Operational Semantics of Rewriting with the On-demand Evaluation Strategy

Kazuhiro Ogata and Kokichi Futatsugi
Graduate School of Information Science
JAIST
Tatsunokuchi, Ishikawa 923-1292, JAPAN
{ogata, kokichi}@jaist.ac.jp

Keywords

operational semantics, reduction strategy, the E-strategy, CafeOBJ

ABSTRACT

The on-demand evaluation strategy (abbr. the on-demand E-strategy) is an extension of the evaluation strategy (abbr. the E-strategy) initiated by OBJ2. The strategy removes the restriction that the E-strategy imposes on constructing rewrite rules: if non-variable terms are put on lazy positions in the left sides, some terms cannot be rewritten as intended. We have written the operational semantics of rewriting with the on-demand E-strategy in CafeOBJ so that we can deeply understand rewriting with the on-demand E-strategy. The operational semantics can be used to observe the dynamic behavior of rewriting with the on-demand E-strategy thanks to the executability of CafeOBJ. A hint about the use of the on-demand E-strategy is given as well.

1. INTRODUCTION

A reduction strategy is a function that takes a set of rewrite rules and a ground term as arguments, and prescribes which redex in the term has to be rewritten next. Although lazy evaluation is fascinating because it has a better termination behavior than eager evaluation, pure lazy evaluation is not efficiently implementable. Therefore some efficiently implementable compromises between lazy and eager evaluation have been proposed. The on-demand evaluation strategy (abbr. the on-demand E-strategy) is one of them, which is used in CafeOBJ [2]. The on-demand E-strategy is an extension of the evaluation strategy (abbr. the E-strategy) initiated by OBJ2 [4]. The E-strategy not only simulates a variant of lazy evaluation such as the functional strategy [10], but also is flexible because it can control the order in which terms are rewritten by giving a local strategy to each operator (or function symbol). However, it imposes a restriction on constructing rewrite rules: if non-variable terms are put on lazy positions in the left sides, some terms cannot be rewritten as intended. The on-demand E-strategy can remove this restriction of the E-strategy.

We have written the operational semantics of rewriting with the on-demand E-strategy so that we can deeply understand it. The operational semantics has been written in CafeOBJ [2], an executable algebraic specification language. The reason why we used CafeOBJ to write the operational semantics is that we can confirm the partial correctness of the specification with the CafeOBJ system and can also observe the dynamic behavior of rewriting with the on-demand E-strategy using the operational semantics as an interpreter. We believe that the formal operational semantics of rewriting with the strategy should be useful not only for making sure of what rewriting with the strategy performs, but also as a formal specification that can be used when implementing rewriting with the strategy.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to CafeOBJ. Section 3 describes the E-strategy and the operational semantics of the E-strategy in CafeOBJ. In Sect. 4, the restriction imposed by the E-strategy and the on-demand E-strategy that removes the restriction are first mentioned, and then an example that needs the on-demand E-strategy is given. After that the operational semantics of rewriting with the on-demand E-strategy in CafeOBJ is described and the dynamic behavior of rewriting with the strategy is observed. Section 5 presents a hint about specifying local strategies. Section 6 discusses the related work. Finally, Section 7 gives a conclusion.

We suppose the reader familiar with the basic concepts of term rewriting systems [1] (abbr. TRSs).

2. ALGEBRAIC SPECIFICATION LANGUAGE CAFEOBJ

CafeOBJ [2] is descended from OBJ [4; 6], probably the most famous algebraic specification language. Although CafeOBJ provides many fascinating features and functionalities, we here take up only a few of them that are needed for writing the operational semantics of rewriting. One of them is the powerful module system inherited from OBJ. Modules can have parameters and import other modules. We give a parameterized module *LIST* as an example.

Example 1 (A parameterized module LIST).

mod! *LIST* (*DAT* :: *TRIV*) {
  pr(*NAT*)

```
  [List]
  op nil : -> List
  op __ : Elt.DAT List -> List {r-assoc}
  op _++_ : List List -> List
  op take : List Nat -> Elt.DAT
  op replace : List Nat Elt.DAT -> List
  vars E E1 : Elt.DAT
  vars L L1 : List
  eq nil ++ L1 = L1 .
  eq (E L) ++ L1 = E (L ++ L1) .
  eq take(E L,0) = E .
  eq take(E L,X:NzNat) = take(L,sd(X,1)) .
  eq replace(E L,0,E1) = E1 L .
  eq replace(E L,X:NzNat,E1)
           = E replace(L,sd(X,1),E1) .
}
```

A module (declaration) begins with the keyword **mod!** or **mod*** (which corresponds to tight or loose denotation), and has its name (*LIST* in this example) and a list of parameters if exist ((*DAT* :: *TRIV*) in this example) after the keyword. A parameter consists of a parameter name (*DAT* in this example) and a module name (*TRIV*[1] in this example). The module name describes requirements that the actual parameters must meet. The module body is enclosed by braces. The module body consists of some declarations such as import, sort, operator, variable and/or equation declarations. The module *LIST* imports the built-in module *NAT* with **protecting** (abbr. **pr**) mode that requires all the intended models of imported modules (*NAT* in this example) be preserved as they are (i.e. no junk and no confusion). There are two more importation modes: **extending** (abbr. **ex**) and **using** (abbr. **us**) modes. The **ex** mode allows the models of imported modules to be inflated, but does not allow them to be collapsed (i.e. no junk). The **us** mode imposes nothing on the models of imported modules. The built-in module *BOOL*, in which *true*, *false* and some logical operators are declared, is implicitly imported into any module by default. *LIST* declares a new sort *List*. Sorts in algebraic specification languages may correspond to types in programming languages.
Operator declarations begin with **op** or **ops** and have the name, the sorts of the arguments and the sort of the result. They may have some attributes such as **r-assoc** meaning that the operator is right associative. Some operators such as *nil* have no arguments and are called constants. CafeOBJ makes it possible to declare not only standard operators such as *f* and *g* in *f*(*g*(1),2), but also mixfix operators such as _++_ (i.e. infix ones) and __ (i.e. juxtaposition ones). Underbars reserve the places where arguments are inserted. Variable declarations begin with **var** or **vars**, and have the name and its sort. Sorts may be quantified by modules such as *Elt.DAT*. Variables may be declared in equation declarations such as *X*:*NzNat*[2].
Equation declarations begin with **eq** or **ceq** and end with a full stop. Conditional equations are declared with **ceq**.
An instance of parameterised modules is created by binding actual parameters to formals. The process of binding is called instantiation. The result of instantiation is a new module. For example, when *LIST* is instantiated by binding *NAT* to *TRIV*, we write *LIST*(*NAT*). When instantiating the module *LIST*, it is also possible to rename the sort *List* another such as *NatList* by writing *LIST*(*NAT*)*{**sort** *List* -> *NatList*}.
CafeOBJ specifications may be executed by regarding equations as left-to-right rewrite rules by a rewrite engine. Thanks to the executability, we can observe the dynamic behavior of rewriting with the on-demand E-strategy using the formal operational semantics in CafeOBJ.

