遅延ナローイング計算系のための抽象機械
(extended abstract)

鈴木太朗 井田哲雄 奥居哲

筑波大学電子情報工学系

遅延ナローイング計算系を実行する抽象機械を示す。遅延ナローイング計算系（以下では LNC と略す）は関数・論理型言語の計算のモデルである。抽象機械（以下では LNAM と略す）は LNC のコンパイラの対象機械となるものである。まず、LNC を条件付き閉包書換えシステムに基づく計算系として定義する。そして、LNAM の形式的定義を与える。LNAM の動作は LNC のプログラムを LNAM のコードに翻訳するコンパイラをとおして明らかになる。LNAM は WAM の拡張として設計されている。拡張部分は LNAM の最も重要な部分である。この拡張により遅延評価と項の簡約が実現される。

An Abstract Machine for a Lazy Narrowing Calculus
(extended abstract)

Taro Suzuki, Tetsuo Ida, Satoshi Okui

Institute of Information Sciences and Electronics, University of Tsukuba

An abstract machine for a lazy narrowing calculus is presented. The lazy narrowing calculus (to be called LNC) is a computation model for functional-logic programming language. The abstract machine (to be called LNAM) is the target machine of the compiler of LNC. We first give the overview of LNC based on conditional closure reduction scheme. We then give the formal definition of LNAM. To see how LNAM works we show the compilation of programs of LNC into LNAM code. LNAM is characterized by an extended WAM. The extension is essential since it realizes the lazy evaluation and term reduction.

1 Introduction

In the previous papers[5, 6] we presented lazy narrowing calculi to be used as a computation model for functional-logic programming languages. In this paper we present an abstract machine for a particular lazy narrowing calculus to be called *LNC* [1]. The abstract machine is an implementation model for *LNC*, and hence for the functional-logic programming language that we discussed in the previous papers. For our subsequent discussion we call the lazy narrowing abstract machine LNAM (Lazy Narrowing Abstract Machine).

2 Closure rewriting system

Theories of lazy narrowing have been studied on first-order terms that are conceptually represented in trees. In the implementations of the lazy narrowing calculus it is more convenient to treat terms as dags (directed acyclic graphs) for the following reason: Full laziness in functional programming implies that terms to be reduced are never copied during the reduction, and we want to realize this kind of full laziness in the implementation of the lazy narrowing calculus. A direct implementation of terms as dags has turned out to be an unnecessary complication, however, since it would imply direct modification of dags and copying of terms (of the righthand side) of rewrite rules. We develop in this paper a formalism which in essence realizes a more efficient treatment of rewriting of dags. It is based on the rewriting of closures. The idea goes back to refined basic narrowing of Nutt et al. [4] and of Hölldobler[3]. We first define a closure rewriting system.

2.1 Basics

Let $\mathcal{V}(\ni x,y,z)$ and $\mathcal{T}(\ni s,t,l,r)$ denote a set of variables and terms, respectively. We distinguish a special term $\perp$ which denotes an undefined value. A binding is a pair of a term and a variable, written as t/x where $t \in \mathcal{T}$ and $x \in \mathcal{V}$.

A set $\theta = \{t/x | t \in \mathcal{T}, x \in \mathcal{V}\}$ of bindings subjected to the condition:

$$t/x, t'/x' \in \theta \text{ and } x \equiv x' \Rightarrow t \equiv t'$$

is called a *substitution*.

A substitution is used as a mapping from $\mathcal{V}$ to $\mathcal{T}$. That is, for $\theta = \{t_i/x_i | i \in \mathcal{I}\}$, $\theta x_i = t_i, i \in \mathcal{I}$ and $\theta y = y$ for $y \notin \{x_i | i \in \mathcal{I}\}$.

Let $\theta[t/x]$ denote a substitution such that

$$\theta[t/x](y) = \begin{cases} t & \text{if } y = x \\ \theta(y) & \text{otherwise} \end{cases}$$

$\theta[t/x]$ in set representation is $(\theta - \{\theta x/x\}) \cup \{t/x\}$. As usual, substitutions are extended to a homomorphism from $\mathcal{T}$ to $\mathcal{T}$.

[1]In the papers[5, 6] *LNC* is called LNC_1.

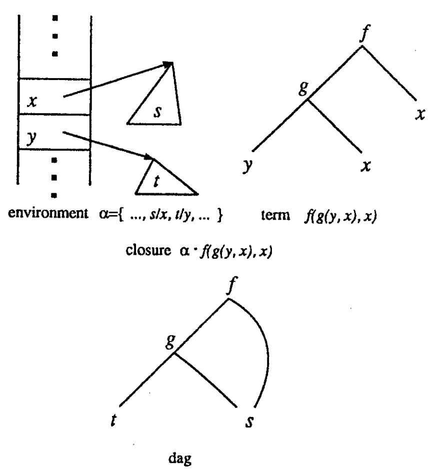

Figure 1: closure and dag

We are interested in a particular class Θ of substitutions that satisfy the following property:

$$\forall \theta \in \Theta \; \forall t \in \mathcal{T} \; \exists n \geq 0, \theta(\theta^n t) = \theta^n t \qquad (1)$$

A substitution θ in Θ is called an *environment.* $\theta^n t$ is called an *instance* of t.

