Implementation of Lazy Narrowing Calculi 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

This paper describes the implementation of narrowing calculi developed by Symbolic COmputation REsearch group (SCORE) at the University of Tsukuba. Narrowing was originally conceived as an E-unification procedure in the framework of equational theorem proving, but was rediscovered as a computation model of functional logic programming languages.

Narrowing as given in the standard literature is a complex operation. Several attempts have been made to simulate narrowing by a simple inference system defined by a small set of inference rules. Among these, are narrowing calculi called OINC (Outside-In Narrowing Calculus) of T. Ida and K. Nakahara [8], LNC (Lazy Narrowing Calculus) of A. Middeldorp, S. Okui, and T. Ida [13], and LCNC (Lazy Conditional Narrowing Calculus) of M. Hamada and A. Middeldorp [3].

In this paper we give full implementation of these three narrowing calculi using Mathematica. We demonstrate that the implementation of the narrowing calculi is feasible. This means that existentially quantified equations are solvable by our narrowing calculi with respect to given rewrite rules, and thus we extended the symbolic computation language Mathematica based on higher-order conditional rewriting.

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

Contents

1 Introduction **1**

2 Preliminaries **2**
- 2.1 Signature . . . 2
- 2.2 Variable . . . 3
- 2.3 Constructor . . . 3
- 2.4 Defined-function symbols . . . 4
- 2.5 Function Term . . . 4
- 2.6 Rewrite Rule . . . 5
  - 2.6.1 Unconditional Rewrite Rule . . . 5
  - 2.6.2 Conditional Rewrite Rule . . . 6

3 Outside-In Narrowing Calculi (OINC) **8**
- 3.1 Definition of OINC . . . 8
- 3.2 Implementation of OINC . . . 9
  - 3.2.1 Base case of induction over the equations (goal) . . . 9
  - 3.2.2 [v] Variable elimination . . . 9
  - 3.2.3 [o] Outermost narrowing . . . 10
  - 3.2.4 [d] Decomposition . . . 10
  - 3.2.5 Auxiliary Definitions . . . 10
- 3.3 Definition of OINC with strict equality (s-OINC) . . . 11
- 3.4 Implementation of s-OINC . . . 12
  - 3.4.1 [o-s] Outermost narrowing for strict equations . . . 12
  - 3.4.2 [i-s] Imitation for strict equations . . . 13
  - 3.4.3 [d-s] Decomposition for strict equations . . . 14
  - 3.4.4 [t-s] Elimination of trivial strict equations . . . 14
  - 3.4.5 Term Domain Solver . . . 14
  - 3.4.6 How to use this system . . . 15
  - 3.4.7 Example . . . 16

4 Lazy Narrowing Calculus (LNC) **16**
- 4.1 Definition of LNC . . . 17
- 4.2 Implementation of LNC . . . 18
  - 4.2.1 [v] Variable elimination . . . 19
  - 4.2.2 [d] Decomposition . . . 19

4.2.3 [o] Outermost narrowing 19
4.2.4 [i] Imitation 20
4.2.5 [t] Trivial elimination 20

5 Lazy Conditional Narrowing Calculus (LCNC) 21
5.1 Definition of LCNCd 21
5.2 Implementation of LCNCd 23
5.2.1 Base case of induction over the equations (goal) 24
5.2.2 [t] & [v] Trivial and Variable elimination 24
5.2.3 [o] Outermost narrowing 25
5.2.4 [i] & [v] Imitation and Variable elimination 25
5.2.5 [d] Decomposition 26
5.2.6 [v-p] Variable elimination for parameter passing 26
5.2.7 [o-p] Outermost narrowing for parameter passing . . . 27
5.2.8 [d-p] Decomposition for parameter passing 27
5.2.9 Auxiliary Definitions 27
5.3 Usage of the System 28
5.3.1 Term Domain Solver 28
5.3.2 Typical Session 30
5.3.3 Extensions 30
5.3.4 Example 30

6 Sequence variable of Mathematica 31

7 Conclusion 32

1 Introduction

This paper describes the implementation of narrowing calculi developed by Symbolic COmputation REsearch group (SCORE) at the University of Tsukuba. Narrowing was originally conceived as an E-unification procedure in the framework of equational theorem proving. It was rediscovered as a computation model of functional logic programming languages [4]. Narrowing has been studied intensively recently, main efforts being centered on its efficient implementation. Narrowing as given in the standard literature [11] is a complex operation. Several attempts have been made to simulate narrowing by a simple inference system defined by a small set of inference rules (e.g. [6], [5], [8], [13]). Among these, in this paper we are interested in Outside-In Narrowing Calculi (OINC for short) introduced by T. Ida and K. Nakahara [8], Lazy Narrowing Calculus (LNC for short) introduced by A. Middeldorp, S. Okui, and T. Ida [13], and Lazy Conditional Narrowing Calculus (LCNC for short) introduced by M. Hamada and A. Middeldorp [3].

In this paper we present the implementation of these calculi in Mathematica. One of the immediate objectives of our implementations was to realize quick and elegant runnable calculi to study narrowing on wide classes of examples beyond hand calculated ones. Our efforts result in calculi that are usable in variety applications such as a first-hand functional logic language interpreter as well as an equational theorem prover.

