The CFLP calculus: A *Mathematica* Implementation[1] (Draft 0.4)

Mircea Marin
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Mircea.Marin@risc.uni-linz.ac.at

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

April 24, 1998

Abstract

This report describes the full Mathematica implementation of the CFLP calculus resulted by extending with constraint solving capabilities the narrowing calculi LCNC and HLNC developed by **S**ymbolic **C**omputation **RE**search group (SCORE) at the University of Tsukuba.

Narrowing was originally introduced in automated theorem proving as a calculus for solving equations by computing unifiers with respect to an equational theory expressed by a confluent and terminating term rewriting system. Because of the high complexity of the narrowing inference rule, several attempts have been made to simulate it by a number of more elementary inference rules. Among these, the calculi invented by the SCORE group at the University of Tsukuba were shown to be feasible to implement, in particular the calculi OINC (Outside-In Narrowing Calculus) and LCNC (Lazy Conditional Narrowing Calculus). These calculi were fully implemented in *Mathematica* language and could be used as a research tool for studying equation solving with respect to various classes of (conditional) term rewriting systems.

CFLP is a functional logic language whose computation model is an extension of LCNC and HLNC with inference rules for handling constraints by means of cooperation with external solvers. The interpreter, scheduler and constraint solvers are separate processes of the system that run independently. For the moment, all the processes are implemented in Mathematica 3.0 and communicate via MathLink.

This report describes *work in progress* and is therefore continuously evolving. The current version of report and software can be downloaded from the project Web page

http://www.risc.uni-linz.ac.at/projects/basic/distcon/

Contents

Chapter 1

Introduction

In recent years most proposals with a sound and complete operational semantics for the integration of functional and logic programming languages were based on narrowing. Narrowing was originally introduced in automated theorem proving as a calculus for solving equations by computing unifiers with respect to an equational theory expressed by a confluent and terminating term rewriting system. Informally, narrowing proceeds by unifying a term with the left-hand side of a rewrite rule and fires the rule on the instantiated term.

Because the narrowing inference rule is quite complicated, several attempts have been made to replace it with more elementary ones. One of the succesful proposals is the lazy narrowing calculus (LNC) of A. Middledorp, S. Okui, and T. Ida [8]. LNC simulates a lazy evaluation strategy in the sense that it avoids unnecessary computations and allows us to deal with infinite data structures. Therefore the termination of the rewrite relation associated to the rewrite rule program may not be required. The lazy narrowing calculus has been extensively studied by SCORE group at the University of Tsukuba. Among the recent results, we mention

- the deterministic versions of LNC developed by A. Middledorp and S. Okui [8] by imposing strong restrictions on the rewrite rule system and on the meaning of a solution; usually, the following restrictions are imposed on the rules of functional logic languages with a lazy operational semantics to ensure completeness:

  1. *constructor-based:* the functional logic program is constructor-based.
  2. *left-linearity:* the functional logic program is left-linear, i.e., no variable appears more than once in the left-hand side of any rule.
  3. *free variables:* for any rule, the variables of the right-hand side appear in the left-hand side.

4. *non-ambiguity:* the left-hand sides of two different rules are not unifiable.

- the conditional LNC calculus (LCNC) developed by M. Hamada and A. Middledorp [10] as an extension to conditional term rewriting systems,
- the Higher-Order LNC proposed by T. Suzuki, K. Nakagawa and T. Ida as a computation model for a higher-order functional and logic programming language based on LNC.

Most of these calculi were fully implemented in *Mathematica 3.0*, a language of symbolic computation based on higher order conditional rewriting.

In this paper we describe the implementation of a functional logic programming language extended with constraints over constraint theories for which various solvers are available. Constraints are equalities of a distinguished signature that can not be interpreted in a pure functional logic framework, but by a separate constraint solving component which is responsible to carry out resolutions in the corresponding constraint theories. The calculus is called CFLP (**C**onstrained **F**unctional **L**ogic **P**rogramming) and is an extension of the narrowing calculi HLNC and CFLP.

CFLP differs from a narrowing calculus in the following respects:

- the signature of the functional logic language was extended with distinguished function symbols for expressing constraints. The calculus is designed to recognize constraints by looking at the outermost function symbol of an equation
- the goal (i.e. system of equations to be solved) to the calculus is composed of two disjoint components:
  - a set of equations to be treated by the lazy narrowing calculus,
  - a set of constraints,

CFLP is implemented as a distributed system consisting of three layers

- an interpreter that implements the functional logic component of the CFLP calculus.
- a generic constraint solver — we call it scheduler — which solves constraints by cooperation with particular constraint solvers.
- external constraint solvers.

Another feature of the system is the presence of a polymorphic type checker that verifies the type consistency of systems of equations and (conditional) rewrite rules of the term rewriting system.

The CFLP interpreter is a continuation of the implementations of the calculi LCNC and HLNC written by Tetsuo Ida. The interpreter, scheduler and constraint solvers are separate processes of the system that run independently. For the moment, all the processes are implemented in Mathematica 3.0 and communicate via MathLink.

The paper is organized as follows. Chapter 2 introduces the syntax of the CFLP interpreter and the semantics of the CFLP commands. Chapter 3 describes the type checker. In Chapter 4 we briefly review some narrowing calculi developed so far. Chapter 5 describes the CFLP calculus.

In the sequel we assume the reader has some background in using *Mathematica* (see [15] for a complete reference).

We are grateful to professor Tetsuo Ida and the members of SCORE group for their support in making us understand the narrowing calculi developed far. The Mathematica notebooks with the implementations of HLNC and CFLP provided by Tetsuo Ida were the main starting point of my activity. The idea of how to model constraints and to extend the calculus with suitable inference rules was also inspired by professor Ida.

The design of the scheduler is inspired from Hoon's work on the design of a cooperative constraint logic programming system over the domain of complex functions (see [4]).

This report describes *work in progress* and is therefore continuously evolving. The current version of report and software can be downloaded from the project Web page

http://www.risc.uni-linz.ac.at/projects/basic/distcon/

Chapter 2

The CFLP Language

In this chapter we define the syntax of the language of the CFLP interpreter. We call this language the *CFLP language*.

Before giving the concrete syntax of the language constructos, we introduce some basic definitions which will be used when describing the CFLP calculus and its implementation.

A *signature* consists of the following classes of symbols:

1. a class of *defined-function* symbols,
2. a class of *constructor* symbols,
3. a class of *external functions* symbols,
4. a class of variables.

These classes are pairwise disjoint. Symbols that are either defined functions, constructors or external functions are called *function* symbols.

The CFLP language is composed of two sublanguages:

1. the object sublanguage,
2. the language of type expressions, used in typing the function symbols of the object (sub)language.

In the next sections we will describe these two languages in terms of their underlying signatures. We also show how are they connected.

2.1 Type Expressions

The underlying signature of this language consists of

1. a class $\mathcal{F}_{type}$ of constructor symbols, consisting of
   (a) basic types: `Int` for integers, `Real` for floating-point numbers, `Bool` for boolean values, `Symbol` for symbols, and $\Box$ for the trivial type;
   (b) non-constant (type) constructor symbols: `Cross` for Cartesian product, `FuncType` for function types, and `TyList` for lists;
   (c) user defined type constructors;
2. a class $\mathcal{V}_{type}$ of type variables.

Basic types are regarded as 0-ary type constructors, and `Cross`, `FuncType` and `TyList` are type constructors of positive arity.

The context-free grammar of this sublanguage is given below.

$$\begin{array}{rcll} \texttt{Type} & ::= & f[\texttt{Type},\ldots,\texttt{Type}] & \text{if } f \in \mathcal{F}_{type} \\ \texttt{TypeVariable} & ::= & v & \text{if } v \in \mathcal{V}_{type} \end{array}$$

$\mathcal{F}_{type}$ is the set of type constructors, and it always contains the following constructor symbols: the variadic constructor `Cross` for Cartesian product (with arity ≥ 2), the unary constructor `TyList` for lists of elements of the same type, the nullary constructor $\Box$ for the trivial type, and the binary function type constructor `FuncType`. $\mathcal{F}_{type}$ can be extended by means of type constructor declarations.

Type expressions are used in connection with the CFLP object language to attach polymorphic types to function symbols. The type information attached to function symbols is used by a type checker for verifying the type consistency of expressions built in the CFLP object language.

When referring to this syntax domain we find convenient to introduce the following terminology.

Given a type expression $\tau \in \mathcal{TE}\text{xpr}[\mathcal{V}_{type}]$, we define the set of type variables of a type expression

$$\mathcal{V}_{type}[\tau] = \begin{cases} \tau & \text{if } \tau \in \mathcal{V}_{type}, \\ \mathcal{V}_{type}[\tau_1] \cup \cdots \cup \mathcal{V}_{type}[\tau_n] & \text{if } \tau \equiv \tau_1 \times \cdots \times \tau_n, \\ \mathcal{V}_{type}[\tau_1] \cup \mathcal{V}_{type}[\tau_2] & \text{if } \tau \equiv \texttt{FuncType}[\tau_1, \tau_2] \\ \mathcal{V}_{type}[\tau_1] & \text{if } \tau \equiv \texttt{TyList}[\tau_1] \end{cases}$$

A *type substitution* is a mapping $\theta : \mathcal{V}_{type} \to \mathcal{V}_{type}$ with finite support, i.e. the set $\texttt{supp}[\theta] = \{x \in \mathcal{V}_{type} \mid \theta[x] \neq x\}$ is finite. From now on we will use the notation $x\theta$ instead of $\theta[x]$ to denote the value of θ at x. Also, sometimes we write $\{x_1 \mapsto x_1\theta, \ldots, x_n \mapsto x_n\theta\}$ instead of θ, where $\{x_1, \ldots, x_n\} = \mathtt{supp}[\theta]$.

The homomorphic extension of θ to $\mathcal{TE}\mathrm{xpr}[\mathcal{V}_{type}]$ will also be denoted by θ.

2.1.1 Implementation

We distinguish type constructors from other symbols by means of the *Mathematica* classifier `IsTypeConstructor`. The function call

`IsTypeConstructor`[f]

returns `True` if f is a type constructor and `False` otherwise. This function is initialized to return `True` for the *Mathematica* symbols given in the table below, and `False` for everything else.

Mathematica symbol	Meaning
□	trivial type
`Int`	type of integers
`Real`	type of reals
`Bool`	boolean type
`Symbol`	type of symbols
`FuncType`	function type constructor
`Cross`	Cartesian product type constructor
`TyList`	list type constructor

This initialization is made via the function call

```
TypeConstructor[Int, Bool, Real, Symbol, □, TyList, FuncType, Cross];
```

where

```
TypeConstructor[c_Symbol]:=
  Block[{},
    IsTypeConstructor[c]=True;
    IsConstructor[c]=True;
    {c}
  ];
TypeConstructor[cs__Symbol]:=Apply[Join, Map[TypeConstructor,{cs}]];
```

Note that the type constructors are also declared as constructors by entering `TypeConstructor` calls. This is so because of the type checker which assumes non-variable type expressions to be terms of the form $f[type_1, \ldots, type_n]$ where f is a constructor, i.e. a symbol for which `IsConstructor`[f] is `True`.

`TypeConstructor` calls can be invoked by the user to extend the signature of type expressions.

Notes on Typing in Type Expressions

The *Mathematica* front-end of the CFLP interpretor is configured such that type expressions can be entered in a *visual representation* that is more close to their intended meaning. Also, type expressions are always displayed in such a representation. The figure below illustrates the correspondence between the syntax representation of type expressions and the visual (or displayed) representation.

internal representation	visual representation
`FuncType`$[\tau_1, \tau_2]$	$\tau_1 \rightarrow \tau_2$
`Cross`$[\tau_1, \tau_2, \ldots]$	$\tau_1' \times \tau_2' \times \cdots$

Figure 2.1: internal and visual representation of type expressions

The *Mathematica* special characters appearing in the visual representations of type expressions are

visual representation	*Mathematica* special character
$\square$	\[Placeholder]
$\rightarrow$	\[ShortRightArrow]
$\times$	\[Cross]

Here, τ' denotes the visual representation of τ. So, for example, the type expression denoting the type of real functions taking two integer arguments has the internal representation `Cross`[`Int`, `Int`] $\rightarrow$ `Real` and the visual representation `Int` $\times$ `Int` $\rightarrow$ `Real`.

The implementation of this behavior is carried out in two steps. First, we use the *Mathematica* `MakeExpression` mechanism to construct the internal type expression corresponding to the visual representation

```
MakeExpression[RowBox[x_,"::",y_], StandardForm]:=
  MakeExpression[RowBox[{"TypeOf","[",x,",",y,"]"}], StandardForm];
MakeExpression[RowBox[{x_,"→" ,y_}], StandardForm] :=
  MakeExpression[RowBox[{"FuncType","[",x,",",y,"]"}], StandardForm];
```

After this sequence of code is executed, type expressions can be provided to the system in both internal and visual representation. However, the output represenation of our expressions is the internal one. We can change this too by making definitions for the low-level Mathematica function `MakeBoxes`. They specify how to construct expressions from so-called *boxes*.

```
FuncType /:  MakeBoxes[FuncType[ τ1_, τ2_ ], StandardForm]:=
RowBox[{MakeBoxes[τ1,StandardForm]," → ",MakeBoxes[τ2,StandardForm]}];
```

We provide visual representation support also for equations, (conditional) rewrite rules and type annotations. When interacting with the CFLP interpreter, the user must only know the meaning of expressions in visual representation and how to type them in. Typing in expressions in visual representation is nicely supported by the facility of palletes. The CFLP distribution contains such a pallete (`CFLPsyntax.nb`) with templates for all the visual representations of our language.

2.2 The CFLP Object Language

The underlying signature of the CFLP object language consists of

1. a class $\mathcal{F}_d$ of defined function symbols (or defined functions),
2. a class $\mathcal{F}_c$ of constructor symbols,
3. a class $\mathcal{F}_e$ of external function symbols,
4. a class $\mathcal{V}$ of variables.

We denote by $\mathcal{F}$ the union $\mathcal{F}_d \cup \mathcal{F}_c \cup \mathcal{F}_e$ and refer to it as the class of function symbols.

In our calculus, the set $\mathcal{F}_d$ contains those function symbols that are "defined" by some conditional rewrite rule. This means that for every defined function symbol f we assume the presence of a conditional rewrite rule r with f the head of its left hand side (see 2.5.1).

The symbols of class $\mathcal{F}_e$ are used in our calculus to identify constraints, i.e. equations that can not be solved in the functional logic framework. For example, we consider `Plus`, `Times`, ... as external function symbols over reals because they are used in expressing equalities that can be solved by solvers over reals. This class makes the CFLP signature differ from the usual signature of a narrowing calculus.

Constructor symbols (abbreviated as *constructors* by convention) are used to construct data structures.

2.2.1 Terms

The syntax domain `Term` of terms is defined by the following grammar:

`Term`	::=	`Variable` $\mid$ `Literal` $\mid$ `FunctionSymbol`
		$\mid$ `Term[Term, ..., Term]` $\mid$ `LambdaTerm`
`Variable`	::=	v if $v \in \mathcal{V}$
`Literal`	::=	l if $l \in \mathcal{C}_{\text{Bool}} \cup \mathcal{C}_{\text{Int}} \cup \mathcal{C}_{\text{Symbol}} \cup \mathcal{C}_{\text{Real}}$
`FunctionSymbol`	::=	`ConstructorSymbol`
		$\mid$ `DefinedFunctionSymbol`
		$\mid$ `ExternalFunctionSymbol`
`ConstructorSymbol`	::=	c if $c \in \mathcal{F}_c$
`DefinedFunctionSymbol`	::=	f if $f \in \mathcal{F}_d$
`ExternalFunctionSymbol`	::=	f if $f \in \mathcal{F}_e$
`LambdaTerm`	::=	λ[`VarList`, `Term`]
`VarList`	::=	{`Variable`, ..., `Variable`}

Apart from the syntax definition of `VarList`, we also impose the constraint that all the variables of a `VarList` expression must be distinct. Therefore, if we assume that $f \in \mathcal{F}_c$ and $x, y \in \mathcal{V}$, then the expression $\lambda[\{x, x\}, g[x, x]]$ is not a term because the list of variables given as first argument of λ does not consist of distinct variables.

set	meaning
$\mathcal{V}$	the set of object variables.
$\mathcal{F}_c$	the set of constructor symbols.
$\mathcal{F}_d$	the set of defined function symbols.
$\mathcal{F}_e$	the set of external function symbols.
$\mathcal{C}_{\text{Bool}}$	the set of boolean literals.
$\mathcal{C}_{\text{Symbol}}$	the set of symbol literals.
$\mathcal{C}_{\text{Int}}$	the set of integer literals.
$\mathcal{C}_{\text{Real}}$	the set of real literals.

Figure 2.2:

We note that the set of CFLP terms is determined by the sets $\mathcal{V}$, $\mathcal{C}_{\text{Bool}}$, $\mathcal{C}_{\text{Int}}$, $\mathcal{C}_{\text{Symbol}}$, $\mathcal{C}_{\text{Real}}$, $\mathcal{F}_c$, $\mathcal{F}_d$ and $\mathcal{F}_e$, which are assumed pairwise disjoint.

