TR-151

Features of A Meta-Unification Based Languge Talos

by
T. Kanamori
(Mitsubishi Electric Corp.)

October, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Features of A Meta-Unification Based Language Talos

Tadashi KANAMORI

Mitsubishi Electric Corporation
Central Research Laboratory
8-1-1 Tsukaguchi-Honmachi
Amagasaki, Hyogo, JAPAN 661

Abstract

In this paper we describe the language features of a meta-unification based language Talos. In Talos everything is done by controlled sequences of meta-uification, as is by controlled sequences of unification in Prolog. This is a generalizations of the conventional term rewriting as well. We present four features considered special meta-unification processes such as (1) conditional computation (generalization of the **where** abstraction), (2) nondeterministic computation (Prolog-like backtracking search), (3) "call by need" computation (control of delay and force of evaluation) and (4) computation with streams (Lucid-like description of iterations). We describe the behaivor of the Talos interpreter by stepwise extension with several Talos programs.

Keywords : Equational Theories,Term Rewriting Systems,Unification, Operational Semantics.

Contents

1. Introduction

Prolog [5] is a relational language based on first-order predicate calculus. Operational semantics of Prolog is usually explained by the SLD-resolution, a strategy of the resolution complete for Horn clauses. Prominent features of Prolog are procedure invocation by unification and nondeterministic search (automatic backtracking). Results of procedures are passed through variables within each clause, while,in functional programs like Lisp, nested composition of functions is the main construct.

Functional programming languages are more classical and share semantical clearness with Prolog ([4],[11],[21]). They can be considered special logic programming languages based on equational logic. When an equation is considered a term rewriting rule, equational logic turns into computation, which is the basis of the operational semantics of functional programs. Though functional programs are superior to Prolog in some points (readability etc), they lack some powerfull features of Prolog such as nondeterministic search. When we accomodate these features to functional programming, we need to carry it out not by an *ad hoc* device but by a unified approach. Several such attempts have been done from different point of views ([3],[7],[9],[20]).

In this paper we describe the language features of a meta-unification based language Talos. In Talos everything is done by controlled sequences of meta-uifications, as is by controlled sequences of unifications in Prolog. This is a generalization of the conventional term rewriting as well. Both invocations of functions by unification and automatic backtracking are integrated into Talos.

We describe four features and the behaivor of Talos interpreter by stepwise extension. These computation can be considered special meta-unification processes. After reviewing the basic computation based on usual term rewriting systems and the Fay-Hullot's meta-unification algorithm in section 2, we introduce conditional computation in section 3, which is a generalization of the **where** abstraction. In section 4, we present how to utilize the interpreter to compute nondeterministic functions and show the Prolog-like backtracking search. In section 5, we introduce evaluation strategies and control mechanism to delay anf force evaluation. In section 6, we introduce stream as a data type, which enables us to program using Lucid-like description of iterations. Lastly in section 7, we discuss the relations to other works and problems left for future.

In this paper we assume familiarity with (many-sorted) equational logic and term rewriting systems. As syntactical variables, we use X, Y, Z for variables, f, g, h for function symbols, a, b, c for constants, r, s, t, γ, δ for terms, u, v for occurrences and $\theta, \sigma, \tau, \eta, \varsigma$ for substitutions, possibly with primes and subscripts. $\equiv$ is used to denote the syntactical identity. We denote the set of all terms on a signature Σ and variables $\mathcal{V}$ by $\mathcal{T}(\Sigma \bigcup \mathcal{V})$ (or simply $\mathcal{T}$), the set of all ground terms on a signature Σ by $\mathcal{G}(\Sigma)$ (or simply $\mathcal{G}$), sets of all variables in a syntactical object e by $\mathcal{V}(e)$, subterm of t at an occurrence u by t/u, replacements of a subterm of t at an occurrence u with a term s by $t[u \Leftarrow s]$ and restriction of a substitution σ to a set of variables V by $\sigma|V$. (see [12],[15]).

2. Preliminaries

2.1. Definitions of Data Types and Basic Functions

Definition of data types in Talos is similar to the algebraic specification of abstract data types except the separation of constructors. Constructors are operators from which every instance of types is freely and uniquely constructed. For example, a data type *list* has two constructors *nil* ([ ]) and *cons* ([|]). (We follow the DEC-10 Prolog-like syntax [19].) The choice of constructors is left to programmers.

Example 2.1.1. A data type *number* is defined as follows.

```
data number = new.
  constructor.
    zero.
    suc(N:number).
  operator.
    add(M,N:number):number.
      M+0=M.
      M+(N+1)=(M+N)+1.
    less-than(M,N:number):boole.
      0<N+1=true.
      M<0=false.
      M+1<N+1=M<N.
end.
```

We assume a data type *boole* is already defined. $0, +, 1, <$ are built-in symbols and $suc^i(N)$ is represented by $N + i$.

Basic functions are also defined by a set of equations.

Example 2.1.2. A function appending two lists is defined by equations as follows.

```
function append(L,M:list):list.
  append([ ],M)=M.
  append([X|L],M)=[X|append(L,M)].
end.
```

The equations in the definition of data types and functions form the axioms of an *equational theory $\mathcal{E}$ corresponding to* the Talos program P. These equations are considered rewriting rules from the left hand side to the right hand side, which forms the axioms of a *term rewriting system $\mathcal{R}$ corresponding to* the Talos program P.

By separating constructors in the definition of data types, we have the signature Σ partitioned into $\mathcal{C} \uplus \mathcal{D}$. We call operators in $\mathcal{C}$ the *constructors.* (We assume there are at least one constant constructors.) A *constructor term* is a term on $\mathcal{C}$. The set of all constructor terms is denoted by $\mathcal{T}_C$ and the set of all ground constructor terms is denoted by $\mathcal{G}_C$. A *semi-constructor term* is either a variable or a term whose root function symbol is a constructor.

A binary relation $\rightarrow$ on the set of all terms $\mathcal{T}$ is said to be *stable* iff $\sigma(s)\rightarrow\sigma(t)$ for any substitution σ when $s\rightarrow t$ and said to be *compatible* iff $r[u\Leftarrow s]\rightarrow r[u\Leftarrow t]$ for any occurrence u of r when $s\rightarrow t$ ([12] p.809). Let $\underline{R} = (\rightarrow, \mathcal{T})$ be a compatible stable relation, $\rightarrow^*$ be the reflexive transitive closure of $\rightarrow$ and $\downarrow$ be a relation based on $\rightarrow$ such that $s \downarrow t$ iff there exists r satisfying $s\rightarrow^* r$ and $t\rightarrow^* r$. $\underline{R}$ is said to be *confluent* when,for any terms t,t_1,t_2 such that $t\rightarrow^* t_1$ and $t\rightarrow^* t_2$, there exists a term t' such that $t_1\rightarrow^* t'$ and $t_2\rightarrow^* t'$. When $\underline{R} = (\rightarrow, \mathcal{T})$ is confluent, $\underline{R}$ defines a binary congruence relation $=_{\underline{R}}$ which is the reflexive symmetric transitive closure of $\rightarrow$. $\underline{R}$ is said to be *terminating* when,for any term t_0,there is no infinite derivation $t_0\rightarrow t_1\rightarrow t_2\rightarrow\cdots$ such that $t_i\rightarrow t_{i+1}$ is in $\underline{R}$ $(0\leq i)$. A term s is said to be *in $\underline{\mathcal{R}}$-normal form* when there is no t such that $s \rightarrow t$ is in $\underline{\mathcal{R}}$. A term t is called *$\underline{\mathcal{R}}$-normal form* of a term s and denoted by $s\downarrow$ when $s \rightarrow^* t$ holds for $\underline{\mathcal{R}}$ and t is in $\underline{\mathcal{R}}$-normal form. A substitution η is said to be *$\underline{\mathcal{R}}$-normalized* iff $\eta(X)$ is in $\underline{\mathcal{R}}$-normal form for all X.

2.2. Meta-Unification for Equational Theories with Constructors

2.2.1. Narrowing for Equational Theories

Let $\mathcal{E}$ be an equational theory. s and t are said to be *$\mathcal{E}$-unifiable* iff there exists a substitution θ such that $\theta(s) =_{\mathcal{E}} \theta(t)$. The set of all $\mathcal{E}$-unifiers of s and t is denoted by $\mathcal{U}_{\mathcal{E}}(s,t)$. In general the most general unifier does not always exists when $\mathcal{E} \neq \emptyset$.

Example 2.2.1. Let $\mathcal{E}$ be the equational theory of the associativity,i.e. $\mathcal{E} = \{X \bullet (Y \bullet Z) = (X \bullet Y) \bullet Z\}$ and s, t be terms $X \bullet a$ and $a \bullet Y$ respectively. Then

$\theta_i = < X \Leftarrow a \bullet (a \bullet (a \bullet \cdots (a \bullet a))), Y \Leftarrow a \bullet (a \bullet (a \bullet \cdots (a \bullet a))) >$

are all meta-unifiers of s and t, where the term substituted for X and Y consists of i a's. There is no $\preceq$ relations between these substitutions. Hence there is no single most general meta-unifier of s and t.

The set of all variables X such that $\sigma(X) \not\equiv X$ is called the *domain* of σ and denoted by $\mathcal{D}(\sigma)$. The set of all variables in $\sigma(X)$ for all $X \in \mathcal{D}(\sigma)$ is called the *variables introduced by* σ and denoted by $\mathcal{I}(\sigma)$. A substitution σ is said to be *away from* a set of variables W when $\mathcal{I}(\sigma) \bigcap W = \emptyset$.

Let s be a term, W be a set of variables containing $\mathcal{V}(s)$ and $\gamma \rightarrow \delta$ be a rule in $\mathcal{R}$. A substitution σ is called a *narrowing substitution* of s away from W, if a nonvariable subterm s/u and the left hand side γ is unifiable by a most general unifier σ. We assume $\mathcal{V}(s)$ is away from W by renaming away the variables in $\gamma \rightarrow \delta$ from W. s is said to be *narrowed to* $t \equiv \sigma(s[u \Leftarrow \delta])$ and denoted by $s \rightsquigarrow t$. The set of narrowing substitutions σ of s away from W is denoted by $NS(s, W)$. In particular,when $s \rightsquigarrow t$ and $\sigma|\mathcal{V}(s)$ is the empty substitution $<>$, s is said to be *reduced to* $t \equiv \sigma(s[u \Leftarrow \delta])$ and denoted by $s \rightarrow t$. Note that the reduction is included in the narrowing, i.e.,$\rightarrow \subseteq \rightsquigarrow$.