A pair of an environment and a term is called a *closure.* A closure consisting of an environment α and a term t is written as $\alpha \cdot t$.

Property (1) means that for a given closure $\alpha \cdot t$ the environment α has a unique fixed point which is an instance of t. We denote the fixed point as $|\alpha \cdot t|$

The domain of a substitution θ is denoted as $Dom(\theta)$. A set of occurrences of x in a term t is denoted as $O(t,x)$, and a set of occurrences of all subterms of t is denoted as $O(t)$. An example of pictorial representation of a closure is shown in Fig. 1. Let $\Theta(\ni \alpha, \beta)$ and $\mathcal{C}$ be a set of environments and closures, respectively. Two closures $\alpha \cdot t$ and $\alpha' \cdot t'$ are equal (written as $\alpha \cdot t \equiv \alpha' \cdot t'$) iff $|\alpha \cdot t|$ and $|\alpha' \cdot t'|$ are syntactically equal.

A subterm $(\alpha \cdot t)/u$ at an occurrence u of closure $\alpha \cdot t$ is defined as follows:

$$\begin{aligned} (\alpha \cdot t)/\Lambda &= t \\ (\alpha ; x)/i.u &= \begin{cases} (\alpha \cdot \alpha(x))/i.u & \text{if } \alpha(x) \not\equiv x \\ \text{undefined} & \text{otherwise} \end{cases} \\ (\alpha \cdot f(\ldots, t_i, \ldots))/i.u &= (\alpha \cdot t_i)/u \end{aligned}$$

Let $(\alpha \cdot t)[u \leftarrow s]$ denote a closure which is created by substituting term s for the subterm at an occurrence u. It is defined formally as follows:

$$(\alpha \cdot x)[u \leftarrow s] = \alpha'[t'/z] \cdot x$$

$$\begin{aligned}
&\qquad\text{where } \alpha' \cdot t' = \alpha \cdot t[u \leftarrow s] \\
&\qquad z \text{ and } t \text{ satisfies:} \\
&\qquad \exists i \geq 0, \alpha^{i+1}(x) \equiv t \notin \mathcal{V}, \alpha^{i}(x) \equiv z \\
(\alpha \cdot f(\ldots, t_i, \ldots))[\Lambda \leftarrow s] &= \alpha \cdot s \\
(\alpha \cdot f(\ldots, t_i, \ldots))[i.u \leftarrow s] &= \alpha' \cdot f(\ldots, t_i', \ldots) \\
&\qquad\text{where } \alpha' \cdot t_i' = (\alpha \cdot t_i)[u \leftarrow s]
\end{aligned}$$

2.2 CCRS

Since we need to define *LNC* in terms of a conditional rewriting system, we will first define a closure rewriting system. Let $\mathcal{R}$ be conditional term rewriting systems of type II[1] (or standard conditional term rewriting systems [2]).

Given a conditional term rewriting system $\langle \mathcal{T}, \mathcal{R} \rangle$, we call $\langle \mathcal{C}, \mathcal{R} \rangle$ *conditional closure rewriting system* (CCRS for short) whose underlying system is $\langle \mathcal{T}, \mathcal{R} \rangle$.

A binary relation on a set of closures $\xrightarrow{i}$ $(i \geq 0)$ is inductively defined as follows:

1. $\xrightarrow{0}$ is $\emptyset$ (empty set).

2. $\alpha \cdot t \xrightarrow{i} \beta \cdot s$ if the following condition is satisfied: there exists $l \rightarrow r \Leftarrow s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$ of a new variant of a conditional rewrite rule in $\mathcal{R}$, non-variable occurrence u of term t and substituion σ such that

   (a) $|\alpha \cdot t| / u = \sigma l$

   (b) $\theta = \{((\alpha \cdot t)/u.v)/x \mid v \in O(l, x),\ x \in Dom(\sigma)\}$

   (c) $\beta \cdot s \equiv (\theta \cup \alpha \cup \gamma \cdot t)[u \leftarrow r]$ where $\gamma = \{\perp/z \mid z \in Var(s_1 \downarrow t_1, \cdots, s_n \downarrow t_n) - Var(l)\}$

   (d) $\forall\ k = 1, \ldots, n,\ \exists\ \beta_k \cdot q_k,\ \ \theta \cdot s_k \overset{i-1}{\twoheadrightarrow} \beta_k \cdot q_k \overset{i-1}{\twoheadleftarrow} \theta \cdot t_k$

   Note that θ is well defined since $\alpha \cdot t/u.v$ for all $v \in O(l, x)$ are the same because of 2 (a). Relation $\overset{i-1}{\twoheadrightarrow}$ denotes a reflexive and transitive closure of $\xrightarrow{i-1}$.