Terminology

In the sequel we denote by $\mathcal{C}$ the set $\mathcal{C}_{\text{Bool}} \cup \mathcal{C}_{\text{Symbol}} \cup \mathcal{C}_{\text{Int}} \cup \mathcal{C}_{\text{Real}}$, and by $\mathcal{T}\text{erm}[\mathcal{F}, \mathcal{V}]$ the set of valid terms determined by $\mathcal{V}$ and $\mathcal{F}$. $\mathcal{C}$ is called the set of *constants*. For a term t of the form $t_0[t_1, \ldots, t_n]$, the term t_0 is called *root* of t, and denoted by $\text{root}[t]$.

We denote by $\mathcal{V}[t]$ the set of variables of a term $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$. Formally:

$$\mathcal{V}[t] = \begin{cases} \{t\} & \text{if } t \in \mathcal{V}, \\ \mathcal{V}[t_0] \cup \bigcup_{i=1}^{n} \mathcal{V}[t_i] & \text{if } t = t_0[t_1, \ldots, t_n] \\ \mathcal{V}[t] - \{x_1, \ldots, x_n\} & \text{if } t = \lambda[\{x_1, \ldots, x_n\}, t] \\ \emptyset & \text{otherwise} \end{cases}$$

2.2.2 Implementation

Function Symbol Classifiers

We distinguish the classes of function symbols by the way they are introduced in the language. In the sequel we mention the syntax constructs for introducing these classes but their full implementation will be given later in separate sections.

Constructors are recognized by the constructor classifier `IsConstructor`. The class of symbols recognized by `IsConstructor` may be extended with `Constructor` calls. The *Mathematica* program for `Constructor` takes as arguments a sequence of *Mathematica* symbols or of type-annotated *Mathematica* symbols (or type assignments); for every symbol of the sequence, if the symbol was not previously declared, the definition of `Constructor` to yield `True` for it. By default, all the constants of basic types `Int`, `Bool`, `Symbol` and `Real` are assumed to be constructors. The *Mathematica* program for recognizing constructors is

```
IsConstructor[c_?AtomQ] /; (c[[0]] =!= Symbol) := True;
IsConstructor[_]:= False;
```

and is extended with new definitions via `Constructor` calls. The following call is performed during initialization [1]

```
Constructor[True::Bool, False::Bool, eq::α × α → α,
    RewriteRule::α × α × Bool → Bool,
    Cons::α × TyList[α] → α, •::TyList[α]
]
```

We omit here the implementation details related to type specifications and only mention that this call has the effect to add the `IsConstructor` definitions

```
IsConstructor[f]=True;
```

for $f \in \{$`True`, `False`, `eq`, `RewriteRule`, `Cons`, $\bullet\}$. Here, `True` and `False` are the `Bool` constants of our language. The symbol $\bullet$ stands for the empty list, and `Cons` is the list constructor. In the current implementation, we consider lists

[1] :: is the visual representation of the *Mathematica* symbol \[Proportion]

as constructed types with two constructors: the nullary constructor $\bullet$ and the binary constructor `Cons`. The list term $\texttt{Cons}[h, t]$ represents the list obtained by prepending term h to list t. The visual representation of the list term $\texttt{Cons}[h, t]$ is $\langle h', t' \rangle$, where h', t' are the visual representations of h and t. The special *Mathematica* characters appearing in the visual representation of lists are

visual representation	*Mathematica* special character
$\bullet$	\\[Bullet]
$\langle$	\\[LeftAngleBracket]
$\rangle$	\\[RightAngleBracket]

For the sake of convenience, we also allow representations of list terms in *Mathematica* notation, i.e. we write $\{term_1, \ldots, term_n\}$ for the list with elements $term1, \ldots, term_n$. This representation is transformed into the CFLp representation $\langle term_1, \ldots, \langle term_n, \bullet \rangle \cdots \rangle$ by the *Mathematica* function `ListToCons`. See §2.7.1 for details about when `ListToCons` is called and how it is implemented.

Similar to `IsConstructor`, the *Mathematica* functions `IsDefinedFunctionSymbol` and `IsExternal` are provided as classifiers for the sets $\mathcal{F}_d$ and respectively $\mathcal{F}_e$. Their default implementations are

```
IsDefinedFunctionSymbol[_]:=False;
IsExternal[_]:=False;
```

and are extended to recognize more symbols by invoking respectively `DefinedFunctionSymbol` and `External`. An `IsExternal` call is performed during system initialization

```
External[Plus::Real × Real → Real, Times::Real × Real → Real,
    Power::Real × Real → Real, Sin::Real → Real, Cos::Real → Real
    And::Bool × Bool → Bool, Or::Bool × Bool → Bool, Not::Bool → Bool,
]
```

and has the effect of adding the definitions

$\texttt{IsExternal}[f] = \texttt{True};$

for $f \in \{$`Plus`, `Times`, `Power`, `Sin`, `Cos`, `And`, `Or`, `Not`$\}$. These external functions have the following meaning:

`Plus`	addition over reals
`Times`	multiplication over reals
`Power`	power operator over reals
`Sin`, `Cos`	trigonometric functions over reals
`And`,`Or`,`Not`	the corresponding operators over booleans

Because $\mathcal{F}_e$, $\mathcal{F}_c$ and $\mathcal{F}_d$ are required to be pairwise disjoint sets, the commands `External, DefinedFunctionSymbol` and `Constructor` for declaring them prevent the user from introducing function symbols that belong to more than one such set.

NOTE.
The command `External` is not intended for the user but to be called by the CFLP interpreter during the initialization phase. $\square$

All the *Mathematica* symbols that do not belong to $\mathcal{F}_e \cup \mathcal{F}_c \cup \mathcal{F}_d$ are assumed to be either `Symbol` object variables or `Symbol` literals; such a symbol is a variable (i.e. element of $\mathcal{V}$) if it has attribute `Temporary`, otherwise it is a `Symbol` literal.

Term Classifiers

The classification of function symbols induces also a term classification which is useful when describing and implementing the narrowing calculus of the functional logic interpreter.

We introduce the following term classifiers

- flex term classifier

  ```
  IsNotFlex[x_?IsVar] := False;
  IsNotFlex[x_?IsVar[___]] := False;
  IsNotFlex[_] := True;
  ```

- constructor term classifier

  ```
  IsConstructorTerm[(_?IsConstructor)[___] | _?IsConstructor] := True;
  ```

- classifier for terms with defined function symbol head.

  ```
  IsFunctionTerm[(_?IsDefinedFunctionSymbol)[___]
  | _?IsDefinedFunctionSymbol] := True;
  ```

Variable Classifiers

We mentioned that in our implementation a variable is a *Mathematica* symbol with the list of attributes {`Temporary`}. The corresponding classifier for this class is implemented by

```
IsVar[x_] :=
  If[Head[x] === Symbol,
    Attributes[x] === Temporary,
    False];
```

Variable Manipulation

For declaring variable symbols, the function call

`SetVariables`[*varlist*];

is performed to make `Temporary` the symbols of *varlist*. `SetVariables` can also be used to assign a type to some variable symbol: in this case *varlist* will contain a type annotation instead of a symbol (see next section).

The dual command for undeclaring variables is

`ClearVariables`[*varlist*];

Session Example

In[1]=	`DefinedFunctionSymbol`[Plus,f,g];
	Warning: Plus is already declared as external function symbol
Out[1]=	{f,g}
In[2]=	`Constructor`[f,h]
	Warning: f is already declared as constructor symbol
Out[2]=	{h}
In[3]=	`IsConstructor`[f]
Out[3]=	False
In[4]=	`IsDefinedFunctionSymbol`[f]
Out[4]=	True
In[5]=	`IsExternal`[f]
Out[5]=	False
In[6]=	`IsFunctionTerm`[f[u,v]]
Out[6]=	True
In[7]=	`IsConstructor`[u+v+w]
Out[7]=	False

2.3 Type Assignments

Type assignments are language constructs for specifying the types of function symbols. The set of polymorphic types of our language is obtained by adding universal quantification to type expressions. From now on we denote by $\mathcal{T}\text{ype}[\mathcal{V}_{type}]$ the set of polymorphic types:

$$\mathcal{T}\text{ype}[\mathcal{V}_{type}] = \mathcal{TE}\text{xpr}[\mathcal{V}_{type}] \cup \{\forall_{\alpha_1,\ldots,\alpha_n}\tau, \mid \begin{array}{l} \tau \in \mathcal{TE}\text{xpr}[\mathcal{V}_{type}], n > 0, \\ \{\alpha_1,\ldots,\alpha_n\} \subseteq \mathcal{V}_{type}[\tau]\} \end{array}$$

2.3.1 Syntax Domains

We will distinguish four classes of type assignments: for defined function symbols (syntax domain `DefinedFSSig`), for constructor symbols (syntax domain `ConstructorSig`), for external function symbols (syntax domain `ExternalFS-Sig`), and for variables (syntax domain `VarSig`).

$$
\begin{aligned}
\texttt{ConstructorSig} &::= \texttt{ConstructorSymbol::Type}\\
\texttt{DefinedFSSig} &::= \texttt{DefinedFunctionSymbol::Type}\\
\texttt{ExternalFSSig} &::= \texttt{ExternalSymbol::Type}\\
\texttt{VarSig} &::= \texttt{Variable::Type}
\end{aligned}
$$

Figure 2.3: The syntax domains `ConstructorSig`, `DefinedFSSig`, `ExternalFSSig` and `VarSig`

2.3.2 The Semantics of Type Expressions

When we talked about the underlying signature of type expressions (see §2.1), we mentioned that a type expressions consists only of type constructors and free variables. Therefore, a type expression τ is assumed to denote the polymorphic type $\forall_{\alpha_1,\ldots,\alpha_n}\tau$ where $\{\alpha_1,\ldots,\alpha_n\}$ is the set $\mathcal{V}_{type}[\tau]$ of variables appearing in τ. Sometimes we will simply write $\forall\tau$ instead.

In the sequel we denote by $\mathcal{T}\text{ype}[f]$ the polymorphic type associated to a symbol $f \in \mathcal{F}$. Assume $f{::}\tau$ is a type annotation for a symbol $f \in \mathcal{F}$. There are two possibilities:

1. $f \in \mathcal{F}_d \cup \mathcal{F}_e$ then
$$\mathcal{T}\text{ype}[f] = \forall\tau$$
2. $x \in \mathcal{V}$; then
$$\mathcal{T}\text{ype}[x] = \tau'$$
where τ' is a fresh type instance of $\forall\tau$
3. $f \in \mathcal{F}_c$ then
   (a) if τ is a function type expression $\tau_1 \to \tau_2$ then $\mathcal{T}\text{ype}[f] = \forall\tau$
   (b) otherwise, $\mathcal{T}\text{ype}[f] = \forall(\Box \to \tau)$.

2.3.3 Type Declarations

Type assignments are grouped together inside so called type declarations. We distinguish among the following categories of type declarations:

> **for external symbols**: they contain type assignments for external function symbols.
>
> **for constructors**: they contain type assignments for constructor symbols;
>
> **for defined function symbols**: they contain type assignments for defined function symbols;
>
> **for variables**: they contain type signatures for variable symbols.

The last 3 cathegories can be either *local* or *global*. Therefore we have finally five categories of type declarations: for external function symbols, local/global for constructors, and local/global for defined function symbols.
The syntax rule of the domain `TypeDecl` of type declarations is

`TypeDecl`	::=	`ExternalTypeDecl`
		| `LocalConstrTypeDecl`
		| `GlobalConstrTypeDecl`
		| `LocalDefTypeDecl`
		| `GlobalDefTypeDecl`
`ExternalTypeDecl`	::=	`External`[`ExternalFSSig`, ..., `ExternalFSSig`];
`LocalConstrTypeDecl`	::=	`Constructor -> ConstrTSsList`
`ConstrTSsList`	::=	{} | {`ConstrTSsSeq`}
`ConstrTSsSeq`	::=	`ConstructorSig`
		| `ConstructorSig`, ..., `ConstructorSig`
`GlobalConstrTypeDecl`	::=	`Constructor`[ ];
		`Constructor`[`ConstrTSsseq`];
`LocalDefTypeDecl`	::=	`DefinedFunctionSymbol -> DefTSsList`
`DefTSsList`	::=	{ } | {`DefTSsSeq`}
`DefTSsSeq`	::=	`DefinedFSSig`
		| `DefinedFSSig`, ..., `DefinedFSSig`
`GlobalDefTypeDecl`	::=	`DefinedFunctionSymbol`[ ]
		| `DefinedFunctionSymbol`[`DefTSsSeq`]

Semantically, the distinction between local and global type declarations is the following: a local type declaration is considered only for the goal into which it is specified; in contrast, a global type declaration is taken into account for all the subsequent goals that are issued to the interpreter (see §2.7).

2.3.4 Implementation

Type Variables

In type expressions we consider type variable any *Mathematica* symbol different from a type constructor or basic type. For determining the list of type variables of type expressions, the auxiliary function **TyVar**[ ] is provided.

TyVar[τ_] :=
  Union[Cases[{τ}, _Symbol?(Not[IsTypeConstructor[#1]]&), -1]];

Polymorphic Types

The polymorphic type $\forall\alpha_1, \ldots, \alpha_n.\tau'$ associated to a function symbol $f \in \mathcal{F}$ via a type assignment $f{::}\tau$ is stored into **Type**[f] in the following format

$$\texttt{Function}[\{\}, \texttt{Module}[\{\alpha_1, \ldots, \alpha_n\}, \{\{\alpha_1, \ldots, \alpha_n\}, \tau'\}]]$$

This encoding is generated by the function call

FormTypeExpr[$\{\alpha_1, \ldots, \alpha_n\}, \tau'$]

and is stored in **Type**[f]. From now on, fresh type instances of f are generated by the invocation

Type[f][]

This call returns a pair of the form $\{\{\alpha'_1, \ldots, \alpha'_n\}, \tau'\theta\}$ where $\theta = \{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}$ and $\alpha'_1, \ldots, \alpha'_n$ are fresh type variables.

Type Declarations

The *Mathematica* functions for declaring types coincide with those for declaring function symbols. We mentioned partially these commands in §2.2.2. We now discuss the way they are implemented.

Constructor is defined as follows:

```
Constructor[c_Symbol]:=
 If[IsDeclared[c],
  { },
  Type[f]=FormTypeExpr[{},NewC[]];
  AppendTo[$FunctionSymbol,c];
  Arity[c]=0;
  IsConstructor[c]=True;
  {c}];
Constructor[TypeOf[c_,τ_?IsFunType]]:=
```

```
 If[IsDeclared[c],
  { },
  IsConstructor[c]=True;
  If[Not[MemberQ[{RewriteRule,eq,ppeq},c]],
   AppendTo[$FunctionSymbol,c];
   Arity[c]=If[IsFunOnCPType[τ],
    Length[First[τ]],
    If[First[τ]===□,0,1]]
    ];
   DeclareType[TypeOf[c,τ]];
   {c}];
Constructor[TypeOf[c_,τ_]]:=
 (If[IsDeclared[c],
  { },
  IsConstructor[c]=True;
  AppendTo[$FunctionSymbol,c];
  Arity[c]=0;
  DeclareType[TypeOf[c,FuncType[□,τ]]];{c}]);
Constructor[cs__]:=Apply[Join,Map[Constructor, {cs}]];
```

The first thing that is done when declaring a symbol f as constructor is to verify if f was not already declared. This fact is determined by the call `IsDeclared`[f] where

```
IsDeclared[f_]:=
 If[IsExternal[f],
  Print["Warning: ", f, " is already declared as external function symbol"];
  True,
  If[IsDefinedFunctionSymbol[f],
   Print["Warning: ", f, " is already declared as defined function symbol"];
   True,
   If[IsConstructor[f],
    Print["Warning: ", f, " is already declared as constructor symbol"];True,
    False ]]];
```

Next, if the symbol f is declared for the first time, then

- if f was not type annotated then assign to f a fresh type constant created by the call `NewC`[ ], define `Arity`[f] to be 0 and add the definition `IsConstructor`[f] = `True`.
- if f has specified the polymorphic type $\forall\tau$ (see §2.3.2) then define Arity[f] in accordance with τ and add it to the list $FunctionSymbol.

Finally, all newly declared constructors different from `RewriteRule` and `eq` are added to the list $FunctionSymbol.

The rationale of maintaining Arity and $FunctionSymbol in the system is for the implementation of the outermost narrowing inference rules on flex terms (see §4.4.2).

The functions `External` and `DefinedFunctionSymbol` corresponding to (type) declarations for external symbols and defined function symbols are identical to `Constructor`, except that:

- instead of defining `IsConstructor` they define `IsExternal` and respectively `IsDefinedFunctionSymbol`,
- Arity is undefined for external symbols, and external symbols are not added to $FunctionSymbol.

Type declarations for variables are performed by type annotating the parameters and variable symbols provided to a CFLP query (see §2.7). Parameter specification reduce to `Constructor` calls, and specifications of eliminable variables reduce to `SetVariables` calls. For example, specifying $\{x{::}\texttt{Real}, y{::}\texttt{Int}\}$ as second argument to a `TSolve` call will trigger

$\texttt{SetVariables}[x{::}\texttt{Real}, y{::}\texttt{Int}]$;

inside the `TSolve` body (see Figure 2.4).

The function `SetVariables` is defined as follows.

$\texttt{SetVariables}[l\_\mathrm{List}] := \mathrm{Join}\ @@\ \texttt{SetVariables}\ /@\ l$;
$\texttt{SetVariables}[v\_\mathrm{Symbol}] := (\mathrm{Attributes}[v] = \{\mathrm{Temporary}\}; \{\})$;
$\texttt{SetVariables}[v\_{::}\tau\_] :=$
  $\mathrm{If}[\mathrm{DeclareType}[v{::}\tau] =!= \{\},$
    $\mathrm{Attributes}[v] = \mathrm{Temporary};$
    $\mathrm{Type}[v] = \mathrm{FormTypeExpr}[\{\}, \mathrm{Type}[v][\,]];$
    $\{v\},$
    $\{\}]$;

Global type declarations are implemented simply as calls of the functions mentioned above.

