$\mathcal{TOY}$

A Multiparadigm Declarative Language

Version 2.3

Rafael Caballero, Jaime Sánchez (Eds.)

Document Version 1.0

Purificación Arenas Sánchez Antonio J. Fernández Leiva Ana Gil Luezas
Francisco J. López Fraguas Mario Rodríguez Artalejo Fernando Sáenz Pérez

Universidad Complutense de Madrid

December 2006

Contents

Preface **6**

1 Getting Started **8**
1.1 Supported Platforms and System Requirements . . . 8
1.2 Downloading and Running $\mathcal{TOY}$ . . . 8
1.3 First Steps in $\mathcal{TOY}$ . . . 9
1.4 Compiling and Loading Programs . . . 10
1.5 Commands . . . 12
1.5.1 Syntax . . . 12
1.5.2 Interaction with the Operating System . . . 13
1.5.3 Compiling and Loading Programs . . . 13
1.5.4 Information about Programs . . . 14
1.5.5 Commands for Constraint Solvers . . . 15
1.5.6 Miscellanea . . . 16
1.5.7 Defining New Commands . . . 17

2 The Language **18**
2.1 Expressions . . . 18
2.2 Types . . . 19
2.3 Datatype Definitions . . . 21
2.4 Predefined Types . . . 23
2.5 Type Synonyms . . . 24
2.6 Function Definitions . . . 26
2.7 Operators and Sections . . . 30
2.8 Primitive and Predefined Functions . . . 31
2.9 Including Programs . . . 34
2.10 Layout Rule . . . 35
2.11 Predicate Definitions . . . 35
2.12 Non-Deterministic Functions . . . 38
2.13 Goals and Computed Answers . . . 42
2.14 Higher-Order Programming . . . 46
2.15 Finite Failure . . . 51
2.16 Constraints . . . 54

2.16.1 Syntactical Equality and Disequality Constraints . . . 56
2.16.2 Arithmetical Constraints over Real Numbers . . . 60

3 Constraints over Finite Domains 66
3.1 Introduction . . . 66
3.1.1 Efficiency . . . 67
3.1.2 Modes and Bidirectionality . . . 67
3.1.3 Loading the $\mathcal{FD}$ Constraint Library . . . 67
3.2 $\mathcal{FD}$ Constraints and Functions . . . 68
3.2.1 Predefined Data Types . . . 68
3.2.2 Membership Constraints . . . 68
3.2.3 Enumeration Functions . . . 74
3.2.4 Relational Constraints . . . 77
3.2.5 Arithmetic Constraint Operators . . . 78
3.2.6 Arithmetic Constraints . . . 79
3.2.7 Combinatorial Constraints . . . 81
3.2.8 Propositional Constraints . . . 86
3.2.9 Reflection Functions . . . 87
3.2.10 $\mathcal{FD}$ Set Functions . . . 92
3.2.11 Statistics Functions . . . 101
3.3 Introductory Programs . . . 103
3.3.1 The Length of a List (`haslength.toy`) . . . 103
3.3.2 Send + More = Money (`smm.toy`) . . . 104
3.3.3 N-Queens (`queens.toy`) . . . 105
3.3.4 A Cryptoarithmetic Problem (`alpha.toy`) . . . 106
3.3.5 Magic Series (`magicser.toy`) . . . 107

4 Cooperation of Solvers 109
4.1 Introduction . . . 109
4.1.1 Efficiency . . . 109
4.1.2 Libraries necessaries for Cooperation . . . 110
4.2 The Constraint Cooperation *Bridge* . . . 110
4.3 Binding . . . 110
4.4 Propagation . . . 111
4.4.1 Propagation from $\mathcal{FD}$ to $\mathcal{R}$ . . . 111
4.4.2 Propagation from $\mathcal{R}$ to $\mathcal{FD}$ . . . 115
4.5 Introductory Programs . . . 118
4.5.1 Intersection of a discrete grid and a continuous region (`bothIn.toy`) . . . 118
4.5.2 Distribution of raw material between suppliers and consumers (`distribution.toy`) . . . 121

5 Input/Output **125**
5.1 Standard Input/Output Functions . . . 126
5.2 Sequencing Input/Output Functions . . . 126
5.3 Do Notation . . . 128
5.4 Files . . . 129
5.4.1 The `io_file` System Module . . . 130
5.5 Graphic Input/Output . . . 131
5.5.1 Mutable Variables . . . 137
5.6 List Comprehensions . . . 138

6 Debugging Tools **142**
6.1 Introduction . . . 142
6.2 Declarative Debugging . . . 142
6.3 An Example . . . 143
6.3.1 Starting $\mathcal{DDT}$ . . . 143
6.3.2 Looking for buggy nodes . . . 144
6.3.3 Strategies . . . 145
6.4 Limitations . . . 147
6.5 How Does it Work? . . . 148

A Programming Examples **153**
A.1 Logic Programming . . . 153
A.1.1 Peano Numbers (`peano.toy`) . . . 153
A.1.2 Paths in a Graph (`paths.toy`) . . . 154
A.2 Functional Programming . . . 155
A.2.1 Arithmetic for Peano Numbers (`arithmetic.toy`) . . . 155
A.2.2 Infinite Lists (`inflists.toy`) . . . 156
A.2.3 Prime Numbers (`primes.toy`) . . . 157
A.2.4 Hamming Codes (`hamming.toy`) . . . 158
A.2.5 Process Network: Client-Server Interaction (`clientserver.toy`) . . . 159
A.3 Functional Logic Programming . . . 161
A.3.1 Inserting an Element in a List (`insert.toy`) . . . 161
A.3.2 The Choice Operator (`choice.toy`) . . . 161
A.3.3 The Inverse Function (`inverse.toy`) . . . 162
A.4 Programming with Failure . . . 162
A.4.1 Default Rules (`automata.toy`) . . . 162
A.4.2 A Propositional Tautology Generator (`tautology.toy`) . . . 165
A.5 Programming with Equality and Disequality Constraints . . . 167
A.5.1 The Cardinal of a List (`cardinal.toy`) . . . 167
A.6 Programming with Real Constraints . . . 168
A.6.1 Defining Regions of the Plane (`regions.toy`) . . . 169
A.7 Programming with Finite Domain Constraints . . . 172
A.7.1 A Colouring Problem (`colour.toy`) . . . 172
A.7.2 Linear Equations (`eq10.toy` and `eq20.toy`) . . . 173

A.7.3 DNA Sequencing (`dna.toy`) . . . 176
A.7.4 A Scheduling Problem (`scheduling.toy`) . . . 178
A.7.5 A Hardware Design Problem . . . 181
A.7.6 Golomb Rulers: An Optimization Problem (`golomb.toy`) . . . 187
A.7.7 Lazy Constraint Programs . . . 189
A.7.8 Programmable Search (`search.toy`) . . . 194

B A Miscellanea of Functions **196**
B.1 `misc.toy` . . . 196
B.2 `misc.toy` . . . 204

C Sample Modules **206**

D Syntax **216**
D.1 Lexicon . . . 216
D.2 Grammar . . . 219

E Type Inference **227**
E.1 Dependency Analysis . . . 227
E.2 Type Inference Algorithm . . . 228

F Declarative Semantics **234**
F.1 Motivation . . . 234
F.2 A Constructor-Based Rewriting Logic . . . 236
F.3 Correctness of Computed Answers . . . 237
F.4 Models . . . 238
F.5 Extensions . . . 239

G Operational Semantics **240**
G.1 First-Order Translation of $\mathcal{TOY}$ Programs . . . 240
G.1.1 Enhancing the Signature . . . 241
G.1.2 First-Order Translation of Program Rules and Rules for @ . . . 241
G.2 Introduction of Suspensions . . . 242
G.3 Prolog Code Generation . . . 242
G.3.1 Clauses for Goal Solving (Solve) . . . 243
G.3.2 Clauses for Computing Head Normal Forms (Hnf_Σ) . . . 245
G.3.3 Clauses for Function Definitions (Prolog_{FS}) . . . 246

H Release Notes History **249**
H.1 Version 2.3. Launched on December, 2006 . . . 249
H.2 Version 2.2.3. Launched on July, 2006 . . . 250
H.3 Version 2.2.2. Launched on March, 2006 . . . 252
H.4 Version 2.2.1. Launched on November, 2005 . . . 255
H.5 Version 2.2.0. Launched on August, 2005 . . . 255
H.6 Version 2.1.0. Launched on May, 2005 . . . 256

H.7 Version 2.0. Launched on February, 2002 256

I Known Bugs **257**

Preface

$\mathcal{TOY}$ is a multiparadigm programming language and system, designed to support the main declarative programming styles and their combination.

Programs in $\mathcal{TOY}$ can include definitions of lazy functions in Haskell style, as well as definitions of predicates in Prolog style. Both functions and predicates must be well-typed with respect to a polymorphic type system. Moreover, $\mathcal{TOY}$ programs can use constraints in CLP style within the definitions of both predicates and functions. The constraints supported by the system include symbolic equations and disequations, linear arithmetic constraints over the real numbers and finite domain constraints.

Computations in $\mathcal{TOY}$ solve goals by means of a demand driven lazy narrowing strategy, first proposed in [22] and closely related to the needed narrowing strategy independently introduced in [1]. This computation model subsumes both lazy evaluation and SLD resolution as particular cases. For $\mathcal{TOY}$ programs using constraints, demand driven lazy narrowing is combined with constraint solving, and constraints in solved form can occur within computed answers.
$\mathcal{TOY}$ provides some additional features which are not found in traditional functional or logic languages. *Non-deterministic functions*, combined with lazy evaluation, help to improve efficiency in search problems. *Higher-order patterns* enable an intensional representation of functions, useful both for functional and logic computations, and they do not introduce undecidable unification problems. These and other features of the language, including a declarative semantics in the spirit of logic programming, have a firm mathematical foundation.

This report is intended both as a manual for $\mathcal{TOY}$ users and as an informal (but hopefully precise) description of the language and system. Here is a brief description of the overall purpose of each chapter:

- Chapter 1: Explains how to download and install the $\mathcal{TOY}$ system. This chapter also gives a first overview of the system's features and tools, sufficient to start writing and running program.
- Chapter 2: Introduces a detailed description of $\mathcal{TOY}$ programs covering syntax, informal semantics, and pragmatics.
- Chapter 3: Presents the finite domain solver and their applications.
- Chapter **??**: Presents the cooperation between finite domain and real solvers and their applications.
- Chapter 5: Explains how to include input/output interactions in $\mathcal{TOY}$.
- Chapter 6: Introduces the declarative tool for debugging $\mathcal{TOY}$ programs.

In addition, a series of appendices include programming examples, predefined functions and library modules, and some technical specifications concerning the syntax and semantics of the language.

$\mathcal{TOY}$ has been designed by the *Declarative Programming Group* at the *Universidad Complutense de Madrid*, following previous experiences with the design of declarative languages. The finite domain library has been developed in cooperation with Antonio Fernández Leiva from the *University of Málaga*. In addition to the authors of the report, many people have contributed to the development of the system. We would like to acknowledge the work of: Mercedes Abengózar Carneros, Juan Carlos González Moreno, Javier Leach Albert, Narciso Martí Oliet, Juan M. Molina Bravo, Ernesto Pimentel Sánchez, María del Mar Roldán García, and José J. Ruz Ortiz.

Chapter 1

Getting Started

1.1 Supported Platforms and System Requirements

The current version $\mathcal{TOY}$ works on most operating systems, which are the ones supported by SICStus Prolog [36]. This Prolog system (version 3.8.4 or higher) is required whenever you want to use $\mathcal{TOY}$ from a Prolog interpreter as will be described in the next section. Anyway, you can also use the binary distributions which do not require any installed Prolog system. In this case, $\mathcal{TOY}$ has been tested to work on the following operating systems: Windows 98/NT/ME/2000/XP/2003, Linux and Solaris.
In addition, some features of Toy need additional software:

Feature	Requirements
Graphical I/O	Tcl/Tk
Declarative Debugger	Java Runtime Environment 1.3 or higher

Notice that these extra requirements are not needed for installing and running $\mathcal{TOY}$, but only for using the corresponding features. Tcl/Tk come pre-installed on most *nix systems, as well as on Mac OS X; nevertheless, it can be downloaded from `http://www.tcl.tk/software/tcltk/`. The Java Runtime Environment is part of most of the modern OS distributions. It can be otherwise downloaded form `http://java.com`.

1.2 Downloading and Running $\mathcal{TOY}$

The latest version of the system can be downloaded from `http://toy.sourceforge.net`. The same distribution file is valid for all the platforms. After downloading the file and unzipping it, the system is ready; no further installation steps are needed. $\mathcal{TOY}$ can be alternatively started as follows:

- From the binary distribution: Move to the folder `toy` and, depending on the OS:
  - Windows. Execute `toywin`
  - Linux. Execute `toylinux`

  - Solaris. Execute `toy`

- From the source distribution (regardless of the OS):

  1. Move to the `toy` folder
  2. Start SICStus Prolog
  3. Type | `?- [toy].` (where | `?-` is the SICStus prompt).

1.3 First Steps in $\mathcal{TOY}$

After starting the system, the following prompt is shown:

```
Toy>
```

At this point, the system is ready to work. For example, using the addition (predefined function `+`) and the equality (predefined constraint `==`), we can submit the *goal*:

```
Toy> X == 3+5
```

The system solves the goal producing the following *computed answer*:

```
Toy> X == 3+5
      { X -> 8 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
```

meaning that the replacement of the variable `X` by `8` in the goal expression satisfies the goal. At this stage, $\mathcal{TOY}$ informs that it has found a solution and asks whether the user does (`y`) or does not (`n`) require the next solution, asks for a debugging session (`d`) or requires all the pending solutions (`a`). `y` is the default request, so that typing Intro is equivalent to request the next solution.
Notice that the symbol `==` is used instead of `=` in order to distinguish the syntactical strict equality from the function definition operator, respectively.
Goals and computed answers are discussed in Section 2.13 (pg. 42), but at the moment we can introduce some initial concepts:

- `X` is a *logical variable*, represented in $\mathcal{TOY}$ by identifiers starting with an uppercase letter.
- The symbol `==` stands for the predefined strict *equality constraint*.
- In order to solve a goal $e_1 == e_2$, the system will look for values of the variables occurring in the *expressions* e_1, e_2 such that both can be reduced to the same *pattern*. The notions of expressions and patterns are introduced more formally in Section 2.1 (pg. 18).

Now type `n` (standing for *no*) to indicate that we do not need more solutions for the goal. In fact there are no more solutions, so typing `y` (*yes*) will have the same effect. The answer `d` (*debug*) is used to indicate that the user detects that the answer is incorrect (which obviously is not the

case) and we want to start the debugger. The debugger is discussed in detail in Chapter 6 (pg. 142) .
It is also possible to directly evaluate (narrow [22]) an expression by preceding the expression by the symbol `>`. For example, the evaluation of the following goal:

```
Toy> > 3+5
```

displays the result:

```
8
```

To quit $\mathcal{TOY}$ at any moment, simply type:

```
Toy> /q
```

1.4 Compiling and Loading Programs

$\mathcal{TOY}$ programs can be written with any text editor. The default extension of source program files in $\mathcal{TOY}$ is `.toy` and the system will associate this extension to every file name if there is not another explicit extension. Nevertheless, any explicit extension is allowed.
Let us try to compile and run a simple program:

1. Create a new text file with any text editor. Let us assume that the file is saved with name `insert.toy` in the folder `examples` which you can find in the $\mathcal{TOY}$ distribution (of course, any other name and place is also allowed).

2. Using the text editor write the following short program:

   ```
   insert:: A -> [A] -> [A]
   insert X [] = [X]
   insert X [Y | Ys] = [X, Y | Ys]
   insert X [Y | Ys] = [Y | insert X Ys]
   ```

   The code of the program consists of a single function `insert`. The first line is the *type declaration* of the function (which is not compulsory), and the three next lines are the *rules* defining the function. Types and functions are explained in Sections 2.2 (pg. 19) and 2.6 (pg. 26), respectively.

3. After saving the file `insert.toy` we are ready to compile the program. But first we must move to the directory where the file has been stored. For this, type:

   ```
   Toy> /cd(examples)
   ```

   in order to change to the folder `examples`.

4. Compile the program by typing:

```
Toy> /compile(insert)
```

If there is some error, the compilation will stop with a (hopefully) informative error message. If so, fix the error and repeat the command `/compile(insert)` until no error is found.

5. Load the compiled file with:

```
Toy> /load(insert)
```

The program is ready to be used.

6. Now we can submit a goal as:

```
Toy> insert 0 [1,2,3] == R
```

and $\mathcal{TOY}$ will yield a first computed answer:

```
      { R -> [ 0, 1, 2, 3 ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
```

If we type `y`, the system will provide another computed answer. In this way we can check that the goal has 4 computed answers, which correspond to the 4 possible forms of inserting 0 in the list `[1,2,3]`. This an example of an *non-deterministic* computation.

7. Type `/q` when you want to leave the system.

Some remarks about this session:

- $\mathcal{TOY}$ uses Prolog as target code; so, in the process of compilation, the user programs are translated into Prolog code. The compilation of a source file <*filename*>`.toy` produces the file <*filename*>`.pl` that contains the code corresponding to the translation of the functions and predicates in <*filename*>`.toy`, together with other information.
- As we have seen, in order to compute a goal from a program, it is needed to 1) compile the program, and 2) to load the target code. These processes are performed by means of three commands:
  - `/compile(`<*filename*>`)` compiles the file <*filename*> (<*filename*>`.toy` if there is not an explicit extension in the name) and produces the file <*filename*>`.pl` as a result. This target file is not loaded; so, it is not possible to evaluate goals from this program yet. The compilation of a program implies several kinds of analysis of the source code. The system checks the syntax, the functional dependencies and the types. During these phases, the system delivers messages about the evolution of the process and detected errors. If there is not any error, the target code is produced.

– `/load(`<*filename*>`)` loads the target code of the program <*filename*>`.toy` (previously compiled with `/compile`). Then, goals can be evaluated from this program.

– `/run(`<*filename*>`)` compiles and loads the program. This is the most frequently used command for processing programs. Therefore, in the example above, we could have tried simply:

```
Toy> /run(insert)
```

Note about Loaded Code Each time a code (a compiled code from either a user program or a system library – CFLP($\mathcal{R}$), CFLP($\mathcal{FD}$), IO File or IO Graphic) is loaded, the definitions previously loaded for user programs are removed. This means the following. First, if you want to load two different programs, you have to include one inside the other (via the `include` statement – see Section 2.9 – or pasting the contents of one file to the other). And, second, if you load any system library, the user program definitions are lost, so you have to reload the user program.

If you are interested in trying more examples of $\mathcal{TOY}$ programs, look in the directory `examples` and read Appendix A (pg. 153). Chapter 2 (pg. 18) explains in detail all the components that can occur in a $\mathcal{TOY}$ program.
From the previous discussion it is easy to notice that at the $\mathcal{TOY}$ prompt you can type not only goals but also *commands* line `/q, /compile`, etc., which always start with a symbol `/`. Next section explains $\mathcal{TOY}$ commands.

1.5 Commands

1.5.1 Syntax

$\mathcal{TOY}$ provides an easy command interpreter for manipulating (editing, compiling, loading, etc.) programs at the system prompt. Commands in $\mathcal{TOY}$ must be preceded by the special symbol `/`. Type `/h` for an on-line reference of the available commands.
Arguments of commands, when used, must be enclosed between parentheses (both '`(`' and '`)`') and there can not be any blank between the name of the command and the symbol `(`. As an example, the following commands are recognized by the system:

```
/compile(my_program)
/   compile(  my_program        )
/load(one_of_my_programs)
/system(cd ..)
/cd(..)
/cd(/home/local/bin)
/q
/help
```

but the following are not valid commands:

```
/compile (my_program)
/cd
/cd ../..
```

Next sections describe the available commands.

1.5.2 Interaction with the Operating System

There are three special commands for interacting with the underlying operating system:

- `/q`, `/quit`, `/e` or `/exit` end the $\mathcal{TOY}$ session
- `/cd(`<*dir*>`)` sets the directory <*dir*> as the current directory
- `/cd` changes the current working directory to the distribution root where the system was installed
- `/pwd` Displays the current working directory
- `/ls` Displays the contents of the working directory
- `/ls(`<*dir*>`)` Displays the contents of the given directory <*dir*>
- `/system(`<*comm*>`)` submits the command <*comm*> to the operating system (to the corresponding shell). For example, for copying a file:

  ```
  /system(cp prog.toy /home/users/axterix/.)
  ```

  This command allows to interact with the operating system ant it accepts any valid input at the OS command prompt. In particular, the command `/cd(`<*dir*>`).` can be seen as a shorthand for `/system('cd` <*dir*>`').`;

  There is a shorthand for executing operating system commands: Simply type */comm*, wherever *comm* is not recognized as a $\mathcal{TOY}$ command.

1.5.3 Compiling and Loading Programs

The following commands are used for compiling and loading programs:

- `/compile(`<*file*>`)` Compiles the file <*file*>.toy
- `/load(`<*file*>`)` Loads the file <*file*>.pl provided that <*file*>.toy was compiled
- `/run(`<*file*>`)` Compiles and loads the file <*file*>.toy
- `/cut` Enables compilation with dynamic cut optimization
- `/nocut` Disables compilation with dynamic cut optimization

1.5.4 Information about Programs

$\mathcal{TOY}$ provides two commands for obtaining information about functions and predicates defined in a program. The first one displays the type of an expression without evaluating it:

- `/type(`<*expr*>`)` Shows the type of the expression <*expr*> according to the definitions of the current session. Consider the following examples:

```
Toy> /type(div)
div::int -> int -> int

Elapsed time: 0 ms.

Toy> /type(1)
$int(1)::real

Elapsed time: 0 ms.

Toy> /type(1+2)
$int(1)+$int(2)::real

Elapsed time: 0 ms.

Toy> /type(1+2==3)
$int(1)+$int(2)==$int(3)::bool

Elapsed time: 0 ms.

Toy> /type(+)

Line 1. Error 11: Error in expresion. Unexpected operator +.
Toy> /type((+))
+::real -> real -> real

Elapsed time: 0 ms.
```

  Note that because associativity, we have to type parentheses enclosing operators in order to determine their types, as illustrated in the last example above.

- `/tot` Enables totality constraints as part of the system answers
- `/notot` Disables totality constraints as part of the system answers
- `/tree(`<*filename*>`,`<*fun*>`)` This command is useful only for experienced users with knowledge about the evaluation strategy used by $\mathcal{TOY}$, and it shows the definitional tree [22] of the function <*fun*> in the file <*filename*>, that must have been previously compiled, where <*filename*> will be the desired name of the dumped state.

1.5.5 Commands for Constraint Solvers

- `/cflpr` Enables arithmetical constraints over real numbers.

  The effect of this command is the loading of the library for constraint solving over real numbers. Once executed, the system prompt changes to `Toy(R)>`, and equations as the following can be solved:

```
Toy(R)> X + 3 == 5
      { X -> 2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

  or more complex equation systems as:

```
Toy(R)> 2*X - 3*Y - Z == 3, X + 6*Y == 2*Z, Z - 2*Y == X
      { X -> -2,
        Y -> -1,
        Z -> -4 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

  For a detailed explanation of this kind of constrains see Section 2.16.

- `/nocflpr` Disables arithmetical constraints over real numbers

- `/cflpfd` Enables constraints over finite domains.

  The effect of this command is the loading of the library for constraint solving over finite domains. Once executed, the system prompt changes to `Toy(FD)>`, and $\mathcal{TOY}$ can solve goals as:

```
Toy(FD)> domain [X,Y] 1 10, X #> Y, Y #< 3
```

  This goal asks for integral values of X, Y belonging to the interval $[1, 10]$, and that $X - 3 > Y$ (the symbols `#-` and `#>` are used in finite domain constraints instead of `-` and `>`, respectively). The answer provided by the system is:

```
      { Y #< X,
        X in 2..10,
        Y in 1..2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

An exhaustive description of the finite domain solver is carried out in Chapter 3 (page 66).

- `/nocflpfd` Disables finite domain constraints

Constraint solvers should only be activated when needed; otherwise, the system performance would be slightly degraded.
The solver for syntactical constraints (equality and disequality) over constructed terms are always enabled.

1.5.6 Miscellanea

This section collects the commands which do not fit into former sections.

- `/io_file` loads the library IO File, which provides functions to handle text files
- `/noio_file` unloads this library
- `/io_graphic` loads the library IO Graphic, which provides functions for graphical user interfaces
- `/noio_graphic` unloads this system module.
- `/prop` enables propagation.
- `/noprop` disables propagation.
- `/statistics(`$< level >$`)` Sets the level $< level >$ of statistics:
  - 0: no statistics
  - 1: runtime
  - 2: runtime and hnf counter
  - 3: full statistics
- `/statistics` Displays the current level of statistics
- `/prolog(`$<goal>$`)` Executes the Prolog goal $<goal>$
- `/status` Informs about the loaded libraries, as follows:

```
Toy> /status
Libraries
=========
CFLPR      : not loaded
CFLPFD     : not loaded
IO File    : not loaded
IO Graphic : not loaded
```

- `/version` displays the $\mathcal{TOY}$ system version number, as follows:

```
Toy> /version
% :: B Sonia
2.3
% :: E Sonia
```

- `/help`, `/h` Shows the system help about the command interpreter, including evaluation and debugging

1.5.7 Defining New Commands

In the previous sections, we have described the collection of commands provided by $\mathcal{TOY}$. This repertory is small, simple and enough for accessing the capabilities of the system. Nevertheless, there are some other commands that are not essential but can allow a more friendly use of $\mathcal{TOY}$. The user can define commands for accessing the operating system from the $\mathcal{TOY}$ prompt. The file `'toy.ini'` included with the distribution contains such definitions in the form of Prolog facts. The user can edit and modify this file. For example, for defining a new command `/dir` that shows every file in the current directory, the file `toy.ini` includes the following Prolog fact:

```
command(dir,0,[],["ls -l"]).
```

The first argument is the name of the command (`dir`), the second is the number of arguments (0, in this case), the third is the list of arguments represented as Prolog variables (starting with capital letters) that will be used by the command, and the last one is the list of strings that must be appended for building the command (only one in this case). With this definition, the command `/dir` will work in the next $\mathcal{TOY}$ session.
Another useful command may be for accessing a text editor. For example, if the user wants to use the editor *emacs*, the following line must be included in the file `toy.ini`:

```
command(edit,1,[File],["emacs ",File,".toy &"]).
```

In this case, the name of the command is `edit`, it uses one argument (`File`) and the effect will be to edit the corresponding file with extension `.toy`.

Chapter 2

The Language

A $\mathcal{TOY}$ program is a collection of definitions of various entities, including types, functions, predicates, and operators. Once a program has been successfully compiled, users can interact with the $\mathcal{TOY}$ system by proposing goals. $\mathcal{TOY}$ computations solve goals and display computed answers, which can include the result of a functional reduction and/or a constraint. In particular, answer constraints can represent bindings for logic variables, as in answers computed by a Prolog system. The different sections of this chapter explain the features available in $\mathcal{TOY}$ for writing programs and solving goals.

2.1 Expressions

As every programming language, $\mathcal{TOY}$ uses expressions built from operation symbols and variables. Following Prolog conventions, *variables* in $\mathcal{TOY}$ are represented by identifiers starting with an uppercase letter. Following the conventions adopted by Haskell and other functional languages, operation symbols are classified into two disjoint sets: *data constructors*, corresponding to so-called *functors* in Prolog, and *defined functions*. $\mathcal{TOY}$ uses identifiers starting with a lowercase letter (as well as some special symbols) both for data constructors and for defined functions. Data constructors are easily recognized, because they are either predefined or introduced in some *datatype declaration* (see Section 2.3). *Defined predicates* in $\mathcal{TOY}$ are viewed as a particular case of defined functions, whose result is boolean. As in functional languages, expressions which include defined functions can be evaluated in $\mathcal{TOY}$ to compute a result, while those expressions which are built only from data constructors can be seen as symbolic representations of computed results.
In the rest of this document we often use capital letters X, Y, Z, ... (possibly with subscripts) as variables. Moreover, we use the notation $c \in DC^n$ to indicate that c is a data constructor which expects n arguments, and the notation $f \in FS^n$ to indicate that f is a defined function symbol whose definition expects n formal parameters. The number n is called the *arity* of c resp. f. Constant values belonging to primitive datatypes can be viewed as nullary data constructors in DC^0. Under these conventions, the abstract syntax for $\mathcal{TOY}$ *expressions* is as follows:

$$e ::= X \mid c \mid f \mid (e\ e_1) \mid (e_1, \cdots, e_n)$$

Expressions $(e_1, \cdots, e_n)$ $(n \geq 0)$ represent tuples, while $(e\ e_1)$ represents the *application* of the

function denoted by e to the argument denoted by e_1. Following usual conventions, application associates to the left, and outermost parentheses can be omitted. Therefore, $((f\ X)\ ((g\ Y)\ Z))$ can be written more simply as $f\ X\ (g\ Y\ Z)$. In programming examples, we will use meaningful identifiers rather than these abstract notations. The concrete syntax of $\mathcal{TOY}$ expressions (described in Appendix D) supports some additional features, including numerals, `if - then - else` conditional expressions, and infix operators (see Section 2.7). However, λ-abstractions are not supported. Otherwise, $\mathcal{TOY}$ syntax is quite similar to Haskell [16] syntax, although some predefined symbols and conventions are different. Most notably, Haskell reserves identifiers starting with an uppercase letter for types and data constructors, while variable identifiers must start with a lowercase letter. In $\mathcal{TOY}$, identifiers starting with an uppercase letter are reserved for variables, as said before; while identifiers for types and data constructors must start with a lowercase letter.

Two important subclasses of expressions are *data terms*, defined as

$$t ::= X \mid (t_1, \cdots, t_n) \mid c\ t_1 \cdots\ t_n \quad (c \in DC^n)$$

and *patterns*, defined as

$$t ::= X \mid (t_1, \cdots, t_n) \mid c\ t_1 \cdots\ t_m \quad (c \in DC^n,\ 0 \leq m \leq n) \mid$$
$$f\ t_1 \cdots\ t_m \quad (f \in FS^n,\ 0 \leq m < n)$$

Data terms in $\mathcal{TOY}$ correspond to the notion of pattern in typical functional languages such as Haskell. Patterns in $\mathcal{TOY}$ are more general. For instance, assume a program with functions `times`, `twice` $\in FS^2$, defined as follows:

```
times X Y  = X * Y

twice F X  = F (F X)
```

Then we can build the pattern `twice (times 2)`, which is not a data term (neither a pattern in a Haskell-like language). More generally, patterns of the form $f\ t_1 \cdots\ t_m$ $(f \in FS^n,\ 0 \leq m < n)$ are not data terms; they are called *functional patterns* or also *higher-order patterns*, because they represent functions. When comparing patterns, $\mathcal{TOY}$ relies on syntactic equality. For instance, the two patterns `twice (times 2)` and `times 4` are regarded as different, although they behave in the same way as functions. Therefore, the representation of functional values as patterns in $\mathcal{TOY}$ is *intensional* rather than *extensional*. As a consequence, pattern unification has the same behaviour and complexity as syntactic unification of terms in Prolog.

2.2 Types

$\mathcal{TOY}$ is a typed programming language, based essentially on the Damas-Milner polymorphic type system [7]. Programs are tested for well-typedness at compile time by means of a type inference algorithm (see Appendix E). In particular, each occurrence of an expression in a $\mathcal{TOY}$ program has a type that can be determined at compile time. Syntactically, types are built from *type variables* and *type constructors*. Any identifier starting with an uppercase letter can be used as a type variable, while identifiers for type constructors must start with a lowercase letter. Type constructors are introduced in *datatype declarations*, along with data constructors (see Section 2.3). Primitive types (such as `bool` and `int`, see Section 2.4) can be viewed as type

constructors of arity 0. Assuming the abstract notation $A,\ B\ldots$ for type variables and $\delta \in TC^n$ for type constructors of arity n, the syntax of $\mathcal{TOY}$ types can be defined as follows:

$\tau ::= A \mid \delta\ \tau_1 \cdots\ \tau_n\ \ (\delta \in TC^n) \mid (\tau_1, \cdots, \tau_n) \mid (\tau_1$ `->` $\tau)$

Here, $(\tau_1, \cdots, \tau_n)$ stands for the *cartesian product* of the types $(\tau_1, \cdots, \tau_n)$, while a *functional type* of the form $(\tau_1$ `->` $\tau)$ stands for the type of all functions which can be applied to arguments of type τ_1 and return results of type τ. Syntactically, `->` behaves as a predefined, binary type constructor, written in infix notation. Following common practice, we assume that `->` associates to the right and omit parentheses accordingly. For instance, the type

```
((int -> bool) -> ((int -> int) -> (int -> int)))
```

can be abbreviated as

```
(int -> bool) -> (int -> int) -> int -> int.
```

The notation $e\ ::\ \tau$ is used to indicate that expression e has type τ. For instance, some typing rules have the following form:

$IF\ e_i\ ::\ \tau_i$ for all $1 \leq i \leq n\ THEN\ (e_1, \cdots, e_n)\ ::\ (\tau_1, \cdots, \tau_n)$

$IF\ e\ ::\ \tau_1$ `->` τ and $e_1\ ::\ \tau_1\ THEN\ e\ e_1\ ::\ \tau$

Using these and other rules, the *type inference algorithm* described in Appendix E can decide whether a given expression e is well-typed. In the positive case, the algorithm computes the *most general type* τ of e (also called *principal type*) along with a set TA of type assumptions $X\ ::\ \tau_X$ for the variables X occurring in e, such that $e\ ::\ \tau$ follows from the assumptions in TA and the type inference rules. For instance, given the expression

```
X /\ not Y
```

(involving two boolean operations), the type inference algorithm computes

```
X /\ not Y :: bool
```

under the assumptions `X :: bool, Y :: bool`.

The type inference algorithm also checks whether the function definitions given in the text of a $\mathcal{TOY}$ program are well-typed. In the positive case, the algorithm infers a principal type of the form

$f\ ::\ \tau_1$ `->` $\cdots$ `->` τ_n `->` τ

for each n-ary function symbol $f \in FS^n$ occurring in the program. *Boolean functions* with type

$p\ ::\ \tau_1$ `->` $\cdots$ `->` τ_n `->` `bool`

are called also *predicates* in $\mathcal{TOY}$.

Note that two functions

$f\ ::\ \tau_1$ `->` τ_2 `->` τ and $g\ ::\ (\tau_1, \tau_2)$ `->` τ

have related, but different types. Function f can be applied to an argument of type τ_1 and returns a function that expects another argument of type τ_2, returning in turn a result of type τ. On the other hand, g expects a pair of product type (τ_1, τ_2) as argument, and returns a result of type τ. Assumming that the final result is always the same, i.e.

$f\ X\ Y\ =\ g\ (X, Y)$ for all $X\ ::\ \tau_1,\ Y\ ::\ \tau_2$,

one says that function f is the *curried version* of function g. The word *curried* comes from *Haskell Curry*, one of the first mathematicians who discovered this idea.

The principal type of a function can include type variables. In this case the function, as well as its type, is called *polymorphic*. A polymorphic function has an implicit universal quantification

over the type variables occurring in its principal type. Different substitutions of types for those type variables give rise to different *instances* of the function. For example, from the definition of `twice` shown in Section 2.1, the $\mathcal{TOY}$ system infers the principal type

```
twice :: (A -> A) -> A -> A.
```

As an instance of this polymorphic type, we get also

```
twice :: ((A -> A) -> A -> A) -> (A -> A) -> A -> A.
```

From the two previous facts, it follows that `twice twice` is also a well-typed expression, with principal type

```
twice twice :: (A -> A) -> A -> A.
```

More generally, different uses of a polymorphic function within one and the same expression can correspond to different instances of its principal type. In this way, polymorphism promotes more *generic* and thus *reusable* programs.

Finally, the following fact is interesting. A function f with principal type f `::` τ_1 `->` $\cdots$ `->` τ_n `->` τ can be used to build well-typed expressions $e \equiv f\ e_1 \cdots\ e_m$ with different choices for the number m of arguments:

- In the case $m = n$, e `::` τ', provided that there is some instance f `::` τ'_1 `->` $\cdots$ `->` τ'_n `->` τ' of f's principal type, such that e_i `::` τ'_i for all $1 \leq i \leq n$. For example, `twice not true :: bool`.

- In the case $m < n$, e is called a *partial application* of f. Moreover, e `::` τ'_{m+1} `->` $\cdots$ `->` τ'_n `->` τ', provided that there is some instance f `::` τ'_1 `->` $\cdots$ `->` τ'_n `->` τ' of f's principal type, such that e_i `::` τ'_i for all $1 \leq i \leq m$. For example, `twice not :: bool -> bool`.

- In the case $m > n$, e is called an application of f with *additional arguments*. Moreover, e `::` τ', provided that there is some instance f `::` τ'_1 `->` $\cdots$ `->` τ'_m `->` τ' of f's principal type, such that e_i `::` τ'_i for all $1 \leq i \leq m$. This may be possible if τ is a type variable or a functional type. For example, using the instance obtained from `twice`'s principal type by the substitution

  `A` $\mapsto$ `(bool -> bool) -> bool -> bool`

  one obtains `twice twice not true :: bool`.

2.3 Datatype Definitions

Datatypes, also called *constructed types*, are defined along with data constructors for their values. Datatype definitions in $\mathcal{TOY}$ are introduced by the keyword `data` and have the following schematic syntax:

`data` $\delta\ \overline{A}$ = $c_0\ \overline{\tau}_0$ | $\cdots$ | $c_{m-1}\ \overline{\tau}_{m-1}$

where δ is the *type constructor*; $\overline{A}$ is a shorthand for $A_1 \cdots A_n$ ($n \geq 0$), a sequence of different type variables acting as formal type parameters; c_i ($0 \leq i < m$) are the *data constructors* for values of type $\delta\ \overline{A}$; and $\overline{\tau}_i$ is a shorthand for $\tau_{i,1} \cdots \tau_{i,n_i}$, a series of types corresponding to the arguments of c_i. The types $\tau_{i,j}$ must be built from the type variables $\overline{A}$, predefined types and user-defined type constructors, according to the syntax of types explained in Section 2.2.

Different datatype definitions must declare disjoint sets of data constructors. The *principal type* of any data constructor c occurring in a program is determined by the datatype definition where c has been declared. In terms of the general scheme above:

c_i :: $\tau_{i,1}$ -> $\cdots$ -> τ_{i,n_i} -> δ $\overline{A}$ (for all $0 \leq i < m$)

A datatype is called *polymorphic* (or also *generic*) iff the number n of formal type parameters in its definition is greater than zero, and *monomorphic* otherwise. Moreover, a datatype δ is called *recursive* whenever δ itself is involved in the type of the arguments of at least one of the data constructors declared in δ's definition.

To illustrate the possibilities of datatype definitions, we present a series of typical examples. The simplest datatypes are *enumerated types*, whose definition declares finitely many *constant constructors*. For instance, the datatype of boolean values is predefined as

```
data bool = false | true
```

and the following enumerated type represents the days of the week:

```
data weekDay = monday | tuesday | wednesday | thursday |
               friday | saturday | sunday
```

Next, we show a polymorphic datatype with one parameter and a single data constructor, whose values bear the same information as pairs of type `(A,A)`. Nevertheless, $\mathcal{TOY}$ treats the two types as different.

```
data pairOf A = pair A A
```

The next example is a datatype with two data constructors, which represents the *disjoint union* of the types given as parameters:

```
data either A B = left A | right B
```

Another useful datatype is `maybe A`, which behaves as the disjoint union of the type `A` given as parameter with a special value `nothing` which can be used as an error indication. It can be defined as follows:

```
data maybe A = nothing | just A
```

Finally, we show some examples of *recursive* datatype definitions. Type `nat` represents Peano natural numbers built from a constant `zero` and a unary constructor `suc`.

```
data nat = zero | suc nat
```

Different kinds of trees can be represented by means of suitable recursive datatypes. For instance, binary trees storing information of type `A` at their leaves can be defined as follows:

```
data leafTree A = leaf A | node (leafTree A) (leafTree A)
```

Mutually recursive datatype definitions are also possible. They can be presented in any textual order. The mutually recursive types `even` and `odd` defined below correspond to even and odd natural numbers:

```
data even = evenZero | evenSuc odd
data odd  = oddSuc even
```

Note that

```
oddSuc evenZero :: odd     suc zero :: nat     1 :: int
```

are treated as different values belonging to different types in $\mathcal{TOY}$. More generally, different datatypes are always disjoint and different from predefined types.

2.4 Predefined Types

Predefined types in $\mathcal{TOY}$ include

- `bool`, defined as `data bool = true | false`.
- `int`, the type of single-precision integers.
- `real`, the type of single-precision floating-point numbers.
- `char`, the type of characters, which must be written by enclosing them in single quotes, like e.g. `'a'`, `'@'`, `'9'` or `'-'`. Some values of this type are *control characters* rather than visible signs. These are written in a special way, as follows:

Toy character	*meaning*
`'\\'`	backslash
`'\''`	quote
`'\"'`	double quote
`'\n'`	new line
`'\r'`	carriage return
`'\t'`	horizontal tabulator
`'\v'`	vertical tabulator
`'\f'`	form feed
`'\a'`	alarm
`'\b'`	backspace
`'\d'`	delete
`'\e'`	escape

- `[A]`, the recursive datatype of lists with elements of type `A`. It behaves as if it were defined by

```
data [A] = [] | A : [A]
```

with the constant constructor `[]` for the *empty list* and the infix (right associative) data constructor `:` to build a *non-empty* list `(X:Xs)` with *head* `X :: A` and *tail* `Xs :: [A]`. The Prolog notation `[X|Xs]` is also supported as an alias of `(X:Xs)`. As in Prolog and Haskell, the list notation $[elem_0, elem_1, \cdots, elem_{n-1}]$ is allowed as a shorthand for the reiterated application of the list constructor. For instance, `[1,2,3]` is viewed as shorthand for `1:2:3:[]`.

- *Product types* and *tuples* are supported by $\mathcal{TOY}$, as explained in Section 2.2.

Of course, predefined types, products, functional types, and user-defined types can be freely combined to build more complex types, following the type syntax explained in Section 2.2. For example:

- `[leafTree (char,int)]` is the type of lists of binary trees storing pairs of type `(char,int)` at their leaves.
- `[int] -> int` is the type of functions which accept a list of integers as an argument and return an integer result.
- `[int -> int]` is the type of the lists whose elements are functions of type `int -> int`.
- `(A -> B) -> leafTree A -> leafTree B` is the type of polymorphic functions which expect two parameters (a function of type `A -> B` and a tree of type `leafTree A`) and return as result another tree of type `leafTree B`.

The last two items above illustrate the use of functions as first-class values that can be stored in data structures, passed as parameters to other functions, and also returned as results.

2.5 Type Synonyms

Type synonyms, also called *type alias*, allow to declare a new identifier as shorthand for a more complex type. This facility helps to write more concise and readable programs. Definitions of type synonyms are introduced by the keyword *type* and have the following schematic syntax:

`type` $\nu\ \overline{A}\ =\ \tau$

where ν is the identifier chosen as alias; $\overline{A}$ is a sequence of different type variables acting as formal type parameters; and τ is a type built from the type variables $\overline{A}$, built-in types, and user-defined type constructors.

The effect of a `type` definition like this is to declare $\nu\ \overline{A}$ as an alias of τ. In fact, type synonyms work like parametric user-defined *macros*, that are eliminated at compile time. Therefore, definitions of type synonyms cannot be recursive. As long as this restriction is respected, the definition of a type synonym may depend on other type synonyms.

As a simple example, we can define a type alias for representing coordinates as numbers, and points of the plane as pairs of coordinates:

```
type x-coordinate = real
type y-coordinate = real

type point = (x-coordinate, y-coordinate)
```

Another example, involving a type parameter, is the definition

```
type myPairOf A = (A,A)
```

In contrast to the *datatype* `pairOf A` shown in Section 2.3, this definition does not introduce data constructors. Values of types `pairOf A` and `myPairOf A` are treated as different, although they bear equivalent information.
A very useful example of synonym is the type for character strings, defined as follows:

```
type string = [char]
```

The previous definition is currently *not predefined* in $\mathcal{TOY}$. It is left to the user's choice to include it in any program, if desired. In any case, $\mathcal{TOY}$ supports a special syntax for lists of characters, which can be written as a text enclosed in double quotation marks. For instance, the following are three different representations of the same string:

```
"Hello, World!"
['H','e','l','l','o',' ','W','o','r','l','d','!']
'H':'e':'l':'l':'o':' ':'W':'o':'r':'l':'d':'!':[]
```

In particular, the *empty string* can be written either as `[]` or as `'' ''`. *Unitary strings* consisting of a single character must not be confused with that character. For instance, `''a'' :: string` is not the same as `'a' :: char`.
As a last example, we define a type synonym `person` along with three auxiliary types:

```
type person   = (name, address, age)

type name     = string

type address  = string

type age      = int
```

Note that persons with negative age will not give rise to a type error, because `person` is treated as a shorthand for `(string, string, int)`.

2.6 Function Definitions

A function definition in $\mathcal{TOY}$ consists of an optional *type declaration* and one or more *defining rules*, which are possibly conditional rewrite rules with optional local definitions. The schematic form of such a defining rule is:

$$\begin{array}{lcl} f & :: & \tau_1 \texttt{ -> } \cdots \tau_n \texttt{ -> } \tau \\ & & \cdots \\ f\ t_1 \cdots t_n & = & r \texttt{ <== } C_1, \cdots, C_m \texttt{ where } D_1, \cdots, D_p \\ & & \cdots \end{array}$$

Such a definition is understood to introduce a defined function $f \in FS^n$ of arity n. For each defining rule, the *left-hand side* $f\ t_1 \cdots t_n$ must be *linear*, i.e., it cannot include repeated variables. The *formal parameters* t_i must be patterns[1]. The *right-hand side* r can be any expression built from variables, built-in operations, data constructors, and functions occurring in the program. C_i are *conditions*, and D_i are *local definitions*; more details about them will be explained below. Often a defining rule needs no conditions, in which case the symbol `<==` can be omitted. Similarly, the reserved word `where` can be omitted if there are no local definitions. For the moment, we assume that each condition must have the form $e_1 \diamondsuit e_2$ for some $\diamondsuit \in \{==, /=, <, >, <=, >=\}$, e_1 and e_2 being expressions[2]. Conditions of the form $b == \mathit{true}$ can be abbreviated as b. *Strict equality* conditions $e_1 == e_2$ are solved by evaluating the two expressions e_1 and e_2 to a common finite value, represented by a pattern; while *disequality conditions* $e_1 /= e_2$ are solved by evaluating e_1 and e_2 to conflicting values. In the case of expressions e_i whose evaluation yields an infinite value (e.g., an infinite list), a strict equality never holds, and the attempt to solve it can lead to a non-terminating computation. On the other hand, lazy evaluation (explained below) allows to solve disequality conditions without demanding complete evaluation of the expressions e_i. A more formal explanation of strict equality and disequality can be found in Section 2.16 and Appendix F. *Inequality conditions* $e_1 \diamondsuit e_2$ with $\diamondsuit \in \{<, >, <=, >=\}$ work for numbers of type `int` and `real`, with the obvious arithmetic meaning. In addition to their rôle in conditions, the symbols $\diamondsuit \in \{==, /=, <, >, <=, >=\}$ can be freely used as boolean functions (in infix notation).
Each local definition D_i must have the form $s_i = d_i$, where d_i is an arbitrary expression and s_i is a *linear flat pattern* having either the form X (variable) or $(c\,X_1 \cdots, X_l)$ (c being an l-ary data constructor). Moreover, the sequence $D_1, \cdots, D_p$ of local definitions must be such that the following three properties are satisfied:

1. For all $1 \leq i \leq p$: s_i and $(f\,t_1 \cdots t_n)$ share no variables.
2. For all $1 \leq i < j \leq m$: s_i and s_j share no variables.
3. For all $1 \leq i \leq m, 1 \leq j \leq i$: s_i and d_j share no variables.

Intuitively, a local condition $s_i = d_i$ has the effect of matching the pattern s_i to the value obtained by evaluating the expression d_i, and binding the variables occurring in s_i accordingly. Therefore, local definitions can be said to define values for the variables occurring in the patterns

[1] Actually, *transparent patterns*. See Section 2.14 for an explanation of this technical notion.
[2] More general conditions will be introduced in Section 2.16.

s_i. Items 1. and 2. above require that the locally defined variables must be different from each other and away from the variables occurring in the rule's left-hand side, that act as formal parameters. Moreover, item 3. ensures that the variables defined in local definition number i can be used in local definition number j only if $j > i$. In particular, this means that local definitions cannot be recursive.
The intended meaning of a defining rule $f\ t_1 \cdots t_n = r$ `<==` $C_1, \cdots C_m$ `where` $D_1, \cdots, D_p$ is the same as in functional programming languages, namely: an expression of the form $f\ e_1 \cdots e_n$ can be reduced by reducing the actual parameters e_i until they match the patterns t_i, checking that the conditions are satisfied, and then reducing the right hand side r (affected by the pattern matching substitution); all this by using the local definitions to obtain the values of locally defined variables. In particular, this means that the equality symbol '=' in defining rules is implicitely oriented from left to right. The $\mathcal{TOY}$ system can apply the left-to-right oriented defining rules in order to compute the value of expressions, using *lazy evaluation*. This means that subexpressions occurring as actual parameters in function calls are not evaluated eagerly, but *delayed* until their values are eventually needed. The computations which obtain the values of locally defined variables are also delayed, until they are needed.
In addition to purely functional computations, $\mathcal{TOY}$ has also the ability to solve goals; see Section 2.13. Goal solving requires to combine lazy evaluation and the computation of bindings and constraints for logic variables. A more precise explanation of $\mathcal{TOY}$'s semantics is given in Appendices F and G.
Next, we show some definitions of simple functions, related to *list processing*. Note that the symbol `'%'` is used in $\mathcal{TOY}$ to start a *comment*, which is assumed to fill the rest of the current line. To write longer comments, it is also possible to enclose the commented text between the symbols `'/*'` (*open comment*) and `'*/'` (*close comment*). Nested comments are allowed.

```
% List recognizer.

null :: [A] -> bool
null []     = true
null (X:Xs) = false

% List selectors.

head :: [A] -> A
head (X:Xs) = X

tail :: [A] -> [A]
tail (X:Xs) = Xs

% Accessing the Nth element of a list.

infixl 90 !!

