T: A simple reduction language based on combinatory term rewriting* - Draft -

AKIRA AIBA
Institute for New Generation Computer Technology,
1-4-28, Mita, Minato-ku, Tokyo 108, Japan

TETSUO IDA
Information Science Laboratory,
Institute of Physical and Chemical Research
2-1 Hirosawa, Wako-shi, Saitama 351-01, Japan

YOSHIHITO TOYAMA
NTT Basic Research Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo 180, Japan

November 4, 2008

Abstract

A programming language, called T, based on a combinatory term rewriting system is presented. Conditions are discussed to make the term rewriting system a viable programming language. Descriptive power of the language is derived by making the system combinatory and by allowing pattern matching in seeking applicable rewrite rules. Normalizing reduction strategies are discussed in conjunction with the implementation. We also discuss the methods for efficient implementation of the language.

Keywords: Reduction language,Term rewriting system

1 Introduction

Recent progress in the studies of new programming paradigms makes it clear that logical systems developed in mathematics can serve as a practical computation model of a programming language. λ-calculus and first order predicate logic, which are respectively the computation models of functional languages and Prolog are good examples.

*This work is partly based on the activities of TRS WG of ICOT

In this paper, we present a language (to be called T, T for term) based on a combinatory term rewriting system and discuss its effectiveness as a practical programming language. We start from the following general observations about a combinatory term rewriting system:

(1) Rewrite rules with some restrictions have enough power to describe computation both from theoretical and pragmatic point of view.

(2) Combinatory reduction has very simple semantics and can be designed to possess properties, such as the Church-Rosser property and the referential transparency that are desirable in manipulating and reasoning about programs.

(3) Efficient implementation is possible.

We discuss more about the design objectives of T in section 2 and the syntax and semantics in section 3. Considerations for efficient implementation are given in section 5.

A programming language whose computation model is explicitly based on a term rewriting system, as far as we know, is O'Donnel's equational language [11]. T is different from it in which in T programming, programmers reason about programs in terms of combinatory term reduction, whereas in O'Donnel's language, reasoning is purely equational. T turns out to be simialr to HOPE [2] and KRC [13] in its semantics than equational languages.

2 T and term rewriting systems

It is widely known that term rewriting captures the notion of reduction. A term rewriting system (abbreviated as TRS hereafter) is a set of rewrite rules

$$\{L_i \rightarrow R_i,\ i = 0, 1, \ldots\}$$

and $\mathrm{Var}(L_i) \supset \mathrm{Var}(R_i)$, where L_i and R_i are terms, and $\mathrm{Var}(M)$ is a set of variables that appear in term M. A (general) TRS provides a computation mechanism for theorem proving, equational (specification) languages, abstract data type checking and validation, and programming languages. We are interested in a TRS which serves as a computation model of a practical programming language. For a TRS to be a computation model of a programming language, we need to impose some restrictions on the formation of rewrite rules.

A single criterion, apart from efficiency criteria, is that the TRS should satisfy the uniqueness of the answer (consequence of the Church-Rosser property), if it exists.

Overall design objectives of T are as follows.

(1) T is a functional language based on reduction, whose programs satisfy the Church-Rosser property.

(2) It can be an "assembly language" for an abstract reduction machine.

(3) From (2), we require T to be a type-free language. We assume objects of T carry types in the form of a tag.

(4) Despite the low-level-ness of T as inferred from (2) and (3) we provide proper syntax (in macro) so that a large class of algorithms are written in T naturally.

(5) We tried purposely not to incorporate high level language features which will incur run-time overhead, or which may be incorporated into a higher level language that is built on T. These include typing facilities and equational specification.

We describe in the following how these objectives are realized by a restricted TRS.

3 Syntax and semantics of T

3.1 Term

The syntax and semantics of language T is briefly described with the help of BNF. First we consider terms;

$$
\begin{aligned}
< \mathit{term} > \;&::=\; < \mathit{basic{-}term} > \mid < \mathit{term} >< \mathit{term} > \\
< \mathit{basic{-}term} > \;&::=\; < \mathit{combinator} > \mid < \mathit{variable} >
\end{aligned}
$$

Association of terms are to the left, and parentheses are used freely to change the association. It is understood that each syntactic category introduced in BNF may be parenthesized, if necessary. The syntax of combinators and variables is usual, hence we omit the explanation. (See the examples below.)

Figure 1: Classification of basic-terms with examples

Structured data are constructed by constructors. For example, "succ" is used to construct natural numbers and "cons" to construct list structures e.g. 2≡succ(succ 0), where ≡ denotes the syntactic equality (conceptually, natural numbers are constructed this way), "list of 1 and 2"≡cons (succ 0) (cons (succ (succ 0)) nil). Note that this construction should not be confused with actual implementation of data structures. Implementation will make use of all available hardware to attain efficient processing of heavily used data structures.