Example 2.2.2. Let s be $append(A, [b, c])$ and u be the occurrence of s itself. Because $append(A, [b, c])$ is unifiable with the left hand side of the second rewrite rule in the definition of $append$ by an m.g.u. $\sigma = < A \Leftarrow [X'|L'], X \Leftarrow X', L \Leftarrow L' M \Leftarrow [b, c] >$, s is narrowed to $\sigma([X|append(L, M)])$, i.e. $[X'|append(L', [b, c])]$. Because there is no other narrowing substitution, $NS(s, W)$ includes only one substitution. (In general,$NS(s, W)$ may be empty or include more than two substitutions.).

The reductions for a term rewriting system $\mathcal{R}$ define a relation $\underline{\mathcal{R}}$ on $\mathcal{T}$ by

$\underline{\mathcal{R}} = \{s \rightarrow t | s \rightarrow t \text{ is an instance of a reduction in } \mathcal{R}\}$.

We assume $\mathcal{E}$ and $\underline{\mathcal{R}}$ satisfies the following three conditions.

(A) $\underline{\mathcal{R}}$ is confluent and terminating.

(B) Every left hand side of the equations in $\mathcal{E}$ is not a semi-constructor term.

(C) For any ground term $s \in \mathcal{G}$, there exists a ground constructor term $t \in \mathcal{G}_C$ such that $s \rightarrow^* t$.

Remark. The condition (B) implies for every ground constructor terms $s, t \in \mathcal{G}_C$ we have $s =_{\mathcal{E}} t$ only if $s \equiv t$. The condition (C) with the condition (B) guarantees that the initial algebra of $\mathcal{E}$ is isomorphic to $\mathcal{G}_C$. A sufficient condition for (C) is investigated in [13].

2.2.2. Fay-Hullot's Meta-Unification Algorithm

Suppose we have a confluent and terminating term rewriting system $\mathcal{R}$ corresponding to an equational theory $\mathcal{E}$. Then the following algorithm is a nondeterministic $\mathcal{E}$-unification algorithm by Fay [6] (revised by Hullot [16]), which is complete in the sense that, for any $\mathcal{E}$-unifier ρ, there exists a path to generate a $\mathcal{E}$-unifier θ more general "modulo $\mathcal{E}$" than ρ. Note that W is initialized to W_0 ($\supseteq \mathcal{V}(s,t)$) before *meta-unify*$(s,t)$ and global during the computation.

```
meta-unify($s, t$:term) : substitution;
$\theta := <>$;
**repeat**
  **when** $s$ and $t$ are unifiable by $\theta'$ away from $W$
    stop with answer $\theta' \circ \theta$
  **when** $NS(s,W) \neq \emptyset$
    select $\sigma \in NS(s,W)$ and let the corresponding rule be "$\gamma \rightarrow \delta$" ($\sigma(\gamma) \equiv \sigma(s/u)$);
    **if** there exists a variable $X \in W$ for which $\sigma \circ \theta(X)$ is not in $\mathcal{R}$-normal form
    **then** stop with failure
    **else** $s := \sigma(s[u \Leftarrow \delta])$; $t := \sigma(t)$; $\theta := \sigma \circ \theta$; $W := W + I(\sigma)$
  **when** $NS(t,W) \neq \emptyset$
    select $\sigma \in NS(t,W)$ and let the corresponding rule be "$\gamma \rightarrow \delta$" ($\sigma(\gamma) \equiv \sigma(t/v)$);
    **if** there exists a variable $X \in W$ for which $\sigma \circ \theta(X)$ is not in $\mathcal{R}$-normal form
    **then** stop with failure
    **else** $s := \sigma(s)$; $t := \sigma(t[v \Leftarrow \delta])$; $\theta := \sigma \circ \theta$; $W := W + I(\sigma)$
**endrepeat**
```

Figure 2.2.2. Fay-Hullot's Meta-Unification Algorithm

Example 2.2.3. Let s be $append(A, [b,c])$ and t be $[a|B]$. Because s and t is not unifiable and $NS(t,W) = \emptyset$, the Fay-Hullot's algorithm selects the second **when** and s is narrowed to $[E'|append(L', [b,c])]$ by $\sigma|V = < A \Leftarrow [E'|L'] >$. Then in the next repetition it is unifiable with $[a|B]$ by $\theta' = < X' \Leftarrow a, L' \Leftarrow L'', B \Leftarrow append(L'', [b,c]) >$. Hence $< A \Leftarrow [a|L''], B \Leftarrow append(L'', [b,c]) >$ is a meta-unifier of s and t.

Remark. At then branches in the Fay-Hullot's algorithm, unnecessary search detected in the **if** test is pruned away. We call the check *normalization check*. Note that,even without it, the meta-unification algorithm is still complete.

2.2.3. Interpreter for Basic Computation

A query is a coditional term of the form

$?\text{-}t$ **when** $s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m$.

For such a query,the Talos interpreter generates a set of equations

$\mathcal{E}_0 = \{ !Value = t, s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m \}$,

where $!Value$ is a special variable annotated by "!". (Variables with "!" annotations are called *eager variabls*. Variables without "!" annotations are called *lazy variables*. The annotation "!" of a variable $!X$ is inherited to variables in t, whenever $!X$ is instanciated to t.) Then the Talos interpreter initialize W to $W_0(\supseteq \mathcal{V}(\mathcal{E}_0))$, compute $\theta = execute(\mathcal{E}_0)$ and returns $\theta|\mathcal{V}(\mathcal{E}_0)$ as the result. Because we assume constructors and treat eager variables, the Fay-Hullot's meta-unification algorithm is modified as follows.

```
execute($\mathcal{E}_0$:set of equations) : substitution ;
$\theta :=<>$ ;
while $\mathcal{E}_0 \neq \emptyset$ delete one of the equations from $\mathcal{E}_0$
  when the equation is of the form $X = X$ or $!X = !X$
    do nothing.
  when the equation is of the form $X = t$ or $t = X$ ($X$ does not occur in $t$)
    apply lazy-variable-elimination to $X$ and $t$
  when the equation is of the form $!X = t$ or $t = !X$ ($t$ is a semi-constructor term)
    apply eager-variable-elimination to $!X$ and $t$
  when the equation is of the form $s = t$ (either $s$ or $t$ is a non-variable term)
    if root function symbols are different constructors
    then stop with failure
    else apply term-reduction to $s$ and $t$
endwhile
return $\theta$.

lazy-variable-elimination(X:lazy variable,t:term);
  let $\sigma$ be a renaming of variables in $t$ away from $W$ and $\tau$ be $< X \Leftarrow \sigma(t) >$;
  apply $\tau \circ \sigma$ to $\mathcal{E}_0$; $\theta := (\tau \circ \sigma) \circ \theta$; $W := W + I(\tau \circ \sigma)$
eager-variable-elimination(!X:eager variable,t:term);
  when $t$ is a variable $Y$ (either lazy or eager)
    let $< Y \Leftarrow !Z >$ be a renaming of the variable $Y$ away from $W$
    and $<!X \Leftarrow !Z>$ be a renaming of the variable $!X$ away from $W$;
    apply $<!X \Leftarrow !Z, Y \Leftarrow !Z>$ to $\mathcal{E}_0$; $\theta := <!X \Leftarrow !Z, Y \Leftarrow !Z> \circ \theta$; $W := W + \{!Z\}$
  when $t$ is $f(t_1, t_2, \ldots, t_m)$ ($f$ is a constructor)
    add $!X_1 = t_1, !X_2 = t_2, \ldots, !X_m = t_m$ to $\mathcal{E}_0$;
    apply $< X \Leftarrow f(!X_1, !X_2, \ldots, !X_m) >$ to $\mathcal{E}_0$;
    $\theta := <!X \Leftarrow f(!X_1, !X_2, \ldots, !X_m)> \circ \theta$; $W := W + \{!X_1, !X_2, \ldots, !X_m\}$;
    ($!X_1, !X_2, \ldots, !X_m$ are fresh eager variables)
term-reduction(s,t:term);
  when $s$ and $t$ are of the form $f(s_1, s_2, \ldots, s_m)$ and $f(t_1, t_2, \ldots, t_m)$ ($f$ is a constructor)
    add $s_1 = t_1, s_2 = t_2, \ldots s_m = t_m$ to $\mathcal{E}_0$
  when $NS(s, W) \neq \emptyset$
    select $\sigma \in NS(s, W)$ and let the corresponding rule be "$\gamma \rightarrow \delta$" ($\sigma(\gamma) \equiv \sigma(s/u)$);
    if there exists a variable $X \in W$ for which $\sigma \circ \theta(X)$ is not in $\underline{R}$-normal form
    then stop with failure
    else add $s([u \Leftarrow \delta]) = t$ to $\mathcal{E}_0$; apply $\sigma$ to $\mathcal{E}_0$; $\theta := \sigma \circ \theta$; $W := W + I(\sigma)$
  when $NS(t, W) \neq \emptyset$
    select $\sigma \in NS(t, W)$ and let the corresponding rule be "$\gamma \rightarrow \delta$" ($\sigma(\gamma) \equiv \sigma(t/v)$);
    if there exists a variable $X \in W$ for which $\sigma \circ \theta(X)$ is not in $\underline{R}$-normal form
    then stop with failure
    else add $s = t([v \Leftarrow \delta])$ to $\mathcal{E}_0$; apply $\sigma$ to $\mathcal{E}_0$; $\theta := \sigma \circ \theta$; $W := W + I(\sigma)$
  otherwise
    stop with failure
```

Figure 2.2.3. Talos Interpreter for Basic Computation

Remark. Note that in the second **when** of ***execute***, it is checked whether X occurs in t or not. We call the check *occur check*. The occur check defers some binding. For example,when the equation deleted from $\mathcal{E}_0$ is $X = [car([A|X])|Y]$, this equation is forced to be transformed to $X = [A|Y]$ in the third **when** once, becaues X occurs in $[car([A|X])|Y]$. For efficiency,we did not implement the occur check.

2.3. An Example of Basic Computation

Suppose we have given a query

?- A **when** append(A,[b,c])=[a|B].

to the Talos interpreter. Then the Talos interpreter generates a set of equations

{ !Value=A,append(A,[b,c])=[a|B] }.

which is processed as follows. The underlined expressios are processed.

{ $\underline{\text{!Value=A}}$,append(A,[b,c])=[a|B] }
⇓ eager-variable-elimination
{ $\underline{\text{append(!A,[b,c])=[a|B]}}$ }
⇓ term-reduction
{ $\underline{\text{[!E'|append(!L',[b,c])]=[a|B]}}$ }
⇓ term-reduction
{ $\underline{\text{!E'=a}}$,append(!L',[b,c])=B }
⇓ eager-variable-elimination
{ $\underline{\text{append(!L',[b,c])=B}}$ }
⇓ lazy-variable-elimination
{ }

The interpreter returns an answer

!Value=[a|L],
A=[a|L],
B=append(L,[b,c]).

