Electronic Notes in Theoretical Computer Science 5 (1998) *to appear*

Term Rewriting with Operator Evaluation Strategies [1]

Steven Eker

Computer Science Laboratory, SRI International
Menlo Park, CA 94025, USA
eker@csl.sri.com

Abstract

We consider a number of problems which arise when performing term rewriting in algebraic specifications with each operator having an evaluation strategy for its arguments. Each problem is illustrated with an OBJ3 example. We propose a solution developed for the evaluation of functional modules in the Maude interpreter which is based on maintaining an invariant on the term graph by copying subgraphs where necessary. An executable Maude specification of this technique is presented.

1 Introduction

Strategies are very important for languages based on rewriting logic. This issue has been addressed in ELAN[7,1] and Maude[2,3]. In Maude we have strategies at two levels. For system modules we have user-definable strategy languages using the metalevel as well as a fair top-down default strategy. For functional modules we use operator evaluation strategies in the tradition of OBJ2[5] and OBJ3[6], which are also used in CafeOBJ[9,4].

Even when we only consider functional modules, the correct implementation of operator evaluation strategies becomes a subtle issue, due to conditions and subterm sharing [2]. In fact, the existing OBJ3 implementation fails to produce the expected behavior in a number of important cases.

The goal of this paper is to explain precisely a technique for correctly handling operator evaluation strategies in the presence of sharing, and to formally specify this technique using rewriting logic. The formal executable specification models in detail the sharing of subterm as term graphs using the formal representation of graph rewriting in rewriting logic proposed in [8].

[1] Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grant CCR-9633363.

[2] These problems do not arise with user-definable strategy languages because rewrites controlled by these strategies are performed in a non-destructive (but considerably less efficient) way.

1.1 Operator evaluation strategies

Given a term rewriting system R and a term t to be reduced, we are faced with the problem of selecting a rule from R and a redex in t in order to rewrite it. Of course if R is Church-Rosser (CR) and terminating our choice will not affect the final result when no further rewrites can be done. It may affect the efficiency; i.e., the number of rewrites needed to reach the reduced form.

However, when using term rewriting systems for programming it is natural to write rules which only terminate under some particular rewriting strategy. One situation in which this commonly occurs is when the `if_then_else_fi` operator is used; for example:

```
eq fact(N) = if  N == 0 then 1 else N * fact(N - 1) fi .
```

Here, it is natural to reduce the first argument of `if_then_else_fi` , and then rewrite at the top before considering the second and third arguments. Another situation is where one computes with infinite data structures which are evaluated on demand, such as the Sieve of Eratosthenes example in Appendix C.5 of [6].

A simple, intuitive approach that gives the programmer some control over the redex location, but not over the choice of the rule to be used, is to give an *evaluation strategy* to each operator; see [5] and §2.4.5 of [6]. An evaluation strategy for an n-ary operator f consists of an integer sequence $\{0, \ldots, n\}^*$. The idea of such a strategy is that each integer corresponds to a command, and the commands are executed in sequence. A non-zero integer i means: reduce the ith argument up to strategy. The integer 0 means: apply a rule at f if possible. If this succeeds, then the rest of the strategy is ignored as f may well disappear. We can give a semi-formal description of this process in terms of one step rewriting at the top of some term by the following equations.

$$reduce(f(\vec{a})) = eval(f(\vec{a}), strat_f)$$

$$\begin{array}{l}
eval(f(\alpha_1, \ldots, \alpha_m), (p_1, \ldots, p_m)) = \\
\quad \text{if } p_1 = 0 \text{ then} \\
\quad\quad \text{if } f(\alpha_1, \ldots, \alpha_m) \stackrel{top}{\Longrightarrow} \beta \text{ then } reduce(\beta) \\
\quad\quad \text{else } eval(f(\alpha_1, \ldots, \alpha_m), (p_2, \ldots, p_m)) \text{ fi} \\
\quad \text{else} \\
\quad\quad eval(f(\alpha_1, \ldots, \alpha_{p_1-1}, reduce(\alpha_{p_1}), \alpha_{p_1+1}, \ldots, \alpha_m), (p_2, \ldots, p_m)) \\
\quad \text{fi}
\end{array}$$

$$eval(f(\alpha_1, \ldots, \alpha_m), \epsilon) = f(\alpha_1, \ldots, \alpha_m)$$

Here $t \stackrel{top}{\Longrightarrow} t'$ means t rewrites to t' in one step, where the rule matches at the top of t and $strat_f$ is the strategy associated with f. From now on we will assume that every operator is equipped with an evaluation strategy. Constants implicitly get the strategy (0).

```
obj NZ-STRAT-BUG is
  sort Foo .
  op f : Foo -> Foo [strat (0 1)] .
  ops g h : Foo -> Foo [strat (1 0)] .
  ops a b c success : -> Foo .
var X : Foo .
  eq a = b .
  eq f(b) = c .
  eq g(X) = h(X) .
  eq h(c) = success .
endo
```

```
reduce in NZ-STRAT-BUG : g(f(a))
*********** rule
eq a = b
empty substitution
a ---> b
*********** rule
eq g(X) = h(X)
X:Foo --> f(b)
g(f(b)) ---> h(f(b))
rewrites: 2
result Foo: h(f(b))
```

Fig. 1. Example illustrating the invalidation of a reduced flag by a non-zero terminated strategy.