Example 3.1
add n m ≡ (add n) m. *Then* "add n m" *is seen as the addition of integers* n *and* m*, if* "add" *is predefined as a* δ*-combinator with the rewrite rules of addition.* "add n" *is seen as plus-n function.*

Variables are syntactically indistinguishable from combinators, hence they have to be introduced by declaring

$$\mathrm{Var}\ \{\mathrm{x}_1,\ \mathrm{x}_2,\ \ldots\};.$$

3.2 Rewrite rules and Reduction

Rewrite rules are of the form

$$\begin{array}{rcl} \mathrm{f\ P_{1,1}\ P_{1,2}\ \ldots\ P_{1,n_1}} & \rightarrow & \mathrm{E_1;} \\ \mathrm{f\ P_{2,1}\ P_{2,2}\ \ldots\ P_{2,n_2}} & \rightarrow & \mathrm{E_2;} \\ \vdots & & \\ \mathrm{f\ P_{m,1}\ P_{m,2}\ \ldots\ P_{m,n_m}} & \rightarrow & \mathrm{E_m;} \end{array}$$

where f is a defined-combinator. That is, the combinator "f" is defined by these rewrite rules.

More precisely, the syntax of rewrite rules is the following.

$$\begin{array}{rcl} < definition > & ::= & < defconst > \rightarrow < term >; \mid \\ & & < defcombinator >< arg{-}list > \rightarrow < term >; \\ < arg{-}list > & ::= & < arg > \mid < arg >< arg{-}list > \\ < arg > & ::= & < variable > \mid < constructor > \mid < constructor >< arg > \end{array}$$

No defined-combinator appears in the argument list of rewrite rules to make easy the check of the weakly non-overlapping property discussed in section 3.3.

A program of T is a set R of rewrite rules formed according to the above rule. R defines a TRS. Reduction by pattern matching (denoted by $\Rightarrow$) and the normal form are defined as in term rewriting systems [4,5,6,11,12].

As criteria for a computation model of T, we seek the following:

(1) R has a unique normal form, if it exists.

(2) There should be an effective reduction strategy which always delivers the normal form, if it exists.

In the next two sections we shall discuss more syntactical restrictions on rewriting rules for securing these criteria.

3.3 The Church-Rosser Property for T

It is well known that if a reduction system has the Church-Rosser property then it satisfies (1). Several sufficent conditions have been proposed to guarantee the Church-Rosser property [4,6,11,12]. We have chosen the following condition for T:

(1) linear (same variables does not appear more than once in the lefthand side of a rewrite rule), and

(2) weakly non-overlapping (any critecal pair <P,Q> between overlapping rewriting rules [4,6] is trivial, i.e., P is identical to Q).

For instance, a rule

$$\text{eq x x} \;\rightarrow\; \text{true;}$$

cannot be written in T, since the lefthand side of this rule is non-linear, and

$$\begin{array}{rcl} \text{a x y} & \rightarrow & \text{x;} \\ \text{a x y} & \rightarrow & \text{y;} \end{array}$$

cannot be also written in T, since these rules, which overlap in the term (a x y), have a non-trivial critecal pair <x,y>. Though the rules

$$\begin{array}{rcl} \text{a (b 1)} & \rightarrow & \text{a 0;} \\ \text{b x} & \rightarrow & \text{0;} \end{array}$$

overlap in the term (a (b 1)), we can write them in T since they have only trivial critical pair <(a 0), (a 0)>.

Note that this sufficent condition is a special case of Huet's parallel closed condition [4,6]. Then the theory for linear and non-overlapping (i.e., there exists no critecal pair) term rewriting systems as, e.g., in [5], on which most of the sequel is based, still holds, though we have weakened the non-overlapping limitaion into the weakly non-overlapping limitaion.

The linearity can be easily checked. Regarding weakly non-overlapping, the lefthand side of rewriting rules discussed above guarantees the easy automatic check of this limitaion, since every defined-combinater occurs as the leftmost combinater (this means that we can check overlaping between two rules at only the leftmost position). Moreover many equational programs can be given by linear and (weakly) non-overlapping term rewritng systems [11]. Hence we think that the above limitation on rewriting rules is suitable to obtain theoritical correctness and practical efficiency of T.

Example 3.2 (Append of lists in T)

$$\begin{array}{rcl} \text{append nill z} & \rightarrow & \text{z;} \\ \text{append (cons x y) z} & \rightarrow & \text{cons x (append y z);} \end{array}$$

In T, integers ..., -2, -1, 0, 1, 2, ... are presented by ..., (pred (pred 0)), (pred 0), 0, (succ 0), (succ (succ 0)), ... To provide the usual meaning of integers, T has the following special rules as primitaves:

$$\begin{aligned} \text{(succ (pred x))} &\rightarrow \text{x;} \\ \text{(pred (succ x))} &\rightarrow \text{x;} \end{aligned}$$

