A Deterministic Lazy Conditional Narrowing and its Implementation in Mathematica

Mohamed Hamada
Doctoral Program in Engineering, University of Tsukuba
Tsukuba 305, Japan
E-mail: hamada@score.is.tsukuba.ac.jp
and

Tetsuo Ida
Institute of Information Sciences and Electronics, University of Tsukuba
Tsukuba 305, Japan
E-mail: ida@score.is.tsukuba.ac.jp

Abstract

Narrowing was studied in the literature to give operational semantics to functional logic languages. In this paper we introduce a (deterministic) Lazy Conditional Narrowing Calculus (LCNC_d for short). We describe a full implementation of LCNC_d using Mathematica 3.0. We demonstrate that our implementation results in a (first hand) functional-logic language interpreter as well as an equational theorem prover.

1 Introduction

Narrowing was discussed in the literature on programming as an operation for giving a promising operational semantics of declarative languages based on the integration of functional and logic languages, see [5] for a recent survey. Narrowing in its original definition contains two types of non-determinism: the choice of the subexpression to be narrowed and the choice of the rewrite rule. This non-determinism behavior of narrowing results in a huge search space even for simple problems. Due to this fact any implementation of a functional logic language based on narrowing (in its original definition) will be very inefficient. To improve this inefficiency, several attempts are made.

One that we investigated is to decompose narrowing into more basic operations. These basic operations are represented as inference rules and the set of inference rules is formalized as a calculus. In [10] a calculus called lazy narrowing calculus LNC is introduced and shown complete for various classes of term rewriting systems (TRSs for short). Moreover, in [9] a deterministic version of LNC (called LNC_d) was given. Completeness of LNC_d was also studied.

In this paper we extend LNC_d to conditional term rewriting systems (CTRSs for short), resulting in the deterministic Lazy Conditional Narrowing Calculus LCNC_d. Although LCNC_d is a straightforward extension of LNC_d, its completeness is rather difficult to show. Completeness means that for every solution of a given goal a more general solution can be found by LCNC_d. However soundness of LCNC_d is easy to show. Soundness means that starting from a given goal G every successful LCNC_d derivation yields a solution of G.

In this paper we give a full implementation of LCNC_d. We use Mathematica 3.0 in our implementation. Existentially quantified equations can be solved by our calculus with respect to given rewrite rules. Thus our implementation extends the symbolic computation language Mathematica based on higher-order conditional rewriting. Our implementation results in a calculus that is usable in a variety of applications such as a first-hand functional logic language interpreter as well as an equational theorem prover.

In addition, our implemented calculus can be used as a research tool usable within Mathematica for studying equation solving with respect to various classes of (conditional) term rewriting systems.

The paper is organized as follows. In the rest of this Section we recall some basic definitions of rewriting and narrowing. LNC and its deterministic version LNC_d will be given in Section 2. In Section **??** the extension LCNC_d of LNC_d will be introduced. Our implementation of LCNC_d will be discussed in Section 4.

We assume familiarity with the basics of (conditional) term rewriting and narrowing. Surveys can be found in [2, 7, 8].

A conditional term rewriting system (CTRS for short) over a signature $\mathcal{F}$ is a set $\mathcal{R}$ of (conditional) rewrite rules of the form $l \to r \Leftarrow c$ where the conditional part c is a (possibly empty) sequence $s_1 \approx t_1, \ldots, s_n \approx t_n$ of equations. All terms $l, r, s_1, \ldots, s_n, t_1, \ldots, t_n$ must belong to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and we require that l is not a variable. We assume that every CTRS contains the rewrite rule $x \approx x \to \mathtt{true}$. Here $\approx$ and $\mathtt{true}$ are function symbols that do not occur in the other rewrite rules. These symbols may only occur at the root position of terms.

An equation is a term of the form $s \approx t$. The constant $\mathtt{true}$ is also viewed as an equation. A goal is a sequence of equations. A substitution θ is a ($\mathcal{R}$-)solution of a goal G if $G\theta \to_{\mathcal{R}}^* \top$. Here $\top$ stands for any sequence of $\mathtt{true}$'s. For confluent $\mathcal{R}$ this is equivalent to validity of the equations in $G\theta$ in all models of the underlying (conditional) equational system of $\mathcal{R}$ (Kaplan [6]). The conditional narrowing calculus CNC consists of the following inference rule:

$$\frac{G', e, G''}{(G', e[r]_p, c, G'')\theta}$$

if there exist a fresh variant $l \to r \Leftarrow c$ of a rewrite rule in $\mathcal{R}_+$, a non-variable position p in e, and a most general unifier θ of $e_{|p}$ and l.

In the above situation we write $G', e, G'' \leadsto_\theta (G', e[r]_p, c, G'')\theta$. This is called a CNC-step. A sequence $G_1 \leadsto_{\theta_1} \cdots \leadsto_{\theta_{n-1}} G_n$ of CNC-steps is called a CNC-derivation and abbreviated to $G_1 \leadsto_\theta^* G_n$ where $\theta = \theta_1 \cdots \theta_{n-1}$. A CNC-derivation which ends in $\top$ is called a CNC-refutation. If $G \leadsto_\theta^* \top$ then θ is a solution of G. This is known as the soundness of conditional narrowing.