Local rule declarations are also implemented as calls to `Constructor` and `DefinedFunctionSymbol`, but inside a `TSolve` call (see §2.7.1) and such that, when `TSolve` resumes, the effect of their calls is removed.

Arity

We implement a separate function `Arity` to yield the arity of function symbols. In our current implementation, `Arity` is of interest only for defined function symbols and constructor symbols different from `RewriteRule` and `eq`. These two symbols are reserved for the system use when constructing equations and conditional rewrite rules.

Function symbol arities are defined when function symbols are declared, in accordance with the type associated to them.

Session Example

In[1]=	(* declare the type signature for length of lists *) DefinedFunctionSymbol[Length::`TyList`[α] $\to$ Int]
Out[1]=	{`Length`}
In[2]=	(* Length can not be redefined; a warning is displayed *) Constructor[Length]
Out[2]=	"Warning: Length is already declared as defined function symbol"
In[3]=	(* ask for a valid type instance of Length *) Type[Length][ ]
Out[3]=	$\{\{\alpha\$1\}, \mathrm{TyList}[\alpha\$1] \to \alpha\$1\}$
In[4]=	(* get the arity of Length *) Arity[Length]
Out[4]=	1
In[5]=	(* external functions, e.g. Plus, have undefined arity *) Arity[Plus]
Out[5]=	Arity[Plus]
In[6]=	$FunctionSymbol
Out[6]=	{True, False, •, Cons, Length}

2.4 Goals

A CFLP *goal* denotes a system of equations that has to be solved. In our language, a goal is represented by a list of *equations*. Goals and their variable sets are specified inside CFLP queries (see §2.7) which are syntax constructs corresponding to calls of the CFLP interpreter.

From the user point of view, all the equations that may appear into a goal are of the same kind, namely *strict* equations. However, during the computation, the user specified goal is successively transformed into goals that may also contain so called *parameter passing* equations. This distinction between equations is necessary when describing the underlying calculus of the CFLP interpreter. The syntax domains `Goal` for goals and `InitialGoal` for initial goals are defined by the following grammar

`Goal`	::=	{`GoalSeq`}
`InitialGoal`	::=	{`InitialGoalSeq`}
`GoalSeq`	::=	`Equation` | `Equation`, `GoalSeq`
`InitialGoalSeq`	::=	`InitialEquation` | `InitialEquation`, `InitialGoalSeq`
`Equation`	::=	`InitialEquation`
		| `PPEquation`
`InitialEquation`	::=	`Term` $\approx$ `Term`
`PPEquation`	::=	`Term` $\asymp$ `Term`

Here **`InitialEquation`** is the syntax domain of initial (or user-defined) equations and **`PPEquation`** is the domain of parameter passing equations.

We note that in the CFLP syntax, an equation has the equality sign $\approx$ (for strict equations) or $\asymp$ (for parameter passing equations). The equality symbols $\asymp$ and $\approx$ correspond to the *Mathematica* special characters

Mathematica symbol	visual representation
\\`[TildeTilde]` or `eq`	$\approx$
\\`[CupCap]` or `ppeq`	$\asymp$

Terminology

For a given set $\mathcal{V}$ of variables and a set $\mathcal{F}$ of function symbols, we denote by $\mathcal{E}\mathrm{q}[\mathcal{F},\mathcal{V}]$ the set of equations, by $\mathrm{s}\mathcal{E}\mathrm{q}[\mathcal{F},\mathcal{V}]$ the set of initial equations, by $\mathrm{pp}\mathcal{E}\mathrm{q}[\mathcal{F},\mathcal{V}]$ the set of parameter passing equations, by $\mathcal{G}\mathrm{oal}[\mathcal{F},\mathcal{V}]$ the set of goals, and by $\mathcal{I}\mathcal{G}\mathrm{oal}[\mathcal{F},\mathcal{V}]$ the set of initial goals.

The set $\mathcal{V}[e]$ of variables of an equation $e \in \mathcal{E}\mathrm{q}[\mathcal{F},\mathcal{V}]$ of the form $t_1 \approx t_2$ or $t_1 \asymp t_2$ with $t_1, t_2 \in \mathcal{T}\mathrm{erm}[\mathcal{F},\mathcal{V}]$ is

$$\mathcal{V}[e] = \mathcal{V}[t_1] \cup \mathcal{V}[t_2].$$

The set $\mathcal{V}[g]$ of variables of a goal $g \in \mathcal{G}\mathrm{oal}[\mathcal{F},\mathcal{V}]$ of the form $\{e_1, \ldots, e_n\}$ with $e_1, \ldots, e_n \in \mathcal{E}\mathrm{q}[\mathcal{F},\mathcal{V}]$ is

$$\mathcal{V}[g] := \bigcup_{i=1}^{n} \mathcal{V}[e_i].$$

2.4.1 Implementation

Internally, the goals have the following representation

visual representation	internal representation
$lhs \approx rhs$	$\mathrm{eq}[lhs', rhs']$
$lhs \asymp rhs$	$\mathrm{ppeq}[lhs', rhs']$

where lhs', rhs' are the internal representations of lhs and respectively rhs. For the CFLP user, the internal representation is not important to know, since the visual representation can be used instead. However, note that the symbols $\approx$, $\asymp$, `eq` and `ppeq` are reserved for the use of the system.

2.5 Conditional Rewrite Rules

A *conditional rewrite rule* (CRR for short) is an oriented equation that holds under the assumption that certain equations called *conditions* also hold. Conditional rewrite rules are the constituents of the equational theory with respect to which we solve the systems of equations represented by CFLP goals.

2.5.1 Syntax

The syntax domain `RwRule` of conditional rewrite rules has the syntax rule

$$\begin{array}{lll} \texttt{RwRule} & ::= & \texttt{Term} \longrightarrow \texttt{Term} \\ & & \mid \texttt{Term} \longrightarrow \texttt{Term} \Leftarrow \texttt{InitialEquation} \\ & & \mid \texttt{InitialEquation} \wedge \cdots \wedge \texttt{InitialEquation} \end{array}$$

Remark. The *Mathematica* special symbols appearing in the visual representation of conditional rewrite rules are

Mathematica input	visual representation
`\[RightArrow]`	$\longrightarrow$
`\[DoubleLeftArrow]`	$\Leftarrow$
`\[And]`	$\wedge$

Also, the infix expression $equ_1 \wedge \cdots \wedge equ_n$ can be replaced by the prefix expression $\texttt{And}[equ_1, \ldots, equ_n]$. □

Apart from the syntactic description of conditional rewrite rules, we also consider the following constraint: an expression R of the form

$$lhs \longrightarrow rhs \quad \text{or} \quad lhs \longrightarrow rhs \Leftarrow cond$$

with $lhs, rhs \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ and $cond \in \mathcal{IG}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$ is a conditional rewrite rule if $\mathcal{V}[rhs] \subseteq \mathcal{V}[lhs] \cup \mathcal{V}[cond]$, and lhs is of the form $f[t_1, \ldots, t_n]$ with $f \in \mathcal{F}_d, t_1, \ldots, t_n \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$. For the degenerate case $n = 0$ we allow the notation f instead of $f[\,]$ for denoting lhs.

Terminology

lhs is called the *left hand side term* of R, *rhs* the *right hand side* of R, *cond* the *conditional part* of R, and f the head function symbol of R. A conditional rewrite rule with empty condition is called simply *rewrite rule*. A *conditional term rewriting system* (CTRS for short) is a set of CRRs. A *term rewriting system* (TRS for short) is a set of rewrite rules.

The set of variables of a conditional rewrite rule $R\ :\ lhs \longrightarrow rhs \Leftarrow cond$ is

$$\mathcal{V}[R] := \mathcal{V}[lhs] \cup \mathcal{V}[cond].$$

The meaning of a conditional rewrite rule is

> the terms *lhs* and *rhs* are equal for those (type consistent) variable instances that are solutions of the system *cond* of equations. If *cond* is not provided then *lhs* and *rhs* are equal for every (type consistent) instance of their variables.

2.5.2 Implementation

We define an optimized encoding of conditional rewrite rules. The idea is based on the observation that a conditional rewrite rule

$$f[l_1, \ldots, l_n] \longrightarrow r \Leftarrow \{e_1, \ldots, e_m\}$$

can be represented by

$$f[x_1, \ldots, x_n] \longrightarrow x_1 \asymp l_1, \ldots, x_n \asymp l_n, r \Leftarrow \{e_1, \ldots, e_m\}$$

This representation makes possible

- to eliminate runtime creation of parameter passing equations, and
- to use the parameter binding mechanism of *Mathematica* rather than applying the variable elimination rules of our system by encoding every conditional rewrite rule as a function

  $\mathrm{Function}[\{v_1, \ldots, v_n\},$

  $\qquad \mathrm{Module}[\{x_1, \ldots, x_p\}, \mathrm{RewritesTo}[r, \{v_1 \asymp l_1, \ldots, v_n \asymp l_n\}, \{e_1, \ldots, e_m\}]]]$

  where $v_1, \ldots, v_n \in \mathcal{V}$ are fresh variables and $x_1, \ldots, x_p \in \mathcal{V}$ are the variables of R.

This encoding is generated in two steps:

1. call

   Uniform$[f[l_1, \ldots, l_n]]$

   This call returns a list $\{\{v_1, \ldots, v_n\}, \{v_1 \asymp l_1, \ldots, v_n \asymp l_n\}\}$ with $v_1, \ldots, v_n \in \mathcal{V}$ fresh variables.

2. call

   FormFunction$[\{v_1, \ldots, v_n\}, \{x_1, \ldots, x_p\}, \{\{v_1 \asymp l_1, \ldots, v_n \asymp l_n\}\ \{e_1, \ldots, e_n\}\}]$

   This call returns the encoding of R.

The encodings of rewrite rules whose left-hand side is a term of the form $f[\_\_\_]$ or f are maintained into a list `RewriteRuleList[f]`.

2.6 Rule Declarations

The equational theory of the CFLP calculus is defined by a system of conditional rewrite rules. The language constructs for specifying a system of CRRs are the *rule declarations*.

2.6.1 Syntax

The syntax rule for rule declarations (syntax domain `RuleDecl`)is shown in Figure 2.4.

`RuleDecl`	::=	`GlobalRwRuleDecl` | `LocalRuleDecl`
`GlobalRuleDecl`	::=	`Def[VList, RuleList, TypeCheck ->`b`];`
		| `Def[VList, RwRuleList];`
		where b is a `Bool` literal
`LocalRuleDecl`	::=	`Rule->RuleList`
`VList`	::=	{} | `VarList`
`RuleList`	::=	{} | {`RwRule`, ..., `RwRule`}
`TypeCheck`	::=	`True` | `False`

Figure 2.4: Syntax domain `RuleDecl`

2.6.2 Implementation

Global CRR declarations are implemented as invocations of the function `Def`. The syntax call for `Def` is

`Def`[*varList*, *rrList*, `TypeCheck ->` *b*]

or

`Def`[*varList*, *rrList*]

where

> *rrList* = the list of CRRs to be inserted into `RewriteRuleList` ,
>
> *varList* = the list of variables of the conditional rewrite rules of `rrList`,
>
> *b* is a `Bool` literal; if it is `True` then the CRRs of *rrList* are type checked, otherwise not.

The second call (without `TypeCheck` option) is equivalent to the call

`Def`[*varList*, *rrList*, `TypeCheck ->` *tyChk*]

where `TypeCheck ->` *tyChk* is the default option value of the function `TSolve` (see §2.7.1)

The call performs the following operations

1. $i := 1$; assume $rrList = \{R_i, \ldots, R_n\}$ for the case when $rrList \neq \{\}$;
2. if *rrList* is {} then return "⊤" (indicating success) else goto 3;
3. if *tyChk* is `False` then goto 6 else goto 4;
4. type check the rewrite rule R_i in which *varList* is considered to be the list of variables. If type checking succeeds then goto 6 else goto 5;
5. discard the type assignments made for the variables of *varList*; return "⊥" (indicating failure);
6. discard the type assignments made for the variables of *varList*;
7. append the encoding of R_i to the list `RewriteRuleList[`*f*`]` of encodings attached to the head symbol f of R_i;
8. $i := i + 1$; $rrList := \{R_{i+1}, \ldots, R_n\}$; goto 2;

A local CRR declaration

`Rule->` *rrList*

is an optional argument to a `TSolve` call (see §2.7.1). If provided, this argument triggers a local call

Def[*varList*,*rrList*,...];

inside `TSolve`, whose effect is cancelled from the *Mathematica* environment by restoring the definition of `RewriteRuleList` to the status before the `Def` call.

REMARKS.

- By default, for all defined function symbols $f \in \mathcal{F}_d$ we have

$$\texttt{RewriteRuleList}[f] === \{\ \};$$

- The call fails (i.e. the return value is "$\perp$") only if some rewrite rule is ill typed (see §3);
- Besides returning "$\top$" or "$\perp$", the following information is displayed:
  - if an error occurs, the user is informed which one is the first rewrite rule that is ill typed;
  - rewrite rules associated to a non-defined function symbol f are considered possible errors; in this case the system redeclares f to be a defined function symbol and notifies the user about this change.

Session example

In[1]=	(* declare a defined function symbol f and its type *) `DefinedFunctionSymbol`[f::Real×Real×Real→Real];
Out[1]=	{f}
In[2]=	(* declare constructors g, h, r and their types *) `Constructor`[g::Real×Real→Real, h::Cross[Real]→Int,r::Real→Int];
Out[2]=	{g,h,r}
In[3]=	(* define a conditional rewrite rule for f *) `Def`[{x,y,z},{f[y,x,z]⟶g[x,y] ⇐ (x ≈ z ∧ y ≈ 2.7)}] Adding rule #1: f[y,x,z]⟶g[x,y] ⇐x ≈ z && y ≈ 2.7 Symbols x,y,z declared as variables
Out[3]=	⊤
In[4]=	(* assign a CRR to a constructor*) Def[{x, y},{g[x,y]⟶h[r[x],1,2] ⇐ {1 ≈ 2}, f[y,x,z]⟶g[x,y] ⇐ {{x,y} ≈ {y,x}}}] Constructor symbol g redefined as a defined symbol. Adding rule #1: g[x,y]⟶ 1 Symbols x, y declared as variables. Adding rule #2: f[y,x,z]⟶ 1. Symbols x, y declared as variables.
Out[4]=	⊤
In[5]=	RewriteRuleList[f]
Out[5]=	{Function[{w$62,w$63,w$64}, Module[{x,y,z}, RewritesTo[g[x,y],{w$62≍y,w$63≍x, w$64≍z}, {x≈z,y≈2.7}]]]}

In this case we assume that `TypeCheck -> True` is the default option value of `TSolve`.

2.7 Queries

CFLP queries are expressions corresponding to a complete problem specification for the CFLP interpreter; the query structure contains a goal representing the system of equations to be solved, a list of conditional rewrite rules representing the equational theory into which the system has to be solved, the type assignments of the user specified symbols (constructors and defined function symbols), the maximum number of solutions to look for, and a flag that acts as switch for the type checker. CFLP queries are implemented as calls to the *Mathematica* function `TSolve`. The syntax call of `TSolve` resembles the `Solve` routine call.

$$
\begin{array}{rcl}
\texttt{Query} & ::= & \texttt{TSolve}[\texttt{InitialGoal}, \texttt{ElimList}, \texttt{Option}, \ldots, \texttt{Option}]; \\
 & & \mid \texttt{TSolve}[\texttt{InitialGoal}, \texttt{ElimList}, \texttt{ParamList}, \texttt{Option}, \ldots, \texttt{Option}]; \\
\texttt{ElimList} & ::= & \{\ \} \mid \{\texttt{VarSpec}, \ldots\}; \\
\texttt{ParamList} & ::= & \{\ \} \mid \{\texttt{VarSpec}, \ldots\}; \\
\texttt{VarSpec} & ::= & \texttt{Variable} \mid \texttt{VarSig}
\end{array}
$$

Figure 2.5: `Query` syntax domain

A fully specified query has the form

TSolve[*goal*, *elims*, *params* Rule->*rrList*,
Constructor->*cList*,
DefinedFunctionSymbol->*dList*,
TypeCheck->*tyChk*,
Initialize->*doInit*,
NSolution->n];

where

- *goal* $\in \mathcal{IG}\text{oal}[\mathcal{F}, \mathcal{V}]$ denotes the system of equations to be solved,
- *elims* is the list of (type annotated) eliminable variables of *goal*, and also the rewrite rule variables (i.e. variables in the elements of the list *rrList*),
- *params* is the list of (type annotated) parameter variables of *goal*,
- *rrList* is the list of conditional rewrite rules that together with the globally specified ones form the conditional term rewriting system with respect to which we solve the system of equations,

- *cList* is a list with type assignments for constructors,
- *dList* is the list with type assignments for defined function symbols,
- `tyChk` is a `Bool` literal which is `True` when type checking is required and `False` otherwise,
- if *doInit* is `True` then all previously specified conditional rewrite rules are deleted, otherwise they are not deleted and hence considered in the equational theory of solving the query request.
- n is an `Int` literal indicating the maximum number of solutions to be computed.

