a type other than the type under question is an observer, eg., *front* on *Queue*. We assume that every observer of a module type returns a value of a simple type, and that every observer of a simple type returns a value of another simple type.

An object of a data type is *mutable* if one or more of the operations defined by the data type are implemented so as to destructively update one or more of their argument objects (and *immutable* otherwise). We assume that objects belonging to module types are mutable; it is these objects that we wish to avoid copying. (For brevity we often use the phrase "a module type is mutable".) Objects belonging to simple types are immutable;

therefore we do not attempt to avoid their copying. For convenience, we further assume that every constructor of a module type is destructive, but no observer is destructive.

The aim of our analysis is to ensure that the number of *instances* of each module type remains the same throughout the duration of the computation of an expression. The number of instances of a module type used in an expression E is the same as the number of distinct terminal nodes of the same module type present in $graph(E)$, where $graph(E)$ is the graph of expression E with shared common subexpressions. (This simple scheme to determine the number of module instances to be allocated in the beginning of a computation can be improved as shown in section 2.3.) This implies that all common subexpressions of a module type, as well as expressions E_1 and E_2 of module type, one of which is a subexpression of the other, denote "state values" that are resident in a single instance of that module type. For example, in expression (2), the subexpression q as well as the expression $ins(q, v)$ denote the states of the same queue instance.

In general, if E_1 is a subexpression of E_2 and both are of the same module type, it is the case that E_1 is derived from E_2 by a series of constructor applications. Since we require in-place updates for constructors, E_1 and E_2 denote two different states of the same module instance existing at different times.

Statement of the problem

The language of $\mathcal{E}$ ($L(\mathcal{E})$) defines the class of programs addressed in our work:

$$\mathcal{E} ::= s(\mathcal{E}, ..., \mathcal{E}) \mid var \mid f(M, \mathcal{E}, ..., \mathcal{E}) \mid COND\{p_1 : \mathcal{E}_1; p_2 : \mathcal{E}_2; ...p_n : \mathcal{E}_n\} \mid iterative\_recursion \tag{4}$$

where p_i are called *antecedents* and $\mathcal{E}_i$ are called *consequents* of the conditional expression, $COND$. Syntactic details of $iterative\_recursion$ [15] are provided in section 2.2.2.

Here, s denotes operators of the simple types (including constants), *var* ranges over simple types as well as module types, and f denotes either the tupling operator or a constructor/observer f applied on an instance a module M of a certain module type, with additional expressions as arguments. The problem to be solved by us is:

Given a functional expression E belonging to the language of $\mathcal{E}$, $L(\mathcal{E})$,

1. Develop syntactic conditions to determine whether E is in situ evaluable. If so, determine the partial order of evaluation that achieves this (the in situ evaluation order).

2. In case E cannot be in situ evaluated, find out whether E can be transformed into a semantically equivalent E' that can be evaluated in situ.

2 Syntactic Characterization of In Situ Evaluability

We consider straight-line expressions (expressions without *cond* and recursion) in section 2.1, and prove the necessity and sufficience of our syntactic conditions for them. *cond* and recursion will be considered in section 2.2.

2.1 Syntactic Conditions for Straight-line Expressions

We illustrate our syntactic conditions on the following two expressions:

$$ins(Q, front(ins(Q, v))) \tag{5}$$
$$read(M, read(write(M, a1, d1), a2)) \tag{6}$$

The first expression denotes a computation on a Queue object, and purports to advance a queue Q to a state $ins(Q, v)$, observes its front to get a value u, and inserts u into the original queue Q. The second expression denotes a computation on an object M of the $Memory$ data type, supporting $read$ and $write$ operations. $read$ takes a $Memory$ and an address and returns a data item of a simple type. $write$ takes a $Memory$, an address and a data item and returns a new $Memory$. The expression 6 purports to read a $Memory$ M at an address $a3$ where $a3$ is obtained by first advancing M to state $write(M, a1, d1)$ and then reading this state at address $a2$.

To distinguish repeated occurrences of the same operator (such as ins occurring twice), we append the suffix ": $rank$" to them. This suffix will be omitted for distinct operators. We rewrite the above two expressions as:

$$E_q = ins : 1(Q, front : 1(Q, ins : 2(Q, v))) \tag{7}$$
$$E_m = read : 1(M, read : 2(write : 1(M, a1, d1), a2)) \tag{8}$$

2.1.1 Definition of $graph(E)$, a Graphical Representation of Expressions

1. Define the $\rightarrow$ relation (read "directly depends") for an expression as the least relation containing all pairs $< operator1, operator2 >$ such that $operator1$ can be applied as soon as (but not before) the value created by $operator2$ becomes available. ($operator2$ could also be a variable or a constant. Since their values are trivially available, we prefer to overload $operator2$ rather than introduce another separate category containing variables and constants alone.)