If we use the unconditional rule $l \to r$ instead of the conditional one and then remove the conditional part c from the lower part of CNC defined above we get the narrowing calculus NC. All other notations such as NC-step, NC-derivation and NC-refutation are the same as CNC without conditions.

2 Lazy Narrowing Calculus

In this section we recall the lazy narrowing calculus LNC of Middeldorp, Okui and Ida [10], and its deterministic version LNC_d of Middeldorp, and Okui [9].

2.1 Non-deterministic LNC

The *lazy narrowing calculus* LNC as introduced in [10] consists of the following five inference rules:

[o] *outermost narrowing*

$$\frac{G', f(s_1, \ldots, s_n) \simeq t, G''}{G', s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, G''}$$

if there exists a fresh variant $f(l_1, \ldots, l_n) \to r$ of a rewrite rule in $\mathcal{R}$.
The equations $s_1 \approx l_1, \ldots, s_n \approx l_n$ is called parameter passing equations and the equation $r \approx t$ is called body equation.

[i] *imitation*

$$\frac{G', f(s_1, \ldots, s_n) \simeq x, G''}{(G', s_1 \approx x_1, \ldots, s_n \approx x_n, G'')\theta}$$

if $\theta = \{x \mapsto f(x_1, \ldots, x_n)\}$ with $x_1, \ldots, x_n$ fresh variables,

[d] *decomposition*

$$\frac{G', f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n), G''}{G', s_1 \approx t_1, \ldots, s_n \approx t_n, G''},$$

[v] *variable elimination*

$$\frac{G', s \simeq x, G''}{(G', G'')\theta}$$

if $x \notin \mathcal{V}\mathrm{ar}(s)$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{G', x \approx x, G''}{G', G''}.$$

In the rules [o], [i], and [v], $s \simeq t$ stands for $s \approx t$ or $t \approx s$.

If G and G' are the upper and lower goal in the inference rule [α] ($\alpha \in \{o, i, d, v, t\}$), we write $G \Rightarrow_{[\alpha]} G'$. This is called an LNC-step. The applied rewrite rule or substitution may be supplied as subscript, that is, we write things like $G \Rightarrow_{[o], l \to r} G'$ and $G \Rightarrow_{[i], \theta} G'$. A finite LNC-derivation $G_1 \Rightarrow_{\theta_1} \cdots \Rightarrow_{\theta_{n-1}} G_n$ may be abbreviated to $G_1 \Rightarrow_{\theta}^{*} G_n$ where $\theta = \theta_1 \cdots \theta_{n-1}$. An LNC-refutation is an LNC-derivation ending in the empty goal $\Box$.

In [10] the following completeness results for LNC was obtained.

Theorem 1 [10]. Let $\mathcal{R}$ be a confluent TRS, and G a goal. For every normalized solution θ of G, LNC is complete provided one of the following conditions is satisfied:
(1) $\mathcal{R}$ is terminating,
(2) $\mathcal{R}$ is orthogonal and $G\theta$ has an $\mathcal{R}$-normal form, or
(3) $\mathcal{R}$ is right linear.

The following result is more interesting since it replaces the strong termination restriction on TRSs with a much simpler one.

Theorem 2 [10]. Let $\mathcal{R}$ be a confluent TRS, and G a goal. For every normalized solution θ of G, LNC is complete with respect to the selection function $\mathcal{S}_{left}$.

Here the selection function $\mathcal{S}_{left}$ means that in every LNC-step we select the leftmost equation in the goal.

LNC has three sources of non-determinism: the choice of the equation, the choice of the inference rule, and the choice of the rewrite rule (in case of the outermost narrowing rule). Theorem 2 improves this non-deterministic behavior by restricting the equation selection to $\mathcal{S}_{left}$, this means that the first source of non-determinism is disappear. However, the other two sources make any implementation of LNC inefficient. The following section will discus a (more) deterministic version of LNC.

2.2 LNC_d: a deterministic LNC

A deterministic version of LNC called LNC_d was given in [9]. LNC_d results in a significant reduction of the search space and ease the implementation.

In LNC_d we distinguish between the descendants of parameter passing equations and descendants of initial equations, we use $\asymp$ rather than $\approx$ to denote the former. LNC_d consists of the following two groups of inference rules. The first group of rules are designed for descendants of initial equations:

[o] *outermost narrowing*

$$\frac{f(s_1,\ldots,s_n) \approx t, G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, G}$$

and

$$\frac{t \approx f(s_1,\ldots,s_n), G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, G}$$

if there exists a fresh variant $f(l_1,\ldots,l_n) \to r$ of a rewrite rule in $\mathcal{R}$, where the root of t is not a defined function symbol.