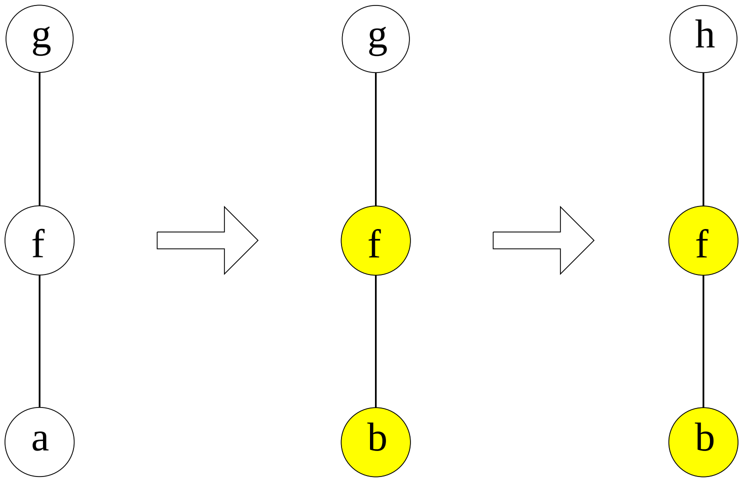

Fig. 2. Term graphs produced by the example in Figure 1.

1.2 Reduced flags

The brute force application of the above algorithm is inefficient. Consider the rewrite rule

```
eq f(X) = g(X) .
```

Now suppose that we apply this to some term $f(\alpha)$ where both f and g have the strategy (1 0). When we apply the rule, α will have already been reduced up to strategy. When we come to reduce $g(\alpha)$ we will try to reduce α again, with the possibility of many failed match attempts. To avoid this waste most practical implementations of term rewriting keep a flag in each node which is set once the node is reduced. The flag is checked before any reduce attempt to avoid wasting time trying to reduce a subterm that is already reduced. In the term graph diagrams that follow, nodes with their *reduced flag* set will be shown as shaded.

2 Non-zero terminated strategies and reduced flags

Consider the CR and terminating rewriting system shown in Figure 1. One would expect `g(f(a))` to reduce to `success`, since `f(a)` rewrites to `f(b)` by

```
obj NON-LINEAR-BUG is
  sort Foo .
  op f : Foo Foo -> Foo .
  ops a b : -> Foo .
  op success : -> Foo .
var X : Foo .
  eq a = b .
  eq f(X, X) = success .
endo
```

```
reduce in NON-LINEAR-BUG : f(a,b)
*********** rule
eq a = b
empty substitution
a ---> b
rewrites: 1
result Foo: f(b,b)
```

Fig. 3. Example illustrating the bad OBJ3 default strategy caused by a non-linear rule.

the 1st rule and then `g(f(b))` rewrites to `h(f(b))` by the 3rd rule. The strategy of `h` is eager and hence its argument must rewrite to `c` by the 2nd rule allowing the 4th rule to apply. In fact, OBJ3 produces the result `h(f(b))`. The problem is that after `a` is rewritten to `b`, the strategy for `f` does not allow a rewrite at the top. So `f(b)` is marked as reduced, and even when it becomes the argument of `h`, which is supposed to eagerly evaluate its argument, the reduced flag is believed and no further rewrites are attempted. See Figure 2

Strategies that do not end in zero have a basic problem in that after the strategy is finished, if the symbol having the strategy was not replaced then that node is not necessarily reduced because a second application of the strategy could cause a rewrite at the top.

2.1 Problems with OBJ3's default strategies

The OBJ3 interpreter will compute a default strategy that does not end in zero for a function symbol f in the case that there exists an i, such that for all lefthand side patterns with f on top, the ith place under that topmost f contains a variable of maximal sort.

The idea is that such places under f need not be evaluated until rewriting at the level of f has failed, since the results of such evaluations seemingly could not affect the outcome of match attempts at the level of f. There are two flaws with this approach. The first is simply that non-left-linear variables are not taken into account in determining the default strategy. Consider the CR and terminating rewriting system shown in Figure 3. OBJ3 computes a strategy (0 1 2) for `f` so that `f(a, b)` reduces to `f(b, b)` rather than to `success` as one might expect.

A more subtle flaw with OBJ3's default strategies is only apparent when there are shared subterms. Consider the CR and terminating rewriting system shown in Figure 4. As we see in the trace, `s(a)` reduces to `f(l(b),b)` rather than to `success` or `f(l(a), b)`. The problem is that the default strategy (1 0 2) computed for `f` fails to take into account that the evaluation of the second argument may change the first argument, allowing a pattern that failed to match at the top previously, to match at the top now.

```
obj SHARE-STRAT-BUG is
  sort Foo .
  op l : Foo -> Foo [strat (0)] .
  op s : Foo -> Foo [strat (0)] .
  op f : Foo Foo -> Foo .
  ops a b : -> Foo .
  op success : -> Foo .
var X : Foo .
  eq a = b .
  eq f(l(b), X) = success .
  eq s(X) = f(l(X), X) .
endo
```

```
reduce in SHARE-STRAT-BUG : s(a)
*********** rule
eq s(X) = f(l(X),X)
X:Foo --> a
s(a) ---> f(l(a),a)
*********** rule
eq a = b
empty substitution
a ---> b
rewrites: 2
result Foo: f(l(b),b)
```

Fig. 4. Example illustrating the bad OBJ3 default strategy exposed by a shared subterm.

2.2 The Maude solution

In the Maude interpreter we avoid all the above problems by taking a simple approach: all strategies are required to end in zero. If the user supplies an explicit strategy which does not end in zero, a zero is added. The default strategy for an n-ary operator is $(1\ 2\ \cdots\ n\ 0)$.

3 Simplifying strategies

From now on we will only consider strategies that end in zero.

Theorem 3.1 *Consider unsorted unconditional rewriting without the use of reduced flags or the sharing of subterms. Suppose all strategies are zero-terminated. Then, for any ground term t, $reduce(reduce(t)) = reduce(t)$; i.e., if some ground term t reduces to t' using the operator evaluation strategies then t' cannot be further rewritten using the given operator evaluation strategies.*

Proof. Let $t' = reduce(t)$. We proceed by induction on the structure of t'. If $t' = c$ for some constant c, then clearly there can be no rewrite rule of the form $c = \alpha$; otherwise, the reduction of t would not have stopped with c. Thus, t' cannot be further rewritten. Now suppose $t' = f(\alpha_1, \ldots, \alpha_n)$, where f has some strategy $strat_f$. Now each place $k = 1, \ldots, n$, is either mentioned in S or it is not.

