Vol. 39 No. 3 Transactions of Information Processing Society of Japan Mar. 1998

Regular Paper

Deterministic and Non-deterministic Lazy Conditional Narrowing and their Implementations

MOHAMED HAMADA[†] and TETSUO IDA[††]

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 functional logic language interpreter as well as an equational theorem prover. We also show that determinism in LCNC provides an efficient implementation.

1. Introduction

Narrowing is discussed in the recent literature on programming as an operation for giving a promising operational semantics of functional logic languages, see Ref. 5) for a recent survey. Implementation of narrowing in its original definition is difficult and inefficient. Difficulty comes from the complexity of a single narrowing step. In each narrowing step we have to do the following: select a subexpression to be narrowed (narex), select a rewrite rule r in such a way that its left-hand side can be unified with the selected narex via a most general unifier θ, and replace the narex with the instance of the right-hand side of r under θ. Inefficiency comes from the non-determinism in each narrowing step (due to narex selection and rewrite rule selection).

To overcome this problem, several attempts are made[6),11),12)]. 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. For such calculi, soundness and completeness are important properties. Soundness means that every successful derivation, starting from a given goal G, yields a solution of G. While completeness means that for every solution of a given goal a more general solution can be found by the calculus. In Ref. 11) a calculus called lazy narrowing calculus LNC is introduced and shown complete for various classes of term rewriting systems (TRSs for short). In our previous work 4) we extend LNC to the conditional case resulting in a lazy conditional narrowing calculus (LCNC for short). We also showed its completeness for various classes of conditional term rewriting systems (CTRSs for short).

LNC (as well as LCNC) contains three sources of non-determinism: the selection of the inference rule, the selection of the equation in the goal to be solved, and the selection of the rewrite rule.

While LNC (and LCNC) solve the difficulty problem and ease the implementation of (conditional) narrowing, the inefficiency problem remains unsolved due to the existence of non-determinism. To deal with the inefficiency problem a deterministic version of LNC (called LNC_d) was given in Ref. 10) and shown complete for various classes of TRSs.

In this paper we extend LNC_d to conditional term rewriting systems, 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. However soundness of LCNC_d is easy to show which is sufficient in some applications where we are interested in one or several solutions.

We give a full implementation of LCNC_d using Mathematica 3.0. Existentially quantified equations can be solved by our calculus with respect to given rewrite rules. Thus our implementation extends the symbolic computation language Mathematica. 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.

† Doctoral Program in Engineering, University of Tsukuba
†† Institute of Information Sciences and Electronics, University of Tsukuba

We also implement LCNC (the non-deterministic calculus) to show the significant efficiency of LCNC$_d$ (the deterministic calculus) compared with LCNC.

The paper is organized as follows. In Section 2 we recall some basic definitions of rewriting and narrowing. LNC and its deterministic version LNC$_d$ will be given in Section 3. In Section 4 the extensions LCNC and LCNC$_d$ of LNC and LNC$_d$ respectively, will be introduced. Our implementation of LCNC and LCNC$_d$ will be discussed in Section 5.

2. Preliminaries

We assume familiarity with the basics of (conditional) term rewriting and narrowing as expounded in Refs. 2), 8), 9).

A conditional term rewriting system 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. Where $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is the set of first order terms defined over $\mathcal{F}$ and a countable set $\mathcal{V}$ of variables. We assume that every CTRS contains the rewrite rule $x \approx x \to \mathtt{true}$. Here $\approx$ and true are function symbols that do not occur in the other rewrite rules. These symbols may only occur at the head position of terms.

An equation is a term of the form $s \approx t$. The constant 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 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[7]).

Definitions of notions like terminating, decreasing, confluent, level-confluent, shallow-confluent, $\mathcal{R}$-normal form, right linear, orthogonal, normalized solution, and n-CTRS (where $n \in \{1, 2, 3\}$), which will be used in this paper, can be found in Ref. 9).

3. Lazy Narrowing Calculus

In this section we recall the lazy narrowing calculus LNC of Middeldorp, et al.[11], and its deterministic version LNC$_d$ of Middeldorp and Okui[10].