[i] *imitation*

$$\frac{f(s_1,\ldots,s_n) \approx x, G}{(s_1 \approx x_1, \ldots, s_n \approx x_n, G)\theta}$$

and

$$\frac{x \approx f(s_1,\ldots,s_n), G}{(s_1 \approx x_1, \ldots, s_n \approx x_n, G)\theta}$$

where f is a constructor symbol, $x \in \mathcal{V}\text{ar}(f(s_1,\ldots,s_n))$ or $f(s_1,\ldots,s_n)$ is not a constructor term and $\theta = \{x \mapsto f(x_1,\ldots,x_n)\}$ with $x_1,\ldots,x_n$ fresh variables,

[d] *decomposition*

$$\frac{f(s_1,\ldots,s_n) \approx f(t_1,\ldots,t_n), G}{s_1 \approx t_1, \ldots, s_n \approx t_n, G},$$

where f is a constructor symbol,

[v] *variable elimination*

$$\frac{s \approx x, G}{G\theta}$$

and

$$\frac{x \approx s, G}{G\theta}$$

where s is a non-variable constructor term, $x \notin \mathcal{V}\text{ar}(s)$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{x \approx x, G}{G}.$$

The second group of inference rules of LNC_d deals with descendants of parameter passing equations:

[o-p] *outermost narrowing for parameter passing equations*

$$\frac{f(s_1, \ldots, s_n) \asymp t, G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \asymp t, G}$$

if there exists a fresh variant $f(l_1, \ldots, l_n) \to r$ of a rewrite rule in $\mathcal{R}$, where the root of t is not variable,

[d-p] *decomposition for parameter passing equations*

$$\frac{f(s_1, \ldots, s_n) \asymp f(t_1, \ldots, t_n), G}{s_1 \asymp t_1, \ldots, s_n \asymp t_n, G},$$

where f is a constructor symbol,

[v-p] *variable elimination for parameter passing equations*

$$\frac{s \asymp x, G}{G\theta}$$

and

$$\frac{x \asymp s, G}{G\theta}$$

where s is a non-variable term, and $\theta = \{x \mapsto s\}$.

LNC_d drops the second source of non-determinism in LNC (i.e. the choice of the inference rule). Although LNC_d still has one source of non-determinism (i.e. the choice of the rewrite rule in case of outermost narrowing) it gives more significant reduction of the search space as well as easing the implementation.

In functional logic programming two expressions are said to be *strictly equal* if and only if they reduce to the same ground constructor normal form. A substitution θ is said to be *strict* solution of a goal G if for every equation e in G, there exists a ground constructor term t such that both sides of e instantiated by θ reduces to t.

The following completeness result was obtained in [9] for LNC_d with respect to strict solutions..

Theorem 3 [9]. LNC_d is complete for orthogonal constructor based TRSs with respect to strict normalized solutions.

3 Lazy conditional narrowing calculus

In this section we extend LNC_d to the conditional case, resulting in the deterministic lazy conditional narrowing calculus LCNC_d. The only difference between LNC_d and LCNC_d is the addition of the conditional part of the applied rewrite rule in the case of the outermost narrowing rule. So in LCNC_d the outermost narrowing rule for descendants of initial equations is defined as follows:

[o] *outermost narrowing*

$$\frac{f(s_1,\ldots,s_n) \approx t, G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, c, G}$$

and

$$\frac{t \approx f(s_1,\ldots,s_n), G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, c, G}$$

if there exists a fresh variant $f(l_1,\ldots,l_n) \to r \Leftarrow c$ of a rewrite rule in $\mathcal{R}$, where the root of t is not a defined function symbol.

Similarly, the outermost narrowing for parameter passing equations is defined as follows:

[o-p] *outermost narrowing for parameter passing equations*

$$\frac{f(s_1,\ldots,s_n) \asymp t, G}{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \asymp t, c, G}$$

if there exists a fresh variant $f(l_1,\ldots,l_n) \to r \Leftarrow c$ of a rewrite rule in $\mathcal{R}$, where the root of t is not variable.

All the other rules are exactly the same as in LNC$_d$.

Soundness of LCNC$_d$ is easy to see while completeness for large classes of CTRSs of interest is difficult to prove. In our future work we will investigate the completeness of LCNC$_d$.

Although completeness is desirable property, soundness is sufficient in some practical applications in which one solution is sufficient.

It is worth mentioning that In [4] LCNC (the extension of LNC to the conditional case) is given and shown (strong) complete for various classes of CTRSs.

4 Implementation of LCNC$_d$

In this section we give a full implementation of LCNC$_d$ using Mathematica 3.0. We assume that the reader is familiar with Mathematica system, see [11] for a complete reference.

Although Mathematica is a good framework for working with rewrite systems [1] basic data structures for our systems are necessary. Mathematica has some idiosyncrasy (seen from a rewriting point of view) and this makes our implemented system slightly more lengthy than is desired, although the extra code seems small compared with the power Mathematica provides for the implementation of the calculus.

4.1 Signature

A *signature* consists of the following three classes of symbols.

1. a class of defined-function symbols,
2. a class of constructor symbols, and
3. a class of variables.

A class of symbols that are defined-function symbols or constructors are called function symbols. Pragmatically, these classes are distinguished by the ways they are introduced. A signature is initially set as follows:

```
DefineSignature[]:=
      (IsConstructor[s_]/; s[[0]] ≠ Symbol ∧ AtomQ[s]=True;
      IsFunctionSymbol[s_]/; s[[0]] ≠ Symbol ∧ AtomQ[s]=True;
      Constructor[List,Sequence,eq,ppeq];)
```

Functions `IsConstructor`, `IsFunctionSymbol`, and `Constructor` are defined in the following subsections.