Every CFLP query is equivalent with a fully specified query. The fully specified query corresponding to a given query is determined as follows

- If an `Option` argument is missing then we add the *default* value of that option. The default option values are

  `Rule->`{ },
  `Constructor->`{ },
  `DefinedFunctionSymbol->`{ },
  `TypeCheck->True`,
  `Initialize->False`
  `NSolution->`256.
- if an option appears more than once, keep just the leftmost occurrence of that option.
- after exhausting the previous two transformations, all the queries generated by permuting the `Option` arguments are assumed to be equivalent.

EXAMPLE. The query

$$\begin{aligned}&\mathrm{TSolve}[\{f[x,a]\approx g[a,b]\},\{x,y\},\mathrm{Rule}\texttt{->}\{f[x,x]\longrightarrow g[a,b]\},\\&\quad\mathrm{DefinedFunctionSymbol}\texttt{->}\{f{::}\mathrm{Symbol}\times\mathrm{Symbol}\to\mathrm{Int}\},\\&\quad\mathrm{Constructor}\texttt{->}\{g{::}\alpha\times\alpha\to\mathrm{Int}\},\\&\quad\mathrm{Rule}\texttt{->}\{f[a,a]\approx 1, f[x,a]\approx 2\Leftarrow x\approx a\},\\&\quad\mathrm{DefinedFunctionSymbol}\texttt{->}\{f{::}\square\to\mathrm{Bool}, g{::}\texttt{TyList}[\mathrm{Real}]\}];\end{aligned}$$

is equivalent to

$$
\begin{aligned}
&\text{TSolve}[\{f[x,a] \approx g[a,b]\}, \{x,y\},\\
&\quad \text{Rule->}\{f[x,x] \longrightarrow g[a,b]\},\\
&\quad \text{Constructor->}\{g{::}\alpha \times \alpha \to \text{Int}\},\\
&\quad \text{DefinedFunctionSymbol->}\{f{::}\text{Symbol} \times \text{Symbol} \to \text{Int}\},\\
&\quad \text{TypeCheck->True},\\
&\quad \text{Initialize->False},\\
&\quad \text{NSolution->256}];
\end{aligned}
$$

2.7.1 Implementation

Queries are calls of the *Mathematica* function `TSolve`. Its function call is similar to that of the function `Solve`. The values for "named" optional arguments are specified by including the appropriate transformation rules in the function call. The default values of these arguments are stored as a list of transformation rules assigned to `Options[TSolve]`: we assume that

```
Options[TSolve] = {
    Rule->{ },
    Constructor->{ },
    DefinedFunctionSymbol->{ },
    TypeCheck->True,
    Initialize->False,
    NSolution->256}
```

We already mentioned that `TSolve` restores the *Mathematica* environment to the state that has been before performing the call. This behavior is illustrated in Figure 2.6.

We notice that whenever an environment entity X is about to be modified, we store it into a local variable save$X$ that is used to restore X in the end. Because all the modifiable entities X are *Mathematica* functions, we assign to save$X$ the definitions in which X appears as the head function symbol, termed downvalues of X:

$$\text{save}X = \text{DownValues}[X];$$

In the end, these downvalues are restored:

$$\text{DownValues}[X] = \text{save}X;$$

For getting the values of the optional arguments, first the rewrite rules provided as arguments to `TSolve` are looked up, and next the default options. The implementation of the operation to get the current value of an option with name X is

```
TSolve[ goal_List, vars_List, opts___] := TSolve[goal, vars, {}, opts];
TSolve[ goal_List, vars_List, elims_List, opts___] :=
  Block[{
     saveRewriteRuleList = DownValues[RewriteRuleList],
     saveIsDefinedFunctionSymbol = DownValues[IsDefinedFunctionSymbol],
     saveIsConstructor = DownValues[IsConstructor],
     saveType = DownValues[Type],
     Global`NSolutions,sol },
    Global`NSolutions=NSolution /. {opts} /. Options[TSolve];
    If[Global`NSolutions ≤ 0,
     {},
     If[Initialize/.{opts}/.Options[TSolve],BeginRule[]];
     Constructor @@ Join[Constructor /. {opts} /. Options[TSolve], elims];
     DefinedFunctionSymbol @@ (DefinedFunctionSymbol /. {opts} /. Options[TSolve]);
     If[TypeCheck /. {opts} /. Options[TSolve],
      If[Def[vars,
          Rule /. {opts} /. Options[TSolve],
          TypeCheck -> True] === "T",
        SetVariables[vars];
        sol=Function[TSolve[Sequence @@ #,TypeCheck-> False, NSolution-> 2]]
             [TcX[goal,TyList[Bool],{ },{ }]];
        sol=If[sol === { }],
          Print["Type check error: can not type the goal."],
          DownValues[Type]=DownValues[Type]/.Normalize[sol[[1]];
          CallCFLP[ListToCons /@ goal,vars]
         ];
        ClearVariables[vars]
     ],
     Def[vars, Rule /. {opts} /. Options[TSolve], TypeCheck -> False];
     sol=CallCFLP[ListToCons /@ goal, vars ];
     ];
     DownValues[RewriteRuleList]=saveRewriteRuleList;
     DownValues[IsDefinedFunctionSymbol]=saveIsDefinedFunctionSymbol;
     DownValues[IsConstructor]=saveIsConstructor;
     DownValues[Type]=saveType;
     sol
    ]
  ]

  TSolve[goal_, vars_List, opts___]:=TSolve[{goal}, vars, opts];
```

Figure 2.6: The `TSolve` function.

$$X \ /. \ \{\text{opts}\} \ /. \ \text{Options[TSolve]}$$

After setting the proper environment for performing the calculus, the CFLP interpreter is called:

`CallCFLP`[ListToCons /@ goal, vars];

We found convenient to allow typing in list terms represented in the usual *Mathematica* notation (i.e. by enumerating the list elements between curly braces), but before passing these terms to the CFLP calculus, they are converted in the CFLP list representation. This conversion is carried out by the function `ListToCons`.

```
ListToCons[{}]:=•;
ListToCons[h_,t___]:=Cons[ListToCons[h],ListToCons[{t}]];
ListToCons[λ[l_,t_]]:=λ[l,ListToCons[t]];
ListToCons[t_?AtomQ]:=t;
ListToCons[t_]:=Map[ListToCons,t];
```

Chapter 3

The Type Checker

The CFLP interpreter has a built-in polymorphic type checker for verifying the type correctness of goals and conditional rewrite rules. It is based on the Hindley-Milner type system and acts on an extended set of terms

$$\mathtt{ExtendedTerm} ::= \mathtt{Term} \mid \mathtt{InitialGoal} \mid \mathtt{RwRule}$$

NOTATION.

Having specified the set $\mathcal{V}$ of variables and the set $\mathcal{F}$ of function symbols, we denote by $\mathcal{F}_{ext}$ the set $\mathcal{F} \cup \{\cdot \approx \cdot, \cdot \longrightarrow \cdot \Leftarrow \cdot\}$, by $\mathcal{R}[\mathcal{F}, \mathcal{V}]$ the set of expressions of `RwRule`, and by $\mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ the set of expressions of `ExtendedTerm`.

The new function symbols added to $\mathcal{F}$ are assumed to have the following types

	type
$\approx$	$\mathtt{Bool} \times \mathtt{Bool} \rightarrow \mathtt{Bool}$
$\cdot \longrightarrow \cdot \Leftarrow \cdot$	$\alpha \times \alpha \times \langle \mathtt{Bool} \rangle \rightarrow \mathtt{Bool}$

where $\alpha \in \mathcal{V}_{type}$.

The problem of type checking an extended term is reduced to the equivalent problem of looking for the existence of a solution of a system S of equations between type expressions. The inference rules of the type checker illustrate the way this system is generated. Finally, the existence of a solution for the S is decided by a `TSolve` call on the goal S and with type checking turned off.

In the next section we give the inference rules of the type checker. The last section refers to the *Mathematica* implementation.

3.1 Polymorphic Type Inference

In this section we use the notation $t{:}\tau$ to indicate that $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ has the polymorphic type $\tau \in \mathcal{T}\mathrm{ype}[\mathcal{V}_{type}]$. The type inference rules are presented by means of sequents of the form

$$\sigma \vdash t{:}\sigma', g$$

where σ, σ' are list of type assignments of the form $\{s_1{:}\tau_1, \ldots, s_n{:}\tau_n\}$ with $s_i \in \mathcal{V}_{type} \cup \mathcal{F}$ and $\tau_i \in \mathcal{T}\mathrm{ype}[\mathcal{V}_{type}]$ for $1 \leq i \leq n$ $(n \geq 0)$, and $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$, $g \in \mathcal{G}\mathrm{oal}[\mathcal{F}_{type}, \mathcal{V}_{type}]$. We denote by $\cup$ the appending operation for lists.

The type inference rules of the CFLP interpreter are

- for variables $x \in \mathcal{V}$
$$\frac{\sigma \vdash \sigma', g \cup \{\tau \approx \tau'\}}{\sigma \vdash \{x{:}\tau\} \cup \sigma', g} \text{ if } x{::}\tau' \in \sigma$$
- for literals $c \in \mathcal{C}_{\tau'}$
$$\frac{\sigma \vdash \sigma', g \cup \{\tau' \approx \tau\}}{\sigma \vdash \{c{:}\tau\} \cup \sigma', g}$$
- for function symbols $f \in \mathcal{F}$
$$\frac{\sigma \vdash \sigma', g \cup \{\tau'' \approx \tau'\}}{\sigma \vdash \{f{:}\tau\} \cup \sigma', g}$$
  if $f{::}\forall\alpha_1 \ldots \alpha_n.\tau' \in \sigma$ and $\tau'' = \tau'\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}$ with $\alpha'_1, \ldots, \alpha'_n \in \mathcal{V}_{type}$ new variables.
- for Λ-terms $\lambda[\{x_1, \ldots, x_n\}, t]$
$$\frac{\sigma \cup \{x'_1{:}\alpha_1, \ldots, x'_n{:}\alpha_n\} \vdash \{t'{:}\alpha_1 \times \cdots \times \alpha_n \to \alpha\} \cup \sigma', g \cup \{\alpha_1 \times \cdots \times \alpha_n \to \alpha \approx \tau\}}{\sigma \vdash \{\lambda[\{x_1, \ldots, x_n\}, t]{:}\tau\} \cup \sigma', g}$$
  where $x'_1, \ldots, x'_n \in \mathcal{V}$, $\alpha_1, \ldots, \alpha_n \in \mathcal{V}_{type}$ are fresh variables, and $t' = t\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$
- for applicative terms $t[t_1, \ldots, t_n]$
$$\frac{\sigma \vdash \{t_1{:}\alpha_1, \ldots, t_n{:}\alpha_n, t{:}\alpha_1 \times \cdots \times \alpha_n \to \tau\} \cup \sigma', g}{\{\sigma \vdash t[t_1, \ldots, t_n]{:}\tau\} \cup \sigma', g}$$
  where $\alpha_1, \ldots, \alpha_n \in \mathcal{V}_{type}$ are fresh variables.
- for final sequents
$$\sigma \vdash \{\}, g$$
  if the system $g \in \mathcal{IG}\mathrm{oal}[\mathcal{F}, \mathcal{V}][\mathcal{F}_{type}, \mathcal{V}_{type}]$ has a solution in the algebra of type expressions.

3.1.1 Implementation

The type checker is implemented as a *Mathematica* function with the syntax call

`Tc`[t];

for type-checking an extended term $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$. This call returns the most general instance for the type of t if type checking succeeds, and "$\perp$" otherwise.

`TcX` is an auxiliary function of `Tc` and `TSolve`. It has the syntax call

`TcX`$[term, type, teqs, tvars]$

where $term$ is a CFLP term, $type$ is a type expression, $teqs$ is a list of type sharing equalities, and $tvars$ the list of type variables of $teqs$. The call returns a pair $\{teqs', tvars'\}$ where $teqs' = teqs \cup teqs''$, $vars' = vars \cup vars''$ such that the following two assertions are equivalent

- $term$ has type $type$ and the type sharing equations $teqs$ in variables $tvars$ hold
- the type sharing equations $teqs'$ in variables $tvars'$ hold.

The implementation of TcX is straightforward. The function calls `NewTyVar`[] are used to generate fresh type variables.

(* variables *)
TcX[x_?IsVar,τ_, τeqn,αvar_]:=
 If[IsTyped[x],
  Type[x][]/.{αv_, $\tau 1$_} :> {Append[τeqn, eq[$\tau 1$, τ]],
  $\alpha v \cup \alpha var$,
  Type[x]=FormTypeExpr[{},τ];
  {τeqn, αvar}];
(* constant literals *)
TcX[_Real | _Integer,τ_, τeqn_, αvar_]:=
  {Append[τeqn, eq[Real,τ]],αvar};
(* function symbol or symbol literal *)
TcX[x_Symbol, τ_, τeqn_, αvar_] :=
 Block[{$\tau expr$, αv = {}},
  {Append[τeqn, eq[If[IsTyped[x],{αv, $\tau expr$}=Type[x][ ]; $\tau expr$, Symbol], τ]],
   $\alpha var \cup \alpha v$}
  ];
(* λ-terms *)
TcX[λ[{___, x_, ___, x_, ___}, t_], τ_, τeqn_, αvar_] := {{eq[Int,Bool]}, {}};
TcX[λ[x_Symbol, t_], τ_, τeqn_, αvar_]:= TcX[λ[{x}, t], τ, τeqn, αvar];

```
TcX[λ[xs_List, t_], τ_, τeqn_, αvar_] :=
Block[{xsNew, tNew, β, newTyVars},
  {xsNew, tNew} =(Module[#1, {#1, #2}] & )[xs,t];
  Scan[(Type[#1] = FormTypeExpr[{}, NewTyVar[]]) & ,xsNew];
  newTyVars=(Last[Type[#1][]]& ) /@ xsNew;
  β =If[Length[newTyVars] > 1,
   FuncType[Cross @@ newTyVars,NewTyVar[]],
   FuncType[First[newTyVars],NewTyVar[]]
  ];
  TcX[tNew, Last[β], Append[τeqn, eq[β, τ]],
   Append[Union[αvar, newTyVars], Last[β]]]
];
(* lists *)
TcX[l_List, τ_, τeqn_αvar_]:=
Function[TcXList[l, Table[#, {Length[l]}],
Append[τeqn, eq[τ, TyList[#]]], Append[αvar, #]]][NewTyVar[]];
(*application for flattened functions *)
TcX[Plus[x_, y_] |Times[x_, y_],τ_, τeqn_, αvar_]:=
TcXList[{x, y}, {Real,Real}, Append[τeqn, eqτ, Real]], αvar];
TcX[And[x_, y_] | mapstOr[x_, y_], τ_, τeqn_, αvar_]:=
 TcXList[{x, y}, {Bool,Bool}, Append[τeqn, eq[τ,Bool]], αvar];
(* application *)
TcX[f_[], τ_τeqn_αvar_]:=
 TcX[f, FuncType[□, τ], τeqn, αvar];
TcX[f_[arg_], τ_τeqn_, αvar_]:=
 Function[
  TcXList[{f, arg}, {FuncType[#1, τ], #1}, τeqn,
   Append[αvar, #1]]][NewTyVar[ ]];
TcX[f_[args__], τ_, τeqn_, αvar_]:=
 Function[
  TcXList[f,args,
   {FuncType[Apply[Cross ,#1], τ],
    Apply[Sequence,#1]}, τeqn, Join[αvar, #1]]] [NewTyVar/@{args}];
TcX[_, _, _, _]:={{eq[Int,Bool]}, { }};

TcXList[{}, {}, τeqn_, αvar_] :={τeqn, αvar};
TcXList[{t_, ts___}, α_αv___, τeqn_, αvar_]:=
 TcXList[{ts}, {αv}, Sequence @@ TcX[t, α, τeqn, αvar]];
TcXList[_, _, _, _]:={{eq[Int,Bool]}, {}};
```

Note the special treatment for the operators `Plus`, `Times`, `And` and `Or`. The reason is that these operators are flattened and as a consequence there is no correspondence between their arity and their admissible number of operands. We treat these operators as binary and make use of the fact that the variable

bindings performed by a pattern $f[u\_,v\_]$ to a term $f[a_1, \ldots, a_n]$ (where $f \in \{$Plus, Times, And, Or$\}$) are $u \mapsto a_1, v \mapsto f[a_2, \ldots, a_n]$.
In the current version we assimilate the integer constants to the Real type. This is not a problem when all the calculus is performed in the functional logic programming frame. It also does not confuse the constraint solvers, since the only constraint solvers consdered by now are for solving constraints over reals.

```
Tc[t_] :=
 Module[{sol, teqn, tvar, saveType =DownValues[Type],β =NewTyVar[], result},
  {teqn, tvar} =TcX[t, β, {}, {β}];
  DownValues[Type]=saveType;
  SetVariables[tvar];
  sol= Block[{Global`AnswerList={},
hskip 2em Global`Substitution={},
     Global`goalVar={β},
     Global`NSolutions=2},
       CFLP[teqn,];
       Global`AnswerList];
  ClearVariables[tvar];
  If[sol =! = {}, β//.sol[[1]],"⊤"
 ];
```

Usually, the user invokes the type checker only indirectly, either by calling `Def` with option argument `TypeCheck -> True` (see Subsection 2.5.2) or `TSolve` with `TypeCheck -> True` (see Subsection 2.7.1).