[1]*TRIV* is a built-in module in which only one sort *Elt* is declared.
[2]The sort *NzNat* is declared in the module *NAT-VALUE* imported into *NAT* and stands for non-zero natural numbers.

3. THE E-STRATEGY

The E-strategy is a reduction strategy initiated by OBJ2 [4]. It not only simulates a variant of lazy evaluation such as the functional strategy [10], but also is flexible because it can control the order in which terms are rewritten by giving a local strategy to each operator (or function symbol). A local strategy given to a function symbol f indicates the order in which terms such as $f(t_1,\ldots,t_n)$ that each have the function symbol f at the head of them are evaluated. The order is prescribed with a list of integers ranging from zero through the arity (the number of the arguments) of the function symbol. A term $f(t_1,\ldots,t_n)$ is evaluated according to the local strategy of its top function symbol f. If the top of the local strategy is a positive integer k, the kth argument t_k is first evaluated, the result t'_k next replaces the argument, and then the altered term $f(\ldots,t'_k,\ldots)$ is evaluated according to the remainder of the local strategy. If the top of the local strategy is zero, the term is tried to be matched with the left sides of rewrite rules: if there exists a rewrite rule whose left side matches with the term (i.e. the term is a redex), the term is replaced by the corresponding instance of the right side (i.e. the contractum of the term), and then the new term is evaluated according to the local strategy of its top function symbol; otherwise the term is continuously evaluated according to the remainder of the local strategy. Such rewriting is going on until a local strategy becomes empty. We here show an example that needs lazy evaluation.

Example 2 (A function nth to take the nth element from an infinite list).

```
mod! TEST1 {
  pr(NAT)
  [S]
  op cons : Nat S -> S { strat: (1 0) }
  op inf : Nat -> S
  op nth : Nat S -> Nat
  eq inf(X:Nat) = cons(X,inf(X + 1)) .
  eq nth(0,cons(X:Nat,L:S)) = X .
  eq nth(X:NzNat,cons(Y:Nat,L:S)) = nth(sd(X,1),L) .
}
```

The operator *cons* has the local strategy (1 0), which means that a term with *cons* as its top function symbol would be replaced by another term, which is then evaluated, after evaluating the first argument. The second argument is not evaluated unless some rewrite frees it from domination of the operator *cons*. Although the other operators are given no explicit local strategies, a default strategy such as (1 2 $\cdots$ 0) is implicitly given to each of them.
Let us evaluate the term *nth*(1,*inf*(1)) w.r.t. *TEST1*. It is rewritten as follows:

$$\begin{aligned} nth(1,\underline{inf(1)}) &\to \underline{nth(1,cons(1,inf(1+1)))} \\ &\to nth(\underline{sd(1,1)},inf(1+1)) \end{aligned}$$

$\rightarrow nth(0,inf(\underline{1+1}))$
$\rightarrow \underline{nth(0,\overline{inf(2)})}$
$\rightarrow \underline{nth(0,\overline{cons(2,inf(2+1))})}$
$\rightarrow \overline{2}$.

Subterms to be rewritten are underlined.
We present an excerpt from the operational semantics of rewriting with the E-strategy in CafeOBJ.

Semantics 1 (Excerpt from the operational semantics of rewriting with the E-strategy).

```
ceq eval(T,TRS) = T  if  eval?(T) .
ceq eval(T,TRS) = reduce(T,strat(T, sig(TRS)),TRS)
                        if  not eval?(T) .
eq reduce(T,nil,TRS) = setEFlag(T) .
eq reduce(T,(0 LS),TRS)
    = reduce2(match(rules(TRS),T,TRS),LS,TRS) .
eq reduce(T,(PI LS),TRS)
    = reduce(evalArg(T,PI,TRS),LS,TRS) .
eq reduce2(⟨ true T ⟩,LS,TRS) = eval(T,TRS) .
eq reduce2(⟨ false T ⟩,LS,TRS) = reduce(T,LS,TRS) .
```

The operator *eval* is a reducer that takes a term T to be reduced and a TRS *TRS* (i.e. a pair of a signature and a set of rewrite rules), and returns the result[3] of evaluating the term. *eval* returns T immediately if T has been already evaluated, or otherwise it evaluates T according to the local strategy of the top function symbol of T by calling *reduce*. The operators *sig* and *strat* return the signature (i.e. a pair of sets of function symbols and variables) of *TRS* and the local strategy of the top function symbol of T. The operator *reduce* takes a term T, the strategy list of the top function symbol of T and a TRS *TRS*, and evaluates T according to the local strategy w.r.t. *TRS*. If the local strategy is or becomes empty (i.e. *nil*), *reduce* returns T after marking T with a flag (called an evaluated flag) meaning that T has been evaluated. If the top of the local strategy is zero, the pattern matcher *match* tries to match T with the left sides of the rewrite rules of *TRS*: if there exists a rewrite rule whose left side matches with T, *match* returns the pair of *true* and the corresponding instance of the right side (i.e. the contractum of T) and *eval* evaluates the new term (i.e. the instance); otherwise *match* returns the pair of *false* and the term passed to *match* as the second argument, and *reduce* continues to evaluate the term according to the remainder *LS* of the local strategy. If the top of the local strategy is a positive integer *PI*, the *PI*th argument is first evaluated and then the altered term is continuously evaluated according to the remainder *LS* of the local strategy.