(!!) :: [A] -> int -> A
```

```
(X:Xs) !! N = if N == 0 then X else Xs !! (N-1)

% List concatenation.

infixr 50 ++

(++) :: [A] -> [A] -> [A]
[]     ++ Ys = Ys
(X:Xs) ++ Ys = X:(Xs ++ Ys)

% Flattenning a list of lists into a single list.

concat          :: [[A]] -> [A]
concat []        = []
concat (Xs:Xss)  = Xs ++ concat Xss

% Splitting a list in two parts.

take :: int -> [A] -> [A]
take N Xs = if N == 0 then [] else take' N Xs

take' N []     = []
take' N (X:Xs) = X : take (N-1) Xs

drop :: int -> [A] -> [A]
drop N Xs = if N == 0 then Xs else drop' N Xs

drop' N []     = []
drop' N (X:Xs) = drop (N-1) Xs

% Building an infinite list of consecutive integers.

from   :: int -> [int]
from N  = N : from (N+1)
```

In these examples we have used the *infix operators* `(++)`, `(!!)` and `(+)` and `(-)` as function symbols. See Section 2.7 for more information about infix operators. The definitions of `(!!)`, `take`, and `drop` also use the `if - then - else` notation. *Conditional expressions* of the form `if b then e1 else e2` are syntactic sugar for `if-then-else b e1 e2`, where the `if-then-else` function is predefined as follows:

```
if-then-else          :: bool -> A -> A -> A
if-then-else true  X Y = X
if-then-else false X Y = Y
```

The use of local definitions is illustrated by the following function `roots`, intended to compute the real roots of a quadratic equation $Ax^2 + Bx + C = 0$ with real coefficients. The definition of `roots` uses some predefined functions, explained in Section 2.8.

```
type coeff = real

type root  = real

roots       :: coeff -> coeff -> coeff -> (root,root)
roots A B C  = ((-B-D)/E,(-B+D)/E) <== E > 0
               where D = sqrt (B*B-4*A*C)
                     E = 2*A
```

As explained in Section 2.2, $\mathcal{TOY}$ uses a *type inference* algorithm at compile time to infer most general types for functions, on the basis of the function and the datatype definitions occurring in the program (see Appendix E). For the purposes of type inference, $\mathcal{TOY}$ assumes the following types for the operators involved in conditions:

```
(==), (/=)            :: A -> A -> bool

(<), (>), (<=), (>=) :: real -> real -> bool
```

In fact, $\mathcal{TOY}$ supports an implicit conversion of values of type `int` to type `real`. Therefore, in absence of user-given type declarations, the type inference algorithm always assumes the type `real` for the arguments and results of numeric functions. As an exception to this general rule, some of the built-in arithmetic operations require some arguments of type `int`; see Section 2.8 for details.
In case that some function definition is ill-typed, the $\mathcal{TOY}$ system raises *type error messages* and compilation is aborted. Otherwise, $\mathcal{TOY}$ compares the types inferred for defined functions with those declared in the program. For each defined function f there are three possible cases:

- There is no declared type for f, or the declared type is equivalent to the inferred type (up to a renaming of type variables). Then the inferred type is used.
- There is a declared type which is strictly less general than the inferred type. Then a warning to the user is emitted, but *the declared type is assumed*. This may prevent a succesful type inference for some other defined function. For example, the following program fragment would cause a type error, because the declared type of `length` requires lists of integers, which is not general enough to render the definition of `sameLength` well-typed. On the contrary, the inferred type of `length` is general enough to well-type `sameLength`.

  ```
  length       :: [int] -> int
  length []     = 0
  length (X:Xs) = 1 + length Xs

  sameLength      :: [A] -> [B] -> bool
  sameLength xs ys = length Xs == length Ys
  ```

- There is a declared type which is not unifiable with the inferred type. In this case the declared type is wrong. The system emits an error message, and compilation is aborted. For example, the following function definition would be rejected as ill-typed, because the declared type of `length` is incompatible with the inferred type.

```
length        :: [A] -> bool
length []     = 0
length (X:Xs) = 1 + length Xs
```

2.7 Operators and Sections

As in Haskell and many other programming languages, those function symbols used in *infix* notation are called *operators* in $\mathcal{TOY}$. For instance, the simple expression `X * 3` uses the primitive multiplication operator. In addition to the predefined operators (see Section 2.8), $\mathcal{TOY}$ users can introduce new operators by means of *operator declarations*. These start with one of the three keywords `infixr`, `infixl` or `infix` (intended for *right-associative*, *left-associative* and *non-associative*, respectively), followed by the operator's *precedence* and *name*. For instance, in order to declare right-associative operators for boolean conjunction and disjunction, one could write

```
infixr 40  /\

infixr 30  \/
```

These declarations inform the parser that `/\` must be given higher priority than `\/`. More generally, precedences are positive numbers, with bigger values standing for higher priorities. Moreover, the implicit operation which applies a function to its argument has always greater priority than any infix operator. For instance, the expression `f x + 3` is parsed as `(f x) + 3` rather than `f (x + 3)`.
Operators may also be used for data constructors. In such cases, the operator's name must start with the character `':'`. For instance, a datatype for complex numbers can be defined as follows:

```
infix 40 :+

data complex = real :+ real
```

$\mathcal{TOY}$ has two syntactic conventions used to relate the use of operators and ordinary functions:

- An operator symbol, enclosed between brackets, becomes a function identifier to be used in prefix notation. For instance, the two expressions `(+) 2 3` and `2 + 3` are equivalent.
- An ordinary function identifier, enclosed betwenn backward quotes, becomes an infix operator. For instance, the two expressions `mod X 2` and `X ‘mod‘ 2` are equivalent.

$\mathcal{TOY}$ also supports the useful notion of *section*, borrowed from languages of the Haskell family. For any binary operator $(\oplus)$ :: τ_1 -> τ_2 -> τ, sections are obtained by supplying one of the two missing arguments. More precisely, assuming given expressions a_1 :: τ_1, a_2 :: τ_2, one can build the sections $(a_1\oplus)$ :: τ_2 -> τ and $(\oplus a_2)$ :: τ_1 -> τ, which stand for the functions waiting for the missing argument. For instance, `(2*)` is a section of the multiplication operator; it behaves as a function which doubles its argument. Since $\mathcal{TOY}$ does not support λ-abstractions, sections and partial applications (see Section 2.2) are the two main tools available for expressing the computation of functional values.

2.8 Primitive and Predefined Functions

$\mathcal{TOY}$ has a number of *primitive functions* which are not defined by means of defining rules. Their definitions are included in a special system file called `basic.toy`. Currently, the primitive functions supported by the system are:

```
(+), (-), (*), min, max :: real -> real -> real

(^) :: real -> int -> real

(/), (**), log  :: real -> real -> real

div, mod, gcd :: int -> int -> int

round, trunc, floor, ceiling :: real -> int

toReal   :: int -> real

uminus, abs, sqrt, ln,
exp, sin, cos, tan, cot,
asin, acos, atan, acot,
sinh, cosh, tanh, coth,
asinh, acosh, atanh, acoth  :: real -> real

(<), (<=), (>), (>=) :: real -> real -> bool
(==), (/=) :: A -> A -> bool
```

Tables 2.1 and 2.2 show the intended meaning for each one of the primitive functions.
As we have remarked already in Section 2.6, the symbols ==, /=, >, <, >=, <= have a double rôle: they can be used to build conditions in defining rules, and also as operators denoting functions which return a boolean result. Both rôles can be distinguished by the context. As we will see in Section 2.16, they are conceptually different and not to be confused. A natural relation between the two rôles of strict equality and disequality is shown by the following definitions:

```
X == Y = true  <== X == Y
```

`+` XY	$X + Y$
`-` XY	$X - Y$
`/` XY	X/Y (real division)
`*` XY	$X * Y$ (real multiplication)
`**` XY	X raised to the power of Y
`^` XY	X raised to the (positive integer) power of Y, i.e., X^Y
`<` XY	$X < Y$
`<=` XY	$X \leq Y$
`>` XY	$X > Y$
`>=` XY	$X \geq Y$
`==` XY	$X = Y$
`/=` XY	$X \neq Y$
`abs` X	Absolute value of X
`acos` X	Arc cosine of X
`acosh` X	Hyperbolic arc cosine of X
`acot` X	Arc cotangent of X
`acoth` X	Hyperbolic arc cotangent of X
`asin` X	Arc sine of X
`asinh` X	Hyperbolic arc sine of X
`atan` X	Arc tangent of X
`atanh` X	Hyperbolic arc tangent of X
`ceiling` X	The least integer greater or equal to X, i.e., $\lceil X \rceil$

Table 2.1: Primitive Functions 1/2

`cos` X	Cosine of X
`cosh` X	Hyperbolic cosine of X
`cot` X	Cotangent of X
`coth` X	Hyperbolic cotangent of X
`div` XY	Integer quotient of X and Y
`exp` X	Natural exponent of X, i.e., e^X
`floor` X	The greatest integer less or equal to X, i.e., $\lfloor X \rfloor$
`gcd` XY	Geatest common divisor of X and Y
`ln` X	Neperian logarithm of X
`log` XY	Logarithm of Y in base X
`max` XY	Greater value of X and Y
`min` XY	Lesser value of X and Y
`mod` XY	Integer remainder after dividing X by Y
`round` X	Closest integer to X. If X is exactly half-way between two integers, it is rounded up to the least integer greater than X
`sin` X	Sine of X
`sinh` X	Hyperbolic sine of X
`sqrt` X	Square root of X
`tan` X	Tangent of X
`tanh` X	Hyperbolic tangent of X
`toReal` X	The real number equal to the integer X
`trunc` X	The integer part of X
`uminus` X	$-X$

Table 2.2: Primitive Functions 2/2

```
X == Y = false <== X /= Y

X /= Y = false <== X == Y
X /= Y = true  <== X /= Y
```

Similar defining rules could be also written for the behaviour of >, <, >=, and <= as boolean functions. Actually, the $\mathcal{TOY}$ system runs optimized versions of these definitions.
The system file `basic.toy` includes also definitions for some other *predefined functions*, given by $\mathcal{TOY}$ defining rules of the kind described in Section 2.6. Currently, the predefined functions are:

```
if_then_else :: bool -> A -> A -> A
if_then_else true X Y = X
if_then_else false X Y = Y

if_then :: bool -> A -> A
if_then true X = X

flip :: (A -> B -> C) -> B -> A -> C
flip F X Y = F Y X
```

Instead of writing `if_then_else b e e'` and `if_then b e` in prefix notation, $\mathcal{TOY}$ allows to use the standard syntax `if b then e else e'` and `if b then e`. Nevertheless, the prefix versions are useful for enabling partial applications.
The function `flip` has been predefined for technical reasons, since it is used internally by the $\mathcal{TOY}$ system in order to express operator sections as partial applications. The coexistence of both primitive and predefined function definitions in the system file `basic.toy` also obeys to technical reasons.
Many other (usual and useful, but not strictly needed) functions can be found in the file `misc.toy` (see Appendix B), which reproduces in $\mathcal{TOY}$ syntax a subset of Haskell's prelude.

2.9 Including Programs

In many cases you will want to split a program into smaller pieces, or simply use some frequent functions which have been previously collected into a given file (this is, for instance, the case of `misc.toy`). For this purpose, when writing a program in a file $< File1 >$, you can use the directive `include "`$< File2 >$`"` whose effect, while compiling $< File1 >$, is the same as if the content of $< File2 >$ were literally included into $< File1 >$. For example, you can add (at any point) a line with the text:

```
include "misc.toy"
```

to any program you write, being careful not to redefine functions already defined in `misc.toy`.

2.10 Layout Rule

In the examples above we have used some implicit rules about indentation. Now we are going to define precisely the *layout rule*, which is borrowed from Haskell. The use of the layout rule becomes really useful when dealing with *nested local definitions*. Currently, $\mathcal{TOY}$ does not support nested local definitions; but nevertheless, the parser is prepared to take adventage of the layout rule, whose knowledge is also helpful for understanding some common error messages. The basic unit of $\mathcal{TOY}$ source code is the *section*. A *section* begins with an open bracket '{' followed by a *sentence* and ends with a closed bracket '}'. Each sentence can be either an *include*, a *datatype definition*, a *type synonym definition*, an *operator definition*, a *function type declaration*, or a *function defining rule*. For instance, the following is a valid program text in $\mathcal{TOY}$:

```
    {
      include "misc.toy"
  }
  {
(+.) :: [real] -> [real] -> [real]
  }
{
       X +. Y = zipWith (+) X Y <== length X == length Y
}
```

Different sentences belonging to the same level must be separated by semicolons `';'`. The layout rule allows to omit the symbols '{', '}', and `';'`, replacing them by certain natural layout conventions. Briefly, the layout rule specifies:

- Before reading the first character of each section an open bracket is inserted automatically.
- The end of the section is reached when the only open bracket is the one described in the previous item, and the first character of the current line is found at some column position less or equal than the column position occupied by the section's first character.
- If the first character of the current line is at the same column position as the first character of the innermost level of opened brackets, and this level is strictly greater than one, a semicolon `';'` is inserted before.
- A closed bracket will also be inserted wherever an unknown token is found.
- Empty lines, as well as lines containing only blank spaces, tabulators, and comments, are ignored by the layout rule. That is, they are handled just like blank spaces.

2.11 Predicate Definitions

Predicates in $\mathcal{TOY}$ are viewed as a particular kind of functions, with type τ_1 `->` $\cdots$ `->` τ_n `->` `bool`. As a syntactic facility, $\mathcal{TOY}$ allows to use *clauses* as a shorthand for defining rules whose

right-hand side is `true`. This allows to write Prolog-like predicate definitions, according to the following scheme:

$$\begin{array}{lll} p & :: & \tau_1 \texttt{ -> } \cdots \texttt{ -> } \tau_n \texttt{ -> bool} \\ & & \cdots \\ p\ t_1 \cdots t_n & :- & C_1, \cdots, C_m \texttt{ where } D_1, \cdots, D_p \\ & & \cdots \end{array}$$

Here, each clause abbreviates a defining rule of the form

$$p\ t_1 \cdots t_n = \texttt{true <==}\ C_1, \cdots, C_m \texttt{ where } D_1, \cdots, D_p$$

When using clausal notation, the conditions correspond to the *body* of a clause. A clause body cannot be empty in $\mathcal{TOY}$, but it can be simply `true`. The following is a simple example of Prolog-like programming in $\mathcal{TOY}$:

```
% Type alias for persons:

type person = string

% fatherOf predicate.
% fatherOf(X,Y) holds if Y is father of X.

fatherOf                 :: (person,person) -> bool
fatherOf("peter","john") :- true
fatherOf("paul","john")  :- true
fatherOf("sally","john") :- true
fatherOf("bob","peter")  :- true
fatherOf("lisa","peter") :- true
fatherOf("alan","paul")  :- true
fatherOf("dolly","paul") :- true
fatherOf("jim","tom")    :- true
fatherOf("alice","tom")  :- true

% motherOf predicate.
% motherOf(X,Y) holds if Y is mother of X.

motherOf                  :: (person,person) -> bool
motherOf("peter","mary")  :- true
motherOf("paul","mary")   :- true
motherOf("sally","mary")  :- true
motherOf("bob","molly")   :- true
motherOf("lisa","molly")  :- true
motherOf("alan","rose")   :- true
motherOf("dolly","rose")  :- true
motherOf("jim","sally")   :- true
motherOf("alice","sally") :- true
```

```
% parentOf predicate.
% parentOf(X,Y) holds if Y is parent of X.

parentOf      :: (person,person) -> bool
parentOf(X,Y) :- fatherOf(X,Y)
parentOf(X,Y) :- motherOf(X,Y)

% ancestorOf predicate.
% ancestorOf(X,Y) holds if Y is ancestor of X.

ancestorOf      :: (person,person) -> bool
ancestorOf(X,Y) :- parentOf(X,Y)
ancestorOf(X,Y) :- parentOf(X,Z), ancestorOf(Z,Y)
```

As another syntactic facility, $\mathcal{TOY}$ allows defining rules whose left-hand sides are not linear. Such rules are understood as a shorthand for the result of transforming them as follows: repeated occurrences of a variable X (say) in the left-hand side are replaced by fresh variables X_i, and new conditions $X == X_i$ are added. This facility is very helpful for writing predicate definitions in clausal notation. For instance, a Prolog-like definition of the well-known *permutation sort* algorithm could include the following clauses:

```
permutationSort       :: [real] -> [real] -> bool
permutationSort Xs Ys :- permutation Xs Ys, sorted Ys

permutation           :: [A] -> [A] -> bool
permutation []     [] :- true
permutation [X|Xs] Zs :- permutation Xs Ys,
                         insertion X Ys Zs

insertion                    :: A -> [A] -> [A] -> bool
insertion X []     [X]       :- true
insertion X [Y|Ys] [X,Y|Ys]  :- true
insertion X [Y|Ys] [Y|Zs]    :- insertion X Ys Zs
```

The second defining clause for `insertion` abbreviates the defining rule

```
insertion X [Y|Ys] [X',Y'|Ys'] = true <== X == X', Y == Y', Ys == Ys'
```

The `permutationSort` example also illustrates the use of curried predicates. Uncurried predicates, like those in the `ancestorOf` example, are also allowed. More generally, programmers are free to choose either curried or uncurried style for each particular function or predicate definition. Curried style is often favoured, because partial applications of curried functions can be used to express functional values; see Sections 2.1 and 2.14.

2.12 Non-Deterministic Functions

Prolog-like predicates illustrate a typical feature of logic programming, namely the ability to express *don't-know non-determinism*. For instance, the $\mathcal{TOY}$ system can solve the *goal* `permutation [1,2] Ys`, looking for values of `Ys` that make the goal `true`. The system will compute the two expected answers, namely `Ys == [1,2]` and `Ys == [2,1]`, via a backtracking mechanism. More generally, don't-know non-determinism is useful for solving many search problems in a simple (although not always efficient) way.
Don't-know non-determinism in $\mathcal{TOY}$ can also be achieved by means of *non-deterministic functions*. A function f is called non-deterministic iff some function call $(f\ t_1 \cdots t_n)$ with fixed, already evaluated arguments, can return more than one result. $\mathcal{TOY}$ allows to define non-deterministic functions by means of defining rules with overlapping left-hand sides. For instance, the following $\mathcal{TOY}$ definitions compute ancestors by means of non-deterministic functions:

```
% Functions fatherOf and motherOf

fatherOf         :: person -> person
fatherOf "peter" = "john"
fatherOf "paul"  = "john"
fatherOf "sally" = "john"
fatherOf "bob"   = "peter"
fatherOf "lisa"  = "peter"
fatherOf "alan"  = "paul"
fatherOf "dolly" = "paul"
fatherOf "jim"   = "tom"
fatherOf "alice" = "tom"

motherOf         :: person -> person
motherOf "peter" = "mary"
motherOf "paul"  = "mary"
motherOf "sally" = "mary"
motherOf "bob"   = "molly"
motherOf "lisa"  = "molly"
motherOf "alan"  = "rose"
motherOf "dolly" = "rose"
motherOf "jim"   = "sally"
motherOf "alice" = "sally"

% Non-deterministic functions parentOf and ancestorOf

parentOf    :: person -> person
parentOf X --> fatherOf X
parentOf X --> motherOf X
```

```
ancestorOf      :: person -> person
ancestorOf X   --> parentOf X
ancestorOf X   --> ancestorOf (parentOf X)
```

As in this example, definitions of non-deterministic functions use the sign '`-->`' in place of '`=`'. More precisely, programmers are expected to use '`=`' rather than '`-->`', to indicate that a particular function is *deterministic*. Determinism is an undecidable semantic property. There are sufficient conditions for determinism that can be decided efficiently, see [11]. The current system, however, does not check wether the functions defined by '`=`' rules are indeed deterministic.
A goal such as `ancestorOf "alan" == P` can be solved by the $\mathcal{TOY}$ system. The expected answers will be computed by backtracking. They are:

```
P == "paul";  P == "rose";  P == "john";  P == "mary"
```

Maybe the most paradigmatic example of non-deterministic function is the non-deterministic choice from two given values. This *choice operator* can be programmed in $\mathcal{TOY}$ as follows:

```
infixr 20 //

(//)    :: A -> A -> A
X // Y --> X
X // Y --> Y
```

The choice operator allows a more succint presentation of other non-deterministic functions. For instance, an alternative definition of the `ancestorOf` function can be written as

```
ancestorOf X --> parentOf X // ancestorOf (parentOf X)
```

or also as follows, taking advantage of a ocal definition to avoid repeated computations of parents:

```
ancestorOf X --> Y // ancestorOf Y
             where Y = parentOf X
```

A well known problem solving method involving non-determinism is *generate-and-test*, which works by generating candidates and testing wether they are actual solutions for the problem at hand. Programming generate-and-test in Prolog style leads often to very expensive algorithms, because candidates are totally computed before testing. A typical example of this situation is the Prolog-like permutation sort algorithm shown in Section 2.11 above. $\mathcal{TOY}$ allows to express a *lazy generate-and-test* method, where the generator is a non-deterministic function rather than a predicate. Candidates returned by the generator are passed as arguments to the tester. Since candidates are evaluated lazily, they can be rejected without being totally computed, and a greater efficiency can be achieved. A lazy generate-and-test version of the permutation sort algorithm is shown below:

```
permutationSort     :: [real] -> [real]
permutationSort Xs   = checkIsSorted (permutation Xs)

checkIsSorted   :: [real] -> [real]
checkIsSorted Ys = Ys <== isSorted Ys

isSorted         :: [real] -> bool
isSorted []       = true
isSorted [X]      = true
isSorted [X,Y|Zs] = X <= Y /\ isSorted [Y|Zs]

permutation        :: [A] -> [A]
permutation []     --> []
permutation [X|Xs] --> insert X (permutation Xs)

insert          :: A -> [A] -> [A]
insert X []     --> [X]
insert X [Y|Ys] --> [X,Y|Ys] // [Y|insert X Ys]
```

Here, the non-deterministic function `permutation` acts as generator, while `checkIsSorted` is the tester. When sorting a list `Xs` by means of this program, the variable `Ys` acting as formal parameter in the left-hand side of the tester gets bound to the different permutations of `Xs` returned by the generator. Therefore, it is essential for the correctness of the algorithm that the two occurrences of `Ys` in the right-hand side and in the condition of the defining rule of `checkIsSorted` share the value passed as actual parameter for `Ys`. More generally, the $\mathcal{TOY}$ system supports *sharing* for the values of all variables which occur in the left-hand sides of defining rules and have multiple occurrences in the right-hand side and/or the conditions.
As in lazy functional languages, sharing improves efficiency. Moreover, sharing in $\mathcal{TOY}$ is needed for correctness, as shown by the previous example. Sharing implements so-called *call-time choice* semantics of non-deterministic functions. Intuitively, this semantics means the following: given any function call of the form $f\ e_1 \cdots e_n$, where the actual parameter expressions e_i may involve non-determinism, the evaluation proceeds by *first* computing some possible value v_i for each e_i, and *then* invoking f with actual parameters v_i. Note that the values v_i are not computed eagerly; rather, the evaluation of the actual parameters e_i is delayed until needed, and the eventually obtained values v_i are shared.
A more formal description of call-time choice can be found in Appendix F and in [11]. From a practical viewpoint, $\mathcal{TOY}$ users must be aware of call-time choice, since it affects the set of possible solutions in the case of non-deterministic computations. For instance, given the $\mathcal{TOY}$ definitions:

```
coin :: int
coin --> 0
coin --> 1
```

```
double   :: int -> int
double X  = X + X
```

the goal `double coin == R` has two possible solutions, namely `R == 0` and `R == 2`. These correspond to the two possible call-time choices of a value `v` for `coin` before invoking `double v`. Sometimes, call-time choice gives rise to more subtle behaviours, as illustrated next. Consider the $\mathcal{TOY}$ definitions:

```
coins :: [int]
coins --> [coin|coins]

repeat  :: A -> [A]
repeat X = [X|repeat X]

rcoins :: [int]
rcoins --> repeat coin
```

Due to call-time choice semantics, `coins` computes an infinite list of integers, where each element is chosen non-deterministically as `0` or `1`, independently of the choice for the other elements. There are uncountably many such lists. On the other hand, `rcoins` computes an infinite list consisting of repeated occurrences of the same integer value `v`, which is non-deterministically chosen as `0` or `1`. This behaviour is also consistent with call-time choice semantics. Now there are only two possibilities for the resulting list.
Strict equality conditions and disequality conditions must be properly understood w.r.t. non-determinism. In order to solve $e_1 == e_2$ the $\mathcal{TOY}$ system tries to compute *some* common finite value for e_1 and e_2. On the other hand, $e_1 /= e_2$ is solved by computing *some* possible values of e_1 and e_2 which are in conflict to each other. For instance,

- `coin == coin` can be obviously solved.
- `coin == double coin` can be solved, due to the common value `0` of both sides.
- `coin /= coin` can also be solved, due to the possibility to compute the value `0` for one side and the conflicting value `1` for the other side.

In a similar way, eventual non-determinism must be taken into account when computing with disequality and inequality conditions.
All the examples shown in this section up to now achieve non-determinism by means of overlapping left-hand sides. However, $\mathcal{TOY}$ also supports another way of defining non-deterministic functions, namely to use *extra variables* in the right-hand sides of defining rules. "Extra variables" simply means variables that do not occur in the corresponding left-hand side. $\mathcal{TOY}$ allows extra variables both in the right-hand sides and in the conditions of defining rules. Extra variables behave as *logic variables* in the logic programming sense: they can be introduced in the course of a computation and become bound to different values later on, thus giving rise to non-determinism. For instance, the following non-deterministic function computes the inverse of another function, given as a parameter:

```
inverse     :: (A -> B) -> (B -> A)
inverse F Y --> X <== F X == Y
```

Note that `inverse F` is intended to behave as the inverse of the function given as actual parameter for `F`. However, due to the condition `F X == Y`, it turns out that `inverse F Y` is defined only if `Y` is a *finite* value returned by some computation of `F`. For instance, the goal `inverse length 3 == Xs` has the solution `Xs == [X0,X1,X2]`, standing for all possible lists whose length is `3`. On the other hand, the goal `inverse repeat Xs == X` leads to an attempt to solve `repeat X == Xs` and therefore to non-termination, because the function `repeat` returns an *infinite* list.
Functions whose definition uses extra variables in right-hand sides sometimes return results whose types also depend on extra type variables. As an example, consider the following function:

```
zipWithAny       :: [A] -> [(A,B)]
zipWithAny []     = []
zipWithAny [X|Xs] = [(X,Y)|zipWithAny Xs]
```

Note that `zipWithAny` admits a list `Xs` of any type as parameter. The returned result is a list of pairs `(X,Y)` whose first components `X` are the elements of `Xs` and whose second components `Y` are fresh logic variables, which could become bound later on if the call to `zipWithAny` was part of a bigger computation. From the viewpoint of type discipline, it is consistent to assume the existence of a common type for the values of the new variables `Y`. More precisely, for arbitrary types `A` and `B`, and for any list `Xs :: [A]` given as actual parameter, `zipWithAny Xs` *can* return results of type `[(A,B)]`, provided that the fresh logic variables `Y` introduced by the computation are bound to values of type `B` (or remain unbound and are assumed to have type `B`). Of course, choosing *arbitrary* bindings for the extra variables `Y` could lead to an ill-typed list of pairs. However, the $\mathcal{TOY}$ system never attempts to produce arbitrary variable bindings.
More generally, whenever a function $f \in FS^n$ with principal type f `::` τ_1 `->` $\cdots$ `->` τ_n `->` τ is defined by defining rules with extra variables in the right-hand side, the type can contain extra type variables, and it must be understood in the following way: for any choice of the types to be replaced for the type variables occurring in τ_1 `->` $\cdots$ `->` τ_n `->` τ, and for any actual parameters e_i `::` τ_i $(1 \leq i \leq n)$, the function call $f\ e_1 \cdots e_n$ *can* return results of type τ, provided that the logic variables introduced during the computation as fresh renamings of extra variables in right-hand sides, are bound to values of the expected types. The $\mathcal{TOY}$ system always leaves logic variables unbound until some bindings are demanded by the rest of the current computation. This ensures a well-typed behaviour, except in some special cases to be discussed at the end of Section 2.14.

2.13 Goals and Computed Answers

In the preceding section we have already met some examples of goals and answers. In general, there are two different modes of computing in $\mathcal{TOY}$: solving goals and evaluating expressions. For the first modality, a goal is a set of conditions $C_1, ..., C_n$ where, as in function definitions, each condition C_i must have the form $e_1 \Diamond e_2$, being $\Diamond \in \{==, /=, <, >, <=, >=\}$, and e_1, e_2

expressions. Each condition C_i is interpreted as a constraint to solve, using the definitions of functions and predicates included in the current program as well as the intended meanings of $\{==, /=, <, >, <=, >=\}$, which have been explained in Section 2.6. More precisely, goal solving in $\mathcal{TOY}$ involves a combination of lazy evaluation and computation of bindings for logic variables; see Appendix G for a more formal specification. Whenever a goal is satisfiable, the system answers `yes` and shows the computed answer consisting of a set of constraints in *solved form*.
For example, using the list concatenation function `(++)` defined in Section 2.6, we can solve the goal `[1,2] ++ [3] == [1,2,3]` by typing at the prompt of the system:

```
Toy> [1,2] ++ [3] == [1,2,3]
      yes
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The previous goal is satisfiable with an empty computed answer. After finding a solution for a goal, the system can search for more solutions by backtracking, like in Prolog interpreters. In order to ask for the next solution, the user must respond to the system's question `more solutions [y]?` (default option) by typing 'y' (or simply $< Intro >$). The system will answer `no` if no more solutions can be computed (as in this example), and it will display the next solution otherwise. As an example of a goal with several solution, consider:

```
Toy> Xs ++ Ys == [1,2]
      { Xs -> [],
        Ys -> [ 1, 2 ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { Xs -> [ 1 ],
        Ys -> [ 2 ] }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { Xs -> [ 1, 2 ],
        Ys -> [] }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In general, computed answers can contain three kinds of constraints that will be showed by $\mathcal{TOY}$ in the following order:

- First, the equality constraints in solved form that may be understood as bindings of variables to normal forms. For example,

```
Toy> [1,2] ++ [3] == L
      { L -> [ 1, 2, 3 ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Another example may be:

```
Toy> (++) [1] == F
      { F -> ([ 1 ] ++) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In this case the normal form `((++) [ 1 ])` is a partial application of the function `(++)` and it is shown in infix notation. See also the next example based on the former one, which applies the pattern `F` (the partial application) to a list:

```
Toy> (++) [1] == F, F [2] == L
      { F -> ([ 1 ] ++),
        L -> [ 1, 2 ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

- After equalities, disequalities are shown enclosed between curly brackets. The solved form for a disequality is $X /= t$ where t is a normal form. A computation involving disequalities may be:

```
Toy> [1] ++ Xs /= [1,2]
      { [ 2 ] /= Xs }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

- If the constraint solver over the real numbers is working (see Section 1.5.5), then the computed answer can also contain constraints of this kind that will be showed in the last place. The solved form for them is provided by the solver of Sicstus Prolog. For example, if the constraint solver for real numbers has been activated, one can solve goals as:

```
Toy(R)> X + Y + Z == 3, X - Y + Z == 1
      { Y -> 1 }
      { _C==1.0+X,
        Z==2.0-X,
        _D== -1.0+X }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

or:

```
Toy(R)> X^2 == 4, X>0
      { 4.0-X^2.0==0.0,
        X>0.0 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Non-linear arithmetic constraints, like $X^2 == 4$ in this example, are suspended by the constraint solver until they become linear because of variable instantiation. At the end of the computation the remaining suspended constraints are shown as part of the computed answer, although they might be unsatisfiable.

Another example using the non-deterministic function `permutationSort` from Section 2.12 and the constraints over reals may be the sorting of a list of numbers, one of whose elements is unknown. In the goal, the unknown element is represented as a logic variable `X`:

```
TOY> permutationSort [3,2,X] == L

      yes
      L ==  [ 2, 3, X ]
            { X>=3.0 }

      Elapsed time: 10 ms.


more solutions [y]?

      yes
      L ==  [ 2, X, 3 ]
            { X>=2.0 }
            { X=<3.0 }
```

```
      Elapsed time: 0 ms.


more solutions [y]?

      yes
      L ==  [ X, 2, 3 ]
            { X=<2.0 }

      Elapsed time: 0 ms.
```

Notice that the three computed answers constrain the position and value of `X` in all possible ways.

The second way for computing in $\mathcal{TOY}$ is to reduce or evaluate expressions, as in functional programming languages. In order to compute in this mode, the user must type the symbol `>` followed by the expression e to be evaluated. The system will display the value resulting from the evaluation of e; more precisely, the first possible value, in case that e includes some non-deterministic function. If this value happens to be infinite, the evaluation will continue until the user interrupts it by pressing the Ctrl-C key, or the available memory is exhausted. When computing in this mode, it is expected that e includes no logic variables. Otherwise the system's behaviour is unpredictable, although no error message will occur.
Three simple examples of $\mathcal{TOY}$ computations in evaluation mode are shown below:

```
Toy> > 3*(6-2)
      12
      Elapsed time: 0 ms.
Toy> > [1,2] ++ [3,4]
       [ 1, 2, 3, 4 ]
      Elapsed time: 0 ms.
Toy> > 0 // 1
      0
      Elapsed time: 0 ms.
```

In the third example, the expression is non-deterministic, and the first possible value is computed. In order to compute more possible values a user should work in gaal solving mode, with a goal of the form `0 // 1 == R`.

2.14 Higher-Order Programming

$\mathcal{TOY}$ supports higher-order programming in the style of Haskell and other functional languages, except that λ-abstractions are not allowed. The syntax for function defining rules (see Section

2.6) does not preclude the possibility of functions being passed as arguments or returned as results. In previous sections we have already met some higher-order functions, such as `twice` in Sections 2.1 and 2.2, and `inverse` in Section 2.12. Some other typical examples are shown below.

```
% Composition operator.

infixr 90 .

(.) :: (B -> C) -> (A -> B) -> (A -> C)
(F . G) X = F (G X)

% Combinators for currying and uncurrying.

curry :: ((A,B) -> C) -> A -> B -> C
curry F X Y = F (X,Y)

uncurry :: (A -> B -> C) -> (A,B) -> C
uncurry F (X,Y) = F X Y

% Function iteration.

iterate :: (A -> A) -> A -> [A]
iterate F X = [X|iterate F (F X)]

% Functions map and filter.

map :: (A -> B) -> [A] -> [B]
map F []     = []
map F [X|Xs] = [F X | map F Xs]

filter :: (A -> bool) -> [A] -> [A]
filter P []     = []
filter P [X|Xs] = if P X then [X | filter P Xs]
                  else filter P Xs
```

One familiar application of functional languages consists in programming generic problem-solving schemes as higher-order functions. In $\mathcal{TOY}$, this technique can be naturally combined with non-determinism. As an example, recall the lazy generate-and-test technique used in Section 2.12 in order to opmitize a permutation sort algorithm. In fact, all the applications of the lazy generate-and-test method to different particular problems follow the same scheme, and it is easy to program a single generic solution, by means of a higher-order function `findSol`. This function expects three parameters: a generator `Generate`, a tester `Test`, and an input `Input`, and it returns a solution `Solution`. The generator is applied to the input in order to produce

candidate solutions, and any candidate which passes the tester can be returned as solution. As we already know, thanks to lazy evaluation, partially computed candidates can be rejected by the tester. The $\mathcal{TOY}$ definitions realizing this algorithm are as follows.

```
findSol :: (Input -> Solution) -> (Solution -> bool) -> Input -> Solution
findSol Generate Test Input --> check Test (Generate Input)

check :: (Solution -> bool) -> Solution -> Solution
check Test Solution --> Solution <== Test Solution
```

An alternative definition of the function `permutationSort` as a particular instance of the generic function `findSol` is now possible:

```
permutationSort :: [real] -> [real]
permutationSort  = findSol permutation isSorted
```

where `permutation` and `isSorted` are defined as in Section 2.12.
The $\mathcal{TOY}$ language also supports *higher-order predicates*. These are just higher-order functions which return a boolean result. $\mathcal{TOY}$ users can program higher-order predicates using the clausal notation explained in Section 2.11. For instance, the following higher-order predicate is a relational version of the `map` function:

```
mapRel :: (A -> B -> bool) -> [A] -> [B] -> bool
mapRel R []     []     :- true.
mapRel R (X:Xs) (Y:Ys) :- R X Y, mapRel Xs Ys
```

In relational logic programming languages such as λ-Prolog [28], user-defined functions are not available. Therefore, `mapRel` must be used instead of `map`. In a multiparadigm language there is more room for choice. Often, functions behave better than predicates if there is some need to compute with potentially infinite structures, where lazy evaluation is helpful. For instance, `map` is a better choice than `mapRel` in case that a potentially infinite list must be processed. More precisely, the attempt to solve a `mapRel` goal involving an infinite list will not terminate, while expressions of the form `take n (map fun list)` can be lazily evaluated, even if the value of `list` turns out to be infinite. Recall the definition of function `take` from Section 2.6.
A special feature of higher-order programming in $\mathcal{TOY}$ is the availability of *higher-order patterns*. This notion has already been introduced in Section 2.1, as a syntactic device to represent functional values. Higher-order patterns behave as intensional representations of functions. They can be compared for strict equality `(==)` and disequality `(/=)`, in the same way as first-order data terms.
In actual programs, higher-order patterns can play a useful rôle both as patterns occurring in the left-hand sides of defining rules, and as computed results. In order to illustrate this idea, let us consider family relationships, modelled as non-deterministic functions of type `person -> person`, as we have seen in Section 2.12. It is natural to think of complex family relationships being built as the functional composition of basic relationships, such as `fatherOf` and `motherOf`.

Using the composition operator `(.)` defined at the beginning of this section, *higher-order predicates* which recognize both *basic* and *composite* family relationships are easy to define. The auxiliary type alias `rank` is used to compute the number of compositions involved in a composite relationship.

```
basicFamilyRelation :: (person -> person) -> bool
basicFamilyRelation fatherOf   :- true
basicFamilyRelation motherOf   :- true
basicFamilyRelation sonOf      :- true
basicFamilyRelation daughterOf :- true
basicFamilyRelation brotherOf  :- true
basicFamilyRelation sisterOf   :- true

type rank = nat

someRank  :: rank
someRank --> zero // suc someRank

familyRelation :: rank -> (person -> person) -> bool
familyRelation z      F          :- basicFamilyRelation F
familyRelation (s N)  ((.) F G)  :- basicFamilyRelation F,
                                    familyRelation N G
```

These definitions, along with suitable definitions for the behaviour of the basic family relations, could be used to solve *goals* such as

```
R == someRank, familyRelation R F, F "alice" == "allan"
```

where the non-deterministic nullary function `someRank` is used to search for ranks. The goal has the following solution (among others):

```
R == suc (suc zero), F == sonOf . brotherOf . motherOf
```

Note that the answer computed for `F` can be equivalently written as

```
(.) sonOf ((.) brotherOf motherOf)
```

which is a higher-order pattern, according to the syntactic characterization given in Section 2.1. Unfortunately, some $\mathcal{TOY}$ computations involving higher-order patterns can break the type discipline, as we explain in the rest of this section. A first kind of problem arises when solving goals which include expressions of the form $F\ e_1 \cdots e_n$ $(n > 0)$, where a logic variable F is acting as a function. Let us speak of *goals with higher-order logic variables* to describe this situation. Some binding for the higher-order logic variable F must be computed in order to solve such goals. This can happen in two ways:

1. *Generated bindings:* some previous goal-solving step binds variable F before coming to the computation of $F\ e_1 \cdots e_n$. This was the case in the family relationships example above.

2. *Guessed bindings:* no previous goal-solving step has bound F. Then, in order to continue the computation of $F\ e_1 \cdots e_n$, $\mathcal{TOY}$ guesses some pattern of the form $h\ X_1 \cdots X_m$ (with fresh logic variables X_i) as binding for F. In fact, the system uses backtracking to try all possible bindings of this form. In each case, the computation continues and may lead to different solutions of the goal.

Generated bindings of higher-order logic variables are unproblematic. Guessed bindings, on the other hand, can give rise to ill-typed solutions. For instance, solving the well-typed goal

```
map F [true, X] == [Y, false]
```

requires to guess a binding for the higher-order logic variable `F`. One possibility is `((++) [])`, which leads to an ill-typed solution, because the principal type of `((++) [])` is incompatible with the type that must be assumed for `F` in order to well-type the goal. In general, guessed bindings of higher-order variables can produce ill-typed solutions because the current $\mathcal{TOY}$ system does not check type consistency of such bindings at run time. As an additional disadvantage, guessing bindings for higher-order variables generally leads to the exploration of a huge search space, and often even to non-termination.
A second kind of problem can arise when solving goals of the form $f\ e_1 \cdots e_m == f\ e'_1 \cdots e'_m$, where f is a defined function of arity $n > m$. Such a goal must be solved by decomposing it into m new subgoals $e_1 == e'_1, \cdots, e_m == e'_m$. Assume that the principal type of f is τ_1 `->` $\cdots$ `->` τ_n `->` τ, such that some type variable occurring in τ_i for some $1 \leq i \leq m$ does not occur in τ_{m+1} `->` $\cdots$ `->` τ_n `->` τ. If this is the case, some of the m new subgoals $e_i == e'_i$ may be ill-typed. This problem is called *opaque decomposition*; the name points to the fact that the type of $f\ e_1 \cdots e_m$ does not uniquely determine the types of the argument expressions e_i.
For example, using a function

```
snd     :: A -> B -> B
snd X Y  = Y
```

as well as other functions presented in previous examples, one can build the well-typed goal

```
snd (head [Xs,[true]]) == snd (head [Ys,[15]])
```

where both sides of the strict equality have type `B -> B` under the type assumptions `Xs :: [bool]`, `Ys :: [ int]`. In order to solve this goal, $\mathcal{TOY}$ starts with an opaque decomposition step leading to the new goal

```
head [Xs,[true]] == head [Ys,[1]]
```

which is ill-typed. In spite of this, $\mathcal{TOY}$ can succesfully complete a computation which returns the ill-typed solution `Xs == Ys`.

Opaque decomposition steps can also arise when solving disequalities $f\ e_1 \cdots e_m \,/=\, f\ e'_1 \cdots e'_m$, where f is a defined function of arity $n > m$. This is because any solution of $e_i \,/=\, e'_i$ (for any $1 \leq i \leq m$) is a solution of the disequality. As we have seen, those solutions whose computation involves opaque decomposition can be ill-typed. The current $\mathcal{TOY}$ system gives no warning in such cases.

2.15 Finite Failure

$\mathcal{TOY}$ provides the primitive boolean function *fails* as a direct counterpart to finite failure in Prolog. A call of the form *fails e* returns *true* if there is not any possible reduction for *e* and *false* otherwise[3]. For example, using the function *head* introduced in page 27 we have that *fails (head [])* reduces to *true* because the function *head* is not defined for the empty list, while *fails (head [1,2])* reduces to *false*.
The semantics of failure in functional logic programming has been widely studied in [23, 20, 24, 21, 25, 19]. For the moment $\mathcal{TOY}$ incorporates a restricted version of the *constructive failure* investigated in such works[4]. The restriction is similar to that of negation as finite failure in Prolog: in order to ensure the correctness of the answers, the expression *e* in a call of the form *fails e* must not contain variables. Despite of this restriction in the use of this function, it provides an interesting resource for programming in a wide range of applications.
As we have seen (Section 2.11) $\mathcal{TOY}$ allows predicate definition in a Prolog style. Using the function *fails*, we can even write predicates with negation replacing the predicate *not* by the function *fails*. For example, assume the following Prolog program:

$$\begin{array}{ll} member(X,[X|Xs]). & \\ member(X,[Y|Ys]) :- & member(X,Ys). \\ insert(X,Xs,[X|Xs]) :- & not(member(X,Xs)). \\ insert(X,Xs,Xs) :- & member(X,Xs). \end{array}$$

The translation into a $\mathcal{TOY}$ program is straightforward (file `member.toy` in the `examples` directory of the distribution):

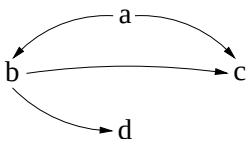
```
member X [X|Xs] :- true
member X [Y|Ys] :- member X Ys

insert X Xs [X|Xs] :- fails (member X Xs)
insert X Xs Xs     :- member X Xs
```

But failure has a variety of interesting uses in the context of functional logic programming itself. It is useful in general search problems. For example assume the following graph:

[3]The formal semantics, according to lazy evaluation is that *fails e* returns *true* if *e* can not be reduced to *head normal form* (variable or constructor rooted term) and *false* otherwise.

[4]There prototype *OOPS* [33] available at `http://babel.dacya.ucm.es/jaime/systems.html` incorporates *constructive failure* in a functional logic language.



We can represent it in *FLP* and define a function *path* for checking if there is a path between two nodes (file `graph.toy` in the `examples` directory of the distribution):

```
data node = a | b | c | d

next a = b
next a = c
next b = c
next b = d

path X X = true
path X Y = path (next X) Y <== X /= Y
```

Now, assume we want to define the predicate *safe* for nodes (as a boolean function):

safe X::= X is not connected with d *(there is no path from* X *to* d*)*

In order to define this function we would have to use a *deterministic* functional style, changing the representation of graphs and rewriting the full program. But using failure, the definition is direct:

```
safe X = fails (path X d)
```

According to this definition, *safe c* reduces to *true*, while for the rest of nodes it reduces to *false*.

Failure may also be used for programming two-person finite games in an easy and elegant way (following the idea of [10] in logic programming). Assume such a game in which players make a move in turn, until it is not possible to continue. At this stage, the player that cannot move losses, so the winner is the one that makes the last movement. We assume that it is defined the function *move State* that produces (in a non-deterministic way) a new state from the given *State* using a legal movement.
We are interested in a function *winMove State* that provides a *winning movement* from *State*, that is, a movement that eventually bring us to the victory, with independence of the movements of the adversary. This is easy to express using failure (file `nim.toy` in the `examples` directory of the distribution):

```
winMove State = State' <== State' == move State, fails (winMove State')
```

This simple function provides a way for winning the game whenever it is possible. The idea is to find a legal movement *State'*, such that the other player *fails* to find a winning movement form it. This scheme can be applied for a variety of games. As an example we introduce the Nim

game: we have a set of rows with sticks and a movement consists in dropping one or more sticks from an unique row. The state of the game can be represented by a list of integers, where each one represents the number of sticks of a row. A legal movement is defined in a non-deterministic way with the function *move* as follows:

```
pick N = N-1 <== N>0
pick N = pick (N-1) <== N>0

move [N|Ns] = [pick N|Ns]
move [N|Ns] = [N|move Ns]
```

Here the function *pick* takes one or more sticks from a row and *move* picks some sticks from one row. Now we can evaluate

```
Toy> winMove [1,2,2] == S
      { S -> [ 0, 2, 2 ] }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

That is, we must drop the stick from the first row. It is easy to check that for any possible movement of the adversary we can assure the victory. The reader may try to program the tic-tac-toe game by providing a representation for the panel and the function *move*.

Apart from *fails* $\mathcal{TOY}$ provides some other related primitives. In a non-deterministic language it is interesting in some cases to have a set of solutions more than a single solution for a given problem. The usual way for providing these solutions is to get one by one by backtracking, like in Prolog or also in $\mathcal{TOY}$. But there are situations for which it is useful to obtain the full set of solutions together in an structure, like a list (Prolog provides several predicates for this issue like *findall*).
Using failure we can obtain all the results of the evaluation of any expression, but in fact $\mathcal{TOY}$ facilitates the work by providing a number of primitives for such issues. The first one is *collect* e that returns the list of possible values for the expression e. For example, for the program of graphs introduced above we can evaluate:

```
Toy> collect (next a) == L
      { L -> [ b, c ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

This is the list of the nodes adjacent to the node a, as expected. For the Nim's game we can evaluate:

```
Toy> collect (winMove [2,2,2]) == L
      { L -> [ [ 0, 2, 2 ], [ 2, 0, 2 ], [ 2, 2, 0 ] ] }
      Elapsed time: 78 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy> collect (winMove [2,2]) == L
      { L -> [] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The first goal provides the list of states (lists) that represent the possible winning moves from the state *[2,2,2]*, while the second one shows that there is no possible winning movement from *[2,2]* (this means that there is not any possible movement that *guaranties* our victory; of course the other player can make a wrong movement like *[0,2]* in such a way that we will win).
Another primitive is *collectN M e* that returns a list with (the first) *M* results for the expression *e*. For example we can obtain the first two solutions for the goal *winMove [2,2,2]* solving the goal:

```
Toy> collectN 2 (winMove [2,2,2]) == L
      { L -> [ [ 0, 2, 2 ], [ 2, 0, 2 ] ] }
      Elapsed time: 94 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

There is another primitive *once e* that returns the first reduction for the expression *e*. For example:

```
Toy> once (winMove [2,2,2]) == L
      { L -> [ 0, 2, 2 ] }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

For more complex uses of these primitives see Appendix A.4.

2.16 Constraints

$\mathcal{TOY}$ can be described as a narrowing-based functional logic language, which means that the the basic operational mechanism is narrowing, a combination of unification and rewriting. But narrowing is not the only computational process, as we see now.

As we have explained in Section 2.6, functions in $\mathcal{TOY}$ programs are defined functions, which are defined by means of conditional rules with local definitions, of the form

$$ft_1 \ldots t_n = r \Leftarrow C_1, \ldots, C_m \texttt{ where } D_1, \ldots, D_p$$

Each condition C_i is a *constraint* of the form $e \diamond e'$, where $\diamond$ can be `==` , `/=` ,< , <= ,> or >= [5]. Well-typedness of constraints implies that e and e' should have the same type for the case of `==` and `/=`, and the same numeric type (`int` or `float`) for the case of < , <= ,> or `>=`. The reading of the program rule above is that $ft_1 \ldots t_n$ can be reduced to r if the constraints $C_1, \ldots, C_m$ are satisfied, using the local definitions to obtain the values of locally defined variables. Thus, the comma ',' in the condition part can be read as conjunction. Conjunctions of constraints are also called constraints.
At the top level, a $\mathcal{TOY}$ computation consist in solving a goal [6], where goals are also constraints. To check the satisfiability of constraints requires a process of *constraint solving* which is somehow independent of narrowing, although cooperates with narrowing when constraints involve subexpressions $f(e_1, \ldots, e_n)$. Constraint solving may produce one of the following results:

- A *failure*, if the constraint was unsatisfiable. As for the case of failure during narrowing, failure of constraint solving is silent and simply forces backtracking. Only in case of failure of the whole computation the user is noticed of that.