A reduction relation $\rightarrow$ w.r.t. CCRS $\langle \mathcal{C}, \mathcal{R} \rangle$ is defined as $\bigcup_{i \geq 0} \xrightarrow{i}$.

3 Lazy Narrowing Calculus Based on CCRS

The lazy narrowing calculus *LNC* is a realization of lazy narrowing based on CCRS which is the starting point for developing the interpreter and the lazy narrowing abstract machine in the following sections. The conditional reduction introduced in the previous section can be simulated by *LNC*. *LNC* realizes conditional narrowing which is the same as conditional reduction but uses unifiers instead of matchers. It allows us to solve equations. We first extend the closure defined in the previous section to a sequence of equations. That is, we also call a pair of an environment and a sequence of equations a *closure*. In particular we call a closure whose sequence part is empty an *empty closure*.

3.1 Language for *LNC*

The language for *LNC* is a language of first-order Horn clause logic equipped with only equality predicate symbol, which is used as an infix predicate symbol.

The alphabet consists of

1. function symbols (denoted by $f, g, \ldots$),

2. constructor symbols (denoted by $c, \ldots$),

3. predicate symbols ($=$) and

4. logical connectives ($\Leftarrow$, , (comma)).

Terms are classified into three categories, i.e. function terms, constructor terms and variables, depending on the leftmost symbol of the terms. A *function term* (respectively *constructor term*) is a term whose leftmost symbol is a function (respectively constructor) symbol. A *data term* is either a variable or a constructor term whose proper subterms are data terms.

A *conditional equation* is formed according to the following syntax.

$$\begin{aligned}
\langle\text{conditional equation}\rangle &::= \langle\text{head}\rangle\langle\text{body}\rangle \mid \langle\text{head}\rangle \\
\langle\text{head}\rangle &::= \langle\text{equation}\rangle \\
\langle\text{body}\rangle &::= \Leftarrow \langle\text{equation}\rangle_1, \ldots, \langle\text{equation}\rangle_n \\
&\qquad \text{where } n \geq 1 \\
\langle\text{equation}\rangle &::= \langle\text{term}\rangle = \langle\text{term}\rangle
\end{aligned}$$

A *program* $\mathcal{R}$ is a set of conditional equations. We impose the following conditions on $\mathcal{R}$.

C0. $\mathcal{R}$ is a conditional term rewriting system.

C1. A conditional equation in $\mathcal{R}$ is of the form:

$$f(d_1, \ldots, d_n) = t \Leftarrow t_1 = e_1, \ldots, t_m = e_m$$

where $d_1, \ldots, d_n$ are data terms and $e_1, \ldots, e_m$ are ground data terms.

C2. $\mathcal{R}$ is weakly non-ambiguous and left-linear.

C0 $\sim$ C2 implies that $\mathcal{R}$ is a restricted III_n CTRS.

A *goal* is formed according to the following syntax.

$$\langle\text{goal}\rangle ::= \langle\text{environment}\rangle \cdot \langle\text{equation}\rangle_1, \ldots, \langle\text{equation}\rangle_n \quad \text{where } n \geq 0$$

We impose the following conditions on a goal.

C3. A goal $\alpha \cdot E, t = d, E'$ satisfies the following conditions:

C3-1. $Var(|\alpha \cdot t|) \cap Var(|\alpha \cdot d|) = \emptyset$.

C3-2. $Var(|\alpha \cdot E|) \cap Var(|\alpha \cdot d|) = \emptyset$.

C3-3. $|\alpha \cdot d|$ is a linear data term.

- Φ_1: transformation of a conditional equation to homogeneous form

$$\Phi_1[\![f(\ldots,d,\ldots) = s \Leftarrow E]\!] = \Phi_1[\![f(\ldots,x,\ldots) = s \Leftarrow x = d, E]\!]$$

  where d is a non-variable data term and x is a fresh variable.

- Φ_2: shallowing for the bodies of a conditional equation:

$$A \Leftarrow \Phi_2[\![\ldots, s = c(\ldots,d,\ldots),\ldots]\!] = A \Leftarrow \Phi_2[\![\ldots, s = c(\ldots,x,\ldots), x = d,\ldots]\!]$$

  where d is a non-variable data term, x is a fresh variable and A is an equation.

- Φ_3: shallowing for a goal:

$$\Phi_3[\![\alpha \cdot (\ldots, s = c(\ldots,d,\ldots),\ldots)]\!] = \Phi_3[\![\alpha \cup \{\bot/x\} \cdot (\ldots, s = c(\ldots,x,\ldots), x = d,\ldots)]\!]$$

where d is a non-variable data term or a variable occurring in the initial goal, and x is a fresh variable.

Figure 2: transformation rules

3.2 Transformation to the basic forms

In order for *LNC* to be simple and efficient as a calculus, we transform conditional equations and goals to structurely simpler forms called *basic conditional equations* and *basic goals* respectively. The transformation consists of three transformation rules: transformation to a homogeneous form and shallowing on the bodies of conditional equations and on initial goals. The transformation rules are given in Fig. 2.