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

We assume that the reader is familiar with the terminology of rewriting and narrowing. For surveys we refer to ([9], [2], [4], and [11].) We also assume that the reader has some background in Mathematica (see [15] for a complete reference.)

The paper is organized as follows. In Section 2 we give definitions of functions common to the implementation of the three calculi. In Section 3 we describe our implementation of OINC calculi. LNCd calculus (the deterministic version of LNC) and LCNC calculus (the extended version of LNC to the conditional case) implementation well be described in Sections 4 and 5

respectively. We comment on Mathematica sequence variables in Section 6. Finally we conclude our paper by some remarks and future works. A full version of the Mathematica programs for OINC and LCNC calculi can be found in the Appendices A and B respectively.

2 Preliminaries

In this section we give some basic definitions which will be used as a common part in the implementation of OINC, LNCd, and LCNCd described in the following sections. In the sequel we will use basic data structures of Mathematica as much as possible.

2.1 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-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.

2.2 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 3.4.5 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

$$\mathrm{Is}\,W[x] \;=\; \begin{cases} \texttt{True} & \text{if } x \in W, \\ \text{non–}\texttt{True}\ \mathit{expression} & \text{if } x \notin W. \end{cases}$$

2.3 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.

2.4 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 2.6 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})
```

2.5 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.

2.6 Rewrite Rule

We adopt our own representation for both unconditional and 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 design internal data structures for rewrite rules of our systems.

We use different date structures for unconditional and conditional rewrite rules, although the representation for unconditional ones can be made a special case of the conditional ones.

2.6.1 Unconditional Rewrite Rule

Unconditional rewrite rules are represented as follows. For each defined symbol `f` we maintain a list of rewrite rules whose left-hand side is a term of the form `f[___]`. Namely we define,

`RewriteRuleList[f] :=` $\{$`NewRule`$[v_1, s_1, t_1], \cdots,$ `NewRule`$[v_k, s_k, t_k]\}$
where `NewRule`$[v_i, s_i, t_i]$ holds an expression that generates a new variant of a rewrite rule $s_i \rightarrow t_i$ when demanded, where v_i is a set of variables in s_i and the head of s_i is `f` for $i = 1, \cdots, k$. For example:

```
RewriteRuleList[f] :={NewRule[{x}, f[x,0], x],
                      NewRule[{x, y}, f[y, c[x]], x]}.
```

We do not use a pattern in our representation of rewrite rules since in narrowing, variables occurring originally in the right-hand side of a rewrite rule may get instantiated before corresponding pattern variables get instantiated. In this situation there is not much point in distinguishing variables in the left-hand sides and in the right-hand sides. In Mathematica, sequence variable plays a dual role; marking the symbol as a variable and denoting a sequence of variables. The notion of sequence variable is very important, and we will consider its incorporation in our future implementation (see Section 7 for more discussion about sequence variables).

`RewriteRule[lhs, rhs]` is used to define a rewrite rule `lhs` $\rightarrow$ `rhs`, where `lhs` and `rhs` are left and right hand sides of a defining rewrite rule. `RewriteRule[_,_]` is defined as follows:

```
RewriteRule[s_/;AtomQ[s],t_]:=
           (IsDefinedFunctionSymbol[s]=True;
            IsFunctionSymbol[s]=True;
            AppendTo[RewriteRuleList[s], NewRule[{},s,t]])

RewriteRule[s_,t_]:=
           (IsDefinedFunctionSymbol[s[[0]]]=True;
            IsFunctionSymbol[s[[0]]]=True;
            AppendTo[RewriteRuleList[s[[0]]], NewRule[Var[s],s,t]])

RewriteRuleList[_]:={}

NewRule[vset:{___},s_,t_]:=Hold[Module[vset,Rewrite[s,t]]]
```

In the above definitions we use the following auxiliary function, which is used elsewhere as well. `Var[t_]` gives a list of variables in a term `t`.

```
Var[term_Symbol]:=If[IsFunctionSymbol[term],{},{},{term}]
Var[s_/;AtomQ[s]]:={}
Var[f_[term___]]:=ReplacePart[Var/@{term},Sequence,0] ∪ Var[f]
```

2.6.2 Conditional Rewrite Rule

We define an optimized representation of conditional rewrite rules. The idea is based on the observation that the conditional rewrite rule $f[s] \rightarrow t \Leftarrow c$ can be represented by $f[x] \rightarrow t \Leftarrow x = s, c$, 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` $\rightarrow$ `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. `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]
```

3 Outside-In Narrowing Calculi (OINC)

The Outside-In Narrowing Calculus (OINC) introduced by T. Ida and K. Nakahara [8] is a sound and complete calculus to simulate narrowing. OINC is defined for arbitrary orthogonal term rewriting systems. OINC is based on the notion of the leftmost outside-in reduction of Huet and Lévy [7] for orthogonal term rewriting systems.

3.1 Definition of OINC

OINC calculus consists of the following inference rules.

[o]*outermost narrowing*

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

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

[d]*decomposition*

$$\frac{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E}{s_1 = t_1, \ldots, s_n = t_n, E},$$

[v]*variable elimination*

-[v1]

$$\frac{t = x, E}{E\theta}$$

-[v2]

$$\frac{t = x, E}{E\theta}$$

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

3.2 Implementation of OINC

The following OINC program is a direct implementation of the formal definition of OINC given in the previous section. The OINC program operates on a list of equations (goal) and computes answer substitutions restricted on the set of variables that occur in the initial goal. In the program we use `eq[s,t]` to represent an equation `s=t`.