- A constraint in *solved form*, i.e., a constraint in some kind of simplified format ensuring its satisfiability. As a matter of fact, a single constraint can be transformed into a family a solved forms; the original constraint would be equivalent to the disjunction of such family. Disjuncts correspond to independent branches of the computation, to be tried in sequence in case of bactracking. This happens, for instance, for the goal `[X,0] /= [1,Y]` for which a first solution `X /= 1` is obtained. If, after being asked, the user wants to see more solutions, a second one, `Y/=0` is shown.

- A *suspended* constraint, for which the constraint solver is not powerful enough to ensure the satisfiability. This happens in $\mathcal{TOY}$ with non linear arithmetic constraints. Suspended constraints can be re-activated if some variables are instantiated in a later step of the computation.

During computations, solved and suspended constraints act as a *constraint store*. As the computation proceed, new constraints can be added to the store, which must be checked again for satisfiability. When the computation finishes, the store is a part of the answer (it *is* the answer, it we see the substitution part of the answer as a solved form for equality constraints).
Currently, $\mathcal{TOY}$ is able to manage three kinds of constraints:

- Syntactical constraints (equality and disequality) over constructor terms. These are embedded in the system, and the constraint solver is implemented from the scratch.

[5] More generally, C_i can be any boolean expression e, but in this case the condition e `== true` is assumed by the system.

[6] At least when using the *goal solving mode* of evaluation of $\mathcal{TOY}$.

- Arithmetical constraints over real numbers. Their use is optional, and the implementation mostly relies on the constraint solver for linear arithmetic constraints that comes with Sicstus Prolog.
- Constraints over finite domains. As well, their use is optional, and the implementation mostly relies on the Sicstus Prolog finite domain constraint solver.

In this section, we introduce the first two items. The constraints over finite domains are explained in detail in chapter 3.

2.16.1 Syntactical Equality and Disequality Constraints

Equality tests are frequently used in daily programming. As an example, consider the following simple function definition, likely to appear in many real functional or functional logic programs.

```
member X [ ]       = false
member X [Y | Ys]  = if (X `eq' Y) then true else member X Ys
```

The symbol `eq` expresses here some suitable notion for equality. In a language like $\mathcal{TOY}$ with lazy evaluation, the sensible notion for `eq` is that of *strict equality*, which means that *e* `eq` *e'* reduces to *true* if both e and e' are reducible to the same pattern, and to `false` if e and e' can be reduced to some extent as to detect inconsistency, i.e., different constructor symbols at the same position in e and e'.
Using the program above as a functional program, the expression `member zero [(s loop), zero]` could be reduced to the value `true` ; this has involved the evaluation of `zero eq (s loop)` (yielding the value `false`, in spite of the fact that `loop` is a diverging function), and `zero eq zero` (yielding the value `true`).
Now, a functional logic language like $\mathcal{TOY}$ is also able to perform reductions over expressions containing variables, which may become instantiated in the process. This *reversibility* property is one of the nicest that functional logic programming shares with logic programming. But reversibility is not easy to achieve in programs requiring equality tests, like the example above. Consider for instance an expression containing variables, such as `member X [Y]`. One possible reduction could yield the value `true` together with the substitution `X` $\mapsto$ `Y`, which can be also seen as a constraint `X==Y`. But `member X [Y]` can also be reduced to `false` if `X` and `Y` are given different values. An attempt of covering such values by means of substitutions results in infinitely many answers [7]. For instance, if the program contains the datatype declaration

```
nat = zero | suc nat
```

then the possible answers would include the following:

```
X==zero,Y==suc U; X==suc U,Y==zero; X==suc zero,Y==suc (suc U);  X==suc(suc U),Y==suc
                                    zero; ...
```

[7] At least if the set of terms corresponding to the given signature is infinite.

This family of answers cannot be replaced by any equivalent finite set of substitutions. The situation changes drastically if we consider disequality constraints, since *one single* disequality (namely `X /= Y`) collects all the information embodied in all those substitutions. $\mathcal{TOY}$ exhibits such behavior because of its ability of explicitly handle equality (`==`) and disequality `/=`) constraints, corresponding respectively to the positive and negative cases of strict equality, as described above.
Using these constraints, the function `eq` can be defined by the rules:

```
X eq Y = true   <== X == Y
X eq Y = false  <== X /= Y
```

For the sake of clarity, we have used throughout this discussion the symbol `eq`. However, $\mathcal{TOY}$ overloads the symbols `==` and also `/=`, so that they can be used both for constraints and for functions.

Strict Equality Constraints

To understand how constraint solving proceeds, it is useful to explain two different things:

- When a constraint is *satisfied* (or *valid*), that is, it is true for all values of its variables.
- When a constraint is *satisfiable*, that is, there are values of its variables making it true.

The first notion fixes the 'semantics' of the constraint, while the second is more related to constraint solving, which tries to check satisfiability by reducing constraints to solved form.
For the case of equality constraints, we can say that

- e `==` e' is satisfied if e and e' can be reduced to the same pattern.
- A constraint e `==` e' is satisfiable if e and e' can be reduced to unifiable patterns. Solved forms are the results of such unifications, that it, idempotent substitutions which can be seen (and showed) as sets of solved equations.

According to the semantics, a constraint e `==` e' could be solved by reducing e and e' to normal forms (patterns) and then unifying them. But in practice, to interleave the reduction of e and e' is better from the point of view of detecting failure. For instance, the constraint `suc` e `== zero` is not satisfiable, whatever the expression e is. To reduce e (which can be costly or even diverging) is needed to reach a normal form for `suc` e, but not to detect the failure of the constraint. $\mathcal{TOY}$ solves equality constraints is such interleaved way, which also includes an optimized occur-check (see Appendix G for details).

Disequality Constraints

A constraint e `/=` e', is satisfied if e and e' can be reduced to an extent as to have a constructor clash, which means having different constructor symbols at the same 'external position' (that is, positions not insidea function application). A disequality e `/=` e' can be satisfied even if e or e' do not have a normal form.

Examples With the following definitions

```
loop = loop                          % divergence
undefined = undefined <== false      % failed reduction
from N = [N | from (suc N)]          % infinite list
```

the following ground goals have in each case the indicated behaviour:

```
[0,0] /= [1,0]                       % succeeds
from zero /= from (suc zero)         % succeeds
[0,undefined] /= [1,undefined]       % succeeds
[undefined,0] /= [undefined,1]       % succeeds
[0,loop] /= [1,loop]                 % succeeds
[loop,0] /= [loop,1]                 % succeeds
[0,0] /= [0,0]                       % fails
[0,undefined] /= [0,undefined]       % fails
[0,loop] /= [0,loop]                 % diverges
from zero /= from zero               % diverges
```

In presence of variables, the constraint solving mechanism to check satisfiability of disequality constraints tries to reduce them to solved forms of the shape `X /=` t_1`,...,X /=` t_n`,Y /=` s_1`,...,Y /=` s_m`,...`, which are known to be satisfiable, if the set of ground terms is infinite [8]. During a computation, the accumulation of these solved forms constitutes a *disequality constraint store* which plays a dynamic role: if at a given step of the computation a variable `X` is going to be bound to a term `t`, it must be checked that the binding is compatible with all the stored disequalities for `X`. This *constraint propagation* amounts to solve `t /=` $\mathtt{s}_1$`,...,t /=`$\mathtt{s}_n$, if `X /=` $\mathtt{s}_1$`,...,X /=`$\mathtt{s}_n$ was the store for `X`.
As an example, we show below some stepts for the solving of a goal involving equality and disequality constraints, and how the accumulated substitution and constraint store evolve . The symbols `a,b,c,d` are constructor symbols and $*_i$ indicates a don't know choice.

[8] Otherwise, satisfiability of solved forms is not guaranteed. For instance, `X /= Y, X /= Z, Y /= Z` is not satisfiable if there are only two ground terms. Since $\mathcal{TOY}$ is a typed system, this might happen for some types. In particular, this *does* happen for the type *bool*, which has only the constants `true` and `false` . Since the type *bool* is frequently used, there is a special treatment of disequality in this case, as to avoid unsatisfiable answers.

Goal	Substitution	*Constraint store*
`c X /= c (c Y), d X Y /= d a a, d X X == d (c Z) (c b)`		
`X /= c Y, d X Y /= d a a, d X X == d (c Z) (c b)`		
`d X Y /= d a a, d X X == d (c Z) (c b)` $(*_1)$		$X /= c\ Y$
`X /= a , d X X == d (c Z) (c b)`		$X /= c\ Y$
`d X X == d (c Z) (c b)`		$X /= c\ Y,\ X /= a$
`X == c Z, X == c b`		$X /= c\ Y,\ X /= a$
`c Z /= c Y, c Z /= a, c Z == c b`	$X \mapsto c\ Z$	
`Z /= Y, c Z /= a, c Z == c b`	$X \mapsto c\ Z$	
`c Z /= a, c Z == c b`	$X \mapsto c\ Z$	$Z /= Y$
`c Z == c b`	$X \mapsto c\ Z$	$Z /= Y$
`Z == b`	$X \mapsto c\ Z$	$Z /= Y$
`b /= Y`	$X \mapsto c\ b,\ Z \mapsto b$	
	$X \mapsto c\ b,\ Z \mapsto b$	$Y /= b$

The computed answer consists of the final accumulated substitution and constraint store [9]. An alternative branch for the the computation at $(*_1)$ is:

Goal	Substitution	*Constraint store*
`d X Y /= d a a, d X X == d (c Z) (c b)` $(*_1)$		$X /= c\ Y$
`d X X == d (c Z) (c b)`		$X /= c\ Y$
`d X X == d (c Z) (c b)`		$X /= c\ Y,\ Y /= a$
`X == c Z, X == c b`		$X /= c\ Y,\ Y /= a$
`c Z /= c Y, c Z == c b`	$X \mapsto c\ Z$	$Y /= a$
`Z /= Y, c Z == c b`	$X \mapsto c\ Z$	$Y /= a$
`c Z == c b`	$X \mapsto c\ Z$	$Y /= a,\ Z /= Y$
`Z == b`	$X \mapsto c\ Z$	$Y /= a,\ Z /= Y$
`b /= Y`	$X \mapsto c\ b,\ Z \mapsto b$	$Y /= a$
	$X \mapsto c\ b,\ Z \mapsto b$	$Y /= a,\ Y /= b$

Strict Equality and Disequality Functions

As we have discussed before, for practical purposes one needs not only the possibility of using the constraints `==` and `/=` in conditions of rules, but also a two-valued equality function, for which the system overloades the symbol `==`. We also mentioned that `==` can be defined *in* the language by means of the rules

```
X == Y  = true  <== X == Y
X == Y  = false <== X /= Y
```

Dually, there is also a two-valued function `/=` which could be defined as:

```
X /= Y  = true  <== X /= Y
X /= Y  = false <== X == Y
```

[9] The system cleans up the final store to eliminate irrelevant disquequalities not involving, directly or indirectly, the goal variables.

These definitions of `==, /=` are semantically correct but not very efficient. As an example, consider the evaluation of `[0,1,...,99] == [0,1,...,99,100]`, where `==` is the equality function (this always can be known by the context). With the code above, the system tries first the first rule, attempting then to solve the constraint `[0,1,...,99] == [0,1,...,99,100]`. After 100 decompositions, we reach the constraint `[] == [100]`, which fails. Then the second rule is tried, which requires to redo all the decompositions to reach finally the constraint `[] /= [100]`, which succeeds.
Things are not done in this way in the system. Instead, the evaluation of an equality proceeds as long as possible without branching between the positive and the negative case. In the example above, the decompositions are done until the evaluation of `[] == [100]` is reached. Since `[] == [100]` evaluates to `false`, this is the value of the initial equality.

Inequality Functions

Similarly to the case of `==` and `/=`, each of the constraint symbols `<, <=, >, >=` is overloaded to represent also a boolean function, according to the following definitions:

```
X < Y  = true  <== X < Y
X < Y  = false <== X >= Y
```

and similarly for the rest.

2.16.2 Arithmetical Constraints over Real Numbers

As explained in Section 1.5.5, $\mathcal{TOY}$ includes the possibility of activating (deactivating resp.) a linear constraint solver by means of the command `/cflpr.` (`/nocflpr.` resp. [10]).
The interest of arithmetical constraints is known in the logic programming field since the appearance of *CLP($\mathcal{R}$)*, the first constraint logic programming language. In contrast to the very *ad hoc* Prolog treatment of arithmetic, constraints give a clear logical status to real numbers. From the point of view of programming, constraints provide many benefits like more reversibility, reusability and independence of control.
Something similar happens in the functional logic programming setting. Consider for instance the following boolean function to determine if a given point in the plane - represented by its two coordinates - lies in a given straight line - represented by the three coefficients of its equation - :

```
type point = (real,real)
type line =  (real,real,real)

inLine:: point -> line -> bool
inLine (X,Y) (A,B,C) =  A*X+B*Y+C==0
```

[10] We nevertheless discourage the use of `nocflpr`, at least if efficiency is required, since this command inhibits the use of constraint solver within $\mathcal{TOY}$, but is not able to deactivate it for the Sicstus session running underground. This implies a strong penalty in efficiency.

Even without constraints, this works fine as a functional program. We can evaluate ground expressions like `inLine (1,2) (1,-1,1)` or `inLine (1,0) (1,-1,1)` to obtain `true` and `false` respectively. But an attempt of solving a goal with variables results in a runtime error

```
Toy> inLine (2,Y) (1,-1,1)

RUNTIME ERROR: Variables are not allowed in arithmetical operations.
      no
      Elapsed time: 0 ms.
```

instead of obtaining the 'logical' answer `Y == 3`. This makes programs less reversible, abstract and reusable. For instance, to calculate the intersection point of two lines, we cannot use the following definition, despite its logical soundness:

```
intersctPoint L L' = P <== inLine P L, inLine P L'
```

Notice that this definition is quite 'declarative' - uses the logic of the problem - and abstract - it does not depend of the concrete represention of lines and points - but unfortunately it results again in runtime errors.

```
Toy> intersctPoint (1,-1,1) (1,1,0) == P

RUNTIME ERROR: Variables are not allowed in arithmetical operations.
      no
      Elapsed time: 0 ms.
```

To overcome the problem, without the use of constraints, we must do some 'homework' to calculate explicitly the coordinates of the intersection:

```
intersctPoint' (A,B,C) (A',B',C') = ((B*C'-B'*C)/(A*B'-A'*B),
                                     (-C -A*X)/B)
```

Now

```
TOY> intersctPoint' (1,-1,1) (1,1,0) == P
     yes
     P == (-0.5, 0.5)
```

With the use of constraints, arithmetic becomes reversible:

```
Toy> /cflpr
Toy(R)> /run(regions)
Toy(R)> inLine (2,Y) (1,-1,1)
      { Y -> 3 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
```

```
      no
      Elapsed time: 0 ms.
Toy(R)> intersctPoint (1,-1,1) (1,1,0) == P
      { P -> (-0.5, 0.5) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Furthermore, we can solve goals with several solutions, obtaining a constraint as an answer:

```
Toy(R)> inLine P (1,-1,1)
      { P -> (_A, _B) }
      { _B==1.0+_A }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Linear constraints are active constraints, meaning that their satisfiability is effectively checked. For instance, an attempt of finding the intersection of two concrete parallel lines fails:

```
Toy(R)> intersctPoint (1,1,1) (1,1,0) == P
      no
      Elapsed time: 0 ms.
```

However, nonlinear constraints are passive, and their satisfiability is not checked. For instance, an attempt of finding the intersection of two generic parallel lines does not fail, but returns a (hard to read) non linear constraint as answer, which is in fact unsatisfiable:

```
Toy(R)> intersctPoint (A,B,1) (A,B,0) == P
      { P -> (_C, _D) }
      { _E== -1.0-_F,
        _G== -(_H),
        -(_C*A)+_F==0.0,
        -(_D*B)+_E==0.0,
        _H-_C*A==0.0,
        _G-_D*B==0.0 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The previous examples use only `==` as arithmetical constraint. The system supports also `/=,<,<=,>,>=`. The use of some of them is illustrated in Section A.6.1.

-	*	**	/
^	+	<	<=
>	>=	==	/=
abs	acos	acosh	acot
acoth	asin	asinh	atan
atanh	cos	cosh	cot
coth	exp	ln	log
max	min	sin	sinh
sqrt	tan	tanh	uminus

Table 2.3: Real Constraint Operators and Functions

ceiling	div	floor	gcd
mod	round	toReal	trunc

Table 2.4: Real Constraint Operators and Functions unmanaged by the Solver

Table 2.3 summarizes the real constraint operators which are sent to the real solver. Other arithmetical functions cannot be managed by this solver, as shown in Table 2.4. Recall that nonlinear constraints are delayed until they become linear due to variable instatiation.

Optimization Functions

Optimization problems have been acknowledged as an important field of operations research, and have been identified in real-life applications as planning, scheduling, and timetabling, to name a few. An optimization problem is understood in the context of a set of constraints and a cost function which needs to be maximized or minimized. A cost function may represent profit, resources, or time, for instance. $\mathcal{TOY}$ provides several optimization functions for real constraints:

- `minimize :: real -> real`
  The application `minimize X` returns the minimum for `X` (which represents the cost function) in the context of the constraints in the real constraint store. `X` is demanded to be in head normal form. The evaluation of this application raises an exception if either there are suspended non-linear constraints or unboundedness is detected.

- `maximize :: real -> real`
  The application `maximize X` is equivalent to the application `minimize -X`.

  The following is an example of using this function:

```
Toy(R)> 2*X+Y <= 16, X+2*Y<=11, X+3*Y <= 15,
        Z==30*X+50*Y, maximize Z == Z
      { X -> 7.00000000000001,
```

```
        Y -> 1.9999999999999956,
        Z -> 310.00000000000006 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In this example, if minimization is used instead of maximization, an exception is raised because of unboundedness.

Another example shows an optimization problem with infinite values for the variables involved in the cost function:

```
Toy(R)> X-Y==Z, X>=0, X<=2, Y>=0, Y<=2, X==Y, minimize Z == I
      { Y -> X,
        Z -> 0,
        I -> 0 }
      { X=<2.0,
        X>=-0.0 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

- `bb_minimize :: real -> [real] -> real`
  The application `bb_minimize X Is` returns the minimum for `X` in the context of the constraints in the real constraint store, assuming that the variables in the list `Is` can only take integral values. This allows the evaluation of mixed integer linear problems. `X` is demanded to be in head normal form, and `Is` in normal form. The evaluation of this application raises an exception if unboundedness is guessed (because either the problem is actually unbounded or boundness cannot be detected because of suspended non-linear constraints). A value is considered integral with an error of 0.001. Strict bounds on the decision variables are honored, but strict inequalities and disequalities are not handled. For example:

```
Toy(R)> X>=Y+Z, Y>1, Z>1, bb_minimize X [Y,Z] == X
      { X -> 4,
        Y -> 2,
        Z -> 2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Note that this function only generates one particular solution, in contrast to both `minimize` and `maximize`, which deliver a general solution. Consider, for instance, the same problem presented to illustrate the function `minimize`, but adding integral constraints for `X` and `Y`:

```
Toy(R)> X-Y==Z, X>=0, X<=2, Y>=0, Y<=2, X==Y, bb_minimize Z [X,Y] == I
      { X -> 0,
        Y -> 0,
        Z -> 0,
        I -> 0 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

That is, the answers { `X -> 1, Y -> 1, Z -> 0, I -> 0`} and { `X -> 2, Y -> 2, Z -> 0, I -> 0`} are missing.

- `bb_maximize :: real -> [real] -> real`
  The application `bb_maximize X Is` is equivalent to `bb_minimize -X Is`.

Chapter 3

Constraints over Finite Domains

In this chapter, we present the implementation of Constraint Functional Logic Programming over Finite Domains ($CFLP(\mathcal{FD})$) with the lazy functional logic programming language $\mathcal{TOY}$ which seamlessly embodies finite domain ($\mathcal{FD}$) constraints. $CFLP(\mathcal{FD})$ increases the expressiveness and power of constraint logic programming over finite domains ($CLP(\mathcal{FD})$) by combining functional and relational notation, curried expressions, higher-order functions, patterns, partial applications, non-determinism, lazy evaluation, logical variables, types, domain variables, constraint composition, and finite domain constraints.

3.1 Introduction

$CFLP(\mathcal{FD})$ provides the main characteristics of $CLP(\mathcal{FD})$, i.e., $\mathcal{FD}$ constraint solving, non-determinism and relational form. Moreover, $CFLP(\mathcal{FD})$ provides a sugaring syntax for LP predicates and thus, any pure $CLP(\mathcal{FD})$-program can be straightforwardly translated into a $CFLP(\mathcal{FD})$-program. In this sense, $CLP(\mathcal{FD})$ may be considered as a strict subset of $CFLP(\mathcal{FD})$ with respect to problem formulation. As a direct consequence, our language is able to cope with a wide range of applications (in particular, all those pure programs that can be formulated with a $CLP(\mathcal{FD})$ language).
Due to its functional component, $CFLP(\mathcal{FD})$ adds further expressiveness to $CLP(\mathcal{FD})$ as allows the declaration of functions and their evaluation in the FP style.
Functions are first-class citizens, which means that a function (and thus any $\mathcal{FD}$ constraint) can appear in any place where data do. As a direct consequence, an $\mathcal{FD}$ constraint may appear as an argument (or even as a result) of another function or constraint. The functions managing other functions are called higher-order (HO) functions. Also, polymorphic arguments are allowed in $CFLP(\mathcal{FD})$.
In contrast to logic languages, functional languages support *lazy evaluation*, where function arguments are evaluated to the required extend (the *call-by-value* used in LP vs. the call-by-need used in FP). Strictly speaking, lazy evaluation may also correspond to the notion of *only once evaluated* in addition to *only required extent* [31]. $\mathcal{TOY}$ increases the power of $CLP(\mathcal{FD})$ by incorporating a novel mechanism that combines *lazy evaluation* and $\mathcal{FD}$ constraint solving, in such a way that only the demanded constraints are sent to the solver. For example, it is possible

to manage infinite lists of constraints.

3.1.1 Efficiency

In [9] we showed that $\mathcal{TOY}$ is fairly efficient as, in general, behaves closely to SICStus. Despite this is not surprisingly as it is implemented on top of SICStus, we think that it is important to show that the wrapping of SICStus by $\mathcal{TOY}$ does not increase significantly the computation time.
Moreover, in the same paper we showed that $\mathcal{TOY}$ is about two and five times faster (and even much more in scalable problems) than another $CFLP(\mathcal{FD})$ implementation [26] which is said to be efficient in its Web page (`http://redstar.cs.pdx.edu/~pakcs/`).

3.1.2 Modes and Bidirectionality

Each `N`-arity predefined constraint and (constraint-related-)function is listed with its corresponding modes for arguments and results as:
`function :=: { modes }`
where `modes` is a comma-separated list of modes of the form:
`mode0 -> mode1 -> ... modeN`, where `function` has arity `N`, `mode`i ($0 \leq i < N$) is the mode for the i-th argument, `modeN` is the mode for the result, and `mode`i can be:

- A `basic mode`:
  - `in`. An `in` mode demands a ground expression.
  - `inout`. An `inout` mode accepts any expression.
- A `constructed mode`:
  - `[basic mode]`. Demands a list of elements, each one of `basic mode`.
  - `(basic mode1, ..., basic modeN)`. Demands an `N`-tuple of elements, in which the i-th element is of `basic mode`i.

Arguments of predefined functions and constraints with finite types are not demanded and can otherwise be produced. For example, the second argument of the arithmetic constraint `sum` stands for a relational constraint operator. If it is not provided by the user, the system produces all the possible operators. The mode for `sum` is listed as: `sum :=: {[in] -> inout -> in}`. Some other functions can have several usage modes.

3.1.3 Loading the $\mathcal{FD}$ Constraint Library

$\mathcal{TOY}$ does not incorporate automatically $\mathcal{FD}$ constraints and thus it is necessary to load the $\mathcal{FD}$ extension in order to perform constraint solving. This is done with the command `/cflpfd`. See also Section 1.5.5 in page 15 for an introductory system session.

3.2 $\mathcal{FD}$ Constraints and Functions

In this section, $\mathcal{TOY}$ constraints and (constraint-related-)functions are listed and explained. Several of them are reifiable, i.e., a reified constraint represents its entailment in the constraint store with a Boolean variable. In general, constraints in reified form allow their fulfillment to be reflected back in a variable. For example, `B #= (X #+ Y #> Z)` constrains `B` to `true` as soon as the disequation is known to be true and to `false` as soon as the disequation is known to be false. On the other hand, constraining `B` to `true` imposes the disequation, and constraining `B` to `false` imposes its negation. Incidentally, in CLP(FD) languages, the Boolean values `false` and `true` usually correspond to the numerical values 0 and 1, respectively.

3.2.1 Predefined Data Types

In $\mathcal{TOY}$, we have defined a set of predefined datatypes that are used to define the constraints. Table 3.1 shows the set of these datatypes.

Table 3.1: Predefined Datatypes for $\mathcal{FD}$ Constraints

```
data labelingType = ff | ffc | leftmost | mini | maxi | step | enum | bisect | up
            | down | each | toMinimize int | toMaximize int | assumptions int
data reasoning = value | domains | range
data allDiffOptions = on reasoning | complete bool
data typeprecedence = d (int,int,liftedInt)
data liftedInt = superior | lift int
data serialOptions = precedences [typeprecedence] | path_consistency bool
            | static_sets bool | edge_finder bool | decomposition bool
data range = cte int int | uni range range | inter range range | compl range
data fdinterval = interval int int
type fdset = [fdinterval]
data statistics = resumptions | entailments | prunings | backtracks
            | constraints
```

In the following, we describe all the $\mathcal{FD}$ constraints, functions, and operators currently supported by $\mathcal{TOY}$ (see Table 3.2), as well as the set of impure $\mathcal{FD}$ functions (see Table 3.3), in the sense that they depend on the program state. Despite they are useful in several applications (like defining search strategies), the programmer should be aware of its impure functional behaviour.

3.2.2 Membership Constraints

Membership constraints restrict the values that a variable can be assigned to. There are several such constraints:

`domain/3`

- **Type Declaration:**

Table 3.2: The Predefined Set of $\mathcal{FD}$ Constraints, Functions, and Operators

MEMBERSHIP CONSTRAINTS
domain :: [int] → int → int → bool
subset, inset, setcomplement :: int → int → bool
intersect :: int → int → int
belongs :: int → [int] → bool
ENUMERATION FUNCTIONS
labeling :: [labelType] → [int] → bool
indomain :: int → bool
RELATIONAL CONSTRAINT OPERATORS
#=, #\=, #<, #<=, #>, #>= :: int → int → bool
ARITHMETIC CONSTRAINT OPERATORS
#+, #-, #*, #/, #& :: int → int → int
ARITHMETIC CONSTRAINTS
sum :: [int] → (int → int → bool) → int → bool
scalar_product :: [int] → [int] → (int → int → bool) → int → bool
COMBINATORIAL CONSTRAINTS
all_different :: [int] → bool
all_different' :: [int] → [allDiffOptions] → bool
circuit :: [int] → bool
assignment, circuit', serialized :: [int] → [int] → bool
serialized' :: [int] → [int] → [serialOptions] → bool
count :: int → [int] → (int → int → bool) → int → bool
element, exactly :: int → [int] → int
cumulative :: [int] → [int] → [int] → int → bool
cumulative' :: [int] → [int] → [int]→ int → [serialOptions] → bool
PROPOSITIONAL CONSTRAINTS
#<=>, #=>, #\/ :: bool → bool → bool

Table 3.3: The Predefined Set of Impure $\mathcal{FD}$ Functions

REFLECTION FUNCTIONS
fd_min, fd_max, fd_size, fd_degree :: int → int
fd_set :: int → fdset
fd_dom :: int → fdrange
fd_neighbors :: int → [int]
fd_closure :: [int] → [int]
inf, sup :: int
FD SET FUNCTIONS
is_fdset, empty_fdset :: fdset → bool
fdset_parts :: int → int → fdset → fdset
empty_interval :: int → int → bool
fdset_interval :: int → int → fdset
fdset_singleton :: int → fdset
fdset_min, fdset_max, fdset_size :: fdset → int
list_to_fdset :: [int] → fdset
fdset_to_list :: fdset → [int]
range_to_fdset :: [int] → range
fdset_to_range :: range → [int]
fdset_add_element, fdset_del_element :: fdset → int → fdset
fdset_intersection, fdset_subtract,
fdset_union :: fdset → fdset → fdset
fdset_complement :: fdset → fdset
fdsets_intersection, fdsets_union :: fdset → fdset
fdset_equal, fdset_subset, fdset_disjoint,
fdset_intersect :: fdset → fdset → bool
fdset_member :: int → fdset → bool
fdset_belongs :: int → int → bool
STATISTICS FUNCTIONS
fd_statistics :: bool
fd_statistics' :: statistics → int → bool

`domain :: [int] → int → int → bool`

- **Modes:**

  `domain :=: {[inout] → in → in → inout}`

- **Definition:** `domain L A B` returns `true` if each element in the list `L` (with integers and/or $\mathcal{FD}$ variables) belongs to the interval [`A`,`B`] and also constrains each $\mathcal{FD}$ variable in L to have values in the integer interval [`A`,`B`]. It returns `false` if no element in L belongs to [`A`,`B`].

- **Reifiable:** Yes.

- **Examples:**

  The next goal returns `true` and constrains `X` and `Y` to have values in the range [1,10].

```
Toy(FD)> domain [X,Y] 1 10
      { X in 1..10,
        Y in 1..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

  The next goal shows a use of reification:

```
Toy(FD)> domain [X] 0 1 == B
      { B -> true }
      { X in 0..1 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { X in(inf.. -1)\/(2..sup) }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

  Observe that in the first solution (`B -> true`) `X` is constrained to have values in the closed integer interval [0..1], whereas in the second solution (`B -> false`) `X` is constrained to have a value different from those.

- **Note:**

  The possible values for a finite domain variable lies in the interval [`inf`,`sup`], where `inf` and `sup` denote, respectively, the functions that return the minimum and maximum integer values that a finite domain variable can take. See section 3.2.9 for the definition of these functions.

`subset/2, inset/2, setcomplement/2`

- **Type Declaration:**

  `subset, inset, setcomplement :: int → int → bool`

- **Modes:**

  `subset, inset, setcomplement :=: {inout → inout → inout}`

- **Definition:** `subset A B` returns `true` if the domain of `A` is a subset of the domain of `B`. It returns `false` in other case. `inset A B` returns `true` if `A` is an element of the $\mathcal{FD}$ set of `B`. It returns `false` in other case. `setcomplement A B` returns `true` if no value of the domain of `A` is in the domain of `B`. It returns `false` in other case. `setcomplement` is the complement of `subset`.

- **Reifiable:** Yes.

- **Examples:**

```
Toy(FD)> domain [X] 1 4, domain [Y] 3 6, subset X Y == B
      { B -> true }
      { subset X Y,
        X in 3..4,
        Y in 3..6 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { not subset X Y,
        X in 1..4,
        Y in 3..6 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(FD)> domain [X] 1 4, domain [Y] 3 6, inset X Y == B
      { B -> true }
      { X in 3..4,
        Y in 3..6 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { X in 1..2,
        Y in 3..6 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
```

```
      no
      Elapsed time: 0 ms.

Toy(FD)> domain [X] 0 1, domain [Y] 1 4, setcomplement X Y == B
      { B -> true }
      { not subset X Y,
        X in 0..1,
        Y in 1..4 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { X -> 1,
        B -> false }
      { subset 1 Y,
        Y in 1..4 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`intersect/2`

- **Type Declaration:**

  `intersect :: int → int → int`

- **Modes:**

  `intersect :=: {inout → inout → inout}`

- **Definition:** `intersect A B` returns a variable with the intersected domains of `A` and `B`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X] 0 3, domain [Y] 1 4, intersect X Y == Z
      { X in 0..3,
        Y in 1..4,
        Z in 1..3 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`belongs/2`

- **Type Declaration:**

  `belongs :: int → [int] → bool`

- **Modes:**

  `belongs :=: {inout → [in] → inout}`

- **Definition:** `belongs A L` returns `true` if the integer or variable domain takes a value in the list `L` (with integers and/or $\mathcal{FD}$ variables). It returns `false` in other case.

- **Reifiable:** Yes.

- **Example:**

  The next goal shows a use of reification:

```
Toy(FD)> domain [X] 0 4, belongs X [1,3] == B, labeling [] [X]
      { X -> 1,
        B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]? a
      { X -> 3,
        B -> true }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { X -> 0,
        B -> false }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      { X -> 2,
        B -> false }
      Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
      { X -> 4,
        B -> false }
      Elapsed time: 0 ms.
sol.5, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Note: `labeling` is intended to enumerate the possible solutions when propagation alone is not enough to deliver concrete values. This enumeration function is covered in the next section.

3.2.3 Enumeration Functions

Propagation algorithms usually found in constraint solving are incomplete due to efficiency reasons. This means that the solving process can terminate without determining whether a

constraint store is entailed or not. Enumeration constraints allow to reactivate the search for solutions when no more constraint propagation is possible. They ask to the solver for all the solutions via backtracking by assigning values to $\mathcal{FD}$ variables following an enumeration strategy (satisfaction problems). Also, it is possible to select, from all the solutions, the one which optimizes some cost function (optimization problems). It does not make any sense to reify enumeration functions. $\mathcal{TOY}$ provides two enumeration constraints:

`labeling/2`

- **Type Declaration:**

  `labeling :: [labelType] → [int] → bool`

- **Modes:**

  `labeling :=: {inout → [inout] → inout}`
  Note: Since the first argument is `inout`, the lists of valid options can be produced. However, not all of them are produced because two of them demands a variable to be optimized, as will be explained next.

- **Definition:** `labeling Options L` is true if an assignment of the variables in `L` can be found such that all of the constraints already presented in the constraint store are satisfied. `L` is a list of integers or domain variables with a bounded domain. `Options` is a list of at most four elements that allow to specify an enumeration strategy.

  Each element in this list can have a value in one of the following groups:

  1. The first group controls the order in which variables are chosen for assignment (i.e., variable ordering) and allows to select the leftmost variable in `L` (`leftmost`), the variable with the smallest lower bound (`mini`), the variable with the greatest upper bound (`maxi`) or the variable with the smallest domain (`ff`). The value `ffc` extends the option `ff` by selecting the variable involved in the higher number of constraints.
  2. The second group controls the value ordering, that is to say, the order in which values are chosen for assignment. For instance, from the minimum to the maximum (`enum`), by selecting the minimum or maximum (`step`) or by dividing the domain in two choices by the mid point (`bisect`). Also the domain of a variable can be explored in ascending order (`up`) or in descending order (`down`).
  3. The third group demands to find all the solutions (`all`) or only one solution to maximize (resp. minimize) the domain of a variable X in L (`toMinimize X`) (resp. `toMaximize X`).
  4. The fourth group controls the number `K` of assumptions (`choices`) made during the search (`assumptions K`).

- **Reifiable:** Not applied.

- **Example:**

The next goal looks (via backtracking) for assignments to all variables in the list `L` and follows a first fail strategy.

```
Toy(FD)> L == [X,Y], domain L 1 2, labeling [ff] L
      { L -> [ 1, 1 ],
        X -> 1,
        Y -> 1 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { L -> [ 1, 2 ],
        X -> 1,
        Y -> 2 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { L -> [ 2, 1 ],
        X -> 2,
        Y -> 1 }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      { L -> [ 2, 2 ],
        X -> 2,
        Y -> 2 }
      Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`indomain/1`

- **Type Declaration:**

  `indomain :: int → bool`

- **Modes:**

  `indomain :=: {inout → inout}`

- **Definition:** `indomain A` assigns a value, from the minimum to the maximum in its domain, to `A`, which is an integer or a finite domain variable with a bounded domain. It always returns `true`.

- **Reifiable:** Not applied.

- **Example:**

  The evaluation of the next goal assigns to `X` each value in its domain via backtracking.

```
Toy(FD)> domain [X] 1 2, indomain X
      { X -> 1 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { X -> 2 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.4 Relational Constraints

Relational constraints include equality and disequality constraints in the form $e \diamond e'$ where $\diamond \in \{$ `#=`, `#\=`, `#<`, `#<=`, `#>`, `#>=` and e and e' are either integers, or $\mathcal{FD}$ variables or functional expressions.

`#=` / 2, `#\=` / 2, `#<` / 2, `#<=` / 2, `#>` / 2, `#>=` / 2

- **Type Declaration:**

  `#=, #\=, #<, #<=, #>, #>= ::` `int` → `int` → `bool`

- **Modes:**

  `#=, #\=, #<, #<=, #>, #>= :=:` {`inout` → `inout` → `inout`}

- **Definition:** `#Op A B`, also written in infix notation as `A #Op B`, where `Op` ∈ { `#=`, `#\=`, `#<`, `#<=`, `#>`, `#>=` }, returns `true` if posting the relation `A #Op B` entails the constraint store. It returns `false` in other case.

- **Reifiable:** Yes.

- **Remarks:** Infix notation allowed.

- **Priorities:**

```
infix 20 #=, #\=                  (infix)
infix 30 #<, #<=, #>, #>=         (infix)
```

- **Examples:**

  The next goal returns `true` and restricts `X` and `Y` to have values in the closed integer intervals [2,10] and [1,9] respectively (as a consequence of range narrowing).

```
Toy(FD)> domain [X,Y] 1 10, X #> Y
      { Y #< X,
        X in 2..10,
        Y in 1..9 }
```

```
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The next goal shows a use of reification:

```
Toy(FD)> domain [X,Y] 0 1, X #> Y == B
      { X -> 1,
        Y -> 0,
        B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { X #=< Y,
        X in 0..1,
        Y in 0..1 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.5 Arithmetic Constraint Operators

Arithmetic constraint operators allow to build linear and nonlinear constraints involving primitive arithmetic functions as addition (`#+`), subtraction (`#-`), multiplication (`#*`), and division (`#/`). It does not make any sense to reify these operators.

`#+` / 2, `#-` / 2, `#*` / 2, `#/` / 2, `#&` / 2

- **Type Declaration:**

  `#+, #-, #*, #/, #& ::` `int` → `int` → `int`

- **Modes:**

  `#+, #-, #*, #/, #& :=:` {`inout` → `inout` → `inout`}

- **Definition:** `#Op A B`, also written in infix notation as `A #Op B`, where Op ∈ { `+`, `-`, `*`, `/`, `&` }, imposes the constraint `A #Op B` and returns `true` if it is entailed.

- **Reifiable:** Not applied.

- **Remarks:** Infix notation allowed. `&` stands for integer modulus.

- **Priorities:**

```
infixr 90 #*, #/              (infix, right-associative)
infixl 50 #+, #-              (infix, left-associative)
infixl 90 #&                  (infix, left-associative)
```

- **Example:**

  The next goal returns `true` and constrains `X`, `Y` and `Z` to have values in the intervals [1,10], [1,10] and [1,100] respectively.

```
Toy(FD)> domain [X,Y] 1 10, X #* Y #= Z
      { X #* Y #= Z,
        X in 1..10,
        Y in 1..10,
        Z in 1..100 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.6 Arithmetic Constraints

The following constraints express a relation between a sum or scalar product and a value, and use dedicated algorithms that avoid to allocate temporary variables holding intermediate values.

`sum/3`

- **Type Declaration:**

  `sum :: [int] → (int → int → bool) → int → bool`

- **Modes:**

  `sum :=: {[inout] → inout → inout}`

- **Definition:** `sum L Op V` is true if the sum of all elements in `L` is related with `V` via the relational constraint operator `Op`, i.e., if $\Sigma_{e \in L}\ Op\ V$.

- **Reifiable:** Yes.

- **Note**: This constraint implements a dedicated algorithm which posts a single sum instead of a sequence of elementary constraints.

- **Example:**

```
Toy(FD)> L == [X,Y,Z], domain L 1 3, sum L (#<) 4 == B
      { L -> [ 1, 1, 1 ],
        X -> 1,
        Y -> 1,
```

```
        Z -> 1,
        B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { L -> [ X, Y, Z ],
        B -> false }
      { X#+Y#+Z #>= 4,
        X in 1..3,
        Y in 1..3,
        Z in 1..3 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`scalar_product/4`

- **Type Declaration:**

  `scalar_product :: [int] → [int] → (int → int → bool) → int → bool`

- **Modes:**

  `scalar_product :=: {[inout] → [inout] → inout → inout → inout}`

- **Definition:** `scalar_product L1 L2 Op V` is true if the scalar product (in the sense of $\mathcal{FD}$ constraint solving) of the integers in `L1` and the integers or $\mathcal{FD}$ variables in `L2` is related with the value `V` via the relational constraint operator `Op`, i.e., if `(L1` $*_s$ `L2) Op V` is satisfied with $*_s$ defined as the usual scalar product of integer vectors.

- **Reifiable:** Yes.

- **Note**: This constraint implements a dedicated algorithm which posts a single scalar product instead of a sequence of elementary constraints.

- **Example:**

```
Toy(FD)> domain [X,Y,Z] 1 10, scalar_product [1,2,3] [X,Y,Z] (#<) 10 == B
      { B -> true }
      { X#+2#*Y#+3#*Z #< 10,
        X in 1..4,
        Y in 1..2,
        Z in 1..2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { X#+2#*Y#+3#*Z #>= 10,
```

```
        X in 1..10,
        Y in 1..10,
        Z in 1..10 }
      Elapsed time: 16 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

As expected, the expressions constructed from both arithmetic and relational constraints may be non-linear.

3.2.7 Combinatorial Constraints

Combinatorial constraints include well-known global constraints that are useful to solve problems defined on discrete domains. Often, these constraints are also called *symbolic constraints* [2]. These constraints implement dedicated algorithms which are more efficient than posting elementary constraints.

`all_different/1`
`all_different'/2`

- **Type Declaration:**

  `all_different :: [int] → bool`
  `all_different' :: [int] → [allDiffOptions] → bool`

- **Modes:**

  `all_different :=: {[inout] → inout}`
  `all_different' :=: {[inout] → inout → inout}`

- **Definition:** `all_different L` returns `true` if each finite domain variable (with bounded domain) in `L` is constrained to have a value that is unique in the list `L` and there are no duplicate integers in the list `L`, i.e., this is equivalent to say that for all $X, Y \in$ `L`, $X \neq Y$.

  The extended version `all_different' L Options` allow one more argument which is a list of options. This list can take at most one value from the following two groups:

  1. `on value`, `on domains` or `on range` to specify that the constraint has to be woken up, respectively, when a variable becomes ground, when the domain associated to a variable changes, or when a bound of the domain (in interval form) associated to variable changes.
  2. `complete true` or `complete false` to specify if the propagation algorithm to apply is complete or incomplete.

- **Reifiable:** No.

- **Example:**

```
Toy(FD)> L == [X, Y, Z], domain L 1 3,
          all_different' L [complete true, on range]
       { L -> [ X, Y, Z ] }
       { all_different [X,Y,Z] [complete(true),on(range)],
         X in 1..3,
         Y in 1..3,
         Z in 1..3 }
       Elapsed time: 15 ms.
sol.1, more solutions (y/n/d/a) [y]?
       no
       Elapsed time: 0 ms.
```

`assignment/2`

- **Type Declaration:**

  `assignment :: [int] → [int] → bool`

- **Modes:**

  `assignment :=: {[inout] → [inout] → inout}`

- **Definition:** `assignment L1 L2` is a function applied over two lists of domain variables with length n where each variable takes a value in $\{1, \ldots, n\}$ which is unique for that list. Then, it returns `true` if for all $i, j \in \{1, \ldots, n\}$, and $X_i \in$ `L1`, $Y_j \in$ `L2`, then $X_i = j$ if and only if $Y_j = i$.

- **Reifiable:** No.

- **Example:**

  The next goal returns `true` and constrains `X`, `Y` and `Z` to be 3, 1 and 2 respectively.

```
Toy(FD)> domain [X,Y,Z] 1 3, assignment [X,Y,Z] [2,3,D]
       { X -> 3,
         Y -> 1,
         Z -> 2,
         D -> 1 }
       Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
       no
       Elapsed time: 0 ms.
```

`circuit/1`
`circuit'/2`

- **Type Declaration:**

```
circuit ::  [int] → bool
circuit' ::  [int] → [int] → bool
```

- **Modes:**

```
circuit :=:  {[inout] → inout}
circuit' :=:  {[inout] → [inout] → inout}
```

- **Definition:** `circuit L1` and `circuit' L1 L2` return `true` if the values in `L1` form a Hamiltonian circuit. This constraint can be thought of as constraining n nodes in a graph to form a Hamiltonian circuit where the nodes are numbered from 1 to n and the circuit starts in node 1, visits each node and returns to the origin. `L1` and `L2` are lists of $\mathcal{FD}$ variables or integers of length n, where the i-th element of `L1` (resp. `L2`) is the successor (resp. predecessor) of i in the graph.

- **Reifiable:** No.

- **Example:**

```
Toy(FD)> domain [X,Y,Z] 1 3, circuit [X,Y,Z]
      { circuit [X,Y,Z],
        X in 2..3,
        Y in{1}\/{3},
        Z in 1..2 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`count/4`

- **Type Declaration:**

```
count ::  int → [int] → (int → int → bool) → int → bool
```

- **Modes:**

```
count :=:  {in → inout → inout → inout → inout}
```

- **Definition:** `count V L Op Y` returns `true` if the number of elements of `L` that are equal (in the sense of $\mathcal{FD}$ constraint equality) to `V` is `N` and also `N` is related with `Y` via the relational constraint operator `Op` (i.e., `N Op Y` holds).

- **Reifiable:** No.

- **Example:**

  The next goal returns `true` and constrains `X` to be 1 as the number of elements in `L` that are equal to 1 is imposed to be exactly 2.

```
Toy(FD)> L == [X,1,2], domain [X] 1 2, count 1 L (#=) 2
      { L -> [ 1, 1, 2 ],
        X -> 1 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`element/3`

- **Type Declaration:**

  `element :: int → [int] → int → bool`

- **Modes:**

  `element :=: {inout → [inout] → inout → inout}`

- **Definition:** `element I L X` returns `true` if `X` is the `I`-th element in the list `L` (in the sense of $\mathcal{FD}$). `I`, `X`, and the elements of `L` are integers or domain variables.

- **Reifiable:** No.

- **Example:**

```
Toy(FD)> L == [X,Y,Z], domain L 1 10, element 2 L 7
      { L -> [ X, 7, Z ],
        Y -> 7 }
      { X in 1..10,
        Z in 1..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`exactly/3`

- **Type Declaration:**

  `exactly :: int → [int] → int → bool`

- **Modes:**

  `exactly :=: {inout → inout → inout → inout}`

- **Definition:** `exactly X L N` returns `true` if `X` occurs `N` times in the list `L`.

- **Reifiable:** No.

- **Example:**

  The next goal imposes that the two elements of a list have to be equal to 2.

```
Toy(FD)> L == [X,Y], domain L 1 2, exactly 2 L 2
      { L -> [ 2, 2 ],
        X -> 2,
        Y -> 2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`serialized/2`
`serialized'/3`
`cumulative/4`
`cumulative'/5`

- **Type Declaration:**

```
serialized ::  [int] → [int] → bool
serialized' ::  [int] → [int] → [serialOptions] → bool
cumulative ::  [int] → [int] → [int] → int → bool
cumulative' ::  [int] → [int] → [int]→ int → [serialOptions] → bool
```

- **Modes:**

```
serialized :=:  {[inout] → [inout] → inout}
serialized' :=:  {[inout] → [inout] → inout → inout}
cumulative :=:  {[inout] → [inout] → [inout] → inout}
cumulative' :=:  {[inout] → [inout] → [inout] → inout → inout}
```

- **Definition:** `serialized`, `serialized'`, `cumulative` and `cumulative'` are useful to solve scheduling and placements problems. In general,

  `serialized [S1,...,Sn] [D1,...,Dn]`

  `serialized' [S1,...,Sn] [D1,...,Dn] Options`

  `cumulative [S1,...,Sn] [D1,...,Dn] [R1,...,Rn] Limit`

  `cumulative' [S1,...,Sn] [D1,...,Dn] [R1,...,Rn] Limit Options`

  return `true` if it is possible to constrain n tasks `Ti` $(1 \leq i \leq n)$, each with a start time `Si` and duration `Di` so that no task overlaps.

  Particularly, cumulative constraints also impose a limit to check that, given a resource amount `Ri` for each task, the total resource consumption does not exceed the limit `Limit` at any time. `Si`, `Di`, and `Ri` are integers or domain variables with finite bounds. `Options` is a list of elements of type `serialOptions` that enables certain options, usually dependent on the problem, in order to improve the search of the solutions. Specifically speaking:

- `path_consistency true` enables a redundant path consistency algorithm to improve the pruning;
- `static_sets true` and `edge_finder true` active the use of redundant algorithms to take advantage of the precedence relations among the tasks;
- `decomposition true` actives attempts to decompose the constraints each time the search is resumed;
- `precedences L` provides a list `L` of precedence constraints where each element in `L` has the form (V_1, V_2, I), and I is the greatest value (to denote a fictitious -lifted- top element) or lift I with $I \in$ Integers. Each element imposes the constraint $S_{V_1} + I \leq S_{V_2}$, if I is an integer; and $S_{V_2} \leq S_{V_1}$ otherwise.

- **Reifiable:** No.
- **Examples:**

```
Toy(FD)> cumulative [0,5,8] [1,1,2] == U, U [1,1,1] 10 == I
      { U -> (cumulative [ 0, 5, 8 ] [ 1, 1, 2 ]),
        I -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(FD)> serialized' [1,3] [1,1]
         [path_consistency true,
          static_sets true,
          edge_finder true,
          decomposition true,
          precedences [d(1,2,lift(2))] ] == B
      { B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.8 Propositional Constraints

Relational constraint functions returning Boolean values can be arranged into propositional (Boolean) formulae by means of propositional combinators. $\mathcal{TOY}$ provides the following propositional combinators:

`#<=>` / 2, `#=>` / 2, `#\/` / 2

- **Type Declaration:**

  `#<=>, #=>, #\/ ::  bool` → `bool` → `bool`

- **Modes:**

  `#<=>, #=>, #\/ :=:` {inout → inout → inout}

- **Definition:**

  - `P #<=> Q` (equivalence) returns `true` if `P` and `Q` are both `true` or `false`.
  - `P #=> Q` (implication) returns `true` if `P` is `false` or `Q` is `true`.
  - `P #\/ Q` (disjunction) returns `true` if at least one of `P` and `Q` is `true`.

  Note that these constraints demand Boolean expressions, which may be Boolean constraints or Boolean expressions in general.

- **Priorities:**

```
infix 15 #<=>, #\/, #=>            (infix)
```

- **Reifiable:** No.

- **Example:**

```
Toy(FD)> (X#>0)#\/Y
      { X in 1..sup }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { Y -> true }
      { X in inf..0 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.9 Reflection Functions

$\mathcal{TOY}$ provides a set of (impure) functions called *reflection functions*, that allow to both recover information about variable domains, and to manage $\mathcal{FD}$ sets during goal solving. It does not make any sense to reify reflection functions.