4. THE ON-DEMAND E-STRATEGY

4.1 Restriction Imposed by the E-strategy

Although the E-strategy makes it possible to simulate a variant of lazy evaluation, it imposes a restriction on constructing rewrite rules. If non-variable terms are put on lazy positions in the left sides, some terms cannot be rewritten as intended. Suppose that *cons* is a list constructor and has (1 0) as its local strategy, and *tl* and *2nd* are usual functions, whose rewrite rules are $tl(cons(X,L)) \rightarrow L$ and $2nd(cons(X,cons(Y,L))) \rightarrow Y$, to return the tail and the second element of a list respectively, the term $2nd(cons(a, tl(cons(b,cons(c,nil)))))$ is not rewritten to c. The term is rewritten as follows. The argument is first evaluated because of the local strategy of *2nd*. But no rewrite occurs owing to the local strategy of *cons*. Although a pattern matcher tries to match the term with the left sides of rewrite rules succeedingly, there is no rewrite rule whose left side can match with the term because the second one of the term's argument is $tl(cons(b,cons(c,nil)))$. Therefore c is not got as a result.
CafeOBJ adopts the on-demand E-strategy that is an extended version of the E-strategy in order to remove the restriction of the E-strategy. The on-demand E-strategy allows us to declare local strategies with negative integers as well as zero and positive integers. A negative integer $-k$ in a local strategy given to a function symbol f means that each subterm in the kth argument t_k of a term $f(t_1, \ldots, t_n)$ is marked with an on-demand flag. While a term is tried to be matched with the left sides of rewrite rules, some of the subterms marked with on-demand flags may be rewritten.

4.2 Example Using the On-demand E-strategy

We show an example that needs the on-demand E-strategy.

Example 3 (Lazy lists and related functions).

```
mod! TEST2 {
  [S]
  op cons : S S -> S { strat: (1 -2 0) }
  op tp : S -> S
  op tl : S -> S
  op 2nd : S -> S
  op 3rd : S -> S
  ops a b c nil : -> S
  eq tp(cons(X:S,L:S)) = X .
  eq tl(cons(X:S,L:S)) = L .
  eq 2nd(cons(X:S,cons(Y:S, L:S))) = Y .
  eq 3rd(cons(X:S,cons(Y:S, cons(Z:S,L:S)))) = Z .
}
```

Let us think of evaluating the term $2nd(cons(a,tl(cons(b, cons(c,nil)))))$ with respect to *TEST2*. The argument $cons(a,tl(cons(b,cons(c,nil))))$ is first evaluated because of the local strategy of *2nd*. Although no rewrite occurs while the argument is evaluated, all the subterms in the second argument are marked with on-demand flags owing to the local strategy (1 −2 0) of *cons*. After that the input term is tried to be matched with the left sides of the rewrite rules. While a pattern matcher tries to match the term with the left side of the third rewrite rule, it finds that the second one of the term's argument (i.e. $tl(cons(b,cons(c,nil)))$) cannot be matched with the corresponding sub-pattern but it has been marked with an on-demand flag. Hence the pattern matcher has a reducer evaluate the subterm marked with the on-demand flag and retries to match the evaluated subterm with the corresponding sub-pattern. In this case the pattern match succeeds because the result of evaluating the subterm $tl(cons(b,cons(c,nil)))$ is $cons(c,nil)$. Consequently the input term is replaced by c that is the final result of rewriting the original term.

4.3 Operational Semantics of the On-demand E-strategy

The following is an excerpt from the operational semantics of rewriting with the on-demand E-strategy in CafeOBJ.

Semantics 2 (Excerpt from the operational semantics of rewriting with the on-demand E-strategy).

[3]The result is a reduct of the term, but it might not be in normal form. For example, the result of evaluating $inf(1)$ is $cons(1,inf(1+1))$ that is not in normal form w.r.t. *TEST1*.

ceq $eval(T,TRS) = T$ **if** $eval?(T)$.
ceq $eval(T,TRS) = reduce(T,strat(T, sig(TRS)),TRS)$
  if not $eval?(T)$.
eq $reduce(T,nil,TRS) = setEFlag(T)$.
eq $reduce(T,(0\ LS),TRS)$
  $= reduce2(match(rules(TRS),T,TRS),LS,TRS)$.
eq $reduce(T,(PI\ LS),TRS)$
  $= reduce(evalArg(T,PI,TRS),LS,TRS)$.
ceq $reduce(T,(NI\ LS),TRS)$
  $= reduce(upODF(T,(-\ NI)),LS,TRS)$ **if** $NI < 0$.
eq $reduce2(\langle\ true\ T\ \rangle,LS,TRS) = eval(T,TRS)$.
eq $reduce2(\langle\ false\ T\ \rangle,LS,TRS) = reduce(T,LS,TRS)$.

Clearly the only difference between Semantics 1 and Semantics 2 is the fourth equation for *reduce* of Semantics 2. If the top of the local strategy given to a term T is a negative integer NI, the operator *upODF* marks an on-demand flag on each subterm in the $-NI$th argument of T. In Semantics 1, if there is no rewrite rule whose left side matches with a term, the second element of the pair returned by *match* is exactly the same as the term passed to *match* as its second argument. In Semantics 2, however, the second element of the pair returned by *match* might be different from the term passed to *match* as its second argument because some subterms in the term might be rewritten while *match* tries to match the term with the left sides of rewrite rules.
Since terms might be rewritten while *match* tries to match them with the left sides of rewrite rules, we should write the operational semantics of pattern matching (i.e. *match*) in more detail in order to specify the operational semantics of rewriting with the on-demand E-strategy more precisely. Therefore we present the operational semantics of *match*.

Semantics 3 (Operational semantics of *match*).

eq $match(nil,T,TRS) = \langle\ false\ T\ \rangle$.
eq $match((RR\ RRs),T,TRS)$
  $= match2(pm(T,lhs(RR),TRS),(RR\ RRs),TRS)$.
eq $match2(\langle\ true\ S\ T\ \rangle,(RR\ RRs),TRS)$
  $= \langle\ true\ instantiate(rhs(RR),S)\ \rangle$.
eq $match2(\langle\ false\ S\ T\ \rangle,(RR\ RRs),TRS)$
  $= match(RRs,T,TRS)$.