If place k is mentioned in $strat_f$, then during the reduction of t we would have applied strategy $strat_f$ and reduced the kth place of f, yielding β_k. Since there is no sharing of subterms, subsequent rewrites occurring below other places cannot affect the kth place, therefore, we conclude that $\alpha_k = \beta_k$ and, by the induction hypothesis, any further attempt to rewrite α_k will fail. If place k is not mentioned in $strat_f$, then there can be no attempt to rewrite α_k. Thus, none of subterms in t' can be rewritten.

Finally, since $strat_f$ ends in a zero, during the reduction of t we would have attempted and failed to rewrite t' at the top. Thus, t' cannot be further rewritten. □

This theorem justifies the use of reduced flags. In particular, for any t,

```
obj CONDITIONAL-BUG is
  sorts Foo Bar .
  op l : Foo -> Bar [strat (0)] .
  ops a b c : -> Foo .
  ops success fail : -> Bar .
var X : Foo .
  eq a = b .
  eq l(b) = success .
  cq l(X) = fail if X == c .
  eq l(a) = success .
endo
```

```
reduce in CONDITIONAL-BUG : l(a)
*********** trial #1
cq l(X) = fail if X == c
X:Foo --> a
*********** rule
eq a = b
empty substitution
a ---> b
*********** rule
beq XU == YU =
  (obj_bool$coerce_to_bool
   (term$equational_equal xu yu))
YU:Universal --> c
XU:Universal --> b
b == c ---> false
*********** failure #1
*********** exhausted (#1)
rewrites: 2
result Bar: l(b)
```

Fig. 5. Example illustrating a side effect caused by a conditional equation.

reduce(t) can be flagged as reduced, since further attempts to reduce it will have no effect.

Corollary 3.2 *Under the assumptions of Theorem 3.1, a strategy*

$$s = (p_1 \ldots p_n)$$

where $p_i = p_j \neq 0$ *and* $j > i$*, can be replaced by a strategy*

$$s' = (p_1 \ldots p_{j-1}\ p_{j+1} \ldots p_n)$$

with no effect on the sequence of rewrites for any ground term.

This enables us to simplify strategies by deleting all but the first occurrence of repeated non-zero components.

Corollary 3.3 *Under the assumptions of Theorem 3.1, a strategy*

$$s = (p_1 \ldots p_i\ 0\ 0\ p_{i+3} \ldots p_n)$$

can be replaced by a strategy

$$s' = (p_1 \ldots p_i\ 0\ p_{i+3} \ldots p_n)$$

with no effect on the sequence of rewrites for any ground term.

This enables us to simplify strategies by deleting all but the first of a string of consecutive 0 components.

4 Conditional equations can cause side effects

Consider the CR and terminating conditional rewriting system shown in Figure 5. One would expect `l(a)` to reduce to `success`, using either the 1st and 2nd rules (if the strategy specified for `l` is ignored) or just the 4th rule. In fact OBJ3 produces the result `l(b)`. The problem is that the strategy for `l` does not allow its argument to be evaluated, so instead OBJ3 tries to rewrite `l(a)`

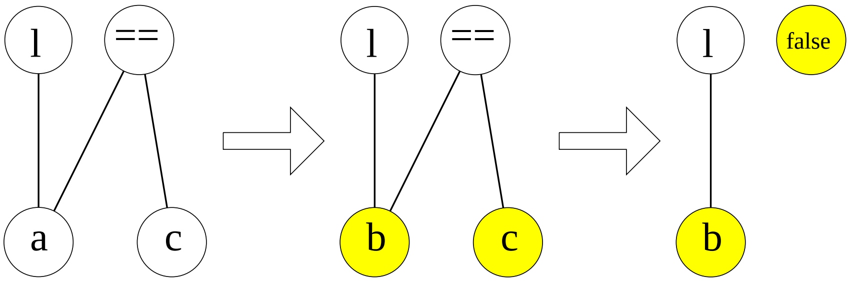

Fig. 6. Term graphs produced by the example in Figure 5.

at the top. The 2nd rule fails to match, the 3rd rule matches but the condition fails. However, evaluating the condition has the side effect of rewriting `l`'s argument, allowing it to match the 2nd rule (which is not tried again) but causing it to fail to match the 4th rule. See Figure 6 Unless special precautions are taken, this problem can occur whenever an unevaluated subterm is shared with a term in the condition, since evaluation of the condition can expose it and then rewrite it.

5 Problems with sharing

Consider the rule

```
eq s(X) = f(X, X) .
```

Most practical implementations of term rewriting will *share* the binding of X in constructing $f(X, X)$ for efficiency. Thus, we are working not with terms but with *term graphs*. If the binding of X is unreduced, then a number of pathological situations arise. In the Maude interpreter, sharing takes place on common subexpressions in the righthand side, and not just on common variables. For example, consider the equation

```
eq s(X) = f(g(X), g(X)) .
```

In the Maude interpreter, sharing (normally) takes place on $g(X)$, and only one node for g is created. Since common subexpressions will typically be unreduced, we will save work by doing the reduction of a common subexpression just once, but the problems detailed below will arise more frequently, and thus it is more important that a solution is found.

5.1 Sharing can cause side effects in subterms

Consider the CR and terminating rewriting system shown in Figure 7. One would expect `s(a)` to reduce to `f(l(a), b)` if the strategy for `l` is observed, or to `f(success, b)` if the strategy for `l` is ignored. Instead, OBJ3 produces the result `f(l(b),b)`. The problem is that the first argument of `f(l(a), a)` has no possible rewrites. Now, the second argument is rewritten to `b`, but, as this subterm is shared with the first argument, the first argument is changed, as shown Figure 8. It is now possible to rewrite the first argument, but since the strategy for `f` has already considered the first argument, this is not attempted.

```
obj SIDE-EFFECT-BUG is
  sort Foo .
  ops s l : Foo -> Foo [strat (0)] .
  op f : Foo Foo -> Foo [strat (1 2 0)] .
  ops a b success : -> Foo .
var X : Foo .
  eq a = b .
  eq l(b) = success .
  eq s(X) = f(l(X), X).
endo
```

```
reduce in SIDE-EFFECT-BUG : s(a)
*********** rule
eq s(X) = f(l(X),X)
X:Foo --> a
s(a) ---> f(l(a),a)
*********** rule
eq a = b
empty substitution
a ---> b
rewrites: 2
result Foo: f(l(b),b)
```

Fig. 7. Example illustrating a side effect caused by a shared subterm.