`fd_var/1`

- **Type Declaration:**

  `fd_var :: int → bool`

- **Modes:**

  `fd_var :=:` {inout → inout}

- **Definition:** `fd_var V` returns `true` if `V` is an unbound $\mathcal{FD}$ variable in the constraint store.

- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X] 0 1, fd_var X == BX, fd_var Y == BY, fd_var 1 == B1,
         X == 1, fd_var X == BX1
      { X -> 1,
        BX -> true,
        BY -> false,
        B1 -> false,
        BX1 -> false }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Note: In particular, this example shows the impure behaviour of this reflection function, since the meaning of `fd_var X` depends on its location in the goal expression.

`fd_min/1, fd_max/1`

- **Type Declaration:**

  `fd_min, fd_max :: int → int`

- **Modes:**

  `fd_min, fd_max :=: {inout → inout}`

- **Definition:** `fd_min V` and `fd_max V` return the smallest (greatest) value in the current domain of `V`.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X] 0 10, Min == fd_min X, Max == fd_max X
      { Min -> 0,
        Max -> 10 }
      { X in 0..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_size/1`

- **Type Declaration:**

  `fd_size :: int → int`

- **Modes:**

  `fd_size :=: {inout → inout}`

- **Definition:** `fd_size V` returns the cardinality of the domain of `V`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X] 0 10, Card == fd_size X
      { Card -> 11 }
      { X in 0..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_set/1`

- **Type Declaration:**

  `fd_set :: int → fdset`

- **Modes:**

  `fd_set :=: {inout → inout}`

- **Definition:**

  `fd_set V` returns the $\mathcal{FD}$ set denoting the internal representation of the current domain of `V`.

- **Reifiable:** Not applied.

- **Examples:**

```
Toy(FD)> domain [X] 0 10, X #\= 5, FDSet == fd_set X
      { FDSet -> [ (interval 0 4), (interval 6 10) ] }
      { X in (0..4)\/(6..10) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(FD)> domain [X] 0 1, P == fd_set, P X == [(interval (1-1) X)]
```

```
      { X -> 1,
        P -> fd_set }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_dom/1`

- **Type Declaration:**

  `fd_dom :: int → fdrange`

- **Modes:**

  `fd_dom :=: {inout → inout}`

- **Definition:** `fd_dom V` returns a constant range denoting the current domain of `V`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X] 0 10, X #\= 5, Dom == fd_dom X
      { Dom -> (uni (cte 0 4) (cte 6 10)) }
      { X in (0..4)\/(6..10) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_degree/1`

- **Type Declaration:**

  `fd_degree :: int → int`

- **Modes:**

  `fd_degree :=: {inout → inout}`

- **Definition:** `fd_degree V` returns the number of constraints that are attached to `V`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X,Y] 0 10, X #> Y, Degree == fd_degree X
      { Degree -> 1 }
      { Y #< X,
        X in 1..10,
        Y in 0..9 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_neighbors/1`

- **Type Declaration:**

  `fd_neighbors :: int → [int]`

- **Modes:**

  `fd_neighbors :=: {inout → inout}`

- **Definition:** `fd_neighbors V` returns the set of $\mathcal{FD}$ variables that can be reached from `V` via constraints posted so far.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X,Y] 0 10, X #\= 5, X #< Y, Vars == fd_neighbors X
      { Vars -> [ _C, X, Y ] }
      { Y #> X,
        X in(0..4)\/(6..9),
        Y in 1..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_closure/1`

- **Type Declaration:**

  `fd_closure :: [int] → [int]`

- **Modes:**

  `fd_closure :=: {[inout] → [inout]}`

- **Definition:** `fd_closure [V]` returns the set of $\mathcal{FD}$ variables (including `[V]`) that can be transitively reached via constraints posted so far. Thus, `fd_closure/1` returns the transitive closure of `fd_neighbors/1`.

- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X,Y,Z] 0 10, X #< Y, Y #< Z, fd_closure [X] == L
      { L -> [ _D, _E, X, Y, Z ] }
      { Z #> Y,
        Y #> X,
        X in 0..8,
        Y in 1..9,
        Z in 2..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`inf, sup/0`

- **Type Declaration:**

  `inf, sup :: int`

- **Modes:**

  `inf, sup :=:` {}

- **Definition:** `inf` and `sup` return, respectively, the minimum and maximum values that a finite domain variable can be assigned to.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> Min == inf, Max == sup
      { Min -> -33554432,
        Max -> 33554431 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.10 $\mathcal{FD}$ Set Functions

$\mathcal{TOY}$ provides a set of (impure) functions for managing $\mathcal{FD}$ sets during goal solving. It does not make any sense to reify $\mathcal{FD}$ set functions.

`is_fdset/1`

- **Type Declaration:**

  `is_fdset :: int → bool`

- **Modes:**

  `is_fdset :=: {in → inout}`

- **Definition:** `is_fdset S` returns `true` if `S` is a valid $\mathcal{FD}$ set. Otherwise, it returns `false`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> is_fdset [ (interval 0 1) ] == B1,
         is_fdset [ (interval 1 0) ] == B2
      { B1 -> true,
        B2 -> false }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`empty_fdset/1`

- **Type Declaration:**

  `empty_fdset :: int → bool`

- **Modes:**

  `empty_fdset :=: {inout → inout}`

- **Definition:** `empty_fdset S` returns `true` if `S` is an empty $\mathcal{FD}$ set. Otherwise, it returns `false`.

- **Reifiable:** Not applied.

- **Example:**

```
Finite Domain Constraints library loaded.
Toy(FD)> empty_fdset S == B
      { S -> [],
        B -> true }
      Elapsed time: 10 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { B -> false }
      { [] /= S }
      Elapsed time: 10 ms.
```

```
sol.2, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_parts/3`, `fdset_split/3`

- **Type Declaration:**

  ```
  fdset_parts ::  int → int → fdset → fdset
  fdset_split ::  fdset → (int, int, fdset)
  ```

- **Modes:**

  ```
  fdset_parts :=:  {in → in → in → inout}
  fdset_split :=:  {in → (inout, inout, inout)}
  ```

- **Definition:** `fdset_parts Min Max P` returns the $\mathcal{FD}$ set which is the union of the non-empty interval `[Min,Max]` and the $\mathcal{FD}$ set `P`, and all elements of `P` are greater than `Max+1`. Both `Min` and `Max` are integers or the functions `inf` and `sup`. `fdset_split S` returns the tuple of parameters (`Min`, `Max`, `P`) which constructs the $\mathcal{FD}$ set `S` in the same way as `fdset_parts`.

- **Reifiable:** Not applied.

- **Example:**

  ```
  Toy(FD)> fdset_parts 0 1 [interval 3 4] == S, fdset_split S == T
        { S -> [ (interval 0 1), (interval 3 4) ],
          T -> (0, 1, [ (interval 3 4) ]) }
        Elapsed time: 0 ms.
  sol.1, more solutions (y/n/d/a) [y]?
        no
        Elapsed time: 0 ms.
  ```

`empty_interval/2`

- **Type Declaration:**

  ```
  empty_interval ::  int → int → bool
  ```

- **Modes:**

  ```
  empty_interval :=:  {in → in → inout}
  ```

- **Definition:** `empty_interval Min Max` returns `true` if `[Min, Max]` is the empty $\mathcal{FD}$ set. Otherwise, it returns `false`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> empty_interval 0 1 == B1, empty_interval 1 0 == B2
      { B1 -> false,
        B2 -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_to_interval/2`, `interval_to_fdset/2`

- **Type Declaration:**

```
fdset_to_interval ::  fdset → (int, int)
interval_to_fdset ::  int → int → fdset
```

- **Modes:**

```
fdset_to_interval :=:  {in → (inout, inout)}
interval_to_fdset :=:  {in → in → inout}
```

- **Definition:** `fdset_to_interval S` returns a tuple, which is the non-empty interval `[Min, Max]`. `interval_to_fdset Min Max` is the inverse operation of `fdset_to_interval`.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> fdset_to_interval [(interval 0 1)] == (Min, Max),
         interval_to_fdset 0 1 == S
      { Min -> 0,
        Max -> 1,
        S -> [ (interval 0 1) ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_singleton/2`

- **Type Declaration:**

```
fdset_singleton ::  fdset → int → bool
```

- **Modes:**

```
fdset_singleton :=:  {in → inout → inout, inout → in → inout}
```

- **Definition:** `fdset_singleton S E` returns `true` if the $\mathcal{FD}$ set `S` only contains the element `E`. Otherwise, it returns `false`.
- **Reifiable:** Not applied.
- **Examples:**

```
Toy(FD)> fdset_singleton S 1 == B
      { S -> [ (interval 1 1) ],
        B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(FD)> fdset_singleton S 1 == false
      no
      Elapsed time: 0 ms.

Toy(FD)> fdset_singleton [(interval 0 1)] 1 == B
      { B -> false }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_min/1, fdset_max/1, fdset_size/1`

- **Type Declaration:**

  `fdset_min, fdset_max, fdset_size :: fdset → int`

- **Modes:**

  `fdset_min, fdset_max, fdset_size :=: {in → inout}`

- **Definition:** `fdset_min S` and `fdset_max S` returns the lower bound, upper bound, resp., of the $\mathcal{FD}$ set `S`. `fdset_size S` returns the cardinality of the $\mathcal{FD}$ set `S`.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X] 0 1, S == fd_set X, Min == fdset_min S,
         Max == fdset_max S, Size == fdset_size S
      { S -> [ (interval 0 1) ],
        Min -> 0,
```

```
        Max -> 1,
        Size -> 2 }
      { X in 0..1 }
      Elapsed time: 10 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`list_to_fdset/1, fdset_to_list/1`

- **Type Declaration:**

  `list_to_fdset :: [int] → fdset`
  `fdset_to_list :: fdset → [int]`

- **Modes:**

  `list_to_fdset, fdset_to_list :=: {in → inout}`

- **Definition:** `fdset_to_list S` returns the list equivalent to the input $\mathcal{FD}$ set `S`. `list_to_fdset` is the inverse function of `fdset_to_list`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> fdset_to_list (list_to_fdset [0,1]) == L
      { L -> [ 0, 1 ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`range_to_fdset/1, fdset_to_range/1`

- **Type Declaration:**

  `range_to_fdset :: range → fdset`
  `fdset_to_range :: fdset → range`

- **Modes:**

  `range_to_fdset, fdset_to_range :=: {in → inout}`

- **Definition:** `fdset_to_range S` returns the range equivalent to the input $\mathcal{FD}$ set. `range_to_fdset` is the inverse function of `fdset_to_range`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> fdset_to_range [(interval 0 1), (interval 2 3)] == R,
         range_to_fdset R == S
      { R -> (uni (cte 0 1) (cte 2 3)),
        S -> [ (interval 0 3) ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_add_element/2, fdset_del_element/2`

- **Type Declaration:**

  `fdset_add_element, fdset_del_element :: fdset → int → fdset`

- **Modes:**

  `fdset_add_element, fdset_del_element :=: {in → in → inout}`

- **Definition:** `fdset_add_element S E` and `fdset_del_element S E` return the $\mathcal{FD}$ set which is the result of adding (resp., deleting) the element `E` to (resp., from) `S`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> domain [X] 0 1, S == fd_set X, SA == fdset_add_element S 2,
         SD == fdset_del_element SA 0
      { S -> [ (interval 0 1) ],
        SA -> [ (interval 0 2) ],
        SD -> [ (interval 1 2) ] }
      { X in 0..1 }
      Elapsed time: 10 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_intersection/2, fdset_subtract/2, fdset_union/2, fdset_complement/1`

- **Type Declaration:**

  `fdset_intersection, fdset_subtract, fdset_union :: fdset → fdset → fdset`
  `fdset_complement :: fdset → fdset`

- **Modes:**

  `fdset_intersection, fdset_subtract, fdset_union :=:` $\{$`in` $\rightarrow$ `in` $\rightarrow$ `inout`$\}$
  `fdset_complement :=:` $\{$`in` $\rightarrow$ `inout`$\}$

- **Definition:** `fdset_intersection S1 S2`, `fdset_subtract S1 S2`, and `fdset_union S1 S2` return the $\mathcal{FD}$ set which is the result of `S1` $\cap$ `S2` (resp., `S1` $-$ `S2`, and `S1` $\cup$ `S2`). Finally, `fdset_complement S` is the complement (wrt. `inf..sup`) of the set `S`.

- **Reifiable:** Not applied.

- **Examples:**

```
Toy(FD)> S1 == [(interval 0 4)], S2 ==[(interval 3 7)],
         fdset_union (fdset_intersection S1 S2) (fdset_subtract S1 S2) == S1
      { S1 -> [ (interval 0 4) ],
        S2 -> [ (interval 3 7) ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(FD)> fdset_complement [interval 0 1] == S
      { S -> [ (interval inf -1), (interval 2 sup) ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdsets_intersection/1, fdsets_union/1`

- **Type Declaration:**

  `fdsets_intersection, fdsets_union ::` `[fdset]` $\rightarrow$ `fdset`

- **Modes:**

  `fdsets_intersection, fdset_union :=:` $\{$`in` $\rightarrow$ `inout`$\}$

- **Definition:** `fdsets_intersection SL` and `fdsets_union SL` return the $\mathcal{FD}$ set which is the result of the intersection (resp., union) of each $\mathcal{FD}$ set in the list `SL`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> SL == [[(interval 0 4)], [(interval 3 7)]],
         US == fdsets_union SL, IS == fdsets_intersection SL
```

```
      { SL -> [ [ (interval 0 4) ], [ (interval 3 7) ] ],
        US -> [ (interval 0 7) ],
        IS -> [ (interval 3 4) ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_equal/2, fdset_subset/2, fdset_disjoint/2, fdset_intersect/2`

- **Type Declaration:**

  `fdset_equal, fdset_subset, fdset_disjoint, fdset_intersect :: fdset → fdset → bool`

- **Modes:**

  `fdset_equal, fdset_subset, fdset_disjoint, fdset_intersect :=: {in → in → inout}`

- **Definition:** `fdset_equal S1 S2` return `true` if both `S1` and `S2` represent the same $\mathcal{FD}$ set. `fdset_subset S1 S2` return `true` if `S1` is a subset of the $\mathcal{FD}$ set `S2`. `fdset_disjoint S1 S2` return `true` if `S1` and `S2` have no elements in common. `fdset_intersect S1 S2` return `true` if `S1` and `S2` have at least one element in common. If the conditions do not hold, the functions return otherwise `false`.

- **Reifiable:** Not applied.

- **Example:**

```
Toy(FD)> fdset_intersect [interval 0 3] [interval 3 4] == B
      { B -> true }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fdset_member/2, fdset_belongs/2`

- **Type Declaration:**

  `fdset_member :: int → fdset → bool`
  `fdset_belongs :: int → int → bool`

- **Modes:**

  `fdset_member, fdset_belongs :=: {inout → in → inout}`

- **Definition:** `fdset_member X S` return `true` if `X` is a member of the $\mathcal{FD}$ set `S`. `fdset_belongs X Y` return `true` if the domain of `X` is in the domain of `Y`. Otherwise, they return `false`.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X] (-4) 4, domain [Y] 0 1, fdset_belongs X Y == B
      { X -> 0,
        B -> true }
      { Y in 0..1 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { X -> 1,
        B -> true }
      { Y in 0..1 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { B -> false }
      { X in-4..4,
        Y in 0..1 }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.2.11 Statistics Functions

$\mathcal{TOY}$ provides a set of (impure) functions that allow to to display execution statistics about constrained $\mathcal{FD}$ variables and their associated domains during goal solving.

`fd_statistics'/1`

- **Type Declaration:**

  `fd_statistics' :: statistics → int`

- **Modes:**

  `fd_statistics' :=: {inout → inout}`

- **Definition:** `fd_statistics' Key` returns a value `V` if either
  - `V` unifies with the number of constraints created for `Key == constraints` or
  - `V` unifies with the number of times that:
    - a constraint is resumed and `Key == resumptions`,

* a (dis) entailment was detected by a constraint and `Key == entailments`,
* a domain was pruned and `Key == prunings`,
* a backtracking was executed because a domain becomes empty and `Key == backtracks`.

Each value for `Key` stands for a counter which is zeroed whenever it is accessed by `fd_statistics'`.

- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> domain [X,Y,Z] 0 10, X #< Y, Y #< Z, fd_statistics' prunings == V
      { V -> 12 }
      { Z #> Y,
        Y #> X,
        X in 0..8,
        Y in 1..9,
        Z in 2..10 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`fd_statistics/0`

- **Type Declaration:**

  `fd_statistics :: bool`

- **Modes:**

  `fd_statistics :=: {inout}`

- **Definition:** `fd_statistics` always returns `true` and displays a summary of the statistics computed by `fd_statistics/1`. All counters are zeroed.
- **Reifiable:** Not applied.
- **Example:**

```
Toy(FD)> L== [X,Y,Z], domain L 1 4, all_different' L [complete true],
         labeling [] L, fd_statistics
Resumptions: 9
Entailments: 1
Prunings: 9
Backtracks: 0
```

```
Constraints created: 3
      { L -> [ 1, 2, 3 ],
        X -> 1,
        Y -> 2,
        Z -> 3 }
      Elapsed time: 10 ms.
sol.1, more solutions (y/n/d/a) [y]? n
```

3.3 Introductory Programs

The distribution provides the directory `examples/cflpfd` that contains a number of examples of $\mathcal{TOY}$ programs that make use of $\mathcal{FD}$ constraints. Each one of the programs listed in this section, as well as those listed in Section A.7 (page 30) are included in that directory. The program title is annotated with the corresponding file name (with extension `.toy`).

3.3.1 The Length of a List (`haslength.toy`)

Recall the definition of the length of a list in Section 2.6:

```
length       :: [int] -> int
length []     = 0
length [X|Xs] = 1 + length Xs
```

If we pose a goal as the following:

```
Toy(FD)> length L == 2
      { L -> [ _A, _B ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
```

and, then, we request a second answer, we get no termination. This behaviour is due to the fact that there is a pending alternative for the second rule in the definition of `length`, which is a rule that can always be applied provided that its first argument is unbound. Observe that the length of the list is not limited to be positive on the right hand side of this rule. Finite domain constraints can be applied to impose such a limit. Note that this cannot be done without the use of constraints since the function `length` should allow different modes of usage. Then, a new definition using constraints is as follows:

```
include "cflpfd.toy"

hasLength :: [A] -> int -> bool
hasLength [] 0     :- true
hasLength [X|Xs] N :- N #> 0, hasLength Xs (N #- 1)
```

Now, we can submit the following goal and request all the solutions:

```
Toy(FD)> hasLength L 2
      { L -> [ _A, _B ] }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

This definition for `hasLength` accepts all modes of usage.

3.3.2 Send + More = Money (`smm.toy`)

Below, a $\mathcal{TOY}$ program for solving the classical arithmetic puzzle "send more money" is shown. Observe that $\mathcal{TOY}$ allows to use infix constraint operators such as `#>` to build the expression `X #> Y`, which is understood as `#> X Y`. The intended meaning of the functions should be clear from their names, definitions, Tables 3.1 and 3.2, and explanations given in Section 3.2.

```
include "cflpfd.toy"

smm :: [int] -> [labelingType] -> bool
smm [S,E,N,D,M,O,R,Y] Label = true <== domain [S,E,N,D,M,O,R,Y] 0 9,
       S #> 0,
       M #> 0,
       all_different [S,E,N,D,M,O,R,Y],
       sum [S,E,N,D,M,O,R,Y],
       labeling Label [S,E,N,D,M,O,R,Y]

sum :: [int] ->  bool
sum [S,E,N,D,M,O,R,Y] = true <==
                    1000 #* S #+ 100 #* E #+ 10 #* N #+ D
   #+               1000 #* M #+ 100 #* O #+ 10 #* R #+ E
   #=
      10000 #* M #+ 1000 #* O #+ 100 #* N #+ 10 #* E #+ Y
```

This code is included in the file `smm.toy` provided with the distribution. After compiling and loading it (see Section 1.4), we can solve goals as:

```
Toy(FD)> smm L [ff]
      { L -> [ 9, 5, 6, 7, 1, 0, 8, 2 ] }
      Elapsed time: 20 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

3.3.3 N-Queens (queens.toy)

Here we present a $\mathcal{TOY}$ solution for the classical problem of the N-Queens problem: place N queens in a chessboard in such a way that no queen attacks each other. Observe the use of the directive `include` to make use of the function `hasLength`/2. Note also that it is not needed to include the $\mathcal{FD}$ library since `haslength.toy` loads it already.

```
include "cflpfd.toy"
include "miscfd.toy"

queens :: int -> [int] -> [labelingType] -> bool
queens N L Label = true <==
       length L == N,
       domain L 1 N,
       constrain_all L,
       labeling Label L

constrain_all :: [int] ->  bool
constrain_all [] = true
constrain_all [X|Xs] = true <==
       constrain_between X Xs 1,
       constrain_all Xs

constrain_between :: int -> [int] -> int -> bool
constrain_between X [] N = true
constrain_between X [Y|Ys] N = true <==
        no_threat X Y N,
        N1 == N+1,
        constrain_between X Ys N1

no_threat:: int -> int -> int -> bool
no_threat X Y I = true <==
        X #\= Y,
        X #+ I #\= Y,
        X #- I #\= Y
```

Again, if we compile and load this program, we can solve goals as:

```
Toy(FD)> queens 4 L [ff]
      { L -> [ 2, 4, 1, 3 ] }
      Elapsed time: 20 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { L -> [ 3, 1, 4, 2 ] }
      Elapsed time: 10 ms.
sol.2, more solutions (y/n/d/a) [y]?
```

```
no
Elapsed time: 0 ms.
```

3.3.4 A Cryptoarithmetic Problem (alpha.toy)

Alpha is a cryptoarithmetic problem where the numbers 1 - 26 are randomly assigned to the letters of the alphabet. The numbers beside each word are the total of the values assigned to the letters in the word. e.g for LYRE L,Y,R,E might equal 5,9,20 and 13 respectively or any other combination that add up to 47. The problem consists of finding the value of each letter under the following equations:

```
BALLET = 45,
GLEE = 66,
POLKA = 59,
SONG =  61,
CELLO = 43,
JAZZ  = 58,
QUARTET = 50,
SOPRANO = 82,
CONCERT = 74
LYRE = 47,
SAXOPHONE = 134,
THEME = 72,
FLUTE = 30,
OBOE = 53,
SCALE = 51,
VIOLIN = 100,
FUGUE = 50,
OPERA = 65,
SOLO = 37,
WALTZ = 34
```

A $\mathcal{TOY}$ solution, included in the distribution in the directory Examples in the file alpha.toy, is shown below:

```
include "cflpfd.toy"

alpha :: [labelingType] -> [int] -> bool
alpha Label LD = true <==
  LD == [A,B,C,_D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z],
  domain LD 1 26,
  all_different LD,
  B #+ A #+ L #+ L #+ E #+ T                #= 45,
  C #+ E #+ L #+ L #+ O                     #= 43,
  C #+ O #+ N #+ C #+ E #+ R #+ T           #= 74,
```

```
  F #+ L #+ U #+ T #+ E                    #= 30,
  F #+ U #+ G #+ U #+ E                    #= 50,
  G #+ L #+ E #+ E                         #= 66,
  J #+ A #+ Z #+ Z                         #= 58,
  L #+ Y #+ R #+ E                         #= 47,
  O #+ B #+ O #+ E                         #= 53,
  O #+ P #+ E #+ R #+ A                    #= 65,
  P #+ O #+ L #+ K #+ A                    #= 59,
  Q #+ U #+ A #+ R #+ T #+ E #+ T          #= 50,
  S #+ A #+ X #+ O #+ P #+ H #+ O #+ N #+ E #= 134,
  S #+ C #+ A #+ L #+ E                    #= 51,
  S #+ O #+ L #+ O                         #= 37,
  S #+ O #+ N #+ G                         #= 61,
  S #+ O #+ P #+ R #+ A #+ N #+ O          #= 82,
  T #+ H #+ E #+ M #+ E                    #= 72,
  V #+ I #+ O #+ L #+ I #+ N               #= 100,
  W #+ A #+ L #+ T #+ Z                    #= 34,
  labeling Label LD
```

Solving at the command prompt is shown below:

```
Toy(FD)> alpha [ff] L
      { L ->  [ 5, 13, 9, 16, 20, 4, 24, 21, 25, 17, 23, 2, 8, 12,
                10, 19, 7, 11, 15, 3, 1, 26, 6, 22, 14, 18 ] }
      Elapsed time: 9735 ms.
```

3.3.5 Magic Series (`magicser.toy`)

We present another simple (and well-known) example, the magic series problem [37]. Let $S = (s_0, s_1, \ldots, s_{N-1})$ be a non-empty finite serial of non-negative integers. As convention, we number its elements from 0. The serial S is said N-Magic if and only if there are s_i occurrences of i in S, for all i in $\{1, \ldots, N-1\}$.
Below we show a possible $\mathcal{TOY}$ solution, included in the distribution in the directory Examples in the file `magicseq.toy`, to this problem.

```
include "cflpfd.toy"
include "miscfd.toy"

magic :: int -> [int] -> [labelingType] -> bool
magic N L Label = true <==
    hasLength L N,
    domain L 0 (N-1),
    constrain L  L  0  Cs,
    sum L (#=) N,                  % redundant #1
    scalar_product Cs  L  (#=) N,  % redundant #2
```

```
    labeling Label L

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain []  A  B  [] = true
constrain [X|Xs]  L  I  [I|S2] = true <==
    count I  L  (#=)  X,
    constrain Xs  L  (I+1)  S2
```

Below, we show an example goal:

```
Toy(FD)> magic 10 L []
      { L -> [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ] }
      Elapsed time: 31 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Chapter 4

Cooperation of Solvers

In this chapter, we present the cooperation of solvers. Cooperation is based on the communication between the $\mathcal{FD}$ and $\mathcal{R}$ solvers by means of special communication constraint called *bridge*. Bridges are used for two purposes, namely *binding* and *propagation*. Solver cooperation is allowed with binding alone or both binding and propagation.

4.1 Introduction

$CFLP$ goal solving takes care of evaluating calls to program defined functions by means of lazy narrowing, and decomposing hybrid constraints by introducing new local variables. Eventually, pure $\mathcal{FD}$ and $\mathcal{R}$ constraints arise, which must be submitted to the respective solvers.
Cooperation of solvers is based on the communication between the $\mathcal{FD}$ and $\mathcal{R}$ solvers by means of special communication constraints called *bridges*. A bridge `u #== v` constrains `u::int` and `v::real` to take the same integer value. Bridges are used for two purposes: *Binding* and *propagation*. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end of the bridge becomes a numeric value. Propagation is a more complex operation which takes place whenever a pure constraint is submitted to the $\mathcal{FD}$ or $\mathcal{R}$ solver. At that moment, propagation rules relying on the available bridges are used for building a mate constraint which is submitted to the mate solver (think of $\mathcal{R}$ as the mate of $\mathcal{FD}$ and viceversa). Propagation enables each of the two solvers to take advantage of the computations performed by the other. In order to maximize the opportunities for propagation, the $CFLP$ goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules.

4.1.1 Efficiency

Example 4.5.1 compares the timing results for executing the same goals with only binding and binding and propagation, shown that the propagation of mate constraints dramatically cuts the search space, thus leading to significant speedups in execution time, as it is shown in Table 4.3.

4.1.2 Libraries necessaries for Cooperation

In order to use cooperation is necessary to activate finite domain and real solvers, it is done by means of the commands `/cflpfd` and `/cflpr` as explained in Section 1.5.5. With both solvers activated is possible compile programs that use the communication constraint `bridge`, this allows only binding. Additionally, constraints can be propagate to other solver, a $\mathcal{FD}$ constraint can be propagate to $\mathcal{R}$ solver and viceversa. Propagation is enabled by means of the command `/prop` and disabled with `/noprop`.

4.2 The Constraint Cooperation *Bridge*

`Bridge` is a constraint of the form `u #== v` where `u` and `v` are variables with integer and real types respectively. Furthermore `u` and `v` are constrained to take the same integer value.

`#==` / 2

- **Type Declaration:**

  `#== :: int → float → bool`

- **Modes**

  `#== :: inout → inout → inout`

- **Reifiable:** Yes

- **Examples:**

```
Toy(R+FD)> X #== RX, domain [X] 2 4, RX < 3.4
      { X #== RX,
        RX<3.4,
        X in 2..4 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

4.3 Binding

As cooperation mechanism, binding instantiates a variable occurring at one end of a bridge whenever the other end of the bridge becomes a numeric value. For example, next goals unify `RX` with the real value 2 (first goal) and `X` with the integer value 2 (second goal).

```
Toy(R+FD)> X #== RX, X == 2
      { X -> 2,
        RX -> 2 }
```

```
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.

Toy(R+FD)> X #== RX, RX == 2
      { X -> 2,
        RX -> 2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

4.4 Propagation

Propagation takes place whenever a constraint is submitted to the $\mathcal{FD}$ or $\mathcal{R}$ solver. At that moment, propagation rules relying on the available bridges are used for building a mate constraints which are submitted to the mate solver (think of $\mathcal{R}$ as the mate of $\mathcal{FD}$ and viceversa). Propagation enables each of the two solvers to take advantage of the computations performed by the other. In order to maximize the opportunities for propagation, the goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules listed in Tables 4.1 and 4.2. These tables use the notation t_1 `#+` $t_2 \rightarrow! \ t_3$ in order to illustrate the process `flattering` of goal solving procedure. For example, the goal $RX\# + 2\# < 3$ is flattered to constraints $RX\#+2 \rightarrow! Z$ and $Z\# < 3$ where Z is a new variable. When constraints are totaly flatted then they are submitted to solver, and in this moment take place propagation, it is explained in [8].

4.4.1 Propagation from $\mathcal{FD}$ to $\mathcal{R}$

In order to propagate a constraint from $\mathcal{FD}$ to $\mathcal{R}$ we follow two steps. First, is checked if each variable has its corresponding bridge, if the variable has not bridge then a bridge is created using a new real variable. From $\mathcal{FD}$ to $\mathcal{R}$ always is possible to create bridges since all integer value is a real value. Afterwards new real constraints are building and submitted to real solver. Constraints that support propagation from $\mathcal{FD}$ to $\mathcal{R}$ are listed in Table 4.1 and explained below.

`domain` $[X_1, \ldots, X_n]$ a b

- **Creation of Bridges:** If X_i has no bridge then $X_i\#== RX_i$ (1≤i≤n) is created with RX_i new.
- **Propagation Constraint:** Afterwards the constraints $a \leq RX_i$, $RX_i \leq b$ (1≤i≤n) are built as mate constraints and submitted to the real solver.
- **Example:** In next goal a new bridge is created for variable `Y`.

Constraint	*Bridges Created*	*Constraints Created*
domain $[X_1, \ldots, X_n]$ a b	$\{X_i\#== RX_i \mid 1\leq i\leq n,\ X_i$ has no bridge, RX_i new$\}$	$\{a \leq RX_i,\ RX_i \leq b \mid 1\leq i\leq n\ \}$
belongs X $[a_1, \ldots, a_n]$	$\{\ X\#== RX \mid X$ has no bridge, RX new$\}$	$\{min(a_1, \ldots, a_n)\leq RX,$ $RX\leq max(a_1, \ldots, a_n)\ \}$
$t_1\#<t_2$ (analogously $\#<=,\#>,\#=>,\#=)$	either t_i is an integer constant, or else is a variable X_i. $\{X_i\#== RX_i \mid 1\leq i\leq 2$, if t_i is a variable X_i with no bridge, RX_i new$\}$	$\{t_1^{\mathcal{R}} < t_2^{\mathcal{R}} \mid$ For $1\leq i\leq 2$: Either t_i is an integer constant n and $t_i^{\mathcal{R}}$ is n, or else t_i is a variable X_i, and $t_i^{\mathcal{R}}$ is $RX_i\}$
$t_1\#+t_2 \rightarrow! t_3$ (analogously $\#-$, $\#*$)	$\{X_i\#== RX_i \mid 1\leq i\leq 3$, t_i is a variable X_i with no bridge, RX_i new$\}$	$\{t_1^{\mathcal{R}} + t_2^{\mathcal{R}} \rightarrow! t_3^{\mathcal{R}} \mid$ For $1\leq i\leq 3$: $t_i^{\mathcal{R}}$ is determined as in the previous case$\}$

Table 4.1: Propagations from $\mathcal{FD}$ to $\mathcal{R}$

```
Toy(R+FD)> X #== RX, domain [3,X,Y] 2 5
      { X #== RX,
        Y #== _D,
        RX=<5.0,
        _D=<5.0,
        RX>=2.0,
        _D>=2.0,
        X in 2..5,
        Y in 2..5 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The same goal without propagation is

```
Toy(R+FD)> X #== RX, domain [3,X,Y] 2 5
      { X #== RX,
        X in 2..5,
        Y in 2..5 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

`belongs` X $[a_1, \ldots, a_n]$

- **Creation of Bridges:** If X has no bridge then $X\#== RX$ is created with RX new.

- **Propagation Constraint:** Afterwards the constraints $min(a_1,..a_n) \leq RX$, $RX \leq max(a_1,..a_n)$ are buildt as mate constraints and submitted to the real solver.
- **Example:**

```
Toy(R+FD)> belongs X [1,5,3]
      { X #== _B,
        _B=<5.0,
        _B>=1.0,
        X in{1}\/{3}\/{5} }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

The same goal without propagation is

```
Toy(R+FD)> belongs X [1,5,3]
      { X in{1}\/{3}\/{5} }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

t_1 `#<` t_2 (analogously `#<=`,`#>`,`#>=`)

t_1 and t_2 can be either integer constants or else a integer variables.

- **Creation of Bridges:** If t_1 and/or t_2 are variables X and/or Y and they have no bridges then $X\#== RX$ and/or $Y\#== RY$ are created with RX and/or RY new.
- **Propagation Constraint:** Afterwards the constraint $t_1^{\mathcal{R}} \# < t_2^{\mathcal{R}}$ is buildt as mate constraint and submitted to the real solver where $t_1^{\mathcal{R}}$ and/or $t_2^{\mathcal{R}}$ are either the same constant that t_1 and/or t_2, or else they are the variables RX and/or RY.
- **Example:**

```
Toy(R+FD)> X #==RX, X #< 5
      { X #== RX,
        RX<5.0,
        X in inf..4 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
```

```
      no
      Elapsed time: 0 ms.
Toy(R+FD)> X #==RX, 5 #< X
      { X #== RX,
        RX>5.0,
        X in 6..sup }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
Toy(R+FD)> X #==RX, X #< Y
      { X #== RX,
        Y #== _D,
        RX-_D<-0.0,
        Y #> X }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In the last goal, a new bridge is created for the variable Y.

t_1 `#+` t_2 $\rightarrow!$ t_3 (analogously `#-`,`#*`)

t_1 `#+` t_2 $\rightarrow!$ t_3 was explained in section 4.4. t_1 and t_2 can be either integer constants or else a integer variables, but one of this must be a variable so that the propagation is possible.

- **Creation of Bridges:** If t_1 and/or t_2 are variables X and Y, and they have no bridges then $X\#== RX$ and/or $Y\#== RY$ are created with RX and/or RX new. If t_1 or t_2 are variables then t_3 is a new variable Z and a bridge $Z\# == RZ$ is buildt for Z with RZ also new.
- **Propagation Constraint:** Afterwards the constraint $RX + RY \rightarrow! RZ$ is buildt as mate constraint.
- **Example:**

```
Toy(R+FD)>  X #==RX, Y #==RY, X #+ Y #< 2
      { X #== RX,
        Y #== RY,
        _E #== _F,
        _F<2.0,
        RY== -(RX)+_F,
```

```
        X #+ Y #= _E,
        _E in inf..1 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
Toy(R+FD)> X #==RX, 5 #+ X #< 10
      { X #== RX,
        _C #== _D,
        RX<5.0,
        _D==5.0+RX,
        5 #+ X #= _C,
        X in inf..4,
        _C in inf..9 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In the first goal, new variable `_E` links with result of `X #+ Y`. New bridge `_E #== _F` is created with the new real variable `_F`, therefore a real constraint is buildt `(_F == RX + RY)`, afterwards the constraint `_F < 2.0` is buildt and submitted to the real solver. Second goal is similar.

4.4.2 Propagation from $\mathcal{R}$ to $\mathcal{FD}$

In order to propagate a constraint from $\mathcal{R}$ to $\mathcal{FD}$ is necessary that each variable has its corresponding bridge, if there is a variable without bridge then it not possible to propagate. Now we can not created bridges because a real value is not a integer value. Despite, sum of two real variables with bridges is a new variable for the flattering process explained previously, a new bridge will be created for this new variable.
Constraints that support propagation from $\mathcal{R}$ to $\mathcal{FD}$ are listed in Table 4.2 and explained below.

RX `<` RY (analogously RX `<=` RY, RX `>` RY, RX `>=` RY)

- **Checking Bridges:** $X\#== RX$ and $Y\#== RY$
- **Propagation Constraint:** If checking bridges is successful then the constraint $X\# < Y$ is buildt as mate constraint and submitted to the finite domain solver.
- **Example:**

```
Toy(R+FD)>  X #==RX, Y #==RY,  RX < RY
      { X #== RX,
```

Constraint	*Bridges Tested*	*Constraint Created*
$RX<RY$	$X\#==RX$ and $Y\#==RY$	$X\#<Y$
$RX<a$	$X\# == RX$	$X\#<\lceil a \rceil$ with $a\in\mathbb{R}$
$a<RY$	$Y\# == RY$	$\lfloor a \rfloor\# < Y$ with $a\in\mathbb{R}$
$RX<= RY$	$X\#==RX$ and $Y\#==RY$	$X\# <= Y$
$RX<=a$	$X\#==RX$	$X\#<=\lfloor a \rfloor with\ a\in\mathbb{R}$
$a <= RY$	$Y\# == RY$	$\lceil a \rceil\#<=Y with\ a\in\mathbb{R}$
$RX>RY$	$X\#==RX$ and $Y\#==RY$	$X\#>Y$
$RX>a$	$X\# == RX$	$X\#>\lfloor a \rfloor$ with $a\in\mathbb{R}$
$a>RY$	$Y\# == RY$	$\lceil a \rceil\# > Y$ with $a\in\mathbb{R}$
$RX>= RY$	$X\#==RX$ and $Y\#==RY$	$X\# >= Y$
$RX>=a$	$X\#==RX$	$X\#>=\lceil a \rceil with\ a\in\mathbb{R}$
$a >= RY$	$Y\# == RY$	$\lfloor a \rfloor\#>=Y with\ a\in\mathbb{R}$
$t_1+t_2\rightarrow!t_3$ (analogously for $-$, $*$)	t_3 is a new variable RX_3 with no bridge, $X_3\#== RX_3$ is created with X_3 new. For $1\leq i\leq 2$ t_i is either an integer constant or a variable RX_i with bridge $X_i\#== RX_i$	$t_1^{\mathcal{FD}} \# + t_2^{\mathcal{FD}} \rightarrow! t_3^{\mathcal{FD}}$ .For $1\leq i\leq 2$ either t_i is an integer constant n and $t_i^{\mathcal{FD}}$ is n, or else t_i is a variable RX_i where $X_i\# == RX_i$, and $t_i^{\mathcal{FD}}$ is X_i
t_1 / t_2 $\rightarrow!$ t_3	as previous case	$\{t_2^{\mathcal{FD}} \# * t_3^{\mathcal{FD}} \rightarrow! t_1^{\mathcal{FD}}$ | For $1\leq i\leq 3$ is determined as in the previous case$\}$

Table 4.2: Propagations from $\mathcal{R}$ to $\mathcal{FD}$

```
        Y #== RY,
        RX-RY<-0.0,
        Y #> X }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Goal below not propagates the constraint $RX < RY$ since there is not bride for the real variable RX.

```
Toy(R+FD)> X #==RX, RX < RY
      { X #== RX,
        RY-RX>0.0 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
```

```
      Elapsed time: 0 ms.
```

RX `<` a (analogously w.r.t. Table 4.2 RX `<=` a, RX `>` a, RX `>=` a)

- **Checking Bridges:** $X\#{=}{=} RX$.
- **Propagation Constraint:** Afterwards the constraints $X < \lceil a \rceil$ is buildt as mate constraints and submitted to the finite domain solver.
- **Example:**

```
Toy(R+FD)>  X #==RX, RX < 2.3
      { X #== RX,
        RX<2.3,
        X in inf..2 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In the previous example the constraint $X < \lceil 2.3 \rceil$ is submitted to the finite domain solver and therefore the value of X is between `inf` and `2` (`X in inf..2`).

a `<` RX (analogously w.r.t. Table 4.2 a `<=` RX, a `>` RX, a `>=` RX)

- **Checking Bridges:** $X\#{=}{=} RX$.
- **Propagation Constraint:** Afterwards the constraints $\lfloor a \rfloor < X$ is buildt as mate constraints and submitted to the finite domain solver.
- **Example:**

```
Toy(R+FD)> X #==RX, 2.3 < RX
      { X #== RX,
        RX>2.3,
        X in 3..sup }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

In the previous example the constraint $\lfloor 2.3 \rfloor < X$ is submitted to the finite domain solver and therefore the value of X is between `3` and `sup` (`X in 3..sup`).

t_1 `+` $t_2 \rightarrow !t_3$ (analogously `#-`,`#*`)

t_1 `+` $t_2 \rightarrow !$ t_3 was explained in section 4.4. t_1 and t_2 can be either constants or else variables, but one of this must be a variable so that the propagation is possible.

- **Checking Bridges:** If t_1 and/or t_2 are variables RX and RY, they must have bridges in order to propagate, therefore $X\# == RX$ and/or $RX\# == RY$ are checked.
- **Creation of Bridges:** If checking is successful then t_3 is a new variable RZ and a bridge $Z\# == RZ$ is building for RZ with Z new.
- **Propagation Constraint:** Afterwards the constraint $X\# + Y \rightarrow !Z$ is buildt as mate constraint.
- **Example:**

```
Toy(R+FD)> X #==Xr, Y#==Yr, Xr + Yr < 7
      { X #== Xr,
        Y #== Yr,
        _E #== _F,
        _F<7.0,
        Yr==_F-Xr,
        X #+ Y #= _E,
        _E in inf..7 }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

Previous goal matches `_F` with `RZ` and `_E` with `Z`.

4.5 Introductory Programs

The directory `examples/cooperation` provides by distribution contains examples of $\mathcal{TOY}$ programs that make use of $\mathcal{FD}$ and R constraints, bridges constraints and cooperation between solvers. The program title is annotated with the corresponding file name.

4.5.1 Intersection of a discrete grid and a continuous region (`bothIn.toy`)

We consider a generic program written in $\mathcal{TOY}$ which solves the problem of searching for a $2D$ point lying in the intersection of a discrete grid and a continuous region.

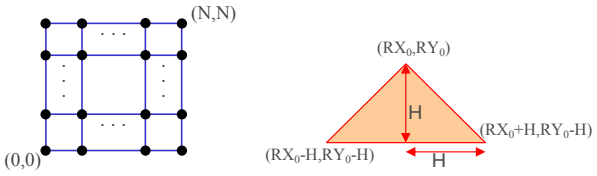


Figure 4.1: Discrete grid and a continuous region

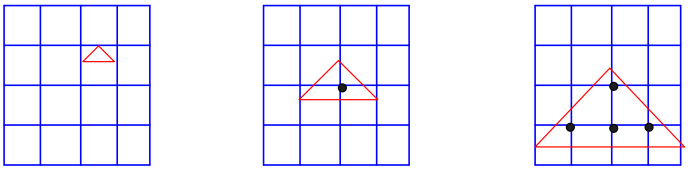


Figure 4.2: Possible intersections

Sizes of square grid and triangular region is parametric.
Both grids and regions are represented as Boolean functions. They can be passed as parameters because our programming framework supports higher-order programming features.

```
type dPoint = (int, int)
type cPoint = (real, real)

type setOf A = (A -> bool)
isIn :: (setOf A) -> A -> bool
isIn Set Element = Set Element

type grid = setOf dPoint

square:: int -> grid square N (X, Y) :- domain [X, Y] 0 N

type region = setOf cPoint

triangle :: cPoint -> real -> region
triangle (RX0, RY0) H (RX, RY):- RY>=RY0-H, RY-RX<=RY0-RX0, RY+RX<=RY0+RX0

bothIn:: region -> grid -> dPoint -> bool
bothIn Region Grid (X, Y) :- X#==RX, Y#==RY, isIn Region (RX,RY),
                             isIn Grid (X,Y), labeling [] [X,Y]
```

We build an isosceles triangles from a given upper vertex (RX_0, RY_0) and a given height H. The three vertices are (RX_0, RY_0), $(RX_0 - H, RY_0 - H)$, $(RX_0 + H, RY_0 - H)$, and the region inside the triangle is enclosed by the lines $RY = RY_0 - H$, $RY - RX = RY_0 - RX_0$ and $RY + RX = RY_0 + RX_0$ and characterized by the conjuntion of the three linear inequalities:

$RY \geq RY_0 - H$, $RY - RX \leq RY_0 - RX_0$ and $RY + RX \leq RY_0 + RX_0$. This explains the real arithmetic constraints in the `triangle` predicate.
As an example of goal solving for this program, we consider three *goals* computing the intersection of this fixed square grid with three different triangular regions:

- ```
  Toy(R+FD)> bothIn (triangle (2.5,3) 0.5) (square 4) (X,Y)
        no
        Elapsed time: 0 ms.
  ```

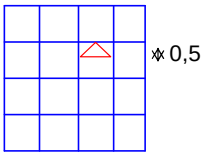


Figure 4.3: No solutions

This goal fails.

- ```
  Toy(R+FD)> bothIn (triangle (2,2.5) 1) (square 4) (X,Y)
        { X -> 2,
          Y -> 2 }
        Elapsed time: 15 ms.
  sol.1, more solutions (y/n/d/a) [y]?
        no
        Elapsed time: 0 ms.
  ```

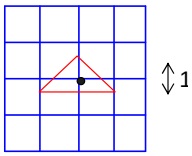


Figure 4.4: One solution

This goal computes one solution for `(X,Y)`, corresponding to the point $(2, 2)$.

- ```
  Toy(R+FD)> bothIn (triangle (2,2.5) 2) (square 4) (X,Y)
        { X -> 1,
          Y -> 1 }
        Elapsed time: 16 ms.
  sol.1, more solutions (y/n/d/a) [y]?
        { X -> 2,
          Y -> 1 }
        Elapsed time: 0 ms.
  ```
  ```

```
sol.2, more solutions (y/n/d/a) [y]?
      { X -> 2,
        Y -> 2 }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      { X -> 3,
        Y -> 1 }
      Elapsed time: 0 ms.
sol.4, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

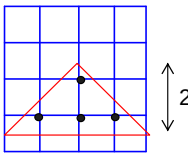


Figure 4.5: Four solution

This goal computes four solutions for `(X,Y)`, corresponding to the points $(2,2)$, $(1,1)$, $(2,1)$ and $(3,1)$.

Table 4.3 compares the timing results for executing below goals. The first column indicates the height of triangle, second indicate the vertice (RX_0,RY_0), third indicate the size of the square grid. The next columns show running times (in milliseconds) in the form (t_B/t_{BP}), where t_B stands for the system using bridges for binding alone and t_{BP} for the system using bridges also for propagation. These last columns are headed with a number i which refers to the i-th solution found, and the last column with numbers stands for the time needed to determine that there are no more solutions. In this simple example we see that the finite domain search space has been hugely cut by the propagations from $\mathcal{R}$ to $\mathcal{FD}$. Finite domain solvers are not powerful enough to cut the search space in such an efficient way as simplex methods do for linear real constraints.
The cooperation of a $\mathcal{FD}$ solver and a $\mathcal{R}$ solver via communication bridges can lead to great reductions of the $\mathcal{FD}$ search space, manifesting as significant speedups of the execution time.

4.5.2 Distribution of raw material between suppliers and consumers (`distribution.toy`)

A company has six suppliers of raw material, three of them supply integer quantities (discreet suppliers) and the other ones three supply real quantities (continuous suppliers). The problem is to satisfy the demand of raw material minimizing the cost with the following data:

H	(RX_0, RY_0)	N	1	2	3	4	5
0.5	(20000.5,20001)	40000	1828/0	-	-	-	-
	(2000000.5,2000001)	4000000	179000/0	-	-	-	-
1	(20000,20000.5)	40000	1125/0	2172/0	-	-	-
	(2000000,2000000.5)	4000000	111201/0	215156/0	-	-	-
2	(20000,20000.5)	40000	1125/0	1485/0	0/0	1500/0	2203/0
	(2000000,2000000.5)	4000000	111329/0	147406/0	0/0	147453/0	216156/0

Table 4.3: Performance Results

- D1, D2, and D3 are the quantities of raw material provided by the discreet suppliers. They can provide until a maximum of 100.
- R1, R2, and R3 are the quantities of raw material provided by the continuous suppliers. They can provide until a maximum of 100.
- The cost of the raw material is 3 plus its quantity (CD1 #= 3 #+ D1), 2 plus its quantity (CD2 #= 2 #+ D2) and 2 multiply its quantity (CD3 #= 2 #* D3) for the discreet suppliers and (CR1 == 2 * R1), (CR2 == 2 + 1.5 * R2), and (CR3 == 3 + 1.2 * R3) for the continuous suppliers.
- The demand to satisfy is of X units of weight.