Note that the assignment to B is more general than that for a query ?-*append*(A, [b, c], [a|B]) in Prolog. It has the effect to localize the backtracksearch because the answer substitution is kept more general. The "!" annotation forces the evaluation to semi-constructor terms and the computation depends on the top variables. For example,suppose we have given a query

?-[A,B] **when** append(A,[b,c])=[a|B].

to the Talos interpreter. Then the generated set of equations is processed as follows.

{ $\underline{\text{!Value=[A,B]}}$,append(A,[b,c])=[a|B] }
⇓ eager-variable-elimination
{ $\underline{\text{!V}_1\text{=A}}$,!V$_2$=[B],append(A,[b,c])=[a|B] }
⇓ eager-variable-elimination
{ $\underline{\text{!V}_2\text{=[B]}}$,append(!A,[b,c])=[a|B] }
⇓ eager-variable-elimination

```
{ [illegible]=B,!V₄=[ ],append(!A,[b,c])=[a|B] }
   ⇓ eager-variable-elimination
{ !V₄=[ ],append(!A,[b,c])=[a|!B] }
   ⇓ eager-variable-elimination
{ append(!A,[b,c])=[a|!B] }
   ⇓ term-reduction
{ [!E'|append(!L',[b,c])]=[a|!B] }
   ⇓ term-reduction
{ !E'=a,append(!L',[b,c])=!B }
   ⇓ eager-variable-elimination
{ append(!L',[b,c])=!B }
   ⇓ term-reduction
{ [b,c]=!B }
   ⇓ eager-variable-elimination
{ !B₁=b,!B₂=[c] }
   ⇓ eager-variable-elimination
{ !B₂=[c] }
   ⇓ eager-variable-elimination
{ !B₃=c,!B₄=[ ] }
   ⇓ eager-variable-elimination
{ !B₄=[ ] }
   ⇓ eager-variable-elimination
{ }
```

The interpreter returns an answer

```
!Value=[[a],[b,c]],
A=[a],
B=[b,c];
```

Remark. Because we have followed the algorithm faithfully, it takes 5 eager-variable-eliminations to delete !Value $= [A, B]$ and $[b, c] =!B$. Of course,in our actual implementation, we perform these steps in one step. In addition, we used the well-known structure sharing techmieque so that we were able to satisfy the seemingly cumbersome condition "away from W" very easily.

3. Conditional Computation in Talos

3.1. Definition of Conditional Functions

Conditional functions are defined by a set of conditional equations of the form

$$\gamma = \delta \textbf{ where } \gamma_1 = \delta_1,\ \gamma_2 = \delta_2,\ \ldots,\ \gamma_m = \delta_m.$$

When $m = 0$,the condition part (including **where**) is omitted. Corresponded to these definitions, universal formulas of the form

$$\gamma_1 = \delta_1 \wedge \gamma_2 = \delta_2 \wedge \cdots \wedge \gamma_m = \delta_m \supset \gamma = \delta$$

are added to the axioms of the equational theory $\mathcal{E}$ and universal formulas of the form

$$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$$

are added to the axioms of the term rewriting system $\mathcal{R}$.

Example 3.1.1. A function inserting an elemnt into a binary tree labelled with numbers is defined as follows.

```
function insert(X:number,T:tree):tree.
    insert(X,∅)=tree(∅,X,∅).
    insert(X,tree(L,Y,R))=tree(L,Y,R) where X=Y.
    insert(X tree(L,Y,R))=tree(insert(X,L),Y,R) where X<Y.
    insert(X,tree(L,Y,R))=tree(L,Y,insert(X,R)) where Y<X.
end.
```

The comparison of the element being inserted and the root element is done in the condition parts. We have added syntactic sugar for boolean-valued function p to denote $p(t_1, t_2, \ldots, t_n)$ in place of $p(t_1, t_2, \ldots, t_n) = true$.

Example 3.1.2. **where** has been used as syntactical sugar to avoid writing same expressions more than twice. But the conditional rewriting is a more expressive generalization, which makes implicit definitions possible. For example,we can define a function computing pairs of the head element and the last element of lists as follows.

```
function edge(L:list):list.
    edge([ ])=[ ].
    edge([X|L])=[X,Y] where reverse([X|L])=[Y|M].
end.
```

Remark. Belia et al [3] also used conditional equations, but they imposed the restriction that each equation in $\gamma_1 = \delta_1, \gamma_2 = \delta_2, \ldots, \gamma_n = \delta_n$ be of the form "constructor term = term". Note that we imposed no restriction on the form of terms in the condition part.

3.2. Meta-Unification with Constructors

3.2.1. Narrowing for Conditional Equational Theories

We define *narrowing* and *meta-unifiability* for conditional term rewriting systems mutually recursively as follows.

(a) Let s be a term, W be a set of variables containing $\mathcal{V}(s)$ and $\gamma \rightarrow \delta$ be an unconditioal rule numbered k in $\mathcal{R}$. Then a substitution σ is called a *pre-narrowing substitution* of s away from W, if a nonvariable subterm s/u and the left hand side γ of the head rule is unifiable by a most general unifier σ. We assume $\mathcal{V}(\gamma)$ is away from W by renaming away the variables in $\gamma \rightarrow \delta$ from W. s is said to be *narrowed to* $t \equiv \sigma(s[u \Leftarrow \delta])$ *with* degree 1 and denoted by $s \rightsquigarrow_{[u,k,<>\circ\sigma]} t$.

(b) Let s be a term, W be a set of variables containing $\mathcal{V}(s)$ and $\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$ be a conditional rule numbered k in $\mathcal{R}$. Then a substitution σ is called a *pre-narrowing substitution* of s away from W, if a nonvariable subterm s/u and the left hand side γ of the head rule is unifiable by a most general unifier σ. We assume $\mathcal{V}(\gamma)$ is away from W by renaming away the variables in $\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$ from W. s is said to be *narrowed to* $t \equiv \tau \circ \sigma(s[u \Leftarrow \delta])$ with degree $d+1$ and denoted by $s \rightsquigarrow_{[u,k,\tau\circ\sigma]} t$ when the instance of two terms composed of the condition part $\sigma(h_m(\gamma_1, \gamma_2, \ldots, \gamma_m))$ and $\sigma(h_m(\delta_1, \delta_2, \ldots, \delta_m))$ are metaunifiable with degree d by τ away from $W \cup \mathcal{I}(\sigma)$, where h_m is a fresh m-ary function symbol.

(c) Let s_0 and t_0 be two terms. W_0 be a set of variables containing $\mathcal{V}(s_0) \bigcup \mathcal{V}(t_0)$ and h_2 be a fresh binary function symbol. Then s_0 and t_0 is said to be *meta-unifiable with* *degree* d by $\theta' \circ (\tau_{n-1} \circ \sigma_{n-1}) \circ \cdots \circ (\tau_1 \circ \sigma_1) \circ (\tau_0 \circ \sigma_0)$ away from W_0 when there exists a sequence

$$h_2(s_0, t_0) \rightsquigarrow_{[u_0, k_0, \tau_0 \circ \sigma_0]} h_2(s_1, t_1) \rightsquigarrow_{[u_1, k_1, \tau_1 \circ \sigma_1]} \cdots \rightsquigarrow_{[u_{n-1}, k_{n-1}, \tau_{n-1} \circ \sigma_{n-1}]} h_2(s_n, t_n)$$

such that each $\rightsquigarrow_{[u_i, k_i, \tau_i \circ \sigma_i]}$ is a narrowing with degree less than d away from W_i and s_n and t_n are unifiable by a most general unifier θ', where $W_{i+1} = W_i + I(\tau_i \circ \sigma_i)$.

In particular, when $s \rightsquigarrow_{[u,k,\nu \circ \mu]} t$ and $(\nu \circ \mu)|\mathcal{V}(s)$ is $<>$, s is said to be *reduced to* t and denoted by $s \rightarrow_{[u,k,\nu \circ \mu]} t$. Again the reduction is included in the narrowing, i.e.,$\rightarrow \subseteq \rightsquigarrow$. The set of all pre-narrowing substitutions for s away from W is denoted by $NS_{pre}(s, W)$. This is computable from s and the conditional rules of $\mathcal{R}$ directly.

The reductions for a conditional term rewriting system $\mathcal{R}$ define a relation $\underline{\mathcal{R}}$ on $\mathcal{T}$ by

$$\underline{\mathcal{R}} = \{s \rightarrow t | s \rightarrow t \text{ is a reduction in } \mathcal{R}\}.$$

We assume $\mathcal{E}$ and $\mathcal{R}$ satisfies the conditions (A),(B),(C) in 2.2.1.

Remark. The introduction of conditional rewriting rules makes it difficult to guarantee the confluent property of the reduction. For example, the conditional term rewriting system corresponding to the following definition

$f(X, Y) = a$ **where** $X = Y$
$g(X) = f(X, g(X))$
$c = g(c)$

is not confluent, though the left-hand side of them are linear and non-overlapping.