Chapter 4

Lazy Narrowing Calculi

This chapter is intended to present the narrowing calculi that were considered in the design of the CFLP calculus. Mathematica implementations are already available, see [3] for a review. The organization of this chapter is as follows. In section 4.1 we introduce the first-order lazy narrowing calculus called LNC as our base calculus. Section 4.2 presents the calculus HLNC together with the completeness theorem. In Section 4.3 we introduce the concept of *constraint* and extend the HLNC calculus to operate on constraints. Section 4.4 introduces new extensions to the calculus by adding constructs to support AND parallelism. The last section refers to the *Mathematica* implementation of this final calculus.

4.1 Preliminaries

A *term substitution* is a mapping $\theta : \mathcal{V} \to \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ with finite support, i.e. the set $\mathrm{supp}[\theta] := \{x \in \mathcal{V} \mid \theta[x] \neq x\}$ is finite. The restriction of θ to a set $V \subseteq \mathcal{V}$ is the substitution $\theta|_V$ defined by

$$x\theta|_V = \begin{cases} x & \text{if } x \notin V, \\ x\theta & \text{if } x \in V. \end{cases}$$

We write $\theta_1 \equiv \theta_2[V]$ for the situation when $\theta_1, \theta_2 \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ and $V \subseteq \mathcal{V}$ such that $\theta_1|_V \equiv \theta_2|_V$. Like for type substitutions, we extend θ to denote its homomorphic extension on $\mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$, and write $t\theta$ for $\theta[t]$ when $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$. Formally,

$$t\theta = \begin{cases} x\theta & \text{if } t \equiv x \in \mathcal{V}, \\ c & \text{if } t \equiv c \in \mathcal{C}, \\ t_0\theta[t_1\theta, \ldots, t_n\theta] & \text{if } t \equiv t_0[t_1, \ldots, t_n], \\ \lambda[\{x_1, \ldots, x_n\}, t\theta_{|(\mathrm{supp}[\theta] - \{x_1, \ldots, x_n\})}] & \text{if } t \equiv \lambda[\{x_1, \ldots, x_n\}, t]. \end{cases}$$

Also, we sometimes write $\{x_1 \mapsto x_1\theta, \ldots, x_n \mapsto x_n\theta\}$ for θ with $\mathrm{supp}[\theta] = \{x_1, \ldots, x_n\}$. $\mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ denotes the set of term substitutions. A *variable renaming* is a bijective substitution from $\mathcal{V}$ to $\mathcal{V}$. $t' \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ is a *variant* of an extended term $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ if $t' = t\theta$ where θ is a variable renaming. A fresh variant of $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ is a variant $t' \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ such that $\mathcal{V}[t] \cap \mathcal{V}[t'] = \emptyset$.

Let $\theta_1, \ldots, \theta_n \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$. Then $\theta_1 \cdots \theta_n \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ denotes the substitution defined by $x\theta_1 \cdots \theta_n = (\cdots(x\theta_1)\cdots\theta_n)$. $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ is *normalized* if $x\theta\theta \equiv x\theta$ holds for all $x \in \mathcal{V}$.

We define $\theta_1 \leq \theta_2[V]$ if there exists $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ such that $\theta\theta_1 \equiv \theta_2[V]$, and $\theta_1 \leq \theta_2$ if $\theta_1 \leq \theta_2[\mathcal{V}]$.

Let $s, t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ and $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$. θ is called a *unifier* of s and t if $s\theta \equiv t\theta$. It is well-known that unifiable terms posses a *most general unifier* $\sigma \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$, i.e. σ is a unifier of s and t and $\sigma \leq \tau$ for every other unifier θ of s and t. Most general unifiers are unique up to variable renaming; we denote a most general unifier of two terms $s, t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ by $\mathrm{mgu}[s, t]$.

We introduce now the notion of *position*, which is a precise formalism for describing subterm occurrences. The set $\mathcal{O}[t]$ of positions in $t \in \mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ is inductively defined as follows

$$\mathcal{O}[t] = \begin{cases} \{\varepsilon\} & \text{if } t \in \mathcal{V}, \\ \{\varepsilon\} \cup \{i \cdot p \mid 1 \leq i \leq p \text{ and } p \in \mathcal{O}[t_i]\} & \text{if } t \equiv f[t_1, \ldots, t_n] \end{cases}$$

So positions are sequences of natural numbers denoting subterm occurrences. If $p \in \mathcal{O}[t]$ then $t_{|p}$ denotes the *subterm* of t at position p, i.e.

$$t_{|p} = \begin{cases} t & \text{if } p = \varepsilon \\ (t_i)_{|q} & \text{if } t \equiv f[t_1, \ldots, t_n] \text{ and } p = i \cdot q \end{cases}$$

The set $\mathcal{O}[t]$ is partitioned into $\overline{\mathcal{O}}[t]$ and $\mathcal{O}_{\mathcal{V}}[t]$ as follows: $\overline{\mathcal{O}}[t] = \{p \in \mathcal{O}[t] \mid t_{|p} \notin \mathcal{V}\}$ and $\mathcal{O}_{\mathcal{V}}[t] = \{p \in \mathcal{O}[t] \mid t_{|p} \in \mathcal{V}\}$. If $s, t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ and $\in \mathcal{O}[t]$ then $t[s]_p$ denotes the term that is obtained from t by replacing the subterm at position p by the term s. Formally

$$t[s]_p = \begin{cases} s & \text{if } p = \varepsilon, \\ f[t_1, \ldots, t_i[s]_q, \ldots, t_n] & \text{if } t \equiv f[t_1, \ldots, t_n] \text{ and } p = i \cdot q. \end{cases}$$

Let now $\mathcal{R}$ be a term rewriting system and $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$. t is a *constructed term* if $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c, \mathcal{V}]$. $\mathcal{R}$ is a *constructed system* (CS for short) if every rewrite rule $R \in \mathcal{R}$ is of the form $f[t_1, \ldots, t_n] \longrightarrow r$ with $f \in \mathcal{F}_d$, $r \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$, $t_1, \ldots, t_n \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c, \mathcal{V}]$. t is a *linear* term if it has no multiple occurrences of the same variable. $\mathcal{R}$ is a *left (right) linear* term rewriting system if every $R \in \mathcal{R}$ is of the form $l \to r$ with l linear (r linear).

The *rewrite relation* $\longrightarrow_{\mathcal{R}}$ associated with $\mathcal{R}$ is defined on $\mathcal{T}\mathrm{erm}_{ext}[\mathcal{F}_{ext}, \mathcal{V}]$ as follows: $s \longrightarrow_{\mathcal{R}} t$ if there exist a variant $l \longrightarrow r$ of a rewrite rule, $p \in \mathcal{O}[s]$, and $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ such that $s_{|p} \equiv l\theta$ and $t \equiv s[r\theta]_p$. We call $s \longrightarrow_{\mathcal{R}} t$ a *rewrite step*. The transitive closure of $\longrightarrow_{\mathcal{R}}$ is denoted by $\longrightarrow^{+}_{\mathcal{R}}$, the reflexive-transitive closure of $\longrightarrow_{\mathcal{R}}$ by $\longrightarrow^{*}_{\mathcal{R}}$, and the reflexive, symmetric-transitive closure of $\longrightarrow_{\mathcal{R}}$ by $\longleftrightarrow^{*}_{\mathcal{R}}$.

$\mathcal{R}$ is called *terminating* if there is no infinite rewriting sequence

$$t_1 \longrightarrow_{\mathcal{R}} t_2 \longrightarrow_{\mathcal{R}} \cdots$$

A *critical pair* of $\mathcal{R}$ is a pair of terms $((l_1[r_2]_p)\sigma, r_1\sigma)$ where $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ are variants of rewrite rules of $\mathcal{R}$ without common variables such that $p \in \overline{\mathcal{O}}[l_1]$, $(l_1)_{|p}$ and l_2 are unifiable, $\sigma = \mathrm{mgu}[(l_1)_{|p}, l_2]$; if $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ are variants, we do not consider the case $p = \varepsilon$.
$\mathcal{R}$ is called *non-ambiguous* or *non-overlapping* if it has no critical pairs. $\mathcal{R}$ is *orthogonal* if it is left linear and non-ambiguous. $\mathcal{R}$ is *confluent* if for all terms $s, t_1, t_2 \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ with $s \longrightarrow^{*}_{\mathcal{R}} t_1$ and $s \longrightarrow^{*}_{\mathcal{R}} t_2$ there exists $t_3 \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ such that $t_1 \longrightarrow^{*}_{\mathcal{R}} t_3$ and $t_2 \longrightarrow^{*}_{\mathcal{R}} t_3$. $\mathcal{R}$ is *complete* if it is *terminating* and *confluent*.

We denote the transitive closure of $\longrightarrow_{\mathcal{R}}$ by $\longrightarrow^{+}_{\mathcal{R}}$, the reflexive-transitive closure by $\longrightarrow^{*}_{\mathcal{R}}$, and the reflexive, symmetric and transitive closure by $\longleftrightarrow^{*}_{\mathcal{R}}$.

Let $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$. θ is a *solution* of an equation $s \approx t$ ($s \asymp t$) with respect to $\mathcal{R}$ if $s\theta \longleftrightarrow^{*}_{\mathcal{R}} t\theta$. θ is a solution of a goal $G \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$ if θ is a solution for every equation in G.

4.2 The First-Order Lazy Narrowing Calculus

This calculus is a specialization of Hölldobler's calculus TRANS [5], which is defined for general equational systems and is based on paramodulation, to (confluent) term rewriting systems and narrowing. It has been extensively studied in Middeldorp, Okui and Ida [9] and Middeldorp and Okui [8]. During this Section we restrict the syntax domains of terms and conditional rewrite rules by imposing the following syntax rules for `Term` and `RwRule`:

$$\begin{array}{rcl}
\mathtt{Term} & ::= & \mathtt{FunctionSymbol}[\mathtt{Term}, \ldots, \mathtt{Term}] \mid \mathtt{Literal} \mid \mathtt{Variable} \\
\mathtt{FunctionSymbol} & ::= & \mathtt{ConstructorSymbol} \mid \mathtt{DefinedFunctionSymbol} \\
\mathtt{RwRule} & ::= & \mathtt{Term} \longrightarrow \mathtt{Term}
\end{array}$$

LNC is defined over well-typed goals with respect to well typed systems of rewrite rules.

In the sequel we make use of the following notations

e, e', e_i $(i > 0)$	equations
$E, E', E_1, E_2 \vert$	sequences of equations
$\mathcal{R}$	a (well typed) set of rewrite rules
R	rewrite rule
$\theta, \theta', \theta_i$ $(i > 0)$	substitutions
f	defined function symbol
g	function symbol
c	constructor symbol
s, r, t, s_i, l_i, t_i $(i > 0)$	terms
x, y, x_i, y_i $(i > 0)$	variables
G, G', G_i $(i \geq 0)$	goals

The LNC calculus consists of the following five inference rules:

[o] *outermost narrowing*

$$\frac{\{E_1, f[s_1, \ldots, s_n] \simeq t, E_2\}}{\{E_1, s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, E_2\}}$$

if $f[s_1, \ldots, s_n] \longrightarrow r$ is a fresh variant of a rule $R \in \mathcal{R}$.

[i] *imitation*

$$\frac{\{E_1, f[s_1, \ldots, s_n] \simeq x, E_2\}}{\{E_1, s_1 \approx x_1, \ldots, s_n \approx x_n, E_2\}\theta}$$

if $\theta = \{x \mapsto f[x_1, \ldots, x_n]\}$ with $x_1, \ldots, x_n \in \mathcal{V}$ fresh.

[d] *decomposition*

$$\frac{\{E_1, f[s_1, \ldots, s_n] \approx f[t_1, \ldots, t_n], E_2\}}{\{E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2\}}$$

[v] *variable elimination*

$$\frac{\{E_1, s \simeq x, E_2\}}{\{E_1, E_2\}\theta}.$$

if $x \notin \mathcal{V}[s]$ and $\theta = \{x \mapsto s\}$.

[t] *removal of trivial equations*

$$\frac{\{E_1, t \approx t, E_2\}}{\{E_1, E_2\}}$$

The new equations $s_1 \approx l_1, \ldots, s_n \approx l_n$ generated by the outermost narrowing rule [**o**] are called *parameter passing equations.*

If G and G' are the upper and lower goal in the inference rule $[\alpha]$ ($\alpha \in \{\mathbf{o}, \mathbf{i}, \mathbf{d}, \mathbf{v}, \mathbf{t}\}$), we write $G \leadsto_{[\alpha]} G'$ or simply $G \leadsto G'$. This is called an LNC-step. The applied rewrite rule or substitution may be supplied as subscript, so we may write $G \leadsto_{[\alpha], l \longrightarrow r} G'$ or $G \leadsto_{[\alpha],\theta} G'$. A finite LNC-derivation $G_1 \leadsto_{\theta_1} \cdots \leadsto_{\theta_{n-1}} G_n$ may be abbreviated to $G_1 \leadsto^*_{\theta} G_n$ where $\theta = \theta_1 \cdots \theta_{n-1}$. An LNC-refutation of $G \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$ is a finite sequence of LNC-steps

$$G \equiv G_0 \leadsto_{\theta_1} G_1 \leadsto \cdots \leadsto_{\theta_{n-1}} G_n \equiv \{\}$$

An LNC-*solution* is a substitution $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ such that $G \leadsto^*_{\theta} \{\}$.

We note that rules [**d**], [**v**] and [**t**] constitute the syntactic unification algorithm of Martelli and Montanari [13].

The outermost narrowing inference rule [**o**] asserts that if $s_1 \approx l_1, \ldots, s_n \approx l_n$, and $f[l_1, \ldots, l_n] \longrightarrow r$ then $f[s_1, \ldots, s_n] \approx t$ holds. This is an obvious obvious consequence of the replacement and tranzitivity properties of euqtional theories.

The imitation rule [**i**] can be viewed as instance of [**o**] with respect to the fictive rule $f[x_1, \ldots, x_n] \longrightarrow x$, followed by n successive applications of the variable elimination rule.

Obviously, the calculus is correct, i.e. if $G \leadsto^*_{\theta} \{\}$ then θ represents a solution of the goal G in the equational theory defined by $\mathcal{R}$.

An important property of a calculus is *completeness*, i.e. the capability to find all the solutions of a goal. In [] it has been shown that LNC is complete for confluent term rewriting systems with respect to normalized substitutions. More precisely, the following result has been proved:

Theorem 1 (Completeness of LNC) *Let $\mathcal{R}$ be a confluent term rewriting system and G a goal. For every normalized solution θ of G there exists an LNC-refutation $G \leadsto^*_{\theta'} \{\}$ such that $\theta' \leq \theta[\mathcal{V}[G]]$.*

We note that there are three sources of nondeterminism of the LNC calculus:

(1) the choice of the equation in the current goal,

(2) the choice of the inference rule, and

(3) the choice of the rewrite rule (in the case of [**o**]).

In [9], Middeldorp *et al.* showed that these choices are *don't know non-deterministic*. This means that in general all possible choices have to be considered in order to guarantee completeness, resulting in a huge search space. In particular, LNC lacks *strong completeness*, i.e. completeness is not independent of the selection function of the goal equation.

In [9], Middeldorp, Okui and Ida showed that the first source of non-determinism can be removed, without losing the important completeness property, by using the selection function $\mathcal{S}_{left}$ that selects the leftmost equation in every goal.

Theorem 2 *Let $\mathcal{R}$ be a confluent term rewriting system and $G \in \mathcal{IG}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$. For every normalized solution θ of G there exists an LNC-refutation $G \rightsquigarrow^*_{\theta'} \{\}$ respecting $\mathcal{S}_{left}$ such that $\theta' \leq \theta[\mathcal{V}[G]]$.*

Therefore, from now on we consider only LNC-derivations with respect to the leftmost selection function.

Concerning the second kind of non-determinism, Middeldorp and Okui showed in [8] that it can also be removed without losing completeness by restricting the underlying term rewriting system to left-linear confluent constructor systems and interpreting two terms to be equal if and only if they both reduce to the same constructor normal form (i.e. *strict equality*). Formally, a *strict solution* of a goal is defined as follows

Definition 1 (strict solution) *Let $\mathcal{R}$ be a term rewriting system and G a goal. A substitution $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ is a* strict solution *of G if for every equation $s \approx t$ in G there exist a constructor term u such that $s \longrightarrow^*_{\mathcal{R}} u$ and $t \longrightarrow^*_{\mathcal{R}} u$.*

The deterministic version of the lazy narrowing calculus relies on the notion of *descendants* of LNC derivations. The selected equation $f[s_1, \ldots, s_n] \approx t$ or $t \approx f[s_1, \ldots, s_n]$ in the outermost narrowing rule **[o]** has the equation $r \approx t$ as only one-step descendant. In the imitation rule **[i]** all equations $s_i\theta \approx x_i$ $(1 \leq i \leq n)$ are one-step descendants of the selected equation $f[s_1, \ldots, s_n] \approx x$ (or $x \approx f[s_1, \ldots, s_n]$). The selected equation $f[s_1, \ldots, s_n] \approx f[t_1, \ldots, t_n]$ in the decomposition rule **[d]** has all equations $s_1 \approx t_1, \ldots, s_n \approx t_n$ as one-step descendants. Finally, the selected equations in **[v]** and **[t]** have no one-step descendants. One-step descendants of non-selected equations are defined as expected. Descendants are obtained from one-step descendants by reflexivity and transitivity. We observe that every equation in an LNC-derivation descends either from a parameter-passing equation or an equation in the initial goal.