Below, a $\mathcal{TOY}$ program that solve problem is shown.

```
include "cflpfd.toy"

distribution :: int->int->int->int->int->int->int->
                real->real->real->real->real->real->real->real->real
distribution D1 D2 D3 CD1 CD2 CD3 CD R1 R2 R3 CR1 CR2 CR3 CR X = C <==

   % new bridges are created for discreet quantities of raw material.
   D1 #== D1r, D2 #== D2r, D3 #== D3r,

   % quantities integer of raw material.
   domain [D1,D2,D3] 0 100,

   % quantities real of raw material.
   R1 >= 0.0, R2 >= 0.0, R3 >= 0.0, R1 <=100, R2 <=100, R3 <=100,

   % Cost of the raw material provided by the discreet suppliers.
   CD1 == 3 #+ D1, CD2 == 2#+ D2, CD3 == 2#* D3,
   CD #== CDr, CD == CD1 #+ CD2 #+ CD3,

   % Cost of the raw material provided by the continuous suppliers.
```

```
    CR1 == 2 * R1, CR2 == 2 + 1.5 * R2, CR3 == 3 + 1.2 * R3,
    CR == CR1 + CR2 + CR3,

    % E is necessary to avoid rounding errors.
    -1 <= E, E <= 1, C == CDr + CR + E,

    % X is units of weight to satisfy.
    X == D1r + D2r + D3r + R1 + R2 + R3,

    minimize C == C, labeling [mini] [D1, D2,D3]
```

If we compile and load this program, we can submit a goal which the demand to satisfy is of 9 units of weight. As the propagation has not been activated this goal is solved using only binding.

```
Toy(R+FD)> distribution D1 D2 D3 CD1 CD2 CD3 CD R1 R2 R3 CR1 CR2 CR3 CR 9 == C
SYSTEM ERROR: system_error(Non-linear constraints or unbounded problem.)
```

The system shows us that is not able to solve it. However if we enable propagation with the command `/prop` then is possible to obtain a solution. This goal returns ten answers, only two are shown.

```
Toy(R+FD)> distribution D1 D2 D3 CD1 CD2 CD3 CD R1 R2 R3 CR1 CR2 CR3
CR 9 == C
      { D1 -> 0,
        D2 -> 9,
        D3 -> 0,
        CD1 -> 3,
        CD2 -> 11,
        CD3 -> 0,
        CD -> 14,
        R1 -> 0,
        R2 -> 0,
        R3 -> 0,
        CR1 -> 0,
        CR2 -> 2,
        CR3 -> 3,
        CR -> 5,
        C -> 17.999999999999996 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { D1 -> 9,
        D2 -> 0,
        D3 -> 0,
        CD1 -> 12,
```

```
        CD2 -> 2,
        CD3 -> 0,
        CD -> 14,
        R1 -> 0,
        R2 -> 0,
        R3 -> 0,
        CR1 -> 0,
        CR2 -> 2,
        CR3 -> 3,
        CR -> 5,
        C -> 17.999999999999996 }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?n
```

Chapter 5

Input/Output

$\mathcal{TOY}$ provides a monadic I/O approach similar to that used in the purely functional language Haskell [16, 3]. To this end, I/O operations can only be used in deterministic computations[1]. I/O actions are expected to transform the outside world and to yield a value of some type `A` upon termination. Therefore, they are represented as values of the special type `io A`, which in fact can be understood as a type synonym:

```
data io A = world -> (A,world)
```

The type `world` is as an abstract dataype whose representation is implementation-dependent. Therefore, the outer world is not accessible directly, and only I/O operations can modify it. The monadic style ensures that in any step of a computation there is a unique version of the world. Output operations have type `io unit`, because they yield an uninteresting value of type `unit` (a predefined type which contains one single value). Input operations which yield a value of some interesting type $\tau \neq$ `unit` have type `io` τ. The two monadic binding functions used to sequence input/output operations are >> and >>= which will be carefully described in Section 5.2. As an alternative to writing combinations of >>=, $\mathcal{TOY}$ offers the possibility of using the *do notation*, whose description can be found in Section 5.3.
In addition to the standard input/output, $\mathcal{TOY}$ provides also text files (see Section 5.4) and a declarative *Graphical User Interface* (GUI), which has been developed following the ideas presented in [15]. More concretely, the graphic input/output has been implemented by means of a GUI system module, based on Tcl/Tk [30] (see Section 5.5).
Finally, in Section 5.6 we describe *list comprehensions*, a powerful notation to write expressions of type list which is available also in Haskell and some other functional (logic) languages. List comprehensions are a useful device for computing several solutions while avoiding non-deterministic search, in a context where monadic I/O is needed. For this reason, they have been added to $\mathcal{TOY}$ during the development of the monadic I/O facilities, although they are also useful for other purposes.

[1] In fact, they can be used in any part of a program, but a correct behaviour only is ensured when there is no non-deterministic search.

5.1 Standard Input/Output Functions

Standard *output* functions write to the standard output device, which is usually the user's terminal. The standard output functions provided by $\mathcal{TOY}$ are the following:

- `putChar :: char -> io unit`
  which prints a character.
- `done :: io unit`
  which does nothing.
- `putStr :: string -> io unit`
  which prints a string.
- `putStrLn :: string -> io unit`
  which is similar to `putStr`, except that it prints a newline character after printing the string.

Standard *input* functions read input from the standard input device, which is usually the user's keyboard. The standard input functions provided by $\mathcal{TOY}$ are the following:

- `getChar :: io char`
  which returns the first character read from the standard input.
- `return :: A -> io A`
  which is a generalization of `done`, and does not change the world, returning a value of type `io A`.
- `getLine :: io string`
  which reads a line of text from the standard input.

5.2 Sequencing Input/Output Functions

As in Haskell [16] or Curry [14], the way of combining sequences of input/output operations is done by means of the following two binding functions:

```
(>>)  :: io A -> io B -> io B
(>>=) :: io A -> (A -> io B) -> io B
```

Supposing that `p` and `q` are input/output operations, then `p >> q` is a new operation that, when performed, first does `p` (ignoring the value returned) and then does `q`. For instance, the built-in function `putStr` in Section 5.1, could be defined as follows:

```
putStr       :: string -> io unit
putStr []     = done
putStr [C|Cs] = putChar C >> putStr Cs
```

Similarly, the built-in function `putStrLn` in Section 5.1 can be defined using `(>>)` and the function `putStr`; it prints a newline character after printing the string.

```
putStrLn    :: string -> io unit
putStrLn Cs  = putStr Cs >> putChar '\n'
```

Note that the use of the operator `(>>)` is only useful when the value returned by the first argument is not needed in the computation (more precisely, it is not needed by the second argument). If the value of the first action should be taken into account by the second action, it is necessary to use the more general operator `(>>=)`. The combination `p >>= q` is an action that, when performed, first does `p`, returning a value `x` of type `A`; after this, the action `q x` is executed, returning a final value `y` of type `B`. In fact, `(>>)` can be easily defined in terms of `(>>=)`:

```
(>>)   :: io A -> io B -> io B
P >> Q  = P >>= constant Q

constant     :: B -> A -> B
constant Y X  = Y
```

As a more practical example, the built-in function `getLine` in Section 5.1 can be implemented as follows:

```
getLine :: io string
getLine = getChar >>= follows_1

follows_1 :: char -> io string
follows_1 C = if C=='\n' then return [] else getLine >>= follows_2 C

follows_2 :: char -> string -> io string
follows_2 C Cs = return [C|Cs]
```

As a second practical example, here we have a function which reads three characters from the keyboard and prints them on the screen.

```
read_3_chars :: io unit
read_3_chars = getChar >>= continue_1

continue_1 :: char -> io unit
continue_1 C1 = getChar >>= continue_2 C1

continue_2 :: char -> char -> io unit
continue_2 C1 C2 = getChar >>= continue_3 C1 C2

continue_3 :: char -> char -> char -> io unit
continue_3 C1 C2 C3 = (putChar C1 >> putChar C2) >> putChar C3
```

Note that the definition of the "very simple" function `read_3_chars` requires three auxiliary functions. In order to facilitate the use of input/output, we have incorporated the `do` notation, described in next section. This facility is available also in Haskell and Curry.

5.3 Do Notation

Do notation provides a more comfortable syntax for using input/output in $\mathcal{TOY}$, avoiding the need of auxiliary functions. Using the construction `do` instead of sequences of the operator `(>>=)` leads to a programming style which achieves a declarative specification of I/O actions, while rendering their intended imperative effects intuitively clear. For example, the `do` notation allows to write more intuitive definitions for the functions `getLine` and `read_3_chars` in the previous section:

```
getLine :: io string
getLine = do {C <- getChar ;
              if C == '\n' then return []
                 else do {Cs <- getLine ;
                          return [C|Cs]}}

read_3_chars :: io unit
read_3_chars = do {C1 <- getChar ;
                   C2 <- getChar ;
                   C3 <- getChar ;
                   (putChar C1 >> putChar C2) >> putChar C3}
```

In general, a `do` construction has the form:

```
do  { C1 <- p1 ;
      C2 <- p2 ;
      ⋮
      Ck <- pk ;
      r          }
```

where $\mathtt{C}_i$, $1 \leq i \leq k$, are pairwise distinct variables and $\mathtt{p}_i$ are expressions of type `io` τ_i, for some types τ_i. Each $\mathtt{C}_i$ only can be used after its definition, i.e., $\mathtt{C}_i$ can appear in $\mathtt{p}_j$ only if $j > i$. If $\mathtt{p}_i$ is an expression of type `io` τ_i, then the type of $\mathtt{C}_i$ is τ_i. In the particular case $\tau_i =$ `unit`, $\mathtt{C}_i$ does not need to be used by any other $\mathtt{p}_j$, and $\mathtt{C}_i$ `<-` $\mathtt{p}_i$ can be abbreviated to $\mathtt{p}_i$. Finally, `r` is an expression of type `io` τ, which is also the type of the entire `do` expression.
The above do construction has also a declarative meaning, because it is equivalent to the following combinations of the operator `(>>=)`:

```
p1 >>= cont1
cont1 C1 = p2 >>= cont2 C1
cont2 C1 C2 = p3 >>= cont3 C1 C2
⋮
contk−1 C1 C2 ...Ck−1 = pk >>= contk C1 C2 ...Ck−1
contk C1 C2 ...Ck = r
```

5.4 Files

File names are values of type `path`, where:

```
type path = string
```

There are two standard functions which operate on text files. These functions are:

```
readFile  :: path -> io string
writeFile :: path -> string -> io unit
```

Function `readFile` reads a file (named by a string) and returns the contents of the file as a string. The file is read lazily on demand; that is, the effect of the action `readFile FileName` only opens the file named `FileName`. A character in the file only is read if it is needed in following computations. As an example, consider the function:

```
only_one_char :: io unit
only_one_char = do {putStr "Name of the file: " ;
                    Name <- getLine ;
                    Cs <- readFile Name ;
                    putStrLn (head Cs)}
```

The execution of the expression `only_one_char` has the following effect: The text file named `Name` is opened when the action `readFile Name` is executed. After this, the action `putStrLn (head Cs)` demands the head of `Cs`. Then, the first character of the file is read. Since the computation finishes, the rest of the file is not read and the file is closed by the system.
The function `writeFile` writes the string (its second argument) to the file (its first argument). Writing to a file is an "eager" action, in the sense that the second argument must be totally evaluated before writing it to the file. Once the process terminates the file is closed by the system.
The action `copyFile` below shows how to copy the contents of a file into another:

```
copyFile :: io unit
copyFile = do {putStr ``First File: " ;
               File1 <- getLine ;
               putStr "Second File: " ;
```

```
File2 <- getLine ;
Cs <- readFile File1;
writeFile File2 Cs}
```

It is important to point out that the evaluation of the action `readFile` (respectively, `writeFile`) opens a file, and that such a file remains opened as far as the end of the evaluation of the expression in which it is involved.
On the other hand, $\mathcal{TOY}$ *forbids to open a file previously opened* (independently of the mode). This is the reason why the system closes a file as soon as it detects that a `readFile` or `writeFile` operation has finished. This remark must be taken into account in the rest of this section.

5.4.1 The `io_file` System Module

By loading the system module `io_file` (using the command `/io_file`), $\mathcal{TOY}$ provides new functions to handle text files. This system module can be unloaded by executing the command `/noio_file`.
The system module `io_file` defines operations to read from and write to files, represented by values of a new datatype `handle` (which is implementation-dependent). More precisely, `io_file` provides a dataype `ioMode` and a collection of I/O operations, as shown below:

```
data ioMode = readMode | writeMode | appendMode

openFile      :: path -> ioMode -> io handle
closeFile     :: handle -> io unit
end_of_file   :: handle -> io bool
getCharFile   :: handle -> io char
putCharFile   :: handle -> char -> io unit
putStrFile    :: handle -> string -> io unit
putStrLnFile  :: handle -> string -> io unit
getLineFile   :: handle -> io string
```

The execution of the action `openFile Name Mode` opens the file named `Name` in the mode established by `Mode`, and yields a handle to manage the file in subsequent computations. If `Mode` is `readMode`, then the file is opened for input. This implies that the file `Name` must exist. Otherwise, a runtime error will appear. If `Mode` is `writeMode` then the file is opened for output. Here there are two possibilities. If the file `Name` already exists, then its content is removed and replaced by the following writings. Otherwise, the file `Name` is created. Finally, if `Mode` is `appendMode` then the file `Name` is opened for output, and the current I/O position indicating where the next output operation will occur, is positioned at the end of the file. As for `writeMode`, the file `Name` may exist or not, and the behaviour is similar.
The execution of the action `closeFile Handle` closes the text file referenced by `Handle`, whereas the execution of `end_of_file Handle` returns `true` if no further input can be taken from the file referenced by `Handle`. Otherwise, it returns `false`.

`getCharFile Handle` reads the character situated in the current I/O position of the file referenced by `Handle`. `putCharFile Handle C` has a similar behaviour, but it writes the character `C`. In both cases, the current I/O position advances one character.
Finally, `putStrFile Handle Cs` writes the string `Cs` from the beginning of the current I/O position, putting this at the end of the string. `putStrLnFile Handle Cs` does the same but it prints a newline character after printing the string. `getLineFile Handle` reads the rest of the line from the current I/O position and puts the new I/O position at the beginning of the following line.
As an example, let us redefine the function `copyFile` using the operations contained in the system module `io_file`.

```
copyFile :: io unit
copyFile = do {putStr ``First File: '' ;
               File1 <- getLine ;
               putStr ``Second File: '' ;
               File2 <- getLine ;
               H1 <- openFile File1 readMode ;
               H2 <- openFile File2 writeMode ;
               copy_content H1 H2}

copy_content       :: handle -> handle -> io unit
copy_content H1 H2 = do {B <- end_of_file H1 ;
                         if B then closeFile H1 >> closeFile H2
                              else do {C <- getCharFile H1 ;
                                       putCharFile H2 C ;
                                       copy_content H1 H2}}
```

5.5 Graphic Input/Output

$\mathcal{TOY}$ provides a graphical user interface for application programs similar to that existing in the lazy functional logic language Curry [15]. To this end, $\mathcal{TOY}$ incorporates a system module, based on Tcl/Tk, which provides different functions to create friendly interfaces. In order to use the graphic functions, it is needed to load the system module `io_graphic` using the command `/io_graphic`. The system module can be unloaded by executing the command `/noio_graphic`. The system module `io_graphic` defines operations allowing the communication between $\mathcal{TOY}$ and Tcl/Tk, by means of a new datatype *channel* (which is implementation-dependent). The basic functions provided by the system module `/noio_graphic` are the following:

```
openWish  :: string  -> io channel
readWish  :: channel -> io string
writeWish :: channel -> string  -> io unit
closeWish :: channel -> io unit
```

The execution of the expression `openWish Title` opens a Tcl/Tk window with title `Title` and returns a bidirectional communication channel. After executing this command, the operations

`readWish` and `writeWish` allow communication with the new window by means of the returned channel.
The action `writeWish Channel Str` writes the string `Str` (which must be a well-formed Tcl/Tk command) in the channel `Channel`, whereas `readWish Channel` returns the string read from the channel `Channel`. The communication between the Tcl/Tk window and $\mathcal{TOY}$ may be closed by executing the command `closeWish Channel`.
As a first simple example, consider the following "Hello World" program:

```
hello_world :: io unit
hello_world = do {Ch <- openWish "Hello World" ;
                  writeWish Ch "button .butExit -text \"Exit\" -command {puts \"e\" }";
                  writeWish Ch "pack .butExit" ;
                  Str <- readWish Ch ;
                  closeWish Ch}
```

The execution of the expression `> hello_world` opens the window below, in such a way that a click on the button closes the window.

As a second example, here we have an extension of the program above. In this version, one more button has been added, whose associated command has as effect to write `Hello World` in the standard output. The $\mathcal{TOY}$ code of the program is:

```
hello_world2 :: io unit
hello_world2 =
       do {Ch <- openWish "Hello World" ;
           writeWish Ch "button .butHello -text \"Hello World\"
                         -command {puts \"h\"}" ;
           writeWish Ch "button .butExit -text \"Exit\"
                         -command {puts \"e\"}" ;
           writeWish Ch "pack .butHello .butExit -side left" ;
           runHelloWorld Ch}

runHelloWorld :: channel -> io unit
runHelloWorld Ch = do {Str <- readWish Ch ;
                       if Str == "e" then do {putStrLn "Goodbye" ;
                                              closeWish Ch}
                                     else do {putStrLn "Hello World" ;
                                              runHelloWorld Ch}}
```

Note that the buttons have associated the Tcl/Tk command `puts String`. The execution of this command writes in the communication channel the string `String`. This string is used to detect

which button has been clicked on. The auxiliary recursive function `runHelloWorld` executes the corresponding action in each case, depending on the read string. Thus, when clicking the button `Exit`, the message `Goodbye` is written in the standard output of $\mathcal{TOY}$ and the window is closed, and when clicking on the other button writes `Hello World` and the execution of the program continues in the same way. Finally, the form of the window generated by the program is the following:

It is important to point out that the four basic operations previously presented allow to program with graphic I/O by using purely Tcl/Tk commands. However, the system module `io_graphic` incorporates also new datatypes and functions avoiding the writing of Tcl/Tk commands, and facilitating the programming. As in the functional logic language Curry[2] [14], graphic I/O is specified by means of the datatype `tkwidget A`, whose definition is the following:

```
data tkWidget A =
  tkButton (channel -> A) [tkConfItem A] % A simple press button
| tkCanvas                 [tkConfItem A] % For drawing lines, circles, etc.
| tkCheckButton            [tkConfItem A] % A botton that holds a state of either
                                          % on or of
| tkEntry                  [tkConfItem A] % A text entry field
| tkLabel                  [tkConfItem A] % A simple label
| tkListBox                [tkConfItem A] % A box containing a list of options
| tkMessage                [tkConfItem A] % A multi-line text widget
| tkMenuButton             [tkConfItem A] % A button which when pressed
                                          % offers a selection of choices
| tkScale int int          [tkConfItem A] % Resizes the widget when moving
                                          % the slider
| tkScrollV tkRefType      [tkConfItem A] % Adds vertical scrollbars to
                                          % windows or canvases
| tkScrollH tkRefType      [tkConfItem A] % Adds horizontal scrollbars to
                                          % windows or canvases
| tkTextEdit               [tkConfItem A] % A sophisticated multi-line text widget
                                          % that can also display other widgets
                                          % such as buttons
| tkRow [tkConfCollection] [tkWidget A]  % A row of widgets
| tkCol [tkConfCollection] [tkWidget A]  % A column of widgets
```

In fact, the polymorphic type `tkWidget A` is only used for the instance `io unit`. However, we have preferred to use a polymorphic type in sight to future improvements.

[2]In fact, the rest of definitions referring to Tcl/Tk are an adaptation of an existing graphic I/O Curry library to $\mathcal{TOY}$ syntax.

A GUI can be either a simple widget or a collection (`tkRow`, `tkCol`) of widgets whose first argument `[tkConfCollection]` specifies the geometric alignment. The argument `[tkConfItem A]` of simple widgets allows to define the configuration options. For a `tkButton` widget, it is compulsory to associate a command to the "press" event of the button.
The possible configurations of a simple widget are the following:

```
data tkConfItem A =
      tkActive bool           % Active state of buttons, entries,..
    | tkAnchor string         % Alignment of information in a widget
                              % (argument must be: n, ne, e, se,
                              % s, sw, w, nw, center)
    | tkBackground string     % Background color
    | tkCmd (channel -> A)    % An associated command
    | tkHeight int            % The height of a widget (chars for text,
                              % pixels for graphics)
    | tkInit string           % Initial value for checkbuttons
    | tkItems [tkCanvasItem]  % List of items in a canvas
    | tkList [string]         % Value list in a list box
    | tkMenu [tkMenuItem A]   % The items of a menu button
    | tkRef tkRefType         % A reference to the widget
    | tkText string           % An initial text contents
    | tkWidth int             % the width of widget (chars for text,
                              % pixels for graphics)
    | tkTcl string            % Further options in tcl syntax

data tkMenuItem A =
      tkMButton (channel -> A) string     % A button with an associated
                                          % command an a label
    | tkMSeparator                        % A separator between entries
    | tkMMenuButton string [tkMenuItem A] % A submenu

data tkCanvasItem = tkLine [(int,int)] string
                  | tkPolygon [(int,int)] string
                  | tkRectangle (int,int) (int,int) string
                  | tkOval (int,int) (int,int) string
  % The last arguments are further options in tcl syntax
```

The type `tkRefType` allows to reference widgets. This datatype is built-in and the references are built automatically. Thus, the data constructor `tkRef` must always be used with a variable as an argument. Such a variable will be able to be used by the programmer whenever he wants to reference the widget in order to access or to change its configuration. We will show the use of the constructor `tkRef` in following examples.
The datatype defining the options for geometric alignment of widgets is:

```
data tkConfCollection =
   tkCenter   % Centered alignment
 | tkLeft     % Left alignment
 | tkRight    % Right alignment
```

```
| tkTop      % Top alignment
| tkBottom   % Bottom alignment
| tkExpand   % Expands subwidgets if extra space is avalaible
| tkExpandX  % Expands subwidgets horizontally if extra space is avalaible
| tkExpandY  % Expands subwidgets vertically if extra space is avalaible
```

As an example, consider the function `hello_world2` defined previously. The associated layout of the GUI can be defined now by using the new datatypes. The resulting code would be:

```
win_hello :: tkWidget (io unit)
win_hello = tkRow [] [tkButton eventHello [tkText "Hello World"],
                      tkButton eventExit  [tkText "Exit"]]

eventExit :: channel -> io unit
eventExit Ch = do {putStrLn "Goodbye";
                   tkExit Ch}

eventHello :: channel -> io unit
eventHello Ch = putStrLn "Hello World"
```

The execution loop defined in the function `runHelloWorld` is automated by the built-in function `runWidget :: string -> tkWidget (io unit) -> io unit`
Thus, the evaluation of the expression:

```
> runWidget "Hello World" win_hello
```

has the same effect as the evaluation of `> hello_world2`.
`runWidget Title Widget` runs the widget `Widget` in a new window with title `Title`. The evaluation of such expression generates an event-oriented loop which terminates when closing the window. In our example at hand, there are two possible events: the "exit" button is pressed or the " hello world" button is pressed, whose associated handlers are the defined functions `eventExit` and `eventHello` respectively. Note that the function `eventExit` invokes the built-in function `tkExit :: channel -> io unit` which closes the window. The function `eventHello` is not recursive because the execution loop is done by the function `runWidget`. In general, to program with function `runWidget` requires to define the layout of the GUI (in our example at hand, function `win_hello`) and the functions defining the event handlers (in our example at hand, functions `eventExit`, `eventHello`).
The built-in functions which facilitate the programming of the event handlers are the following:

- `tkVoid :: channel -> io unit`
  the void event handler.
- `tkExit :: channel -> io unit`
  the event handler for closing a window.
- `tkGetValue :: tkRefType -> channel -> io string`
  gets the value (a string) of the text variable of the widget referenced by the first argument.

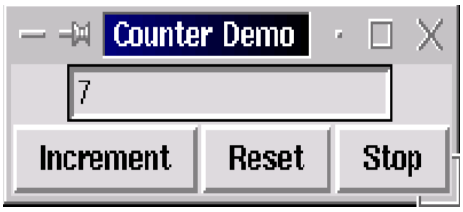
- `tkSetValue :: tkRefType -> string -> channel -> io unit`
  sets the value (a string) of the text variable of the widget referenced by the first argument.
- `tkUpdate :: (string -> string) -> tkRefType -> channel -> io unit`
  updates the value of the text variable of the widget referenced by the second argument with respect to the update function given by the first argument.
- `tkConfig :: tkRefType -> tkConfItem A -> channel -> io unit`
  changes the configuration (except for commands) of the widget;
- `tkFocus :: tkRefType -> channel -> io unit`
  sets the input focus of the GUI (given by the second argument) to the widget referenced by the first argument.
- `tkAddCanvas :: tkRefType -> [tkCanvasItem] -> channel -> io unit`
  adds a list of canvas items to the canvas referenced by the first argument.

In the following example, we illustrate the use of some of the functions above.

```
counter :: tkWidget (io unit)
counter =
   tkCol []
   [tkEntry [tkRef Val, tkText "0"],
    tkRow [] [tkButton (tkUpdate incrText Val) [tkText "Increment"],
    tkButton (tkSetValue Val "0") [tkText "Reset"],
    tkButton tkExit [tkText "Stop"]]]

incrText :: string -> string
incrText Cs = intTostr ((strToint Cs) + 1)
```

The evaluation of the expression `> runWidget "Counter Demo" counter` generates the following window, in which the button "Increment" has been pressed seven times.



In order to increment the counter we have used a reference (`tkRef Val`), which allows to modify the display by using `Val`. The function `tkUpdate` has been used to increment the display. On the other hand, the action associated to button "Reset" puts "0" in the display, by using the built-in function `tkSetValue`.
Note that the event handler of the button "Increment" (function `incrText`) does not require extra information, because all that it needs is contained in the proper GUI; but in general, the event handlers are more complicated and it is usually useful to use *mutable variables* [16, 38] in order to store states.

5.5.1 Mutable Variables

$\mathcal{TOY}$ incorporates the possibility of using mutable variables. The system module `io_graphic` defines operations to manipulate the State Reader monad by means of a new datatype `varmut` (which is implementation dependent). The built-in functions associated to mutable variables are the following:

- `newVar :: A -> io (varmut A)`
  `newVar V` creates a mutable variable with an initial value `V`.
- `readVar :: (varmut A) -> io A`
  `readVar Mut` returns the value of the mutable variable `Mut`.
- `writeVar :: A -> (varmut A) -> io unit`
  `writeVar V Mut` writes a new value `V` in the mutable variable `Mut`.

As an example, here we have a $\mathcal{TOY}$ program which implements a calculator. In the associated code we have used a mutable variable consisting of a pair, where the first component is used to store the accumulated value (an integer), and the second component stores the arithmetic operations pending to be executed. When the button "=" is clicked on, the function stored in the second component will be applied to the first component in order to get the result.
The $\mathcal{TOY}$ code for the calculator is the following:

```
type calcState = varmut (int, int -> int)

calculator :: io unit
calculator = do {St <- newVar (0,id) ;
                 runWidget "calculator" (calc_GUI St)}

calc_GUI :: calcState -> tkWidget (io unit)
calc_GUI St = tkCol []
              [tkEntry [tkRef Display, tkText "0"],
               tkRow [] (map (cbutton St Display) ["1","2","3","+"]),
               tkRow [] (map (cbutton St Display) ["4","5","6","-"]),
               tkRow [] (map (cbutton St Display) ["7","8","9","*"]),
               tkRow [] (map (cbutton St Display) ["C","0","=","/"])]

cbutton :: calcState -> tkRefType -> string -> tkWidget (io unit)
cbutton St Display C =
               tkButton (button_pressed St Display C) [tkText C]

button_pressed :: calcState -> tkRefType -> string -> channel -> io unit
button_pressed St Display C Ch =
   if (digit C)
      then do {Acu <- readVar St ;
               FC <- return ((10*(fst Acu)) + (strToint C));
               writeVar (FC, snd Acu) St ;
               tkSetValue Display (intTostr FC)  Ch}
      else if (C=="=")
               then do {Acu <- readVar St ;
                        F <- return ((snd Acu) (fst Acu)) ;
```

```
                writeVar (F,id) St ;
                tkSetValue Display (intTostr F) Ch}
        else if (C=="C")
               then do {writeVar (0,id) St;
                        tkSetValue Display "0" Ch}
               else do {Acu <- readVar St ;
                        writeVar (0,(op ((snd Acu) (fst Acu)) C)) St}

op ::  int -> string  -> (int -> int)
op N "+"  = (N +)
op N "-"  = (N -)
op N "*"  = (N *)
op N "/"  = (N `div`)
```

Note that the mutable variable is created in the function `calculator` with an initial value `(0,id)`, where `id` is the identity function. This variable is passed as argument to be used by those functions which require to manipulate the state. On the other hand, we have also used a reference to the display (`tkRef Display`), which is also passed as argument in order to allow to modify it. The third argument of function `button_pressed` is the label of the pressed button. Finally, the execution of the expression `> calculator` generates the following window, whose behaviour is the expected one:

5.6 List Comprehensions

$\mathcal{TOY}$, in common with some other functional (logic) languages like Haskell and Curry , provides a powerfull list notation called *list comprehensions*. This notation allows to define lists in a compact way, describing the properties which must satisfy the elements of a list, instead of enumerating its elements. Here is an example:

```
[X*X || X <- [1,2,3,4,5,6,7,8,9,10], odd X]
```

The evaluation of the above comprehension list returns the list `[1,9,25,49,81]` as result, i.e., it returns the list of squares of odd[3] numbers in the range `1` to `10`. Formally, a list comprehension takes the form:

$$[\mathtt{e} \mid\mid \mathtt{q}_1, \mathtt{q}_2, \ldots, \mathtt{q}_n]$$

where `e` is any $\mathcal{TOY}$ expression whose type is the type of the elements of the comprehension list, and $\mathtt{q}_i$, $1 \leq i \leq n$, is a *qualifier*, that is either

- a *generator* of the form `X <- xs`, where `X` is a fresh variable of type τ, for some type τ, and `xs` is an expression of type $[\tau]$, or
- a *condiition* given as a *boolean expression* `b`, or
- a *condition* given as a *goal* `G`. In this case, `G` is treated as a boolean expression, i.e.,
  - if the evaluation of `G==true` succeeds then `G` returns `true`;
  - if the evaluation of `G==false` succeeds then `G` returns `false`;
  - otherwise, the evaluation of the comprenhension list fails.

  As an example, given the two partial functions:

  ```
  f 1 = 1      g 3 = 1
  f 2 = 2      g 4 = 2
  ```

  the comprehension list `[X || X <- [1,2], Y <- [3,4], f X == g Y]` can be evaluated to the list `[1,2]`. Note that, if we would change the generator `X <- [1,2]` by the new one `X <- [1,2,3]`, then the evaluation of the comprehension list would fail. In fact, for any element `y` of `[3,4]`, neither `(f 3 == g y) == true` nor `(f 3 == g y) == false` succeeds (because `f 3` is not defined).

As in functional languages, generators must occur before those boolean conditions using the generated variable, and as near as possible to them, in order to increase the efficiency.
Comprehension lists are understood as a shorthand for computations involving conditional expressions and the functions `map` and `concat` (defined in Sections 2.14 and 2.6). More concretely, the precise meaning of a comprehension list is given by the following translation rules:

```
[e || ] = [e]
[e || b, q] = if b then [e || q] else []
      (where b is a boolean expression or a goal)
[e || X <- L, q] = concat (map f_x Xs)wheref_x X = [e || q]
```

As an example, consider again the comprehesion list:

[3]We are assuming that the file `misc.toy`, containing the function `odd`, has been previously included.

```
[X*X || X <- [1,2,3,4,5,6,7,8,9,10], odd X]
```

The translation of this comprehension list by applying the translation rules is the following:

```
[X*X || X <- [1,2,3,4,5,6,7,8,9,10], odd X]  =
concat (map f_x [1,2,3,4,5,6,7,8,9,10])
```

where

```
f_x X = [X*X || odd X]
```

Since

f_x i = [i*i || odd i], $1 \leq i \leq 10$

then:

```
f_x i = [i*i], if odd i
f_x i = [], otherwise
```

Hence:

```
[X*X || X <- [1,2,3,4,5,6,7,8,9,10], odd X] =
concat [[1*1],[],[3*3],[],[5*5],[],[7*7],[],[9*9],[]] =
[1,9,25,49,81]
```

Differently to functional programming, the conditions occurring in a comprehension list (named *guards* in Haskell), may contain variables not generated previously. Similarly, in a comprehension list `[e || Q]`, `e` may contain variables not occurring in `Q`. In both cases, the non-generated variables aree treated as logic variables; depending on the computational context, they can become bound to some pattern or stay free.
For instance, for generating a list containing 5 pairwise distinct variables, we can execute the following allowed goal:

```
[Y || X <- [1,2,3,4,5]] == L
```

and the computed answer is:

```
L ==  [ _C, _D, _E, _F, _G ]
```

As a second example, suppose the following function definition:

```
p :: int -> bool
p 1 = true
p 2 = true
p 3 = false
```

Then, the allowed goal `[X || p X] == L` has the following three solutions:

```
X == 1          X == 2          X == 3
L ==  [ 1 ]     L ==  [ 2 ]     L == []
```

However, note that the goal

```
[X || X <- [1,2,3], p X] == L
```

has as unique solution `L=[1,2]`. This is because we have generated previously the domain of the function `p`, using the comprehension list as a picker of solutions for `p`.
To conclude with this section, we remark that comprehension lists provide a solution to the problem of programming monadic I/O in the context of a search for multiple solutions. Since the monadic I/O operations are not guaranteed to work correctly within non-deterministic computations, the idea is to use a comprehension list to obtain all the desired solutions within one single deterministic computation. As a concrte example, we show a function that computes all the permutations of a character string, writing each one of them in a different line.

```
include "misc.toy"

show_perms :: string -> io unit
show_perms L = putListStr (perms L)

putListStr :: [string] -> io unit
putListStr  [] = "\n"
putListStr  [S|Ss] = putStrLn S >> putListStr Ss

perms :: [A] -> [[A]]
perms [] = [[]]
perms L = [[Y|Ys] || Y <- L, Ys <- perms (remove Y L)]

remove :: A -> [A] -> [A]
remove Y [X|Xs] = if Y == X then Xs else [X | remove Y Xs]
```

The execution of the expression

```
> show_perms "abc"
```

will show on the screen:

```
abc
acb
bac
bca
cab
cba
```

Chapter 6

Debugging Tools

6.1 Introduction

In this chapter we describe the debugging tool $\mathcal{DDT}$, which is part of the $\mathcal{TOY}$ system. As explained in Section 1.1 (pg. 8), the debugger needs the *Java Runtime Environment*, which is usually part of most operating system distributions, and can otherwise be downloaded from `http://java.com`.
Section 6.2 introduces briefly *declarative debugging*, the theoretical basis of our debugger. Section 6.3 presents a simple example. The limitations of the debugger are discussed in section 6.4. Finally, section 6.5 gives some hints about the internals of the tool. Some examples of buggy programs are included in the examples/debug directory of the distribution.

6.2 Declarative Debugging

The impact of declarative languages on practical applications is inhibited by many known factors, including lack of *debugging tools*, whose construction is recognized as difficult for lazy functional languages. As argued in [39], such debuggers are needed, and much of interest can be still learned from their construction and use. Debugging tools for lazy functional logic languages are even harder to construct.
A promising approach is *declarative debugging*, which starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. In the case of (constraint) logic programs, error symptoms can be either *wrong* or *missing* computed answers. The overall idea behind declarative debugging is to build a *computation tree* [29] as logical representation of the computation. Each node in such a tree represents the result of a computation step, which must follow from the results of its children nodes by some logical inference. Diagnosis proceeds by traversing the computation tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* [29] is found, whose result is erroneous, but whose children have all correct results. The user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, an the debugger can display the program fragment responsible for it.

6.3 An Example

6.3.1 Starting $\mathcal{DDT}$

Recall the datatype `nat` and the functions `head`, `tail`, and `map`, all defined in Chapter 2. Consider also the functions `from`, `plus` and `times`, defined as follows:

```
from   :: nat -> [nat]
from N  = N : from (suc N)

plus           :: nat -> nat -> nat
plus z    Y     = Y
plus (suc X) Y  = suc (plus X Y)

times           :: nat -> nat -> nat
times z       Y  = z
times (suc X) Y  =  plus X (times X Y)    % Should be plus Y (times X Y)
```

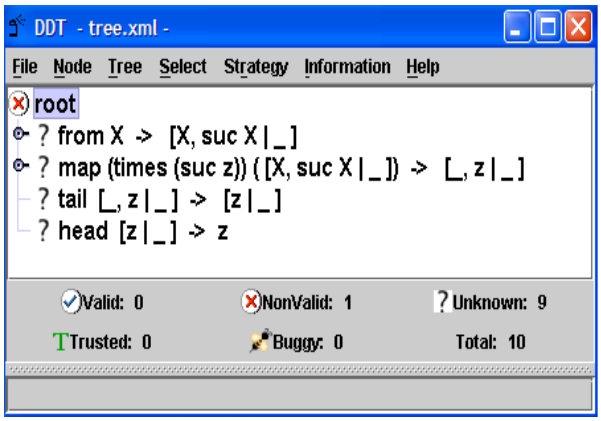
The intended meaning of this program fragment should be clear. In particular, the functions `plus` and `times` are expected to compute the addition and multiplication of natural numbers, in Peano notation. However, the second rule for `times` is incorrect. A user might detect an error symptom when asking for solutions of the following goal, intended to compute in `Y` the second element of the infinite list `N*X : N*(X+1) :  N*(X+2) :  ...`

```
TOY>  head (tail (map (times N) (from X))) == Y
      yes
      N == z
      Y == z

more solutions (y/n/d) [y]? y
      yes
      N == (suc z)
      Y == z

more solutions (y/n/d) [y]? d
```

The first computed answer is correct: if `N` is `z` then the second element of the list will be zero (in fact the list will be an infinite list of zeros). But the second answer is wrong: if `N` is `suc z`, then the second element of the list should be `suc X`, not `z`. Therefore the user answers `d` to the question `more solutions (y/n/d) [y]?` and the debugging process starts. After some previous work (explained briefly in Section 6.5) the debugger displays:



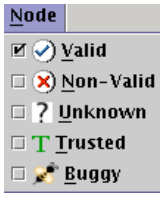
The complete tree can be examined by choosing the option *Tree+Expand Tree*. The root of the tree corresponds to the initial goal and is not displayed. The children nodes correspond to the function calls occurred at the patent. The nodes contain *basic facts* of the form:

$$f\ \bar{t}_n\ \rightarrow\ t$$

where $\bar{t}_n$, t are patterns and i is the position of the fact in the list. A basic fact $f\ \bar{t}_n\ \rightarrow\ t$ must be recognized as valid iff t represents a finite approximation of the result expected for the function call $f\ \bar{t}_n$ according to the *intended model* of the program; see Appendix F. For instance, if the intended meaning of `sort` is a list-sorting function, then the fact `sort [2.0, 1.0, 3.0] -> [3.0, 1.0]` is not valid, since `[3.0, 1.0]` is not a valid approximation of the expected result of `sort [2.0, 1.0, 3.0]`.
The patterns t_i, t can be *partial*, i.e., they can include the special symbol (_) representing an unknown value. This is used in place of suspended function calls whose evaluation has not been demanded by the main computation. In fact, (_) corresponds to the undefined value $\perp$ used to formalize declarative semantics in Appendix F. Thanks to this symbol, questions to the oracle are semantically correct, but much simpler than they would be if suspended subexpressions were displayed. Let us consider some simple examples based on the list processing functions from Section 2.6. The fact `head [1| _ ] -> 1` is valid, because the first element of a list beginning by `1` is indeed `1`. The two `from 0 -> [0 | _ ]` and `from 0 -> [0,1 | _ ]` are also valid, because `[0 | _ ]` and `[0,1 | _ ]` are finite approximations of the expected result of `from 0` (namely, the infinite list of all non-negative integers). On the other hand, the fact `from 0 -> [0,2,4 | _ ]` is not valid.

6.3.2 Looking for buggy nodes

The user can navigate the tree providing information about the *state* of the nodes by using the menu option *Node*:



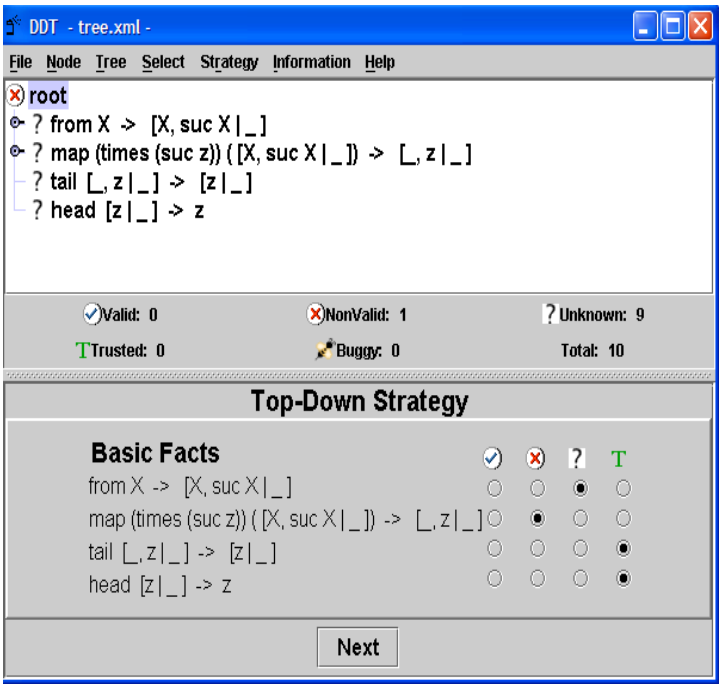
Apart from *valid* and *Non-valid* there are three other possible states:

- *Buggy*: The user cannot indicate that a node is buggy, only the system changes a node state to buggy after checking that it is not valid and all its children are valid. We have proved that a buggy node has always an incorrect program rule associated, the program rule used to compute the function call represented in the node. After finding a buggy node, the debugger displays the number of the rule associated and the debugging session ends.
- *Unknown*: The user doesn't know if the node is valid or not. All the nodes are marked as *unknown* at the beginning.
- *Trusted*: The function associated to the basic fact in the node can be trusted, i.e. the node is valid and all the nodes corresponding to the same function are also valid. Hence changing a node to this state will change all the nodes containing basic facts associated to the same function to *trusted* as well.

6.3.3 Strategies

For large trees the process of finding a buggy node can be tiresome. In these cases (in fact in almost all the cases) it is convenient to use a fixed strategy. $\mathcal{DDT}$ provides two of such strategies: the *top-down* and the *divide-and-query*.
The top-down strategy behaves essentially like the textual debugger presented in the previous section. The process starts with a computation tree whose root is considered non-valid. Then the children of the root are examined looking for some non-valid child. If such child is found the debugging continues examining its corresponding subtree. Otherwise all the children are valid and the root of the tree is pointed out as buggy, finishing the debugging.

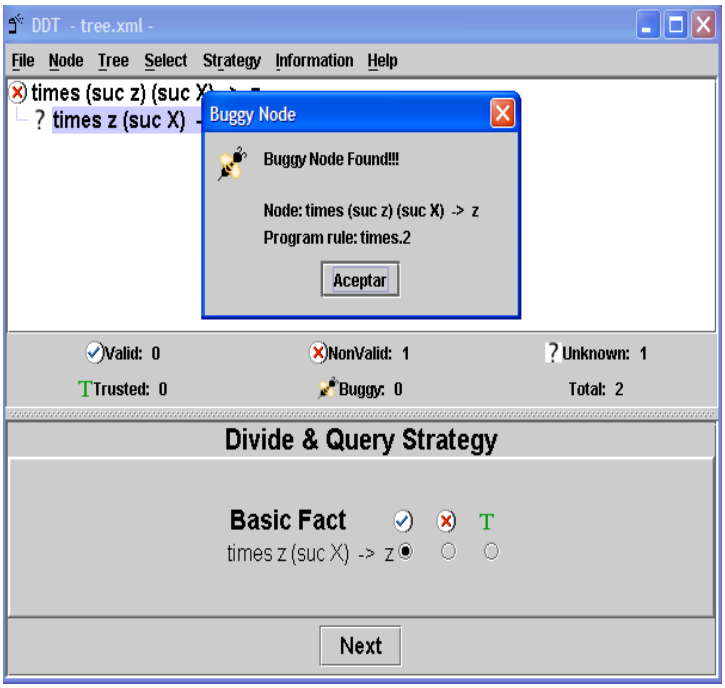
The next display shows the starting point of the a debugging session using the top-down strategy, where the user has marked the second node as non-valid, and the last two nodes as trusted:



Notice that after each subsequent step the selected subtree has a smaller size and an invalid root, until a buggy node is eventually reached.

As in the top-down strategy, the *divide and query* strategy starts with a computation tree whose root is not valid. The idea is to choose a node N such that the number of nodes inside and outside of the subtree rooted by N are the same. Although such node (called the *center* of the tree) does not exist in most of the cases, the system looks for the node that better approximates the condition. Then the user is queried about the validity of the basic fact labeling this node. If the node is non-valid its subtree will be considered at the next step. If it is valid then its subtree is deleted from the tree and debugging continues. The process ends when the subtree considered has been reduced to a single non-valid node.

Is easy to observe that, as in the top-down strategy, the number of nodes in the tree T considered is reduced after each step, and that $nonvalid(root(T))$ holds and that the strategy will find some buggy node. The divide and query requires a number of questions that proportional, in average, to the $log_2\ n$, where n is the number of nodes of the tree, and must usually preferred to the top-down strategy. In our example, after four questions the system finds a buggy node, pointing at the second rule of function times as invalid:



Since these strategies modify the structure of the tree, $\mathcal{DDT}$ includes options to save and load computation trees (see the options of the menu "File"). The files are stored in XML format. These options can be also used to restore a previous version of the debugging session if the user realizes after some steps that she or he made a mistake when providing information about the validity of the nodes, a situation that often arises.

The theoretical results presented in [4, 5] ensure that if debugging starts with an erroneous answer and the user answers the questions correctly, then the debugger eventually locates in the program an incorrect function rule as cause of the error. Note, however, that each debugging session locates only one program error. Other incorrect program rules can still remain, and further debugging session can be required.

6.4 Limitations

The main limitations of our current debugging tool are:

- *Missing answers.* Whenever finitely many answers are computed for a certain goal (followed by a `no more answers` indication), and the user expects some other answer which is not covered by the computed ones, one speaks of missing answers. Finitely failing goals with expected solutions are an important subcase of goals with missing answers. Declarative diagnosis of missing answers should detect a function whose set of defining rules in the given program is incomplete. Currently, our debugger can diagnose wrong answers, but not missing answers.

- *Constraints.* Programs and goals in $\mathcal{TOY}$ can use different kinds of constraints as conditions; see sections 2.6 and 2.16. Currently the debugger supports *strict equality* and

disequality constraints and constraints over *real numbers*, but not programs including *finite domain* constraints.

- *Non-terminating goals.* Generally, declarative debuggers work only after some completed computation whose outcome is found incorrect by the user. This is also the case for our debugger. Tools to prove termination of a given goal are undoubtedly useful, but they are outside the scope of declarative debugging as such.
- *Input/output.* Although the $\mathcal{TOY}$ system supports I/O interaction in monadic style (see Chapter 5), the current debugger can only be applied to computations that do not involve any kind of I/O operations.

Extensions of the debugger to deal with missing answers and other constraints are currently being investigated.

6.5 How Does it Work?

Our debugger uses a program transformation approach to obtain the computation tree associated to a wrong computation. The process is described in [5] and can be summarized as follows:

1. The original program P is transformed into a new program P'. Each function $f :: \overline{\tau}_n \to \tau$ in P is transformed into a function $f' :: \overline{\tau'}_n \to (\tau, cTree)$ in P' that associates a computation tree to each produced result. This new program is then compiled by $\mathcal{TOY}$ and loaded as the current program.
2. The goal is also transformed and solved, using the new program P'. In this way, a computation tree corresponding to the wrong answer is obtained.
3. Finally the computation tree is searched, asking questions to the user as described in the previous sections. When the debugger ends, the original program P is again loaded into memory.

The first two points are performed automatically as soon as the user answers `d` to the question

```
more solutions (y/n/d) [y]?
```

and can take some time, depending on the size of the original program. The last point corresponds to the interaction with the user. If she is able to answer all questions correctly, an incorrect program rule will be diagnosed. Otherwise, the debugging process will abort.

Bibliography

[1] Sergio Antoy, Rachid Echahed, and Michael Hanus. A Needed Narrowing Strategy. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 268–279, New York, NY, 1994.

[2] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In Rina Dechter, editor, *6th International Conference on Principles and Practice of Constraint Programming (CP'97)*, number 1894 in LNCS, pages 52–66, Singapore, 2000. Springer-Verlag.

[3] R. Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall (Series in Computer Science), 1998.

[4] Rafael Caballero, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 170–184, London, UK, 2001. Springer-Verlag.

[5] Rafael Caballero and Mario Rodríguez-Artalejo. A declarative debugging system for lazy functional logic programs. *Electr. Notes Theor. Comput. Sci.*, 64, 2002.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[8] S. Estévez-Martín, A. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez, and R. del Vado-Vírseda. A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming. *ENTCS*, 2007. In Press.

[9] A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In P. Wadler and V. Dahl, editors, *5th International Symposium on Practical Aspects of Declarative Languages (PADL'2003)*, number 2562 in LNCS, pages 320–338, New Orleans, Louisiana, USA, 2003. Springer-Verlag.

[10] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[11] J. C. González-Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[12] J.C. González-Moreno. *Programación Lógica de Orden Superior con Combinadores*. PhD thesis, Universidad Complutense de Madrid, July 1994. In Spanish.

[13] Juan Carlos González-Moreno, Maria Teresa Hortalá-González, and Mario Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), 2001.

[14] M. Hanus. Curry: An Integrated Functional Logic Language. Version 0.7.1., June 2000. Available at `http://www.informatik.uni-kiel.de/ curry/report.html`.

[15] Michael Hanus. A functional logic programming approach to graphical user interfaces. *Lecture Notes in Computer Science*, 1753:47–??, 2000.

[16] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[17] J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994.

[18] T. Klove. Bounds and construction for for difference triangle sets. *IEEE Transactions on Information Theory*, 35:879–886, July 1989.

[19] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory Pract. Log. Program.*, 4(2):41–74, 2004.