   Example: for E_q,

   $ins : 1 \rightarrow Q, ins : 1 \rightarrow front : 1, front : 1 \rightarrow ins : 2$
   $ins : 2 \rightarrow Q, ins : 2 \rightarrow v.$

   We use $\stackrel{*}{\rightarrow}$ to denote the transitive closure of $\rightarrow$.

2. Define the relation $\rightsquigarrow$ (read "c-after-o") for an expression as the least relation containing all
   $< constructor, observer >$ pairs $< c, o >$ such that they have a common subexpression of module-type $Mtype$ as an argument (they share a module instance in the same state), i.e.

   $c \rightsquigarrow o \Leftrightarrow$
   $\exists x. c \rightarrow x \wedge o \rightarrow x \wedge operator\ x\ creates\ a\ value\ of\ module\_type\ Mtype).$

Example: for E_m, $write : 1 \rightsquigarrow read : 1$,
where the common sub-expression that is shared among $write : 1$ and $read : 1$ ("$\exists x$ part") is the variable M of module type 'Memory'.

Intuitively, $c \rightsquigarrow o$ means that "the constructor c must be applied on module M only after all observers o have been applied on module M."

3. Take the union of $\rightarrow$ and $\rightsquigarrow$ for an expression E, and depict it as a graph with *shared common subexpressions*. This graph, $graph(E)$, depicts all the data dependencies of E as well as all evaluation orderings among constructors and observers. $graph(E_q)$ and $graph(E_m)$ are illustrated in figure 1.

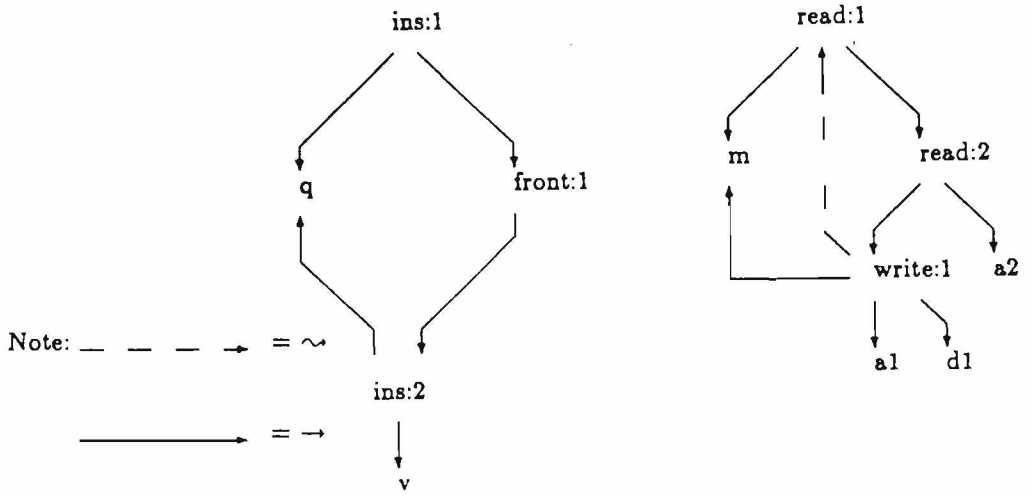

Figure 1: Dags of E_q and E_m

Theorem. E can be evaluated in situ if and only if:

(Clause "*Acyclic*"): $graph(E)$ is acyclic; and

(Clause "*Chain*"): For each *module instance* M, there exists a **single** chain of $\rightarrow$ arrows in $graph(E)$ such that all the constructors c acting on this module instance lie on this chain.

E.g.: We can see that E_m violates the clause *Acyclic* and E_q violates the clause *Chain*. Hence neither E_q nor E_m can be evaluated in situ.

Proof of Necessity and Sufficience

2.1.2 Informal Proof

The clause *Acyclic* means that both data dependencies as well as "constructor after observer" orderings can be satisfied. Viewed another way, the various operator precedences to be observed are deadlocked if clause *Acyclic* is violated.

The clause *Chain* captures the fact that the state of each module instance M is successively updated (along the chain) by constructor applications. Instead of a chain if we had

a branching structure, there is an attempt to simultaneously update the state of a module in two distinct ways which is not possible without copying.