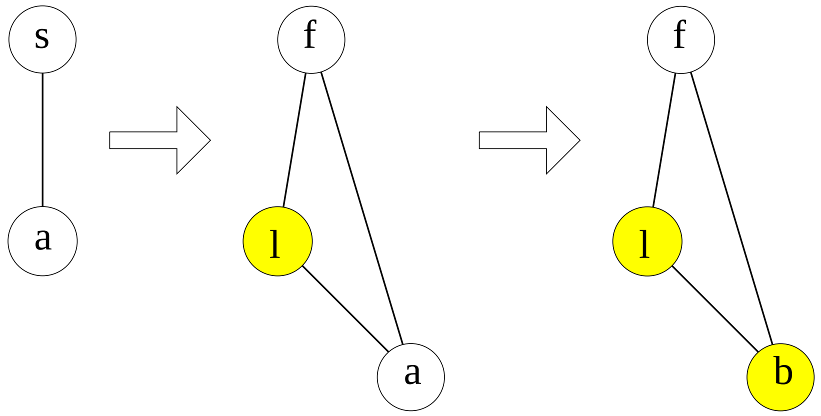

Fig. 8. Term graphs produced by the example in Figure 7.

```
obj REDUCE-BUG is
  sort Foo .
  ops s l : Foo -> Foo [strat (0)] .
  op f : Foo Foo -> Foo [strat (1 2 0)] .
  op k : Foo -> Foo .
  ops a b c success : -> Foo .
var X : Foo .
  eq a = b .
  eq l(b) = c .
  eq f(X, b) = k(X) .
  eq k(c) = success .
  eq s(X) = f(l(X), X).
endo
```

```
reduce in REDUCE-BUG : s(a)
*********** rule
eq s(X) = f(l(X),X)
X:Foo --> a
s(a) ---> f(l(a),a)
*********** rule
eq a = b
empty substitution
a ---> b
*********** rule
eq f(X,b) = k(X)
X:Foo --> l(b)
f(l(b),b) ---> k(l(b))
rewrites: 3
result Foo: k(l(b))
```

Fig. 9. Example illustrating the invalidation of a reduced flag by sharing.

5.2 Sharing can invalidate reduced flags

The previous example not only missed a rewrite because the first argument of f was not reconsidered after a successful rewrite of the second argument; it also set the reduce flag for the first argument and this flag was invalidated by the rewrite of the second argument. To illustrate this, consider the modified rewriting system shown in Figure 9. As we see in the trace, `s(a)` reduces to `k(l(b))` rather than to `k(l(a))` or `success`. The problem is that the subterm

```
obj SORT-BUG is
  sorts Foo Bar Baz .
  subsort Bar < Foo .
  op l : Foo -> Foo [strat (0)] .
  op l : Bar -> Bar [strat (0)] .
  op s : Foo -> Baz [strat (0)] .
  op f : Foo Foo -> Baz [strat (1 2 0)] .
  op a : -> Foo .
  op b : -> Bar .
  op success : -> Baz .
var X : Bar .
var Y : Foo .
  eq a = b .
  eq f(X, Y) = success .
  eq s(Y) = f(l(Y), Y) .
endo
```

```
reduce in SORT-BUG : s(a)
*********** rule
eq s(Y) = f(l(Y),Y)
Y:Foo --> a
s(a) ---> f(l(a),a)
*********** rule
eq a = b
empty substitution
a ---> b
rewrites: 2
result Baz: f(l(b),b)
```

Fig. 10. Example illustrating the invalidation of sort information by sharing.

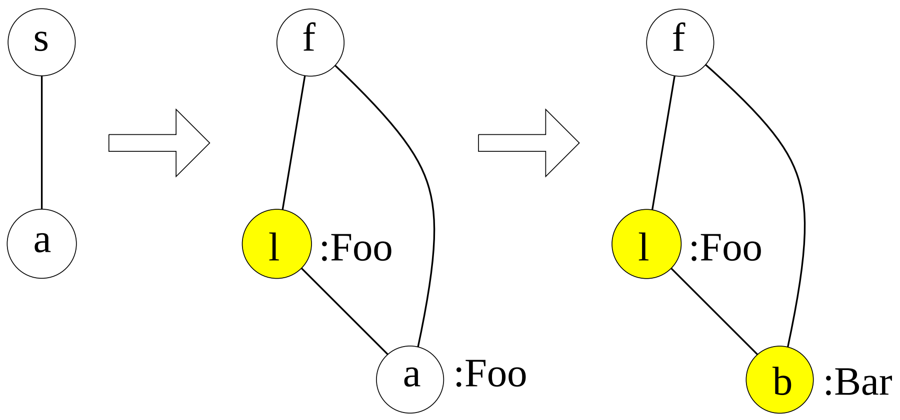

Fig. 11. Term graphs produced by the example in Figure 10.

`l(a)` has its reduced flag set, so that when it becomes `l(b)` via the sharing of `a`, no more rewriting is attempted, even when it becomes the argument of `k` which has an eager strategy.

5.3 Sharing can invalidate sort information

Consider the CR and terminating rewriting system shown in Figure 10. One would expect `s(a)` to rewrite to `f(l(a), b)` if the strategy for `l` is observed or to `success` if the strategy for `l` is ignored. Instead, OBJ3 produces the result `f(l(b),b)`. The problem is that after `s(a)` rewrites to `f(l(a), a)` the subterm `l(a)` is reduced up to strategy and the sort `Foo` is computed for it. When the second argument of `f` rewrites, this `l(a)` becomes `l(b)` because of the sharing of `a` and now its sort is incorrect. See Figure 11. This invalid sort prevents `f(l(b), b)` from matching.