The operator *match* takes a list of rewrite rules, a term that is tried to be matched with the left sides, and a TRS. It returns the pair of *true* and a contractum if there exists a rewrite rule whose left side can match with the term, or otherwise it returns the pair of *false* and the term passed to *match* as its second argument that might be rewritten. The operator *pm* actually tries to match a term with the left side of a rewrite rule. It takes a term, the left side of a rewrite rule and a TRS, and returns the triple of *true*, the substitution obtained through the pattern match and the term that might be rewritten if the term can match with the left side, or otherwise it returns the triple of *false*, a dummy substitution and the term that might be rewritten. The auxiliary operator *match2* returns the pair of *true* and the corresponding instance of the right side (i.e. the contractum) if the term can match with the left side of the rewrite rule, or otherwise it calls *match* in order to try to match the term with the remainder of the rewrite rules.
We present the operational semantics of *pm* that tries to match a term with the left side of a rewrite rule.

Semantics 4 (Operational semantics of *pm*).

eq $pm(T,(\mathrm{var}\ V),TRS) = \langle\ true\ (\langle(\mathrm{var}\ V)\ T\ \rangle\ nil)\ T\ \rangle$.
ceq $pm(T,Sym\{Ps\},TRS) = pml(nil,1,T,\ Sym\{Ps\},TRS)$
  if $\mathrm{top}(T) == Sym$.
ceq $pm(T,Sym\{Ps\},TRS)$
  $= pm(eval(downODF(T),TRS),Sym\{Ps\},TRS)$
  if $top(T)\ =/=\ Sym$ and $upODF?(T)$.
ceq $pm(T,Sym\{Ps\},TRS)$
  $= \langle\ false\ nil\ T\ \rangle$
  if $top(T)\ =/=\ Sym$ and (not $upODF?(T)$) .
ceq $pml(S,N,T,P,TRS)$
  $= pml2(pm(arg(T,N),arg(P,N),TRS),S,N,T,P,TRS)$
  if $N <= arity(T,sig(TRS))$.
ceq $pml(S,N,T,P,TRS) = \langle\ true\ S\ T\ \rangle$
  if $N > arity(T,sig(TRS))$.
eq $pml2(\langle\ true\ S1\ T1\ \rangle,S,N,T,P,TRS)$
  $= pml(S1\ ++\ S,N+1,replace(T,N,T1),P,TRS)$.
eq $pml2(\langle\ false\ S1\ T1\ \rangle,S,N,T,P,TRS)$
  $= \langle\ false\ nil\ replace(T,N,T1)\ \rangle$.

There are two types of constructors *var_* and _{_} for patterns (i.e. the left and right sides of rewrite rules). The first constructor *var_* is one for variables. It takes a string as its argument that denotes a variable name, e.g. *var* "X" denotes a variable X. The second constructor _{_} is one for non-variable patterns such as $tp(cons(X,L))$. It takes a string and a list of patterns as its first and second arguments. The string and the list denote the top function symbol name and the arguments of a pattern, e.g. "tp"{("cons"{(*var* "X") (*var* "L") *nil*}) *nil*} denotes the pattern $tp(cons(X,L))$.
If the pattern is a variable *var* V, *pm* returns the triple of *true*, the pair of the variable and the term T (more precisely the singleton list of the pair denoting a substitution), and the term. If the pattern is a non-variable one $Sym\{Ps\}$, the situation divides into three cases. If the top function symbol of the term T is equal to Sym the top function symbol of the pattern, *pml* tries to match the arguments of the term with those of the pattern. The next case is the most interesting in rewriting with the on-demand E-strategy. If the top function symbol of the term is not seemingly equal to Sym but the term has been marked with an on-demand flag, *eval* evaluates the term after removing the on-demand flag from it, and then *pm* retries to match the term that might have been rewritten with the pattern. No removing the on-demand flag may lead to an infinite loop. If the top function symbol of the term is not equal to Sym and the term has not been marked with an on-demand flag, *pm* returns the triple of *false*, the empty substitution *nil* and the term, which means failure in the pattern match.
The operator *pml* tries to match the arguments of a term with those of a pattern and returns the triple of *true*, the substitution obtained through the pattern match and the term that might be rewritten, which means success in the pattern matching, if each of the arguments of the term can match with each of them of the pattern. The reason why the Nth argument of the term T is replaced by $T1$ at the right sides of the two rewrite rules for *pml2* is that the argument might have been rewritten through the pattern match.

4.4 Operational Semantics as Interpreter

Since we can make use of the operational semantics as an interpreter thanks to the executability of CafeOBJ, we can observe the dynamic behavior of rewriting with the on-demand E-strategy using the operational semantics.
We describe the data structure for terms to be rewritten prior to observing the dynamic behavior of rewriting with the on-demand E-strategy. _{_} is a constructor for terms to be rewritten. The first and second arguments are a node and

```
SIMULATOR> red eval(t1,Lcons) .
-- reduce in SIMULATOR : eval(t1,Lcons)
("c" [ false true ]) { nil } : Subject
(0.000 sec for parse, 1821 rewrites(0.445 sec), 2378 matches)

SIMULATOR> red eval(t2,Lcons) .
-- reduce in SIMULATOR : eval(t2,Lcons)
("d" [ false true ]) { nil } : Subject
(0.008 sec for parse, 12338 rewrites(2.516 sec), 16363 matches)

SIMULATOR> red eval(t3,Lcons) .
-- reduce in SIMULATOR : eval(t3,Lcons)
("d" [ false true ]) { nil } : Subject
(0.000 sec for parse, 12523 rewrites(2.570 sec), 16583 matches)
```

Figure 1: Snapshot of the simulator

a list of arguments. The node consists of a function symbol name and two flags (i.e. an on-demand flag and an evaluated flag), and its constructor is _[__]. For example, "cons"[false false]{("a"[false false]{nil}) ("nil"[false false]{nil}) nil} denotes the term *cons*(a,*nil*). We write a TRS corresponding to *TEST2* in the operational semantics.