3.1 Non-deterministic LNC

The *lazy narrowing calculus* LNC as introduced in Ref. 11) consists of the following five inference rules: let $\mathcal{R}$ be a TRS,

[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 $t \approx s$, and $\mathcal{V}\mathrm{ar}(s)$ stands for the set of variables that occur in a term s.

If G and G' are the upper and lower goals 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 $G \Rightarrow_{[o], l \to r} G'$ 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. □

In Ref. 11) the following completeness results for LNC were obtained.

Theorem 1[11]. 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[11]. 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 eliminated. However, the other two sources make any implementation of LNC inefficient. The following section will discuss a (more) deterministic version of LNC.

3.2 LNC$_d$: A deterministic LNC

A deterministic version of LNC called LNC$_d$ was given in[10]. LNC$_d$ results in a significant reduction of the search space and eases the implementation.

In LNC$_d$ we distinguish between the descendants of parameter passing equations and descendants of initial equations, we hereafter use $\asymp$ to denote a parameter passing equation. Let $\mathcal{R}$ be a TRS, 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 head of t is not a defined function symbol (i.e. not a head of a left-hand side of a rewrite rule in $\mathcal{R}$) .

[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 (symbols in $\mathcal{F}$ other than defined functions are called constructors), $x \in \mathcal{V}\mathrm{ar}(f(s_1,\ldots,s_n))$ 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, $x \notin \mathcal{V}\mathrm{ar}(s)$ 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 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$_d$ eliminates 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 is much simpler and, in practice, more efficient than LNC.

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 reduce to t.

The following completeness result was obtained in Ref. 10) for LNC$_d$ with respect to strict solutions.

Theorem 3 [10]. LNC$_d$ is complete for orthogonal constructor based TRSs with respect to strict normalized solutions.

4. Lazy Conditional Narrowing Calculus

In this section we extend LNC and LNC$_d$ to the conditional case, resulting in the conditional calculus LCNC and the deterministic conditional calculus LCNC$_d$ respectively. The only difference between the extended calculi and the original ones is the addition of the conditional part of the applied rewrite rule in case of the outermost narrowing rule. In LCNC the outermost narrowing rule is defined as follows: let $\mathcal{R}$ be a CTRS,

[o] *outermost narrowing*

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

if there exists a fresh variant $f(l_1,\ldots,l_n) \to r \Leftarrow c$ of a rewrite rule in $\mathcal{R}$.

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

In LCNC$_d$ the outermost narrowing rule for descendants of initial equations is defined as follows: let $\mathcal{R}$ be a CTRS,

[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 head 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 t is not variable.

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

The following completeness result for LCNC was given in Ref. 4).

Theorem. LCNC is strong complete for the following classes of CTRSs:

(1) decreasing and confluent CTRS,
(2) level-confluent and terminating 2-CTRS, and
(3) shallow-confluent and terminating 3-CTRSs. □

Strong completeness means that the selection of the equation in the given goal is don't-care non-determinism. This results in a significant reduction of the search space.

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$_d$.

5. 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 Ref. 13) for a complete reference.

Although Mathematica is a suitable framework for working with rewrite systems (since its kernel is based on higher order rewriting[1]) design of basic data structures for our system is 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.

In the rest of this section, we will first describe the data structures used in our implementation. Followed by a description of the program. We also will show how our implementation extends the Mathematica built-in solvers. Finally, an illustrative example will be given followed by an experimental comparison between LCNC and LCNC$_d$ via various examples.

5.1 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.

5.2 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[_,_]`, (`TSolve[]` is the term domain solver that we introduce as an interface between the user and our system), is considered as a variable. All 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).

5.3 Constructor

A constructor symbol (abbreviated as constructor by convention) is used to construct a data structure. A constructor symbol is also declared as `FunctionSymbol` at the same time of declaration.

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.

5.4 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 5.5 for the definition of `RewriteRule[___]`).