3.2.2. Extended Fay-Hullot's Meta-Unification Algorithm

The following algorithm is an adaptation of the Fay-Hullt's $\mathcal{E}$-unification algorithm for conditional equational theories.

```
meta-unify($s, t$:term) : substitution;
$\theta := <>$;
**repeat**
  **when** $s$ and $t$ are unifiable by $\theta'$ away from $W$
    stop with answer $\theta' \circ \theta$
  **when** $NS_{pre}(s, W) \neq \emptyset$
    select $\sigma \in NS_{pre}(s, W)$ and let the corresponding rule be
    "$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$" $(\sigma(\gamma) \equiv \sigma(s/u))$;
    $W := W + I(\sigma)$; let $\tau$ be meta-unify$(\sigma(h_m(\gamma_1, \gamma_2, \ldots, \gamma_m)), \sigma(h_m(\delta_1, \delta_2, \ldots, \delta_m)))$;
    **if** there exists a variable $X \in W$ for which $(\tau \circ \sigma) \circ \theta(X)$ is not in $\underline{\mathcal{R}}$-normal form
    **then** stop with failure
    **else** $s := \tau \circ \sigma(s[u \Leftarrow \delta])$; $t := \tau \circ \sigma(t)$; $\theta := (\tau \circ \sigma) \circ \theta$
  **when** $NS_{pre}(t, W) \neq \emptyset$
    select $\sigma \in NS_{pre}(t, W)$ and let the corresponding rule be
    "$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$" $(\sigma(\gamma) \equiv \sigma(t/v))$;
    $W := W + I(\sigma)$; let $\tau$ be meta-unify$(\sigma(h_m(\gamma_1, \gamma_2, \ldots, \gamma_m)), \sigma(h_m(\delta_1, \delta_2, \ldots, \delta_m)))$;
    **if** there exists a variable $X \in W$ for which $(\tau \circ \sigma) \circ \theta(X)$ is not in $\underline{\mathcal{R}}$-normal form
    **then** stop with failure
    **else** $s := \tau \circ \sigma(s)$; $t := \tau \circ \sigma(t[v \Leftarrow \delta])$; $\theta := (\tau \circ \sigma) \circ \theta$
**endrepeat**
```

Figure 3.2.3. Extended Fay-Hullot's Meta-Unification Algorithm

Example 3.2. Let s be $insert(A, insert(B, tree(\emptyset, 1, S)))$ and t be $tree(tree(\emptyset, C, \emptyset), 1, T)$. Because s and t are not unifiable, the Fay-Hullot's algorithm selects the second **when** since $NS_{pre}(t, W) = \emptyset$. Then s can be narrowed to

$s_1 \equiv insert(A, tree(insert(0, \emptyset), 1, S_1))$

by $< B \Leftarrow 0, S \Leftarrow S_1 >$. After adequate two succeeding narrowings, we have

$s_3 \equiv tree(tree(\emptyset, 0, \emptyset), 1, insert(suc(suc(A_3)), S_3))$.

Then in the next repetition, s_3 is unifiable with $t \equiv tree(tree(\emptyset, C, \emptyset), 1, T)$ by $\sigma = < A_3 \Leftarrow A_4, S_3 \Leftarrow S_4, C \Leftarrow 0, T \Leftarrow insert(suc(suc(A_4)), S_4) >$. Hence

$< A \Leftarrow suc(suc(A_4)), B \Leftarrow 0, C \Leftarrow 0, S \Leftarrow S_4, T \Leftarrow insert(suc(suc(A_4)), S_4) >$

is a meta-unifier of s and t. There are another four meta-unifiers

$< A \Leftarrow 0, B \Leftarrow suc(suc(B_4)), C \Leftarrow 0, S \Leftarrow S_4, T \Leftarrow insert(suc(suc(B_4)), S_4) >$,

$< A \Leftarrow 0, B \Leftarrow 0, C \Leftarrow 0, S \Leftarrow S_4, T \Leftarrow S_4 >$,

$< A \Leftarrow 1, B \Leftarrow 0, C \Leftarrow 0, S \Leftarrow S_4, T \Leftarrow S_4 >$,

$< A \Leftarrow 0, B \Leftarrow 1, C \Leftarrow 0, S \Leftarrow S_4, T \Leftarrow S_4 >$.

3.2.3. Interpreter for Conditional Computation

The Talos interpreter behaves similarly to one in 2.2.3 except that the narrowings are done in a slightly complicated manner.

```
execute($\mathcal{E}_0$:set of equations) : substitution ;
$\theta$ := $<>$ ;
while $\mathcal{E}_0 \neq \emptyset$ delete one of the equations from $\mathcal{E}_0$
  when the equation is of the form $X = X$ or $!X = !X$
    do nothing.
  when the equation is of the form $X = t$ or $t = X$ ($X$ does not occur in $t$)
    apply lazy-variable-elimination to $X$ and $t$
  when the equation is of the form $!X = t$ or $t = !X$ ($t$ is a semi-constructor term)
    apply eager-variable-elimination to $!X$ and $t$
  when the equation is of the form $s = t$ (either $s$ or $t$ is a non-variable term)
    if root function symbols are different constructors
    then stop with failure
    else apply term-reduction to $s$ and $t$
endwhile
return $\theta$.

term-reduction(s,t:term);
  when $s$ and $t$ are of the form $f(s_1, s_2, \ldots, s_m)$ and $f(t_1, t_2, \ldots, t_m)$ ($f$ is a constructor)
    add $s_1 = t_1, s_2 = t_2, \ldots s_m = t_m$ to $\mathcal{E}_0$
  when $NS_{pre}(s, W) \neq \emptyset$
    select $\sigma \in NS_{pre}(s, W)$ and let the corresponding rule be
    "$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \to \delta$" ($\sigma(\gamma) \equiv \sigma(s/u)$);
    $W := W + !(\sigma)$; let $\tau$ be $execute(\sigma(\{\gamma_1 = \delta_1, \gamma_2 = \delta_2, \ldots, \gamma_m = \delta_m\}))$;
    if there exists a variable $X \in W$ for which $(\tau \circ \sigma) \circ \theta(X)$ is not in $\underline{R}$-normal form
    then stop with failure
    else add $s[u \Leftarrow \delta] = t$ to $\mathcal{E}_0$; apply $\tau \circ \sigma$ to $\mathcal{E}_0$; $\theta := (\tau \circ \sigma) \circ \theta$
```

```
when $NS_{pre}(t,W) \neq \emptyset$
   select $\sigma \in NS_{pre}(t,W)$ and let the corresponding rule be
   "$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \to \delta$" $(\sigma(\gamma) \equiv \sigma(t/v))$;
   $W := W + I(\sigma)$; let $\tau$ be $execute(\sigma(\{\gamma_1 = \delta_1, \gamma_2 = \delta_2, \ldots, \gamma_m = \delta_m\}))$;
   if there exists a variable $X \in W$ for which $(\tau \circ \sigma) \circ \theta(X)$ is not in $\underline{R}$-normal form
   then stop with failure
   else add $s = t[v \Leftarrow \delta]$ to $\mathcal{E}_0$; apply $\tau \circ \sigma$ to $\mathcal{E}_0$; $\theta := (\tau \circ \sigma) \circ \theta$
otherwise
   stop with failure
```

Figure 3.2.3. Talos Interpreter for Conditional Computation

Remark. As is proved in [18], for any given ground query **?-** t, Talos answers a ground constructor term s satisfying $s =_{\mathcal{E}} t$. We call this property *ground completeness.*

3.3. An Example of Conditional Computation

Suppose we have given a query

?- C **when** insert(A,insert(B,tree(∅,1,S))) =tree(tree(∅,C,∅),1,T).

to the Talos interpreter. Then the Talos interpreter generates a set of equations

{ !Value=C, insert(A,insert(B,tree(∅,1,S)))= tree(tree(∅,C,∅),1,T) }

which is processed as follows. The underlined expressios are processed.

{ $\underline{\text{!Value=C}}$, insert(A,insert(B,tree(∅,1,S)))= tree(tree(∅,C,∅),1,T) }

$\Downarrow$ eager-variable-elimination

{ insert(A,$\underline{\text{insert(B,tree(∅,1,S))}}$)= tree(tree(∅,!C,∅),1,T) }

$\Downarrow$ term-reduction

{ $\underline{\text{insert(A,tree(insert(0,∅),1,S))}}$= tree(tree(∅,!C,∅),1,T) }

$\Downarrow$ term-reduction

{ $\underline{\text{tree(insert(0,∅),1,insert(suc(suc(A}_1\text{)),S))= tree(tree(∅,!C,∅),1,T)}}$ }

$\Downarrow$ term-reduction

{ $\underline{\text{insert(0,∅)}}$=tree(∅,!C,∅), 1=1, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction

{ $\underline{\text{tree(∅,0,∅)=tree(∅,!C,∅)}}$, 1=1, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction

{ $\underline{\text{∅=∅}}$, 0=!C, ∅=∅, 1=1, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction

{ $\underline{\text{0=!C}}$, ∅=∅, 1=1, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ eager-variable-elimination

{ $\underline{\text{∅=∅}}$, 1=1, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction

{ $\underline{\text{1=1}}$, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction

{ $\underline{\text{0=0}}$, insert(suc(suc(A$_1$)),S)=T }

$\Downarrow$ term-reduction
{ $\underline{\text{insert(suc(suc(A}_1\text{)),S)=T}}$ }
$\Downarrow$ lazy-variable-elimination
{ }

Note that the third equation of the definition of *insert* is used in the first term-reduction, while the fourth equation is used in the second term-reduction. One of the answers is

!Value=0,
A=A_1+2,
B=0,
C=0,
S=S_1,
T=insert(A_1+2,S_1);

Again,the computation depends on the top variables. For example,suppose we have given a query

?- [C,T] **when** insert(A,insert(B,tree(∅,1,S))) =tree(tree(∅,C,∅),1,T).

to the Talos interpreter. Then the corresponding answer is

!Value=0,
A=2,
B=0,
C=0,
S=S_1,
T=tree(∅,2,∅);

Note that we need nondeterministic search of desirable substitutions, though the definition of data types and functions are deterministic. How to search the solution is described in section 4 and 5.

Remark. Again,because we have followed the algorithm faithfully, it takes many steps to delete $\mathit{tree}(\mathit{insert}(0,\emptyset),1,\mathit{insert}(\mathit{suc}(\mathit{suc}(A_1)),S)) = \mathit{tree}(\mathit{tree}(\emptyset,!C,\emptyset),1,T)$. In our actual implementation, we generate $\mathit{insert}(0,\emptyset) = \mathit{tree}(\emptyset,!C,\emptyset)$ in one step from it.

4 Nondeterministic Computation in Talos

4.1. Definition of Nondeterministic Functions

As is seen, the Talos interpreter search the desirable solutions for a given query. We can define functions with this mechanism by permitting nondeterministic functions. A nondeterministic function is an $n+1$-ary relation whose fixed n arguments are always inputs and the remaining one argument is always output. Hence it is not a function in its strict sense and causes confusion in mathematical semantics if it were treated as a function. But we can compute such a nondeterministic function staying within our functional framework. The Talos interpreter behaves in the completely same way as previous.

Example 4.1.1. A nondeterministic function computing some member of lists (the *member* relation with mode declaration $(-,+)$ in DEC-10 Prolog) is defined as follows.
nondeterministic function one-of(L:list):element.

```
        one-of([X|L])=X.
        one-of([Y|L])=one-of(L).
    end.
```

Of course, = does not necessarily mean the true equlity when one hand side of an equation contains a nondeterministic function symbol, though we pretend it for simplicity.

Example 4.1.2. A nondeterministic function computing some prefix of lists is defined by

```
    nondeterministic function initial(L:list):list.
        initial(L)=M where append(M,N)=L.
    end.
```

Remark. One might say permitting such notations of nondeterministic function is confusing. But we claim that such notations show the flow of information explicitly and is superior to purely relational languages in readability. Of course, we do not claim that relations are unnecessary, though we guess the relatios we need are nodeterministic fuctions in many cases.