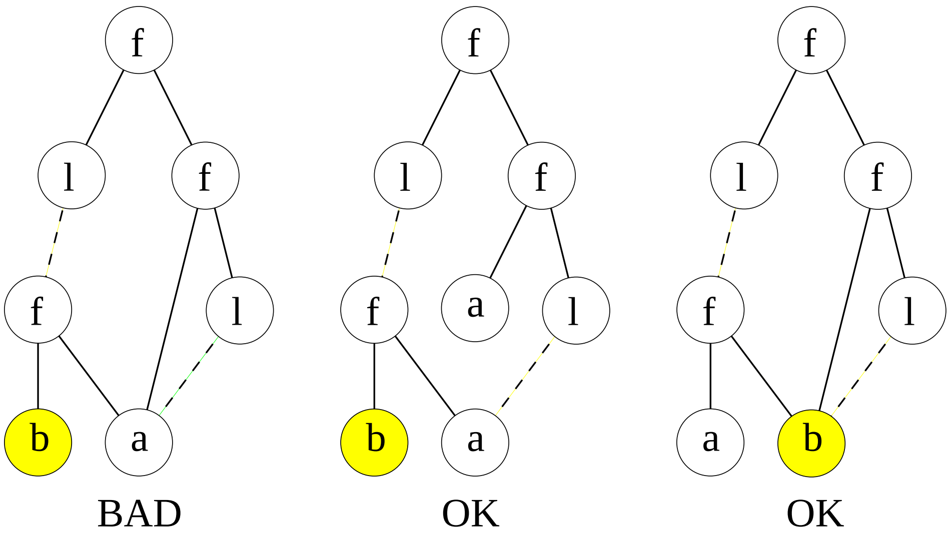

Fig. 12. Term graphs illustrating the invariant.

6 An efficient solution

The problems in the previous two sections arose because we did not observe the user's strategies strictly. Instead, we assumed that it could not hurt to eagerly evaluate some lazy term since it had to be evaluated anyway in a separate branch of the computation—either a condition or a sibling subterm. As we saw above, ignoring the user's strategy can invalidate previously computed sort information and reduced flags, and even the sequential consideration of rules.

The solution implemented in Maude is based on a strict observation of the strategy given by the user. Where necessary, subterms (really term graphs) are copied to render the effects of condition evaluation and sharing invisible.

Definition 6.1 Given an n-ary function symbol f with strategy $strat_f$ we say the ith place under f is *eager* if it occurs in $strat_f$ before the leftmost occurrence of 0. Otherwise, it is *lazy*.

The intuition is that an argument in an eager place of f will be evaluated before any attempt is made to rewrite at f. From now on, arcs exiting from lazy places will be shown as dashed lines in our term graph diagrams.

Definition 6.2 Given a term graph, a path from the root to some node is said to be *eager* if all the edges on the path exit from eager places. Otherwise, it is *lazy*.

In all operations on term graphs we maintain the following invariant on the term graph: No node is reachable from the root by both an eager path and a lazy path, unless it is reduced up to strategy. The intuition is that nodes reachable by lazy paths will never be rewritten: either they are already reduced up to strategy, or there is no eager path that could force their evaluation. Figure 12 shows examples of term graphs that break and maintain this invariant.

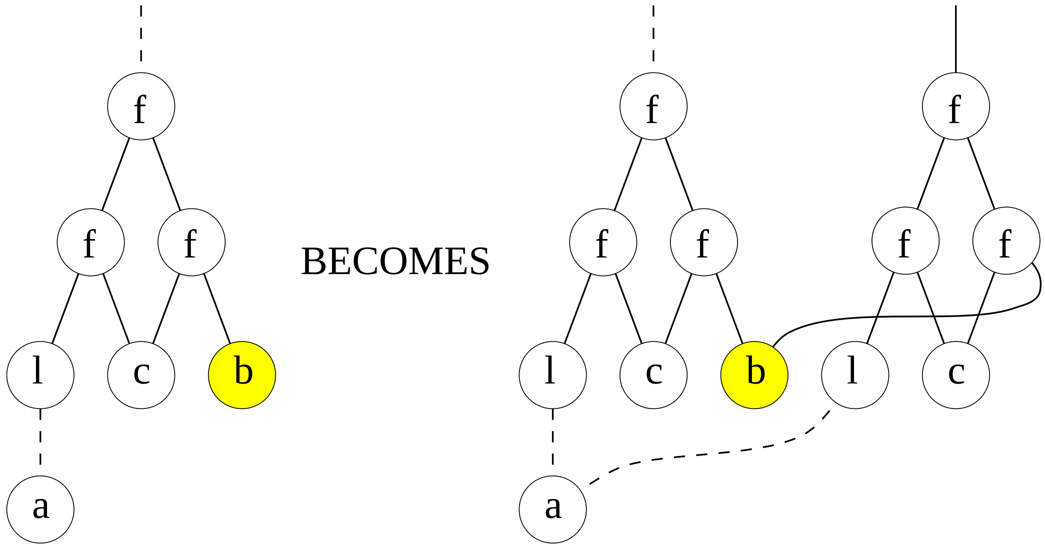

Fig. 13. An example of an eager copy of a term graph.

We will consider lefthand and righthand sides of rewrite rules as trees.

Definition 6.3 A node in a term tree is *eager* if the path to it from the root contains only edges that exit from eager places. Otherwise, it is *lazy*.

Definition 6.4 A variable X in a lefthand side is said to be *eager* if it occurs in at least one eager node. Otherwise, it is *lazy*.

The intuition is that if an eager variable X gets bound to a term t during matching, then t must already be reduced.

Definition 6.5 The *eager copy* of a portion of a term graph starting at a node N with n-ary operator f and descendents $D_1, \ldots, D_n$ is given by the following rules.