[20] Francisco Javier López-Fraguas and Jaime Sánchez Hernández. Proving failure in functional logic programs. In *CL'00: Proceedings of the First International Conference on Computational Logic*, pages 179–193, London, UK, 2000. Springer-Verlag.

[21] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. Narrowing failure in functional logic programming. In *FLOPS '02: Proceedings of the 6th International Symposium on Functional and Logic Programming*, pages 212–227, London, UK, 2002. Springer-Verlag.

[22] Rita Loogen, Francisco Javier Lopez-Fraguas, and Mario Rodriguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *PLILP*, pages 184–200, 1993.

[23] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, number 1631 in LNCS, pages 244–247, Trento, Italy, 1999. Springer-Verlag.

[24] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *LPAR*, pages 455–469, 2001.

[25] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. Failure and equality in functional logic programming. *Electr. Notes Theor. Comput. Sci.*, 86(3), 2003.

[26] M. Hanus (editor). Pakcs 1.4.0, user manual. The Portland Aachen Kiel Curry System. Available from $http://www.informatik.uni-kiel.de/pakcs/$, 2002.

[27] K. Marriot and P. J. Stuckey. *Programming with constraints.* The MIT Press, Cambridge, Massachusetts, 1998.

[28] Gopalan Nadathur and Dale Miller. An overview of lambda-prolog. In *ICLP/SLP*, pages 810–827, 1988.

[29] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[30] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[31] S.P. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, Englewood Cliffs, N.J., 1987.

[32] J. Sánchez-Hernández. $\mathcal{TOY}$: Un lenguaje lógico funcional con restricciones. Research work, Universidad Complutense de Madrid, 1998. In Spanish.

[33] J. Sánchez-Hernández. Implementing constructive failure in functional logic programming. In *Jornadas de Programación y Lenguajes, PROLE*, pages 127–136, 2005.

[34] J.B. Shearer. Some new optimum golomb rulers. *IEEE Transactions on Information Theory*, 36:183–184, January 1990.

[35] J.B. Shearer. Golomb ruler table. `www.research.ibm.com/people/s/shearer/-grtab.html`, 2004.

[36] Sicstus manual. *SICStus Prolog user's manual, release* 3#8. By the Intelligent Systems Laboratory, Swedish Institute of Computer Science, 1999.

[37] P. Van Hentenryck. *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA, 1989.

[38] Ton Vullinghs, Daniel Tuinman, and Wolfram Schulte. Lightweight GUIs for functional programming. In *PLILP*, pages 341–356, 1995.

[39] Philip Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, 1998.

[40] N-F. Zhou. Channel Routing with Constraint Logic Programming and Delay. In *9th International Conference on Industrial Applications of Artificial Intelligence*, pages 217–231. Gordon and Breach Science Publishers, 1996.

Appendix A

Programming Examples

In this chapter we show some of the main feaures of $\mathcal{TOY}$. First we will discuss a few examples that are closest to the logic programming paradigm and functional programming paradigms (sections A.1 and A.2, respectively) but enriched with some specific functional-logic flavour. We consider the pure fragment of *Prolog* and *Haskell* as representatives of the purely logical and lazy functional programming styles, respectively. Next, we present the use of failure. The last three sections present programming with constraints over different domains (constructor terms, reals, and finite domains).
The distribution provides the directory `examples` that contains a number of examples of $\mathcal{TOY}$ programs, included the programs listed in this section. The program title is annotated with the corresponding file name (with extension `.toy`).

A.1 Logic Programming

$\mathcal{TOY}$ embodies all the purely declarative features of logic programming languages such as *Prolog*. Next we show two simple examples in $\mathcal{TOY}$ that correspond to typical logic programs.

A.1.1 Peano Numbers (`peano.toy`)

The next program represents in $\mathcal{TOY}$ a logic program that generates Peano natural numbers:

```
% Datatype for Peano natural numbers
data nat = z | s nat

% Generator for Peano numbers
isNat :: nat -> bool
isNat z     :- true
isNat (s N) :- isNat N
```

The main differences with the corresponding program in Prolog come from the existence of types in $\mathcal{TOY}$. Thus, the type nat is explicitly declared by means of a data definition. In the same way, the type of isNat can be declared. In $\mathcal{TOY}$ predicates are considered as functions returning

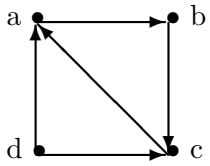
booleans values, and in this case the type of isNat reflects that it is a predicate with a natural number as argument. An example of a goal solved by this program:

```
Toy> isNat N
      { N -> z }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { N -> (s z) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { N -> (s (s z)) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]? n
```

The goal has infinitely many solutions, although in this example the user has stopped after the first three solutions answering n to the question more solutions [y]?. To provide these different answers $\mathcal{TOY}$ performs *backtracking* in the same way as *Prolog* does.

A.1.2 Paths in a Graph (paths.toy)

The aim of the next program is to look for all the paths between two nodes in a given graph. In logic programming this is done via a non-deterministic predicate path(X,Y), taking advantage of Prolog search mechanism to look for all the possible paths between two nodes. The same idea is valid in $\mathcal{TOY}$; the predicate path corresponds in $\mathcal{TOY}$ to a *non-deterministic* boolean function, indicating that X and Y are connected if either there is an arc between them, or there exists an arc from X to a certain node Z such that Z and Y are connected. In the example the graph considered is the following:




```
% Paths in a Graph
data node = a | b | c | d

arc :: (node, node) -> bool
arc(a,b) :- true
arc(b,c) :- true
arc(c,a) :- true
arc(d,a) :- true
arc(d,c) :- true
```

```
path :: (node, node) -> bool
path(X,Y) :- arc(X,Y)
path(X,Y) :- arc(X,Z), path(Z,Y)
```

For instance, the following goal asks $\mathcal{TOY}$ for nodes accessible from node `d`:

```
Toy> path (d,X)
      { X -> a }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X -> c }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X -> b }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]? n
```

A.2 Functional Programming

In this section we show that $\mathcal{TOY}$ can also deal with functional computations. $\mathcal{TOY}$ includes many of the features of *lazy* functional languages such as *Haskell*. In particular, $\mathcal{TOY}$ allows *higher-order function definitions*. We next show all these characteristics by means of examples.

A.2.1 Arithmetic for Peano Numbers (`arithmetic.toy`)

The next example defines addition and multiplication for Peano natural numbers.

```
% Datatype for Peano Natural Numbers
data nat = z | s nat

% Addition and Multiplication for Peano Numbers
infixr 40 +.
(+.) :: nat -> nat -> nat
X +. z     = X
X +. (s Y) = s (X +. Y)

infixr 50 *.
(*.) :: nat -> nat -> nat
X *. z     = z
X *. (s Y) = s (X *. Y) +. X
```

This $\mathcal{TOY}$ program could be considered a *Haskell* program by ignoring the syntactic differences of upper and lowercase identifiers for variables and constructors. We can use this program to compute the value of expressions as in any functional language:

```
Toy> > s z +. (s (s z))

      ( s ( s ( s z ) ) )

      Elapsed time: 0 ms.
```

The symbol > indicates to the system that we are going to evaluate an expression. In this case we could also have tried the same goal in the shape of a $\mathcal{TOY}$ goal:

```
Toy> s z +. (s (s z)) == R
      { R -> (s (s (s z))) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

More noticeable is the skill of $\mathcal{TOY}$ in solving goals with logic variables. For example the next goal asks for all the decompositions of number 3 as the addition of two numbers X and Y:

```
Toy> X +. Y == s (s (s z))
      { X -> (s (s (s z))),
        Y -> z }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X -> (s (s z)),
        Y -> (s z) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X -> (s z),
        Y -> (s (s z)) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X -> z,
        Y -> (s (s (s z))) }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

Thus, even this purely functional program can be used in a functional logic way, by allowing variables in the goals and returning several different answers, if they exist.

A.2.2 Infinite Lists (`inflists.toy`)

The following example takes advantage of lazy computations in $\mathcal{TOY}$ to represent infinite structures in the same way as in *Haskell*:

```
% An infinite list of consecutive integers
from :: int -> [int]
from N = [N | from (N+1)]

%The first N elements in a list
take :: int -> [A]  -> [A]
take N []     = []
take N [X|Xs] = if N > 0 then [X | take (N-1) Xs]
                else []
```

An expression of the form from N represents an infinite list of consecutive numbers starting at number N, while take N L returns the first N elements of list L. Therefore we can evaluate a purely functional expression of shape:

```
Toy> > take 3 (from 0)

       [ 0, 1, 2 ]

      Elapsed time: 0 ms.
```

showing the first three elements of the infinite list on numbers starting at 0. Thus, the infinite list is only evaluated up to the needed point.

A.2.3 Prime Numbers (primes.toy)

In $\mathcal{TOY}$ higher order functions are also allowed, as *Haskell* does. Therefore typical higher-order functions like map, fold, filter, etc, can be defined. The next example defines the infinite list of prime numbers using an algorithm based on the 'sieve of Erathostenes'. It uses the higher-order function filter in order to remove numbers that can be divided by any of previous members of the list. Function type declarations are not mandatory in $\mathcal{TOY}$ and are not included here. Observe also the alternative semantics for defining functions, e.g., --> instead of =. In addition, lists use the constructor symbol ./2.

```
primes --> sieve (from 2)

sieve (X:Xs) --> X : filter (notDiv X) (sieve Xs)

notDiv X Y --> mod Y X > 0

from N --> N : from (N+1)

filter P    []  --> []
filter P (X:Xs) --> if P X
                    then X : filter P Xs
                    else filter P Xs
```

```
take N    []  --> []
take N (X:Xs) --> if N > 0
                  then X : take (N-1) Xs
                  else []
```

Next goal computes the list of the first ten prime numbers

```
Toy> > take 10 primes

       [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]

      Elapsed time: 16 ms.
```

A.2.4 Hamming Codes (`hamming.toy`)

Let H be the smallest subset of N satisfying the following axioms:

- $1 \in \mathcal{H}$
- $\forall x.x \in \mathcal{H} \Leftrightarrow 2x, 3x, 5x \in \mathcal{H}$

The problem consists of obtaining the ordered sequence of elements in $\mathcal{H}$, that is to say:
$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, \ldots$
Let h denotes the list of elements in H. Then, the numbers (i.e., codes) of Hammings can be obtained by *mixing conveniently* the following infinite lists
map (2*) h
map (3*) h
map (5*) h
By the expression '*mixing conveniently*' we mean to order the three lists removing the elements that are duplicate in the lists, that is to say:
$merge3\ U\ V\ W\ =\ (U \alpha V)\ \alpha W$
where the operation $\alpha/2$ is defined as follows:
$[]\ \alpha\ V = V : []$
$U\ \alpha\ [] = U : []$
$[X|Xs]\ \alpha\ [Y|Ys] = X : (Xs\ \alpha\ Ys) \iff X == Y$
$[X|Xs]\ \alpha\ [Y|Ys] = X : (Xs\ \alpha\ [Y|Ys]) \iff X < Y$
$[X|Xs]\ \alpha\ [Y|Ys] = Y : ([X|Xs]\ \alpha\ Ys) \iff X > Y$
and finally to add the element 1 as initial element. A schematic picture is shown in Figure A.1.
A solution is shown below, where `merge2` defines the operator α.

```
include "misc.toy"

merge3 :: [int] -> [int] -> [int] -> [int]
merge3 U  V  W = merge2 (merge2 U V) W
```

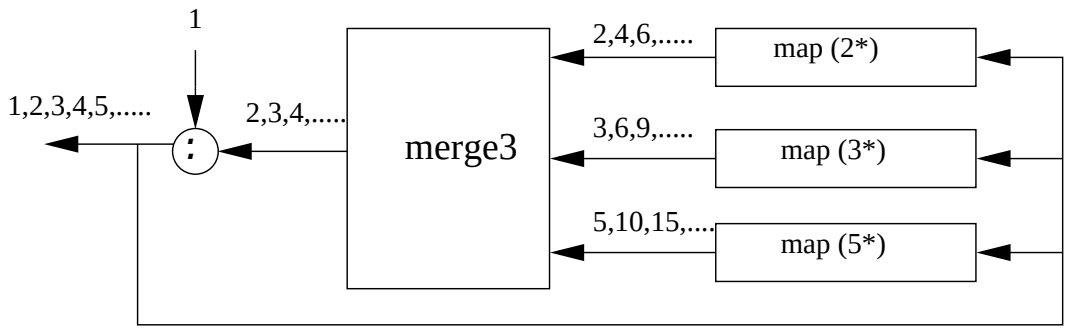


Figure A.1: Hamming Codes

```
merge2 :: [int] -> [int] -> [int]
merge2 []  V = V
merge2 U  [] = U
merge2 [X|Xs] [X|Ys] = [X | merge2 Xs Ys ]
merge2 [X|Xs] [Y|Ys] = [X | merge2 Xs [Y|Ys] ] <== X < Y
merge2 [X|Xs] [Y|Ys] = [Y | merge2 [X|Xs] Ys ] <== X > Y

hamming :: [int]
hamming = 1: (merge3 (map (2*) hamming) (map (3*) hamming) (map (4*) hamming))
```

The next goal computes the 12 first Hamming numbers:

```
Toy(FD)> > take 12 hamming

       [ 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27 ]

      Elapsed time: 47 ms.
```

A.2.5 Process Network: Client-Server Interaction (`clientserver.toy`)

Processes can be considered as functions consuming data (i.e., arguments) and producing values for other functions. Processes are often suspended until the evaluation of certain expression is required (by other process). In these cases, lazy evaluation corresponds to particular coroutines for the processes.
One interesting application is to solve the communication between a client and a server with the Input/Output model via *Streams*: If the client generates requests from one initial requirement, the server will generate answers that will be again processed by the client, and so on. For simplicity, we consider that requests and answers are integer numbers. This process network can be clearly defined in $\mathcal{TOY}(\mathcal{FD})$ with recursive definitions as follows:

```
requests, answers :: [int]
```

```
requests = client initial answers
answers  = server requests
```

Suppose now that the client returns the request and generates a new one (i.e., a next one) from the first answer of the server and that the server processes each request to generate a new answer. This is defined in $\mathcal{TOY}(\mathcal{FD})$ as follows:

```
client :: int -> [int] -> [int]
client Ini [R|Rs] = [Ini | client (next R) Rs]
server :: [int] -> [int]
server [P|Ps] = [process P | server Ps]
```

The architecture is completed by defining adequately the initial requirement, the processing function and the selection of the next request. As an example, and for simplicity, we can define them as follows:

```
process :: int -> int process = (+3)

initial :: int initial = 4

next :: int -> int
next = id  %Idempotence
```

Note that this is not enough to produce an outcome as the goal `requests` goes into a non-ending loop. However, the lazy evaluation mechanism of $\mathcal{TOY}(\mathcal{FD})$ allows to evaluate a finite number (`N`) of requests; this can be done by redefining the functions `client, answers` and `requests` as follows:

```
client :: int -> [int] -> [int]
client Ini Rs = [Ini | client (next (head Rs)) (tail Rs)]

answers :: int -> [int]
answers N = server (requests N)

requests :: int -> [int]
requests N = take N (client initial (take N (answers N)))
```

where `head/1` and `tail/1` return the head and tail of a list respectively. Below, we show an example of solving that evaluates exactly the first 15 requests.

```
Toy(FD)> requests 15 == L
      { L ->  [ 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46 ] }
      Elapsed time: 47 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

A.3 Functional Logic Programming

Now we turn to simple examples that show the skill of $\mathcal{TOY}$ in combining the features of the functional and logic languages.

A.3.1 Inserting an Element in a List (`insert.toy`)

Function insert, defined below, inserts an element X in a list.

```
insert :: A -> [A] -> [A]
insert X []     = [X]
insert X [Y|Ys] = [X,Y|Ys]
insert X [Y|Ys] = [Y|insert X Ys]
```

If the list is empty there is only one possibility, as the first rule of insert reflects. On the contrary, if the list has n elements with $n > 0$, there are $n + 1$ different ways of inserting X. In logic programming, and in $\mathcal{TOY}$ as well, all of these possibilities can be encoded by defining a non-deterministic predicate. However, in $\mathcal{TOY}$ we can also define a suitable *non-deterministic function*, which renders a more expressive solution in this case. Next goal shows how the different output lists are computed.

```
Toy> insert 1 [2,4,6] == X
      { X ->  [ 1, 2, 4, 6 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X ->  [ 2, 1, 4, 6 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X ->  [ 2, 4, 1, 6 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X ->  [ 2, 4, 6, 1 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

A.3.2 The Choice Operator (`choice.toy`)

The following operator captures the essence of non-deterministic functions. It is called the *choice* operator and selects either of its two arguments.

```
infixr 20 //

(//) :: A -> A -> A
```

```
X // Y = X
X // Y = Y
```

For instance, the function insert can be rewritten in the following, more appealing way:

```
insert :: A -> [A] -> [A]
insert X []     = [X]
insert X [Y|Ys] = [X,Y|Ys] // [Y|insert X Ys]
```

which yields the same results as above.

A.3.3 The Inverse Function (`inverse.toy`)

The following is a function that computes the inverse of a given function F. Its definition says that X is image of Y through the inverse of F if $F\ X = Y$. Notice that X does *not* appear at the left-hand side of the function.

```
inverse :: (A -> B) -> (B -> A)
inverse F Y = X <== F X == Y
```

For instance, the inverse of inserting an element in a list consists of removing such element:

```
Toy> inverse (insert 3) [4,5,3] == X
      { X ->  [ 4, 5 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

Of course, if function F is not injective, then inverse F will behave as a non-deterministic function.

A.4 Programming with Failure

A.4.1 Default Rules (`automata.toy`)

An interesting use of failure in functional logic programming is to express *default rules*. These rules are implicit in functional systems like Haskell where pattern matching determines *the* rule to apply for reducing a call to a function: the n-th rule only applies if pattern matching fails for the previous $n-1$ rules. As an example, consider the following function:

```
f 0 = 0
f X = 1
```

If we evaluate the expression $f\ 0$ in Haskell we get the result 0, by the first rule. Here the second rule is working as a default rule as it only applies if the previous fail (in pattern matching). In contrast, if we submit the corresponding goal $f\ 0 == X$ in $\mathcal{TOY}$ we obtain the answers $X \to 0$ and $X \to 1$, due to non-determinism. Nevertheless, it may be usefull in some cases to have in $\mathcal{TOY}$ the behaviour of Haskell and this is possible by using *fails*:

```
f 0 = f' X
f X = 1 <= fails (f' X)
f' 0 = 0
```

It is easy to check that with this definition $f\ 0 == X$ produces only the answer $X \to 0$ and, for example, $f\ 2 == X$ gets $X \to 1$. This scheme of transformation is applicable in the general case. Consider a function h of the form:

$$\begin{array}{l} h\ \bar{t}_1 \to e_1 \Leftarrow \overline{C}_1 \\ \ldots \\ h\ \bar{t}_n \to e_n \Leftarrow \overline{C}_n \\ (''default'')\ h\ \bar{t}_{n+1} \to e_{n+1} \Leftarrow \overline{C}_{n+1} \end{array}$$

The construction *default* is not supported in $\mathcal{TOY}$ (for the moment), but we can translate this function into:

$$\begin{array}{lll} h\ \overline{X} & \to h'\ \overline{X} & \\ h\ \overline{X} & \to e_{n+1} & \Leftarrow \mathit{fails}\ (h'\ \overline{X}) == \mathit{true},\ \overline{C}_{n+1} \\ h'\ \bar{t}_1 & \to e_1 & \Leftarrow \overline{C}_1 \\ \ldots & & \\ h\ \bar{t}_n & \to e_n & \Leftarrow \overline{C}_n \end{array}$$

Notice that with this translation the default rule covers the case of a failure of the previous rules, even in the case that failure does not come from pattern matching.

Default rules can be usefull a variety of situations. As an example we consider the case of finite automata. A finite automata can be defined as a 5-tuple of the form:

$$M = (K, \Sigma, s, F, \delta)$$

where K is a set of states, Σ is an alphabet, s is the initial state, $F \subseteq K$ is the set of final states and $\delta : K \times \Sigma \to K$ is the transiction function. In order to represent automatas in a program we can assume some simplifications: the states are integer values, Σ is the standard set of characters, a word is a string (list of characters). Then we can define in $\mathcal{TOY}$:

```
type state    = int
type symbol   = char
type word     = string
type automata = (state -> symbol -> state,   % delta function
                 state,                      % initial state
                 [state])                    % final states
```

Now we can define a generalized transiction function *deltaGen* to process words by applying δ repeteadly until consume the input symbols:

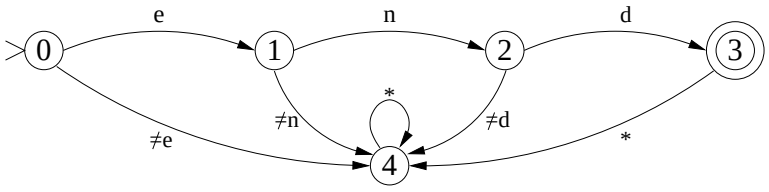
```
deltaGen :: automata -> state -> word -> state
deltaGen (D,I,Fs) Q []     = Q
deltaGen (D,I,Fs) Q [A|As] = deltaGen (D,I,Fs) (D Q A) As
```

And the accepting function:

```
accept :: automata -> word -> bool
accept (D,I,Fs) W = member (deltaGen (D,I,F) I W) Fs

member :: A -> [A] -> bool
member X [] = false
member X [Y|Ys] = if (X==Y) then true else member X Ys
```

When recognizing languages with automotatas, in practise it is usefull to transictions with labels that represent "any character distinct from _" or "any character". For example an automata for recognizing the word "end" could be:



Here 0 is the initial state, 3 is the final state and 4 stands for an *error* state. The label $*$ represents any character and $\neq e$ represents any character except e. According to the previous representation this automata would be:

```
end = (delta,0,[3])
```

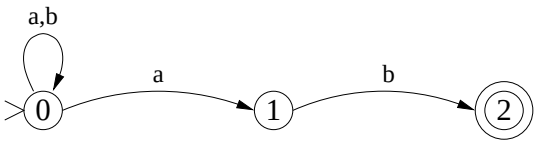
and the transiction function would be expressed as:

$$delta\ 0\ 'e' = 1$$
$$delta\ 1\ 'n' = 2$$
$$delta\ 2\ 'd' = 3$$
$$default\ delta\ S\ C = 4$$

Now, using failure we can eliminate the construction *default* and write in $\mathcal{TOY}$:

```
delta S C = delta' S C
delta S C = 4 <== fails (delta' S C)
delta' 0 'e' = 1
delta' 1 'n' = 2
delta' 2 'd' = 3
```

With this automata the call *accept end "end"* reduces to *true* and with an other string produces *false*, as expected.
Here the use of failure for expressing default rules allows to abreviate definitions (we could define δ for any character). But this use is more critical when programming non-deterministic automata. For example, assume the automata *ends_ab* that accepts words ending with *ab*:



The transiction function δ is programmed as before, but now the function *accept* is wrong because *accept ends_ab "abb"* would reduce to *true* and to *false*: it is possible to make transictions to state 2, but also it is possible to stay at state 0. The function *accept* needs another definition in the case of non-deterministic automata:

$$accept\ (D, I, Fs)\ W = if\ (member\ (deltaGen\ (D, I, Fs)\ I\ W)\ Fs)\ then\ true$$
$$default\ accept\ (D, I, Fs)\ W = false$$

The first rule says reduces to *true* if there is a transiction from the initial state to a final state, while the second rule reduces to *false* otherwise, that is, when there is not any sequence of transictions that reach a final state. This is exactly what we want and it can be implemented using *fails* as:

```
accept (D,I,Fs) W = accept' (D,I,Fs) W
accept (D,I,Fs) W = false <== fails (accept' (D,I,Fs) W)
accept' (D,I,Fs) W = if (member (deltaGen (D,I,Fs) I W) Fs) then true
```

A.4.2 A Propositional Tautology Generator (`tautology.toy`)

The function *fails* allows to transform failure of reduction into the boolean value *false*. In this example we illustrate the reverse use: transform the value *false* into a failure obtaing more control on the non-deterministic reductions.
First of all, to manage success results we introduce the function *success*:

```
success true = true
```

This function is the identity on the value *true*. The interest of such a function is that is not defined for the value *false*, that is, it produces a failure for the value *false*.
Now we build an evaluator for first order logic formulas. For simplicity we assume the proposition symbols p, q and r and define the type of formulas as:

```
type formula = p | q | r | neg formula
               | conj formula formula | disy formula formula
               | implies formula formula | iff formula formula
```

For example, the formula $(p \wedge q) \rightarrow r$ would be represented as *implies (conj p q) r*. An interpretation will be a 3-tuple of boolean values corresponding to the values of p, q and r respectively. The evaluation function *eval F I* evaluates the formula F over the interpretation I and it is defined as:

```
eval p (P,Q,R) = P
eval q (P,Q,R) = Q
```

```
eval r (P,Q,R) = R
eval (neg F) (P,Q,R) = not (eval F I)
eval (conj A B) I = and (eval A I) (eval B I)
eval (disy A B) I = or (eval A I) (eval B I)
eval (implies A B) I = eval (disy (neg A) B) I
eval (iff A B) I = eval A I == eval B I

neg true    = false
neg false   = true
and true  X = X
and false X = false
or  true  X = true
or  false X = X
```

For example, for the previous formula $(p \wedge q) \rightarrow r$ and the interpretation $(true, false, true)$, the expression *eval (imples (conj p q) r) (true,false,true)* will reduce to *true*.
Now we are interested in the satisfactibily of logic formulas, in particular we want a tests for tautologies, contradictions and contingencies. First of all we need a function *inter* for obtaining interpretations (in a non-deterministic way):

```
bval = true
bval = false

inter = (bval,bval,bval)
```

If we evaluate *eval (imples (conj p q) r) inter* we will obtain *true* and *false*, what means that this formula is a contingency. The easier test is that for contradictions that can be programmed as:

```
isContradiction F = fails (success (eval F inter))
```

This rule is read as: a formula is a contradiction if the evaluation function does not success over any interpretation. The other tests are easy to define using this one:

```
isTautology   F = isContradiction (neg F)
isContingency F = and (not (isContradiction F)) (not (isTautology F ))
isSatisfiable F = not (isContradiction F)
```

If we evaluate *isSatisfiable (implies p (conj q (neg p)))* we get *true*.
We can program the generation of formulas that satisfy a given test. Moreover, we can introduce the concept of complexity of a formula in order to obtain more interesting answers. Such a complexity is defined as the number of conectives that they have (we use z and s for representing natural numbers and the usual function *add*):

```
complexity p = z
complexity q = z
complexity r = z
```

```
complexity (neg A) = s (complexity A)
complexity (conj A B) = s (add (complexity A) (complexity B))
complexity (disy A B) = s (add (complexity A) (complexity B))
complexity (implies A B) = s (add (complexity A) (complexity B))
complexity (iff A B) = s (add (complexity A) (complexity B))
```

With this program we can generate for example tautologies of complexity 3 with the goal:

```
Toy> complexity F == s (s (s z)), isContradiction F == true
     { F -> (neg (disy p (iff p p))) }
     Elapsed time: 94 ms.
more solutions (y/n/d) [y]?
     { F -> (neg (disy p (iff q q))) }
     Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
```

In the same way we can produce contradictions or contingencies.

A.5 Programming with Equality and Disequality Constraints

One of the most interesting characteristics of $\mathcal{TOY}$ is that of dealing with *equality* and *disequality* constraints, as well as *arithmetic* constraints over real numbers. We show the power of this feature by means of some examples.

A.5.1 The Cardinal of a List (`cardinal.toy`)

Next example shows that $\mathcal{TOY}$ can represent solutions by using *equality* and *disequality* constraints. Function *card* determines the cardinal of a given list regarded as a set. It relies on the boolean function member that specifies whether an element occurs in a list or not.

```
card ::  [A] -> int
card [] = 0
card [X|Xs] = if member X Xs
              then card Xs
              else 1 + (card Xs)

member :: A -> [A] -> bool
member X [] = 0
member X [Y|Ys] = if X==Y
                  then true
                  else member X Ys
```

Let's consider, as an example:

```
Toy> card [1, 0, 1] == N
      { N -> 2 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

A more involved example, using logic variables, shows the convenience of equality and disequality constraints in solutions:

```
Toy> card  [X, Y, X, Z] == N
      { Y -> X,
        Z -> X,
        N -> 1 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { Y -> X,
        N -> 2 }
      { X /= Z }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { Z -> Y,
        N -> 2 }
      { Y /= X }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { Z -> X,
        N -> 2 }
      { X /= Y }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { N -> 3 }
      { X /= Z,
        X /= Y,
        Y /= Z }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

A.6 Programming with Real Constraints

In order to compile real constraint programs in $\mathcal{TOY}$ the user must first activate the constraint solver over reals by means of the command `/cflpr`. Recall the explanations in Section 1.5.5.

A.6.1 Defining Regions of the Plane (`regions.toy`)

The program below shows the usefulness of using functions with arithmetic constraints over real numbers. It contains some stuff for dealing with regions (subsets) of the plane. Regions are sets of points represented by their characteristic function (see the type region below). This is a typical approach in functional programming. However, the novelty is that constraints provide a much more flexible way of using and defining functions.

```
include "misc.toy"

type point  = (real,real)
type region = point -> bool

infixr 50 <<-

(<<-) :: point -> region -> bool
P <<- R = R P

rectangle :: point -> point -> region
rectangle (A,B) (C,D) (X,Y) =
  (X >= A)  /\ (X <= C)  /\ (Y >= B ) /\ (Y <= D)


circle :: point ->   real  ->  region
circle (A,B) R (X,Y) = (X-A)*(X-A)+(Y-B)*(Y-B) <= R*R

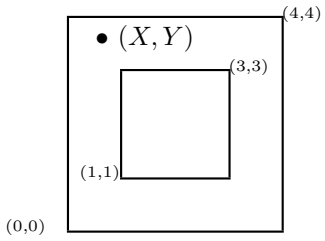
outside :: region -> region
intersect, union :: region -> region -> region
outside   R  P = not (P <<- R)
intersect R1 R2 P = P <<- R1  /\ P <<- R2
union     R1 R2 P = P <<- R1  \/ P <<- R2
```

The operator $<<-$ is defined in order to know whether a point belongs to a region. It simply applies the characteristic function to the point. Then the characteristic functions for circles (given its radius and its center) and rectangles (given its left lower corner and its right upper corner) are defined. Finally some operations over regions are defined: the outside of a region, and the intersection and union of two regions. For instance, a point P belongs to the intesection of two regions R and R' if it belongs to R *and* (operator /\) it belongs to R'. A simple goal could be:

```
Toy(R)> (0.5, 0.5) <<- (circle (0,0) 1)
      yes
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no
```

```
      Elapsed time: 0 ms.
```

asking whether the point $(0.5, 0.5)$ belongs to the circle of center $(0, 0)$ and radius 1. A more interesting (and involved) goal can ask for the points (X, Y) belonging to the region between two rectangles:



This area is the intersection between the big rectangle and the outside of the small rectangle:

```
Toy(R)> (X,Y) <<- intersect (rectangle (0,0) (4,4))
                             (outside (rectangle (1,1) (3,3)))
      { Y>3.0,
        Y=<4.0,
        X>=1.0,
        X=<3.0 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { X>=1.0,
        X=<3.0,
        Y>=-0.0,
        Y<1.0 }
      Elapsed time: 16 ms.
more solutions (y/n/d) [y]?
      { X>3.0,
        X=<4.0,
        Y>=-0.0,
        Y=<4.0 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { Y>=-0.0,
        Y=<4.0,
        X>=-0.0,
        X<1.0 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      no
      Elapsed time: 0 ms.
```

The answers are given in terms of arithmetic constraints for the coordinates of the point (X, Y). The union of the areas (rectangles) given by the four solutions constitute the intersection of the two rectangles.

A.7 Programming with Finite Domain Constraints

In order to compile real constraint programs in $\mathcal{TOY}$ the user must first activate the constraint solver over finite domains by means of the command `/cflpfd`. In addition, it must contain the directive

```
include "cflpfd.toy"
```

as the file `cflpfd.toy` contains the definitions of data types, constraints and functions related to finite domains.

A.7.1 A Colouring Problem (`colour.toy`)

We want to solve the classical map colouring problem. Consider the simple map shown in Figure A.2. To solve this problem, we have to specify that some countries have different colors by using the constraint `all_different L`.

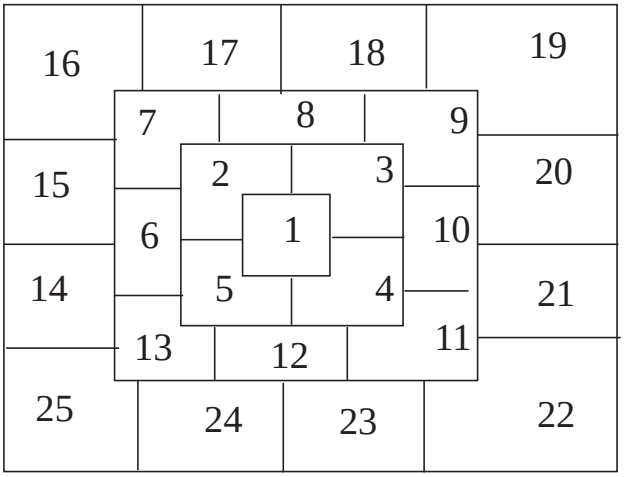


Figure A.2: A Map Colouring Problem

The code to solve this problem is shown below:

```
include "cflpfd.toy"

colour :: [int] -> bool
colour [I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
        I14, I15, I16, I17, I18, I19, I20, I21, I22, I23, I24, I25] = true <==
      domain [I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13, I14,
              I15, I16, I17, I18, I19, I20, I21, I22, I23, I24, I25] 1 4,
      all_different [I1, I2],all_different [I1, I3],
      all_different [I1, I4],all_different [I1, I5],
      all_different [I2, I3],all_different [I2, I5],
```

```
      all_different [I2, I6],all_different [I2, I7],
      all_different [I2, I8],all_different [I3, I4],
      all_different [I3, I8],all_different [I3, I9],
      all_different [I3, I10],all_different [I4, I5],
      all_different [I4, I10],all_different [I4, I11],
      all_different [I4, I12],all_different [I5, I6],
      all_different [I5, I12],all_different [I5, I13],
      all_different [I6, I7],all_different [I6, I13],
      all_different [I6, I14],all_different [I6, I15],
      all_different [I7, I8],all_different [I7, I15],
      all_different [I7, I16],all_different [I7, I17],
      all_different [I8, I9],all_different [I8, I17],
      all_different [I8, I18],all_different [I9, I10],
      all_different [I9, I18],all_different [I9, I19],
      all_different [I9, I20],all_different [I10, I11],
      all_different [I10, I20],all_different [I10, I21],
      all_different [I11, I12],all_different [I11, I21],
      all_different [I11, I22],all_different [I11, I23],
      all_different [I12, I13], all_different [I12, I23],
      all_different [I12, I24], all_different [I13, I24],
      all_different [I13, I25],all_different [I13, I14],
      labeling [ff] [I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
                 I14, I15, I16, I17, I18, I19, I20, I21, I22, I23, I24, I25]
```

Below, we show an example of constraint solving with two different solution to this problem:

```
Toy(FD)> colour L
      { L ->  [ 1, 2, 3, 2, 3, 1, 3, 1, 2, 1, 3, 1, 2,
                3, 2, 1, 2, 3, 1, 3, 2, 1, 2, 3, 1 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      { L ->  [ 1, 2, 3, 2, 3, 1, 3, 1, 2, 1, 3, 1, 2,
                3, 2, 1, 2, 3, 1, 3, 2, 1, 2, 3, 3 ] }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?

 ... up to all the possible solutions
```

A.7.2 Linear Equations (`eq10.toy` and `eq20.toy`)

This classical problem solves a set of 10 linear equations with seven finite domain variables ranging in the interval [0,10].

```
include "cflpfd.toy"
```

```
equation10 :: [labelingType] -> [int] -> bool
equation10 Label LD =
true <==
  LD == [X1,X2,X3,X4,X5,X6,X7],
  domain LD 0 10,
  0 #+ 98527 #* X1 #+ 34588 #* X2 #+ 5872 #* X3 #+ 59422 #* X5 #+ 65159 #* X7
  #= 1547604 #+ 30704 #* X4 #+ 29649 #* X6,
  0 #+ 98957 #* X2 #+ 83634 #* X3 #+ 69966 #* X4 #+ 62038 #* X5 #+ 37164 #* X6
     #+ 85413 #* X7
  #= 1823553 #+ 93989 #* X1,
  900032 #+ 10949 #* X1 #+ 77761 #* X2 #+ 67052 #* X5
  #= 0 #+ 80197 #* X3 #+ 61944 #* X4 #+ 92964 #* X6 #+ 44550 #* X7,
  0 #+ 73947 #* X1 #+ 84391 #* X3 #+ 81310 #* X5
  #= 1164380 #+ 96253 #* X2 #+ 44247 #* X4 #+ 70582 #* X6 #+ 33054 #* X7,
  0 #+ 13057 #* X3 #+ 42253 #* X4 #+ 77527 #* X5 #+ 96552 #* X7
  #= 1185471 #+ 60152 #* X1 #+ 21103 #* X2 #+ 97932 #* X6,
  1394152 #+ 66920 #* X1 #+ 55679 #* X4
  #= 0 #+ 64234 #* X2 #+ 65337 #* X3 #+ 45581 #* X5 #+ 67707 #* X6 #+ 98038 #* X7,
  0 #+ 68550 #* X1 #+ 27886 #* X2 #+ 31716 #* X3 #+ 73597 #* X4 #+ 38835 #* X7
  #= 279091 #+ 88963 #* X5 #+ 76391 #* X6,
  0 #+ 76132 #* X2 #+ 71860 #* X3 #+ 22770 #* X4 #+ 68211 #* X5 #+ 78587 #* X6
  #= 480923 #+ 48224 #* X1 #+ 82817 #* X7,
  519878 #+ 94198 #* X2 #+ 87234 #* X3 #+ 37498 #* X4
  #= 0 #+ 71583 #* X1 #+ 25728 #* X5 #+ 25495 #* X6 #+ 70023 #* X7,
  361921 #+ 78693 #* X1 #+ 38592 #* X5 #+ 38478 #* X6
  #= 0 #+ 94129 #* X2 #+ 43188 #* X3 #+ 82528 #* X4 #+ 69025 #* X7,
        labeling Label LD
```

The next goal uses the first-fail labeling strategy:

```
Toy(FD)> equation10 [ff] L
      { L ->  [ 6, 0, 8, 4, 9, 3, 9 ] }
      Elapsed time: 32 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 15 ms.
```

Another version considers 20 linear equations with the same number of finite domain variables and related domains.

```
include "cflpfd.toy"

equation20 :: [labelingType] -> [int] -> bool equation20 Label LD =
true <==
```

```
LD == [X1,X2,X3,X4,X5,X6,X7],
domain LD 0 10 ,
876370 #+ 16105 #* X1 #+ 6704 #* X3 #+ 68610 #* X6
#= 0 #+ 62397 #* X2 #+ 43340 #* X4 #+ 95100 #* X5 #+ 58301 #* X7,
533909 #+ 96722 #* X5
#= 0 #+ 51637 #* X1 #+ 67761 #* X2 #+ 95951 #* X3 #+ 3834 #* X4 #+ 59190 #* X6
   #+ 15280 #* X7,
915683 #+ 34121 #* X2 #+ 33488 #* X7
#= 0 #+ 1671 #* X1 #+ 10763 #* X3 #+ 80609 #* X4 #+ 42532 #* X5 #+ 93520 #* X6,
129768 #+ 11119 #* X2 #+ 38875 #* X4 #+ 14413 #* X5 #+ 29234 #* X6
#= 0 #+ 71202 #* X1 #+ 73017 #* X3 #+ 72370 #* X7,
752447 #+ 58412 #* X2
#= 0 #+ 8874 #* X1 #+ 73947 #* X3 #+ 17147 #* X4 #+ 62335 #* X5 #+ 16005 #* X6
   #+ 8632 #* X7,
90614 #+ 18810 #* X3 #+ 48219 #* X4 #+ 79785 #* X7
#= 0 #+ 85268 #* X1 #+ 54180 #* X2 #+ 6013 #* X5 #+ 78169 #* X6,
1198280 #+ 45086 #* X1 #+ 4578 #* X3
#= 0 #+ 51830 #* X2 #+ 96120 #* X4 #+ 21231 #* X5 #+ 97919 #* X6 #+ 65651 #* X7,
18465 #+ 64919 #* X1 #+ 59624 #* X4 #+ 75542 #* X5 #+ 47935 #* X7
#= 0 #+ 80460 #* X2 #+ 90840 #* X3 #+ 25145 #* X6,
0 #+ 43525 #* X2 #+ 92298 #* X3 #+ 58630 #* X4 #+ 92590 #* X5
#= 1503588 #+ 43277 #* X1 #+ 9372 #* X6 #+ 60227 #* X7,
0 #+ 47385 #* X2 #+ 97715 #* X3 #+ 69028 #* X5 #+ 76212 #* X6
#= 1244857 #+ 16835 #* X1 #+ 12640 #* X4 #+ 81102 #* X7,
0 #+ 31227 #* X2 #+ 93951 #* X3 #+ 73889 #* X4 #+ 81526 #* X5 #+ 68026 #* X7
#= 1410723 #+ 60301 #* X1 #+ 72702 #* X6,
0 #+ 94016 #* X1 #+ 35961 #* X3 #+ 66597 #* X4
#= 25334 #+ 82071 #* X2 #+ 30705 #* X5 #+ 44404 #* X6 #+ 38304 #* X7,
0 #+ 84750 #* X2 #+ 21239 #* X4 #+ 81675 #* X5
#= 277271 #+ 67456 #* X1 #+ 51553 #* X3 #+ 99395 #* X6 #+ 4254 #* X7,
0 #+ 29958 #* X2 #+ 57308 #* X3 #+ 48789 #* X4 #+ 4657 #* X6 #+ 34539 #* X7
#= 249912 #+ 85698 #* X1 #+ 78219 #* X5,
0 #+ 85176 #* X1 #+ 57898 #* X4 #+ 15883 #* X5 #+ 50547 #* X6 #+ 83287 #* X7
#= 373854 #+ 95332 #* X2 #+ 1268 #* X3,
0 #+ 87758 #* X2 #+ 19346 #* X4 #+ 70072 #* X5 #+ 44529 #* X7
#= 740061 #+ 10343 #* X1 #+ 11782 #* X3 #+ 36991 #* X6,
0 #+ 49149 #* X1 #+ 52871 #* X2 #+ 56728 #* X4
#= 146074 #+ 7132 #* X3 #+ 33576 #* X5 #+ 49530 #* X6 #+ 62089 #* X7,
0 #+ 29475 #* X2 #+ 34421 #* X3 #+ 62646 #* X5 #+ 29278 #* X6
#= 251591 #+ 60113 #* X1 #+ 76870 #* X4 #+ 15212 #* X7,
22167 #+ 29101 #* X2 #+ 5513 #* X3 #+ 21219 #* X4
#= 0 #+ 87059 #* X1 #+ 22128 #* X5 #+ 7276 #* X6 #+ 57308 #* X7,
821228 #+ 76706 #* X1 #+ 48614 #* X6 #+ 41906 #* X7
#= 0 #+ 98205 #* X2 #+ 23445 #* X3 #+ 67921 #* X4 #+ 24111 #* X5,
```

```
  labeling Label LD
```

And a goal for this program:

```
Toy(FD)> equation20 [ff] L
      { L ->  [ 1, 4, 6, 6, 6, 3, 1 ] }
      Elapsed time: 16 ms.
more solutions (y/n/d) [y]?
      no.
      Elapsed time: 47 ms.
```

A.7.3 DNA Sequencing (dna.toy)

In this section, we show a simplified version of restriction site mapping (RSM) taken from [17]. A DNA sequence is a finite string over the elements {A,C,G,T}. An enzyme partitions a DNA sequence into certain fragments. The problem consists of reconstructing the original DNA sequence from the fragments and other information taken from experiments. To keep the problem concise, we consider a simplification of this problem, which only deals with the length of the fragments, instead of the fragments themselves.
Consider the use of two enzymes. The first enzyme partitions the DNA sequence into $\mathtt{A}_1,\ldots,\mathtt{A}_N$ and the second into $\mathtt{B}_1,\ldots,\mathtt{B}_M$. A simultaneous use of the two enzymes also produces a partition into $\mathtt{D}_1,\ldots,\mathtt{D}_K$, which corresponds to the combination of the previous two partitions, that is: $\forall$i $\exists$j: $\mathtt{A}_1\ldots\mathtt{A}_i = \mathtt{D}_1\ldots\mathtt{D}_j \wedge \forall$i $\exists$j: $\mathtt{B}_1\ldots\mathtt{B}_i = \mathtt{D}_1\ldots\mathtt{D}_j$ and, conversely, $\forall$j $\exists$i: $\mathtt{D}_1\ldots\mathtt{D}_j = \mathtt{A}_1\ldots\mathtt{A}_i \vee \mathtt{D}_1\ldots\mathtt{D}_j = \mathtt{B}_1\ldots\mathtt{B}_i$, where '$\mathtt{A}_1\ldots\mathtt{A}_i$' denotes the sequence of fragments $\mathtt{A}_1$ to $\mathtt{A}_i$, and '=' denotes syntactic equality.
Let $\mathtt{a}_i$ ($\mathtt{b}_i$ and $\mathtt{d}_i$, resp.) denote the length of $\mathtt{A}_i$ ($\mathtt{B}_i$ and $\mathtt{D}_i$, resp.). Let $\mathbf{a_i}$ denote the subsequence $a_1 \ldots a_i$, $1 \leq \mathtt{i} \leq \mathtt{N}$ (and similarly for $\mathbf{b_i}$ and $\mathbf{d_i}$)
The problem is stated as follows: given the multisets $a = \{\mathtt{a}_1,\ldots,\mathtt{a}_N\}$, $b = \{\mathtt{b}_1,\ldots,\mathtt{b}_N\}$, and $d = \{\mathtt{d}_1,\ldots,\mathtt{d}_N\}$, construct the sequences $\mathbf{a}_N = \mathtt{a}_1\ldots\mathtt{a}_N$, $\mathbf{b}_M = \mathtt{b}_1\ldots\mathtt{b}_M$, and $\mathbf{d}_K = \mathtt{d}_1\ldots\mathtt{d}_K$.
The algorithm to solve this problem generates $\mathtt{d}_1$, $\mathtt{d}_2$, ... in order, and extends the partitions for **a** and **b** using the following invariant property which can be obtained from the problem definition above. Either

- $\mathtt{d}_k$ is aligned with $\mathtt{a}_i$, that is, $\mathtt{d}_1 + \cdots + \mathtt{d}_k = \mathtt{a}_1 + \cdots + \mathtt{a}_i$, or
- $\mathtt{d}_k$ is aligned with $\mathtt{b}_j$, but not with $\mathtt{a}_i$, (for simplicity, we assume we never have all three partitions aligned except at the beginning and at the end), that is, $\mathtt{d}_1 + \cdots + \mathtt{d}_k = \mathtt{a}_1 + \cdots + \mathtt{a}_i$.

The following Boolean function `solve/6` takes three input lists representing a, b, and d, in its three first arguments respectively. The output represents the possibilities to construct d from the fragments taken from a, and b.

```
include "cflpfd.toy"

solve :: [int] -> [int] -> [int] -> [int] -> [int] -> [int] -> bool
solve A B D [AF|MA] [BF|MB] [DF|MD] :-
```

```
    choose_initial A B D AF BF DF A2 B2 D2,
    rsm A2 B2 D2 AF BF DF MA MB MD

rsm :: [int] -> [int] -> [int] -> int -> int -> int -> [int] ->
[int] -> [int] -> bool rsm [] [] [] LenA LenB LenD [] [] [] = true

rsm A B D LenA LenB LenA [Ai|MA] MB [Dk|MD] :-
    LenA #< LenB,
    Dk #<= LenB #- LenA, Ai #>= Dk,
    choose Ai A == A2, choose Dk D == D2,
    NLenA #= LenA #+ Ai, NLenD #= LenA #+ Dk,
    rsm A2 B D2 NLenA LenB NLenD MA MB MD

rsm A B D LenA LenB LenB MA [Bj|MB] [Dk|MD] :-
    LenB #< LenA,
    Dk #<= LenA #- LenB, Bj #>= Dk,
    choose Dk D == D2, choose Bj B == B2,
    NLenB #= LenB #+ Bj, NLenD #= LenB #+ Dk,
    rsm A B2 D2 LenA NLenB NLenD MA MB MD

choose_initial :: [int] -> [int] -> [int] -> int -> int -> int ->
[int] -> [int] -> [int] -> bool choose_initial A B D AF BF DF A2 B2
D2 :-
    choose AF A == A2,
    choose BF B == B2,
    choose DF D == D2

choose :: int -> [int] -> [int] choose X [] = [] choose Ai [Ai|A2] =
A2 choose Ai [A1, A2|A] = [A1|choose Ai [A2|A]]
```

For instance, one goal for this program could be:

```
Toy(FD)> solve [3,2,4,5,9] [7,8] [3,2,3,4,2,2,3,4] L1 L2 L3
      { L1 -> [ 4, 2, 5, 9, 3 ],
        L2 -> [ 8, 7, 3, _A ],
        L3 -> [ 4, 2, 2, 3, 4, 3, 2, 3 ] }
      { 18 #+ _A #= _B #+ 20,
        _B in 3..sup,
        _A in 5..sup }
      Elapsed time: 31 ms.
sol.1, more solutions (y/n/d/a) [y]? n
```

which means that `L1` (`L2` resp.) constructs `L3` by aligning the fragments as the following table indicates:

L1	L3	L2	L3
4	4	8	4,2,2
2	2	7	3,4
5	2,3	3	3
9	4,3,2	5	2,3
3	3		

In the program code, `rsm/9` provides the choice of partitioning with either one of the two available enzymes. The last three arguments hold the length of the subsequences found so far. The function `choose_initial/9` chooses the first fragment and the first call to `rsm` is made with this invariant holding. Finally, the procedure `choose/2` deletes some element from the given list and returns the resultant list.
Note that the Boolean functions `solve/6`, `rsm/9`, and `choose_initial/9` have been written in a Prolog-like fashion, thanks to the syntactic sugaring allowed in our system, whereas `choose/2` has been written as a function which returns the list resulting from deleting an element of its input list.

A.7.4 A Scheduling Problem (`scheduling.toy`)

Here, we consider the problem of scheduling tasks that require resources to complete, and have to fulfill precedence constraints[1]. Figure A.3 shows a precedence graph for six tasks which are labelled as tX^{Y}_{mZ}, where X stands for the identifier of a task t, Y for its time to complete (duration), and Z for the identifier of a machine m (a resource needed for performing task tX).

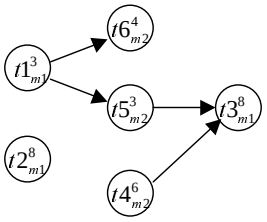


Figure A.3: Precedence Graph.

The following program (included in the distribution in the directory Examples in the file `scheduling.toy`) models the posed scheduling problem. Observe in the syntax that function arguments are not enclosed in parentheses to allow higher order applications. Also, syntactic sugar is provided for expressing Boolean functions à la Prolog. The rules that define a function follow its type declaration. The type declaration consists of the types for each argument and for the result separated by `->`. Lists adhere to the syntax as Prolog lists and **int** is a predefined type for the integers. Note also functional applications in arguments, such as `(End-D)` in the second rule defining `horizon`. (Logic) Variables start with uppercase, whereas the remaining symbols start with lowercase.

[1] Adapted from [27].