4.2. Backtracking Search

The nondeterministic Talos interpreter is transformed to a sequential one with the depth first search and the backtracking by the following modifications.

(a) We make the body of the *execute* (except initialization of θ to $<>$) a self recursive routine (*rewrite-equations*).

(b) We let the *term-reduction* enumerate alternative narrowings.

(c) We insert backtracking control at appropriate places.

4.2.1. Selection of Equations

We need to choose an adequate equation in the *rewrite-equations*. Following the control of Prolog, we choose the leftmost equations. We keep a set of simultaneous equations in a stack. The equation at the top is poped at each call and processed. It is either simply erased (possibly with some application of substitution), or it generates several new equations and the generated equations are pushed at the top.

Remark. In general,a proof procedure, which is nondeterministic in selecting the processing units and enjoys some property φ, is said to be *strongly* φ with respect to the selection when the proof procedure with an arbitraly selction rule still enjoys φ. For example, the SLD-resolution is said to be *strongly complete with respect to* the selection of atoms from conjunction of atoms. Though we have not yet proved formally, we expect with certainty that *execute* is *strongly ground complete with respect to* the selction of an equation from conjunction of equations. Then this selection rule for equations "from left to right" does not loose the ground completeness.

4.2.2. Enumeration of Alternative Narrowings

We also need to choose an adequate narrowing substitution θ in term-reduction. Depending on the choice of the occurrence at which a narrowing is applied, we can integrate various evaluation strategies into Talos such as call-by-value,call-by-name,call-by-need,lazy evaluation and eager evaluation (see [14],[22]). But we defer the discussions on the selection rule for narrowing substitution to the next section. Before choosing it anyway, we set the backtracking point there. Once some occurrence is chosen, the rewriting rules in the definition are tried from top to bottom following the control of Prolog.

Remark. This selection rule for rules "from top to bottom" obviously looses the ground completeness, as does in Prolog.

4.2.3. Interpreter for Nondeterministic Computation

The Talos interpreter behaves similarly to one in 3.2.3 except that now we search desirable solutions sequentially as follows.

```
execute($\mathcal{E}_0$ : stack of equations) : substitution ;
  $\theta := <>$; return rewrite-equations($\mathcal{E}_0$)

rewrite-equations ($\mathcal{E}_0$ : stack of equations) :
if $\mathcal{E}_0 = \emptyset$
then return $<>$
else pop $\mathcal{E}_0$;
        when the equation is of the form $X = X$ or $!X = !X$
          apply rewrite-equations to $\mathcal{E}_0$ recursively;
          if it is in failure
          then undo the pop and fail
        when the equation of the form $X = t$ or $t = X$ ($X$ does not occur in $t$)
          apply lazy-variable-elimination to $X$ and $t$;
          apply rewrite-equations to $\mathcal{E}_0$ recursively;
          if it is in failure
          then undo the lazy-variable-elimination and the pop and fail;
        when the equation is of the form $!X = t$ or $t = !X$ ($t$ is a semi-constructor term)
          apply eager-variable-elimination to $!X$ and $t$;
          apply rewrite-equations to $\mathcal{E}_0$ recursively;
          if it is in failure
          then undo the eager-variable-elimination and the pop and fail;
        when the equation is of the form $s = t$ (either $s$ or $t$ is a non-variable term)
          while alternative choices of term-reduction are not exausted do
                apply term-reduction to $s$ and $t$;
                apply rewrite-equations to $\mathcal{E}_0$ recursiveiy;
                if it is in failure
                then undo the term-reduction
          endwhile
          undo the pop and fail
endif
```

Figure 4.2. Talos Interpreter for Nondeterministic Computation

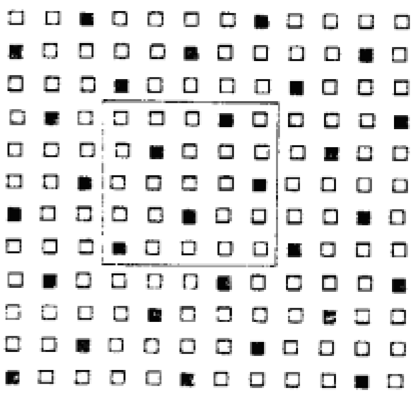
4.3. An Example of Nondeterministic Computation

Now we present an example of nondeterministic functions. The problem we consider is taken from textile manufacturing (cf. [17]). A satin weaving pattern is one of the three basic weaving patterns. Textile is an interlacing of warp (thread in the horizontal direction) and woof (thread in the vertical direction). There can be two states of interlacing,i.e., (1) a warp is on a woof (called OP denoted by balck square) and (2) a woof is on a warf (denoted by white square). A satin weave consists of the minimum unit of repetition (called "repeat") satisfying the following conditions.

(a) "repeat" is a square pattern, i.e., it consists of k warps and k wooves,
(b) there exists exactly one OP on any warp or woof in "repeat",
(c) there is no pair of OPs diagonally connected in textile.

Of course we don't distinguish two patterns superposable by translation and 90°, 180° and 270° rotations. The problem is to "generate a repeat of satin weaving pattern consisting of 8 warps and 8 wooves."

First of all, note that a "repeat" is expressible by a permutation of the numbers from 0 to k-1 by numbering threads (the uppermost leftmost corner is (0,0) point), e.g. [3,1,4,2,0] represents a repeat in the figure below. Note also that we can assume the uppermost leftmost corner is an OP by translation of "repeat" and it removes the ambiguity by geometrical congruence.



So we can take an approach similar to one for the "Eight Queen" problem. We need to take two factors into considerations : (1) the warp on which we are searching an OP (Warp) and (2) the set of the candidates for OPs,i.e., wooves on which the warp can be (Candidate). The crucial part is easily programmed as follows. (We use some syntactical sugar for readability.)

```
nondeterministic function weave(Repeat:list;Warp:number,Candidate:list):list.
    weave(Repeat,Warp,Candidate)=weave(Next-Repeat,Next-Warp,Next-Candidate)
            where 0 < Warp, Now=last(Repeat), Next=one-of(Candidate),
                    Next-Repeat=append(Repeat,[Next]),
                    Next-Warp=Warp+1 mod 8,
                    Next-Candidate=subtract(inhibit(release(Candidate,Now),Next),Next-Repeat).
    weave(Repeat,0,[ ])=Repeat
            where Jump=|first(Repeat)-last(Repeat)|, 1 < Jump, Jump < 7.
end.
```

Definitions of *release* and *inhibit* are simply

```
    release(C,I)=add(C,[I1,I2]) where I1=I-1 mod 8,I2=I+1 mod 8
    inhibit(C,J)=subtract(C,[J1,J2]) where J1=J-1 mod 8,J2=J+1 mod 8
```

where *add* and *subtract* are for union and difference of two sets. The answer is computed by a query ?-*weave*([0], 1, [2, 3, 4, 5, 6]).

Remark: The program above can be still optimized considering efficiency (e.g. representation of *Repeat* in the reverse order). But it makes quick construction of the understandable programs easy before further optimizations.

5. "Call by Need" Computation in Talos

5.1. Definition of "Call by Need" Functions

Note that in the definition of *append* in 2.1, the form of the first argument decides which rule applies to the term and the second argument has no contribution at this point. Hence if *append*(s, t) is given, the rewriting of the first argument s is innevitable, while that of the second argument t can be delayed until it is actually necessary. We declare the need for each argument of functions by $+$ and $-$.

Example 5.1.1. The *append* function delaying the evaluation of the second argument is defined as follows.

```
    function append(L,-M:list):list.
        append([ ],L)=L.
        append([X|L],M)=[X|append(L,M)].
    end.
```

As is known from the first argument L, we stipulate that the default is $+$ for functions. Hence all functions defined so far are evaluated in the "call by value" strategy.

Needs are also declared for data types.

Example 5.1.2. The data type *list* is defined as follows.

```
    data list = new.
        constructor.
            nil.
            cons(X:element,L:list).
        operator.
            length(L:list):number.
                length([ ])=0.
                length([X|L])=length(L)+1.
    end.
```

We stipulate that the default is also $+$ for operators, while it is — for constructors. Hence all operators defined so far are evaluated in the "call by value" strategy, while all constructors defined so far are evaluated lazily.

5.2. Control of Evaluation by Needs and Annotations

5.2.1. Index for "Call by Need" Computation

Now we introduce a selection rule for the narrowing substitution in $NS_{pre}(s, W)$ or $NS_{pre}(t, W)$ in the previous algorithm. An occurrence of subterm of s is called an *index* when the subterm must be narrowed in order that a semi-constructor term is derived from s. A selection rule which always narrow the outermost index (i.e., at the least occurrence) at each narrowing is called *call by need* strategy (cf. [14]).

We assume every semi-constructor term has no index and every non semi-constructor term with root symbol f has indices for any non-constructor symbol f. Hence any subterm s' of s is narrowed in the "call by need" derivation from s in only two cases. One is when the root symbol f of $s \equiv f(s_1, s_2, \ldots, s_n)$ is a non-constructor symbol and

(a) s' is s itself or
(b) there exists a proper subterm s/u at s's outermost index u and s' is narrowed in the "call by need" derivation from s/u.

Another is when the root symbol f of $s \equiv f(s_1, s_2, \ldots, s_m)$ is a constructor and s' is narrowed in the "call by need" derivation from some s_i after f is peeled off in term-reduction applied to $s \equiv f(s_1, s_2, \ldots, s_m)$ and $t \equiv f(t_1, t_2, \ldots, t_m)$.

The following is a well-known example to demonstrate the power of lazy evaluation ([8],[10]).

Example 5.2.1. Let *integers* and *second* be functions defined by

integers(N)=[N|integers(N+1)].
second(L)=car(cdr(L)).

respectively. The usual "call by value" rewriting in Lisp does not terminate for *second*(*integers*(4)).

$$\begin{array}{l}
\text{second}(\underline{\text{integers}(4)}) \\
\quad \Downarrow \text{term-reduction} \\
\text{second}([4|\text{integers}(\underline{4+1})]) \\
\quad \Downarrow \text{term-reduction} \\
\text{second}([4|\underline{\text{integers}(5)}]) \\
\quad \Downarrow \text{term-reduction} \\
\text{second}([4,5|\text{integers}(\underline{5+1})]) \\
\quad \Downarrow \text{term-reduction} \\
\quad \vdots
\end{array}$$

while "lazy evaluation" terminates properly.

$$\begin{array}{l}
\text{second}(\underline{\text{integers}(4)}) \\
\quad \Downarrow \text{term-reduction} \\
\underline{\text{second}([4|\text{integers}(4+1)])} \\
\quad \Downarrow \text{term-reduction} \\
\text{car}(\underline{\text{cdr}([4|\text{integers}(4+1)])}) \\
\quad \Downarrow \text{term-reduction} \\
\text{car}(\text{integers}(\underline{4+1})) \\
\quad \Downarrow \text{term-reduction} \\
\text{car}(\underline{\text{integers}(5)}) \\
\quad \Downarrow \text{term-reduction} \\
\underline{\text{car}([5|\text{integers}(5+1)])} \\
\quad \Downarrow \text{term-reduction} \\
5
\end{array}$$

A conditional equation in the definition of operators and functions is said to *observe need declaration* when the left hand side of the head satisfies the following conditions.