3.2.1 Base case of induction over the equations (goal)

In the base case of induction over the goal we will consider the empty goal $\{\}$. The OINC program for the empty goal is defined as follows.

```
OINC[{}]:=AnswerList=AnswerList ∪ {ComputedAnswer[]}

ComputedAnswer[]:=
     MapThread[#2 → #1&,
               {FixedPoint[#1 /.  Substitution&,GoalVar],GoalVar}]
```

If the goal is not empty, we proceed by applying one of the following rules (programs).

3.2.2 [v] Variable elimination

The following OINC program that implements the variable elimination rule is applied to a goal whose leftmost equation is of the form $s = x$ (rule [v1]) or $x = s$ (rule [v2]) where x is variable.

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

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

3.2.3 [o] Outermost narrowing

The OINC program for the outermost narrowing inference rule is defined as follows.

```
OINC[{eq[s_ /; IsFunctionTerm[s], t_], eqns___}] :=
    Scan[Block[{Substitution=Substitution}, OINC[#1]]&,
         ( NewEqnsByOn[#1, s, t, eqns]&) /@
         NewRewriteRuleList[s[[0]]]]
```

We can intercept the failures of solving by the following definition:

```
OINC[eq[s_/; IsFunctionTerm[s] , t_], eqns___] :=
    Scan[Compose[Catch, OINC],
         Map[NewEqnsByOn[#, s, t, eqns]&,
         NewRewriteRuleList[s[[0]]]]]
```

where we insert the code of `Catch`. At the failure side `Throw` is executed. Note also that we make use of the dynamic scope of the variable `Substitution`.

3.2.4 [d] Decomposition

This should be placed after [o] since we do not check a constructor term. Note also that we have to treat atoms differently since `atom[[0]]` gives only the type of `atom`.

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

OINC[{eq[s_,t_] /; s[[0]]=== t[[0]], eqns___}]:=
   OINC[Join[NewEqns[s,t],{eqns}]]
```

3.2.5 Auxiliary Definitions

`NewEqns[s, t]` generates a new set of equations by equating first level subterms of the same position in terms `s` and `t`. `NewStEqns` generates a new set of strict equations.

```
NewEqns[s_,t_]:= Table[eq[s[[i]], t[[i]]], {i, 1, Length[s]}]
NewStEqns[s_,t_]:= Table[steq[s[[i]], t[[i]]], {i, 1, Length[s]}]
```

`NewEqnsByOn, NewEqnsByOns` are also a generator of a sequence of equations. It is used in [o] and [o-s] respectively.

```
NewEqnsByOn[Rewrite[s_, t_], lhs_, rhs_, eqns___]:=
   Join[NewEqns[lhs, s], {eq[t, rhs], eqns}]
NewEqnsByOns[Rewrite[s_, t_], lhs_, rhs_, eqns___]:=
   Join[NewEqns[lhs, s], {steq[t, rhs], eqns}]
```

`NewRewriteRuleList[f_]` generates a list of new variants of rewrite rules associated with function symbol `f`.

```
NewRewriteRuleList[f_]:=ReleaseHold /@ RewriteRuleList[f]
```

3.3 Definition of OINC with strict equality (s-OINC)

OINC with strict equality (s-OINC) is an extension of OINC which makes it possible to use OINC as a computational model of functional logic programming languages. s-OINC consists of the following set of inference rules.

[o-s]*outermost narrowing for strict equations*

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

and

$$\frac{s \equiv f(t_1,\ldots,t_n), E}{t_1 \equiv l_1,\ldots,t_n \equiv l_n, r \equiv t, E}$$

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

[d-s]*decomposition for strict equations*

$$\frac{c(s_1,\ldots,s_n) \equiv c(t_1,\ldots,t_n), E}{s_1 \equiv t_1,\ldots,s_n \equiv t_n, E},$$

where c is a constructor,

[i-s]*imitation for strict equations*

$$\frac{c(s_1,\ldots,s_n) \equiv y, E}{(s_1 \equiv y_1,\ldots,s_n \equiv y_n)\theta}$$

and

$$\frac{y \equiv f(t_1,\ldots,t_n), E}{(y_1 \equiv t_1,\ldots,y_n \equiv t_n)\theta}$$

where c is a constructor and $\theta = \{y \mapsto c(y_1,\ldots,y_n)\}$,

[t-s]*elimination of trivial strict equations*

$$\frac{x \equiv y, E}{E\theta}$$

where $\theta = \{x \mapsto y\}$ if $x \not\equiv y$ and $\emptyset$ otherwise.

3.4 Implementation of s-OINC

The base case of induction over equations is exactly the same as in Section 3.2.1. If the goal is not empty, we proceed by applying one of the following rules (programs). To distinguish between OINC equations and s-OINC equations we use the form `steq[s,t]` for the later.

3.4.1 [o-s] Outermost narrowing for strict equations

Here we have two variants of the outermost narrowing inference rule. Hence two OINC programs, where right hand side and left hand side are swapped, are necessary.