Note: In general, there will be several mutable modules in a system. A computation on such a system would advance the states of each of the mutable submodules. Our syntactic characterization can handle this important general situation as well. The clause *Chain* would then require that the states of the different submodule instances lie on their own separate chains.

For instance, consider the expression E_{mn} that defines a computation in a system containing two module instances M and N:

$$< c1(m, o1(c2(n))), c3(n, o2(c4(m))) > .$$

In this example, modules M and N are respectively in states m and n to begin with. We then attempt to create a future state for M by using the current state m of M as well as a future state of N; more specifically we apply the constructor $c1$ on m and also using *a value produced by observing (via o) a future state of* N, namely $c2(n)$. Similarly we attempt to create a future state for N using the current state n of N and a future state of M. This example violates clause *Chain* twice, because neither the various states of M nor the various states of N are along chains of $\rightarrow$s (figure 2). The states $c1(m, ...)$ and $c4(m)$ are, for instance non-equivalent, in general.

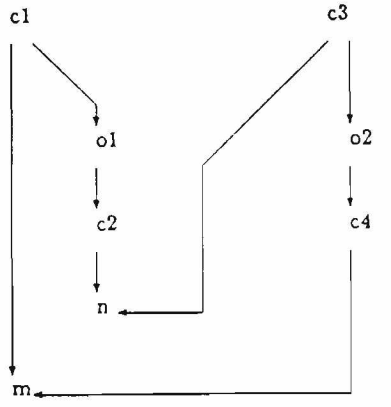

Figure 2: Violation of *Chain*

2.1.3 Formal Proof (In two parts, Theorems 2.1 and 2.2)

In the applicative order evaluation of an expression E, each subexpression (including E) gets evaluated at a certain time instant. (Note: We use the word *time* exactly as in *Linear-time Temporal Logic*, i.e. as a means of ordering events.) If a subexpression E denotes a module state present in a module M, the value ceases to become available as soon as any constructor c is applied on M. If a subexpression E denotes a simple value (a value belonging to a simple type), the value can be regarded as being eternal because we allow these values to be copied and saved at will.

For $graph(E)$, we can now define the following three functions:

avail, that takes an operator node f in $graph(E)$ and returns the instant at which f yields its value;

cease, that returns for each node in the graph, the instant of time after which the value represented by that node gets mutated (i.e. ceases to exist; this concept is analogous to the notion of "liveness" of variables in classical compiler data flow analysis [1]). In our framework, if $c \rightarrow N$, the *start* value for c is the *cease* value for N.

start, defined such that $start(f) = avail(f) - 1$ (i.e. it takes "unit" time for results to become *avail*able once the operator is *start*ed.)

We now define in situ evaluation formally:

Definition 2.1 *(In situ evaluation)* An expression E is evaluable in situ if and only if:

1. For all nodes n in $graph(E)$, $avail(n)$ is finite (i.e. operator n is applied at some finite time).
2. Among the terminal nodes of $graph(E)$, there must be no more than one node denoting the state of each module instance. This amounts to saying that at the start of computation, all module instances are exactly in one state!
3. For every pair of nodes m and n in $graph(E)$, if m and n denote two different states of the same instance of a module,

   $$avail(m) \neq avail(n).$$

   As a special case, If $c2 \rightarrow c1$, then

   $$start(c2) = cease(c1),$$

   meaning that the value created by $c1$ ceases to exist beyond the time at which $c2$ is invoked. This also means that

   $$avail(c2) = cease(c1) + 1.$$

   This means that there can be only one version of the state of a module at each time instant.
4. For every $op_2 \rightarrow op_1$ where one of op_i is an observer,

   $$start(op_2) > start(op_1).$$

   This requirement follows from data dependencies among operators.
5. For every $o \rightarrow c$,

   $$start(o) < cease(c).$$

   This means that the observation of a module state must start before the instant at which the *cessation of the state defined by c commences.*