(a) It is not a semi-constructor term.
(b) It is a linear term,that is, there appears no variable at more than two occurrences.
(c) Each argument with need $+$ must be a term of the form $f(X_1, X_2, \ldots, X_n)$ where f is a constructor and $X_1, X_2, \ldots, X_n$ are distinct variables.
(d) Each argument with need $-$ must be a variable.

When some conditional equation does not satisfy the conditions above, it is first preprocessed and converted to one satisfying the conditions.

Example 5.2.2. Let *fibonacci* be the well-known function computing the fibonacci sequence as follows.

```
    function fibonacci(N:number):number.
        fibonacci(0)=1.
        fibonacci(1)=1.
        fibonacci(N+2)=fibonacci(N)+fibonacci(N+1).
    end.
```

The needs of the argument of *fibonacci* are all + by the default. But it does not satisfy the condition mentioned above. The definition must be once converted to

```
    function fibonacci(N:number):number.
        fibonacci(0)=1.
        fibonacci(M+1)=1 where M=0.
        fibonacci(N+1)=fibonacci(M)+fibonacci(N) where M=N+1.
    end.
```

Slightly strngthening the condition (B) in 2.2, we assume $\mathcal{E}$ and $\mathcal{R}$ satisfy the following conditions.

(A) $\underline{\mathcal{R}}$ is confluent and terminating.
(B) Each definition observes need declaration.
(C) For any $s \in \mathcal{G}$, there exists $t \in \mathcal{G}_C$ such that $s =_{\mathcal{E}} t$.

Remark: The necessary conditions enabling "call by need" strategy have been not yet fully investigated. In [14] $\mathcal{R}$ is assumed to be a linear and nonoverlapping term rewriting system.

5.2.2. General Eager Annotation

Uniform "lazy evaluation" is sometimes inconvinient. For example the rewriting of *append*([1], [2, 3]) stops at [1|*append*([ ], [2, 3])] and the final result [1,2,3] won't be computed. For more minute control, we permit the generalized use of the "!" annotation. So far, the use of the "!" annotation was allowed only for the Talos interpreter. An eager variable appearing in the processing was either the top level variable !*Value* or a variable to which "!" is inhrited from !*Value*. We permit programmers to annotate any variables by "!" freely. In addition, we also allow not to annotate the the top level variable *Value* by "!". When a given query is conditioned by **where** in place of **when** as follows

?- t **where** s_1=t_1,s_2=t_2,...,s_m=t_m.

the Talos interpreter generates

{ Value = t,s_1=t_1,s_2=t_2,...,s_m=t_m }

Example 5.2.3. When Talos is given a query

?- X **when** append(append([ ],[2,1]),cdr([4,3,5]))=[X|!L]

meta-unification proceeds as follows.

{ $\underline{\text{!Value=X}}$,append(append([ ],[2,1]),cdr([4,3,5]))=[X|!L] }
⇓ eager-variable-elimination
{ append($\underline{\text{append([ ],[2,1])}}$,cdr([4,3,5]))=[!X|!L] }
⇓ term-reduction
{ $\underline{\text{append([2,1],cdr([4,3,5]))}}$=[!X|!L] }
⇓ term-reduction

{ [2|append([1],cdr([4,3,5]))]=[!X|!L] }
⇓ term-reduction
{ 2=!X,append([1],cdr([4,3,5]))=!L }
⇓ eager-variable-elimination
{ append([1],cdr([4,3,5]))=!L }

If $!L$ were not an eager variable, *append*([1], *cdr*([4, 3, 5])) would be bound to L and the computation would stop. But here meta-unification continues as follows.

⇓ term-reduction
{ [1|append([ ],cdr([4,3,5]))]=!L }
⇓ eager-variable-elimination
{ 1=!A$_1$,append([ ],cdr([4,3,5]))=!L$_1$ }
⇓ several eager-variable-elimnations
{ append([ ],cdr([4,3,5]))=!L$_1$ }
⇓ several term-reductiona and eager-variable-eliminations
{ }

and $!L$ is $[1, 3, 5]$.

Example 5.2.4. The Talos interpreter answers
Value=[[a|L],append(L,[b,c])],
A=[a|L],
B=append(L,[b,c]);
to ?- $[A, B]$ **where** *append*$(A, [b, c]) = [a|B]$, because it meta-unifies
{ Value=[A,B],append(A,[b,c])=[a|B] }.

Remark : The eager annotation of the top level variable *Value* never makes terminating computations non-terminating. Other eager variables may do it.

5.2.3. Iterpreter for "Call by Need" Computation

For a given query of the form
?-t **when** s$_1$=t$_1$,s$_2$=t$_2$,...,s$_m$=t$_m$.
the Talos interpreter generates a set of equations
$\mathcal{E}_0 = \{!Value = t, s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m\}$
and for a given query of the form
?-t **where** s$_1$=t$_1$,s$_2$=t$_2$,...,s$_m$=t$_m$.
the Talos interpreter generates a set of equations
$\mathcal{E}_0 = \{Value = t, s_1 = t_1, s_2 = t_2, \ldots, s_m = t_m\}$.
Then the Talos interpreter behaves similarly to one in 4.2.3 except both sides of the selected equation are continuously narrowed until the both become semi-constructor terms and the backtracking points are set in it as follows.

```
term-reduction(s,t:term);
  when $s$ and $t$ are of the form $f(s_1, s_2, \ldots, s_m)$ and $f(t_1, t_2, \ldots, t_m)$ ($f$ is a constructor)
    add $s_1 = t_1, s_2 = t_2, \ldots s_m = t_m$ to $\mathcal{E}_0$
  when either $s$ or $t$ is not a semi-constructor term
    add $narrow\text{-}up(s, +) = narrow\text{-}up(t, +)$ to $\mathcal{E}_0$
```

```
narrow-up($s$,$need$)
  if $s$ is a semi-constructor term or $need$ is —
  then return $s$
  else let $s$ be $f(s_1, s_2, \ldots, s_n)$;
      for $i$ from 1 to $n$ do $t_i$ := $narrow\text{-}up(s_i$, $i$-th need of $f)$;
      while rules whose heads are unifiable with $\theta(f(t_1, t_2, \ldots, t_n))$ are not exhausted do
        let the mgu away from $W$ be $\sigma$ and the corresponding rule be
        "$\gamma_1 \downarrow \delta_1 \wedge \gamma_2 \downarrow \delta_2 \wedge \cdots \wedge \gamma_m \downarrow \delta_m \supset \gamma \rightarrow \delta$";
        $W := W + I(\sigma)$; let $\tau$ be execute $(\sigma(\{\gamma_1 = \delta_1, \gamma_2 = \delta_2, \ldots, \gamma_m = \delta_m\}))$;
        if there exists a variable $X \in W$
          for which $(\tau \circ \sigma) \circ \theta(X)$ is not in $\underline{R}$-normal form
        then fail
        else apply $\tau \circ \sigma$ to $\mathcal{E}_0$; $\theta := (\tau \circ \sigma) \circ \theta$; return $narrow\text{-}up(\tau \circ \sigma(\delta), need)$
      endwhile
      fail
```

Figure 5.2. Talos Interpreter for "Call by Need" Computation

Remark. Our strategy may cause more instanciation than is expected. For example, when the selected equation is $cdr([X|L]) = append(M, N)$, the interpreter generates a binding not $< L \Leftarrow append(M, N) >$ but $< M \Leftarrow [\,], L = N >$ or $< M \Leftarrow [X'|M'], L \Leftarrow [X'|append(M', N)] >$.

5.3. An Example of "Call by Need" Computation

Now we show "Call by Need" Prolog. It can be considered a Prolog augmented by functional notation and "call by need" strategy.

A basic data type *boole* is defined as follows.

```
data boole = new.
   constructor.
      true.
      false.
   operator.
      and(X,-Y:boole):boole.
         true∧Y = Y.
         false∧Y = false.
end.
```

Suppose we have specified *append* with the heading

```
function append(L,-M:list):list.
```

and *reverse* is defined as a binary relation as follows.

```
nondeterministic function reverse(L,-M:list):boole.
   reverse([ ],M) where M=[ ].
```

```
    reverse([X|L],M) where reverse(L,N),append(N,[X])=M.
  end.
```

Then,in order to compute the last element of a list, we may give a query

```
    ?- X when reverse([3,1,2],[X|M]).
```

The Talos interpreter trys to meta-unify the condition part recursively and it proceeds as follows. (The indented sets of equations are generated from the condition parts in term-reductions.)

```
  { !Value=X,reverse([3,1,2],[X|M])=true }
     ⇓ eager-variable-elimination
  { reverse([3,1,2],[!X|M])=true }
     ⇓
     { reverse([1,2],N_1)=true,append(N_1,[3])=[!X|M] }
        ⇓
        { reverse([2],N_2)=true,append(N_2,[1])=N_1 }
           ⇓
           { reverse([ ],N_3)=true,append(N_3,[2])=N_2 }
              ⇓
              { N_3=[ ] }
                 ⇓ lazy-variable-elimination
              { }
              ⇓
           { true=true,append([ ],[2])=N_2 }
              ⇓ term-reduction
           { append([ ],[2])=N_2 }
              ⇓ lazy-variable-elimination
           { }
           ⇓
        { true=true,append(append([ ],[2]),[1])=N_1 }
           ⇓ term-reduction
        { append(append([ ],[2])[1])=N_1 }
           ⇓ lazy-variable-elimination
        { }
        ⇓
     { true=true,append(append(append([ ],[2]),[1]),[3])=[!X|M] }
        ⇓ term-reduction
     { append(append(append([ ],[2]),[1]),[3])=[!X|M] }
        ⇓ term-reduction
     { append(append([2],[1]),[3])=[!X|M] }
        ⇓ term-reduction
     { append([2|append([ ],[1])],[3])=[!X|M] }
        ⇓ term-reduction
     { [2|append(append([ ],[1]),[3])]=[!X|M] }
        ⇓ term-reduction
     { 2=!X,append(append([ ],[1]),[3])=M }
        ⇓ 3 eager-variable-eliminations
     { append(append([ ],[1]),[3])=M }
```

```
        ⇓ lazy-variable-elimination
      { }
      ⇓
  { true=true }
      ⇓ term-reduction
  { }
```

Hence the Talos interpreter answers

```
  !Value=2,
  X=2,
  M=append(append([ ],[1]),[3]);
```

Note that it has not constructed the full reversed list and the computation has taken linear time as to the length of the first argument of *reverse*.