4.2 Variable

A variable is used to denote an arbitrary term in the usual way. We distinguish variables by the following rule. Within goals only the symbols for which solutions are to be bound, i.e. the second argument of `TSolve[_,_]`, (see Section 4.8.1 for definition of `TSolve[_,_]`) is considered as a variable. All the other symbols are considered as function symbols. Logically, the variables are existentially quantified variables in the goals.

Within rewrite rules, any symbol that is not a function symbol is considered as a variable. To distinguish variables from other symbols during the solving process, we use Mathematica's attributes attached to symbols. We can set attribute `Temporary` (and only `Temporary` attribute) onto variables (of our system). Thus we define `IsVar[s_]`, which decides whether `s` is a variable, as follows.

```
IsVar[s_Symbol]:=Attributes[s]==={Temporary}
```

In the sequel a decision procedure for $x \in W$ is defined as

$$\text{Is}\, W[x] \quad = \quad \begin{cases} \texttt{True} & \text{if } x \in W, \\ \text{non}\texttt{--True}\ \textit{expression} & \text{if } x \notin W. \end{cases}$$

4.3 Constructor

A constructor symbol (abbreviated as constructor by convention) is used to construct a data structure. To define a constructor symbol, we use `Constructor[___]`. A constructor symbol is also declared as `FunctionSymbol` at the same time of declaration. The Mathematica program for `Constructor[___]` is defined as follows.

```
Constructor[fs___]:=
      (Scan[Function[x,IsFunctionSymbol[x]=True;
                     IsConstructor[x]=True],{fs}];
      {fs})
```

We pre-assign meaning to existing symbols as follows. Mathematica atoms other than those whose type is `Symbol` are automatically declared as a constructor symbol. Mathematica symbols `List` and `Sequence` are defined as a constructor, too.

4.4 Defined-function symbols

A head symbol of a left-hand side of a rewrite rule is called a defined-function symbol. A defined-function symbol is used to name a set of rewrite rules. Whether a symbol is a defined-function symbol or not is checked by a function `IsDefinedFunctionSymbol[_]`. The value of `IsDefinedFunctionSymbol[f]` for symbol `f` is set `True` whenever `f` is introduced by `RewriteRule[f[___],_,___]` (see Section 4.6 for the definition of `RewriteRule[___]`).

The above definition of a defined-function symbol suffers from small technical difficulty because of the way that rewrite rules are introduced by `RewriteRule[__]`.

For efficient manipulation of rewrite rules, the present implementation has to distinguish between variables and function symbols in a rewrite rule at the time the rewrite rule is introduced. It is impossible to detect a defined-function symbol occurring in the left-hand side or the condition part of a rewrite rule if that defined-function symbol is defined by `RewritRule[__]` later. This is a general problem of forward referencing. The usual way to circumvent this problem is to declare a defined-function symbol before it is defined. For this purpose, we introduce `DefineFunctionSymbol[___]`.

```
DefinedFunctionSymbol[fs___]:=
       (Scan[Function[x,IsFunctionSymbol[x]=True;
                       IsDefinedFunctionSymbol[x]=True],{fs}];
       {fs})
```

4.5 Function Term

A function term is a term whose head is a function symbol. Besides the functions that discriminate terms, the following `IsFunctionTerm[]` is used to distinguish function terms.

```
IsFunctionTerm[f_[___]]/;IsDefinedFunctionSymbol[f]:=True
```

Final small remark regarding the data structures that may confuse term rewriting people: In Mathematica `c[]` and `c` are different. So is in our system.

4.6 Rewrite Rule

We adopt our own representation for conditional rewrite rules. We cannot use Mathematica rewrite rules directly. If we defined our rewrite rules as Mathematica rewrite rules, we would have a complex interaction between our rewrite rules and Mathematica rewrite rules. Therefore we newly design internal data structures for rewrite rules of our systems.

We define an optimized representation of conditional rewrite rules. The idea is based on the observation that the conditional rewrite rule $f[s] \to t \Leftarrow c$ can be represented by $f[x] \to t \Leftarrow x = s, c$, where $x = s$ is a parameter passing equation in the case of s being a non-variable term. This representation makes it possible

1. to eliminate runtime creation of parameter passing equations, and
2. to use parameter binding mechanism of Mathematica rather than applying the variable elimination rules of our system by representing the conditional rewrite rule as a function `Function[{x}, t ⇐ x=s,c]` or `Function[{s}, t ⇐ c]` if `s` is a variable.