A conditional equation of the form $f(x_1,\ldots,x_n) = t \Leftarrow F$ where $x_1,\ldots,x_n$ are distinct variables is called *homogeneous* [8]. The transformation rule Φ_1 transforms a conditional equation to a homogeneous conditional equation.

A data term of the form $c(x_1,\ldots,x_n)$ or a variable is called *shallow*. The transformation rule Φ_2 transforms a conditional equation to a conditional equation such that the righthand sides of equations in the body are linear shallow data terms, and Φ_3 transforms a goal to a goal of the form $\alpha \cdot E, t = d, E'$ where $|\alpha \cdot d|$ is a linear shallow data term.

Φ_3 is applicable to a sequence of equations of the form $\alpha \cdot (E, s = c(\ldots,d,\ldots), E')$ even when d is a variable if it occurs also in the initial goal, whereas Φ_2 is not.

To a conditional equation, Φ_1 is first applied and then Φ_2 is applied. The resulting conditional equation is homogeneous and all the righthand sides of the equations in the body are shallow. It is called a *basic* conditional equation. $\Phi_3(E)$ is called a *basic* goal. The basic conditional equation is simply called a *rule* in the sequel, and the leftmost symbol of the lefthand side of the head equation is called the *name* of the rule.

3.3 Inference rules

The inference rules of *LNC* are as follows.

1. outermost reduction [or]

$$\frac{\alpha \cdot (s = d, E)}{\alpha' \cdot (F, s' = d, E)} \quad \begin{array}{l} |\tilde{\alpha} \cdot s| = f(\ldots)\text{where} \\ \tilde{\alpha} \text{ means } \alpha - \{\bot/x | x \in Dom(\alpha)\} \\ \qquad f(x_1,\ldots,x_n) = t \Leftarrow F \end{array}$$

   where $\alpha' \cdot s' = (\{((\alpha \cdot s)/1)/x_1, \ldots, ((\alpha \cdot s)/n)/x_n, \bot/z_1,\ldots,\bot/z_m\} \cup \alpha \cdot s)[\Lambda \leftarrow t]$ and $\{z_1,\ldots,z_m\} = Var(F) - \{x_1,\ldots,x_n\}$.

2. variable elimination of data terms [vd]

$$\frac{\alpha \cdot (s = d, E)}{\alpha[d/x] \cdot E} \quad |\tilde{\alpha} \cdot s| = x$$

   where $\alpha[d/x]$ is an environment part of $(\alpha \cdot x)[\Lambda \leftarrow d]$

3. variable elimination of constructor terms [vc]

$$\frac{\alpha \cdot (s = d, E)}{\alpha[s/x] \cdot E} \quad \begin{array}{l} |\tilde{\alpha} \cdot d| = x \\ |\tilde{\alpha} \cdot s| = c(\ldots) \end{array}$$

4. unification of constructor terms [u]

$$\frac{\alpha \cdot (s = d, E)}{\alpha[t_1/x_1,\ldots,t_n/x_n] \cdot E} \quad \begin{array}{l} |\tilde{\alpha} \cdot s| = c(\ldots), \\ |\tilde{\alpha} \cdot d| = c(x_1,\ldots,x_n) \end{array}$$

   where $t_i \equiv (\alpha \cdot s)/i$, for $i = 1,\ldots,n$

Given a goal, *LNC* tries to reduce the goal to an empty closure by repeatedly applying the inference rules. When a goal $\alpha \cdot E$ is rewritten to $\beta \cdot \varepsilon$, we write $\alpha \cdot E \longrightarrow^* \beta \cdot \varepsilon$, and $|\beta \cdot Var(E)|$ is called an *answer* of the rewrite $\alpha \cdot E \longrightarrow^* \beta \cdot \varepsilon$.

The subset {[or], [vc], [u]} of *LNC* works as conditional reduction defined in section 2.2.

In fact, *LNC* realizes conditional narrowing which is the same as (conditional) reduction but unifiers are used instead of matchers. The rule inference [vd] is used for this purpose.

It should be remarked that the bindings for the variables of initial goals are formed only by [vd].

4 Implementation of *LNC*

A first step towards the implementation of *LNC* is a design of the interpreter of *LNC*. Figure 3 shows the interpreter solve. Since no non-determinsim is involved in the selction of applicable inference rules for a given equation of the goal, a sequential implementation given in the Fig. 3 is possible without losing the completeness. Only non-trivial case is [or]. solve calls procedure $\hat{f}$, in which following actions are taken in sequence.