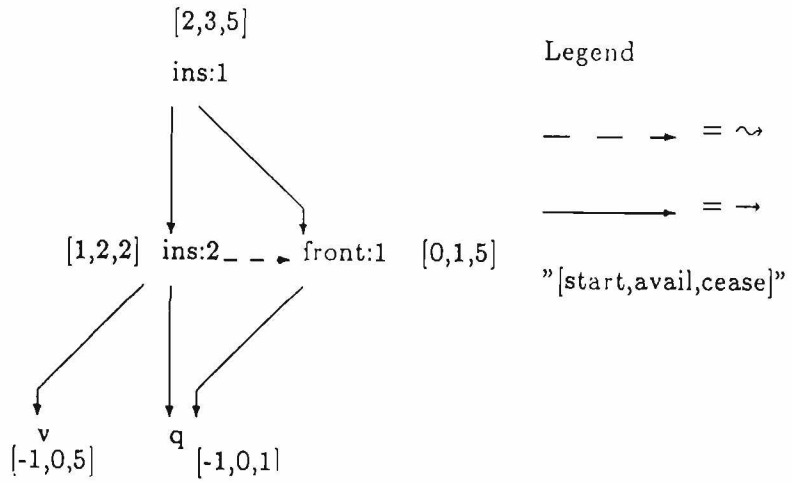

Figure 3: Sufficience of *Chain* and *Acyclic*

Note 2.1 Although the $\rightsquigarrow$ arc is not explicitly mentioned above, the precedence imposed by it has already been captured above. To see this, consider $o \rightarrow c1$ and $c2 \rightarrow c1$. According to the definition of $\rightsquigarrow$, $c2 \rightsquigarrow o$. Since we have $start(c2) = cease(c1)$ (clause 3) and $start(o) < cease(c1)$ (clause 5), we have $start(o) < start(c2)$ which is exactly what we wanted. Therefore,
$c \rightsquigarrow o$ implies $start(o) < start(c)$. (Thus $\rightsquigarrow$ is introduced just for convenience; all our proofs can be carried out without using the $\rightsquigarrow$ arc—albeit more difficultly.)

Theorem 2.1 *(Necessity of Chain and Acyclic)* If *Chain* or *Acyclic* are violated for a given E, then E cannot be evaluated in situ.

Proof: This follows from the definition of in situ evaluation.

If *Chain* is violated, there exist two constructors $c1$ and $c2$ such that

1. either $c1 \stackrel{*}{\rightarrow} x$ and $c2 \stackrel{*}{\rightarrow} x$ for some constructor x where x is the least common descendant for both $c1$ and $c2$; or
2. there exists no x as above.

In the latter case, we have two distinct chains of states for the same module instant, thus immediately violating clause 2. In the former case, there exist $M.c1' \rightarrow x$ and $M.c2' \rightarrow x$. From the definition of in situ evaluation (clause 3),

$$avail(M.c1') = cease(x) + 1 = avail(M.c2'),$$

contradicting clause 3. Thus if *chain* is violated, E cannot be evaluated in situ.

If *Acyclic* is violated, we can assign a high value to the *avail* times of all nodes within the cycle by traversing the cycle arbitrarily many times, jacking the value of the *start* function to infinity (using clauses 4 and Note 2.1), violating clause 1.

Theorem 2.2 *(Sufficience of Chain and Acyclic)* If *Chain* and *Acyclic* hold, then E can be evaluated in situ.

Proof: What we have to prove is that given $graph(E)$ satisfying both *Chain* and *Acyclic*, we can use an evaluation rule such that all the clauses of the in situ evaluation will hold for $graph(E)$. Such an evaluation rule is now defined.

Definition 2.2 *(In situ evaluation ordering)* Respect the orderings imposed by both the $\rightarrow$ and the $\rightsquigarrow$ arcs.

The proof is completed by the following steps that define the functions *avail* and *cease* in such a manner that all the requirements of in situ evaluation are satisfied. (We also illustrate our steps on an example in figure 3.)

1. Define *start* to yield -1 for all terminal nodes of $graph(E)$.

2. For every other node N, *start* assigns one less than the *largest* among all distances from N to all the reachable terminal nodes. (We define the *distance* between two nodes to be the length of the longest path, containing $\rightarrow$ and $\rightsquigarrow$ arcs, connecting the two nodes.) *Avail* is also defined once *start* is.

3. Now we define *cease* for the nodes lying on the chains of each module instance. For each constructor node $N1$ of a module M on this chain, $cease(N1) = avail(N2) - 1$ where $N2 \rightarrow N1$ is part of the chain.

4. For all nodes not on any of the constructor chains (e.g. observer nodes) as well as for those nodes with an in-degree of 0, *cease* assigns a value equal to the total number of nodes in $graph(E)$; essentially, values produced by observers are available throughout the duration of computation of E, and the number of nodes in $graph(E)$ is guaranteed to be larger than the duration of E's computation.

We now list the clauses pertaining to in situ evaluation, and write against them the reason why they are satisfied:

Clause 1 is satisfied since $graph(E)$ is acyclic, all nodes of $graph(E)$ will be assigned finite values.

Clause 2 is satisfied since *avail* and *cease* were defined so as to satisfy Clause 3.

Clause 3 is satisfied due to the way *start* was defined.