`RewriteRule[lhs, rhs, condition]` is used to define a rewrite rule `lhs` → `rhs` ⇐ `condition`, where `lhs` and `rhs` are left and right hand side of a defining rewrite rule and `condition` is a (possibly empty) sequence of equations. `RewriteRule[_,_,___]` is defined as follows:

```
RewriteRule[s:symb_[___],t_,c___]:=
       Module[{parameter,condition,patternV,dummyV,localV},
              IsDefinedFunctionSymbol[symb]=True;
              IsFunctionSymbol[symb]=True;
              {parameter,condition,patternV,dummyV}= Uniform[s];
```

```
            localV=Complement[Var[c],dummyV] ∪ patternV;
            PrintVarDecl[Var[s,c]];
            AppendTo[RewriteRuleList[symb],
                     FormFunction[parameter,localV,t,
                                  Join[condition,{c}]]]]

PrintVarDecl[vset_]:=Switch[vset,
     {},-1,
     {_}, Print[``Symbol ``, vset[[1]], `` declared as variable.''],
     {__}, Print[``Symbols``,Infix[vset,'',''],''declared as variable'']]

FormFunction[par_,localVar_,t_,{c___}]:=
   Function[par,Module[localVar,RewritesTo[t,c]]]

Uniform[s:_[___]]:=
       Block[{Cond={},PatternVar={},DummyVar={}},
             { ReplacePart[Flattening/@s,List,0],
                           Cond,PatternVar,DummyVar}]

Flattening[x_]/;IsFunctionSymbol[x]:=
   Module[{w=Module[{a},a]},AppendTo[Cond,ppeq[w,x]];w]
Flattening[x_]/;AtomQ[x]:=(AppendTo[DummyVar,x];x)

Flattening[s_]:=
    Module[{w},AppendTo[Cond,ppeq[w,s]];
           PatternVar=Join[PatternVar,Var[s]];
           w]
```

4.7 Implementation of the inference rules of LCNC$_d$

This section describes the implementation of the two groups of inference rules that define LCNC$_d$. Our Mathematica programs are straightforward translation of the formal definition of LCNC$_d$ rules given in Section 3. We will use the notation `ppeq[ , ]` instead of `eq[ , ]` to represent the parameter passing equations.

4.7.1 Base case of induction over the equations (goal)

In this section the base case of induction will be given. The basic induction over equations (goal) is concerned with the empty goal. The following `Lcnc` program shows how we can deal with the empty goal {}.

```
Lcnc[{}]:=(AnswerList=AnswerList ∪ {ComputedAnswer[]};
           If[(NSolutions-=1)==0,
BeginRule     Throw[AnswerList],
              AnswerList])

ComputedAnswer[]:=
        With[{newSubst=Substitution /.
                       Inner[Rule,NewGoalVar,GoalVar,List]},
             FormSubst[{GoalVar,
                        FixedPoint[#1/.newSubst&,GoalVar]}]]
```

```
EmptySequence=Sequence[];

FormSubst[x_]:=
   MapThread[Function[{v,t},If[v===t,EmptySequence,v → t]],x]
```

If the goal is not empty then we can apply one of the following Lcnc rules (programs) depending on the structure of the left most equation in the goal.

4.7.2 [t] & [v] Trivial and Variable elimination

For equations in the form `eq[x, y]` where `x` and `y` are variables, if `x` and `y` are the same variable, the following Lcnc program will simply apply the rule [t]. Otherwise (i.e. `x` and `y` are different variables), it will apply [v]. The Lcnc program for [t] and [v] is defined as follows.

```
Lcnc[{eq[x_/;IsVar[x],y_/;IsVar[y]],eqns___}]:=
     Module[{mapsTo},
            If[x===y,
               Lcnc[{eqns}],
               mapsTo=x → y;
               PrependTo[Substitution,mapsTo];
               Lcnc[{eqns}/.mapsTo]]]
```

4.7.3 [o] Outermost narrowing

To deal with both sides of an equation `eq[s, t]`, two variants of Lcnc program for [o] are necessary (depending on either `s` or `t` is a function term). The Lcnc program for [o] is defined as follows.

```
Lcnc[{eq[s_/;IsFunctionTerm[s],t_],eqns___}]:=
   Scan[NewLcnc[#1,t,eqns]&,VariantsRule[s]]

Lcnc[{eq[s_,t_/;IsFunctionTerm[t]],eqns___}]:=
   Scan[NewLcnc[#1,s,eqns]&,VariantsRule[t]]
```

4.7.4 [i] & [v] Imitation and Variable elimination

This section describes the implementation of the imitation and the variable elimination inference rules. For an equation in the form `eq[x, t]` where `x` is variable, if `t` is an atom, the following Lcnc program will simply apply [v]. Otherwise it will apply [i]. For an equation in the form `eq[t, x]` where `x` is variable, similar reasoning is applied. Hence we get four variants of the Lcnc program defined as follows.

```
Lcnc[{eq[x_/;IsVar[x],s_/;AtomQ[s]],eqns___}]:=
     With[{mapsTo=x → s},
          PrependTo[Substitution,mapsTo];
          Lcnc[{eqns}/.mapsTo]]

Lcnc[{eq[x_/;IsVar[x],s_],eqns___}]:=
     Module[{cs=Function[{}, Module[{c},c]]/@s,mapsTo},
            mapsTo=x → cs;
            PrependTo[Substitution,mapsTo];
            Lcnc[{NewEqns[eq,s,cs],eqns}/.mapsTo]]
```

```
Lcnc[{eq[s_/;AtomQ[s],x_/;IsVar[x]],eqns___}]:=
     With[{mapsTo=x → s},
          PrependTo[Substitution,mapsTo];
          Lcnc[{eqns}/.mapsTo]]

Lcnc[{eq[s_,x_/;IsVar[x]],eqns___}]:=
     Module[{cs=Function[{},Module[{c},c]]/@s,mapsTo},
            mapsTo=x → cs;
            PrependTo[Substitution,mapsTo];
            Lcnc[{NewEqns[eq,s,cs],eqns}/.mapsTo]]
```