(1) parameter binding

(2) non-deterministic selection of a rule $f(x_1, \cdots, x_n) = t \Leftarrow F$

(3) call solve(F)

(4) call solve($t = d$)

```
procedure solve($E$)
; input : a sequence of equations $E$
; A global variable to this procedure is
; an environment $\alpha$.
begin
 if $E$ is empty then return
 let $E \equiv t = d, F$
  case type of $|\alpha \cdot t|$
  [or] : $|\alpha \cdot t|$ is a function term $f(s_1, \cdots, s_n)$
        call $\hat{f}$ with terms $s_1, \cdots, s_n, d$
        call solve($F$)
        return
  [vd] : $|\alpha \cdot t|$ is a variable $x$
        $\alpha := \{d/x\} \cup \alpha$
        call solve($F$)
        return
  [vc,u] : $|\alpha \cdot t|$ is a constructor term $c(s_1, \cdots, s_n)$
          case type of $\alpha d$
          [vc] : $|\alpha \cdot d|$ is a variable $x$
                $\alpha := \{c(s_1, \cdots, c_n)/x\} \cup \alpha$
                call solve($F$)
                return
          [u] : $|\alpha \cdot d|$ is a constructor term $c(x_1, \cdots, x_n)$
               $\alpha := \{s_1/x_1, \cdots, s_n/x_n\} \cup \alpha$
               call solve($F$)
               return
          endcase
  endcase
 end
end
```

Figure 3. Interpreter solve.

Since the fairness in non-deterministic selection of rules is difficult to achieve in sequential implementation, we impose predefined ordering of rules and employ backtrack for alternative selection of rules as in Prolog. The procedure $\hat{f}$ is the realization of the following operations: selections of the rules from the set of rules whose name if f, using backtrack if necessary, and the outermost reduction based on the selection. Figure 4 shows procedure $\hat{f}$.

5 Lazy Narrowing Abstract Machine

5.1 Overview of the machine

LNAM is specified by the following domains and the state transition map that is defined for each instruction of LNAM.

(1) program space : *Paddr* $\rightarrow$ *Instr*
(2) heap : *Daddr* $\rightarrow$ *Term*
(3) stack : D^*
(4) trail : *Vaddr**

where

Paddr, *Daddr* and *Vaddr* are a set of addresses of programs, terms and variables, respectively,

Instr is a set of instructions of LNAM,

Term is a set of representations of terms,

$D = Term \cup Paddr \cup \mathcal{N}$ where $\mathcal{N}$ is a set of non-negative integers, and

$D^* = D + D \times D + \cdots$.

The program space is an abstraction of compiled code of the program that LNAM executes. The heap is an abstraction of the collection of representations of terms, and is given by a map from *Daddr* to *Term*. We distinguish a representation of a term and a term itself since later we will discuss data structuring of terms. We write $Term = \{\rho[\![t]\!] \mid t \in \mathcal{T}\}$ where ρ is a map from a term to the representation of a term and is defined in section ?. The stack is an abstraction of an environment and the workspace, and is represented as a sequence of elements in D. LNAM is equipped with a trail which functions like a trail of Prolog systems. The (abstract) trail is represented as a sequence of addresses of variables that are trailed.

The configuration of LNAM is defined by the triple

$$(prog, s_0, \mathcal{M})$$

where $prog : Paddr \rightarrow Instr$ is a program space, s_0 is an initial state of LNAM and $\mathcal{M}$ is a state transition map. The state of LNAM is defined by the following 9-tuple

$$(p, heap, stack, trail, a, e, b, cp, s) \qquad (2)$$

where $p \in Paddr$, $a \in D$, $e \in \mathcal{N}$, $b \in \mathcal{N}$, $cp \in Paddr$, $s \in Daddr \times \mathcal{N}$, and represent the values of program counter P, accumulater A, environment pointer E, backtrack pointer B, continuation pointer CP, structure pointer S, respectively.

Program counter P is a pointer to the program space. Continuation pointer CP which points to a return address of a procedure is also used to address the program space. Accumulater A is used to hold the intermediate results of the computation and serves as the communication workspace among instructions. Registers E and B are pointers to the stack. In addition to the above pointers, LNAM is equipped with auxiliary pointers, S which points to the structure being processed, H which points to the free area of the heap, T which points to the start of the stack and TR which points to the start of the trail.

```
procedure $\hat{f}$
; input : $n$ terms and 1 data term
; $\alpha$ is a global variable representing an environment
; as in solve
; This solves the equation$f(a_1,\ldots,a_n) = d$
; using a set$\{f(x_1,\ldots,x_n) = t_i \Leftarrow F_i | i \in \{1,\ldots,k\}\}$
begin
 $\alpha := \alpha \cup \{a_1/x_1,\cdots,a_n/x_n, d/y, \perp/z_1, \cdot, \perp/z_m\}$
 where $z_1,\cdots,z_m$ are extra-variables.

 ; The following correspond to the first rule
 ; $f(x_1,\cdots,x_n) = t_1 \Leftarrow F_1$.
          create a choice-point for the backtrack to [c_$f_2$]
          call solve($F_1$)
          call solve($t_1 = y$)
          return
 [c_$f_2$] : create a choice-point for [c_$f_3$]
          ; The following correspond to the second rule
          ; $f(x_1,\cdots,x_n) = t_2 \Leftarrow F_2$.
          call solve($F_2$)
          call solve($t_2 = y$)
          return
 [c_$f_3$] :
              ⋮
 [c_$f_k$] : ; The following correspond to the last rule
          ; $f(x_1,\cdots,x_n) = t_k \Leftarrow F_k$.
          remove a choice-point that have been set
          before the entry to $\hat{f}$.
          call solve($F_k$)
          call solve($t_k = y$)
          return
end
```

Figure 4: procedure $\hat{f}$

Semantically, the stack consists of the following three frames:

environment frame $env \in \{v(E)\} \times \{v(CP)\} \times Term^*$

choice point frame $choice \in \{v(P)\} \times \{v(B)\} \times \{v(E)\} \times \{v(CP)\} \times \{v(H)\} \times \{v(TR)\}$

argument frame $args \in \{Daddr \cup \mathcal{N}\}^*$,

where $v(X)$ for $X = E, CP, etc.$ denotes the value held in the pointer X.