Clause 4 is satisfied due to the following argument.

For each c in $o \rightarrow c$, the value of c (a module state) will cease as soon as a constructor $c1$ is triggered, where $c1$ satisfies $c1 \rightarrow c$. But then, $c1 \rightsquigarrow o$. Therefore,

$$start(c1) > start(o) > start(c),$$

which means $cease(c) > start(o)$.

Thus all clauses of in situ evaluation are satisfied if *Chain* and *Acyclic* hold. As can be seen, our evaluation rule is basically the applicative order evaluation rule, with the additional orderings imposed by $\rightsquigarrow$ arcs.

Although the necessity of *Chain* and *Acyclic* is easy to see, their sufficience is not that easy to argue informally. In fact prior to the present syntactic characterization, we tried many weaker characterizations which were necessary but were found insufficient.

```
cond { o1(m) : c1(m)
       c2(m) : c3(c2(m))
       true  : o(c2(m)) }
```

is in situ evaluable because:
- `o1(m)` is in situ evaluable;
- `c2(m)` as well as `c1(m)` are in situ evaluable after evaluating `o1(m)`;
- `<o1(m),c2(m)>` is in situ evaluable;
- `true` as well as `c3(c2(m))` are in situ evaluable after `<o1(m),c2(m)`;
- `o(c2(m))` is in situ evaluable after `<o1(m),c2(m),true>`.

Figure 4: An Example Illustrating the Treatment of *cond* Expressions

2.2 Handling *cond* and recursion

2.2.1 *cond*

We attach sequential semantics to *cond*: the consequent corresponding to the first (in lexical order) true antecedent is evaluated and returned as the value of *cond*. We consider *cond* expressions of the form $cond\{p_i : E_i\}$ with i belonging to a certain index set. We formulate the syntactic conditions for *cond* expressions with respect to a canonical representation defined by the following rewrite rules:

$$f(cond\{p_i : E_i\}) \Rightarrow cond\ \{\ p_i : f(E_i)\} \quad (9)$$

$$cond\{cond\{p_i : E_i\} : E\} \Rightarrow cond\ \{\ p_i \wedge E_i : E\} \quad (10)$$

$$cond\{p : cond\{p_i : E_i\}\} \Rightarrow cond\{p \wedge p_i : E_i\} \quad (11)$$

The number of antecedents of a *cond* that would be evaluated at run-time cannot be predicted in general. Therefore in determining the implementability of *cond* expressions, we pessimistically assume that all the antecedents are to be evaluated. *It is this assumption* that renders our syntactic characterization to be sufficient but not necessary.

Once all the antecedents have been evaluated, one of the consequents will be picked for evaluation. However since the antecedents themselves are evaluated sequentially and since the antecedents could themselves use constructor operations on module types, a *cond* expression with N antecedents p_i and N consequents E_i is in-situ evaluable if, for all i in $1..N$,

- the tuple $< p_1, ..., p_{i-1} >$ is in-situ evaluable;
- p_i is in-situ evaluable after having evaluated the tuple $< p_1, ..., p_{i-1} >$;
- E_i is in-situ evaluable after having evaluated the tuple $< p_1, ..., p_i >$.

For each of the above cases, a separate $graph(E)$ is to be constructed, with a $\rightsquigarrow$ arc capturing the "after" relationship. An example of the treatment of *cond* expressions is provided in figure 4.

2.2.2 Recursion

We consider a system of *iterative* recursive definitions of the form

$$F_i(\overline{X}) \Leftarrow if\ p_i(\overline{X})\ then\ f_i(\overline{X})\ else\ F_j(\overline{g_i(X)}),$$

where p_i, f_i, g_i are constructors or observers, $\overline{X}$ is the formal argument vector and $\overline{g_i(X)}$ is the actual-argument vector, and (in general) $i \neq j$. i and j range over an index set, thus giving a system of mutually recursive definitions.

An iterative evaluation of a "call" $F_i(\overline{E})$ essentially involves a sequence of evaluations

$$p_i(\overline{E}), g_i(\overline{E}), p_j(...), g_j(...),\ etc.$$

because all recursive calls are *outermost*. No information (other than the argument vector) is carried across recursive calls. Hence, it suffices to determine the in situ evaluability of the right-hand sides of each of the individual function definitions separately. Thus, the analysis presented so far can be applied to the right-hand sides, ignoring the outermost recursive function call. (A more formal argument is omitted.)

Handling Non-iterative Recursion

We discuss techniques for handling non-iterative recursion and justify why we don't address it in our work.