Note that though the lefthand sides of the two rules overlap in (succ (pred (succ x))) and (pred (succ (pred x))), each critecal pair is trivial. Overlapping between a special rule presented above and a rewriting rule for a deifined-combinater, i.e.,

$$\mathrm{f\ P_1\ P_2 \ldots P_n} \quad \rightarrow \quad \mathrm{E;}$$

is avoided by assuming that the subterms (succ x) and (pred y) occuring in the arguments $\mathrm{P_1}$, $\mathrm{P_2}$, ..., $\mathrm{P_n}$ mean ($\mathrm{succ^n}$ 0) and ($\mathrm{pred^n}$ 0) ($\mathrm{n} = 1, 2, 3, \ldots$) respectively. Thus, the rule

$$\text{f (succ x) y} \quad \rightarrow \quad \text{g x y;}$$

is theoritically equal to infinite rewriting rules,

$$\begin{aligned} \text{f (succ 0) y} &\rightarrow \text{g 0 y;} \\ \text{f (succ (succ 0)) y} &\rightarrow \text{g (succ 0) y;} \\ \text{f (succ (succ (succ 0))) y} &\rightarrow \text{g (succ (succ 0)) y;} \\ &\ldots \end{aligned}$$

(Remark. Rewriting rules having the subterm (succ x) or (pred y) in the lefthand side can be regarded as membership-conditional rewriting rules; for formal tretment of membership-conditional rewriting, see [13].)

Example 3.3 (Factorial in T)

$$\begin{aligned} \text{fac 0} &\rightarrow \text{1;} \\ \mathrm{fac\ n'} &\rightarrow \mathrm{n'} * \text{(fac n);} \end{aligned}$$

(Equivalently, fac ($\mathrm{succ^{n+1}}$ 0) $\rightarrow$ mult ($\mathrm{succ^{n+1}}$ 0) (fac ($\mathrm{succ^n}$ 0)), where n = 0,1,2, ..., as explained in Section 3.6.) Note that "(fac -1)" is a normal form in T, since no rule is applicable.

3.4 Reduction of terms

It is well-known that if a reduction system has the Church-Rosser property, then it satisfies (1). Several sufficient conditions are known to guarantee (1). We have chosen the conditions of Huet [4] and Rosen [12] for T:

(i) linear (same variables does not appear more than once in the lefthand side of a rewrite rule), and

(ii) non-overlapping (no two redexes overlap each other [6])

For instance, a rule

$$\mathrm{eq\ x\ x} \rightarrow \mathrm{true;}$$

can not be written in T, since the lefthand side of this rule is not linear, and

$$\begin{array}{rcl} \mathrm{a\ x\ y} & \rightarrow & \mathrm{y;} \\ \mathrm{a\ x} & \rightarrow & \mathrm{x;} \end{array}$$

can not be also written in T, since the lefthand sides of these rules are overlap.

The linearity can be checked easily. Regarding non-overlapping, the restrictions of the lefthand side of rewrite rules guarantees the easy automatic check of non-overlapping, because the only position where defined combinators may occur is the left-most posotion of the lefthand sides of rewriting rules.

Example 3.4 (Two definitions of Factorial in T)

$$\begin{array}{rcl} \mathrm{fac\ 0} & \rightarrow & 1; \\ \mathrm{fac\ n'} & \rightarrow & \mathrm{n'} * (\mathrm{fac\ n}); \end{array}$$

This definitions represent the same function; computing factorial of given natural number.

This system is left-linear and non-overlapping. Moreover, strongly normalizing.

Note that "fac (-1)" is a normal form in T, because no rules are applicable.

3.5 Reduction strategies

Regarding the reduction strategies, some freedom in the choice is afforded. Several normalizing strategies are known for various reduction systems e.g. leftmost reduction strategy and the Gross-Knuth reduction strategy in the λ-calculus. Since our first attempt is to implement T on conventional machines, we first investigate whether the leftmost reduction strategy is normalizing in the combinatory reduction system. It is known [5] that under the following conditions the leftmost reduction strategy is indeed normalizing: In the lefthand side of a rewrite rule no variable occurences are to the left of combinator occurrences. Hereafter, we call this property of the lefthand sides of rules "left-sequentiality".

However, the left-sequentiality is severe restriction in programming. Constraint can be mitigated by suitable preprocessing. Namely, the system changes the order of arguments in the lefthand sides of rules to satisfy the left-sequentiality, if possible. We call this property "quasi left sequentiality".

Example 3.5

$$\begin{aligned} \text{member a nil} &\rightarrow \text{false;} \\ \text{member a (cons b c)} &\rightarrow \text{if (eq a b) true (member a c);} \end{aligned}$$

Suppose N $\Longrightarrow^*$ Ω (non-terminating), and M $\Longrightarrow^*$ 1 in "member N M" ($\Longrightarrow^*$ is a reflexive transitive closure of $\Longrightarrow$). "member N M" is non-terminating by the leftmost reduction strategy. We can construct a corresponding left sequential system

$$\begin{aligned} \text{xmember nil a} &\rightarrow \text{false;} \\ \text{xmember (cons b c) a} &\rightarrow \text{if (eq a b) true (xmember c a);} \end{aligned}$$

By the leftmost reduction strategy, the reduction of "xmember M N" can be stopped when M is reduced to 1 since "xmember 1 N" (as a whole) does not match with any rewrite rules, and the leftmost combinator "xmember" never diminishes by the reduction.