The initial state s_0 is $(0, heap_0, \varepsilon, \varepsilon, \perp, \perp, \perp, \perp, \perp)$, where $heap_0$ is the heap in the initial state.

5.2 Data structure of terms

For the discussion of instructions we need to give a more concrete view of data structures of terms. A basic building block of data structures is called a *word*. A word is a pair of *tag* and *val* and is written as $\langle tag\ val\rangle$, where

$tag \in \{\texttt{var}, \texttt{const}, \texttt{list}, \texttt{struc}, \texttt{func}\}$ and $val \in Daddr \cup \mathcal{N}$

Tags `var`, `const`, `list`, `struc` and `func` are used to distinguish the terms, each distinguishing variables, constants, list, structures, funcion terms, respectively. This classification of terms is slightly different from the one we showed in section ?. Since in programming, constructor terms whose arity is 0 are treated differently, and since lists are heavily used structures, we distinguish three subclasses of constructor terms, i.e., constants, lists and structures.

We now define a map $\rho[\![\ ]\!]$ as follows:

variable x
$$\rho[\![x]\!] = \langle var\ addr(\rho[\![x]\!])\rangle$$
constant k
$$\rho[\![k]\!] = \langle const\ k\rangle$$
list $[t_1 \mid t_2]$
$$\rho[\![[t_1 \mid t_2]]\!] = \langle list\ w\rangle$$
$$\text{and}\quad heap(w) = \rho[\![t_1]\!],\ heap(w+1) = \rho[\![t_2]\!]$$
structure $c(t_1,\ldots,t_n)$
$$\rho[\![c(t_1,\ldots,t_n)]\!] = \langle struc\ w\rangle$$
$$\text{and}\quad heap(w) = \langle const\ c/n\rangle,\ heap(w+1) = \rho[\![t_1]\!],\ldots,\ heap(w+n) = \rho[\![t_n]\!]$$
function term $f(t_1,\ldots,t_n)$
$$\rho[\![f(t_1,\ldots,t_n)]\!] = \langle func\ w\rangle$$
$$\text{and}\quad heap(w) = \langle const\ f/n\rangle,\ heap(w+1) = \langle var\ w+1\rangle,\ heap(w+2) = \rho[\![t_1]\!],\ldots,\ heap(w+n+1) = \rho[\![t_n]\!]$$

Here, $addr(t)$: $Term \rightarrow Daddr$ is a map which returns an address of a word where term t is allocated. Unbound variables are represented as a self-referencing pointer. When a binding of x with t is formed, val part of $\rho[\![x]\!]$ is replaced with $addr(\rho[\![t]\!])$. The definition of variables, constants, lists, and structures are the same as in WAM. Function terms are original in LNAM. The function terms are represented by the header $\langle func\ w\rangle$ and the body which consists of words from $heap(w)$ to $heap(w+n+1)$. $heap(w)$ is a word consisting of the function symbol and the associated arity. $heap(w+2),\ldots,heap(w+n+1)$ represent a subterm of occurrences $1,\ldots,n$. $heap(w+1)$ is a special field called a *cache* field. Function terms are reduced during the computation and the result of the reduction is stored in the cache field. This special arrangement is necessary to realize the so-called lazy evaluation.

5.3 Instructions

We give the instruction set of LNAM in Fig. 5.

These instructions are rigorously defined using the state transition map $\mathcal{M}$[7]. Figure 5 shows that LNAM are based on WAM. Underlined instructions are original to LNAM. push instructions are provided since LNAM is a stack-based machine. LNAM has only single accumulator A, whereas WAM is provided with (conceptually infinite) accumulators. Instructions `push_function`, `put_function`, `call_function` and `jump_if_function` are provided for the

- push instructions

push_variable	x
push_value	x
push_nil	
push_constant	k
push_list	
push_structure	c/n
push_function	f/n
push_args	

- put instructions

put_variable	x, y
put_value	x, y
put_nil	
put_constant	k, x
put_list	x
put_structure	$c/n, x$
put_function	$f/n, x$

- get(accumulator) instructions

get_variable	x
get_value	x
get_nil	
get_constant	k
get_list	
get_structure	c/n

- get(stack) instructions

get_variable	x, y
get_value	x, y
get_nil	x
get_constant	k, x
get_list	x
get_structure	$c/n, x$