Remark. A reduction method,called *graph reduction* ([14],[22]), can avoid rewriting subterms of the same form more than twice by sharing same arguments in directed acyclic graphes (DAG). In the current version,we did not implement such sharing.

6. Computation with Streams in Talos

6.1. Definition of Functions with Stream

Lazy evaluation makes it possible to use infinite data structures explicitly. An infinite data structure in Talos is a *stream*. A variable denoting a stream, called a *stream variable* and denoted by X^*, is always a lazy variable. Stream variables can be used to show an infinite sequence of values denoting the history of a usual variable, which,with conditional rewriting rules,makes Lucid-like programming possible (cf. "DO" in MACLISP).

Example 6.1. A program computing $\sum_{i=1}^{N} i^2$ using stream (Ashcroft and Wadge [2]) is represented in Talos as follows.

```
  function square-sum(N:number):number.
     square-sum(N)=S* asa I* > N where I*=[1|I*+1],S*=[0|S*+square(I*)]
  end
```

where we have used the list notation $[X|L]$ for X **fby** L (followed by) in Lucid and **asa** is an infix binary function intended for "as soon as". Functions are applied to streams elementwise and result in streams when they are not ones specific for stream processing. For example,$I^* + 1$ denotes a stream obtained by applying $\lambda X.X + 1$ to each element of I^*.

6.2. Stream Processing

6.2.1. Stream Processing Primitives

When the Talos program is given a query including streams or stream processing functions, it processes them by using the following definitions for the data type stream (see [11]). **if** B **then** L **else** R is a mix-fix notation of *if-then-else*(B, L, R).

```
  nondeterministic function asa(-X,-Y:stream):element.
     asa(X*,Y*) = if first(Y*) then first(X*) else asa(next(X*),next(Y*)))
  end
```

```
function if-then-else(B:boole,-L,-R:element):element.
   if true then L else R = L.
   if false then L else R = R.
end.
```

Other primitives are defined by the following equations.

```
first([E|S*]) = E.
next([E|S*]) = S*.
```

Using *first* and *nest*, elementwise application of a function *h* is done as follows.

```
h(I*)=[h(first(I*))|h(next(I*))].
```

Remark. Talos has several built-in primitives. Especially,arithmetic computation is done in a model, i.e.,the machine representation of the natural numbers. For example,it is stupid to peel off 10000 *suc*'s for $X + 10002 = 10000$ and obtain $X = 2$. But the patterns of combinations of both sides in arithmetic equations are so various that the amount of code in our implementation for corresponding computation in the model is more than expected.

6.2.2. Stream Processing Process

Because stream variables are data structures specified cyclically, it is difficult to depict the process of stream processing exactly. In the followings,we give intuitive explanations.

Example 6.2.2 The computation of *square-sum*(5) proceeds as follows.

```
{ !Value=square-sum(5) }
   ⇓
   { I*=[1|I*+1],S*=[0|S*+square(I*)] }
     ⇓ stream-variable-elimination
   { S*=[0|S*+square(I*)] }
     ⇓ stream-variable-elimination
   { }
   ⇓
{ !Value=asa([0|S*+squre(I*)],[1|I*+1] >5) }
   ⇓ term-reduction
{ !Value=if first([1|I*+1]>5)
           then first([0|S*+square(I*)])
           else asa(next([0|S*+square(I*)]),next([1|I*+1]>5)) }
   ⇓ term-reduction
{ !Value=if first([1>5|I*+1>5])
           then first([0|S*+square(I*)])
           else asa(next([0|S*+square(I*)]),next([1|I*+1]>5)) }
   ⇓ term-reduction
{ !Value=if 1>5
           then first([0|S*+square(I*)])
           else asa(next([0|S*+square(I*)]),next([1|I*+1]>5)) }
   ⇓ term-reduction
{ !Value=if false
           then first([0|S*+square(I*)])
```

```
            else asa(next([0|S* +square(I*)]),next([1|I*+1]>5)) }
     ⇓ term-reduction
  { !Value=asa(next([0|S* +square(I*)]),next([1|I*+1]>5)) }
     ⇓ term-reduction
  { !Value=if first(next([1|I*+1]>5))
            then asa(next([0|S* +square(I*)]))
            else asa(next(next([1|I*+1]>5)),next(next([0|S* +square(I*)]))) }
     ⇓
     ⋮
```

where the notation $[1|I^* + 1]$ is inexact. It should be $[1|[1|[1\cdots] + 1] + 1]$.

Remark. In the computation above, the Talos interpreter has to process

if first(next^i([1|I*+1])) **then** ... **else** ....

In general,when $first(next^i([I_0|h(I^*)]))$ is computed, it must be once converted to $first(h^i(I^*))$ and then $h^i(I_0)$ is computed. Hence without any device, we have to repeat the computation of the $i+1$-th element of I^* from the first element all the way everytime, though we have computed the i-th element just now. In this case,the function $h = \lambda X.X+1$ is simple, but in general,it may be very time-consuming. We need a device to memorize the elements of streams, once they are computed.

6.2.3. Interpreter for Computation with Streams

The Talos interpreter behaves similarly to one in 5.2.3 except the use of special functions for stream and the "occur check" depending on the distinction of variables. For simplicity, we show the nondeterministic version of the Talos interpreter revised for stream processing.

```
execute($\mathcal{E}_0$ : set of equations) : substitution ;
$\theta$ := < > ;
while $\mathcal{E}_0 \neq \emptyset$ delete one of the equations from $\mathcal{E}_0$
  when the equation is of the form $X^* = X^*$, $X = X$ or $!X = !X$
    do nothing
  when the equation is of the form $X^* = t$ or $t = X^*$
    apply stream-variable-elimination to $X^*$ and $t$
  when the equation is of the form $X = t$ or $t = X$ ($X$ does not occur in $t$)
    apply lazy-variable-elimination to $X$ and $t$
  when the equation is of the form $!X = t$ or $t = !X$ ($t$ is a semi-constructor term)
    apply eager-variable-elimination to $!X$ and $t$
  when the equation is of the form $s = t$ (either $s$ or $t$ is a non-variable term)
    apply term-reduction to $s$ and $t$
endwhile
return $\theta$

stream-variable-elimination($X^*$:stream variable,t:term);
  let $\sigma$ be a renaming of variables in $t$ away from $W$ and $\tau$ be $< X^* \Leftarrow \sigma(t) >$;
  apply $\tau \circ \sigma$ to $\mathcal{E}_0$; $\theta := (\tau \circ \sigma) \circ \theta$; $W := W + I(\tau \circ \sigma)$
```

Figure 6.2. Talos Interpreter for Computation with Streams

6.3. An Example of Computation with Stream

Now we present an example of computation with stream. The example is taken from the elementary theory of numbers. Let $\Sigma(n)$ denote the sum of all divisors of n. Then $\Sigma(n) - n$,i.e., the sum of all divisors of a number except itself is as follows.

$\Sigma(6){-}6{=}(1{+}2{+}3{+}6){-}6{=}6$,
$\Sigma(9){-}9{=}(1{+}3{+}9){-}9{=}4$,
$\Sigma(12){-}12{=}(1{+}2{+}3{+}4{+}6{+}12){-}12{=}16$,
$\Sigma(28){-}28{=}(1{+}2{+}4{+}7{+}14{+}28){-}28{=}28$.

In general,a number n is called a *perfect number* when $\Sigma(n) - n = n$ (e.g. 6 and 28), an *abundant number* when $\Sigma(n) - n > n$ (e.g. 12) and a *deficient number* when $\Sigma(n) - n < n$ (e.g. 9). (Even perfect numbers are alway of the form $2^{k-1}(2^k - 1)$, where $2^k - 1$ is a prime number,i.e. Mersenne number. It is known by computer experiments that there is no odd perfect number less than 10^{50}.) The problem is to judge whether a given number is perfect,abundant or deficient.

```
function judge(N:positive-number):kind-of-number.
   judge(N)=perfect where 2×N=divisors-sum(compactify(factorize(N))).
   judge(N)=abundant where 2×N<divisors-sum(compactify(factorize(N))).
   judge(N)=deficient where divisors-sum(compactify(factorize(N)))<2×N.
end.
```

It is well known that

$\Sigma(m \cdot n) = \Sigma(m) \cdot \Sigma(n)$ when m and n are relatively prime.
$\Sigma(p^{k+1}) = 1 + p + p^2 + \cdots + p^{k+1} = p \times \Sigma(p^k) + 1$ when p is a prime number.

Hence it is easy to compute $\Sigma(n)$ once we know the factorization of n into primes $p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l}$. We factorize a given number as follows.

```
function factorize(N:positive-number):numbers-list.
   factorize(1)=[ ].
   factorize(N)=Fs where 1<N,
                         I*=[2|I*+1],
                         D*=sift(I*),
                         Fs=if N<D* ×D* then [N] else [D*|factorize(Q)]
                              asa N<D* ×D* ∨ divide(N,D*,Q,0).
end.
```

where D^* is the stream of primes computed using the Erathosthenes's sieve.

```
   sift([P|N*])=filter(P,N*).
   filter(P,[X|N*])=filter(P,N*) where divide(X,P,Q,0).
   filter(P,[X|N*])=[X|filter(P,N*)] where divide(X,P,Q,R+1).
```

For example,28 is factorized into $[2, 2, 7]$. We use it after compactifying k duplicated factors p to a pair (p, k). For example, $factorize(28) = [2, 2, 7]$ is converted to $[[2, 2], [7, 1]]$ by the function *compactify*. Then $\Sigma(n)$ is computed by multiplying $1 + p + p^2 + \cdots + p^k$ for all compactified factors p^k of n.

```
function divisors-sum(CF:pair-of-numbers-list):number.
    divisors-sum([ ])=1.
    divisors-sum([[P,K]|CFs])=X×divisors-sum(CFs)
                                   where I*=[1|I*+1],
                                         DS*=[1|P×DS*+1],
                                         X=DS* asa K<I*.
end.
```

Note that the recurrence formula for $\Sigma(p^{k+1})$ is directly programmed by $DS^* = [1|P \times DS^*]$. The Talos interpreter answers

!Value=perfect;

to queries ?-*judge*(6), ?-*judge*(28) or ?-*judge*(496) and

!Value=abundant;

to a query ?-*judge*(12).