Of course, there are examples which cannot be converted into left sequential ones by the permutation of the arguments of lefthand sides of rules:

Example 3.6

$$\begin{aligned} \text{f true 0 x} &\rightarrow \text{1;} \\ \text{f false x 1} &\rightarrow \text{0;} \end{aligned}$$

This is the example which cannot be converted into left sequential ones by simple permutation of arguments.

In T, every rules should be left sequential or quasi left sequential. Therefore, the left most reduction is a normalizing strategy.

3.6 Syntax sugar (Summary)

To enhance programmability, the following syntax sugar is provided. These sugared forms are macro-expanded to standard forms upon reading by the system.

(1) if combinators.

Rewrite rules by argument pattern matching alone cannot always discriminate the cases programs are to handle. It would be desirable to have a conditional part (or guard) for a rewrite rule.

However, one will soon encounter difficulties in securing the Church-Rosser property when any term (reduced to true or false) is allowed in the condition. Theoretical analysis in general setting, see [9]. We decided not to introduce conditional part. Instead we introduce 'if' combinator with built-in rewrite rules

$$\begin{aligned} \text{if true p q} &\rightarrow \text{p;} \\ \text{if false p q} &\rightarrow \text{q;} . \end{aligned}$$

This will lead to rather conventional style of programming when case discrimination cannot be handled by pattern matching alone.

For example, gcd is written as

$$\text{gcd a b} \rightarrow \text{if a>b (gcd (a}-\text{b) b) (if a<b (gcd b (a}-\text{b)) a);}$$

in T.

For a more readable syntax of if, see (5) in this section.

(2) Infix notations for commonly used operations. e.g. $+$, $-$, $*$, $/$, : for add, sub, mult, div, cons, respectively.

For example,

$$0 : \text{x} : \text{y} \equiv \text{(cons (cons 0 x)y)}$$

(3) list notation

$$\begin{aligned} \{a_1,\ a_2,\ \ldots,a_n\} &\equiv \text{cons}(a_1 \text{ cons } (a_2 \ldots (\text{cons } a_n \text{ nil}) \ldots)), \\ \{\ \} &\equiv \text{nil}. \end{aligned}$$

(4) postfix notation n' for (succ n):

$$\begin{aligned} \text{succ (succ 0)} &\equiv 0'' \\ \text{fac (succ n)} &\equiv \text{fac n}' \end{aligned}$$

(5) postfix notation "if" and "otherwise" instead of a nesting "if" combinators in the righthand side of rules: We can describe

instead of

$$\text{gcd a b} \rightarrow \text{if a>b (gcd (a}-\text{b) b) (if a<b (gcd b (a}-\text{b)) a);}$$

Example 3.7 (A prime number generator using the sieve of Eratosthenes)

$$
\begin{array}{lll}
\text{intl i} & \rightarrow & \text{i} : (\text{intl } (\text{i}+1)); \\
\\
\text{sieve i} : \text{q r} & \rightarrow & \text{sieve q r} \qquad\qquad \text{if (divides i r)}, \\
 & & \text{i} : (\text{sieve q r}) \qquad \text{otherwise}; \\
\\
\text{divides i j} : \text{r} & \rightarrow & \text{false} \qquad\qquad \text{if i} < \text{j} * \text{j}, \\
 & & \text{true} \qquad\qquad \text{if (mod i j)} = 0, \\
 & & \text{divides i r} \qquad \text{otherwise}; \\
\\
\text{primes} & \rightarrow & 2 : (\text{sieve (intl 3) primes});
\end{array}
$$

3.7 Miscellaneous

1. Delta combinators: add, sub, mult, div are equipped with infinite rewrite rules on any integers theoretically. e.g.

$$
\begin{array}{rcl}
 & \vdots & \\
\text{add } 0\ 0 & \rightarrow & 0; \\
\text{add } 0\ 1 & \rightarrow & 1; \\
\text{add } 0\ 2 & \rightarrow & 2; \\
 & \vdots &
\end{array}
$$

2. In the similar vein, union of constructors is provided to abbreviate writing rewrite rules. e.g.

$$
\text{foo} \quad = \quad \{\text{succ}, \text{float}\};
$$

$$
\text{Kp (foo n)} \quad \rightarrow \quad \text{foo n};
$$

is an abbreviation of the following rules:

$$
\begin{array}{rcl}
\text{Kp (succ n)} & \rightarrow & \text{succ n}; \\
\text{Kp (float n)} & \rightarrow & \text{float n};
\end{array}
$$

4 Programming examples

4.1 Interpreter for T

One of the most interesting program example inn high-level programming language is writing its interpreter, such as Lisp in Lisp, or Prolog in Prolog. Here, we shall give an interpreter of T written in T itself. (For practical implementation, see the next section.) To illustrate the main point, we only give the matcher of arguments and the reducer for input terms. This reducer reduces term t under environment e. It is assumed that input rewrite rules are already checked for non-overlapping and linearity. The following matcher and reducer operates on a closure which is a pair of a term and its enclosing environment. The matcher is similar to a unifier used in resolution-based systems.