- unify instructions

unify_void	n
unify_variable	x
unify_value	x
unify_local_value	x
unify_nil	
unify_constant	k

- call instructions

call	p
call_function	
execute	p
proceed	
jmp	l
allocate	n
deallocate	

- try instructions

try_me_else	l
retry_me_else	l
trust_me_else_fail	

- indexing instructions

switch_on_term	x, lf, lc, ll, ls
jmp_if_function	x, lf

- unification with a cached term (bind instructions)

bind_variable	x
bind_value	x
bind_constant	k
bind_list	
bind_structure	c/n

- others

fail	
pop	n

Figure 5. Instruction Set of LNAM

manipulation of function terms. call_function is used for the call of the procudure that is explained in section 6. bind instructions are provided to realize the lazy evaluation. For example, instruction bind_variable i performs the follwing:

$$\mathcal{M}[\![\texttt{bind\_variable} \;\; i]\!]\,(p, heap, stack, trail, a, e, b, cp, \perp)$$
$$\Longrightarrow$$
$$\langle func\, x\rangle \leftarrow a;$$
$$(p+1, heap, \langle var\ x+1\rangle : stack[i \leftarrow \langle var\ x+1\rangle],$$
$$trail, a, e, b, cp, \perp)$$

How bind instructions are used is explained in the full version of the paper[7].

6 Compilation of programs

A program is divided into sub-programs $P_1, \ldots, P_k$ where each $P_j, j = 1, \ldots, k$ consist of rules of the same name. Let f_j be the name of the rules in P_j. $P_j, j = 1, \ldots, k$ are appropriately ordered and then compiled into a procedure $\hat{f}_j$.

The code of $\hat{f}_j$ is following.
Let $r_1, \ldots, r_m$ (ordered in this way) are rules in P_j.

When $m > 1$,

```
          try_me_else       $\$c\_f_2$
          $C[\![r_1]\!]$
$\$c\_f_2$ :  retry_me_else     $\$c\_f_3$
          $C[\![r_2]\!]$
$\$c\_f_3$ :
              ⋮
$\$c\_f_m$ :  trust_me_else_fail
          $C[\![r_m]\!]$
```

otherwise,

```
$C[\![r_1]\!]$
```

where C be a function which compiles a rule.

The definition of C is given below:

$C[\![f(x_1, \ldots, x_n) = t \Leftarrow E_1, \ldots, E_r]\!]$
$\Longrightarrow$

```
                  allocate    $m$
                  $C_e[\![E_1]\!]$
$\$Next_1$ :         $C_e[\![E_2]\!]$
                    ⋮
$\$Next_{r-1}$ :     $C_e[\![E_r]\!]$
$\$Next_r$ :         $C_e[\![t = y]\!]$
$\$Next_{r+1}$ :     deallocate
                  pop    $n+1$
                  proceed
```

The instruction allocate m allocates a frame whose size is determinded by m in the stack. Following allocate is the code for equations $E_1, \ldots, E_r$ in this order. Equation $t = y$ is for the unification of the result of this rewrite with the righthand side of the equation that invokes the call of $\hat{f}$. The instruction deallocate deallocates the frame. The instruction pop $n+1$ deallocates the argument frame. The instruction proceed return control to the caller.

We next give the definition of C_e which is the most important part of our compilation scheme. C_e is given an equation $t = d$, and, depending upon the types of t and d, generates the code summarized in Table 1. Because of the space limitation we only give typical cases (a) $\sim$ (f).

(a) $C_e[\![x = d]\!]$
$\Longrightarrow$

```
case $d$ of
  variable $y$:
    put_variable      $y, x$
  constant $k$:
    put_constant      $k, x$
  list $[x_1 \mid x_2]$:
    put_list    $x$
    unify_variable    $x_1$
    unify_variable    $x_2$
  structure $c(x_1, \ldots, x_n)$:
    put_structure     $c/n, x$
    unify_variable    $x_1$
      ⋮
    unify_variable    $x_n$
endcase
```

$t\backslash d$	first occurrence of variable $d \equiv y$	non-first occurrence of variable $d \equiv y'$	shallow constructor $d \equiv b(x_1, \ldots, b_m)$
first occurrence of variable $t \equiv x$	(a) set d to t		
non-first occurrence of variable $t \equiv x'$	(b) compile $\alpha \cdot (t = d)$		
function term $t \equiv f(s_1, \ldots, s_n)$	(c) push arguments $s_1, \ldots, s_n$ onto the stack and call procedure $\hat{f}$		
constructor $t \equiv c(s_1, \ldots, s_n)$	(d) set t to d	(e) unify t with d	(f) if $c \equiv b$ and $n \equiv m$ set $s_1, \ldots, s_n$ to $x_1, \ldots, x_n$,respectively otherwise fail

Table.1 Compilation of equation $t = d$ according to the types of t and d

(b) $\mathcal{C}_e[\![x' = d]\!]$ where x' is the non-first occurrence.