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

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

5.5 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 $f[x] \to t \Leftarrow x \approx s, c$, where $x \approx s$ 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` $\Leftarrow$ $x \approx$ `s,c]` or `Function[{s}, t` $\Leftarrow$ `c]` if `s` is a variable.

`RewriteRule[lhs, rhs, condition]` is used to define a rewrite rule `lhs` $\to$ `rhs` $\Leftarrow$ `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.

5.6 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 4. For example the outermost narrowing inference rule can be implemented by the following Mathematica rewrite rules:

```
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]]
```

where `VariantsRule[t]` gives a fresh variant of an applicable rewrite rule, and `NewLcnc[_]` is defined as follows.

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

Here `RewritesTo[t_,c___]` defines the right-hand side `t` and conditions `c` of the applicable rewrite rule. While `Substitution` is a global variable used to hold the set of variables binding.

Although LCNC$_d$ is considered as a deterministic calculus, the selection of the rewrite rule (in case of outermost narrowing) remains a source of non-determinism. In our program we try to solve this non-deterministic behavior of LCNC$_d$ by applying all applicable rewrite rules.

An operation of the program is as follows.

For each inference rule in LCNC$_d$ there is a corresponding Mathematica program. Depending on the structure of the left-most equation in the goal, the program will apply the suitable inference rule to get a new goal G. If G is empty this means that a solution is found by the program and this solution will be added to the global variable `AnswerList` which holds all possible solutions of the given goal. If the goal is not empty and no inference rule is applicable to the leftmost equation then no solution of this goal. At this point the program will try the next applicable rewrite rule. If no applicable rewrite rules, the program terminates and return the `AnswerList` as the set of all possible solutions of the given goal.

In our 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.

5.7 Extension of Mathematica Solvers

The following simple example shows how our system extends the Mathematica equational solvers.

Consider the one-rule term rewriting system $succ[0] \longrightarrow 1$. To solve the simple equation $succ[x] \approx 1$, which has the solution $\{x \mapsto 0\}$, by using the Mathematica solver `Solve[succ[x]==1, {x}]` we get the answer $x = succ^{-1}$`[1]` which is not the desirable answer. Using our solver `TSolve[succ[x]≈ 1, {x}]` we get the correct answer `{{x → 0 }}`.

This simple example shows that our solver (based on narrowing) extends the Mathematica equational solvers (based on rewriting).

5.8 Example

Assume a car dealer wants 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 CTRS

$$\mathcal{R} = \left\{ \begin{array}{ll}
cc(model, White) & \rightarrow A \Leftarrow \\
le(model, s(s(0))) \approx True, & \\
cc(model, White) & \rightarrow B \Leftarrow \\
le(model, s(s(0))) \approx False, & \\
le(model, s(s(s(s(0))))) \approx True, & \\
cc(model, White) & \rightarrow C \Leftarrow \\
le(model, s(s(s(s(0))))) \approx False & \\
cc(model, Red) & \rightarrow A \\
cc(model, Blue) & \rightarrow B \\
cc(model, Green) & \rightarrow B \Leftarrow \\
le(model, s(0)) \approx True, & \\
cc(model, Green) & \rightarrow C \Leftarrow \\
le(model, s(0)) \approx False & \\
& \\
le(s(x), 0) & \rightarrow False, \\
le(0, x) & \rightarrow True, \\
le(s(x), s(y)) & \rightarrow le(x, y)
\end{array} \right.$$

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 $cc(0, White) \approx category$, $cc(model, color) \approx A$, and $cc(model, Green) \approx category$ respectively.

A more general question like what is the model and category of all cars? can be represented by the goal $cc(model, color) \approx category$ Our system can work on such CTRSs and can answer the above questions as follows.

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

delivers the answer:

```
{ {category → A } }.
```

of the first question.

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

gives the answer:

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

of the second question, and so on for the other questions.

5.9 LCNC vs. LCNC$_d$: A Practical Comparison

As a practical comparison between LCNC and LCNC$_d$, using the same data structures defined in the former sections, we implemented LCNC (the nondeterministic calculus).

We experiment the two implementations for various examples. The following table shows an experimental comparison between the two calculi. In the following table the execution time of LCNC and LCNC$_d$ is measured in seconds.