Let t be a term matched against s, and to be reduced under environment e. In the following program we assume the rewrite rules given below are predefined.

(1) "isconstant", "isconstructor", "isvar", "iscons", "issucc", "isdefinedcombinator", "isdeltacombinator", and "iscombinator" are predicates for constant, constructor, variable, list, integer, defined-combinator, δ-combinator, and combinators, respectively.

(2) "valof v e" extracts the value of the variable "v" from the environment "e". If the environment does not include the entry for a variable "v", the variable "v" itself is returned as a value.

(3) "hd" and "tl" are δ-combinators of projections of lists. That is,

$$\begin{array}{rcl} \mathrm{hd}\ \{\ \} & \rightarrow & \{\ \}; \\ \mathrm{hd}\ \mathrm{a} : \mathrm{b} & \rightarrow & \mathrm{a};\ \text{and} \\ \mathrm{tl}\ \{\ \} & \rightarrow & \{\ \}; \\ \mathrm{tl}\ \mathrm{a} : \mathrm{b} & \rightarrow & \mathrm{b}; \end{array}$$

as usual.

(4) "deflhs c" and "defrhs c" extract the lefthand side and the righthand side of the definition for the defined-combinator "c", respectively.

(5) "lmt" and "rmt" return the leftmost term and the rightmost term of their arguments, respectively.

(6) "butlmt" and "butrmt" return the term except the leftmost one and the term except the rightmost one of their arguments, respectively.

(7) "mkterm s t" makes a term represented by "(s t)".

(8) Rewrite rules for the same defined-combinator is kept as a list. Each definition can be extracted by "defcombinatorlist", whose argument is a name of a combinator.

(9) "funof" and "argof" are selectors for terms. "funof" selects the function part and "argof" selects the argument part of the application.

(10) "apply" is for reductions of δ-combinators.

```
match t:e s → equal (red t:e) s if (isconstant s),
              equal t s           if (iscombinator s),
              constructmatch t:e s
                                  if (isconstructor s),
              t:(valof s e):e     if (isvar s)
              (match (funof t):e (funof s)):(match (argof t):e (argof s)):e
                                  otherwise;

constructmatch t:e s →
              listmatch t:e s     if (iscons s),
              nummatch t:e s      if (issucc s),
                                  etc.
listmatch t:e s1:s2 →
              (match (hd t):e s1) & (match (tl t):e s2)
                                  if (iscons t),
              false               otherwise;

red t:e →     valof t e           if (isvar t),
              t                   if (isconstant t),
              matchtest t:e (defcombinatorlist (lmt t))
                                  if (isdefinedcombinator (lmt t)),
              apply t:e           if (isdeltacombinator (lmt t)),
              redconstructor t:e
                                  if (isconstructor (lmt t));

matchtest t:e m →
              matchtest0 (butrmt t):e (rmt t)
                                  if m=nil,
              red (defrhs (hd m)):(match t:e (deflhs (hd m)))
                                  if not (member false (match t:e (deflhs (hd m)))),
              matchtest t:e (tl m)
                                  otherwise;

matchtest0 s1:e s2 →
              (mkterm s1 s2):e
                                  if equal (red s1:e) (s1:e),
              red (mkterm s1 s2):e
                                  otherwise;
```

The main part of the interpreter is a rule "red t:[ ]" which reduces a term t under nil environment. “nummatch” and “listmatch” treat numbers and lists respectively. They are not elaborated in this paper.

The following example is the trace of the reduction of “fac 1” by the above reducer.

Example 4.1

red(fac 1):nil
$\Longrightarrow$ matchtst (fac 1):nil fac 0 $\rightarrow$ 1; fac n’ $\rightarrow$ n’$*$(fac n);
  match (fac 1):nil (fac 0)
  $\Longrightarrow$ (match fac:nil fac):(match 1:nil 0)
  $\Longrightarrow$ true:(constructormatch 1:nil 0)
  $\Longrightarrow$ true:(nummatch 1:nil 0)
$\Longrightarrow$ matchtst(fac 1):e fac n’ $\rightarrow$ n’$*$(fac n)
  match (fac 1):nil (fac n’)
  $\Longrightarrow$ (match fac:nil fac):(match 1:nil n’)
  $\Longrightarrow$ true:(match (succ 0):nil (succ n))
  $\Longrightarrow$ true:((match succ:nil succ):(match 0:nil n))
  $\Longrightarrow$ true:((equal (red succ:nil) succ):(match 0:nil n))
  $\Longrightarrow$ true:((equal succ succ):(match 0:nil n))
  $\Longrightarrow$ true:(true:(match 0:nil n))
  $\Longrightarrow$ true:(true:(equal (red 0 nil) n))
  $\Longrightarrow$ true:(true:(n:(valof 0 nil):nil))
  $\Longrightarrow$ true:(true:((n:0):nil))
$\Longrightarrow$ red (n’$*$(fac n)):(match (fac 1):nil (fac n’))
  $\vdots$
$\Longrightarrow$ red (n’$*$(fac n)):(true:(true:((n:0):nil)))
$\Longrightarrow$ apply(n’$*$(fac n)):(true:(true:((n:0):nil)))
$\Longrightarrow$ apply 1$*$(red (fac n):(true:(true:((n:0):nil))))
$\Longrightarrow$ apply 1$*$(matchtest (fac n):(true:(true:((n:0):nil))) fac 0 $\rightarrow$ 1; fac n’ $\rightarrow$ n’$*$(fac n);
  $\vdots$
$\Longrightarrow$ apply 1$*$1
$\Longrightarrow$ 1