The selection of the inference rule to be applied on the leftmost equation of a goal G depends on the property of that equation to be descendant of an initial equation, or descendant of a parameter passing equation. In fact, the following results have been obtained:

1. if $\mathcal{R}$ is a left linear confluent CS, then the LNC calculus remains complete if we adopt the following selection strategy for descendants of parameter-passing equations

root$[s]$ \ root$[t]$	$\mathcal{V}$	$\mathcal{F}_c$	$\mathcal{F}_d$
$\mathcal{V}$	$[v]$	$[v]$	$\times$
$\mathcal{F}_c$	$[v]$	$[d]$	$\times$
$\mathcal{F}_d$	$[v]$	$[o]$	$\times$

2. if $\mathcal{R}$ is a confluent term rewriting system then LNC remains complete with respect to strict solutions if we adopt the following selection strategy for descendants of initial equations

root$[s]$ \ root$[t]$	$\mathcal{V}$	$\mathcal{F}_c$	$\mathcal{F}_d$
$\mathcal{V}$	$[v]/[t]^a$	$[v]/[i]^b$	$[o]$
$\mathcal{F}_c$	$[v]/[i]^c$	$[d]$	$[o]$
$\mathcal{F}_d$	$[o]$	$[o]$	$[o]^d$

[a] is applied iff $s \equiv t$
[b] is applied iff $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c, \mathcal{V}]$ and $s \notin \mathcal{V}[t]$
[c] is applied iff $s \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c, \mathcal{V}]$ and $t \notin \mathcal{V}[s]$
[d] is applied to the left-hand side s.

In the sequel we use $\asymp$ to denote equality in descendants of parameter passing equations and $\approx$ for eqlaity in descendants of initial equations; $\simeq$ will denote the symmetric closure of the operator $\approx$ and $\underline{\smile}$ the symmetric closure of $\asymp$.

The first result has first been proved by Middeldorp *et al.* in [9] for the case of orthogonal term rewriting systems, and extended in [8] to the case of left-linear confluent constructor systems. It is an answer to the *eager elimination problem* which states that applying the variable elimination rule **[v]** prior to other applicable inference rules eliminates many redundant derivations.

Incorporating strategy 1 into LNC gives rise to the calculus LNC$_{pp}$ which is deterministic in the choice of inference rule for parameter-passing equations, provided we are dealing with linear confluent CSs. The inference rules are given below. The first group of inference rules is designed for descendants of initial equations. The only difference to the initial LNC inference rules is that we select the leftmost equation and mark parameter-passing equations by using $\asymp$ instead of $\approx$ as equality operator.

[o] *outermost narrowing*

$$\frac{\{f[s_1, \ldots, s_n] \simeq t, E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, E\}}$$

if $f[s_1, \ldots, s_n] \longrightarrow r$ is a fresh variant of a rule $R \in \mathcal{R}$.

[i] *imitation*

$$\frac{\{E_1, f[s_1, \ldots, s_n] \simeq x, E_2\}}{\{E_1, s_1 \approx x_1, \ldots, s_n \approx x_n, E_2\}\theta}$$

if $\theta = \{x \mapsto f[x_1, \ldots, x_n]\}$ with $x_1, \ldots, x_n \in \mathcal{V}$ fresh.

[d] *decomposition*

$$\frac{\{E_1, f[s_1, \ldots, s_n] \approx f[t_1, \ldots, t_n], E_2\}}{\{E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2\}}$$

[v] *variable elimination*

$$\frac{\{E_1, s \simeq x, E_2\}}{\{E_1, E_2\}\theta}$$

if $x \notin \mathcal{V}[s]$ and $\theta = \{x \mapsto s\}$.

[t] *removal of trivial equations*

$$\frac{\{t \approx t, E\}}{\{E\}}$$

The second group of inference rules deals with descendants of parametric passing equations.

$[\mathbf{o}]_{\asymp}$ *outermost narrowing for parameter-passing equations*

$$\frac{\{f[s_1, \ldots, s_n] \asymp t, E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \asymp t, E\}} \quad \frac{\{t \asymp f[s_1, \ldots, s_n], E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \asymp t, E\}} \ \mathrm{root}[t] \notin \mathcal{F}_d$$

if $t \notin \mathcal{V}$ and $f[s_1, \ldots, s_n] \longrightarrow r$ is a fresh variant of a rule $R \in \mathcal{R}$.

$[\mathbf{d}]_{\asymp}$ *decomposition for parameter-passing equations*

$$\frac{\{c[s_1, \ldots, s_n] \asymp c[t_1, \ldots, t_n], E\}}{\{s_1 \asymp t_1, \ldots, s_n \asymp t_n, E\}}$$

$[\mathbf{v}]_{\asymp}$ *variable elimination for parameter-passing equations*

$$\frac{\{s \asymp x, E\}}{\{E\}\{x \mapsto s\}} \quad \frac{\{x \asymp s, E\}}{\{E\}\{x \mapsto s\}} \ s \notin \mathcal{V}$$

Theorem 3 *Let $\mathcal{R}$ be a left-linear confluent CS and G a goal. For every normalized solution θ of G there exists an* LNC_{pp}*-refutation $G \leadsto^*_{\theta'} \{\}$ such that $\theta' \leq \theta[\mathcal{V}[G]]$.*

Incorporating strategy 2 in LNC we obtain the version LNC_i which is deterministic in the choice of inference rules for descendants of initial equations, provided we are dealing with confluent term rewriting systems and are looking for strict solutions. Before giving the inference rules of LNC_i we mention the following definition and result which is used in the formulation of the imitation rule $\texttt{[i]}_{\approx}$.

Definition 2 *Let $t \in \mathcal{T}\text{erm}[\mathcal{F}, \mathcal{V}]$. We define the subset $\mathcal{V}_c[t]$ of $\mathcal{V}[t]$ inductively as follows:*

$$\mathcal{V}_c[t] = \begin{cases} \{t\} & \textit{if } t \in \mathcal{V} \\ \bigcup_{i=1}^{n} \mathcal{V}_c[t_i] & \textit{if } t \equiv c[t_1, \ldots, t_n] \textit{ with } c \in \mathcal{F}_c \\ \emptyset & \textit{otherwise} \end{cases}$$

Lemma 1 *Let $\mathcal{R}$ be a confluent TRS. an equation $x \approx t$ (or $t \approx x$) with $x \not\equiv t$ has not strict solution if $x \notin \mathcal{V}_c[t]$.*

The first group of rules is designed for descendants of initial equations.

$\textbf{[o]}_{\approx}$ *outermost narrowing for initial equations*

$$\frac{\{f[s_1, \ldots, s_n] \approx t, E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, E\}} \quad \frac{\{t \approx f[s_1, \ldots, s_n], E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, E\}} \ \text{root}[t] \notin \mathcal{F}_d$$

if $f[s_1, \ldots, s_n] \longrightarrow r$ is a fresh variant of a rule $R \in \mathcal{R}$.

$\textbf{[i]}_{\approx}$ *imitation for initial equations*

$$\frac{\{c[s_1, \ldots, s_n] \simeq x, E\}}{\{s_1 \approx x_1, \ldots, s_n \approx x_n, E\}\theta}$$

if $x \notin \mathcal{V}_c[c[s_1, \ldots, s_n]]$, $f[s_1, \ldots, s_n] \notin \mathcal{T}\text{erm}[\mathcal{F}_c, \mathcal{V}]$, $\theta = \{x \mapsto f[x_1, \ldots, x_n]\}$ with $x_1, \ldots, x_n \in \mathcal{V}$ fresh variables.

$\textbf{[d]}_{\approx}$ *decomposition for initial equations*

$$\frac{\{c[s_1, \ldots, s_n] \approx c[t_1, \ldots, t_n], E\}}{\{s_1 \approx t_1, \ldots, s_n \approx t_n, E\}}$$

$\textbf{[v]}_{\approx}$ *variable elimination for initial equations*

$$\frac{\{t \simeq x, E\}}{\{E\}\{x \mapsto t\}} \quad t \in \mathcal{T}\text{erm}[\mathcal{F}_c, \mathcal{V}] \text{ and } x \notin \mathcal{V}[t].$$

$\textbf{[t]}_{\approx}$ *removal of trivial equations for initial equations*

$$\frac{\{x \approx x, E\}}{\{E\}}$$

$[\mathbf{o}]_{\asymp}$ *outermost narrowing for parameter-passing equations*

$$\frac{\{f[s_1,\ldots,s_n] \underline{\smile}\, t, E\}}{\{s_1 \asymp l_1,\ldots,s_n \asymp l_n, r \asymp t, E\}}$$

if $f[l_1,\ldots,l_n] \longrightarrow r$ is a fresh variant of a rule $R \in \mathcal{R}$.

$[\mathbf{i}]_{\asymp}$ *imitation for parameter-passing equations*

$$\frac{\{f[s_1,\ldots,s_n] \underline{\smile}\, x, E\}}{\{s_1 \asymp x_1,\ldots,s_n \asymp x_n, E\}\theta}$$

if $\theta = \{x \mapsto f[x_1,\ldots,x_n]\}$ with $x_1,\ldots,x_n \in \mathcal{V}$ fresh.

$[\mathbf{d}]_{\asymp}$ *decomposition for parameter-passing equations*

$$\frac{\{c[s_1,\ldots,s_n] \asymp c[t_1,\ldots,t_n], E\}}{\{s_1 \asymp t_1,\ldots,s_n \asymp t_n, E\}}$$

$[\mathbf{v}]_{\asymp}$ *variable elimination for parameter-passing equations*

$$\frac{\{s \simeq x, E\}}{\{E\}\theta}$$

if $x \notin \mathcal{V}[s]$ and $\theta = \{x \mapsto s\}$.

$[\mathbf{t}]_{\asymp}$ *removal of trivial equations for parameter-passing equations*

$$\frac{\{t \approx t, E\}}{\{E\}}$$

Theorem 4 *Let $\mathcal{R}$ be a confluent term rewriting system and G a goal. For every strict and normalized solution θ of G there exists an* LNC$_i$*-refutation $G \rightsquigarrow^*_{\theta'} \{\}$ such that $\theta' \leq \theta[\mathcal{V}[G]]$.*

Combining the calculi LNC$_{pp}$ and LNC$_i$ yields a calculus, call it LNC$_d$, which is fully deterministic in the choice of inference rule for equations. The following completeness theorem holds for LNC$_d$:

Theorem 5 *Let $\mathcal{R}$ be a left-linear confluent constructor system and G a goal. For every strict equation θ of G there exists an* LNC$_d$*-refutation $G \rightsquigarrow^*_{\theta'} \{\}$ such that $\theta' \leq \theta[\mathcal{V}[G]]$.*

4.3 The Lazy Conditional Narrowing Calculus

The lazy conditional narrowing calculus (LCNC for short) is an extension of LNC to conditional term rewriting systems. This means that the syntax domain `Term` of terms is the same as for the lazy narrowing calculus, but `RwRule` is extended to express conditional rewrite rules.

$$\begin{array}{lcl}\texttt{RwRule} & ::= & \texttt{Term} \longrightarrow \texttt{Term} \\ & & \mid \texttt{Term} \longrightarrow \texttt{Term} \Leftarrow \texttt{InitialGoal}\end{array}$$

The declarative meaning of such conditional equations is inherited from standard logic programming: a conditional rewrite rule

$$\mathit{lhs} \longrightarrow \mathit{rhs} \Leftarrow \mathit{cond}$$

indicates that the directed equation $\mathit{lhs} \longrightarrow \mathit{rhs}$ must hold for each assignment that satisfies the conditions.

To use conditional rewrite rules for term rewriting, various rewrite relations have been proposed. The most popular relation is based on the requirement that a conditional rewrite rule can be applied only if all equations in the condition part have a rewrite proof [6]. Hence, for a CTRS $\mathcal{R}$, the rewrite relation $\longrightarrow_{\mathcal{R}}$ is defined as follows. Let

$$\mathit{lhs} \longrightarrow \mathit{rhs} \Leftarrow l_1 \approx r_1, \ldots, l_n \approx r_n$$

be a CRR, $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ and $p \in \overline{\mathcal{O}}[t]$. If there are $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$ with $t_{|p} = \mathit{lhs}\ \theta$ and $u_1, \ldots, u_n \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$ with $l_i \longrightarrow^*_{\mathcal{R}} u_i$ and $r_i \longrightarrow^*_{\mathcal{R}} u_i$ for $i = 1, \ldots, n$, then $t \longrightarrow t[\mathit{rhs}\ \theta]_p$. Note that this definition of conditional rewriting is recursive and therefore the rewrite relation is undecidable for arbitrary conditional systems of rewrite rules. In order to obtain a decidable rewrite relation, it is often required that $\mathcal{R}$ must fulfill strong restrictions like being decreasing or level confluent. Formally, we have the following definitions.

Definition 3 *A CTRS $\mathcal{R}$ is* decreasing *if there exists a well-founded ordering $\succ$ with the following three properties:*

- *$\succ$ contains $\longrightarrow_{\mathcal{R}}$,*
- *$\succ$ has the subterm property,*
- *$l\theta \succ s\theta$, $l\theta \succ t\theta$ for every $\mathit{lhs} \longrightarrow \mathit{rhs} \Leftarrow \mathit{cond} \in \mathcal{R}$, every $s \approx t$ in cond and every $\theta \in \mathcal{S}\mathrm{ubst}[\mathcal{V}_{type}, \mathcal{F}]$.*

Definition 4 *Let $\mathcal{R}$ be a CTRS. We inductively define the term rewriting systems $\mathcal{R}_n$ for $n \geq 0$ as follows:*

$$
\begin{array}{lcll}
\mathcal{R}_0 & = & \{x \approx x \longrightarrow \texttt{True}\} & \\
\mathcal{R}_{n+1} & = & \{lhs\,\sigma \longrightarrow rhs\,\sigma \mid & lhs \longrightarrow rhs \Leftarrow cond \in \mathcal{R} \\
 & & & \text{and } e\sigma \longrightarrow^*_{\mathcal{R}_n} \texttt{True} \text{ for all } e \in cond\}
\end{array}
$$

If $s,t \in \mathcal{T}\mathrm{erm}[\mathcal{F},\mathcal{V}]$, we define $s \longrightarrow_{\mathcal{R}} t$ if $s \longrightarrow_{\mathcal{R}_n} t$ for some $n \geq 0$. The minimum such n is called the depth *of the derivation $s \longrightarrow t$.*[1]

$\mathcal{R}$ is called level-confluent *if each $\mathcal{R}_n$ $(n \geq 0)$ is confluent. We call $\mathcal{R}$* level-complete *if each $\mathcal{R}_n$ $(n \geq 0)$ is complete.*

Definition 5 *A term substitution θ is a* solution *of an equation $s \approx t$ $(s \asymp t)$ with respect to $\mathcal{R}$ if $s\theta \longleftrightarrow^*_{\mathcal{R}} t\theta$. θ is a solution of a goal $G \in \mathcal{G}\mathrm{oal}[\mathcal{F},\mathcal{V}]$ if θ is a solution for every equation in G.*

When conditional rewrite rules are applied in narrowing derivations, it is also necessary to prove the conditional part by narrowing rather than rewriting. The idea is to add the equations in the condition part to the current goal whenever a conditional rewrite rule is applied in a narrowing step.

4.3.1 Inference Rules

The inference rules of LCNC are given below.

[o] *outermost narrowing*

$$\frac{\{E_1, f[s_1,\ldots,s_n] \simeq t, E_2\}}{\{E_1, s_1 \approx l_1,\ldots,s_n \approx l_n, r \approx t, e_1,\ldots,e_p, E_2\}}$$

if there exists a fresh variant $f[l_1,\ldots,l_n] \longrightarrow r \Leftarrow e_1 \wedge \cdots \wedge e_p$ of a CRR $R \in \mathcal{R}$,

[i] *imitation*

$$\frac{\{E_1, f[s_1,\ldots,s_n] \simeq x, E_2\}}{\{E_1, s_1 \approx x_1,\ldots,s_n \approx t_n, E_2\}\theta}$$

where $\theta = \{x \mapsto f[x_1,\ldots,x_n]\}$ with $x_1,\ldots,x_n \in \mathcal{V}$ fresh variables,

[d] *decomposition*

$$\frac{\{E_1, f[s_1,\ldots,s_n] \approx f[t_1,\ldots,t_n], E_2\}}{\{E_1, s_1 \approx t_1,\ldots,s_n \approx t_n E_2\}}$$

[1] Observe that $\mathcal{R}_n \subseteq \mathcal{R}_{n+1}$ for all $n \geq 0$.

[v] *variable elimination*

$$\frac{\{E_1, s \simeq x, E_2\}}{\{E_1, E_2\}\theta}$$

if $x \notin \mathcal{V}[s]$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{\{E_1, t \approx t, E_2\}}{\{E_1, E_2\}}.$$

The only difference to LNC is in the outermost narrowing inference rule: in LCNC we add the conditional part of the applied conditional rewrite rule to the new goal. We use the same notations as for LNC to denote inference steps and LCNC-refutations.

Remarks

In an attempt to extend the results concerning determinism from LNC to LCNC, Hamada and Middeldorp proved in [10] that LCNC is strongly complete whenever basic conditional narrowing is complete. Since basic conditional narrowing is known to be complete for decreasing and confluent CTRSs (Middeldorp and Hamoen [7]), terminating and level-confluent 2-CTRSs (Giovanetti and Moiso [12]), and terminating and shallow-confluent 3-CTRSs, we obtain the strong completeness of LCNC for these three classes of CTRSs. Recently, Hamada [2] showed that LCNC complete for confluent (not necessarily terminating) 1-CTRSs.