Example 4 (The TRS Lcons corresponding to TEST2).

```
mod! LCONS {
  pr(TRS)
  ops x y z l : -> Var
  op V : -> VarList
  op F : -> FSList
  ops cons tp tl 2nd 3rd a b c d nl : -> FSym
  op Sig : -> Signature
  ops r1 r2 r3 r4 : -> Rule
  op R : -> RuleList
  op Lcons : -> Trs
  eq x = var "X" .
  eq y = var "Y" .
  eq z = var "Z" .
  eq l = var "L" .
  eq V = x y z l nil .
  eq cons = op "cons" 2 (1 -2 0 nil) .
  eq tp = op "tp" 1 (1 0 nil) .
  eq tl = op "tl" 1 (1 0 nil) .
  eq 2nd = op "2nd" 1 (1 0 nil) .
  eq 3rd = op "3rd" 1 (1 0 nil) .
  eq a = op "a" 0 (0 nil) .
  eq b = op "b" 0 (0 nil) .
  eq c = op "c" 0 (0 nil) .
  eq d = op "d" 0 (0 nil) .
  eq nl = op "nl" 0 (0 nil) .
  eq F = cons tp tl 2nd 3rd a b c d nl nil .
  eq Sig = ⟨ V F ⟩ .
  eq r1 = ⟨ ("tp"{("cons"{x l nil}) nil}) x ⟩ .
  eq r2 = ⟨ ("tl"{("cons"{x l nil}) nil}) l ⟩ .
  eq r3 = ⟨ ("2nd"{("cons"{x
            ("cons"{y l nil}) nil}) nil}) y ⟩ .
  eq r4 = ⟨ ("3rd"{("cons"{x ("cons"{y
            ("cons"{z l nil}) nil}) nil}) nil}) z ⟩ .
  eq R = r1 r2 r3 r4 nil .
    eq Lcons = ⟨ Sig R ⟩ .
}
```

The constructor for function symbols is *op*___ that takes the function symbol name (i.e. a string), the arity and the local strategy given to it as its arguments. For example, *op* "cons" 2 (1 −2 0 *nil*) corresponds to the function symbol *cons* in *TEST2*. The TRS *Lcons* has four rewrite rules *r1*, *r2*, *r3* and *r4* that conrresponds to $tp(cons(X,L)) \to X$, $tl(cons(X,L)) \to X$, $2nd(cons(X,cons(Y,L))) \to Y$, and $3rd(cons(X,cons(Y,cons(Z,L)))) \to Z$, respectively. The module *TRS* imported into *LCONS* provides the data structure for constructing TRSs such as variables, function symbols, patterns and rewrite rules.
We give some terms to be reduced.

Example 5 (Terms to be reduced w.r.t. Lcons).

```
mod! LCONSTERMS {
  pr(SUBJECT)
  ops consN tpN tlN 2ndN
       3rdN aN bN cN dN nlN : -> Node
  ops t0 t1 t2 t3 : -> Subject
  eq consN = "cons"[false false] .
  eq tpN = "tp"[false false] .
  eq tlN = "tl"[false false] .
  eq 2ndN = "2nd"[false false] .
  eq 3rdN = "3rd"[false false] .
  eq aN = "a"[false false] .
  eq bN = "b"[false false] .
  eq cN = "c"[false false] .
  eq dN = "d"[false false] .
  eq nlN = "nl"[false false] .
  eq t0 = consN{(cN{nil})
           (consN{(dN{nil}) (nlN{nil}) nil}) nil} .
  eq t1 = tpN{t0 nil} .
  eq t2 = 2ndN{(consN{(bN{nil})
           (tlN{t0 nil}) nil}) nil} .
  eq t3 = 3rdN{(consN{(aN{nil})
           (consN{(bN{nil}) (tlN{t0 nil}) nil}) nil}) nil} .
}
```

t1, *t2* and *t3* denote the following terms:

t1	–	$tp(cons(c,cons(d,nil)))$
t2	–	$2nd(cons(b,tl(cons(c,cons(d,nil)))))$
t3	–	$3rd(cons(a,cons(b,tl(cons(c,cons(d,nil))))))$.

The module *SUBJECT* imported into *LCONSTERMS* provides the data structure for terms to be rewritten.
One more module is necessary in order to simulate rewriting with the on-demand E-strategy using the operational semantics.

Example 6 (Simulator for rewriting with the on-demand E-strategy).

```
mod! SIMULATOR
{ pr(REWRITING + LCONS + LCONSTERMS) }
```

The module *SIMULATOR* imports three modules together with module sum. The importation can be separated into three times of importation. The module *SIMULATOR* is equivalent to the following one:

```
mod! SIMULATOR
{ pr(REWRITING) pr(LCONS) pr(LCONSTERMS) } .
```

The module *REWRITING* provides the operational semantics of rewriting with the on-demand E-strategy.
We show the snapshot of simulating the rewrites of the three terms *t1*, *t2* and *t3* with the on-demand E-strategy in Fig. 1.

5. HINT ABOUT SPECIFYING LOCAL STRATEGIES

In this section a hint about giving local strategies to operators (or function symbols) is presented. The on-demand E-strategy gives us two ways of postponing evaluating some terms. If we want to postpone evaluating the kth argument of a term $f(t_1, \dots, t_n)$, the operator f is given a local strategy excluding k or a local strategy including $-k$. There are at least two things that can be carried out by postponing evaluating terms: avoiding wasteful evaluation and dealing with infinite data structures such as infinite lists.
It is possible to avoid wasteful evaluation by giving local strategies excluding some positive integers to operators (i.e. functions). A typical example of avoiding wasteful evaluation is the conditional operator *if_then_else_fi*.

Example 7 (Conditional operator if_then_else_fi*).*

op *if_then_else_fi* : *Bool S S* −> *S* { **strat:** (1 0) }
eq *if true then X:S else Y:S fi* = *X* .
eq *if false then X:S else Y:S fi* = *Y* .