```
OINC[{steq[s_/;IsFunctionTerm[s],t_],eqns___}]:=
    Scan[Block[{Substitution=Substitution},OINC[#1]]&,
         (NewEqnsByOns[#1,s,t,eqns]&)/@
         NewRewriteRuleList[s[[0]]]]
```

```
OINC[{steq[s_,t_/;IsFunctionTerm[t]],eqns___}]:=
    Scan[Block[{Substitution=Substitution},OINC[#1]]&,
         (NewEqnsByOns[#1,t,s,eqns]&)/@
         NewRewriteRuleList[t[[0]]]]
```

3.4.2 [i-s] Imitation for strict equations

Although we have two variants of the imitation rule, we write four OINC programs for this rule. In both variants we distinguish the case whether the term `s` is an atom. If `s` is an atom then we simply write a program similar to the program of the variable elimination defined in Section 3.2.2. The OINC program for imitation is defined as follows.

```
OINC[{steq[s_/;AtomQ[s],x_/;IsVar[x]],eqns___}]:=
    With[{mapsto=x→s},
         PrependTo[Substitution,mapsto];
         OINC[{eqns}/.mapsto]]

OINC[{steq[s_,x_/;IsVar[x]],eqns___}]:=
    Module[{cs,mapsto},
           cs=CSkeleton[s[[0]],Length[s]];
           mapsto=x→cs;
           PrependTo[Substitution,mapsto];
           OINC[Join[NewStEqns[s,cs],{eqns}]/.mapsto]]

OINC[{steq[x_/;IsVar[x],s_/;AtomQ[s]],eqns___}]:=
    With[{mapsto=x→s},
         PrependTo[Substitution,mapsto];
         OINC[{eqns}/.mapsto]]

OINC[{steq[x_/;IsVar[x],s_],eqns___}]:=
    Module[{cs,mapsto},
           cs=CSkeleton[s[[0]],Length[s]];
           mapsto=x→cs;
           PrependTo[Substitution,mapsto];
           OINC[Join[NewStEqns[s,cs],{eqns}]/.mapsto]]
```

We use auxiliary function `CSkeleton[c_, n_]` which creates a constructor term skeleton of the form `c[`$x_1, \cdots, x_n$`]`.

```
CSkeleton[c_,n_]:=c[Array[Unique[]&,n]/.List→Sequence]
```

3.4.3 [d-s] Decomposition for strict equations

The implementation of OINC decomposition program for strict equations is defined as follows.

```
OINC[{steq[s_,t_],eqns___}]/;AtomQ[s]:=
   If[s===t,OINC[{eqns}],False]

OINC[{steq[s_,t_]/; s[[0]]=== t[[0]],eqns___}]:=
   OINC[Join[NewStEqns[s,t],{eqns}]]
```

In this program we use a function `NewStEqns[]` which is variant of the function `NewEqns[]` (that was defined in Section 3.2.5) for strict equations.

3.4.4 [t-s] Elimination of trivial strict equations

The OINC program for the elimination of trivial strict equations is defined as follows.

```
OINC[{steq[x_/;IsVar[x],y_/;IsVar[y]],eqns___}]:=
    Module[{mapsto},
           If[x===y,
              OINC[{eqns}],
              mapsto=x→y;
              PrependTo[Substitution,mapsto];
              OINC[{eqns}/.mapsto]]]
```

3.4.5 Term Domain Solver

`Tsolve` is a term domain solver with the same interface as other solvers of Mathematica like `DSolve, TSolve` and `NSolve. Tsolve` is defined as follows.

```
TSolve[goal_List,varlist_List]:=
      Block[{AnswerList={},GoalVar=varlist},
            Block[{Substitution={}},OINC[goal]];
            AnswerList]

TSolve[eqn_,varlist_List]:=
      Block[{AnswerList={},GoalVar=varlist},
            Block[{Substitution={}},OINC[{eqn}]];
            AnswerList]
```

3.4.6 How to use this system

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]
.
.
RewriteRule[sn,tn]
TSolve[goal,varset]
```

The `BeginRule[]` program is defined as follows.

```
BeginRule[]:=
     (Clear[RewriteRuleList];
      RewriteRuleList[_]:={};
```

```
      Clear[IsConstructor,IsDefinedFunctionSymbol,IsFunctionSymbol];
      DefineSignature[];)

On[General::''spell1'']
```

Note that the declaration of `DefinedFunctionSymbol[__]` is unnecessary in the case of constructor systems.

3.4.7 Example

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

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

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 Lazy Narrowing Calculus (LNC)

The Lazy Narrowing Calculus (LNC) was introduced in A. Middeldorp, S. Okui, and T. Ida [13] as a specialization of Hölldobler [6] TRANS calculus

to confluent TRSs. LNC is a complete calculus for several classes of TRSs, e.g. the class of confluent (not necessarily terminating) TRSs with respect to normalized solution.

We implement this non-deterministic version of LNC to show that the implementation of non-deterministic narrowing calculi is possible.

Such (non-deterministic) implementation is important if we deal with non constructor-based rewriting systems.

4.1 Definition of LNC

LNC 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 $\mathcal{R}$,

[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 $t \approx s$.

4.2 Implementation of LNC

The above version of LNC has three types of non-determinism:

1. the choice of the equation in the goal,
2. the choice of the inference rule to be applied, and
3. the choice of the rewrite rule (in case of the application of the outer most narrowing rule.)