(i) If N is reduced then N.

(ii) Otherwise, if a copy N' of N has already been made (via some other path) then N'.

(iii) Otherwise, create a new node N' with the operator f and no valid sort information. For each edge from N to D_i the corresponding edge from N' goes to D_i if the ith place of f is lazy, or to an eager copy of D_i otherwise.

Figure 13 shows an example of an eager copy performed on a term graph under a lazy path.

In sharing common subterms and in using the bindings of variables in the righthand side (or condition) we enforce the following rules:

(i) Two common (non-variable) subterms in a righthand side (or condition) can only be shared if they both occur at eager positions, or both occur at lazy positions.

(ii) A binding to an eager variable X can be used at both eager and lazy occurrences of X in the righthand side (or condition).

(iii) A binding to a lazy variable Y can only be used at lazy occurrences of Y in the righthand side (or condition). An eager copy of the binding of Y must be made to be used at eager occurrences of Y.

Furthermore, for places which are lazy, but which can still be evaluated, for example, place 1 in f with a strategy (0 1 0), an eager copy of the argument in that place must be made and evaluated in place of the original.

7 Modelling this technique with rewriting logic

There are a number of ways that term graph rewriting could be encoded in rewriting logic. We chose the approach described in [8], whereby rewrite rules are transformed into graph rewrite rules which operate on an associative-commutative soup of graph nodes.

We start by specifying a strategy as a sequence of integers. To simplify our module we make a small transformation on strategies: A strategy such as (1 0 2 3 0) becomes `(1 0 c(2) c(3) 0)` to record the fact that places 2 and 3 are lazy but might be evaluated.

```
sorts Strategy Symbol .
subsort MachineInt < Strategy .
ops reduced idle : -> Strategy .
op c : MachineInt -> Strategy .
op __ : Strategy Strategy -> Strategy [prec 7 assoc id: reduced] .
op getStrategy : Symbol -> Strategy .
```

Our idea is that each node in the term graph will hold a strategy which records the current state of its evaluation. We have two special strategies, `idle`, which indicates that a node cannot yet be evaluated, and `reduced`, which indicates that a node is reduced up to strategy. Strategies stored in nodes will be "used up" as rewriting proceeds, and therefore it is natural to have `reduced` as the identity of the strategy list constructor.

Each node will have an unique address given by an integer. An argument list then just becomes a list of integers.

```
sorts ArgList .
subsort MachineInt < ArgList .
op nil : -> ArgList .
op _,_ : ArgList ArgList -> ArgList [assoc id: nil] .
op _th of_ : MachineInt ArgList -> MachineInt .
```

It is useful to have the `_th of_` operator to extract the kth argument when it exists. In order to keep track of whether a node has been copied and if so where its copy is we define copy pointers.

```
sort CopyPointer .
subsort MachineInt < CopyPointer .
op notCopied : -> CopyPointer .
```

A node is a 5-tuple, containing an address, a symbol, a strategy, a copy pointer and an argument list.

```
sort Node .
op [_,_,_,_,_] : MachineInt Symbol Strategy CopyPointer ArgList -> Node .
```

A term graph consists of a "next free address" index and an associative-commutative soup of nodes.

```
sorts TermGraph Soup .
subsort Node < Soup .
op null : -> Soup .
op __ : Soup Soup -> Soup [assoc comm id: null] .
op {_|_} : MachineInt Soup -> TermGraph .
```

We also need operations to perform the eager copy operation, on single arguments and on all the arguments in eager places in a given argument list.

```
op eagerCopy : MachineInt -> MachineInt .
op copyEagerArgs : Symbol ArgList -> ArgList .
```

Reduction up to strategy is defined by three equations and a rule. The first two equations deal with the case that the current strategy in a node `N` requires a given argument to be reduced. If the node `N2` corresponding to that argument is already reduced the first equation skips that step of the strategy. Otherwise, if the node `N2` is idle, the second equation replaces the `idle` strategy with that determined by `N2`'s symbol. The third equation deals with the case where we need to evaluate an argument in a lazy position, by first making an eager copy of it. The rule handles the case where we tried to write at the top and failed and so continue by just discarding the `0` in the strategy. The reason this is a rule and not an equation is that it must only be used when no other rewrites are possible. In Maude this strategy for applying a rule could be specified using the meta-level, but in practice we can rely on the default operational semantics of the Maude interpreter which applies rules only when no equations apply.

```
ceq [N, S, (NzM T), CP, AL] [N2, S2, reduced, CP2, AL2] =
    [N, S, T, CP, AL]       [N2, S2, reduced, CP2, AL2]
    if NzM th of AL == N2 .

ceq [N, S, (NzM T), CP, AL] [N2, S2, idle, CP2, AL2] =
    [N, S, (NzM T), CP, AL] [N2, S2, getStrategy(S2), CP2, AL2]
    if NzM th of AL == N2 .

ceq [N, S, (c(NzM), T), CP, (AL, N2, AL2)] =
    [N, S, (NzM T), CP, (AL, eagerCopy(N2), AL2)]
    if NzM th of (AL, N2, AL2) == N2 .

rl [failedAtTop] : [N, S, (0 T), CP, AL] => [N, S, T, CP, AL] .
```

Just as in our earlier informal description of the eager copy operation there are three cases in the formal model, each handled by an equation. The final case is the interesting one, where the node `M` which we want an eager copy of has neither been reduced or previously copied. Here we actually make a copy at the next free address, and use `copyEagerArgs` (see appendix for definition) to propagate the eager copy operation to any arguments in eager places.

```
eq {N | [N2, S, T, CP, (AL, eagerCopy(M), AL2)]
        [M, S2, reduced, CP2, AL3] Rest} =
   {N | [N2, S, T, CP, (AL, M, AL2)]
        [M, S2, reduced, CP2, AL3] Rest} .
```

```
eq {N | [N2, S, T, CP, (AL, eagerCopy(M), AL2)]
        [M, S2, idle, M2, AL3] Rest} =
   {N | [N2, S, T, CP, (AL, M2, AL2)]
        [M, S2, idle, M2, AL3] Rest} .

eq {N     | [N2, S, T, CP, (AL, eagerCopy(M), AL2)]
            [M, S2, idle, notCopied, AL3] Rest} =
   {N + 1 | [N2, S, T, CP, (AL, N, AL2)]
            [M, S2, idle, N, AL3]
            [N, S2, idle, notCopied, copyEagerArgs(S2, AL3)] Rest} .
```