4.4 HLNC: a Higher-Order Lazy Narrowing Calculus

One possibility for realizing a higher-order functional logic programming is to develop a higher-order narrowing with higher-order rewrite systems. Prehofer's calculus LN (see [14]) is a higher-order narrowing narrowing calculus which successfully reduces some degree of non-determinism, but still a more efficient higher order narrowing calculus is desired for a functional logic programming language.

Another possibility is to develop a higher-order narrowing based on the first-order narrowing in such a way that we can exploit the efficiency results (strong completeness and deterministic version) of first-order narrowing. This approach was followed by Suzuki, Nakagawa and Ida in the development of the calculus

HLNC. HLNC (cf. [11]) is based on the first-order lazy narrowing calculus LNC described above.

Before specifying the restrictions HLNC is subjected to, we introduce some useful notions.

Terminology

The *α-conversion* (or *renaming*) relation on terms is defined by the following relation scheme

$$\lambda[\{x_1, \ldots, x_n\}, t] \longrightarrow \lambda[\{y_1, \ldots, y_n\}, t\theta]$$

where $\theta = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ is a variable renaming, are $y_1, \ldots, y_n \in \mathcal{V}$ are not free or bound in t.

The β-reduction is given by the relation induced by the following rewrite rule scheme

$$\mathcal{R}_\beta = \{\lambda[\{x_1, \ldots, x_n\}, t][y_1, \ldots, y_n] \longrightarrow t\theta\}$$

where $\theta = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ and $y_i \notin \mathcal{V}[\lambda[\{x_1, \ldots, x_n\}, t]]$ $(1 \leq i \leq n)$.
A *flex term* is a term of the form $x[t_1, \ldots, t_n]$ where $x \in \mathcal{V}$ is free. A β-free term is a term that will never be β-reducible whatsoever substitutions are applied to it. The *η-conversion* is defined by the following relation scheme

$$\lambda[\{x_1, \ldots, x_n\}, t[x_1, \ldots, x_n]] \longrightarrow t$$

if $x_i \notin \mathcal{V}[t]$ $(1 \leq i \leq n)$.
A *long-η-normal form* is an element of the set `ETerm` generated by:

$$\begin{array}{lll} \texttt{ETerm} & ::= & \texttt{DefinedFunctionSymbol}[\texttt{ETerm}, \ldots, \texttt{ETerm}] \\ & & \mid \texttt{Constructor}[\texttt{ETerm}, \ldots, \texttt{ETerm}] \\ & & \texttt{Variable}[\texttt{ETerm}, \ldots, \texttt{ETerm}] \\ & & \lambda[\texttt{VarList}, \texttt{ETerm}] \\ & & \lambda[\texttt{VarList}, \texttt{ETerm}][\texttt{ETerm}, \ldots, \texttt{ETerm}] \end{array}$$

and subject to the following conditions:

- all terms are well typed,
- terms of the form $g[t_1, \ldots, t_n]$ and $x[t_1, \ldots, t_n]$ are of basic type,
- the term t in $\lambda[x_1, \ldots, x_n, t]$ is of basic type and is not of the form $x[x_1, \ldots, x_n]$.

Also, we assume that α-conversion of terms is performed implicitly. This means that two α-equivalent terms are indistinguishable, e.g. $\lambda[\{x, y\}, h[f[x, y], y]]$ and $\lambda[\{a, b\}, h[f[a, b], b]]$.

REMARK.

The definition of long η-normal form excludes abstractions that are η-equivalent to a higher-order variable: a λ-abstraction $\lambda[\{x_1, \ldots, x_n\}, x[x_1, \ldots, x_n]]$ where $x, x_1, \ldots, x_n \in \mathcal{V}$ is written only as x.

4.4.1 Syntax Restrictions

We keep the following syntax restrictions on function symbols and conditional rewrite rules

$$\begin{array}{rcl} \texttt{FunctionSymbol} & ::= & \texttt{Constructor} \\ & & \mid \texttt{DefinedFunctionSymbol} \\ \texttt{RwRule} & ::= & \texttt{DefinedFunctionSymbol}[\texttt{Term}, \ldots, \texttt{Term}] \longrightarrow \texttt{Term} \end{array}$$

In addition, we impose the restriction to have a TRS_λ term rewriting system, i.e. a set $\mathcal{R}$ of well typed rewrite rules of the form $lhs \longrightarrow rhs$ that satisfy the following conditions:

- lhs and rhs are of basic type,
- $\mathcal{V}[rhs] \subseteq \mathcal{V}[lhs]$

Given a TRS_λ $\mathcal{R}$ and $t \in \mathcal{T}\mathrm{erm}[\mathcal{F}, \mathcal{V}]$, we call t a long $\eta\mathcal{R}\beta$-normal form if $t \in \mathcal{T}\mathrm{erm}_\eta[\mathcal{F}_{ext}, \mathcal{V}]$ and t is a $\mathcal{R}_\beta \cup \mathcal{R}$-normal form.

4.4.2 The HLNC Inference Rules

The calculus HLNC consists of the following inference rules:

Inference rules on strict equations

$\texttt{[o]}_\approx$ *outermost narrowing*

$$\frac{\{f[s_1, \ldots, s_n] \simeq t, E\}}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, E\}}$$

if t is outermost irreducible and there exists a fresh variant $f[l_1, \ldots, l_n] \longrightarrow r$ of a rewrite rule in $\mathcal{R}$.

$\texttt{[o}\beta\texttt{]}_\approx$ *outermost $\beta-$reduction*

$$\frac{\{\lambda[\{x_1, \ldots, x_n\}, u][s_1, \ldots, s_n] \simeq t, E\}}{\{u\theta \approx t, E\}}$$

where $\theta = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$, if t is outermost irreducible.

`[ovi]`$_\approx$ *outermost narrowing on flex terms with imitation binding*

$$\frac{\{x[s_1,\ldots,s_n] \simeq t, E\}}{\{g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]] \approx t, E\}\theta}$$

where $\theta = \{x \mapsto \lambda[\{x_1,\ldots,x_n\}, g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]\}$ and $h_1,\ldots,h_m \in \mathcal{V}$ are fresh, if $1 \leq n$, t is outermost irreducible, $x \in \mathcal{V}$,

`[ovp]`$_\approx$ *outermost narrowing on flex terms with projection binding*

$$\frac{\{x[s_1,\ldots,s_n] \simeq t, E\}}{\{s_i[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]] \approx t, E\}\theta} \quad t \text{ not flex, } t \text{ outermost irreducible, } n > 0$$

where $\theta = \{x \mapsto \lambda[\{x_1,\ldots,x_n\}, x_i[h_1[x_1,\ldots,x_n],\cdots(h_m[x_1,\ldots,x_n]]$ and $h_1,\ldots,h_m \in \mathcal{V}$ are fresh, $1 \leq n$, $1 \leq i \leq n$.

`[i]`$_\approx$ *imitation*

$$\frac{\{c[s_1,\ldots,s_n] \simeq x, E\}}{\{s_1 \approx x_1,\ldots,s_n \approx x_n, E\}\theta}$$

where $\theta = \{x \mapsto c[x_1,\ldots,x_n]\}$ with $x_1,\ldots,x_n$ fresh variables if $x \in \mathcal{V}[c[s_1,\ldots,s_m]]$ or $c[s_1,\ldots,s_m] \notin \mathcal{T}\mathrm{erm}[\mathcal{F}_c,\mathcal{V}]$.

`[d]`$_\approx$ *decomposition*

$$\frac{\{u[s_1,\ldots,s_n] \approx u[t_1,\ldots,t_n], E\}}{\{s_1 \approx t_1,\ldots,s_n \approx t_n, E\}} \quad u \in \mathcal{F}_c \cup \mathcal{V}$$

$$\frac{\{\lambda[\{x_1,\ldots,x_n\}, s] \approx \lambda[\{x_1,\ldots,x_n\}, t], E\}}{\{s \approx t, E\}}$$

`[v]`$_\approx$ *variable elimination*

$$\frac{\{s \approx x, E\}}{\{E\}\{x \mapsto s\}} \quad \frac{\{x \approx s, E\}}{\{E\}\{x \mapsto s\}} \quad s \notin \mathcal{V}$$

if $x \notin \mathcal{V}[s]$ and $s \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c,\mathcal{V}]$

`[t]`$_\approx$ *removal of trivial equation*

$$\frac{\{t \approx t, E\}}{\{E\}}$$

Inference rules on parameter passing equations

`[o]`$_\asymp$ *outermost narrowing*

$$\frac{\{f[s_1,\ldots,s_n] \asymp t, E\}}{\{s_1 \asymp l_1,\ldots,s_n \asymp l_n, r \asymp t, E\}} \quad t \notin \mathcal{V}$$

if there exists a fresh variant $f[l_1,\ldots,s_n] \longrightarrow r$ of a rewrite rule in $\mathcal{R}$.

`[o`β`]`$_{\asymp}$ *outermost* β*−reduction*

$$\frac{\{\lambda[\{x_1,\ldots,x_n\},u][s_1,\ldots,s_n]\asymp t,E\}}{\{u\theta\asymp t,E\}}\quad t\notin\mathcal{V}$$

where $\theta=\{x_1\mapsto s_1,\ldots,x_n\mapsto s_n\}$.

`[ovi]`$_{\asymp}$ *outermost narrowing on flex terms with imitation binding*

$$\frac{\{x[s_1,\ldots,s_n]\asymp t,E\}}{\{g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]]\approx t,E\}\theta}$$

where $1\le n$, $\theta=\{x\mapsto\lambda[\{x_1,\ldots,x_n\},g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]]]\}$ and $h_1,\ldots,h_m\in\mathcal{V}$ are fresh,
if $t\notin\mathcal{V}$.

`[ovp]`$_{\asymp}$ *outermost narrowing on flex terms with projection binding*

$$\frac{\{x[s_1,\ldots,s_n]\asymp t,E\}}{\{s_i[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]]\asymp t,E\}\theta}$$

where $\theta=\{x\mapsto\lambda[\{x_1,\ldots,x_n\},x_i[h_1[x_1,\ldots,x_n],\ldots,h_m[x_1,\ldots,x_n]]]\}$ and $1\le i\le n$ if $n\ge 1$.

`[d]`$_{\asymp}$ *decomposition*

$$\frac{\{u[s_1,\ldots,s_n]\asymp u[t_1,\ldots,t_n],E\}}{\{s_1\asymp t_1,\ldots,s_n\asymp t_n,E\}}\quad c\in\mathcal{F}_c\cup\mathcal{V}$$

$$\frac{\lambda[\{x_1,\ldots,x_n\},s]\asymp\lambda[\{x_1,\ldots,x_n\},t],E\}}{\{s\asymp t,E\}}$$

`[v]`$_{\asymp}$ *variable elimination*

$$\frac{\{s\asymp x,E\}}{\{E\}\{x\mapsto s\}}\quad\frac{\{x\asymp s,E\}}{\{E\}\{x\mapsto s\}}\quad s\notin\mathcal{V}$$

The following theorem is crucial to justify the calculus HLNC.

Theorem 6 (Completeness of HLNC) *Let* $\mathcal{R}$ *be a left-linear,* β*-free and confluent* TRS_λ *and* G *a goal subject to the restriction that each side of an equation of* G *is a long* η*-normal form. For every long* $\eta\mathcal{R}\beta$*-normalized strict solution* θ *of* G *there exists an HLNC derivation* $G\rightsquigarrow^*_{\theta'}\{\}$ *such that* $\theta'\le\theta[\mathcal{V}[G]]$.

Chapter 5

The CFLP Calculus

This chapter is still incomplete

This calculus is the result of our effort in combining a functional logic programming language with methods for solving constraints in various constraint theories. The current implementation considers only (linear) constraints over the domain of reals. We are currently working on extending the set of constraints expressible in our calculus (this amounts to extending the constraint solving capabilities of the system).

The basic idea of CFLP is to extend the domain of equations expressible in the functional logic language with constraints, i.e. equations that can be solved by constraint solving procedures. In our setting, constraints are formulae of a distinguished signature that are to be interpreted over some specified constraint theory. Solving constraints, i.e. finding valid variable assignments of the constraints with respect to the constraint theory is a special resolution step which is like a classical step in functional logic programming, but instead of a narrowing step we perform a call to an external solver.

CFLP differs from a narrowing calculus in the following respects:

- the signature of the functional logic language was extended with distinguished function symbols for expressing constraints. The calculus is designed to recognize constraints by looking at the outermost function symbol of an equation
- the goal (i.e. system of equations to be solved) to the calculus is a pair composed of two disjoint sets of equations:
  - a set of equations to be (still) treated by the lazy narrowing calculus,
  - a set of constraints.

Initially, the set of equations for the lazy narrowing calculus coincides with the initial goal of the CFLP query, and the set of constraints is empty. If we represent a CFLP goal as

$$flpEqs \parallel csEqs$$

where $flpEqs$ is the list of equations for function logic programming component of the system (the narrowing calculus), and $csEqs$ is the list of constraints, then a CFLP refutation for an intial goal $goal$ provided in a CFLP query has the structure

$$goal \parallel \{\} \rightsquigarrow^*_\theta \{\} \parallel csEqs \rightsquigarrow^*_\rho \{\} \parallel \{\}.$$

The first part of the CFLP refutation

$$goal \parallel \{\} \rightsquigarrow^*_\theta \{\} \parallel csEqs$$

is carried out by the functional logic component of CFLP and computes a partial solution θ to the initial goal $goal$. The last part of the CFLP refutation

$$\{\} \parallel csEqs \rightsquigarrow^*_\rho \{\} \parallel \{\}$$

is performed by the constraint solving component of CFLP and "completes" the partial solution θ with the solution ρ to the set $csEqs$ of constraints.

This chapter is organized as follows. In section 5.1 we give the inference rules of the functional logic component of CFLP. Section 5.2 describes the structure of the constraint logic component of CFLP. The last section contains implementation details.

5.1 Inference Rules

We now assume that the set of function symbols is partitioned as follows

$\mathcal{F} = \mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{F}_e$	where	$\mathcal{F}_c$is the set of constructor symbols
		$\mathcal{F}_d$is the set of defined function symbols
		$\mathcal{F}_e$is the set of external function symbols
	and	$\mathcal{F}_c \cap \mathcal{F}_d = \emptyset, \mathcal{F}_c \cap \mathcal{F}_e = \emptyset, \mathcal{F}_d \cap \mathcal{F}_e = \emptyset$

For the extended calculus, which we call CFLP, the inference rules act on extended goals of the form $G \parallel C$ where $G, C \in$ `InitialGoal`. An inference step is represented either as

$$\frac{G \parallel C}{G' \parallel C'}$$

or as

$$G \parallel C \leadsto_\theta G' \parallel C'$$

where

$G \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$	is	the current goal to be proccessed
$C \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$	is	the current constraint store
$G' \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$	is	the new goal produced by the inference step
$C' \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$	is	the new constraint store produced by the inference step

The last representation shows also the substitution θ involved in applying the inference rule.

In the sequel stick to the following notational conventions

ϕ, ξ	external function symbols
$h_1, \ldots, h_m$	higher order variables
g	defined function symbol
C	constraint pool ($C \in \mathcal{G}\mathrm{oal}[\mathcal{F}, \mathcal{V}]$)

The inference rules of this extended calculus are shown below.

Inference Rules on Strict Equations

$\texttt{[o]}_\approx$ *outermost narrowing*

$$\frac{\{f[s_1, \ldots, s_n] \simeq t, E\} \parallel C}{\{s_1 \asymp l_1, \ldots, s_n \asymp l_n, r \approx t, E\} \parallel C}$$

if t is outermost irreducible and there exists a fresh variant $f[l_1, \ldots, l_n] \longrightarrow r \Leftarrow e_1 \wedge \ldots \wedge e_p$ of a conditional rewrite rule in $\mathcal{R}$.

$\texttt{[o}\beta\texttt{]}_\approx$ *outermost $\beta-$reduction*

$$\frac{\{\lambda[\{x_1, \ldots, x_m\}, s][s_1, \ldots, s_m] \simeq t, E\} \parallel C}{\{s\theta \approx t, E\} \parallel C}$$

where $\theta = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, if t is outermost irreducible.

$\texttt{[ovi]}_\approx$ *outermost narrowing on flex terms with imitation binding*

$$\frac{\{x[s_1, \ldots, s_n] \simeq t, E\} \parallel C}{\{g[h_1[s_1, \ldots, s_n], \ldots, h_m[s_1, \ldots, s_n]] \approx t, E\}\theta \parallel C\theta}$$

where $\theta = \{x \mapsto \lambda[\{x_1, \ldots, x_n\}, g[h_1[s_1, \ldots, s_n], \ldots, h_m[s_1, \ldots, s_n]\}$ and $h_1, \ldots, h_m \in \mathcal{V}$ are fresh,
if $1 \le n$, t is outermost irreducible, $x \in \mathcal{V}$,

$\texttt{[ovp]}_{\approx}$ *outermost narrowing on flex terms with projection binding*

$$\frac{\{x[s_1,\ldots,s_n] \simeq t, E\} \parallel C}{\{s_i[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]] \approx t, E\}\theta \parallel C\theta}$$

where $\theta = \{x \mapsto \lambda[\{x_1,\ldots,x_n\}, x_i[h_1[x_1,\ldots,x_n],\cdots,h_m[x_1,\ldots,x_n]]$ and $h_1,\ldots,h_m \in \mathcal{V}$ are fresh,
if $x \in \mathcal{V}$ and t is not flex, outermost irreducible, $1 \leq n, 1 \leq i \leq n$.