Remark. These programs suggest several possibilities of optimization. For example, it is wastefull to compute *divisors-sum(compactify(factorize(N)))* for each equations in the definition of *judge* all the way. The computed result should be saved after the first computation and utilized in the computation thereafter.

7. Discussions

(1) Relations to Other Works

Several attempts have been done to amalgamate relational programming languages and functional programming languages. Bellia [3] introduced Horn clauses with equalty into relational program, but their language is substantially completely deterministic functional programming language. Fribourg [7] used equational Horn clause and clarified its semantics based on the paramodulation, which is very similar to the general narrowing. But because he did not impose any conditions (like confluence and termination), his completeness theorem needed superposition between programs and additional functional reflexive axioms. Moreover the narrowings was not restricted to those at occurrences of non-variable terms. Tamaki [20] introduced a reducibility predicate into Prolog and defined its semantics based on source-level expansion of nested terms to conjunction of atoms. Because he did not impose the termination condition, he had to add the reflexivity of $\rightarrow^*$ to the expanded programs, which plays a very important role. Goguen and Meseguer [9] suggested the use of narrowing in computation in their functional-relational language Eqlog based on rigouros logical basis of many sorted logic. They allowed general algebraic specification of abstract data types, which does not always assume existence of constructors, and used the general narrowing.

We claim that our framework makes the programming reasonably easy as well as the interpreter reasonably efficient. Eqlog's general data type specification indeed gives high expressive power. Though Talos lacks such generality, existence of constructors is helpfull not only for programmers but also the meta-unification process. To programmers who use such languages as *programming* languages, it gives concrete symbolic objects to manipulate and conceive easily in mind. From the meta-unification process, it alleviates the too frequent check of unifiability and enables us to compare corresponding terms only when they are semi-constructor terms. (Note that we always have to compare corresponding terms at the first when in the Fay-Hullot's algorithm. cf. comment in [9] p.206). The constructor terms in Talos exactly do play the same role as general terms in Prolog do.

(2) Problems Left for Future

(i) Improvement of Implementation and Experience of Programming

Some assignments of meta-unifiers are decided from purely equational inferences (reduction) and don't need search (narrowing with instanciation to nonvariable terms). For example,the Talos interpreter described so far tries to meta-unify the condition part of the following query

?-N **where** append(L,M)=[A|N],[A|L]=[a,a|K].

from the left to the right as follows.

$\{\ \underline{\mathrm{append}(L,M)}=[A|N],[A|L]=[a,a|K]\ \}$

$\Downarrow$ term-reduction by a narrowing substitution $< L \Leftarrow [\,] >$

$\{\ \underline{M=[A|N]},[A]=[a,a|K]\ \}$

$\Downarrow$ lazy-variable-elimination and obtain $< M \Leftarrow [A|N] >$

$\{\ \underline{[A]=[a,a|K]}\ \}$

$\Downarrow$ fail the meta-unification of $A = [a, a|K]$ and backtrack

But if the second equation is meta-unified (without variable instantiation) first, it goes without backtracking as follows.

$\{\ \mathrm{append}(L,M)=[A|N],\underline{[A|L]=[a,a|K]}\ \}$

$\Downarrow$ term-reduction and obtain $< A \Leftarrow a, L \Leftarrow [a|K] >$

$\{\ \underline{\mathrm{append}([a|K],M)}=[a|N]\ \}$

$\Downarrow$ term-reduction

$\{\ \underline{[a|\mathrm{append}(K,M)]=[a|N]}\ \}$

$\Downarrow$ term-reduction

$\{\ \underline{\mathrm{append}(K,M)=N}\ \}$

$\Downarrow$ lazy-variable-elimination and obtain $< N \Leftarrow \mathit{append}(K, M) >$

$\{\ \}$

This suggests that, it alleviates unnecessary backtracking to separate the "don't care" nondeterministic part (reduction without instanciation of variables) from the "don't know" nondeterministic part (narrowing with instantiations of variables). In our current interpreter, we always normalize $\mathcal{E}$ to $\mathcal{E}\downarrow$, i.e. the set of equations whose terms are all normalized by $\underline{\mathcal{R}}$, before *rewrite-equations* is applied. But the "don't know" determinism assumes the confluence property of $\underline{\mathcal{R}}$, which is sometimes difficult to guarantee syntactically. If the confluence of the definition of functions is guaranteed by some sufficient condition, then the order of rules is irrelevant and ignorable. But if not,the interpretation depends on the sequentiality "from top to bottom" as does in Prolog. In such a case,the optimization above may complicate the backtracking structure.

Talos was implemented in MACLISP from April in 1982 to March in 1983. Several interpreters of functional and logic programming languages (Lazy Lisp,Call by Need Prolog and Lucid) were described in Talos (cf.[11]). But we need more programming experience in Talos to examine its advantages and disadvantages.

(ii) Extensions of Talos

Composition of Data Types : In the definitions of data types in 2.1, new data types are defined by "**data** ⋯ == new". But we need another definition methods to compose from existing data types like PASCAL, e.g. subtype,sum,product,sequence,powerset,mapping,quotient (by congruence), applications(instantiation of type parameters) etc. Eqlog already provides such methods with rigorous logical base [9].

Unfree Data Structure and *Set Abstraction* : Free data structures are not enough to express and solve various problems. The most familiar exception is *array* and we have to resort to the theory of *confluence modulo congruence* ([12],[16]). Description using unfree data structures often provides elegant solutions of considerably complicated problems like "paraffine problem" (Turner [21]). The *set abstraction* is another powerfull expression and we expect it is not very difficult to add set abstraction to Talos.

(iii) Experiment of Program Transformation in Talos

Originally Talos is intended to be a base language for program transformation as is HOPE [4]. For example, the "computation with stream" is included in Talos to treat "recursion elimination" and "recursion introduction" without going out of the functional framework to the imperative one (cf.[1]). The "nondeternistic function" provides us to define nondeterministic representation functions in the same way as data types and functions, which alleviates difficulties so far purely functional language has suffered for in defining relations between data structures. In addition, reduction in Talos carries out the easiest "partial evaluation".

8. Conclusions

We have presented the language features of a meta-unification based language Talos by stepwise extension.

Acknowledgements

The author would like to express deep gratitude to Dr.Joseph.A.Goguen (SRI International and Center for the Study of Language and Information,Stanford University) for kindly informing of Eqlog and Dr.Kō Sakai (ICOT 2nd Laboratory) for pointing out errors in the previos equational metaunification algorithm.

This work is based on the activity of TRS WG (Term Rewriting Systems Working Group) for the FGCS(Fifth Generation Computer System) project. The author would like to thank to Dr.K.Fuchi (ICOT Director) and Dr.S.Yokoi (Chief of ICOT 2nd Laboratory) for the chance of this research.

References

[1] Arsac,J. and Y.Kodoratoff,"Some Techniques for Recursion Removal from Recursive Functions",ACM TOPLAS,Vol.4,No.2,pp.295-322,1982.
[2] Ashcroft,E.A. and W.W.Wadge,"Lucid,a Nonprocedural Language with Iteration",C.ACM Vol.20,No.7,pp.519-526,1977.
[3] Bellia,M.,P.Degano and G.Levi,"The Call-by-Name Semantics of a Clause Language with Functions", in Logic Programming (K.Clark and S-A.Tarnlund Eds),pp.281-295 ,1982.
[4] Burstall,R.M.,D.B.MacQueen and D.T.Sannela,"HOPE:An Experimental Applicative Language", Proc. of 1980 LISP Conference,pp.136-143, 1980.
[5] van Emden,M.H. and R.A.Kowalski,"The Semantics of Predicate Logic as Programing Language",J.ACM,Vol.23,pp.733-742,1976.
[6] Fay,M.,"First-order Unification in Equational Theory",4th Workshop on Automated

Deduction, pp.161-167,1979.
[7] Fribourg,L.,"Oriented Equational Clauses as A Programming Language", J.Logic Programming, Vol.1,No.2,pp.165-177,1984.
[8] Friedman,D., and Wise,D.,"CONS Should Not Evaluate Its Arguments", Automata,Language and Programming, Edinburgh University Press, pp.257-284,1976.
[9] Goguen,J.A.and J.Meseguer,"Equality,Types,Modules and (Why Not?) Generics for Logic Programming",J.Logic Programming,Vol.1,No.2, pp.179-210,1984.
[10] Henderson,P. and J.H.Morris,"A Lazy Evaluator", Conference Record of 3rd ACM Symposium on Principles of Programming Languages, pp.95-103, 1976.
[11] Hoffman,M. and M.J.O'Donnell,"Programming with Equations", ACM TOPLAS, Vol.4,No.1, pp.83-112, 1982.
[12] Huet,G.,"Confluent Reduction : Abstract Properties and Applications to Term Rewriting System", J.ACM,Vol.27,No.4,pp.797-821, 1980.
[13] Huet,G. and J-M.Hullot,"Proofs by Inductions in Equational Theories with Constructors", J.Computer and System Science, 25, pp.239-266, 1982.
[14] Huet,G., and J-J.Levy,"Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems", INRIA Research Report 359, August 1979.
[15] Huet,G. and D.C.Oppen,"Equations and Rewrite Rules : a Survey", in Formal Language Theory : Perspectives and Open Problems (R.V.Book Ed), pp.349-405,Academic Press,1980.
[16] Hullot,G.M.,"Canonical Forms and Unification",in 5th Conference on Automated Deduction (W.Bibel and R.A.Kowalski Eds), pp.318-334,1980.
[17] Isobe,T.,"Eight Satin"(in Japanese),in Special Issue of Mathematical Sciences on "Original Puzzles VI",1982.
[18] Kanamori,T.,"Computation by Meta-Unification with Constructors", ICOT Technical Report,TR-1??,to appear,1985.
[19] Pereira,L.M.,F.C.N.Pereira and D.H.D.Warren"User's Guide to DECsystem-10 Prolog", Occational Paper 15,Dept.of Artificial Intelligence, Edinburgh,1979.
[20] Tamaki,H.,"Semantics of A Logic Programming Language with Reducibility Predicate",Proc. 1984 International Symposium on Logic Programming, pp.259-264,1984.
[21] Turner,D.A.,"The Semantic Elegance of Applicative Languages", Proc. of 1981 Conference on Functional Programming Languages and Computer Architectures, pp.85-92, 1982.
[22] Vuillemin,J.,"Correct and Optimal Computation of Recursion in a Simple Programming Language", J.Computer and System Science 9, pp.332-354, 1974.