4.2 Control structures

Oft-used control constructs are easily realized with rewrite rules, even without losing syntactic correspondence with conventional constructs.

Example 4.2 (“while do” and “repeat until” construct in T.)

while p do f on x	$\rightarrow$	while p do f on (f x)	if (p x),
		x	otherwise;
repeatuntil p do f on x	$\rightarrow$	while not p do f on (f x);	

We note that no higher order concept is necessary to implement “while” and that with no extra cost we add sugaring constants “do” and “on”.

4.3 Construction of structures

Complex data structures can be represented by juxtaposing a constructor d and constituent items, say d_1, d_2, … d_n, where d_i may be a complex structures, i. e. d d_1 d_2 … d_n.

The following example, taken from O'Donnel [11] and modified into T, is an example of constructing and manipulating polynomial.

Example 4.3 (Polynomial addition) *Polynomial* $c_0 + c_1x + \ldots + c_nx^n$ *is represented as* $c_0 + x(c_1 + \ldots c_nx^{n-1})$. *Hence we represent a polynomial as* "plus i (times x c)", *where* "plus" *and* "times" *are constructors. In the following program, we assume that the coefficients are integers.*

Then rewrite rules padd for polynomial addition are as follows:

padd (plus i a) (plus j b) → plus (add i j) (padd a b);
padd 0 (plus j b) → plus j b;
padd i′ (plus j b) → plus (add i′ j) b;
padd (neg i) (plus j b) → plus (sub j i) b;
padd (plus j b) 0 → plus j b;
padd (plus j b) i′ → plus (add i′ j) b;
padd (plus j b) (neg i) → plus (sub j i) b;
padd (times x a) (times x b) →

0	if (eql (plus a b) 0),
mult x (plus a b)	otherwise;

eql (plus i (times x a)) (plus j (times x b)) → (eql i j)&(eql a b);
eql 0 (plus j (times x b)) → false;
eql i′ (plus j (times x b)) → false;
eql (plus i (times x a)) 0 → false;
eql (plus i (times x a)) j′ → false;

Arrays which are discussed in section 4.4 are another good examples of the usage of constructors.

4.4 Array

As an experimental feature, arrays are incorporated in T. In our view, array features are important to prove the feasibility of a reduction language since many programs use arrays, especially in numeric processing.

Given architectures of von-Neumann computers, efficient processing of arrays in purely functional way is difficult (as compared with assignment-based array processing).

In T system, logically, new arrays have to be created, whenever arrays are updated. Creating a new array physically whenever updating occurs is a prohibitely expensive operation. We circumvented this difficulty by the multi-version scheme proposed by Cohen [3]. The scheme consists in associating the version number in each updated array element in such a way that maintaining the version number is entirely transparent to programmers.

With this scheme in mind, we introduce two kinds of arrays, obarray (object array) and funarray (function array).

Obarrays are arrays introduced by giving constructor "obarray", its domain, and the elements of the array, i.e.

$$\text{obarray}<\text{domain}>\ \mathrm{a}_{i_1,i_2,\ \ldots\ ,i_n}\ \mathrm{a}_{i_1+1,i_2,\ \ldots\ ,i_n}\ \cdots\ \mathrm{a}_{j_1,j_2,\ \ldots\ ,j_n}$$

where <domain> is a list of indices $\{\ i_1 : j_1,\ i_2 : j_2,\ \ldots\ i_n : j_n\}$ specifying the domain of the array.

For instance, the matrix

$$\begin{pmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \end{pmatrix}$$

may be represented by

$$\text{obarray}\{1{:}2\ 1{:}3\}\ 1\ 2\ 2\ 3\ 4\ 5$$

In the case each element of an array can be easily computed by a function (or a combinator in our terminology) of domain indices rather than actual look-ups in the store, we can specify the array in the form of computation rule. This array is called a funarray. In other words, given a combinator A with a rewrite rule

$$\mathrm{A}\ \mathrm{k}_1\ \mathrm{k}_2 \ldots \mathrm{k}_\mathrm{m} \rightarrow \mathrm{a}_{\mathrm{k}_1,\mathrm{k}_2,\ \ldots,\ \mathrm{k}_\mathrm{m}};\ (\mathrm{a}_{\mathrm{k}_1,\mathrm{k}_2,\ \ldots,\ \mathrm{k}_\mathrm{m}} \text{ is an element of an array})$$

we simply define such an array by

$$\text{funarray}<\text{domain}>\ \text{combinator.}$$

Associated projection "aget" is a combinator having a rewrite rule

$$\text{aget (funarray}<\text{domain}>\ \mathrm{A})\ \{\mathrm{k}_1, \mathrm{k}_2, \ldots, \mathrm{k}_\mathrm{m}\} \rightarrow \mathrm{A}\ \mathrm{k}_1\mathrm{k}_2\ \ldots\ \mathrm{k}_\mathrm{m};$$

$$(\Longrightarrow \mathrm{a}_{\mathrm{k}_1,\mathrm{k}_2,\ \ldots,\ \mathrm{k}_\mathrm{m}}).$$