Consider the expression $< c(M), F(M) >$ where c is a constructor, F is a defined function and M is an object of module type. This is not in iterative form because F, the defined function symbol, is not textually outermost. Depending on the body of F, copying M at the time of the call may or may not be required. We now propose several abstract interpretations to infer this fact (only the first is practical).

1. By analyzing the body of F, infer whether F has a "constructor status" or not; i.e. whether the body of F would update its argument. This is simply determined by observing (i) whether there exists a constructor in the body of F that is applied to the formal argument variable of F; or (ii) whether F calls a function G passing F's formal argument variable as actual argument to G, and G has a constructor status. (A formal definition via structural induction on the syntax is omitted.)

   We can conclude that $< c(M), F(M) >$ is in situ evaluable if F does not have a constructor status. Otherwise the only alternative is to (pessimistically) rule out in situ evaluation, fearing that the constructor application in the body of F to F's argument would destructively update M.

2. Maintain enough information during the process of abstract interpretation to be able to tell ***all possible ways*** in which F would update its arguments. This is clearly undecidable.

3. Explore abstract interpretations that are intermediate in precision (as well as pessimism).

The simplicity afforded by iterative recursion becomes apparent now: defined function symbols don't appear nested anywhere; hence all above complications are avoided and no inter-procedural analysis is required.

In addition, if our technique is applied during the transformation of functional programs into imperative programs with loops, starting with iterative functional programs has the advantage of almost direct mappability into loops.

2.3 The Number of Module Instances to be Allocated

As mentioned in section 1.2, our analysis has thus far assumed that given a function definition:

$$F(X,Y) \Leftarrow body\_F$$

where X and Y are of the same *module type* M, two separate terminal nodes (instances) of M would be available (allocated) at the beginning of the computation of E.

Consider the call $F(E_M, E_M)$ where E_M is an expression of module type M. According to our strategy so far, X and Y would start out by being assigned to two *distinct copies* of the object bearing the value E_M. This would involve creating an extra copy of E_M while compiling the function call. However if we analyze $body\_F$ and determine that the act of identifying the nodes X and Y does not render $body\_F$ in situ unevaluable, then the creation of this extra copy at the time of the call can be avoided.

3 Transformations for Implementability

One way in which our approach can be generalized is to perform a semantic analysis of the program using the algebraic axioms [7] of the abstract data types to transform the program into one that satisfies our syntactic conditions. We have found the following strategies to be useful in performing these transformations. The first strategy can be incorporated as a part of a program optimizer while the second and third are harder because in general they need equational theorem provers ([11], [13]) as well as user-supplied axioms that make explicit the presence of *inverse operations* (explained below).

The first strategy consists of partially evaluating (i.e., *reducing* [18]) a program using the data type axioms as *rewrite rules*. For instance, the expression $ins(q, front(ins(q,v))$, which is not in-situ evaluable, can be transformed into the in-situ evaluable expression

$$if\ empty(q)\ then\ ins(q,v)\ else\ ins(q, front(q))$$

by partially evaluating the former using the following axiom of $front$:

$$front(ins(q,v)) = if\ empty(q)\ then\ v\ else\ front(q) \tag{12}$$

The second strategy is to check if the conflicting subexpressions inside a non in-situ evaluable expression are indeed equivalent. If so, one can be replaced by the other, there by resolving the conflict. For example, the following expression is not in-situ evaluable because the arguments to the *read* operations cause the chain condition to be violated.

However, one of them can be replaced by the other since they are equivalent because *writes* on distinct addresses commute.

$$read(write(write(Mem, 2, 4), 3, 5), read(write(write(Mem, 3, 5), 2, 4), a)) \qquad (13)$$

Our last strategy involves the use of inverse operations. An inverse operation is one which "undoes" the effect of a constructor operation. For instance, the operation *insinv* on *Queue* is the inverse of *ins* if it such that $insinv(ins(q, v)) = q$,. Thus, an inverse operation can be used to obtain the state in which an object was prior to the application of a constructor. This property can be exploited to resolve conflicts in an offending expression.

For example, assuming the operation *insinv* is defined on *Queue* the expression *ins(q,front(ins(q, v))* can be transformed into the following equivalent one which is in situ evaluable:

$$ins(\ insinv(\ ins(q, v)),\ front(\ ins(q, v))).$$

This method is applicable only when one has a prior knowledge of the existence of inverses for the constructors. It is also practically useful only if the inverse operation is a more efficient than copying.