A term with *if_then_else_fi* as its top operation such as "*if cond then comp1 else comp2 fi*" is replaced by the second argument *comp1* or the third argument *comp2* depending on the result (i.e. *true* or *false*) of evaluating the first argument *cond*. If the result of evaluating the first argument is *true* (or *false*), the second (or third) argument replaces the whole term, and the third (or second) argument is just discarded and does not cause wasteful evaluation.
It is possible to deal with infinite data structures by giving local strategies including negative integers to operators (data constructors). For example, infinite lists can be dealt with by giving the local strategy (1 −2 0) to the list constructor __ as follows: **op** __ : Nat ooList −> ooList { **strat:** (1 −2 0) } (*ooList* stands for infinite lists and lists of natural numbers are considered for brevity). The result of evaluating some term denoting an infinite list is in head normal form, but the evaluation does not lead to infinite rewriting. Suppose that *inf* is declared as **op** *inf* : *Nat* −> *ooList* and **eq** *inf*(X:*Nat*) = X *inf*($X + 1$), the result of evaluating *inf*(0) is "0 *inf*(1)." But the result of evaluating even some term denoting a finite list might be in head normal form. For example, the result of evaluating "*tp*(1 2 *nil*) *tl*(1 2 3 *nil*)" is "1 *tl*(1 2 3 *nil*)" that is in head normal form, but not in normal form. The result of evaluating some term denoting a finite list should be in normal form. Since CafeOBJ supports order-sorted rewriting, infinite lists are allowed to coexist with finite lists well by having the sort *ooList* have a subsort *List* for finite lists and declaring the constructor for finite lists. The constructor for finite lists is given the eager local strategy (1 2 0).

Example 8 (Coexistence of infinite lists with finite lists).

[*List* < *ooList*]
op *nil* : −> *List*
op __ : *Nat List* −> *List* { **strat:** (1 2 0) }
op __ : *Nat ooList* −> *ooList* { **strat:** (1 −2 0) }
op *tp* : *ooList* −> *Nat*
op *tl* : *List* −> *List*
op *tl* : *ooList* −> *ooList*

The result of evaluating "*tp*(1 2 *nil*) *tl*(*inf*(0))" is "1 *tl*(*inf*(0))," while the result of evaluating "*tp*(1 2 *nil*) *tl*(1 2 3 *nil*)" is "1 2 3 *nil*." Since the top operator __ of the former term is the constructor for infinite lists because the second term *tl*(*inf*(0)) denotes an infinite list, the second argument is not evaluated because of the local strategy (1 −2 0) of the operator. The top operator __ of the latter term is the constructor for finite lists because the second argument *tl*(1 2 3 *nil*) is finite, and the both of the first and second arguments are evaluated.

6. DISCUSSION

Lazy evaluation is fascinating because it often has a better termination behavior than eager evaluation, while it is much more difficult to implement efficiently lazy evaluation than eager evaluation. Therefore some compromises have been proposed. The functional strategy [10] is one of them, which is often used in the field of the implementation of functional languages such as Miranda and Haskell. The operational semantics of a rewrite step and the annotated functional strategy in Miranda is also given [10]. They claim that the formal specification in a functional language has two advantages besides the well-defined semantics. First, the partial correctness of the specification can be confirmed by an implementation of the description language. Second, the dynamic behavior of the specified algorithms can be observed. We can say that the operational semantics of rewriting with the on-demand E-strategy in CafeOBJ has the same advantages. Kamperman and Walters have proposed a transformation method for TRSs and terms to be rewritten so that lazy evaluation can be simulated on an implementation of eager evaluation: *lazy rewriting on eager machinery* [9]. The method concisely expresses the intricate interaction between pattern matching and lazy evaluation. Our operational semantics concisely expresses the same thing by the third equation for *pm* in Semantics 4, viz if the top symbol of a term is not seemingly equal to that of a pattern but the term has been marked with an on-demand flag, *pm* retries to match the term with the pattern after evaluating the term.
We introduce two more examples of semantics written in formal languages. One is the operational semantics of an organic programming language GAEA [5] in Maude [3] a specification language based on rewriting logic [11], and the other the algebraic semantics of imperative programs in OBJ3 [6]. Ishikawa et al. [8] have written the operational semantics of GAEA in Maude as an instance through a study on declarative description of reflective concurrent systems. Goguen and Malcolm [7] give the algebraic semantics of imperative programs in OBJ3 so as to introduce Computing Science students to formal reasoning about imperative programs. They do not only write the algebraic semantics of an imperative programming language, but also prove assertions about the behavior of programs written in the imperative programming language.

7. CONCLUSION

We have described the operational semantics of rewriting with the on-demand E-strategy and have observed the dynamic behavior of rewriting with the strategy using the operational semantics as an interpreter. A hint about giving local strategies to operators (or function symbols) has been presented as well.

8. REFERENCES

[1] Baader, F. and Nipkow, T.: *Term Rewriting and All That.* Cambridge University Press. 1998

[2] CafeOBJ home page: http://caraway.jaist.ac.jp/cafeobj

[3] Clavel, M., Eker, S., Lincoln, P. and Meseguer, J.: Principles of Maude. Proc. of the First Int'l. Workshop on Rewriting Logic and its Applications. ENTCS **4** Elsevier. (1996) 65–89

[4] Futatsugi, K., Goguen, J. A., Jouannaud, J. P. and Meseguer, J.: Principles of OBJ2. Conf. Record of the Twelfth Annual ACM Sympo. on Princ. of Prog. Lang. (1985) 52–66

[5] GAEA home page: http://cape.etl.go.jp/gaea

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

[7] Goguen, J. A. and Malcolm, G.: *Algebraic Semantics of Imperative Programs.* Foundations of Computer Series. (Eds. M. Garey and A. Meyer) The MIT Press. 1996

[8] Ishikawa, H., Meseguer, J., Watanabe, T., Futatsugi, K. and Nakashima, H.: On the semantics of GAEA. Proc. of JSSST 3rd Fuji Int'l Sympo. on Functional and Logic Programming. (1998) 123–141

[9] Kamperman, J. F. Th. and Walters, H. R.: Lazy Rewriting and Eager Machinery. Proc. of the Int'l. Conf. on Rewriting Techniques and Applications. LNCS **914** Springer-Verlag. (1995) 147–162