```
include "cflpfd.toy"
include "misc.toy"

data taskName = t1 | t2 | t3 | t4 | t5 | t6
data resourceName = m1 | m2
type durationType = int
type startType = int
type precedencesType = [taskName]
type resourcesType = [resourceName]
type task = (taskName, durationType, precedencesType, resourcesType, startType)

start :: task -> int
start (Name, Duration, Precedences, Resources, Start) = Start

duration :: task -> int
duration (Name, Duration, Precedences, Resources, Start) = Duration

schedule :: [task] -> int -> int -> bool
schedule TL Start End = true <== horizon TL Start End, scheduleTasks TL TL

horizon :: [task] -> int -> int -> bool
horizon [] S E = true
horizon [(N, D, P, R, S)|Ts] Start End = true <==
        domain [S] Start (End#-D),
        horizon Ts Start End

scheduleTasks :: [task] -> [task] -> bool
scheduleTasks [] TL = true
scheduleTasks [(N, D, P, R, S)|Ts] TL = true <==
          precedeList (N, D, P, R, S) P TL,
          requireList (N, D, P, R, S) R TL,
          scheduleTasks Ts TL

precedeList :: task -> [taskName] -> [task] -> bool
precedeList T [] TL = true
precedeList T1 [TN|TNs] TL = true <== member (TN, D, P, R, S) TL,
                          precedes T1 (TN, D, P, R, S),
                          precedeList T1 TNs TL

precedes :: task -> task -> bool
precedes T1 T2 = true <==  ST1 == start T1,
                           DT1 == duration T1,
                           ST2 == start T2,
                           ST1 #+ DT1 #<= ST2

requireList :: task -> [resourceName] -> [task] -> bool
requireList T [] TL = true
requireList T [R|Rs] TL = true <== requires T R TL, requireList T Rs TL
```

```
requires :: task -> resourceName -> [task] -> bool
requires T R [] = true
requires (N1, D1, P1, R1, S1) R [(N2, D2, P2, R2, S2)|Ts] = true <== N1 /= N2,
            member R R2,
            noOverlaps (N1, D1, P1, R1, S1) (N2, D2, P2, R2, S2),
            requires (N1, D1, P1, R1, S1) R Ts
requires T1 R [T2|Ts] = true <== requires T1 R Ts

noOverlaps :: task -> task -> bool
noOverlaps T1 T2 = true <== precedes T1 T2
noOverlaps T1 T2 = true <== precedes T2 T1
```

A task is modelled (via the type `task`) as a 5-tuple which holds its name, duration, list of precedence tasks, list of required resources, and the start time. Two functions for accessing the start time and duration of a task are provided (`start` and `duration`, respectively) that are used by the function `precedes`. This last function imposes the precedence constraint between two tasks. The function `requireList` imposes the constraints for tasks requiring resources, i.e., if two different tasks require the same resource, they cannot overlap. The function `noOverlaps` states that for two non overlapping tasks $t1$ and $t2$, either $t1$ precedes $t2$ or vice versa. The main function is `schedule`, which takes three arguments: a list of tasks to be scheduled, the scheduling start time, and the maximum scheduling final time. These last two arguments represent the time window that has to fit the scheduling. The time window is imposed via domain pruning for each task's start time (a task cannot start at a time so that its duration makes its end time greater than the end time of the window; this is imposed with the function `horizon`). The function `scheduleTasks` imposes the precedence and requirement constraints for all of the tasks in the scheduling. Precedence constraints and requirement constraints are imposed by the functions `precedeList` and `requireList`, respectively.
With this model, we can declare for example a function that defines the solution to the problem.

```
sched :: startType -> startType -> startType -> startType ->
         startType -> startType -> bool
sched S1 S2 S3 S4 S5 S6 :-
    Tasks == [(t1,3,[t5,t6],[m1],S1),
              (t2,8,[],[m1],S2),
              (t3,8,[],[m1],S3),
              (t4,6,[t3],[m2],S4),
              (t5,3,[t3],[m2],S5),
              (t6,4,[],[m2],S6)],
    schedule Tasks 1 20,
    labeling [ff] [S1,S2,S3,S4,S5,S6]
```

where `Tasks` defines the set of tasks. Observe that the problem for a possible scheduling is limited to time window (1,20) by the goal `schedule Tasks 1 20`. An example of goal solving is given next:

```
Toy(FD)> sched S1 S2 S3 S4 S5 S6
```

```
      { S1 -> 1,
        S2 -> 4,
        S3 -> 12,
        S4 -> 1,
        S5 -> 7,
        S6 -> 10 }
      Elapsed time: 31 ms.
more solutions (y/n/d) [y]?
      { S1 -> 1,
        S2 -> 4,
        S3 -> 12,
        S4 -> 1,
        S5 -> 7,
        S6 -> 11 }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]? n
```

A.7.5 A Hardware Design Problem

A more interesting example comes from the hardware arena. In this setting, many constrained optimization problems arise in the design of both sequential and combinational circuits as well as the interconnection routing between components. Constraint programming has been shown to effectively attack these problems. In particular, the interconnection routing problem (one of the major tasks in the physical design of very large scale integration - VLSI - circuits) have been solved with constraint logic programming [40].
For the sake of conciseness and clarity, we focus on a constraint combinational hardware problem at the logical level but adding constraints about the physical factors the circuit has to meet. This problem will show some of the nice features of $\mathcal{TOY}$ for specifying issues such as behavior, topology and physical factors.
Our problem can be stated as follows. Given a set of gates and modules, a switching function, and the problem parameters maximum circuit area, power dissipation, cost, and delay (dynamic behavior), the problem consists of finding possible topologies based on the given gates and modules so that a switching function and constraint physical factors are met. In order to have a manageable example, we restrict ourselves to the logical gates NOT, AND, and OR. We also consider circuits with three inputs and one output, and the physical factors aforementioned. We suppose also the following problem parameters:

Gate	Area	Power	Cost	Delay
NOT	1	1	1	1
AND	2	2	1	1
OR	2	2	2	2

In the sequel we will introduce the problem by first considering the features $\mathcal{TOY}$ offers for specifying logical circuits, what are its weaknesses, and how they can effectively be solved with the integration of constraints in $\mathcal{TOY}(\mathcal{FD})$.

FLP Simple Circuits (`circuit.toy`)

With this example we show the FLP approach that can be followed for specifying the problem stated above. We use patterns to provide an *intensional* representation of functions. The alias `behavior` is used for representing the type `bool` → `bool` → `bool` → `bool`. Functions of this type are intended to represent simple circuits which receive three Boolean inputs and return a Boolean output. Given the Boolean functions `not`, `and`, and `or` defined elsewhere, we specify three-input, one-output simple circuits as follows.

```
i0 :: behavior
i0 I2 I1 I0 = I0

i1 :: behavior
i1 I2 I1 I0 = I1

i2 :: behavior
i2 I2 I1 I0 = I2

notGate :: behavior -> behavior
notGate B I2 I1 I0 = not (B I2 I1 I0)

andGate, orGate :: behavior -> behavior -> behavior
andGate B1 B2 I2 I1 I0 = and (B1 I2 I1 I0) (B2 I2 I1 I0)
orGate  B1 B2 I2 I1 I0 = or  (B1 I2 I1 I0) (B2 I2 I1 I0)
```

Functions `i0`, `i1`, and `i2` represent inputs to the circuits, that is, the minimal circuit which just copies one of the inputs to the output. (In fact, this can be thought as a fixed multiplexer - selector.) They are combinatorial modules as depicted in Figure A.4. The function `notGate` outputs a Boolean value which is the result of applying the NOT gate to the output of a circuit of three inputs. In turn, functions `andGate` and `orGate` output a Boolean value which is the result of applying the AND and OR gates, respectively, to the outputs of three-input circuits (see Figure A.4).
These functions can be used in a higher-order fashion just to generate or match topologies. In particular, the higher-order functions `notGate`, `andGate` and `orGate` take behaviors as parameters and build new behaviors, corresponding to the logical gates NOT, AND and OR. For instance, the multiplexer depicted in Figure A.5 can be represented by the following pattern:

```
orGate (andGate i0 (notGate i2)) (andGate i1 i2).
```

This first-class citizen higher-order pattern can be used for many purposes. For instance, it can be compared to another pattern or it can be applied to actual values for its inputs in order to compute the circuit output. So, with the previous pattern, the goal:

```
Toy(FD) > P == orGate (andGate i0 (notGate i2)) (andGate i1 i2),
          P true false true == O
```

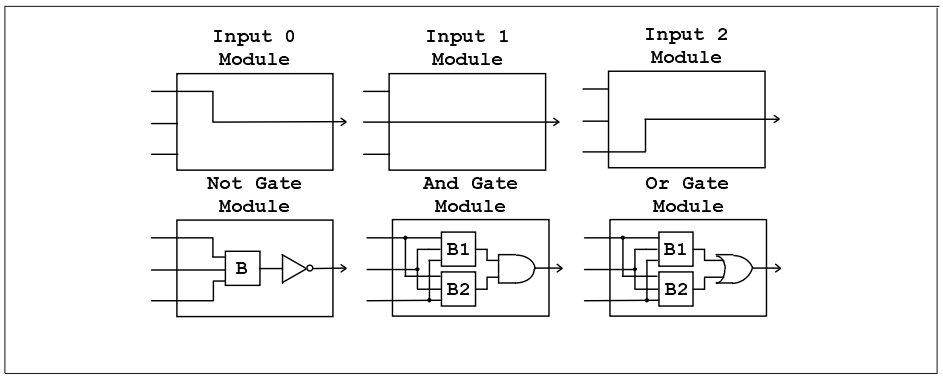


Figure A.4: Basic Modules.

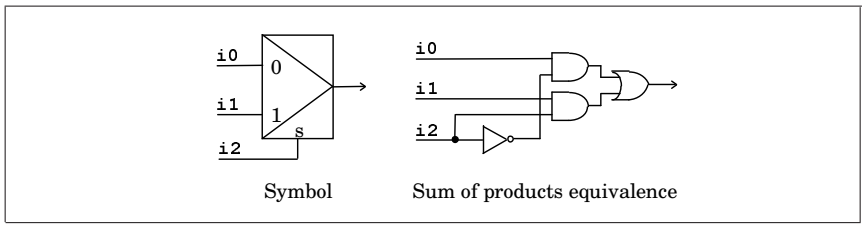


Figure A.5: Two-Input Multiplexer Circuit.

is evaluated to `true` and produces the substitution `O == false`. The rules that define the behavior can be used to generate circuits, which can be restricted to satisfy some conditions. If we use the standard arithmetics, we could define the following set of rules for computing or limiting the power dissipation.

```
power :: behavior -> int
power i0 = 0
power i1 = 0
power i2 = 0
power (notGate C) = notGatePower + (power C)
power (andGate C1 C2) = andGatePower + (power C1) + (power C2)
power (orGate C1 C2) = orGatePower + (power C1) + (power C2)
```

Then, we can submit the goal `power B == P, P < maxPower` (provided the function `maxPower` acts as a problem parameter that returns just the maximum power allowed for the circuit) in which the function `power` is used as a behavior generator[2]. As outcome, we get several solutions
`B==i0, P==0`,
`B==i1, P==0`,
`B==i2, P==0`,

[2]Equivalently and more concisely, `power B` $<$ `maxPower` could be submitted, but doing so we make the power unobservable.

`B==not i0,P==1,`
`...,`
`B==not (not i0),P==2,....`
Declaratively, it is fine; but our operational semantics requires a head normal form for the application of the arithmetic operand `+`. This implies that we reach no more solutions beyond `not (...(not i0)...)`, because the application of the fourth rule of `power` yields to an infinite computation. This drawback is solved by resorting to Peano's arithmetics, that is:

```
data nat = z | s nat

plus :: nat -> nat -> nat
plus z Y = Y
plus (s X) Y = s (plus X Y)

less :: nat -> nat -> bool
less (s X) (s Y) = less X Y
less z (s X) = true

power' :: behavior -> nat
power' i0 = z
power' i1 = z
power' i2 = z
power' (notGate C)     = plus notGatePower (power' C)
power' (andGate C1 C2) = plus andGatePower (plus (power' C1) (power' C2))
power' (orGate C1 C2)  = plus orGatePower  (plus (power' C1) (power' C2))
```

So, we can submit the goal `less (power' P) (s (s (s z)))`, where we have written down explicitly the maximum power (3 power units).
With this the second approach we get a more awkward representation due to the use of successor arithmetics. The first approach to express this problem is indeed more declarative than the second one, but we get non-termination. FD constraints can be profitably applied to the representation of this problem as we show in the next example.

Simple Circuits with FD Constraints (`circuitFD.pl`)

As for any constraint problem, modelling can be started by identifying the FD constraint variables. Recalling the problem specification, circuit limitations refer to area, power dissipation, cost, and delay. Provided we can choose finite units to represent these factors, we choose them as problem variables. A circuit can therefore be represented by the 4-tuple state ⟨area, power, cost, delay⟩. The idea to formulate the problem consists of attaching this state to an ongoing circuit so that state variables reflect the current state of the circuit *during* its generation. By contrast with the first example, we do not "generate" and then "test", but we "test" when "generating", so that we can find failure in advance. A domain variable has a domain attached indicating the set of possible assignments to the variable. This domain can be reduced during the computation. Since domain variables are constrained by limiting factors, during the generation of the

circuit a domain may become empty. This event prunes the search space avoiding to explore a branch known to yield no solution. Let's firstly focus on the area factor. The following function generates a circuit characterized by its state variables.

```
type area, power, cost, delay = int
type state = (area, power, cost, delay)
type circuit = (behavior, state)
genCir :: state -> circuit
genCir (A, P, C, D) = (i0,(A, P, C, D))
genCir (A, P, C, D) = (i1,(A, P, C, D))
genCir (A, P, C, D) = (i2,(A, P, C, D))
genCir (A, P, C, D) = (notGate B, (A, P, C, D)) <==
             domain [A] ((fd_min A) + notGateArea)  (fd_max A),
             genCir (A, P, C, D) == (B, (A, P, C, D))
genCir (A, P, C, D) = (andGate B1 B2, (A, P, C, D)) <==
             domain [A] ((fd_min A) + andGateArea)  (fd_max A),
             genCir (A, P, C, D) == (B1, (A, P, C, D)),
             genCir (A, P, C, D) == (B2, (A, P, C, D))
genCir (A, P, C, D) = (orGate B1 B2, (A, P, C, D)) <==
             domain [A] ((fd_min A) + orGateArea)  (fd_max A),
             genCir (A, P, C, D) == (B1, (A, P, C, D)),
             genCir (A, P, C, D) == (B2, (A, P, C, D))
```

The function `genCir` has an argument to hold the circuit state and returns a circuit characterized by a behavior and a state. (Note that we can avoid the use of the state tuple as a parameter, since it is included in the result.) The template of this function is like the previous example. The difference lies in that we perform domain pruning during circuit generation with the membership constraint `domain`, so that each time a rule is selected, the domain variable representing area is reduced in the size of the gate selected by the operational mechanism. For instance, the circuit area domain is reduced in a number of `notGateArea` when the rule for `notGate` has been selected. For domain reduction we use the reflection functions `fd_min` and `fd_max`. This approach allows us to submit the following goal:

```
domain [Area] 0 maxArea, genCir (Area, Power, Cost, Delay) == Circuit
```

which initially sets the possible range of area between 0 and the problem parameter area expressed by the function `maxArea`, and then generates a `Circuit`. Recall that testing is performed during search space exploration, so that termination is ensured because the add operation is monotonic. The mechanism which allows this "test" when "generating" is the set of propagators, which are concurrent processes that are triggered whenever a domain variable is changed (pruned). The state variable delay is more involved since one cannot simply add the delay of each function at each generation step. The delay of a circuit is related to the maximum number of levels an input signal has to traverse until it reaches the output. This is to say that we cannot use a single domain variable for describing the delay. Therefore, considering a module with several inputs, we must compute the delay at its output by computing the maximum delays

from its inputs and adding the module delay. So, we use new fresh variables for the inputs of a module being generated and assign the maximum delay to the output delay. This solution is depicted in the following function:

```
genCirDelay :: state -> delay -> circuit
genCirDelay (A, P, C, D) Dout = (i0, (A, P,C, D))
genCirDelay (A, P, C, D) Dout = (i1, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (i2, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (notGate B, (A, P, C, D)) <==
     domain [Dout] ((fd_min Dout) + notGateDelay) (fd_max Dout),
     genCirDelay (A, P, C, D) Dout == (B, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (andGate B1 B2, (A, P, C, D)) <==
     domain [Din1, Din2] ((fd_min Dout) + andGateDelay)(fd_max Dout),
     genCirDelay (A, P, C, D) Din1 == (B1, (A, P, C, D)),
     genCirDelay (A, P, C, D) Din2 == (B2, (A, P, C, D)),
     domain [Dout] (maximum (fd_min Din1)(fd_min Din2)) (fd_max Dout)
genCirDelay (A, P, C, D) Dout = (orGate B1 B2, (A, P, C, D)) <==
     domain [Din1, Din2] ((fd_min Dout) + orGateDelay) (fd_max Dout),
     genCirDelay (A, P, C, D) Din1 == (B1, (A, P, C, D)),
     genCirDelay (A, P, C, D) Din2 == (B2, (A, P, C, D)),
     domain [Dout] (maximum (fd_min Din1)(fd_min Din2)) (fd_max Dout)
```

Observing the rules for the AND and OR gates, we can see two new fresh domain variables for representing the delay in their inputs. These new variables are constrained to have the domain of the delay in the output but pruned with the delay of the corresponding gate. After the circuits connected to the inputs had been generated, the domain of the output delay is pruned with the maximum of the input module delays. Note that although the maximum is computed *after* the input modules had been generated, the information in the given output delay has been propagated to the input delay domains so that whenever an input delay domain becomes empty, the search branch is no longer searched and another alternative is tried. Putting together the constraints about area, power dissipation, cost, and delay is straightforward, since they are orthogonal factors that can be handled in the same way. In addition to the constraints shown, we can further constrain the circuit generation with other factors as fan-in, fan-out, and switching function enforcement, to name a few. Then, we could submit the following goal:

```
domain [A] 0 maxArea, domain [P] 0 maxPower, domain [C] 0 maxCost,
domain [D] 0 maxDelay, genCir (A,P,C,D) == (B, S), switchingFunction B == sw
```

where `switchingFunction` can be defined as the switching function that returns the result of a behavior `B` for all its input combinations, and `sw` is the function that returns the intended result (`sw` is referred to as a problem parameter, as well as `maxArea`, `maxPower`, `maxCost`, and `maxDelay`).

```
data functionality = [bool]
switchingFunction :: behavior -> functionality
switchingFunction Behavior = [Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8] <==
```

```
      (Behavior false false false) == Out1,
      (Behavior false false true)  == Out2,
      (Behavior false true  false) == Out3,
      (Behavior false true true)   == Out4,
      (Behavior true  false false) == Out5,
      (Behavior true  false true)  == Out6,
      (Behavior true  true  false) == Out7,
      (Behavior true  true  true)  == Out8
```

Then, to generate a NOR circuit with `maxArea`, `maxPower`, `maxCost` and `maxDelay` equal 6, we could submit the following goal:

```
domain [A, P, C, D] 0  6, genCir (A,P,C,D) == (B, S),
switchingFunction B == [true,false,false,false,false,false,false,false]
```

An example of generating a NOR circuit is shown next:

```
Toy(FD)> F==[true,false,false,false,false,false,false,false],
         genCircuit 6 6 6 6 F == CIRCUIT
      { F -> [ true, false, false, false, false, false, false, false ],
        CIRCUIT -> ((notGate (orGate i0 (orGate i1 i2))), (_A, _B, _C, _D)) }
      { _A in 5..6,
        _B in 5..6,
        _C in 5..6,
        _D in 5..6 }
      Elapsed time: 5312 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { F -> [ true, false, false, false, false, false, false, false ],
        CIRCUIT -> ((notGate (orGate i0 (orGate i2 i1))), (_A, _B, _C, _D)) }
      { _A in 5..6,
        _B in 5..6,
        _C in 5..6,
        _D in 5..6 }
      Elapsed time: 32 ms.
sol.2, more solutions (y/n/d/a) [y]?

 ... up to 24 solutions
```

The solutions shown above to the problem are included in the distribution.

A.7.6 Golomb Rulers: An Optimization Problem (`golomb.toy`)

A Golomb ruler is a class of undirected graphs that, unlike usual rulers, measures more discrete lengths than the number of marks it carries. Its particularity is that on any given ruler, all differences between pairs of marks are unique. This feature makes Golomb Rulers to be really

interesting for practical applications such as radio astronomy, X-ray crystallography, circuit layout, geographical mapping, radio communications, and coding theory.
Traditionally, researchers are usually interested in discovering rulers with minimum length and Golomb rulers are not an exception. An *Optimal Golomb Ruler* (OGR) is defined as the shortest Golomb ruler for a number of marks. OGRs may be multiple for a specific number of marks. However, the search for OGRs is a task extremely difficult as this is a combinatorial problem whose bounds grow geometrically with respect to the solution size [34]. This has been a major limitation as each new ruler to be discovered is by necessity larger than its predecessor. Fortunately, the search space is bounded and, therefore, solvable [18]. To date, the highest Golomb ruler whose shortest length is known is the ruler with 23 marks [35]. Solutions to OGRs with a number of marks between 10 and 19 were obtained by very specialized techniques, and best solutions for OGRs between 20 and 23 marks were obtained by massive parallelism projects (these solutions took several months to be found) [35].
$\mathcal{TOY}(\mathcal{FD})$ enables the solving of optimization problems by using the function `labeling` with the value `toMinimize X` and/or `toMaximize X` (these values are intended for the minimization and maximization, respectively, of an FD variable `X`). Below, we show a $\mathcal{TOY}(\mathcal{FD})$ program to solve OGRs with N marks.

```
golomb :: int -> [int] -> bool
golomb N L = true <==
    hasLength L N,
    NN == trunc(2^(N-1)) - 1,
    domain L 0 NN,
    append [0|_] [Xn] == L,
    distances L Diffs,
    domain Diffs 1 NN,
    all_different Diffs,
    append [D1|_] [Dn] == Diffs,
    D1 #< Dn,
    labeling [toMinimize Xn] L

distances :: [int] -> [int] -> bool
distances []  [] = true
distances [X|Ys]  D0 = true <== distancesB X  Ys D0 D1, distances Ys D1

distancesB :: int -> [int] -> [int] -> [int] -> bool
distancesB _ []  D  D = true
distancesB X [Y|Ys] [Diff|D1] D0 = true <== Diff #= Y#-X, distancesB X Ys D1 D0
```

The next goal solves a rule for `N = 12` marks.

```
Toy(FD)> golomb 12 L
         { L <-  [0,2,6,24,29,40,43,55,68,75,76,85] }
```

$\mathcal{TOY}(\mathcal{FD})$ solves 10-marks OGRs in 17 seconds and 12-marks OGRs in 10,918 seconds (i.e., about three hours), in a Pentium 1.4 Ghz under Windows. See [9] for performance results.

A.7.7 Lazy Constraint Programs

A very powerful characteristic of $\mathcal{TOY}(\mathcal{FD})$ is lazy evaluation of goals (to our knowledge, $\mathcal{TOY}(\mathcal{FD})$ is the first constraint programming language providing laziness in the solving of goals). In this section, we show some examples of programs that combine FD constraint solving and lazy evaluation.

Lazy Magic Series (`lazymagicser.toy`)

Now we present a lazy solution for the problem of the magic series problem that was already treated in Section 3.3.5. With this new solution we illustrate some of the extra capabilities of the $CFLP(\mathcal{FD})$ approach of $\mathcal{TOY}(\mathcal{FD})$ with respect to the traditional $CLP(\mathcal{FD})$ approach.

```
include "misc.toy"    %% To use take/2, map/2 and ./2
include "cflpfd.toy"

generateFD :: int -> [int]
generateFD N = [X | generateFD N] <== domain [X] 0 (N-1)

lazymagic :: int -> [int]
lazymagic N = L <==
    take N (generateFD N) == L,   %% Lazy evaluation
    constrain L  L  0  Cs,
    sum L (#=) N,                 %% HO FD constraint
    scalar_product Cs  L  (#=) N, %% HO FD constraint
    labeling [ff] L

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain []  A  B  [] = true
constrain [X|Xs]  L  I  [I|S2] = true <==
    count I  L  (#=)  X,        %% HO FD constraint
    I1 == I+1,
    constrain Xs  L  I1  S2
```

The goal `lazymagic N`, for some natural number `N`, returns the N-magic series. Observe the lazy evaluation of the condition `take N (generateFD N)` as `(generateFD N)` produces an infinite list (as it was shown above). A possible goal is as follows:

```
Toy(FD)> > lazymagic 10
       [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ]
      Elapsed time: 48 ms.
```

Alternative solutions and more flexibility can be reached in $\mathcal{TOY}$ with $\mathcal{FD}$ constraints. For instance, a more interesting case consists of returning a list of solutions for a (possibly infinite) set of different instances of the problem. This can be done, for example, from a number N that

identified the first instance of the problem. Now we can make use of the concept of infinite lists by defining the following function.

```
magicfrom :: int -> [[int]]
magicfrom N = [lazymagic N|magicfrom(N+1)]
```

Now, it is easy to generate a list of N-magic series. For example, the following goal generates a 3-element list containing, respectively, the solution to the problems of 7-magic, 8-magic and 9-magic series[3].

```
Toy(FD)> take 3 (magicfrom 7) == L
      { L -> [ [ 3, 2, 1, 1, 0, 0, 0 ], [ 4, 2, 1, 0, 1, 0, 0, 0 ],
               [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ] ] }
      Elapsed time: 141 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

More expressiveness is shown by mixing curried functions, HO functions, infinite lists and function composition (another nice feature from the functional component of $\mathcal{TOY}$). For example, consider the code below: :

```
from :: int -> [int]
from N = [N|from (N+1)]

lazyseries :: int -> [[int]]
lazyseries = map lazymagic.from
```

where the operator '.' defines the composition of functions as follows (again, the function `./2` is predefined in the file `misc.toy`):

```
(.):: (B -> C) -> (A -> B) -> (A -> C) (F . G) X = F (G X)
```

Observe that *lazyseries* curries the composition `(map lazymagic).from`. Then, it is easy to generate the 3-element list shown above by just typing the goal

```
Toy(FD)> take 3 (lazyseries 7) == L
      { L -> [ [ 3, 2, 1, 1, 0, 0, 0 ], [ 4, 2, 1, 0, 1, 0, 0, 0 ],
               [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ] ] }
      Elapsed time: 125 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

[3]The function `take/2` is predefined in the file `misc.toy` (see Appendix B).

This goal is equivalent to the following

```
Toy(FD)> take 3 (map lazymagic (from 7)) == L
```

This simple example gives an idea of the nice features of $\mathcal{TOY}$ that combines $\mathcal{FD}$ constraint solving, management of infinite lists and lazy evaluation, curried notation of functions, polymorphism, HO functions (and thus HO constraints), composition of functions and a number of other characteristics that increase the potentialities with respect to $CLP(\mathcal{FD})$.

Pipelines (`pipelines.toy`)

Pipelines can be a powerful tool to solve heterogeneous constraint satisfaction problems, and these are easily expressed at a high level in $\mathcal{TOY}$ via applying HO constraints and curried notation. For example, recall the programs Optimal Golomb ruler (OGR), N-queens and client-server. Then, the goal (for some natural `N`):

```
map (map (queens [ff]))(map golomb (requests N)) == L
```

corresponds directly to the scheme shown in Figure A.6 if we redefine the function *process* of Section A.2.5 as `process = (+1)`. The solving of this goal, as in the preceding example of client-server architecture, generates N answers in the form of a N-elements list `A` from an initial request `initial = 4`; each element $\mathtt{a}_i \in$ `A` (i.e., each answer of the server with `i` $\in \{$`1,...,N`$\}$ and $\mathtt{a}_i$ `= initial + i - 1`) is used to feed the OGR solver with $\mathtt{a}_i$ marks producing a new list `S = [` $\mathtt{s}_{\mathtt{a}_1}$`,...,`$\mathtt{s}_{\mathtt{a}_N}$`]` containing `N` solutions for OGRs with marks $\mathtt{a}_1,\ldots,\mathtt{a}_N$. Finally, each element $\mathtt{o}_k$ (with `k`$\in\{$`1,...,`$\mathtt{a}_i\}$) belonging to the solution to the OGR with $\mathtt{a}_i$ marks in `S` (i.e., $\mathtt{s}_{\mathtt{a}_i}$ `= [`$\mathtt{o}_1$`,...,`$\mathtt{o}_{\mathtt{a}_i}$`]`) feeds the $\mathtt{o}_k$-queens solver and the first solution to the $\mathtt{o}_k$-queens problem is computed.
For example, the goal shown above (for `N = 2`) first calculates the solutions for the OGR with 4 marks (i.e., `[0,1,4,6]`) and 5 marks (i.e., `[0,1,4,9,11]`), and feeds the queens solver with each mark returning the first solution for `0,1,4,6,0,1,4,9` and `11` queens.

```
Toy(FD)> map (map (queens [ff])) (map golomb (requests 2)) == L
      { L ->  [  [ [],  [ 1 ],  [ 2, 4, 1, 3 ],  [ 2, 4, 6, 1, 3, 5 ] ],
                 [ [],  [ 1 ],  [ 2, 4, 1, 3 ],  [ 1, 3, 6, 8, 2, 4, 9, 7, 5 ],
                   [ 1, 3, 5, 7, 9, 11, 2, 4, 6, 8, 10 ] ] ] }
      Elapsed time: 187 ms.
more solutions (y/n/d) [y]? n
```

This example illustrates how easy and natural may be the combination of different problems in $\mathcal{TOY}$ without adding extra code.

A Tiling Problem (`tiling.toy`)

This example addresses the problem of tiling a given rectangular region with a set of available pieces. The simplest piece is a unit square (1×1)[4]. Less simple pieces can be seen as composed of unit squares. For instance, a 1×2 rectangle can be seen as two stacked unit squares.

[4] $X \times Y$ means a rectangle of X horizontal length units by Y vertical length units.

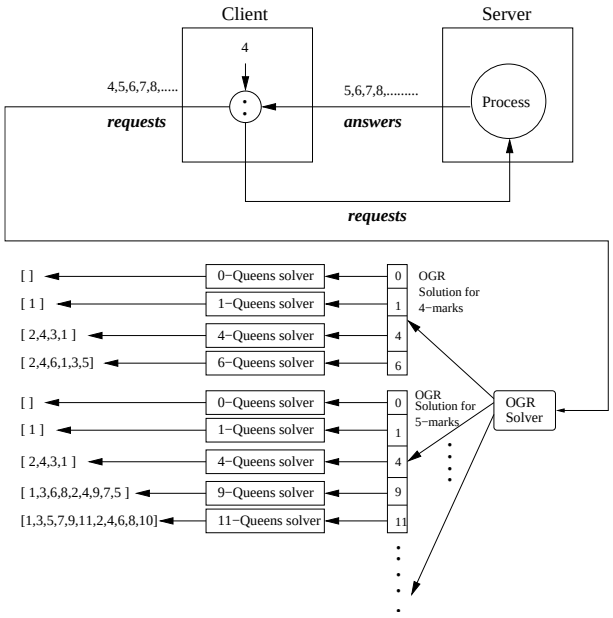


Figure A.6: Pipeline with Client-Server Architecture

For the sake of easing the presentation, we recourse to one-dimensional coordinates for representing the Cartesian plane so that, given a $LX \times LY$ region, we identify each coordinate pair (X, Y) ($X \in \{0, \ldots, LX - 1\}$, $Y \in \{0, \ldots, LY - 1\}$) as $X + Y * LX$. Assume that, for a given identifier I, $coor(I) = (X, Y)$ so that $I = X + Y * LX$. Piece locations are defined by one-dimensional coordinates.
The idea to specify this problem is to have a finite domain variable R representing the region to be filled. A number I in the domain of R means that we have room for placing a unit square at I (i.e., given $coor(I) = (X, Y)$, we can place at I the unit square defined by its borders $(X, Y), (X, Y + 1), (X + 1, Y), (X + 1, Y + 1)$). An absent value I from R means that a unit square is already located at I in the same sense as before. Therefore, we type `type region, location = int`.
Pieces located in the plane can be represented by a list of pairs `(piece, location)`. Each element in this list can be computed by a function returning the name of the piece (its function symbol) and its location, whose type definition is `type piece = ((region -> location), location)`.
We can define an infinite list generator of such pairs with indeterministic choice of pieces, as:

```
pieces :: region -> [piece]
pieces Region = [(Piece,Location)|pieces Region] <==
       choose_piece == Piece, Piece Region == Location
pieces Region = []
```

Note that this function definition embodies lazy evaluation, higher order applications, and constraint solving. `choose_piece` returns the function symbols for pieces, and is defined as:

```
choose_piece :: (region -> location)
choose_piece = square1
choose_piece = rectangle1x2 ...
```

The application of a piece to a region returns possible locations, pruning the region by constraint solving. For domain pruning, each piece application drops the values it occupies from the region domain, so that no other piece can be placed in the same location. For instance, the definition of a unit square is:

```
square1 :: region -> location
square1 R = L <==  domain [L] (fd_min R) (fd_max R),
                   inbounds L 1 1, free L R, R #\= L
```

The first condition restricts the domain of the location L of the unit square to that of the region. Next, a condition tests whether the unit square located at L is not out of bounds (by means of constraints). Then, if there is room for the unit square (tested by the function `free`), it is allocated by the disequality constraint, removing the location L from the domain of R.
The `inbounds` function is implemented as follows:

```
inbounds :: location -> int -> int -> bool
inbounds L X Y = true <== (L #mod spaceX) #+ X #< spaceX,          %X limit
                          (L #+ spaceX #* Y) #< spaceY #* spaceX   %Y limit
```

where `spaceX` and `spaceY` are problem parameters for imposing the horizontal and vertical available space. `free` is implemented via the predefined FD reflection function `fdset_belongs L R` (see Section 3.2.9). Given the complete program, we can submit goals as:

```
Toy(FD)> domain [R] 0 15, pieces R == L
      { L -> [ (square1, 0), (square1, 1), (square1, 2),
               (square1, 4), (square1, 5), (square1, 6),
               (square1, 8), (square1, 9), (square1, 10) ] }
      { R in{3}\/{7}\/(11..15) }
      Elapsed time: 0 ms.
```

which fills the region `R` with unit squares. Note that labeling is not needed since `fdset_belongs` bounds its argument whenever it is not bound. We can also submit the following goal, which tests whether a region can be filled with at least some pieces:

```
Toy(FD)> domain [R] 0 15,
         pieces R == [(rectangle2x1,L1),(rectangle1x2,L2),(square2,L3)|P]
      { L1 -> 0,
        L2 -> 2,
        L3 -> 4,
        P ->  [ (square1, 10) ] }
      { R in {3}\/{7}\/(11..15) }
      Elapsed time: 15 ms.
```

Note that we can request further solutions, obtaining all possible assignments of pieces in the region, even if the region is not completely filled. If we want to fill the region, we have to ensure that the region is empty. A straightforward condition is to test that all the available positions are occupied by pieces (that is, the cardinality of the region domain is the number of points where pieces cannot be allocated). Given the function `empty`, defined as follows:

```
empty :: int -> bool
empty R = (fd_size R #= spaceX + spaceY - 1)
```

So, we can submit the goal:

```
Toy(FD)> domain [R] 0 15, pieces R == L, empty R
```

This problem can be augmented with other constraints, such as the cost of filling the region with particular costs for each piece. Finally, note that regions to be filled can have any shape and even they have neither been connected nor convex.

A.7.8 Programmable Search (`search.toy`)

As an example of practical use of the reflection functions, here we show a two search strategies: First, a naive one (implemented with the function `search_naive/1`), which selects the variables to be labeled following the ordering of the input list, and the values in their domains in ascending order. Second, one of the most popular labeling strategies often supported by most constraint systems, the so-called *first-fail* strategy (implemented with the function `search_ff/1`), that selects the variable to be labeled which has the least number of values in its domain, and selects the values in their domains again in ascending order.

```
search_naive :: [int] -> bool
search_naive [] = true
search_naive [X] = false      <== empty_fdset (fd_set X)
search_naive [X] = true       <== domain [X] (fd_min X) (fd_min X)
search_naive [X] = true       <==
      Next == (fd_min X) + 1,
      domain [X] Next (fd_max X),
      search_naive [X]
search_naive [X,X1|Xs] = true <==
      search_naive [X],
      search_naive [X1],
      search_naive Xs

search_ff :: [int] -> bool
search_ff [] = true
search_ff [X] = false          <== empty_fdset (fd_set X)
search_ff [X] = true           <== domain [X] (fd_min X) (fd_min X)
search_ff [X] = true           <==
```

```
      Next == (fd_min X) +1,
      domain [X] Next (fd_max X),
      search_ff [X]
search_ff [X,X1|Xs] = true     <==
      choose_min_and_remove [X,X1|Xs] Y Ys,
      choose_min_and_remove Ys Y2 Yss,
      search_ff [Y],
      search_ff [Y2],
      search_ff Yss

choose_min_and_remove :: [int] -> int -> [int] -> bool
choose_min_and_remove [X] X [] = true
choose_min_and_remove [X,Y|Ys] M [Y|Rs] = choose_min_and_remove [X|Ys] M Rs <==
      fd_set X SX,
      fd_set Y SY,
      fdset_size SX <= fdset_size SY
choose_min_and_remove [X,Y|Ys] M [X|Rs] = choose_min_and_remove [Y|Ys] M Rs <==
      fd_set X SX,
      fd_set Y SY,
      fdset_size SX > fdset_size SY
```

Observe that when there are several variables to label, this function selects the one with the minimum domain cardinality (via `choose_min_and_remove/3`) by making use of information recovered by the reflection functions `fd_set/2` and `fdset_size/1`. Also, when there is just one variable `X`, it reactivates the search process by dividing its domain by the value `(fd_min X)`. The `search.toy` file contains several modified example programs to be solved with the labeling strategies `search_naive/1` and `search_ff/1`.

Appendix B

A Miscellanea of Functions

This chapter presents the contents of the two files `misc.toy` and `miscfd.toy`, which include useful functions and type declarations for programming and can be found in the directory `include` of the distribution.

B.1 misc.toy

```
% FILE: misc.toy
% A collection of useful functions and type declarations,
% many of them taken from Haskell's prelude

% type alias for strings
type string = [char]

infixl 90 !!        % nth-element selector
infixr 90 .         % function composition
infixr 50 ++        % concatenation of lists
infixr 40 //        % non-deterministic choice
infixr 40 `and`,/\  % parallel and sequential conjunction
infixr 30 `or`,\/   % parallel and sequential disjunction


% boolean functions

and,or,(/\),(\/) :: bool -> bool -> bool
not :: bool -> bool

% Parallel and
false `and` _ = false
_ `and` false = false
true `and`  true = true
```

```
% Parallel or
true ‘or‘   _ = true
_ ‘or‘ true = true
false ‘or‘  false = false

% Sequential and
false /\ _  = false
true  /\ X = X

% Sequential or
true  \/  X = true
false \/  X = X

% Negation
not true = false
not false = true

andL, orL ,orL’       :: [bool] -> bool
andL             = foldr (/\) true
orL              = foldr or false
orL’             = foldr (\/) false
  % orL’ is ’stricter’, but more deterministic, than orL

any, any’,all       :: (A -> bool) -> [A] -> bool
any P           = orL  . (map P)
any’ P          = orL’  . (map P)
  % any’ is ’stricter’, but more deterministic, than any
all P           = andL . (map P)


undefined :: A
undefined = if false then undefined

% (def X) is true if X is finite and totally defined
def X :- X == _

% (not_undef X) is true if X is not undefined
not_undef X :- X /= _

% (nf X) is the identity, restricted to finite and totally defined values
% Operationally, (nf X) forces the computation of a normal form for X,
% if it exists.
nf X = Y <== X==Y
```

```
% (hnf X) is the identity, restricted to not undefined values.
% Operationally, (hnf X) forces the computation of a head normal form for X,
% if it exists.
hnf X = X <== X /= _

% (strict F) is the restriction of F to finite, totally defined arguments.
% It forces the evaluation to nf of the argument before applying F
strict F X = F Y <== X==Y

% (strict' F) is the restriction of F to not undefined arguments.
% It forces the evaluation to hnf of the argument before applying F
strict' F X = F X <== X /= _


% mapping a function through a list
map:: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [(F X)|(map F Xs)]

%% Function composition
(.) :: (B -> C) -> (A -> B) -> (A -> C)
(F . G) X = F (G X)

%% List concatenation
(++) :: [A] -> [A] -> [A]
[] ++ Ys = Ys
[X|Xs] ++ Ys = [X|Xs ++ Ys]

%% Xs!!N is the Nth-element of Xs
(!!) :: [A] -> int -> A
[X|Xs] !! N = if N==0 then X else Xs !! (N-1)

iterate :: (A -> A) -> A -> [A]
iterate F X = [X|iterate F (F X)]

repeat :: A -> [A]
repeat X = [X|repeat X]

copy :: int -> A -> [A]
copy N X = take N (repeat X)

filter           :: (A -> bool) -> [A] -> [A]
filter L []       = []
```

```
filter P [X|Xs]   =
    if P X then  [X|filter P Xs]
           else  filter P Xs

%% Fold primitives:  The foldl and scanl functions, variants foldl1 and
%% scanl1 for non-empty lists, and strict variants foldl' scanl' describe
%% common patterns of recursion over lists.  Informally:
%%
%%  foldl F   a [x1, x2, ..., xn]  = F   (...(f (f a x1) x2)...) xn
%%                                 = (...((a `f` x1) `f` x2)...) `f` xn
%% etc...
%%
%% The functions foldr, scanr and variants foldr1, scanr1 are duals of these
%% functions:
%% e.g.  foldr F a Xs = foldl (flip f) a (reverse Xs) for any finite lists Xs

foldl             :: (A -> B -> A) -> A -> [B] -> A
foldl  F Z []      = Z
foldl  F Z [X|Xs]  = foldl F (F Z X) Xs

foldl1            :: (A -> A -> A) -> [A] -> A
foldl1 F [X|Xs]    = foldl F X Xs

foldl'            :: (A -> B -> A) -> A ->  [B]  -> A
foldl' F A []      =  A
foldl' F A [X|Xs]  =  strict (foldl' F) (F A X) Xs

scanl             :: (A -> B -> A) -> A -> [B] -> [A]
scanl  F Q []      = [Q]
scanl  F Q [X|Xs]  = [Q|scanl F (F Q X) Xs]

scanl1            :: (A -> A -> A) ->  [A]  ->  [A]
scanl1 F [X|Xs]    = scanl F X Xs

scanl'            :: (A -> B -> A) -> A -> [B] -> [A]
scanl' F Q []      = [Q]
scanl' F Q [X|Xs]  = [Q|strict (scanl' F) (F Q X) Xs]

foldr             :: (A -> B  -> B) -> B ->  [A]  -> B
foldr  F Z []      = Z
foldr  F Z [X|Xs]  = F X (foldr F Z Xs)

foldr1            :: (A -> A -> A) ->  [A]  -> A
foldr1 F [X]       = X
```

```
foldr1 F [X,Y|Xs]  = F X (foldr1 F [Y|Xs])

scanr             :: (A -> B  -> B) -> B ->  [A]  ->  [B]
scanr  F Q0 []     = [Q0]
scanr  F Q0 [X|Xs] = auxForScanr F X (scanr F Q0 Xs)
%where
auxForScanr F X Ys = [F X (head Ys)|Ys]

scanr1            :: (A -> A -> A) ->  [A]  ->  [A]
scanr1 F [X]       = [X]
scanr1 F [X,Y|Xs]  = auxForScanr F X (scanr1 F [Y|Xs])


%% List breaking functions:
%%
%%   take n Xs         returns the first n elements of Xs
%%   drop n Xs         returns the remaining elements of Xs
%%   splitAt n Xs      = (take n Xs  , drop n Xs  )
%%
%%   takeWhile P Xs    returns the longest initial segment of Xs   whose
%%                     elements satisfy p
%%   dropWhile P Xs    returns the remaining portion of the list
%%   span P Xs         = (takeWhile P Xs  , dropWhile P Xs  )
%%
%%   takeUntil P Xs    returns the list of elements upto and including the
%%                     first element of Xs   which satisfies p



take :: int -> [A] -> [A]
take N [] = []
take N [X|Xs] = if N==0 then [] else [X|take (N-1) Xs]

drop                  :: int  ->  [A]  ->  [A]
drop N []        = []
drop N [X|Xs]    = if N==0 then [X|Xs] else drop (N-1) Xs


splitAt             :: int  ->  [A]  -> ( [A] ,  [A] )
splitAt N []        =  ([],[])
splitAt N [X|Xs]    = if N==0
                      then ([], [X|Xs])
                      else auxForSplitAt X (splitAt (N-1) Xs)
%where
```

```
auxForSplitAt X (Xs,Ys) = ([X|Xs],Ys)

takeWhile           :: (A -> bool) ->  [A]  ->  [A]
takeWhile P []       = []
takeWhile P [X|Xs]   = if P X then [X| takeWhile P Xs] else []

takeUntil           :: (A -> bool) ->  [A]  ->  [A]
takeUntil P []       = []
takeUntil P [X|Xs]   = if P X then [X] else [X| takeUntil P Xs]

dropWhile           :: (A -> bool) ->  [A]  ->  [A]
dropWhile P []       = []
dropWhile P [X|Xs]   = if P X then dropWhile P Xs else [X|Xs]


span, break         :: (A -> bool) ->  [A]  -> ( [A] , [A] )
span P []            = ([],[])
span P [X|Xs]        = if P X
                       then auxForSpan X (span P Xs)
                       else ([], [X|Xs])

auxForSpan X (Xs,Ys) = ([X|Xs],Ys) % Identical to auxForSplitAt

break P               = span (not . P)

zipWith                  :: (A->B->C) -> [A]->[B]->[C]
zipWith Z []     Bs       = []
zipWith Z [A|As] []       = []
zipWith Z [A|As] [B|Bs]   = [Z A B | zipWith Z As Bs]


zip                      :: [A]->[B]->[(A,B)]
zip Xs Ys                 = zipWith mkpair Xs Ys
%where
mkpair                   :: A -> B ->(A,B)
mkpair X Y                = (X,Y)

unzip                    :: [(A,B)] -> ([A],[B])
unzip  []                 = ([],[])
unzip  [(X,Y)|XsYs]       = auxForUnzip X Y (unzip XsYs)

auxForUnzip X Y (Xs,Ys)   = ([X|Xs],[Y|Ys])

until                  :: (A -> bool) -> (A -> A) -> A -> A
```

```
until P F X             = if P X then X else until P F (F X)

until'                 :: (A -> bool) -> (A -> A) -> A -> [A]
until' P F             = (takeUntil P) . (iterate F)


%% Standard combinators:  %% %% %% %% %% %%

const          :: A -> B -> A
const K X       = K

id             :: A -> A
id    X         = X

% non-deterministic choice
(//)           :: A -> A -> A
X // _          = X
_ // Y          = Y

curry          :: ((A,B)  -> C ) -> A -> B -> C
curry F A B     =  F (A,B)

uncurry        :: (A -> B -> C ) -> (A,B)  -> C
uncurry F (A,B)  = F A B

fst            :: (A,B)  -> A
fst (X,Y)       = X

snd            :: (A,B)  -> B
snd (X,Y)       = Y

fst3           :: (A,B,C)  -> A
fst3 (X,Y,Z)    = X

snd3           :: (A,B,C)  -> B
snd3 (Y,X,Z)    = X

thd3           :: (A,B,C)  -> C
thd3 (Y,Z,X)    = X

subtract  :: real -> real-> real
subtract   = flip (-)

even, odd :: int -> bool
```

```
even X     = (X `mod` 2) == 0
odd        = not . even

lcm       :: int -> int -> int
lcm X Y    = if ((X==0) \/ (Y == 0)) then 0
             else abs ((X `div` (gcd X Y)) * Y)


%%%% Standard list processing functions:  %% %% %% %% %% %%

head             :: [A] -> A
head [X|_]       = X

last             :: [A] -> A
last [X]          = X
last [_,Y|Xs]     = last [Y|Xs]

tail             :: [A] -> [A]
tail [_|Xs]       = Xs

init             :: [A] -> [A]
init [X]          = []
init [X,Y|Xs]     = [X|init [Y|Xs]]

nub              :: [A] -> [A]   %% remove duplicates from list
nub []            = []
nub [X|Xs]        = [X| nub (filter (X /=) Xs)]

length           :: [A] -> int
length []         = 0
length [_|Xs]     = 1 + length Xs


size             :: [A] -> int
size              = length . nub

reverse          :: [A] -> [A]           %% reverse elements of list
reverse           = foldl (flip (:)) []


member,notMember :: A -> [A] -> bool
member            = any' . (==)          %% test for membership in list
notMember         = all . (/=)           %% test for non-membership
```

```
concat           :: [[A]] -> [A]         %% concatenate list of lists
concat            = foldr (++) []

transpose        :: [[A]] -> [[A]]       %% transpose list of lists
transpose         = foldr
                      auxForTranspose
                      []
%where
auxForTranspose Xs Xss = zipWith (:) Xs (Xss ++ repeat [])