4 Concluding Remarks

We have presented a simple graphical model for determining when destructive updating of abstract data types is possible for a class of functional programs in which recursion is limited to an iterative schema. We have formulated sufficient syntactic conditions on a program under which destructive updating of data types may be introduced without violating the applicative order evaluation semantics of the program. Unlike the work of [10], which employs an abstract interpretation technique, our syntactic approach is fairly cheap computationally although not as general.

We have applied the techniques presented in this paper for the automated synthesis of finite state controllers for hardware from functional specifications [6,5]. In addition, we have successfully transformed (by hand) some simple functional programs into both Ada [17] and Smalltalk [4].

Some restrictive assumptions that we make are: (i) Observers may return only simple types; (ii) Constructors may take only one module argument. It appears that the latter restriction can be lifted by adopting the modeling technique for the *trans* operations [12, p.149]. Other restrictive assumptions that we make actually contribute to the simplicity of our technique.

We thank the referees and Gene Stark for their helpful comments.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles Techniques and Tools.* Addison-Wesley, 1986.

[2] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[3] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Volume 4, Prentice Hall, Englewood Cliffs, N.J., 1978.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[5] Ganesh C. Gopalakrishnan. *From Algebraic Specifications to Correct VLSI Systems.* PhD thesis, State University of New York, December 1986.

[6] Ganesh C. Gopalakrishnan, Mandayam K. Srivas, and David R. Smith. From algebraic specifications to correct vlsi circuits. In *Proceedings of the International Working conference From HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble (IFIP)*, North-Holland, 1986.

[7] John V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[8] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[9] Peter Henderson. *Functional Programming.* Prentice Hall, 1980.

[10] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proc. of the 12th Annual Symposium on the Principles of Prog. Langs.*, pages 300–314, ACM, 1985.

[11] Deepak Kapur and Sivakumar. Rewrite rule laboratory. In *Proc. G.E. Workshop on Rewrite-rule Laboratory, Schenectady*, September 1983.

[12] Alfred Laut. Safe procedural implementations of algebraic types. *Inf. Proc. Letters*, 11(4,5):147–151, December 1980.

[13] P. Lescanne. Computer experiments with the reve term rewriting system generator. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, ACM, January 1983.

[14] Barbara Liskov. *CLU Reference Manual.* Springer-Verlag, 1981.

[15] Zohar Manna. *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.

[16] Prateek Mishra and Robert Keller. Static inference of properties of functional programs. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, pages 235–244, ACM, jan 1984.

[17] Dept. of Defense. *The ADA Language Specification, MIL-STD-1815-A.* American National Standards Institute, 1430 Broadway, NY (212) 642-4900, 1983.

[18] Mandayam K. Srivas and Deepak Kapur. Program transformations and synthesis using rewrite rules. In *TapSoft Conference Proceedings*, Springer-Verlag, LNCS, March 1985.

```
% reverse(Mem, 1, N) reverses the contents of a contiguous block of
locations
% ranging between 1 and N inside a memory module Mem.
% A Functional Description

reverse(Mem, i, N) =
  if i > N div 2  % div returns the quotient of integer division
   then Mem
   else reverse(store(store(Mem, N-i, fetch(Mem,i)), i, fetch(Mem, N-i),
                i+1,N)
```