[10] Koopman, P. W. M., Smetsers, J. E. W., van Eekelen, M. C. J. D. and Plasmeijer, M. J.: Graph Rewriting Using the Annotated Functional Strategy. In *Term Graph Rewriting: Theory and Practice* (Eds. R. Sleep, R. Plasmeijer and M. van Eekelen). John Wiley & Sons Ltd. (1993) 317–332

[11] Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. *Theor. Comp. Sci.* **96** (1) Elsevier. (1992) 73–155

APPENDIX

A. OPERATIONAL SEMANTICS OF REWRITING WITH THE ON-DEMAND E-STRATEGY IN CAFEOBJ

We present the full of the operational semantics of rewriting using the on-demand E-strategy in CafeOBJ.

```
mod! LIST (DAT :: TRIV) {
  pr(NAT)
  [List]
  op nil : -> List
  op __ : Elt.DAT List -> List {r-assoc}
  op _++_ : List List -> List
  op take : List Nat -> Elt.DAT
  op replace : List Nat Elt.DAT -> List
  vars E E1 : Elt.DAT
  vars L L1 : List
  eq nil ++ L1 = L1 .
  eq (E L) ++ L1 = E (L ++ L1) .
  eq take(E L,0) = E .
  eq take(E L,X:NzNat) = take(L,sd(X,1)) .
  eq replace(E L,0,E1) = E1 L .
  eq replace (E L,X:NzNat,E1) = E replace(L,sd(X,1),E1) .
}

mod! PAIR (FST :: TRIV,SND :: TRIV) {
  [Pair]
  op <__> : Elt.FST Elt.SND -> Pair
  op 1st : Pair -> Elt.FST
  op 2nd : Pair -> Elt.SND
  eq 1st(< X:Elt.FST Y:Elt.SND >) = X .
  eq 2nd(< X:Elt.FST Y:Elt.SND >) = Y .
}

mod! TRIPLE (FST :: TRIV,SND :: TRIV,TRD :: TRIV) {
  [Triple]
  op <___> : Elt.FST Elt.SND Elt.TRD -> Triple
  op 1st : Triple -> Elt.FST
  op 2nd : Triple -> Elt.SND
  op 3rd : Triple -> Elt.TRD
  var X : Elt.FST
  var Y : Elt.SND
  var Z : Elt.TRD
  eq 1st(< X Y Z >) = X .
  eq 2nd(< X Y Z >) = Y .
  eq 3rd(< X Y Z >) = Z .
}

mod! VARIABLE principal-sort Var {
  pr(STRING)
  [Var]  op var_ : String -> Var
}

mod! LOCALSTRATEGY { pr(LIST(INT)*{sort List -> LStrat}) }

mod! FSYMBOL principal-sort FSym {
  pr(STRING + NAT + LOCALSTRATEGY)
  [FSym]
  op op___ : String Nat LStrat -> FSym
  op arity : FSym -> Nat
  op strat : FSym -> LStrat
  op top : FSym -> String
  var S : String
  var N : Nat
  var L : LStrat
  eq arity(op S N L) = N .
  eq strat(op S N L) = L .
  eq top(op S N L) = S .
}

mod! VARLIST principal-sort VarList
{ pr(LIST(VARIABLE)*{sort List -> VarList}) }

mod! FSYMLIST principal-sort FSList {
  pr(LIST(FSYMBOL)*{sort List -> FSList})
  op st2Fsym : FSList String -> FSym
  var O : FSym
  var Os : FSList
  var Sym : String
  eq st2Fsym((O Os),Sym)
     = if top(O) == Sym then O else st2Fsym(Os,Sym) fi .
}

mod! SIGNATURE principal-sort Signature {
  pr(PAIR(VARLIST,FSYMLIST)*{sort Pair -> Signature})
  op st2Fsym : Signature String -> FSym
  eq st2Fsym(< Vs:VarList Os:FSList >,Sym:String)
     = st2Fsym(Os,Sym) .
}

mod* QUASIPATTERN principal-sort Pattern {
  pr(VARIABLE + STRING)
  [Var < Pattern]
}

mod! PATTERN principal-sort Pattern {
  pr(LIST(QUASIPATTERN)*{sort List -> PList})
  op _{_} : String PList -> Pattern
  op arg : Pattern NzNat -> Pattern
  eq arg(Sym:String{Args:PList},X:NzNat)
     = take(Args,sd(X,1)) .
}
```

```
mod! RULE principal-sort Rule {
  pr(PAIR(PATTERN,PATTERN)*{sort Pair -> Rule})
  ops lhs rhs : Rule -> Pattern
  eq lhs(RR:Rule) = 1st(RR) .
  eq rhs(RR:Rule) = 2nd(RR) .
}

mod! RULELIST principal-sort RuleList
{ pr(LIST(RULE)*{sort List -> RuleList}) }

mod! TRS {
  pr(PAIR(SIGNATURE,RULELIST)*{sort Pair -> Trs})
  op sig : Trs -> Signature
  op rules : Trs -> RuleList
  eq sig(System:Trs) = 1st(System) .
  eq rules(System:Trs) = 2nd(System) .
}

mod! NODE {
  pr(STRING)
  [Node]
  op _[__] : String Bool Bool -> Node
  op top : Node -> String
  op upODF? : Node -> Bool
  op eval? : Node -> Bool
  op upODF : Node -> Node
  op downODF : Node -> Node
  op setEFlag : Node -> Node
  vars OD EF : Bool
  var Sym : String
  eq top(Sym[OD EF]) = Sym .
  eq upODF?(Sym[OD EF]) = OD .
  eq eval?(Sym[OD EF]) = EF .
  eq upODF(Sym[OD EF]) = Sym[true EF] .
  eq downODF(Sym[OD EF]) = Sym[false EF] .
  eq setEFlag(Sym[OD EF]) = Sym[OD true] .
}

mod* QUASISUBJECT { [Subject] }

mod! SUBJECT principal-sort Subject {
  pr(SIGNATURE + NODE
             + LIST(QUASISUBJECT)*{sort List -> SList})