Now we present the transformed term rewriting system from Figure 7. Each original equation become an equation which acts on the whole term graph by finding a node whose current strategy requires a rewrite at the top and whose symbol and arguments match those required by the original lefthand side pattern.[3] The most interesting equation is the last one, which corresponds to `eq s(X) = f(l(X), X)` in the original system. Here we see that the second use of X in the righthand side requires an eager copy.

```
ops s l f a b success : -> Symbol .
eq getStrategy(s) = 0 .
eq getStrategy(l) = 0 .
eq getStrategy(f) = (1 2 0) .
eq getStrategy(a) = 0 .
eq getStrategy(b) = 0 .
eq getStrategy(success) = 0 .

vars A B C D : MachineInt .

eq {N | [A, a, (0 T), CP, nil] Rest} =
   {N | [A, b, getStrategy(b), CP, nil] Rest} .

eq {N | [A, l, (0 T), CP, B]
        [B, b, Y, CP2, nil] Rest} =
   {N | [A, success, getStrategy(success), CP, B]
        [B, b, Y, CP2, nil] Rest} .

eq {N     | [A, s, (0 T), CP, X] Rest} =
   {N + 1 | [A, f, getStrategy(f), CP, (N, eagerCopy(X))]
            [N, l, idle, notCopied, X] Rest} .
```

Finally we can run the model on an encoding of `s(a)` to get:

```
Maude> rewrite {2 | [0, s, getStrategy(s), notCopied, 1]
                    [1, a, idle, notCopied, nil]} .

rewrites: 82 in 0ms cpu (29ms real) (~ rewrites/second)
result TermGraph: {4 | [0, f, reduced, notCopied, (2, 3)]
```

[3] The astute reader will realize that there are problems with this simple translation scheme for equations whose lefthand sides contain unifiable subterms or whose righthand sides are bare variables. The former problem could be handled by adding extra equations whose lefthand sides contain all the alternative possible sharings of unifiable subterms. The latter problem could be handle by defining an auxiliary function which overwrites the symbol, strategy and arguments of one node with those of the node at a given address.

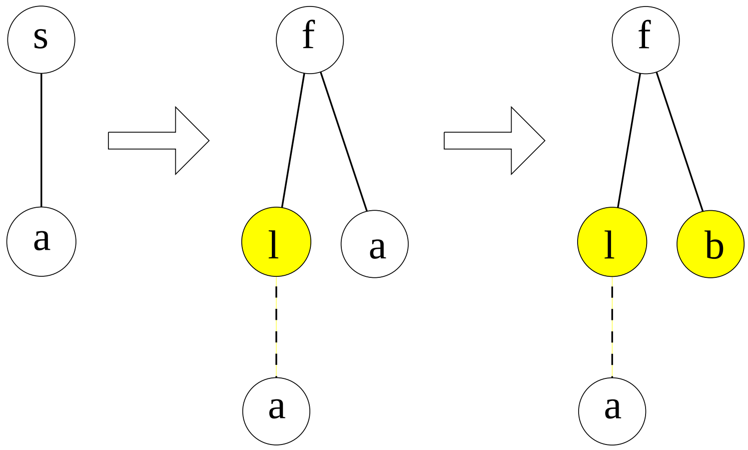

Fig. 14. Term graphs illustrating correct execution of the example in Figure 7.

```
[1, a, idle, 3, nil]
[2, l, reduced, notCopied, 1]
[3, b, reduced, notCopied, nil]}
```

Figure 14 shows the corresponding term graphs.

8 Concluding remarks

We have examined a number of problems that arise when performing term rewriting with operator evaluation strategies. We have presented and formally specified a solution which seems fairly efficient in the average case: For rewrites in which the top operator has the default eager evaluation strategy our solution incurs only the cost of detecting this (typically 3 machine instructions, where the average interpreted rewrite costs several hundred machine instructions). When the top operator has a more complex evaluation strategy some (possibly unnecessary) copying may take place. The cost of detecting whether the copying is really necessary seems prohibitively high.

While developing this solution several alternative techniques were considered. One such technique relied on keeping reference counts and avoiding overwriting shared nodes. Apart from the cost associated with maintaining reference counts (though this might be shared with garbage collection) this approach would rule out sharing the evaluation of common subexpressions. Another technique would be to allow eager evaluation of shared lazy subterms but maintain back pointers to all parents of a given node so if it were updated its parents could be updated. This changes there semantics of operator evaluation strategies somewhat and incurs the considerable expense of maintaining a list of back pointers in each node.

Acknowledgements

I thank José Meseguer for his advice on the structure of this paper and his help with the Maude specification. I also thank Patrick Lincoln and Francisco Durán for their detailed comments on earlier drafts.

References

[1] P. Borovansky, C. Kirchner, and H. Kirchner. Strategies and Rewriting in ELAN. Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction, ed. B. Gramlich and H. Kirchner, 1997.

[2] M. Clavel, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[3] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. Proc. First Intl. Workshop on Rewriting Logic and its Applications, ed. J. Meseguer, Electronic Notes in Theoretical Computer Science, 4, 1996.

[4] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. Proc. of the Kunming International CASE Symposium, Kunming, China, November 1994.

[5] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. pp 52–66, Proc. of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, 1985.

[6] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992.

[7] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. pp 133–160, Principles and Practice of Constraint Programming: The Newport Papers, ed. V. Saraswat and P. van Hentryck, MIT Press, 1995.

[8] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. pp 331–372, Proc. CONCUR'96, Pisa, Springer LNCS 1119, 1996.