Example no.	LCNC	LCNC$_d$	ratio
1	0.11	0.05	2.20
2	0.13	0.06	2.16
3	0.38	0.16	2.37
4	0.44	0.22	2.00
5	0.46	0.30	1.53
6	0.49	0.31	1.58
7	0.87	0.49	1.77
8	1.15	0.50	2.30
9	1.30	0.55	2.36
10	1.50	0.63	2.38

The ten examples in the above table are given in increasing complexity. From the table we can easily see that the LCNC$_d$ implementation has a significant efficiency compared with the LCNC implementation. More precisely, LCNC$_d$ implementation, in the average, takes less than half the time needed by LCNC implementation to solve the same goal.

This comparison reveals the importance of the deterministic version of LCNC.

6. 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-program.nb.

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

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 incorporate types in the implemented system and incorporate Mathematica built-in rewriting such as arithmetic simplification and numeric relations.

In our previous work[3] we described a full implementation of 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 supports and suggestions from Bruno Buchberger and Aart Middeldorp. The first author is supported by the Ministry of Education, Science, Sports and Culture of Japan. This work is partially supported by the Okawa Foundation for Information and Telecommunications and Grants-in-aid Basic Research B07558152 and B0858059, Ministry of Education, Science, Sports and Culture of Japan.

References

1) Buchberger, B.: Mathematica as a Rewrite Language, *Proc. Fuji International Workshop on Functional and Logic Programming, Shonan Village Center, World Scientific, Singapore*, pp.1–13 (1997).

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

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

4) Hamada, M. and Middeldorp, A.: Strong Completeness of a Lazy Conditional Narrowing Calculus, *Proc. Fuji International Workshop on Functional and Logic Programming, Shonan Village Center, World Scientific, Singapore*, pp.14–32 (1997).

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

6) Ida, T. and Nakahara, K.: Leftmost outside-in narrowing calculi, *Journal of Functional Programming*, Vol.7, No.2 (1997).

7) Kaplan, S.: Conditional rewrite rules. *Theoretical Computer Science*, Vol.33, pp.175–193 (1984).

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

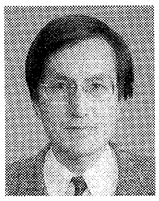
9) Middeldorp, A. and Hamoen, E.: Completeness Results for Basic Narrowing, *Applicable Algebra in Engineering, Communication and Computing*, Vol.5, pp.213–253 (1994).

10) Middeldorp, A. and Okui, S.: A Deterministic Lazy Narrowing Calculus, *Journal of Symbolic Computation* (to appear).

11) Middeldorp, A., Okui, S. and Ida, T.: Lazy Narrowing: Strong Completeness and Eager Variable Elimination, *Theoretical Computer Science*, Vol.167, Nos.1, 2, pp.95–130 (1996).
12) Prehofer, C.: Solving Higher-Order Equations: From Logic to Programming. PhD Thesis, Technische Universität München (1995). Appeared as Technical Report 19508.
13) Wolfram, S.: The Mathematica Book, WolframMedia, Champaing, IL (1996).

(Received August 14, 1997)
(Accepted December 1, 1997)

Mohamed Hamada is a Ph.D. student at the university of Tsukuba since October 1994. He received his M.Sc. in computer science and pure Mathematics from Ain Shams university, Cairo, Egypt in 1993. He worked as an assistance lecturer at Ain Shams university from 1988 to 1994. His research interest is narrowing and term rewriting.



Tetsuo Ida is a professor at the University of Tsukuba, where he leads a research group of symbolic computation (SCORE) in the institute of information sciences and electronics. His research includes integration of functional and logic programming, term rewriting and parallel and distributed symbolic computation. He received a Doctor of Science from the University of Tokyo. He is an editor of the Journal of Symbolic Computation, the Journal of Functional and Logic Programming and Texts and Monographs in Symbolic Computation. He is a member of the ACM, the IPSJ, the JSSST, the IEICE, the Association of Logic Programming and the IEEE Computer Society.