$\texttt{[i]}_{\approx}$ *imitation*

$$\frac{\{c[s_1,\ldots,s_n] \simeq x, E\} \parallel C}{\{s_1 \approx x_1,\ldots,s_n \approx x_n, E\}\theta \parallel C\theta}$$

where $\theta = \{x \mapsto c[x_1,\ldots,x_n]\}$ with $x_1,\ldots,x_n$ fresh variables
if $x \in \mathcal{V}[c[s_1,\ldots,s_m]]$ or $c[s_1,\ldots,s_m] \notin \mathcal{T}\mathrm{erm}[\mathcal{F}_c,\mathcal{V}]$.

$\texttt{[d]}_{\approx}$ *decomposition*

$$\frac{\{c[s_1,\ldots,s_n] \approx c[t_1,\ldots,t_n], E\} \parallel C}{\{s_1 \approx t_1,\ldots,s_n \approx t_n, E\} \parallel C}$$

$$\frac{\{x[s_1,\ldots,s_n] \approx x[t_1,\ldots,t_n], E\} \parallel C}{\{s_1 \approx t_1,\ldots,s_n \approx t_n, E\} \parallel C}$$

$$\frac{\{\lambda[\{x_1,\ldots,x_n\}, s] \approx \lambda[\{x_1,\ldots,x_n\}, t], E\} \parallel C}{\{s \approx t, E\} \parallel C}$$

$\texttt{[v]}_{\approx}$ *variable elimination*

$$\frac{\{s \approx x, E\} \parallel C}{\{E\}\theta \parallel C\theta}$$

$$\frac{\{x \approx s, E\} \parallel C}{\{E\}\theta \parallel C\theta} \quad s \notin \mathcal{V}$$

where $\theta = \{x \mapsto s\}$ if $x \notin \mathcal{V}[s]$ and $s \in \mathcal{T}\mathrm{erm}[\mathcal{F}_c,\mathcal{V}]$.

$\texttt{[t]}_{\approx}$ *removal of trivial equation*

$$\frac{\{t \approx t, E\} \parallel C}{\{E\} \parallel C}$$

Inference rules on Parameter Passing Equations

$\texttt{[o]}_{\asymp}$ *outermost narrowing*

$$\frac{\{f[s_1,\ldots,s_n] \asymp t, E\} \parallel C}{\{s_1 \asymp l_1,\ldots,s_n \asymp l_n, e_1,\ldots \wedge e_p, r \asymp t, E\} \parallel C}$$

if $t \notin \mathcal{V}$ and there exists a fresh variant $f[l_1,\ldots,s_n] \longrightarrow r \Leftarrow e_1 \wedge \ldots, e_p$ of a rewrite rule in $\mathcal{R}$.

`[o`β`]`$_{\asymp}$ *outermost β−reduction*

$$\frac{\{\lambda[\{x_1,\ldots,x_n\},s][s_1,\ldots,s_n] \asymp t, E\} \parallel C}{\{s\theta \asymp t, E\} \parallel C}$$

where $\theta = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ if $t \notin \mathcal{V}$.

`[ovi]`$_{\asymp}$ *outermost narrowing on flex terms with imitation binding*

$$\frac{\{x[s_1,\ldots,s_n] \asymp t, E\} \parallel C}{\{g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]] \approx t, E\}\theta \parallel C\theta}$$

where $1 \le n$, $\theta = \{x \mapsto \lambda[\{x_1,\ldots,x_n\}, g[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]]]\}$
and $h_1,\ldots,h_m \in \mathcal{V}$ are fresh,
if $t \notin \mathcal{V}$.

`[ovp]` $\asymp$ *outermost narrowing on flex terms with projection binding*

$$\frac{\{x[s_1,\ldots,s_n] \asymp t, E\} \parallel C}{\{s_i[h_1[s_1,\ldots,s_n],\ldots,h_m[s_1,\ldots,s_n]] \asymp t, E\}\theta \parallel C\theta}$$

where $\theta = \{x \mapsto \lambda[\{x_1,\ldots,x_n\}, x_i[h_1[x_1,\ldots,x_n],\ldots,h_m[x_1,\ldots,x_n]]]\}$
and $1 \le i \le n$ if $n \ge 1$.

`[d]`$_{\asymp}$ *decomposition*

$$\frac{\{c[s_1,\ldots,s_n] \asymp c[t_1,\ldots,t_n], E\} \parallel C}{\{s_1 \asymp t_1,\ldots,s_n \asymp t_n, E\} \parallel C}$$

$$\frac{\{x[s_1,\ldots,s_n] \asymp x[t_1,\ldots,t_n], E\} \parallel C}{\{s_1 \asymp t_1,\ldots,s_n \asymp t_n, E\} \parallel C}$$

$$\frac{\lambda[\{x_1,\ldots,x_n\},s] \asymp \lambda[\{x_1,\ldots,x_n\},t], E\} \parallel C}{\{s \asymp t, E\} \parallel C}$$

`[v]`$_{\asymp}$ *variable elimination*

$$\frac{\{s \asymp x, E\} \parallel C}{\{E\}\theta \parallel C\theta}$$

$$\frac{\{x \asymp s, E\} \parallel C}{\{E\}\theta \parallel C\theta} \quad \text{if } s \notin \mathcal{V}.$$

where $\theta = \{x \mapsto s\}$.

Inference Rules on External Equations

`[ei]` *imitation rules*

$$\frac{\{\phi[s_1,\ldots,s_m] \approx x, E\} \parallel C}{\{s_1 \approx x_1,\ldots,s_m \approx s_m, \lceil\phi[x_1,\ldots,x_m] \approx x\rceil, E\}\theta \parallel C\theta}$$

$$\frac{\{x \approx \phi[s_1,\ldots,s_m], E\} \parallel C}{\{s_1 \approx x_1,\ldots,s_m \approx s_m, \lceil\phi[x_1,\ldots,x_m] \approx x\rceil, E\}\theta \parallel C\theta}$$

where $x_1,\ldots,x_m \in \mathcal{V}$ are fresh variables and $\theta = \{x \mapsto \phi[x_1,\ldots,x_n]\}$.

$$\frac{\{\phi[s_1,\ldots,s_m] \approx \xi[t_1,\ldots,t_n], E\} \parallel C}{\{s_1 \approx x_1,\ldots,s_m \approx x_m, t_1 \approx y_1,\ldots,t_n \approx y_n, \lceil\phi[x_1,\ldots,x_m] \approx \xi[y_1,\ldots,y_n]\rceil, E\} \parallel C}$$

where $x_1,\ldots,x_m, y_1,\ldots,y_n \in \mathcal{V}$ are fresh variables.

$$\frac{\{\phi[s_1,\ldots,s_m] \asymp t, E\} \parallel C}{\{s_1 \approx x_1,\ldots,s_m \approx x_m, \lceil\phi[x_1,\ldots,x_m] \approx t\rceil, E\} \parallel C}$$

where $x_1,\ldots,x_m, y_1,\ldots,y_n \in \mathcal{V}$ are fresh variables.

`[xfer]` *transfer to the constraint store*

$$\frac{\{\lceil e\rceil, E\} \parallel \{E'\}}{\{E\} \parallel \{e, E'\}}$$

Inference Rules for Constraint Solving

- **[cs]** constraint solver call

$$\frac{\{\ \} \parallel C}{\{E\} \parallel \{\}}$$

where $\{E\} \in \mathrm{CSolve}[C]$ is a solution to the system of constraints C. Here, `CSolve` denotes an invocation of the the constraint solving component.

5.1.1 Remarks

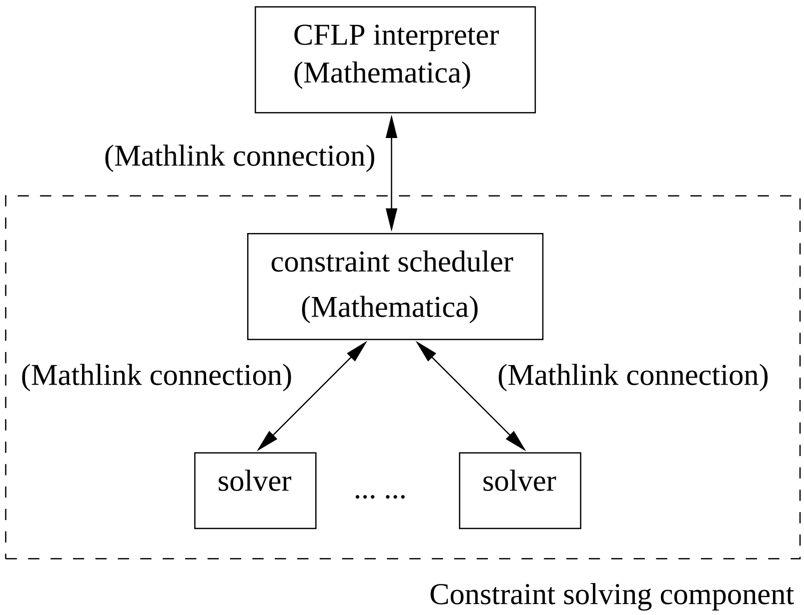
The main difference to the lazy narrowing calculus is the special treatment of equations between terms with external function symbol at the outermost position. These equations are identified with the constraints and collected in the constraint solving component of the CFLP goal (rule **[xfer]**). At the functional logic level, the resolution of constraints is encoded in the rule **[cs]** which is fired when no more equations are available to be processed by the CFLP interpreter (i.e. at a leaf node). This decision is based on the observation that for numerous examples from science and engineering, the calls of the solver at internal nodes only seldom causes search space pruning and more often it causes overhead (see [4]). However, we intend to introduce in a next version a mechanism to override this default behavior.

5.1.2 Implementation

See notebook `Calculus.nb` for implementation details.

5.2 The Constraint Solving Component

The constraint solving component checks for solvability of combined constraints by cooperating with specialized sub-solvers. Schematically, the CFLP system has the following structure:



The CFLP interpreter implements the slightly modified version of the narrowing calculus described in the previous section. The CFLP constraint solving component is structured on two layers:

- the constraint scheduler which cooperates with various sonstraint solvers to find solutions to the set of constraints received from the CFLP interpreter,

- the constraint solvers.

The interpreter, scheduler and constraint solvers run as separate processes communicating via MathLink. The first two processes–interpreter and scheduler–are written in Mathematica 3.0. A constraint solver may be any process that is able to respond to queries from the constraint scheduler via a MathLink connection. In the current implementation, also the constraint solvers are Mathematica processes.

In the sequel we give the algorithmic descriptions of the scheduler and its auxiliary `CSolve` procedure.

$C^* \leftarrow$ **Schedule**$[Cs]$

Inputs	Cs	:	list of lists of constraints (representing a disjunction of conjunctions of constraints)
Globals	Cs^*	:	tree-like structure containing the (partial) solutions to Cs.
	M^*	:	list of all method identifiers
	$IsOver$	:	flag which indicates when the solving loop of constraint scheduler terminates.
Locals	idx	:	label of a leaf node in Cs^*

1. [Setup.]
   Let $Cs = \{C_1, \ldots, C_n\}$.
   $Cs^* \leftarrow \{\{C_1, M^*\}, \ldots, \{C_n, M^*\}\}$.

2. [Loop over the disjunctive constraint sets.]
   $IsOver \leftarrow$ `False`
   While Not$[IsOver]$ do
   For every leaf node label idx of Cs^* do `CSolve`$[Cs^*[[idx]]]$
   end
   return Cs^*

Figure 5.1: The constraint scheduler

Figure 5.1 gives an algorithmic description of the constraint scheduler used in our system, and Figure 5.2 gives the algorithmic description of the constraint solvers acting on constraint pools.

CSolve[idx]

Input	idx	:	leaf node label in Cs^*
Globals	Cs^*	:	tree-like structure containing the (partial) solutions to Cs (see `Scheduler`).
	M^*	:	list of all method identifiers,
	$IsOver$	:	flag which indicates when the solving loop of constraint scheduler terminates.
Locals	i	:	label of a leaf node in Cs^*
	Ci, Mi, C'i	:	$\{Ci, Mi, C'i\}$ is the content of node idx in Cs^*; *Mi* is a list of identifiers, *Ci* is the list of constraints on which the first method of *Mi* is applied, and *C'i* is the list of constraints on which the methods of *Mi* except the first one will be applied.
	$Status[i]$	:	status flag for processing the constraints of node $Cs^*[[i]]$
	$Answer[i]$	:	buffer for temporary storing the partial solutions to the constraint system $Cs^*[[i]]$

1. [look for available results]
   Assume $Cs^*[[idx]] = \{Ci, Mi\}$
   if HasReceivedAnswer[idx] then
     if $Answer[idx] = \{Ci\}$ then
       $Cs^* \leftarrow Cs[idx \leftarrow \{Answer[idx] \cup C'i, \mathrm{rest}[Mi], \{\}\}]$
     else if $Answer[idx] = \{\}$ then $Cs^* \leftarrow Cs[idx \leftarrow \{Ci, \{\}, \{\}\}]$
     else $Cs^* \leftarrow Cs[idx \leftarrow \{Answer[idx] \cup C'i, M^* - \mathrm{first}[Mi], \{\}\}]$,
     $Status[idx] \leftarrow$ "$idle$",
     $IsOver \leftarrow$ `False`
   else
     case $Status[idx]$ of
       "$waiting$" :
         AskForResult[idx],
         $IsOver \leftarrow$ `False`
       "$idle$" :
         if $Mi = \{\}$ then $Status[idx] \leftarrow$ "$done$",
         else
           Let M be the first identifier in Mi
           Let Ci, C'_i Call[M, Ci, idx],
           $status[idx] \leftarrow$ "$waiting$",
           $IsOver \leftarrow$ `False`

Figure 5.2: The `CSolve` procedure

It is desirable to ensure that

- the iteration eventually terminates,
- all methods are applied in a fair frequency, and
- no method is applied more than once consecutively.

We notice that the constraint solver applies several solving methods repeatedly until the constraints cannot be changed anymore. The functions

This scheduling problem can be solved by maintaining a list of the methods that still need to be applied. Initially this list contains all the methods. When a method is applied it is removed from the list. If the resulting constraints are different from the input constraints, then we include all other methods into the list. We repeat this process until the list of methods to be applied becomes empty.

A method is applied on a set of constraints when calling

Call[*method, constraints, i*]

This call is expected to return a result which is stored in Answer[i]. Whether the result is already stored or not in Answer[*i*] is indicated by the boolean value of HasReceivedAnswer[i].

5.2.1 Implementation

The constraint scheduler is implemented as a separate *Mathematica* process that is called when the CFLP interpreter exhausted the list of goal equations. Its implementation is based on the algorithmic descriptions given in Figures 5.1 and 5.2. In the current implementation, the method calls

Call[*method, constraints, i*]

are invocations of the *Mathematica* `Solve` procedure which runs as a separate process.

5.3 Language Extension for OR parallelism

In addition to the inferewnce rules presented above, an inference rule for alternative equations in the goal was introduced.

[or] *inference rule for alternative equations*

$$\frac{\{e_1 \vee \cdots \vee e_n, E\} \parallel C}{\{e_i, E\} \parallel C}$$

This inference rule corresponds to a choice point of the calculus. The implementation considers all choices, except when a limit for the number of solutions is specified reached before exhausting all alternatives.

The **[or]** inference rule is implemented as follows: consider all CFLP derivations

$$\{e_1 \vee \cdots \vee e_n, E\} \parallel C \rightsquigarrow_{\theta_i} \{\} \parallel C_i$$

and call the constraint solver to solve simultaneously the systems of constraints C_i. In addition to the system C_i of constraints to be solved, the constraint solver gets also the substitution θ_i computed by the CFLP interpreter for C_i. The substitution θ is used by the constraint solver to determine the variables for which we try to find a solution.

Bibliography

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

[2] M. Hamada, Personal communication.

[3] M. Hamada and T. Ida, *Implementation of Lazy Narrowing Calculi in Mathematica*, Technical Report 97-02, RISC Linz, 1997.

[4] H. Hong, *CLP(CF) Constraint Logic Programming over Complex Functions*, Technical Report 94-09, RISC Linz, 1994.

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

[6] S. Kaplan. Conditional Rewrite Rules. *Theoretical Computer Science 33, pp. 175-193, 1984.*

[7] A. Middeldorp and E.Hamoen. Completeness results for basic narrowing. Aplicable Algebra in Engineering, Communication and Computing, 5:213-253, 1994.

[8] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. In *Proceedings of the Fuji Inernational Workshop on Functional and Logic Programming, Susuno,* pp. 104-118. World Scientific, Singapore, 1995.

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

[10] M. Hamada and A. Middledorp. Strong Completeness of a Lazy Conditional Narrowing Calculus. Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, World Scientific, Singapore, 1997.

[11] T. Suzuki, K. Nakagawa and T. Ida. Higher Order Lazy Narrowing Calculus: a Computation Model for a Higher-Order Functional Logic Language.

[12] E. Giovannetti and C. Moiso. A completeness result for e-unification algorithms based on conditional narrowing. In *Proceedings of the Workshop on Foundations of Logic and Functional Programming, Trento,* volume 306 of *Lecture Notes in Computer Science,* pp. 157-167, 1986.

[13] A. Martelli and U. Montanari. An efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems* 4(2), pp. 258-282, 1982.

[14] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming.* Phd thesis, Technischen Universität München, 1995.

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