[9] A. T. Nakagawa, T. Sawada, K. Futatsugi. CafeOBJ Manual. Technical Report, JAIST/SRA, 1997.

Appendix

```
mod TERM-GRAPH is
  protecting MACHINE-INT .

  sorts Strategy Symbol .
  subsort MachineInt < Strategy .
  ops reduced idle : -> Strategy .
  op c : MachineInt -> Strategy .
  op __ : Strategy Strategy -> Strategy [prec 7 assoc id: reduced] .
  op getStrategy : Symbol -> Strategy .

  sorts ArgList .
  subsort MachineInt < ArgList .
  op nil : -> ArgList .
  op _,_ : ArgList ArgList -> ArgList [assoc id: nil] .
  op _th of_ : MachineInt ArgList -> MachineInt .
```

```
  vars N N2 M M2 : MachineInt .
  vars AL AL2 AL3 : ArgList .

  eq 1 th of (M, AL) = M .
  ceq N th of (M, AL) = (N - 1) th of AL if N > 1 .

  sort CopyPointer .
  subsort MachineInt < CopyPointer .
  op notCopied : -> CopyPointer .

  sorts Node Soup .
  subsort Node < Soup .
  op [_,_,_,_,_] : MachineInt Symbol Strategy CopyPointer ArgList -> Node .
  op null : -> Soup .
  op __ : Soup Soup -> Soup [assoc comm id: null] .

  sort TermGraph .
  op {_|_} : MachineInt Soup -> TermGraph .
  op eagerCopy : MachineInt -> MachineInt .

***
***  Strategy advancement.
***
  var NzM : NzMachineInt .
  vars X Y : Strategy .
  vars S S2 : Symbol .
  vars CP CP2 : CopyPointer .

  ceq [N, S, (NzM X), CP, AL] [N2, S2, reduced, CP2, AL2] =
      [N, S, X, CP, AL]       [N2, S2, reduced, CP2, AL2]
      if NzM th of AL == N2 .

  ceq [N, S, (NzM X), CP, AL] [N2, S2, idle, CP2, AL2] =
      [N, S, (NzM X), CP, AL] [N2, S2, getStrategy(S2), CP2, AL2]
      if NzM th of AL == N2 .

  ceq [N, S, (c(NzM), X), CP, (AL, N2, AL2)] =
      [N, S, (NzM X), CP, (AL, eagerCopy(N2), AL2)]
      if NzM th of (AL, N2, AL2) == N2 .

  rl [failedAtTop] : [N, S, (0 X), CP, AL] => [N, S, X, CP, AL] .

***
*** Cloning a node for a collapse rule.
***
  op clone : MachineInt MachineInt -> Node .

  vars Dest Source : MachineInt .

  eq [Source, S, X, CP, AL] clone(Dest, Source) =
     [Source, S, X, CP, AL] [Dest, S, getStrategy(S), CP, AL] .

***
***  Eager copy.
***
```

```
  op copyEagerArgs : Symbol ArgList -> ArgList .
  op copyEagerArgs2 : Strategy MachineInt ArgList -> ArgList .
  op eagerArgument : Strategy MachineInt -> Bool .
  eq copyEagerArgs(S, AL) = copyEagerArgs2(getStrategy(S), 1, AL) .
  eq copyEagerArgs2(X, N, nil) = nil .
  eq copyEagerArgs2(X, N, (M, AL)) =
        if eagerArgument(X, N) then
            (eagerCopy(M), copyEagerArgs2(X, N + 1, AL))
        else
            (M, copyEagerArgs2(X, N + 1, AL))
        fi .
  eq eagerArgument((N X), N) = true .
  ceq eagerArgument((NzM X), N) = eagerArgument(X, N) if NzM =/= N .
  eq eagerArgument((0 X), N) = false .

  var User : MachineInt .
  var Rest : Soup .

  *** eager copy reduced case
  eq {N | [N2, S, X, CP, (AL, eagerCopy(M), AL2)]
          [M, S2, reduced, CP2, AL3] Rest} =
     {N | [N2, S, X, CP, (AL, M, AL2)]
          [M, S2, reduced, CP2, AL3] Rest} .

  *** eager copy already copied case
  eq {N | [N2, S, X, CP, (AL, eagerCopy(M), AL2)]
          [M, S2, idle, M2, AL3] Rest} =
     {N | [N2, S, X, CP, (AL, M2, AL2)]
          [M, S2, idle, M2, AL3] Rest} .

  *** eager copy make copy case
  eq {N     | [N2, S, X, CP, (AL, eagerCopy(M), AL2)]
              [M, S2, idle, notCopied, AL3] Rest} =
     {N + 1 | [N2, S, X, CP, (AL, N, AL2)]
              [M, S2, idle, N, AL3]
              [N, S2, idle, notCopied, copyEagerArgs(S2, AL3)] Rest} .

***
***  Transformed term rewriting system.
***
  ops s l f a b success : -> Symbol .
  eq getStrategy(s) = 0 .
  eq getStrategy(l) = 0 .
  eq getStrategy(f) = (1 2 0) .
  eq getStrategy(a) = 0 .
  eq getStrategy(b) = 0 .
  eq getStrategy(success) = 0 .

  vars A B C D : MachineInt .

  eq {N | [A, a, (0 X), CP, nil] Rest} =
     {N | [A, b, getStrategy(b), CP, nil] Rest} .

  eq {N | [A, l, (0 X), CP, B]
          [B, b, Y, CP2, nil] Rest} =
     {N | [A, success, getStrategy(success), CP, B]
```

```
          [B, b, Y, CP2, nil] Rest} .

  eq {N      | [A, s, (0 X), CP, B] Rest} =
     {N + 1 | [A, f, getStrategy(f), CP, (N, eagerCopy(B))]
               [N, l, idle, notCopied, B] Rest} .
endm

rew {2 | [0, s, getStrategy(s), notCopied, 1]
         [1, a, idle, notCopied, nil]} .
```