These types of non-determinism makes it difficult to implement this calculus. The following complicated program shows how we can cope with the last two types of non-determinism.

Let `@` denote `Lnc` that applies all the inference rules of LNC. The result of `Lnc` is a list $\{h_1, \cdots, h_k\}$, where h_i is either a goal or a list of goals. The list of goals are generated when [o] is applied to a goal. Let `@@` denote application of `@` to each h_i.

`@`$\{\}$ $\rightarrow$ add solution, i.e. success of solving of a goal
`@g` $\rightarrow$ `@@` $\{h_1, \cdots, h_k\}$,
`@`$\{g_1, \cdots, g_k\} \rightarrow$ `@`$g_1; \cdots;$ `@`$g_k, k > 0$
to cope with rewrite rule non-determinism
`@@`$\{\}$ $\rightarrow$ `false`, i.e., failure of solving of a goal
`@@`$\{h_1, \cdots, h_k\} \rightarrow$ `@`$h_1; \cdots;$ `@`$h_k, k > 0$
to cope with inference rule non-determinism.

The following `Lnc` program is a bit complicated.

```
Clear[Lnc];
Lnc[Obj[subst_,{}]]:=AnswerList=AnswerList ∪ {Answer[subst]}
Lnc[x:Obj[subst_,goal_]]:=LncToAll[ReplaceList[x,InfRule]]
Lnc[{objs__}]:=Scan[Lnc,{objs}]
LncToAll[{}]:=False
LncToAll[{objs__}]:=Scan[Lnc,{objs}]

Answer[subst_]:=
```

```
    MapThread[#2 → #1&, {FixedPoint[#1/.subst&,GoalVar],GoalVar}]

  VarQ[x_]:=If[IsVar[x],True,False,False]

  FunctionTermQ[x_]:=If[IsFunctionTerm[x],True,False,False]
```

4.2.1 [v] Variable elimination

The variable elimination rule is implemented as follows.

```
Velim1=Obj[{maps___},{eq[s_,x_/;VarQ[x]],eqns___}]/;
          !(MemberQ[Var[s],x]) :→
                   With[{mapsto=x → s},
                        Obj[{mapsto,maps},{eqns}/.mapsto]];

Velim2=Obj[{maps___},{eq[x_/;VarQ[x],s_],eqns___}]/;
          !(MemberQ[Var[s],x]):→
                   With[{mapsto=x → s},
                        Obj[{mapsto,maps},{eqns}/.mapsto]];
```

4.2.2 [d] Decomposition

The decomposition rule is implemented as follows.

```
Decomp=Obj[subst_,{eq[s_,t_],eqns___}]/;
          !(VarQ[s]):→ If[AtomQ[s],
                          If[s===t,Obj[subst,{eqns}],False],
                          If[s[[0]]=== t[[0]],
                             Obj[subst,Join[NewEqns[s,t],{eqns}]],
                             False]];
```

4.2.3 [o] Outermost narrowing

The outermost narrowing rule is implemented as follows.

```
On1=Obj[subst_,{eq[s_/;FunctionTermQ[s],t_],eqns___}] :→
    (Obj[subst,NewEqnsByOn[#1,s,t,eqns]]&)/@
    NewRewriteRuleList[s[[0]]];
```

```
On2=Obj[subst_,{eq[s_,t_/;FunctionTermQ[t]],eqns___}] :→
    (Obj[subst,NewEqnsByOn[#1,t,s,eqns]]&)/@
    NewRewriteRuleList[t[[0]]];
```

4.2.4 [i] Imitation

The imitation rule is implemented as follows.

```
Imi1=Obj[{maps___}, {eq[s_/; !(VarQ[s]),x_/;VarQ[x]],eqns___}] :→
     If[AtomQ[s],
        With[{mapsto=x → s},
             Obj[{mapsto,maps},{eqns}/.mapsto]],
             Module[{cs,mapsto},
                    cs=CSkeleton[s[[0]],Length[s]];
                    mapsto=x → cs;
                    Obj[{mapsto,maps},
                        Join[NewEqns[s,cs],{eqns}]/.mapsto]]];

Imi2=Obj[{maps___}, {eq[x_/;VarQ[x],s_/; !(VarQ[s])],eqns___}] :→
     If[AtomQ[s],
        With[{mapsto=x → s},
             Obj[{mapsto,maps},{eqns}/.mapsto]],
             Module[{cs,mapsto},
                    cs=CSkeleton[s[[0]],Length[s]];
                    mapsto=x → cs;
                    Obj[{mapsto,maps},
                        Join[NewEqns[s,cs],{eqns}]/.mapsto]]];
```

4.2.5 [t] Trivial elimination

The trivial elimination rule is implemented as follows.

```
Triv=Obj[subst_,{eq[x_/;VarQ[x],y_/;VarQ[y]],eqns___}]/;
         x===y :→ Obj[subst,{eqns}];

InfRule={Velim1,Velim2,On1,On2,Imi1,Imi2,Decomp,Triv};
```

```
ReplaceList[expr_,{r_,rules___}]:=
            With[{result=Replace[expr,r]},
                 If[expr===result,
                    ReplaceList[expr,{rules}],
                    Prepend[ReplaceList[expr,{rules}],result]]]

ReplaceList[expr_,{}]:={}
```

5 Lazy Conditional Narrowing Calculus (LCNC)

LCNC was introduced by M. Hamada and A. Middeldorp [3] as a proper extension of LNC to deal with conditional term rewriting systems CTRSs. In [3] they show the (strong) completeness of LCNC for various classes of CTRSs.