$\Longrightarrow$

```
         jmp_if_function   $t, \$fun$
         $\mathcal{C}'_e[\![\alpha \cdot (x' = d)]\!]$   % if $\alpha \cdot x'$ is
                                 % not function term
         jmp   $\$Next$
$\$func$:  $\mathcal{C}'_e[\![\alpha \cdot (x' = d)]\!]$   % if $\alpha \cdot x'$ is
                                 % function term
         jmp   $\$Next$
```

where $\mathcal{C}'_e$, when $\alpha \cdot x'$ is function term $f(t_1, \ldots, t_n)$, is given below:

$\mathcal{C}'_e[\![f(t_1, \ldots, t_n) = d]\!]$

$\Longrightarrow$

```
case $d$ of
   variable $y$:
      push_args
      bind_variable $y$
      call_function
   variable $y'$:
      push_args
      bind_value $y'$
      call_function
   constant $k$:
      push_args
      bind_constant $k$
      call_function
   list $[x_1 \mid x_2]$:
      push_args
      bind_list
      unify_variable   $x_1$
      unify_variable   $x_2$
      call_function
   structure $c(x_1, \ldots, x_n)$:
      push_args
      bind_structure   $c/n$
      unify_variable   $x_1$
        ⋮
      unify_variable   $x_n$
      call_function
endcase
```

(c) $\mathcal{C}_e[\![f(s_1, \ldots, s_n) = d]\!]$

$\Longrightarrow$

```
$Gen_{push}[\![s_1]\!]$
  ⋮
$Gen_{push}[\![s_n]\!]$
$Gen_{push}[\![d]\!]$
call  $f$
```

where
$Gen_{push}[\![x]\!]$ = push_variable
$Gen_{push}[\![x']\!]$ = push_value
$Gen_{push}[\![k]\!]$ = push_constant
⋮

(d) $\mathcal{C}[\![k = y]\!]$

$\Longrightarrow$

```
put_constant   $k, y$
```

(e) $\mathcal{C}[\![k = y']\!]$ where y' is the non-first occurrence.

$\Longrightarrow$

```
get_constant   $k, y'$
```

(f) $\mathcal{C}[\![k = d]\!]$

$\Longrightarrow$

```
case $d$ of
   constant $k_1$:
      get_constant   $k, k_1$
   list $[x_1 \mid x_2]$:
      fail
   structure $c(x_1, \ldots, x_n)$:
      fail
endcase
```

Finally, Fig. 6 shows an example of the program and its generated code.

7 Concluding Remarks

We have shown the essence of the lazy narrowing calculus LNC and its abstract machine LNAM. To fill the gap between the calculus and the machine we briefly explained the compilation of the program of LNC. LNAM together with the LNC compiler have been implemented and used for our research purposes. For further details the readers are referred to the full version of the paper[7].

References

[1] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: confluence and termination. *J. Comput. Syst. Sci.*, 32:323–362, 1986.

[2] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *Proc. Int. Conf. 5th Generation Comp. Syst.*, pages 337–346, 1988.

[3] S. Hölldobler. Foundations of equational logic programming. *LNAI*, 353, 1989.

[4] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. SEKI Report SR-87-07, Universität Kaiserslautern, 1987.

[5] S. Okui and T. Ida. *Lazy Narrowing calculi.* To be published as Technical Report ISE-TR-92-97, University of Tsukuba.

[6] S. Okui and T. Ida. *Narrowing calculi for lazy functional-logic programming languages* (in Japanese). submitted for publication.

[7] T. Suzuki, T. Ida, and S. Okui. An abstract machine for a lazy narrowing calculus. To be published as ISE Technical Report, University of Tsukuba,1992.

[8] M. H. van Emden and J. W. Lloyd. A logical reconstruction of Prolog II. *J. Logic Prog.*, 4:265–288, 1974.

conditional equation

$$len([]) = 0 \Leftarrow . \tag{3}$$
$$len([X|Y]) = succ(len(Y)) \Leftarrow . \tag{4}$$

basic conditional equation

$$len(X) = 0 \Leftarrow X = []. \tag{5}$$
$$len(X) = succ(len(Y)) \Leftarrow X = [Z|Y]. \tag{6}$$

LNAM instruction code

```
compiled code for (5)
              try_me_else          $C1
              allocate             0
              jmp_if_function      A0,$L1
              get_nil
              jmp                  $L2
       $L1 :  push_args
              bind_nil
              call_function
       $L2 :  get_constant         0,A1
              deallocate
              pop                  2
              proceed
compiled code for (6)
       $C1 :  trust_me_else_fail
              allocate             3
              jmp_if_function      A0,$L3
              get_list
              unify_variable       Y0
              unify_variable       Y1
              jmp                  $L4
       $L3 :  push_args
              bind_list
              unify_variable       Y0
              unify_variable       Y1
              call_function
       $L4 :  put_function         len/1,Y2
              unify_value          Y1
              get_structure        succ/1,A1
              unify_value          Y2
              deallocate
              pop                  2
              proceed
```

Figure 6. compiled code for len/1