4.7.5 [d] Decomposition

Here we consider equations in the form `eq[s, t]` where `s` and `t` are non-variables. If the head symbol (root) of `s` is the same as the head symbol (root) of `t`, the following `Lcnc` program will simply apply [d]. Otherwise (i.e. the head symbols of `s` and `t` are different) and if [o] is not applicable, no solution will be yielded. The `Lcnc` program for [d] is defined as follows.

```
Lcnc[{eq[s_/;AtomQ[s],t_],eqns___}]:=
   If[s===t,Lcnc[{eqns}]]

Lcnc[{eq[s_,t_],eqns___}]:=
   If[s[[0]]=== t[[0]], Lcnc[{NewEqns[eq,s,t],eqns}]]
```

4.7.6 [v-p] Variable elimination for parameter passing

As direct implementation of the variable elimination for parameter passing rule (i.e. [v-p]), two variants of the `Lcnc` program for [v-p] will be defined as follows.

```
Lcnc[{ppeq[s_,x_/;IsVar[x]],eqns___}]:=
     With[{mapsTo=x → s},
          PrependTo[Substitution,mapsTo];
          Lcnc[{eqns}/.mapsTo]]

Lcnc[{ppeq[x_/;IsVar[x],s_],eqns___}]:=
     With[{mapsTo=x → s},
          PrependTo[Substitution,mapsTo];
          Lcnc[{eqns}/.mapsTo]]
```

4.7.7 [o-p] Outermost narrowing for parameter passing

Since LCNC_d calculus is defined for constructor-based CTRSs, the right hand side of parameter passing equations in the form `eq[s, t]` (i.e. `t`) should has a constructor symbol as its head. Based on this fact we will only consider the left hand side of the equation (i.e. `s`) in our `Lcnc` program for [o-p] which is defined as follows.

```
Lcnc[{ppeq[s_/;IsFunctionTerm[s],t_],eqns___}]:=
   Scan[NewLcncPP[#1,t,eqns]&,VariantsRule[s]]
```

4.7.8 [d-p] Decomposition for parameter passing

A similar discussion as in Section 4.7.5 for the decomposition rule [d] is applied here. Similarly the `Lcnc` program for [d-p] is defined as follows.

```
Lcnc[{ppeq[s_/;AtomQ[s],t_],eqns___}]:= If[s===t,Lcnc[{eqns}]]

Lcnc[{ppeq[s_,t_],eqns___}]:=
   If[s[[0]]=== t[[0]], Lcnc[{NewEqns[ppeq,s,t],eqns}]]
```

4.7.9 Auxiliary Definitions

In this section some auxiliary definitions, used in the above programs, will be given.

```
NewEqns[h_,s_,t_]:=
   ReplacePart[Table[h[s[[i]], t[[i]]],{i,Length[s]}],Sequence, 0]

NewLcncPP[RewritesTo[t_,c___],rhs_,eqns___]:=
   Block[{Substitution=Substitution},Lcnc[{c,ppeq[t,rhs],eqns}]]

NewLcnc[RewritesTo[t_,c___],rhs_,eqns___]:=
   Block[{Substitution=Substitution},Lcnc[{c,eq[t,rhs],eqns}]]

VariantsRule[f_[arg___]]:=
   ApplyListOfFunctions[RewriteRuleList[f],{arg}]

ApplyListOfFunctions[{f___},arg_]:=Function[{g},g@@arg]/@{f}
```

4.8 Usage of the System

4.8.1 Term Domain Solver

We present the interface with LCNC$_d$ in the form of a term domain solver `TSolve[_,_]`. The parameter formats of `TSolve` is the same as Mathematica's `Solve[___]`.

Since `TSolve[]` gives all possible solutions of a given goal, the parameter `NSolution` is given to specify the number of solutions that we want to compute. We set the default value of `NSolution` to 256.

We can add some local new rules to the (global) set of rules by using the parameter `Rule`. Local means that these rules will work only with the specified goal and will be deleted after evaluating the solution(s) of that goal. By the same way we can add local constructors and local defined-function symbols by using the (optional) parameters `Constructor` and `DefinedFunctionSymbol` respectively.