It is worth mentioning that the only difference between LNC and LCNC is addition of the conditional part of the applied rewrite rule in case of the outer most narrowing rule. This means that LCNC has the same sources of non-determinism as LNC which makes its implementation difficult. The implementation of LCNC is very similar to the implementation of LNC shown in Section 4.

In A. Middeldorp and S. Okui [12] a deterministic version of LNC namely LNCd was given. LNCd results in a significant reduction of search space and ease the implementation. Due to the similarity between LNC and LCNC we can define a deterministic version of LCNC namely LCNCd by adding the conditional part of the applied rewrite rule to the outer most narrowing rule. In the following two (sub)sections we will give the definition and the implementation of LCNCd.

5.1 Definition of LCNCd

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

[o] *outermost narrowing*

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

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.

[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, $x \in \mathcal{V}\mathrm{ar}(f(s_1,\ldots,s_n))$ 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 LCNCd deals with descendants of parameter passing equations:

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

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

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,

[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 $\theta = \{x \mapsto s\}$,

5.2 Implementation of LCNCd

This section describes the implementation of LCNCd. Our Mathematica programs are straightforward translation of the formal definition of LCNCd given in the previous section. We will use the notation `ppeq[ , ]` instead of `eq[ , ]` to represent the parameter passing equations.

5.2.1 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,
              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.

5.2.2 [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]]]
```

5.2.3 [o] Outermost narrowing

To deal with both sides of an equation `eq[s, t]`, two variants of `Lcnc` program for [o] are necessary (depends 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]]
```

5.2.4 [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, a 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]]
```

5.2.5 [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}]]
```

5.2.6 [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]]
```

5.2.7 [o-p] Outermost narrowing for parameter passing

Since LCNCd 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]]
```

5.2.8 [d-p] Decomposition for parameter passing

A similar discussion as in Section 5.2.5 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}]]
```

5.2.9 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}
```

5.3 Usage of the System

5.3.1 Term Domain Solver

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

`TSolve[]` for LCNCd calculus is different from `TSolve[]` of OINC calculus described in Section 3.4.5, not only because the two calculi are different, but also because we add some (optional) parameters to the parameter list of `TSolve[]` of LCNCd.

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]
```

5.3.2 Typical Session

As described in Section 3.4.6 we can easily use the LCNCd system by the same way, 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.)

5.3.3 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})
```

5.3.4 Example

Given the following conditional term rewriting system $\mathcal{R}$,

$$\mathcal{R} \;=\; \begin{cases} f(x,1) & \to x \Leftarrow x = 2, \\ f(y,2) & \to 1 \end{cases}$$

we can solve the goal $f(x,y) = 1$ using our system as follows.

```
BeginRule[]
DefinedFunctionSymbol[f]
Constructor[]
RewriteRule[f[x, 1], x, eq[x,2]];
RewriteRule[f[y, 2], 1];
TSolve[eq[f[x, y], 1], x, y]
```

Note that in this case we can delete the `Constructor[]` line because, in this example, we don't have any constructors.

In the above definition both `f` and the two rewrite rules are considered as global declarations. If we want to localize them we can use the following (equivalent) definition.

```
TSolve[eq[f[x, y], 1], x, y,
       DefinedFunctionSymbol → {f}
       Rule :→ {RewriteRule[f[x, 1], x, eq[x,2]],
                RewriteRule[f[y, 2], 1]}]
```

In both cases the system will then give the following set of all (possible) solutions.

```
{{y→ 2}}
```

6 Sequence variable of Mathematica

In this separate section we comment on a sequence variable of Mathematica. The notion of sequence variable is unique in Mathematica, and it enhances clarity of programs if properly utilized. A sequence variable alleviates `cons` operator (in the sense of Lisp and other functional languages) for inductively defined functions (inductive on list structures).

In our view an inductively defined program using sequence variables retains a flavor of mathematical induction (i.e. induction on natural numbers) rather than of structural induction. This will help a novice programmer to avoid falling into a pitfall of confusion of a list and elements of the list in reasoning about inductive defined programs using `cons`.

With sequence variables we rely on the intuition of enumeration that we use in daily life. Thus we might say that sequence variables play a role in

between induction variables of structural induction and an index of enumeration.

One point that we should be careful about the use of sequence variables is that a sequence variable holds a value `Sequence[___]`, which is not an ordinary data structure. We often need a wrapper of a sequence variable to make the sequence a first-class data structure. Let us take an example. Let $\mathbf{v}$ be a sequence $v_1, \cdots, v_k$ and we want to apply substitution θ to the sequence, to obtain $v_1\theta, \cdots, v_k\theta$. Thus we would write an expression $\mathbf{v}/.\theta$. However, this would not work since this expression is a function evaluation `ReplaceAll[`$\mathbf{v}, \theta$`]` which rewrites to `ReplaceAll[`$v_1, v_2, \cdots, v_k, \theta$ `]`. Here the system will confuse the second argument v_2 with the substitution to apply. In case of $k = 0$, we will have an error message *insufficient number of parameters*. To avoid this error, we need some constructor to wrap the sequence, e.g. $\{\mathbf{v}\}/.\theta$. However, suppose further that we want to feed the sequence $v_1\theta, \cdots, v_k\theta$ to another function, say `g`, as arguments. `g`$[\{\mathbf{v}\}/.\theta]$ would not be what we want. Then we have to write `g[ReplacePart[`$\{\mathbf{v}\}/.\theta$ `,Sequence,0]]`, which is a rather awkward expression. In this case the problem could have been avoided if `ReplaceAll` were defined as `ReplaceAll[`$\theta, v_1, \cdots, v_k$`]`. Then it would rewrite to $v_1\theta, \cdots, v_k\theta$.