Injection "aput" is a combinator having a rewrite rule

$$\text{aput (funarray}<\text{domain}>\ \mathrm{A})\ \{\mathrm{k}_1, \mathrm{k}_2, \ldots, \mathrm{k}_\mathrm{m}\}\ \mathrm{V} \rightarrow (\text{funarray}<\text{domain}>\ \mathrm{A'});$$

where A' is a new projection combinator satisfying

$$\begin{array}{lcll} \mathrm{A}'\ \mathrm{x}_1\ \mathrm{x}_2\ \ldots\ \mathrm{x}_\mathrm{m} & \rightarrow & \mathrm{V} & \text{if } \mathrm{x}_1 = \mathrm{k}_1 \& \mathrm{x}_2 = \mathrm{k}_2 \&\ \ldots\ \& \mathrm{x}_\mathrm{m} = \mathrm{k}_\mathrm{m}, \\ & & \mathrm{A}\ \mathrm{x}_1\ \mathrm{x}_2\ \ldots\ \mathrm{x}_\mathrm{m} & \text{otherwise;} \end{array}$$

When a funarray is updated extensively, the associated function gets complicated, and the conversion from a funarray to an obarray becomes necessary for more efficient processing.

We have two conversion combinators ! and ?, where

$$\begin{array}{rcl} ?\text{obarray} & \rightarrow & \text{funarray} \\ !\text{funarray} & \rightarrow & \text{obarray} \end{array}$$

To be more specific

$!$ (funarray$\{i_1 : j_1, i_2 : j_2, \ldots\ i_m : j_m\}$ A)
$\Longrightarrow$ obarray$\{i_1 : j_1, i_2 : j_2, \ldots\ i_m : j_m\}$ A $i_1\ i_2\ \ldots\ i_m\ \ldots$ A $j_1\ j_2\ \ldots\ j_m$
and
$?$ (obarray$\{i_1 : j_1, i_2 : j_2, \ldots\ i_m : j_m\}$ A $i_1\ i_2\ \ldots\ i_m\ \ldots$ A $j_1\ j_2\ \ldots\ j_m$)
$\Longrightarrow$ funarray $\{i_1 : j_1, i_2 : j_2, \ldots\ i_m : j_m\}$ A

We note the following relations hold between ! and ?.

(1) ! (? OA) = OA

(2) aget (? (! FA)) $\{i_1, i_2, \ldots, i_m\}$ = aget FA $\{i_1, i_2, \ldots, i_m\}$ where $\{i_1, i_2, \ldots, i_m\}$ is within the range of the domain of FA, for obarray OA and funarray FA.

We have not yet implemented this array feature and no performance data is available. However for efficient array processing, we see the importance of introducing rewrite rules which process aggregates of elements at one time rather than processing elements individually and sequentially.

5 Some considerations for practical implementation

5.1 Efficiency problems concerning T

We discuss the efficiency problem concerning T through two stages.

(a) In what kinds of processing T programs are potentially more efficient than other conventional ones?

(b) How can we realize efficient implementation of the T system itself?

To achieve efficiency improvements in both cases, the above problem has to be addressed in the following points;

(1) the elimination of redundant sequentiality (relating to (a) and (b)):

> T programs can eliminate the major source of redundant sequentiality that is created by sequential case discrimination using if–then–else constructs in conventional programming languages. It is realized by giving a rewrite rule for each case to be handled.

(2) the exploitation of parallelism inherent in the semantics of the programming language (relating to (a) and (b)):

> T shares the advantages of exploiting parallelism with other functional languages. Namely, multiple redexes (if any) in a given term can naturally give rise to parallelism. Parallel reduction strategies used in the reduction of λ-calculus are applicable to the reduction of T programs with minor modification.

(3) the exploitation of parallelism inherent in problem domains (relating to (a)).

> T does not provide special constructs for it. We claim that T programs which exploit parallelism in problem domain can be written as naturally as in other functional programming languages.

5.2 Graph copying implementation

At present we are planning a new implementation based on graph copying.

The graph copying reduction scheme consists in the following points:

(1) A rewrite rule is represented by a graph.

(2) The graph is realized in linear store (called block).

(3) For each reduction, the subgraph representing the matched righthand side is copied blockwise into a heap and the reduction is performed on the copied block.