  op _{_} : Node SList -> Subject
  op top : Subject -> String
  op arity : Subject Signature -> Nat
  op strat : Subject Signature -> LStrat
  op upODF? : Subject -> Bool
  op eval? : Subject -> Bool
  op upODF : Subject -> Subject
  op upODF : SList -> SList
  op downODF : Subject -> Subject
  op setEFlag : Subject -> Subject
  op arg : Subject NzNat -> Subject
  op replace : Subject NzNat Subject -> Subject
  var Nd : Node  var T : Subject vars Args Ts : SList
  var Sig : Signature
  eq top(Nd{Args}) = top(Nd) .
  eq arity(Nd{Args},Sig) = arity(st2Fsym(Sig,top(Nd))) .
  eq strat(Nd{Args},Sig) = strat(st2Fsym(Sig,top(Nd))) .
  eq upODF?(Nd{Args}) = upODF?(Nd) .
  eq eval?(Nd{Args}) = eval?(Nd) .
  eq upODF(Nd{Args}) = upODF(Nd){upODF(Args)} .
  eq upODF(nil) = nil .
  eq upODF(T Ts) = upODF(T) upODF(Ts) .
  eq downODF(Nd{Args}) = downODF(Nd){Args} .
  eq setEFlag(Nd{Args}) = setEFlag(Nd){Args} .
  eq arg(Nd{Args},X:NzNat) = take(Args,sd(X,1)) .
  eq replace(Nd{Args},X:NzNat,T:Subject)
     = Nd{replace(Args,sd(X,1),T)} .
}

mod! ASSIGN principal-sort Assign
{ pr(PAIR(VARIABLE,SUBJECT)*{sort Pair -> Assign}) }

mod! SUBSTITUTION principal-sort Subst {
  pr(LIST(ASSIGN)*{sort List -> Subst})
  vars V V1 : Var
  var T : Subject
  var S : Subst
  op var2term : Var Subst -> Subject
  eq var2term(V,< V1 T > S)
     = if V == V1 then T else var2term(V,S) fi .
}

mod! MATCH
{ pr(TRIPLE(BOOL,SUBSTITUTION,SUBJECT)
     *{sort Triple -> Match}) }

mod! CONTRACTUM
{ pr(PAIR(BOOL,SUBJECT)*{sort Pair -> Contractum}) }

mod* REWRITING {
  pr(TRS + SUBJECT + MATCH + CONTRACTUM)
  op pm : Subject Pattern Trs -> Match
  op pml : Subst NzNat Subject Pattern Trs -> Match
  op pml2 : Match Subst NzNat Subject Pattern Trs -> Match
  op match : RuleList Subject Trs -> Contractum
  op match2 : Match RuleList Trs -> Contractum
  op eval : Subject Trs -> Subject
  op reduce : Subject LStrat Trs -> Subject
  op reduce2 : Contractum LStrat Trs -> Subject
  op evalArg : Subject NzNat Trs -> Subject
  op upODF : Subject NzNat -> Subject
  op instantiate : Pattern Subst -> Subject
  op instantiate : PList Subst -> SList
  vars T T1 : Subject
  vars N PI : NzNat
  vars V Sym : String
  var TRS : Trs
  var Ps : PList
  vars S S1 : Subst
  var P : Pattern
  var LS : LStrat
  var RR : Rule
  var RRs : RuleList
  var NI : NzInt
  ceq eval(T,TRS) = T if eval?(T) .
  ceq eval(T,TRS)
     = reduce(T,strat(T,sig(TRS)),TRS) if not eval?(T) .
  eq reduce(T,nil,TRS) = setEFlag(T) .
  eq reduce(T,(0 LS),TRS)
    = reduce2(match(rules(TRS),T,TRS),LS,TRS) .
  eq reduce(T,(PI LS),TRS)
    = reduce(evalArg(T,PI,TRS),LS,TRS) .
  ceq reduce(T,(NI LS),TRS)
     = reduce(upODF(T,(- NI)),LS,TRS) if NI < 0 .
  eq reduce2(< true T >,LS,TRS) = eval(T,TRS) .
  eq reduce2(< false T >,LS,TRS) = reduce(T,LS,TRS) .
  eq match(nil,T,TRS) = < false T > .
  eq match((RR RRs),T,TRS)
    = match2(pm(T,lhs(RR),TRS),(RR RRs),TRS) .
  eq match2(< true S T >,(RR RRs),TRS)
    = < true instantiate(rhs(RR),S) > .
  eq match2(< false S T >,(RR RRs),TRS) = match(RRs,T,TRS) .
  eq pm(T,(var V),TRS) = < true (<(var V) T > nil) T > .
  ceq pm(T,Sym{Ps},TRS)
     = pml(nil,1,T,Sym{Ps},TRS) if top(T) == Sym .
  ceq pm(T,Sym{Ps},TRS)
     = pm(eval(downODF(T),TRS),Sym{Ps},TRS)
     if top(T) =/= Sym and upODF?(T) .
  ceq pm(T,Sym{Ps},TRS) = < false nil T >
     if top(T) =/= Sym and (not upODF?(T)) .
  ceq pml(S,N,T,P,TRS)
     = pml2(pm(arg(T,N),arg(P,N),TRS),S,N,T,P,TRS)
     if N <= arity(T,sig(TRS)) .
  ceq pml(S,N,T,P,TRS) = < true S T >
     if N > arity(T,sig(TRS)) .
  eq pml2(< true S1 T1 >,S,N,T,P,TRS)
    = pml(S1 ++ S,N + 1,replace(T,N,T1),P,TRS) .
  eq pml2(< false S1 T1 >,S,N,T,P,TRS)
    = < false nil replace(T,N,T1) > .
  eq evalArg(T,N,TRS)
    = replace(T,N,eval(downODF(arg(T,N)),TRS)) .
  eq upODF(T,N) = replace(T,N,upODF(arg(T,N))) .
  eq instantiate(var V,S) = var2term(var V,S) .
  eq instantiate(Sym{Ps},S)
    = Sym[false false]{instantiate(Ps,S)} .
  eq instantiate(nil,S) = nil .
  eq instantiate(P Ps,S)
    = instantiate(P,S) instantiate(Ps,S) .
}
```