Whether to restrict or extend the use of `Sequence` and a sequence variable, we need clearer semantics of a sequence variable.

7 Conclusion

In this paper we have presented the implementation of the narrowing calculi OINC, LNC (LNCd), and LCNCd in Mathematica.

Our efforts were centered more on clarity and coherence with Mathematica style of programming, and less on the efficiency of the program. However, we claim that conceptual clarity of our program in fact enhances the efficiency in many parts of our programs. Absolute execution efficiency of our program depends largely on the efficiency of pattern matching and rewriting of the Mathematica system. We hope that the implemented calculi serve as functional logic programming languages interpreters, allowing researchers in this field write functional logic programs combined with powerful Mathematica rewrite rules.

We leave the following extension to the present work as our future research

themes:

- incorporation of Mathematica built-in rewriting into our calculi, especially arithmetic simplification and incorporation of inequality, and other numeric relations, and
- development of tools by which proof processes (by the narrowing calculi) (i.e. refutations) are visualized graphically and in natural language style.

Acknowledgments

We are grateful for various supports and suggestions from Bruno Buchberger and Aart Middeldorp. 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, (to appear 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 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, (to appear 1997).

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

[5] M. Hanus, *Lazy Unification with Simplification*, Lecture Notes in Computer Science **788**, pp. 272-286, 1994.

[6] S. Hölldobler, *Foundations of Equational Logic Programming*, Lecture Notes in Artificial Intelligence **353**, 1989.

[7] Huet and Lévy, *Computations in orthogonal rewriting systems I*, Lassez, J.-L., and Plotkin, G. (eds), *Computational logic: essays in honor of Alan Robinson*. PP. **395-414** The MIT press, 1991.

[8] T. Ida and K. Nakahara, *Leftmost Outside-In Narrowing Calculi*, J. Functional Programming (to appear 1997).

[9] 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.

[10] J.W. Lloyd, *Foundations of Logic Programming*, Springer, second, extended edition, 1987.

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

[12] 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.

[13] 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.

[14] C. Prehopher *Solving Higher-Order Equations: From Logic to Programming*, Ph.D Thesis, 1995.

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

Appendix A

In this appendix we are going to give a full version of the Mathematica program that impelements the OINC calculi.

Defining a Signature

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

Variable

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

Function Term

```
   IsFunctionTerm[t_/;IsDefinedFunctionSymbol[t[[0]]]]:=True;
```

Constructor

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

Defined-function Symbol

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

Defining a RewriteRule

```
RewriteRule[s_/;AtomQ[s],t_]:=
           (IsDefinedFunctionSymbol[s]=True;
            IsFunctionSymbol[s]=True;
            AppendTo[RewriteRuleList[s], NewRule[{},s,t]])

RewriteRule[s_,t_]:=
           (IsDefinedFunctionSymbol[s[[0]]]=True;
            IsFunctionSymbol[s[[0]]]=True;
            AppendTo[RewriteRuleList[s[[0]]], NewRule[Var[s],s,t]])

RewriteRuleList[_]:={}

NewRule[vset:{___},s_,t_]:=Hold[Module[vset,Rewrite[s,t]]]

Var[term_]:=Flatten[GetVar[term]]

GetVar[term_Symbol]:=If[IsFunctionSymbol[term],{},{},{term}]
GetVar[s_/;AtomQ[s]]:={}
GetVar[f_[term___]]:=Prepend[GetVar/@{term},GetVar[f]]
```

Term Domain Solver

```
TSolve[goal_List,varlist_List]:=
      Block[{AnswerList={},GoalVar=varlist},
            Block[{Substitution={}},OINC[goal]];
            AnswerList]

TSolve[eqn_,varlist_List]:=
      Block[{AnswerList={},GoalVar=varlist},
            Block[{Substitution={}},OINC[{eqn}]];
            AnswerList]
```

Definition of OINC

Base case of induction over the equations (goal)

```
OINC[{}]:=AnswerList=AnswerList ∪ {ComputedAnswer[]}
```

```
ComputedAnswer[]:=
     MapThread[#2 → #1&,
               {FixedPoint[#1 /.  Substitution&,GoalVar],GoalVar}]
```

[v] Variable elimination rule

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

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

[o] Outermost narrowing

```
OINC[{eq[s_ /; IsFunctionTerm[s], t_], eqns___}] :=
    Scan[Block[{Substitution=Substitution}, OINC[#1]]&,
         ( NewEqnsByOn[#1, s, t, eqns]&) /@
         NewRewriteRuleList[s[[0]]]]

OINC[eq[s_/; IsFunctionTerm[s] , t_], eqns___] :=
    Scan[Compose[Catch, OINC],
         Map[NewEqnsByOn[#, s, t, eqns]&,
         NewRewriteRuleList[s[[0]]]]]
```