(4) The heap is garbage-collected by a compactifying garbage collector. This storage management facilitates the storage allocation and secures the locality of the structure realized in the heap.

We do not have storage claiming functions like "CONS" in Lisp. The storage used for constructors is automatically allocated in the graph copying process. Figures 2 and 3 show the reduction process.

Figure 2: Definition of the rewrite rules of "intl"

Figure 3: The reduction of "intl 1"

The implementation is characterized by the following features:

(1) Blockwise copy of the definition of the rewrite rule is generally fast, often performed in a single instruction. (This is certainly true in IBM 360 architecture, *cf.* MVC).

(2) A sequence of basic-term t_1 t_2 ... t_n is represented by consecutive storage as is shown in Figure 2.

(3) The cells are not equally sized. For example, the size of the cell which contains a constructor is only a byte. The constructors are, from the view point of the implementation, simply tags denoting the type of the structure realized in the heap.

5.3 Pattern matching by virtual-key hashing

During the reduction, most of the execution time is spent by the selection of rewrite rules by pattern matching, according to our experience with running small, but typical programs by our interpreter. Hence the speeding up of pattern matching will greatly increase the speed of T programs. We are planning to implement a virtual-key hash method [8] to speed up pattern matching. Parallelism can be introduced to hashing, to improve the performance of hashing if necessary.

5.4 Compilation of T programs

At present, we implemented in Lisp the interpreter of T, and the compiler which compiles T programs into Lisp programs which reduce input terms by the leftmost reduction strategy. Internal form of terms, that is S-expression in Lisp, is presently used. This implementation is experimental in that it is intended to check correctness of the T programs, and to find out the bottleneck of the executions. However, some techniques to obtain the efficiency such as "computation by hash table look-ups" can be implemented in the next version of the interpreter.

In the lefthand side of rewriting rules for a defined combinator, the number of arguments is not fixed. This flexibility allows the simple treatment of the higher order functions. However, this flexibility also increases the number of trials of matching terms against the definition of the lefthand side of rewrite rules. Therefore, if the declaration of the number of arguments of defined combinators when it is fixed is introduced, the interpreter will becomes more efficient. For the more gain in speed, we should construct the compiler for T. Compiling techniques for functional programs [1, 7] can be applicable to T as well.

6 Concluding remarks

There is a certain trade-off between the generality of the language specification and the efficiency. Important point to note is that the generality should not hamper the efficiency of the basic operations of the language interpretation.

In our design of T, we tried to make T as simple, yet powerful as possible, and at the same time, we lift high-level language features to the language which can be built on T. Since the efficiency is most important in the language as low as T, we also discussed the implementation techniques and the implications to architectures. As a rough, but intuitive view, T is the language similar to Lisp in its type-free aspect.

Among other functional languages, HOPE and Miranda are closest to T. These languages can be built on T quite easily. One contribution of this paper is to make clear the critical points in the design and implementation of these languages.

Extension of T to general term rewriting system is also an interesting theme to pursue.

References

[1] L. Augustsson "A compiler for lazy ML," *Conf. Record of the 1984 ACM Symposium on Lisp and Functional Programming*, (1984) Austin 218–227.

[2] R. Burstall et. al. "HOPE: An experimental applicative language," *Conf. Record of the LISP Conference*, (1980) Stanford 136–143.

[3] S. Cohen "Multi-version structure in PROLOG," *Proc. International Conference on Fifth Generation Computer Systems*, (1984) Tokyo.

[4] G. Huet "Confluent Reductions: Abstract Properties and Applications to Term Rewriting System," *J. of ACM*, 27(4), (1980) 797–821.

[5] G. Huet, and J.-J. Levy "Call by need computations in non-ambiguous linear term rewriting system," *Rapport Laboria* 359, IRIA (1979).

[6] G. Huet, and D. C. Oppen "Equations and Rewrite Rules: a survey," *Formal Language: Perspectives and Open problems, Ed. R. Book*, Academic Press, (1980) 349–405.

[7] R. J. M. Hughes "Super-combinators: A new implementation method for applicative languages," *Conf. Record of the 1982 ACM Symposium on Lisp and functional programming*, (1982) Pittsburgh 1–10.

[8] T. Ida, and E. Goto "Parallel hash algorithms for virtual key index tables," *JIP*, 1(3), (1978) 130–137.

[9] S. Kaplan "Conditional Rewrite Rules," *Theoretical Computer Science* 33 (1984) 175–193.

[10] P. J. Landin "The next 700 programming languages," *Comm. ACM* 9(3), (1966) 157–166.

[11] M. J. O'Donnel "Equational logic as a programming language," The MIT Press (1985).

[12] B. K. Rosen "Tree-manipulation systems and Church-Rosser theorem," *J. of ACM* 20, (1973) 160–187.

[13] Y. Toyama, "Confluent term rewriting systems with membership conditions', *Proc. of the 1st International Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 308 (Springer-Verlag, 1988) 128-141.

[14] D. A. Turner "Recursive equations as a programming language in Functional programming and its applications," *Ed. J. Darlington et. al.* Cambridge University Press, (1982).