If we want to use only the local constructors, defined-function symbols and rules, then we put the `Initialize` parameter to `True`, its default value is set to be `False`. The `TSolve[]` program is defined as follows.

```
Clear[TSolve]

Options[TSolve]={NSolution → 256,
                 Rule :→ {},
                 Initialize → False,
                 Constructor → {},
```

```
                DefinedFunctionSymbol → {}}

TSolve[goal_List,varList_List,opts___]:=
       Block[{AnswerList={},
              GoalVar=varList,
              NSolutions=NSolution/.{opts}/.Options[TSolve],
              NewGoalVar},
             If[NSolutions ≤ 0,
                {},
                Module[{ saveRewriteRuleList=
                                DownValues[RewriteRuleList],
                         saveFunctionSymbol=
                                DownValues[IsFunctionSymbol],
                         saveDefinedFunctionSymbol=
                             DownValues[IsDefinedFunctionSymbol],
                         saveConstructor=DownValues[IsConstructor],
                         newGoal},
                        {newGoal,NewGoalVar}=
                                Module[varList,{goal,varList}];
                        If[Initialize/.  {opts}/.
                               Options[TSolve], BeginRule[]];
                        Scan[Constructor, Constructor/.{opts}/.
                                          Options[TSolve]];
                        Scan[DefinedFunctionSymbol,
                             DefinedFunctionSymbol/.{opts}/.
                                            Options[TSolve]];
                        Rule/.{opts}/.Options[TSolve];
                        Block[{Substitution={}},
                              Catch[Lcnc[newGoal]]];
                        DownValues[RewriteRuleList]=
                                          saveRewriteRuleList;
                        DownValues[IsFunctionSymbol]=
                                             saveFunctionSymbol;
                        DownValues[IsDefinedFunctionSymbol]=
                                      saveDefinedFunctionSymbol;
                        DownValues[IsConstructor]=saveConstructor];
                        AnswerList]]

TSolve[eqn_,varList_List,opts___]:=TSolve[{eqn},varList,opts]

TSolve[goal_List,var_,opts___]:=TSolve[goal,{var},opts]

TSolve[eqn_,var_,opts___]:=TSolve[{eqn},{var},opts]
```

4.8.2 Typical Session

Using the system is an easy task. First we clear all previous definitions of rewrite rules and signature (if any), to avoid confusion, by calling the `BeginRule[ ]` program. After that the system is ready and we can start to define constructors and defined-function symbols, that we will use, by calling the programs `Constructor[__]` and `DefinedFunctionSymbol[__]` respectively. At this point we can start defining our term rewriting system $\mathcal{R}$ by calling

the program `RewriteRule[lhs,rhs]` for all rules `lhs` $\rightarrow$ `rhs` in $\mathcal{R}$. Finally we are ready to solve a `goal` (with respect to $\mathcal{R}$) simply by calling the program `TSolve[goal,varset]` where `varset` is the set of variables occur in the `goal`. A typical session is as follows:

```
BeginRule[ ]
Constructor[ __]
DefinedFunctionSymbol[__]
RewriteRule[s1,t1,c1]
.
.
RewriteRule[sn,tn,cn]
TSolve[goal,varset, optional-parameters]
```

The optional parameters in `TSolve` can be set to control the behavior of `TSolve` (as described in the previous section.)

4.8.3 Extensions

It is sometimes cumbersome and even error-prone to add rewrite rules one by one, since we often leave unnecessary rewrite rules in the system. We provide a function `RuleList[{f___}, {rules___}]`. It first deletes rewrite rules associated with (a sequence of) names `f` and then add rewrite rules.

```
SetAttributes[RuleList,HoldRest];

RuleList[{f___},{rule___}]:=
   (Scan[Function[x,RewriteRuleList[x]={}],{f}];{rule})
```

4.8.4 Example 1

Consider the following constructor-based (unconditional) term rewriting system $\mathcal{R}$.

$$\mathcal{R} \quad = \quad \begin{cases} \texttt{plus}(0,x) & \rightarrow x, \\ \texttt{plus}(\texttt{succ}(x),y) & \rightarrow \texttt{succ}(\texttt{plus}(x,y)) \end{cases}$$

To solve the goal $\texttt{plus}(x,y) \equiv \texttt{succ}(\texttt{succ}(0))$, our system can be used as follows. First we type:

```
BeginRule[]
Constructor[succ]
DefinedFunctionSymbol[plus]
RewriteRule[plus[0, x], x];
RewriteRule[plus[succ[x], y], succ[plus[x, y]]];
TSolve[steq[plus[x, y], succ[succ[0]]], x, y]
```

Then running the system will give the following set of (possible) solutions.

```
{
   {x → 0, y → succ[succ[0]]},
   {x → succ[0], y → succ[0]},
```

```
   {x → succ[succ[0]], y → 0}
}
```

4.8.5 Example 2

Assume a car dealer want to organize cars into categories A, B, and C depending on the car model (old) and color. Red cars of any model and white cars of model two or less (i.e two years old or less) are of category A. Blue cars of any model, green cars of model less than or equal one, and white cars of model greater than two and less than or equal four are of category B. Green cars of model greater than one, and white cars of model greater than four are of category C.

Let cc be a binary function (stands for car category), the above problem can be represented by the following 1-CTRS

$$\mathcal{R} = \begin{cases} cc(model, White) & \rightarrow A \Leftarrow le(model, s(s(0))) = True, \\ cc(model, White) & \rightarrow B \Leftarrow le(model, s(s(0))) = False, le(model, s(s(s(s(0))))) = True, \\ cc(model, White) & \rightarrow C \Leftarrow le(model, s(s(s(s(0))))) = False \\ cc(model, Red) & \rightarrow A \\ cc(model, Blue) & \rightarrow B \\ cc(model, Green) & \rightarrow B \Leftarrow le(model, s(0)) = True, \\ cc(model, Green) & \rightarrow C \Leftarrow le(model, s(0)) = False \\ \\ le(s(x), 0) & \rightarrow False, \\ le(o, x) & \rightarrow True, \\ le(s(x), s(y)) & \rightarrow le(x, y) \end{cases}$$

where $White, Red, Blue, Green, 0, True$, and $False$ are constants, s is a unary function stands for successor, and le is a binary function stands for "less than or equal".
Questions like:
(1) what is the category of new (model 0) white cars?
(2) which cars are of category A?
(3) what is the model and category of green cars?
can be represented by the goals $cc(0, White) = category$, $cc(model, color) = A$, and $cc(model, Green) = category$ respectively.