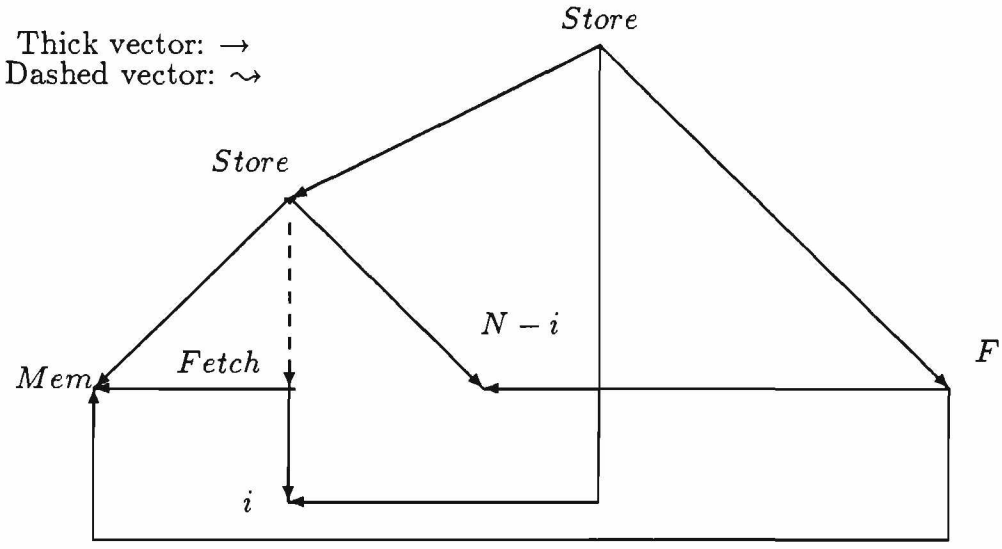

Figure 5: Reversal of a Memory Expressed Functionally

A Appendix

A.1 Example-1: Reversing a Memory Array

A functional program to express the reversal operation on a memory array is shown. Also shown (figures 5,6,7) are the graph to determine in situ evaluability and the imperative code that obeys the in situ evaluation rule written in Smalltalk as well as Ada.

A.2 Example-2: Reversing a Queue

This example illustrates operation invocations inside the conditions of conditional expressions. The following function rotates a queue by moving in order at most n elements from

```
% After our procedure determines the order of op invocation the following
% description can be easily derived.  Note that processes are created for
% running things in parallel.
reverse i N
   i > N div 2 ifTrue:   [^ Mem]
               ifFalse : [[[X <-- Mem fetch i] fork.
                           [Y <-- Mem fetch N-i] fork ]
                           Mem update N-i X.
                           Mem update i Y.
                           Mem reverse i+1 N]
```

Figure 6: Memory Reversal: Incorporating In Situ Evaluation Rule (Smalltalk)

```
reverse( in out Mem: Memory, i,N: NATURAL);

      if i <= N div 2
        then declare
               X,Y : INTEGER;
               begin
                 declare
                   task fetch1;
                   task body begin X := Mem.fetch(i) end;

                   task fetch2
                   task body begin Y := Mem.fetch(N-i) end;
                 begin end;
                 Mem.store(N-i,X);
                 Mem.store(i,Y);
                 reverse(M,i+1,N)
               end
end;
```

Figure 7: Memory Reversal: Incorporating In Situ Evaluation Rule (ADA)

```
rotate(Q,n) =
  { n = 0 or empty(Q) --> Q;
    isodd(front(Q)) --> rem(Q);
    else --> rotate(ins(rem(Q),front(Q)), n-1) }
```

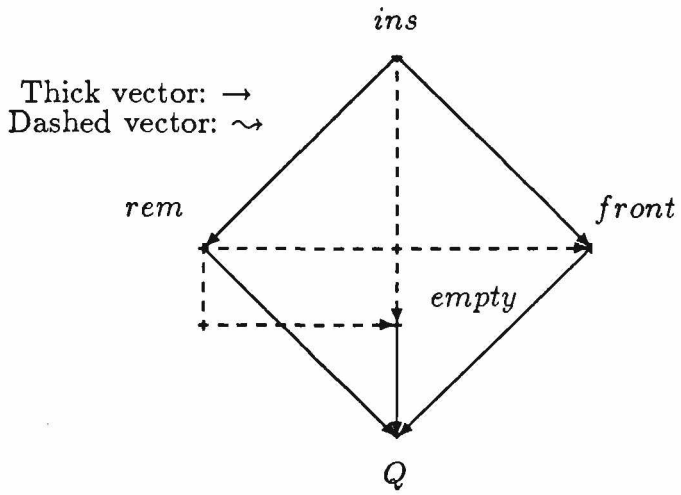

Figure 8: Functional Description of Queue Reversal

the front of the queue to the rear, removing any odd elements in the process. The graph for the conditional is constructed and a $\rightsquigarrow$ arc is introduced from *ins* to *empty* to capture the fact that the first condition of $COND$ is evaluated before *ins* is applied. Another $\rightsquigarrow$ arc is introduced between *rem* and *empty* as well. (Figures 8,9,10.)

```
rotate n
  n eq0 ifTrue:  [^ Q]
        ifFalse: [ Q empty ifTrue:  [^ Q]
                           ifFalse: ((Q front) isodd)
                                        ifTrue:  [^ Q rem]
                                        ifFalse: [X <-- Q front.
                                                  ((Q rem) ins X) rotate
n-1]
                  ]
```

Figure 9: Queue Reversal Incorporating In Situ Evaluation Order (Smalltalk)

```
rotate(in out Q: Queue, in n : INTEGER)
  if n > 0 and not(Q.empty)
   then if isodd(Q.front) then Q.rem
          else declare
                  X : INTEGER
               begin X := Q.front;
                     Q.rem; Q.ins(X);
                     rotate(Q, n-1)
               end
end;
```

Figure 10: Queue Reversal Incorporating In Situ Evaluation Order (ADA)