[d] Decomposition

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

OINC[{eq[s_,t_] /; s[[0]]=== t[[0]], eqns___}]:=
   OINC[Join[NewEqns[s,t],{eqns}]]
```

Auxiliary Definitions

```
NewEqns[s_,t_]:= Table[eq[s[[i]], t[[i]]], {i, 1, Length[s]}]
NewStEqns[s_,t_]:= Table[steq[s[[i]], t[[i]]], {i, 1, Length[s]}]

NewEqnsByOn[Rewrite[s_, t_], lhs_, rhs_, eqns___]:=
   Join[NewEqns[lhs, s], {eq[t, rhs], eqns}]
NewEqnsByOns[Rewrite[s_, t_], lhs_, rhs_, eqns___]:=
   Join[NewEqns[lhs, s], {steq[t, rhs], eqns}]

NewRewriteRuleList[f_]:=ReleaseHold /@ RewriteRuleList[f]
```

Inference Rules for Strict Equations

[o-s] Outermost narrowing for strict equations

```
OINC[{steq[s_/;IsFunctionTerm[s],t_],eqns___}]:=
    Scan[Block[{Substitution=Substitution},OINC[#1]]&,
         (NewEqnsByOns[#1,s,t,eqns]&)/@
         NewRewriteRuleList[s[[0]]]]

OINC[{steq[s_,t_/;IsFunctionTerm[t]],eqns___}]:=
    Scan[Block[{Substitution=Substitution},OINC[#1]]&,
         (NewEqnsByOns[#1,t,s,eqns]&)/@
         NewRewriteRuleList[t[[0]]]]
```

[i-s] Imitation

```
OINC[{steq[s_/;AtomQ[s],x_/;IsVar[x]],eqns___}]:=
    With[{mapsto=x→s},
         PrependTo[Substitution,mapsto];
         OINC[{eqns}/.mapsto]]

OINC[{steq[s_,x_/;IsVar[x]],eqns___}]:=
    Module[{cs,mapsto},
           cs=CSkeleton[s[[0]],Length[s]];
           mapsto=x→cs;
           PrependTo[Substitution,mapsto];
           OINC[Join[NewStEqns[s,cs],{eqns}]/.mapsto]]
```

```
OINC[{steq[x_/;IsVar[x],s_/;AtomQ[s]],eqns___}]:=
    With[{mapsto=x→s},
         PrependTo[Substitution,mapsto];
         OINC[{eqns}/.mapsto]]

OINC[{steq[x_/;IsVar[x],s_],eqns___}]:=
    Module[{cs,mapsto},
           cs=CSkeleton[s[[0]],Length[s]];
           mapsto=x→cs;
           PrependTo[Substitution,mapsto];
           OINC[Join[NewStEqns[s,cs],{eqns}]/.mapsto]]

CSkeleton[c_,n_]:=c[Array[Unique[]&,n]/.List→Sequence]
```

[d-s] Decomposition for strict equations

```
OINC[{steq[s_,t_],eqns___}]/;AtomQ[s]:=
   If[s===t,OINC[{eqns}],False]

OINC[{steq[s_,t_]/; s[[0]]=== t[[0]],eqns___}]:=
   OINC[Join[NewStEqns[s,t],{eqns}]]
```

[t-s] Elimination of trivial strict equations

```
OINC[{steq[x_/;IsVar[x],y_/;IsVar[y]],eqns___}]:=
    Module[{mapsto},
           If[x===y,
              OINC[{eqns}],
              mapsto=x→y;
              PrependTo[Substitution,mapsto];
              OINC[{eqns}/.mapsto]]]
```

Miscellaneous Function Definitions

```
BeginRule[]:=
     (Clear[RewriteRuleList];
      RewriteRuleList[_]:={};
      Clear[IsConstructor,IsDefinedFunctionSymbol,IsFunctionSymbol];
      DefineSignature[];)
```

Appendix B

In this Appendix we will give the full version of the Mathematica program for the LCNCd calculus.

Definition of a Signature

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

Variable

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

Function Term

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

Constructor

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

Defined-function Symbol

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

Defining a RewriteRule

```
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]
```

Auxiliary function definitions

```
   Var[term_Symbol]:=If[IsFunctionSymbol[term],{},{},{term}]
   Var[s_/;AtomQ[s]]:={}
   Var[f_[term___]]:=ReplacePart[Var/@{term},Sequence,0] ∪ Var[f]
```

Definition of LCNC

[t] Removal of trivial equations & Variable elimination

```
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]]]
```

[o] Outermost narrowing

```
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]]
```

[i] Imitation & Variable elimination

```
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]]
```

[d] Decomposition

```
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}]]
```

[v-p] Variable elimination for parameter passing equations

```
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]]
```

[o-p] Outermost narrowing for parameter passingt equations

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

[d-p] Decomposition for parameter passing equations

```
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}]]
```

Base case of induction over the equations (goal)

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

Auxiliary Definitions

```
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}
```

Term Domain Solver

```
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]
```

Extensions

```
SetAttributes[RuleList,HoldRest];

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