In our system such CTRS can be represented easily as follows.

```
BeginRule[]
DefinedFunctionSymbol[cc, le]
Constructor[s, true, false, white, red, blue, green, A, B, C]
RewriteRule[cc[model,white], A, eq[le[model,s[s[0]]], true] ];
.
.
.
RewriteRule[le[s[x], s[y]], le[x,y]];
```

To answer the first question (i.e. to solve the first goal), we can simply type the following line.

```
TSolve[eq[cc[0, white], category], {category}]
```

then the system will give the following answer (set of possible solutions).

```
{ {category → A } }.
```

For the second question

```
TSolve[eq[cc[model, color], A], {model, color}]
```

we get the following answer.

```
{
   {color → Red }
   {model → 0, color → White }
   {model → s[0], color → White }
   {model → s[s[0]], color → White }
}
```

Similarly we can get the following answer for the last question.

```
{
   {model → 0, category → B }
   {model → s[0], category → B }
   {model → s[s[x]], category → C }
}
```

5 Conclusion

In this paper we have presented the deterministic Lazy Conditional Narrowing Calculus LCNC_d and its implementation in Mathematica. Those who are interested in our Mathematica program, a full version can be obtained from

http://www.score.is.tsukuba.ac.jp/ hamada/LCNCd-pr.nb.

To run this file you should first save it with the extension .nb (Mathematica files extension) then you use version 3.0 of Mathematica to run the program.

We tried to use Mathematica built-in functions as much as we can. This makes the absolute execution efficiency of our program depends largely on the efficiency of pattern matching and rewriting of the Mathematica system.

Our future work will be in two directions. The theoretical one is to study the completeness of LCNC_d for various classes of CTRSs and to extend this calculus to higher-order one. The practical one is to extend the current implementation in various ways: to cooperate types in the implemented system and cooperate Mathematica built-in rewriting such as arithmetic simplification and numeric relations.

It is worth mentioning that in our previous work [3] we described a full implementation of a three narrowing calculi developed by SCORE laboratory at the university of Tsukuba. LCNC_d was among these calculi. A full version of that paper can be obtained from

ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1996/97-02.ps.gz.

Acknowledgments

We are grateful for various comments from Aart Middeldorp. We also grateful for various supports and suggestions from Bruno Buchberger. The first author is supported by the

Ministry of Education, Science and Culture (Monbusho), Japan. This work is performed in the framework of Project "Declarative Coordination Programming in Open Computing Environment" supported by the Advanced Information Technology Program (AITP) of Information-technology Promotion Agency(IPA), Japan.

References

[1] B. Buchberger, *Mathematica as a Rewrite Language*, Proceedings of the Fuji International Workshop on Functional and Logic Programming, Shonan, World Scientific, Singapore, 1997.

[2] N. Dershowitz and J.-P. Jouannaud, *Rewrite Systems*, in: Handbook of Theoretical Computer Science, Vol. B (ed. J. van Leeuwen), North-Holland, pp. 243–320, 1990.

[3] M. Hamada and T. Ida, *Implementation of Lazy Narrowing Calculi in Mathematica*, RISC-Linz Report Series No. **97-02**, Johannes Kepler Univ., Austria, 1997.

[4] M. Hamada and A. Middeldorp, *Strong Completeness of a Lazy Conditional Narrowing Calculus*, Proceedings of the Fuji International Workshop on Functional and Logic Programming, Shonan, World Scientific, Singapore, 1997.

[5] M. Hanus, *The Integration of Functions into Logic Programming: From Theory to Practice*, Journal of Logic Programming **19 & 20**, pp. 583–628, 1994.

[6] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

[7] J.W. Klop, *Term Rewriting Systems*, in: Handbook of Logic in Computer Science, Vol. II (eds. S. Abramsky, D. Gabbay, and T. Maibaum), Oxford University Press, pp. 1–116, 1992.

[8] A. Middeldorp and E. Hamoen, *Completeness Results for Basic Narrowing*, Applicable Algebra in Engineering, Communication and Computing **5**, pp. 213–253, 1994.

[9] A. Middeldorp and S. Okui, A Deterministic Lazy Narrowing Calculus, Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susuno, World Scientific, Singapore, pp. 104–118, 1995.

[10] A. Middeldorp, S. Okui, and T. Ida, *Lazy Narrowing: Strong Completeness and Eager Variable Elimination*, Theoretical Computer Science **167**(1,2), pp. 95–130, 1996.

[11] S. Wolfram, *The Mathematica Book*, WolframMedia Inc. Champaing, Illinois, USA, 1996.