 %% (\\) is used to remove the first occurrence of each element in the second
 %% list from the first list.  It is a kind of inverse of (++) in the sense
 %% that  (xs ++ ys) \\ xs = ys for any finite list xs of proper values xs.

infix 50 \\

(\\)             :: [A] -> [A] -> [A]
(\\)              = foldl del
 %where
[]     `del` Y  = []
[X|Xs] `del` Y  = if X == Y then Xs
                    else [X|Xs `del` Y]
```

B.2 misc.toy

```
% FILE: miscfd.toy
% A collection of useful FD functions

% hasLength/2 computes the length of a list, irrespective of its mode of usage
% Whereas a goal as length L == 4 length does not terminate when prompting for all
% solutions, hasLength L 4 does
hasLength :: [A] -> int -> bool hasLength [] 0 :- true hasLength
[X|Xs] N :- N #> 0, hasLength Xs (N #- 1)

% belongsOrd/2 is a version of belongs/2 that allows its second argument to be
% an infinite list
belongsOrd :: int -> [int] -> bool belongsOrd X L =
  belongs X (intersectOrdIntLists (fdset_to_list (fd_set X)) L)

% intersectOrdIntLists/2 returns the intersection of its two input (ordered) integer
% lists, allowing infinite lists
intersectOrdIntLists :: [int] -> [int] -> [int] intersectOrdIntLists
```

```
[] L = [] intersectOrdIntLists [X|Xs] [] = [] intersectOrdIntLists
[X|Xs] [Y|Ys] =
  if X #< Y
     then intersectOrdIntLists Xs [Y|Ys]
     else if X #> Y
             then intersectOrdIntLists [X|Xs] Ys
             else [X|intersectOrdIntLists Xs Ys]
```

Appendix C

Sample Modules

```
% NATURAL NUMBERS
module nat

   exports
      data nat = z | s nat
      (+.)  :: nat -> nat -> nat
      (*.)  :: nat -> nat -> nat
      (-.)  :: nat -> nat -> nat
      (<=.) :: nat -> nat -> bool
      (>.)  :: nat -> nat -> bool
      (<.)  :: nat -> nat -> bool
      mx    :: nat -> nat -> nat
      mn    :: nat -> nat -> nat
      even  :: nat -> bool
      odd   :: nat -> bool
      dv    :: nat -> nat -> nat
      md    :: nat -> nat -> nat

      % dv and md are only defined for non-zero second argument

   body

      z +. N = N
      (s M) +. N = s (M +. N)

      z *. N = z
      (s M) *. N = (M *. N) +. N

      z -. N = z
      (s M) -. z = s M
      (s M) -. (s N) = M -. N

      z <=. N = true
      (s M) <=. z = false
```

```
      (s M) <=. (s N) = M <=. N

      M <. z = false
      z <. (s N) = true
      (s M) <. (s N) = M <. N

      N >. M = M <. N

      mn N M = if M >. N then N else M

      mx N M = if M >. N then M else N

      even z = true
      even (s M) = odd M

      odd z = false
      odd (s M) = even M

      N `dv` P = if P >. N then z else s ((N -. P) `dv` P)
                 <== P /= z

      N `md` P = if P >. N then N else (N -. P) `md` P
                 <== P /= z


% LISTS
module list

   imports
      nat exporting types nat
                    ops z, s

   exports
      data list A = nil | A :+ (list A)
      tail    :: list A -> list A
      head    :: list A -> A
      (++)    :: list A -> list A -> list A
      length  :: list A -> nat
      reverse :: list A -> list A
      take    :: nat -> list A -> list A
      drop    :: nat -> list A -> list A

      % tail and head are only defined for non-empty lists

   body

      tail (N :+ L) = L

      head (N :+ L) = N
```

```
      nil ++ L = L
      (N :+ L) ++ L' = N :+ (L ++ L')

      length nil = z
      length (N :+ L) = s (length L)

      reverse nil = nil
      reverse (N :+ L) = (reverse L) ++ (N :+ nil)

      take z L = nil
      take N nil = nil
      take (s N) (M :+ L) = M :+ (take N L)

      drop z L = L
      drop N nil = nil
      drop (s N) (M :+ L) = drop N L


% ORDERED LISTS
module ordlist

   parameters
      type elt
      le :: elt -> elt -> bool

   imports
      list exporting types list elt
                     ops nil,tail,head,length

   exports
      is_ordered :: list elt -> bool
      insertsort :: list elt -> list elt
      mergesort  :: list elt -> list elt
      quicksort  :: list elt -> list elt

   body
      % local auxiliary operations

      insert_list :: list elt -> elt -> list elt
      insert_list nil M = M :+ nil
      insert_list (N :+ OL) M = if (le M N)
                                then (M :+ (N :+ OL))
                                else (N :+ (insert_list OL M))

      merge :: list elt -> list elt -> list elt
      merge OL nil = OL
      merge nil OL = OL
      merge (N :+ OL) (M :+ OL') = if (le N M)
```

```
                                then N :+ (merge OL (M :+ OL'))
                                else M :+ (merge (N :+ OL) OL')

      le_elems :: list elt -> elt -> list elt
      le_elems nil M = nil
      le_elems (N :+ L) M = if (le N M)
                            then N :+ (le_elems L M)
                            else le_elems L M

      gr_elems :: list elt -> elt -> list elt
      gr_elems nil M = nil
      gr_elems (N :+ L) M = if (le N M)
                            then gr_elems L M
                            else N :+ (gr_elems L M)

      % equations for main operations

      is_ordered nil = true
      is_ordered (N :+ nil) = true
      is_ordered (N :+ (N :+ L)) = is_ordered (N :+ L)
      is_ordered (N :+ (M :+ L)) = if (le N M)
                                   then is_ordered (M :+ L)
                                   else false
                                 <== N /= M

      insertsort nil = nil
      insertsort (N :+ L) = insert_list (insertsort L) N

      mergesort nil = nil
      mergesort (N :+ nil) = N :+ nil
      mergesort L = if length L >. (s z)
                    then merge (mergesort (take ((length L) `dv` (s (s z))) L))
                               (mergesort (drop ((length L) `dv` (s (s z))) L))

      quicksort nil = nil
      quicksort (N :+ L) =
                (quicksort (le_elems L N)) ++ (N :+ (quicksort (gr_elems L N)))


% BINARY TREES
module bintree

   imports
      nat exporting types nat
                  ops z, s

   exports
      data bintree A = empty | tree (bintree A) A (bintree A)
      left  :: bintree A -> bintree A
```

```
      right :: bintree A -> bintree A
      root  :: bintree A -> A
      depth :: bintree A -> nat

      % left, right, and root are only defined for non-empty trees

   body

      left (tree L _ _) = L
      right (tree _ _ R) = R
      root (tree _ N _) = N

      depth empty = z
      depth (tree L N R) = s (mx (depth L) (depth R))


% BINARY TREE TRAVERSALS
module bintree_traversals

   extends
      bintree

   imports
      list exporting all

   exports
      leaves    :: bintree A -> list A
      preorder  :: bintree A -> list A
      inorder   :: bintree A -> list A
      postorder :: bintree A -> list A

   body

      leaves empty = nil
      leaves (tree empty N empty) = N :+ nil
      leaves (tree L N R) = (leaves L) ++ (leaves R)
                        <== (leaves L) ++ (leaves R) /= nil

      preorder empty = nil
      preorder (tree L N R) = N :+ ((preorder L) ++ (preorder R))

      inorder empty = nil
      inorder (tree L N R) = (inorder L) ++ (N :+ (inorder R))

      postorder empty = nil
      postorder (tree L N R) = (postorder L) ++ ((postorder R) ++ (N :+ nil))


% SEARCH TREES
```

```
module searchtree

   parameters
      type elt
      le :: elt -> elt -> bool
      eq :: elt -> elt -> bool

   imports
      bintree renaming (type bintree elt to stree elt)
              exporting types stree elt
                        ops empty, root, left, right

   exports
      insert :: stree elt -> elt -> stree elt
      delete :: stree elt -> elt -> stree elt
      is_in  :: stree elt -> elt -> bool

   body
      % local auxiliary operations
      % mint and maxt are not defined on the empty tree

      mint :: stree elt -> elt
      mint (tree empty N _) = N
      mint (tree L _ _) = mint L
                          <== L /= empty

      maxt :: stree elt -> elt
      maxt (tree _ N empty) = N
      maxt (tree _ _ R) = maxt R <== R /= empty

      % is_stree is not used in the following but may be useful
      % in defining operations such that they check the arguments

      is_stree :: stree elt -> bool
      is_stree empty = true
      is_stree (tree empty _ empty) = true
      is_stree (tree L N empty) = if is_stree L
                                    then le (maxt L) N
                                    else false
                                  <== L /= empty

      is_stree (tree empty N R) = if is_stree R
                                    then le N (mint R)
                                    else false
                                  <== R /= empty

      is_stree (tree L N R) = if is_stree L
                                then if le (maxt L) N
                                       then if is_stree R
```

```
                                        then le N (mint R)
                                        else false
                                  else false
                            else false
                          <== L /= empty, R /= empty

      % equations for main operations

      insert empty M = tree empty M empty
      insert (tree L N R) M = if (eq M N)
                               then tree L M R
                               else if (le M N)
                                     then tree (insert L M) N R
                                     else tree L N (insert R M)

      delete empty _ = empty
      delete (tree L N R) M = tree (delete L M) N R
                              <== le M N
      delete (tree L N R) M = tree L N (delete R M)
                              <== le N M
      delete (tree empty N R) M = R
                                  <== eq N M
      delete (tree L N empty) M = L
                                  <== eq N M
      delete (tree L N R) M = tree L X (delete R X)
                              <== eq N M, L /= empty, R /= empty, X == mint R

      is_in empty _ = false
      is_in (tree L N R) M = if eq M N
                              then true
                              else if le M N
                                    then is_in L M
                                    else is_in R M


% SORTING A LIST USING A SEARCH TREE
module searchtree_sort

   parameters
      type elt
      le :: elt -> elt -> bool
      eq :: elt -> elt -> bool

   extends
      ordlist
      % default instantiation

   imports
      searchtree
```

```
      % default instantiation

   exports
      stree_sort :: list elt -> list elt

   body
      % local auxiliary operations

      listToStree :: list elt -> stree elt
      listToStree nil = empty
      listToStree (N :+ L) = insert (listToStree L) N

      inorder_st :: stree elt -> list elt
      inorder_st empty = nil
      inorder_st T = (inorder_st (left T)) ++
                                 ((root T) :+ (inorder_st (right T)))
                  <== T /= empty

      % Note: There is no way to reuse the inorder traversal
      % from bintree_traversals because parameter types do not match

      % main operation

      stree_sort L = inorder_st (listToStree L)

      % stree_sort only works for lists with no repetitions,
      % because all elements in a search tree are different


% DICTIONARY AS SEARCH TREE
module dictionary

   parameters
      type key
      type contents
      kle,keq :: key -> key -> bool
      default :: contents
      combine :: contents -> contents -> contents

   parbody
      ple,peq :: (key, contents) -> (key, contents) -> bool
      ple (k1,c1) (k2,c2) = kle k1 k2
      peq (k1,c1) (k2,c2) = keq k1 k2

   imports
      searchtree (type elt to (key, contents),
                  op  le  to ple,
                  op  eq  to peq)
            renaming (type stree (key, contents) to dict (key, contents))
```

```
            exporting types dict (key, contents)
                      ops empty

   exports
      find       :: dict (key,contents) -> key -> bool
      insertcont :: dict (key,contents) -> key -> contents -> dict (key,contents)
      deletekey  :: dict (key,contents) -> key -> dict (key,contents)
      lookup     :: dict (key,contents) -> key -> contents

   body

      find D K = is_in D (K,default)

      insertcont D K C = if C' == default
                         then insert D (K,C)
                         else insert D (K, (combine C' C))
                        <== C' == lookup D K

      deletekey D K = delete D (K,default)

      lookup empty K = default
      lookup D K = if (keq K K')
                   then C'
                   else if (kle K K')
                         then lookup (left D)  K
                         else lookup (right D) K
                  <== D /= empty, (K',C') == root D


% MULTISET AS DICTIONARY
module multiset

   parameters
      type elt
      le,eq :: elt -> elt -> bool

   imports
       nat exporting all
       dictionary (type key to elt,
                   type contents to nat,
                   op kle to le,
                   op keq to eq,
                   op default to z,
                   op combine to (+.))

   exports
      type multiset elt
      empty_ms   :: multiset elt
      add_one    :: multiset elt -> elt -> multiset elt
```

```
   add_many   :: multiset elt -> elt -> nat -> multiset elt
   mult       :: multiset elt -> elt -> nat
   delete_one :: multiset elt -> elt -> multiset elt
   delete_all :: multiset elt -> elt -> multiset elt

body

   type multiset elt = dict (elt, nat)

   empty_ms = empty

   add_one MS E = insertcont MS E (s z)

   add_many MS E N = insertcont MS E N

   mult MS E = lookup MS E

   delete_all MS E = deletekey MS E

   delete_one MS E = if (s z) <. N
                     then add_many MS' (N -. (s z))
                     else MS'
                   <== N == mult MS E, MS' == delete_all MS E
```

Appendix D

Syntax

In this appendix the lexicon and grammar of the language are presented.

D.1 Lexicon

First we describe the set of tokens allowed in the language, as well as the reserved words and operations.
A token must belong to one of the following categories:

- *identifier* = A lower case letter followed by any sequence of letters, digits and symbols ’ ’ ’ and ’_’.
- *variable* = Analogous to *identifier* but beginning with an upper case letter.
- *Integer* = sequence of digits optionally preceded by a symbol ’-’.
- *Float* = sequence of digits including a decimal dot and optionally preceded by a symbol ’-’.
- *Char* = Any character enclosed between single quotes. Also some special characters (including control characters) must be preceded by a backslash. The following list presents these special characters:
  - ’\\’ backslash
  - ’\’’ quote
  - ’\”’ double quote
  - ’\n’ new line
  - ’\r’ carriage return
  - ’\t’ horizontal tabulator
  - ’\v’ vertical tabulator
  - ’\a’ bell

`,`	`=`	`:-`	`::`
`<==`	`->`	`data`	`else`
`if`	`in`	`include`	`includecflpfd` (reserved)
`infix`	`infixl`	`infixr`	`primitive`
`then`	`type`	`where`	

Table D.1: Reserved Symbols

		User-Defined		
		Function	Data Type	Data Constructor
Predefined	Function	×	√	×
	Data Type	√	×	√
	Data Constructor	×	√	×

Table D.2: Predefined vs. User-Defined Symbol Compatibility Table

- '\b' backspace
- '\f' form feed
- '\e' escape
- '\d' delete

- *String*: Sequence of characters enclosed by double quotes.
- *Bool*: Either *true* or *false*.
- *Operator*: Sequence of characters in the set { '.', '/', '!', '#', '&', '*', '+', '>', '<', '¿', '?', '@', '\', '^̂', '|, '-', ':', '%', '$' } whose first character is neither '%' nor '$' nor ':'.
- *ConsOperator*: Analogous to an *operator* but beginning by ':'.

Table D.1 shows the set of reserved symbols (keywords) which cannot be used for any other purpose they are deserved to. (See section 2.) In this and following tables, if (*reserved*) occurs next to a symbol, it means that this symbol is reserved for a forthcoming release.
There is also a set of predefined symbols which can be classified as functions, data constructors, and data types. Table D.2 shows the compatibility of user defined symbols versus predefined symbols. This table states, for instance, that the user can define a new data type symbol even if there is a predefined function or data constructor with the same symbol. On the other hand, the user cannot define a new function with a name which is already defined for a predefined function or data constructor.
Tables D.3, D.4, and D.5 summarize the functions, data types, and data constructors in the plain language. The user must be aware of the aforementioned compatibility table when defining new symbols in user programs.
The predefined symbols in the libraries I/O File, I/O Graphic and Finite Domain Constraints are summarized in the following tables. Table D.6 shows the predefined functions for the library

+	-	*	/	**
^	==	/=	<	<=
>	>=	>>	>>=	abs
acos	acosh	acot	acoth	asin
asinh	atan	atanh	ceiling	chr
collect	collectN	cont1	cont2	cos
cosh	cot	coth	div	do
done	dVal	dValToString	else	evalfd (reserved)
exp	fails	flip	floor	gcd
getChar	getConstraintStore	getLine	if_then	if_then_else
ln	log	max	min	mod
once	ord	putChar	putStr	putStrLn
readFile	readFileContents	return	round	selectWhereVariableXi
sin	sinh	sqrt	tan	tanh
toReal	trunc	uminus	writeFile	

Table D.3: Plain Language Predefined Functions

()	(τ_1, ...)	($\tau_1 \rightarrow \tau_2$)	[τ]	bool
atomicConstraint	channel	char	constraint	cTree
handle	int	io τ	ioMode	pVal
real	varmut			

Table D.4: Plain Language Predefined Data Types

,	:	[]	appendMode	atomicConstraint
channel	char	constraint	cTreeNode	cTreeVoid
false	io	pValApp	pValBottom	pValChar
pValNum	pValVar	readMode	stream	true
varmut	writeMode			

Table D.5: Plain Language Predefined Data Constructors

closeFile	contin1	contin2	end_of_file	getCharFile
getLineFile	openFile	putCharFile	putStrFile	putStrLnFile

Table D.6: I/O File Library Predefined Functions

aux	aux1	aux2	aux3	aux4
checkWishConsistency	closeWish	escape_tcl	forkWish	getNumber
getString	initSchedule	intTostr	newVar	newVar1
newVar2	openWish	readVar	readWish	runWidget
runWidgetInit	runWidgetPassive	setlistelems	setmenuelems	strToint
tk2tcl	tkAddCanvas	tkcitem	tkcitems2tcl	tkConf2tcl
tkConfCollection2tcl	tkConfig	tkConfs2handler	tkConfs2tcl	tkExit
tkFocus	tkGetValue	tkGetVar	tkGetVarMsg	tkGetVarValue
tkLabel2Refname	tkmain2tcl	tkMenu2handler	tkMenu2tcl	tkParseInt
tkRef2Label	tkRef2Wtype	tkRefname2Label	tks2tcl	tkSchedule
tkSelectEvent	tkSetValue	tkShowCoords	tkShowErrors	tkslabels
tkTerminateWish	tkUpdate	tkVoid	writeVar	writeVar1
writeWish				

Table D.7: I/O Graphic Library Predefined Functions

I/O File. Note that the data type and data constructors needed for these functions are already defined in the plain language predefinitions. Tables D.7, D.8, and D.9 show, respectively, the predefined symbols for functions, data types, and data constructors for the library I/O Graphic. Finally, tables D.10, D.11, and D.12 show, respectively, the predefined symbols for functions, data types, and data constructors for the library Finite Domain Constraints. Observe that the library Real Constraints does not introduce new symbols and arithmetic operators from the plain language are otherwise reused for building constraints.

D.2 Grammar

This section describes $\mathcal{TOY}$ syntax. Terminals are shown enclosed in square boxes like this. Curly brackets ('{', '}') represent the choice of one among several elements, separated by commas. Square brackets enclose optional elements, while rounded brackets are used for grouping. The postfix operator * represents the repetition of tokens of some syntactic category zero or more times, while $^+$ represents the repetition one or more times. Syntactic categories of tokens are written in italic letters (see *lexicon* above).

tkCanvasItem	tkConfCollection	tkConfItem
tkMenuItem τ	tkRefType	tkWidget

Table D.8: I/O Graphic Library Predefined Data Types

tkActive	tkAnchor	tkBackground	tkBottom	tkButton
tkCanvas	tkCenter	tkCheckButton	tkCmd	tkCol
tkEntry	tkExpand	tkExpandX	tkExpandY	tkHeight
tkInit	tkItems	tkLabel	tkLeft	tkLine
tkList	tkListBox	tkMButton	tkMenu	tkMenuButton
tkMessage	tkMMenuButton	tkMSeparator	tkOval	tkPolygon
tkRectangle	tkRef	tkRefLabel	tkRight	tkRow
tkScale	tkScrollH	tkScrollV	tkTcl	tkText
tkTextEdit	tkTop	tkWidth		

Table D.9: I/O Graphic Library Predefined Data Constructors

#=	#\=	#<	#<=
#>	#>=	#+	#-
#*	#/	#&	#<=>
#=>	#\/	all_different	all_different'
assignment	belongs	circuit	circuit'
count	cumulative	cumulative'	domain
element	empty_fdset	empty_interval	end_fd (reserved)
exactly	fd_closure	fd_degree	fd_dom
fd_global (reserved)	fd_max	fd_min	fd_neighbors
fd_set	fd_size	fd_statistics	fd_statistics'
fd_var	fdmaximize (reserved)	fdminimize (reserved)	fdset_add_element
fdset_belongs	fdset_complement	fdset_del_element	fdset_disjoint
fdset_equal	fdset_intersect	fdset_intersection	fdset_max
fdset_member	fdset_min	fdset_parts	fdset_singleton
fdset_size	fdset_split	fdset_subset	fdset_subtract
fdset_to_interval	fdset_to_list	fdset_to_range	fdset_union
fdsets_intersection	fdsets_union	indomain	inf
init_fd (reserved)	inset	intersect	interval_to_fdset
is_fdset	isin (reserved)	labeling	list_to_fdset
minimum (reserved)	range_to_fdset	scalar_product	serialized
serialized'	setcomplement	subset	sum
sup			

Table D.10: Finite Domain Library Predefined Functions

allDiffOptions	fdinterval	fdset
labelingType	range	reasoning
serialOptions	statistics	wakeOptions (reserved)

Table D.11: Finite Domain Library Predefined Data Types

`assumptions`	`backtracks`	`bisect`	`compl`
`complete`	`constraints`	`cte`	`decomposition`
`domains`	`domm` (reserved)	`down`	`each`
`edge_finder`	`entailments`	`enum`	`ff`
`ffc`	`inter`	`interval`	`leftmost`
`lift`	`liftedInt`	`maxi`	`maxx` (reserved)
`mini`	`minmax` (reserved)	`minn` (reserved)	`path_consistency`
`precedences`	`prunings`	`range`	`reasoning`
`resumptions`	`static_sets`	`step`	`superior`
`toMaximize`	`toMinimize`	`typeprecedence`	`uni`
`up`	`vall` (reserved)	`value`	

Table D.12: Finite Domain Library Predefined Data Constructors

% P R O G R A M S

% A program is a sequence of declarations
program → decl*

% Top-level declarations
decl → includeDecl
| infixDecl
| dataDecl
| typeAliasDecl
| funTypeDecl
| funDecl
| clauseDecl

% T O P L E V E L D E C L A R A T I O N S

% Include declaration
includeDecl → include *String*

% Infix operator declaration
infixDecl → {infix, infixr, infixl} *Integer* op(, op)*

% Datatype declaration
dataDecl → data typeId varId* = dataTypeRhs (| dataTypeRhs)*
dataTypeRhs → type consOp type
| constructor simpleType*

% Type alias declaration
typeAliasDecl → type typeId varId* = type

% Function type declaration
funTypeDecl → function (, function)* :: type

% F U N C T I O N A N D C L A U S E D E F I N I T I O N S

% Function rule declaration
funDecl → rule+

% Function defining rule
rule → ruleLhs { =, -> } expr [conditionDecls] [whereDecls]

% Predicate clause: similar to Prolog, but possibly with *where* declarations
% and not ending in '.'
clauseDecl → ruleLhs :- conditionDecls [whereDecls]

% Right-hand side (both for program rules and for clauses)
ruleLhs → function fPat*
| infixOpLhs
infixOpLhs → fPat funOp fPat
| (fPat funOp fPat) fPat*

% Conditional part (of a function rule or predicate clause)
conditionDecls → <== expr (, expr)*

% Where declarations: the lhs must be a linear pattern
whereDecls → where { patBinding (; patBinding)* }
patBinding → pattern = expr

% T Y P E S

type → cType (`->` cType)*

% Types without any external occurrence of ->
cType → typeId simpleType$^+$
| simpleType

% Simple Types
simpleType → typeId
| varId
| listType
| tupleType
| `(` type `)`
listType → `[` type `]`
tupleType → `(` type (`,` type)* `)`

% P A T T E R N S

% m less than the arity of the function
% m' less or equal than the arity of the constructor
pattern → function simplePat$_1$ ... simplePat$_m$
| constructor simplePat$_1$ ... simplePat$_{m'}$
| simplePat

% Patterns that can appear in the lhs of a function rule
fPat → nPlusk
| asPattern
| simplePat

% Simple patterns
simplePat → varId
| constructor
| function
| constant
| listPat
| tuplePat
| `(` pattern `)`

% Lists of patterns
listPat → [ ]
| (pattern (: pattern)$^+$)
| [pattern (, pattern)* [| pattern]]

% Tuples of patterns
tuplePat → (pattern (, pattern)*)

% (n+k) patterns
nPlusk → (varId + *Integer*)

% *as* patterns
asPattern → varId @ simplePat

% E X P R E S S I O N S

expr → simpleExpr
| appExpr

% Simple expressions
simpleExpr → doExpr
| list
| tuple
| intensionalExpr
| aExpr

% Applications
appExpr → aExpr expr*
| expr op expr

% Argument expressions: expressions that can be applied
% to some arguments
aExpr → aAtom
| section
| (expr)

% Appliable atoms
aAtom → varId
| constructor
| function

% Left and right sections
section → (simpleExpr funOp)
| (funOp simpleExpr)

% *do* expressions
doExpr → do { doDecl (; doDecl)* }
doDecl → varId <- expr
| expr

% Lists
list → [ ]
| (expr (: expr)+)
| [expr (, expr)* [| expr]]

% Tuples
tuple → (expr (, expr*)

% Intensional Lists
intensionalExpr → [expr || qual (, qual)*]
qual → varId <- expr
| pattern == expr
| expr

% C O N S T A N T S

constant → *Integer*
| *Float*
| *Char*
| *String*
| *Bool*

% B A S I C N O N - T E R M I N A L S

% Infix operators and constructors

op → funOp
| consOp
funOp → opFunId
| ‘ funId ‘
consOp → opConsId
| ‘ consId ‘

% Functions and constructors

constructor → consId
| (opConsId)
function → funId
| (opFunId)

% Identifiers

funId → *identifier*
consId → *identifier*
opConsId → *consOperator*
opFunId → *operator*
typeId → *identifier*
varId → *variable*

Appendix E

Type Inference

As we have seen in Section 2.2, $\mathcal{TOY}$ uses a type inference system based on the Damas-Milner polymorphic type system [7]. In this appendix we explain the type inference algorithm implemented in $\mathcal{TOY}$. This algorithm performs two main steps: first it makes a dependency analysis on function definitions to split the program into blocks of mutually dependent functions, and then it uses the resulting blocks to do perform type inference in the proper sense.

E.1 Dependency Analysis

When a function f is defined by using another function g we say that the f depends on g. It is quite natural to assume that in such case the type of f must be inferred before the type of g, of course, if g does not use f in its definition. But if we have a set of mutually recursive functions, then we must infer their types simultaneously. So, first of all we have to study the dependencies between the function definitions of the program to extract blocks of mutually dependent functions. We define the relation $<$ over function symbols of a program as:

$$f < g \Leftrightarrow \text{the definition of } g \text{ uses } f$$

This relation corresponds to a simple syntactical criteria: we have $f < g$ iff there is some defining rule of g that includes the symbol f in its right-hand side, or in its conditions, or in its local definitions. Consider now the reflexive and transitive closure$<^*$ of $<$, and the relation $\sim$ defined as: $f \sim g$ iff $f <^* g$ and $g <^* f$, i.e., the definitions of f and g are mutually dependent. The relation $\sim$ is an equivalence relation and we can consider the quotient set $FS/\sim$ (where FS is the set of function symbols of the program). We need to consider a sorted sequence $B_1, ..., B_n$ of equivalence classes of $FS/\sim$ in such a way that functions of B_i do not use functions of B_j for any $j > i$, i.e., let $B_1, ..., B_n$ a partition of FS such that:

i) $\{B_1, ..., B_n\} = FS/\sim$

ii) if $f \in B_i, g \in B_j$ and $j > i$ then $g \not<^* f$

In the sequel, we call *block* to each element B_i of this partition. The set of blocks $\{B_1, ..., B_n\}$ corresponds to the strongly connected components of a directed graph, in which nodes are labeled

with the function symbols of the program and arcs are defined by the relation $<$. Therefore, it is clear that any ordering of the blocks satisfies item *i)*. For satisfying item *ii)* we consider a new graph in which each node is labeled with a strongly connected component and arcs are defined by the relation $<$ extended to components: $B_i < B_j$ iff there exists $f \in B_i$ and $g \in B_j$ such that $f < g$. Notice that this is a directed acyclic graph and we can perform a topological sort on it in order to obtain an ordered sequence $B_1, ..., B_n$ which verifies item *ii)*. In practice, $\mathcal{TOY}$ implements both algorithms (they can be found e.g. in [6]) to obtain the ordered sequence of blocks.
As an example, consider the following program:

```
data nat = zero | suc nat

even :: nat -> bool
even zero = true
even (suc X) = odd X

odd :: nat -> bool
odd zero = false
odd (suc X) = even X

lstEven :: [nat] -> bool
lstEven []     = true
lstEven [X|Xs] = lstEven Xs <== even X == true
lstEven [X|Xs] = false      <== odd  X == true

from :: nat -> [nat]
from N = [N|from (suc N)]
```

In this program we have $FS = \{even, odd, lstEven, from\}$ and the dependences $odd < even$, $even < odd$, $even < lstEven$ and $odd < lstEven$. One possible sequence of blocks that satisfies *i)* and *ii)* is $\{even, odd\}$, $\{lstEven\}$, $\{from\}$. The functions *even* and *odd* are mutually recursive, so they must appear in the same block; the function $lstEven$ depends on *even*, so the block $\{lstEven\}$ must appear after the block that contains *even*; and as the function $from$ does not depend on any other function, it constitutes a single block. In fact, the block $\{from\}$ could appear at any position of the sequence.
The second and main step of the type inference algorithm takes as input the sequence of blocks $B_1, ..., B_n$ resulting from the dependency analysis.

E.2 Type Inference Algorithm

The sequence of blocks $B_1, ..., B_n$ determines the order in which type inference must be performed. The types of the functions belonging to each block B_j must be inferred simultaneously, after having inferred the types of all functions belonging to the preceding blocks B_i, $1 \leq i < j$. We distinguish two roles for occurrences of function symbols in the defining rules of the program:

an occurrence can have a *definition* role or it can have a *use* role. For example, the symbol f in the left-hand side of a defining rule for f plays a definition role and also every occurrence of f in the right-hand side, the conditions, or the local definitions of the rule (since the rule is part of the definition of f). The role of a symbol g in a defining rule of f depends on the blocks of the program: if f and g are both in the same block then g plays a definition role, while if they belong to different blocks then g has a use role. Intuitively, in the first case as f and g are mutually dependent, the definition of any of them is also part of the definition of the other and, as a consequence, the type of any of them depends on the type of the other. In the second case, if the rule of f contains the symbol g but both functions appear in different blocks, then the type of g must be fixed before inferring the type for f. In this case, the occurrence of g corresponds to a use of this function.
These ideas suggest the way in which the algorithm works. The inference is done sequentially block by block, from the first to the last one. While inferring types for functions of a block, any type information about those functions extracted from definitions is added to the type information of those functions in order to refine the information obtained previously. When a block is completely processed, the types inferred for its functions are fixed for the next blocks, in which those functions can appear with a use role.
The algorithm uses a *context* Δ which stores fixed type information obtained from previously processsed blocks, an *environment* Γ which stores temporary type information for functions in the current block, and an *environment* Θ which stores type information for variables in the current defining rule. The three stores have the same structure: a set of annotations of the form $s :: \tau$, meaning that τ is the type of the symbol s. The symbols s belong to different syntactic categories depending on the store where they occur. In Δ, they can be data constructors or defined function symbols from previously processed blocks. In Γ, they must be defined function symbols from the current block, and in Θ they must be variables from the current program rule. Moreover, the type variables occurring in Δ have an implicit universal quantification. For these reason, whenever a type assumption $s :: \tau$ is taken form Γ, the type variables occurring in τ can be renamed and eventually become bound to other types, if this is needed for subsequent type inferences.
Initially, the context Δ contains the principal types for the constructor symbols of predefined data types (such as $true, false, [\,], (:), ...$), the principal types for predefined functions (such as numeric functions $(+), (-), (div), ...$, equality and disequality functions $(==), (/=)$, and some others as $if\ \_\ then\ \_\ else\ \_$), and the principal types for constructor symbols of user defined data types (these types are directly obtained from the definition of data types). For example, for the previous program the initial context has the form:

$$\Delta = \{true :: bool, false :: bool, [\,] :: [A], (:) :: B \to [B] \to [B], zero :: nat, suc :: nat \to nat, ...\}$$

where he type variables A and B have an implicit universal quantification.
The type inference algorithm uses a function T for obtaining the type of a expression e, making use of the type annotations in Δ, Γ and Θ. A general call to the function T has the form $T(e, \Delta, \Gamma, \Theta)$, and it returns a pair (τ, θ), where τ is the type inferred for e, and θ is a *type substitution* variables which must be applied to enable the type inference. Type substitutions are just substitutions of types for type variables. Tey can be represented as sets of variable

bindings in the usual way; in particular, [ ] represents the empty type substitution. A call $T(e, \Delta, \Gamma, \Theta)$ expects that the three stores Δ, Γ, and Θ contain type annotations for all the symbols occurring in the expression e.
The formal definition of T works by recursion on the syntactic structure of the expression e. The main cases are presented below, assuming that X is a variable, $c \in DC$, $f \in FS$ and e, e' are expressions. We also use the notation $\theta_1\theta_2$ for the composition of two type substitutions, meaning that θ_2 is applied after θ_1.

- $T(X, \Delta, \Gamma, \Theta) = (\tau, [\,])$, if $X :: \tau \in \Theta$;
- $T(f, \Delta, \Gamma, \Theta) = (\tau, [\,])$, if $f :: \tau \in \Gamma$;
- $T(h, \Delta, \Gamma, \Theta) = (\tau', [\,])$, if $h :: \tau \in \Delta$ and τ' is a variable renaming of τ with fresh variables;
- $T((e_1, e_2), \Delta, \Gamma, \Theta) =$
  - $((\tau_1\theta_2, \tau_2), \theta_1\theta_2)$, if
    $T(e_1, \Delta, \Gamma, \Theta) = (\tau_1, \theta_1)$ and
    $T(e_2, \Delta, \Gamma\theta_1, \Theta\theta_1) = (\tau_2, \theta_2)$
- $T((e\ e'), \Delta, \Gamma, \Theta) =$
  - $(A\theta, \theta_1\theta_2\theta)$, if
    $T(e, \Delta, \Gamma, \Theta) = (\tau_1, \theta_1)$,
    $T(e', \Delta, \Gamma\theta_1, \Theta\theta_1) = (\tau_2, \theta_2)$, and
    θ is a m.g.u. for $\tau_1\theta_2$ and $\tau_2 \to A$ (where A is a fresh type variable)
  - $(A, [\,])$, otherwise (with a fresh type variable A). In this case there is some type error in the expression $(e\ e')$.

The first three cases are quite easy to understand; they just consult the stores. Notice that the types taken from the context Δ are renamed with fresh type variables, due to the implicit universal quantification of all the type variables occurring in Δ. The fourth case infers a product type for an expression which represents an ordered pair. Similar cases should be added for n-tuples, $n > 2$.
The most interesting case is the last one, dealing with an application $(e\ e')$. In order to infer the type of this expression, T obtains types for e and e' and then tries to unify the type of e with a functional type, in such a way that e can be applied to e'. This is done with the help of a new type variable A and type unification. Notice that the substitution θ_1 obtained from typing e is applied to Γ and Θ when typing e', but Δ is untouched. The final substitution returned by T is the composition of the three substitutions obtained during the process. If some type error has appeared when typing e or e', or the the m.g.u. θ does not exist, then a type error in the application $(e\ e')$ has been detected. In this case T returns a pair consisting of a fresh type variable and the empty type substitution. This behaviour supports an error recovering policy: since the fresh variable A represents the most general type for an expression, the error is not propagated; moreover, the substitution [ ] guarantees that the stores are not be affected

BUILD THE INITIAL CONTEXT Δ

FOR i=1 TO n DO
  LET Γ BE THE SET OF ANNOTATIONS $f :: A_f$ FOR EACH $f \in B_i$
  FOR EACH DEFINING RULE $R \equiv l = e \Leftarrow CD$ OF $f \in B_i$ DO
    LET Θ BE THE SET OF $X :: A_X$ FOR EACH $X \in var(R)$
    LET $T(l, \Delta, \Gamma, \Theta) = (\tau_l, \theta_l)$
    $(\Gamma, \Theta) \leftarrow (\Gamma\theta_l, \Theta\theta_l)$
    LET $T(e, \Delta, \Gamma, \Theta) = (\tau_e, \theta_e)$
    $(\Gamma, \Theta) \leftarrow (\Gamma\theta_e, \Theta\theta_e)$
    IF THERE EXISTS A M.G.U. θ OF $\tau_l\theta_e$ AND τ_e THEN
      $(\Gamma, \Theta) \leftarrow (\Gamma\theta, \Theta\theta)$
      FOR EACH CONDITION OR LOCAL DEFINITION $e' \diamond e''$ IN CD DO
        LET $T(e', \Delta, \Gamma, \Theta) = (\tau_{e'}, \theta_{e'})$
        $(\Gamma, \Theta) \leftarrow (\Gamma\theta_{e'}, \Theta\theta_{e'})$
        LET $T(e'', \Delta, \Gamma, \Theta) = (\tau_{e''}, \theta_{e''})$
        $(\Gamma, \Theta) \leftarrow (\Gamma\theta_{e''}, \Theta\theta_{e''})$
        IF THERE EXISTS A M.G.U. θ' OF $\tau_{e'}$ AND $\tau_{e''}$ THEN
          $(\Gamma, \Theta) \leftarrow (\Gamma\theta', \Theta\theta')$
        ELSE ERROR IN $e' \diamond e''$
    ELSE ERROR IN RULE R
  FOR EACH $f \in B_i$ DO
    IF f HAS A USER-DEFINED TYPE τ_f THEN
      LET $T(f, \Delta, \Gamma, \Theta) = (\tau'_f, [])$
      IF THERE EXISTS σ SUCH THAT $\tau_f = \tau'_f\sigma$ THEN
        $\Gamma \leftarrow (\Gamma - \{f :: \tau'_f\}) \cup \{f :: \tau_f\}$
      ELSE ERROR IN INFERRED/DECLARED TYPES FOR f
  $\Delta \leftarrow \Delta \cup \Gamma$

Table E.1: Inference Algorithm

by the error. The real implementation, as a collateral effect, produces an error message with appropriate information for the user.
Using the function T we can now present a type inference algorithm which starts with the sequence of blocks $B_1, ..., B_n$ resulting from the dependency analysis of a program. The algorithm is shown in Table E.1, using the notation $a \leftarrow c$ for assignment. In order to simplify the presentation, we assume that the CD part of a program rule $R \equiv l = e \Leftarrow CD$ includes both conditions and local definitions. This makes sense, because any condition or local definition has the syntactic form $e' \diamond e''$ with two 'sides' e' and e''. We write $var(R)$ for the set of variables occurring in R.
The main loop proceeds block by block from B_1 to B_n. For each block, the algorithm builds an environment Γ whose type annotations represent the types inferred for all the functions in the block. First, Γ is initialized by assumming a fresh type variable as the type of each function. Then, the program rules in the current block are processed one by one. The types of the left-hand and the right-hand sides are computed by means of the function T, and an attempt to unify them is made. Any internal type error in the left-hand side or the right-hand side is detected by T. In the case that unification succeeds, the algorithm checks the conditions and the local definitions of the program rule, trying to unify the types obtained for its two sides. After any application of T the environments Γ and Θ are appropriately updated. After processing all the program rules in the current block, Γ contains the inferred types for all the functions in the block. At this point, the algorithm compares the inferred types with the types declared by the user, if available. If there is no declared type, then the inferred type remains in Γ. If there is a declared type which is an instance of the inferred one, then the declared type is plced in Γ and the system emits a warning. If the declared type is not an instance of the inferred one, then the system signals an error, pointing to the inconsistency between the declared and inferred types. Finally, before processing the next block, the algorithm updates the context Δ by adding all the type annotations in Γ to it. At the end, the context Δ contains the principal types inferred for all the functions occurring in the program (or instances of them, if the user has provieded some less general type declarations for some functions).
Now, let us see how the full process works in the example shown in Section E.1. We had the three blocks $\{even, odd\}$, $\{lstEven\}$, an $\{from\}$, and the initial context was:

$$\Delta = \Delta_0 = \{true :: bool, false :: bool, [\,] :: [A], (:) :: B \to [B] \to [B], zero :: nat, suc :: nat \to nat, ...\}$$

For processing the first block we initially build the environment:

$$\Gamma = \{even :: C, odd :: D\}$$

Then we process the four program rules for *even* and *odd* in their textual order. In fact, the order chosen for processing the program rules within a fixed block is not relevant. For the rule *even zero = true* we consider $\Theta = \{\}$. The call $T(even, \Delta, \Gamma, \Theta)$ performs two recursive calls: $T(even, \Delta, \Gamma, \Theta)$ that produces the pair $(C, [\,])$, and $T(zero, \Delta, \Gamma, \Theta)$ that returns $(nat, [\,])$. Then we unify C with $nat \to E$, this type is annotated in Γ as the type of *even*, and the first call to T for the expression *even zero* returns $(E, [C \mapsto (nat \to E)])$. For the right-hand side, the call

$T(true, \Delta, \Gamma, \Theta)$ returns $(bool, [\,])$ and the types inferred for the left-hand side and the right-hand (i.e., E and $bool$) are unified. Finally we obtain:

$$\Gamma = \{even :: nat \to bool, odd :: D\}$$

For the second rule *even (suc X) = odd X* the initial environment $\Theta = \{X :: F\}$ is created. From the expression *suc X*, the type *nat* is inferred for X. The type of the left-hand side, inferred from Γ, is *bool*. Regarding the right-hand side, the call $T((odd\ X), \Delta, \Gamma, \Theta)$ unifies (through recursive calls) the type D asssumed for *odd* in Γ with $nat \to G$, returning G as the type for *odd X*. Then, by unification of types of the left-hand side and the right-hand side, we obtain $odd :: nat \to bool$. This yields:

$$\Gamma = \{even :: nat \to bool, odd :: nat \to bool\}$$

The two rules for *odd* do not supply new information about the type annotations in Γ and do not produce any error. Finally, the algorithm contrasts the inferred types for *even* and *odd* with the declared types, that are identical in this case. The types for the first block are now inferred, and we can update Δ, obtaining:

$$\Delta = \Delta_0 \cup \{even :: nat \to bool, odd :: nat \to bool\}$$

For the second block $\{lstEven\}$, the initial environment is:

$$\Gamma = \{lstEven :: H\}$$

From the first program rule we obtain $lstEven :: [I] \to bool$, where the variable I is obtained by renaming the type $[\,] :: [A]$, taken from Δ. Then, checking the condition $even\ X == true$ leads to the refined type assumption $lstEven :: [nat] \to bool$. The second rule for `lstEven` rule does not supply additional information and does not produce any error. The updated context after processing the second block is then:

$$\Delta = \Delta_0 \cup \{even :: nat \to bool, odd :: nat \to bool, lstEven :: nat \to bool\}$$

Finally, for the last block $\{from\}$ the inferred type $from :: nat \to [nat]$ is the same as the declared one, and type inference ends up with the following context:

$$\Delta = \Delta_0 \cup \{even :: nat \to bool, odd :: nat \to bool, lstEven :: nat \to bool, from :: nat \to [nat]\}$$

$\mathcal{TOY}$ stores this final context Δ in order to perform type checking of the different goals proposed by the user in an interactive session. As explained in Section 2.13, goals are sequences of conditions. Therefore, type checking of goals can be performed by following the innermost loop of the type inference algorithm, using an empty Γ environment and with no modification of the context Δ.

Appendix F

Declarative Semantics

The semantics of programs written in a declarative programming language can be explained in terms of logical deduction. This so-called *declarative semantics* allows to separate **what** programs do (the logical meaning) from **how** it is done (the complex operational execution model). In this appendix we present the core ideas of $\mathcal{TOY}$'s declarative semantics. More technical details can be found in [11] for first-order programs, and in [13] for the higher-order case.

F.1 Motivation

Declarative semantics originated in the field of pure logic programming, where programs are given as sets of definite Horn clauses. In this setting, the logical meaning of a program $\mathcal{P}$ can be modelled as the set of atomic facts $p(t_1, \ldots, t_n)$ which can be derived from $\mathcal{P}$ in the logical calculus HL (for Horn Logic) consisting of one single inference rule:

$$\frac{q_i(t_{i,1}, \ldots, t_{i,m_i})\ (1 \leq i \leq k)}{p(t_1, \ldots, t_n)}$$

IF $p(t_1, \ldots, t_n)$ `<==` $q_1(t_{i,1}, \ldots, t_{1,m_1}), \ldots, q_k(t_{i,1}, \ldots, t_{k,m_k})$
is an instance of some clause in $\mathcal{P}$

HL is a very simple inference system, which reflects the logical reading of a program clause as a universally quantified implication. Atomic facts play a key rôle here because program clauses behave as definitions of predicates. In the case of a multiparadigm declarative language like $\mathcal{TOY}$, program clauses are replaced by defining rules for functions, which are lazy and possibly non-deterministic, as we have seen in Chapter 2. Therefore, a declarative semantics for $\mathcal{TOY}$ must rely on some suitable generalization of atomic facts. Following [13] we consider atomic facts of the form $f\, t_1 \ldots t_n \to t$, whose intended meaning is: "t represents one of the possible approximations of the result returned by a call to the n-ary function f, provided that $t_1, \cdots, t_n$ represent finite approximations of f's arguments." More precisely, in any atomic fact $f\, t_1 \ldots t_n \to t$, $f \in FS^n$ must be a defined function symbol of arity n and t_i, t must be *patterns* with abstract syntax

$$t ::= \bot \mid X \mid (t_1, \ldots, t_n) \mid c\, t_1 \ \ldots\ t_m \ \ (c \in DC^n,\ 0 \leq m \leq n) \mid$$
$$f\, t_1 \ \ldots\ t_m \ \ (f \in FS^n,\ 0 \leq m < n)$$

This syntax is almost the same as the one already introduced in Section 2.1, except that the special symbol $\bot$ (read *bottom*) is now allowed to occur in patterns. The symbol $\bot$ represents an undefined value. A pattern t is called *total* if there is no occurrence of $\bot$ in t, and *partial* otherwise. Partial patterns can be understood as a representation of partially known values, where the occurrences of $\bot$ stand for the unknown parts. Therefore, the result of replacing (some or all) occurrences of $\bot$ in a given pattern is a more defined pattern. More precisely, the notation $t \sqsubseteq t'$ (read: t approximates t'; or t' is more defined than t) indicates a partial order between patterns which holds iff t' can be obtained from t by replacing (some or all occurrences) of $\bot$ by other patterns. This partial order is called the *semantic ordering* between patterns. For instance, using the list constructor `(:)` and integer constants (viewed as nullary constructors) we can build the infinite list of patterns, which is increasing w.r.t. $\sqsubseteq$:

$$\bot \ \sqsubseteq\ 0 : \bot \ \sqsubseteq\ 0 : 1 : \bot \ \sqsubseteq\ 0 : 1 : 2 : \bot \ \sqsubseteq\ \cdots$$

The patterns in this sequence represent more and more defined approximations of the infinite list of all non-negative integers, which is the expected result of the function call `from 0` (remember the definition of `from` given in Section 2.6). Therefore, the atomic facts `from 0` $\rightarrow$ `0:1:...:n-1:`$\bot$ (for all $n \geq 0$) belong to the declarative semantics of any $\mathcal{TOY}$ program which includes the definition of `from`. This example shows that partial patterns are essential to model the semantics of lazy functions, whose returned results can be potentially infinite.

Atomic facts, in the sense of pure logic programming, can be represented in the form

$$p(t_1, \ldots, t_n) \rightarrow \mathit{true}$$

By convention, this may be abbreviated as $p(t_1, \ldots, t_n)$. For instance, the declarative semantics of the Prolog-like $\mathcal{TOY}$ program presented in Section 2.11 includes (among others) the atomic fact `ancestorOf("alice","john")`.

The declarative semantics we are aiming at is also able to reflect non-determinism. As an example, recall the non-deterministic nullary functions `coins, rcoins :: [int]` defined in Section 2.12. Their declarative semantics contains the following atomic facts:

`coins` $\rightarrow 0 : \bot$; `coins` $\rightarrow 1 : \bot$;
`coins` $\rightarrow 0 : 0 : \bot$; `coins` $\rightarrow 0 : 1 : \bot$; `coins` $\rightarrow 1 : 0 : \bot$; `coins` $\rightarrow 1 : 1 : \bot$;
$\ldots$
`rcoins` $\rightarrow 0 : \bot$; `coins` $\rightarrow 0 : 0 : \bot$;
`rcoins` $\rightarrow 1 : \bot$; `rcoins` $\rightarrow 1 : 1 : \bot$;
$\ldots$

but not `rcoins` $\rightarrow 0 : 1 : \bot$ neither `rcoins` $\rightarrow 1 : 0 : \bot$. This reflects the behavioural difference between `coins` and `rcoins` which was intended in their respective definitions. Recall the informal discussion on *call-time choice* semantics of non-deterministic functions, in Section 2.12.

Formally, the declarative semantics of a $\mathcal{TOY}$ program $\mathcal{P}$ is modelled as the set of all the atomic facts $f\ t_1\ \cdots\ t_n \rightarrow t$ which can be derived from $\mathcal{P}$ in the Constructor-based ReWriting Logic CRWL presented in the next section. In the case of pure logic programs, CRWL behaves just as Horn Logic. As seen from the examples in this motivating section, the logical meaning of $\mathcal{TOY}$

programs given by CRWL lacks information on computed answers. In a way, this is a limitation. Note, however, that any semantics which provides information on computed answers must be more dependent on some particular operational model for goal solving. CRWL semantics is more abstract in this sense, and still useful for various purposes, as e.g. investigating the completeness of goal-solving methods (see [11, 13]), or designing provably correct declarative debugging tools, like those described in Chapter 6.

F.2 A Constructor-Based Rewriting Logic

As explained in the previous section, the main aim of CRWL is to derive atomic facts from a program. However, due to the occurrence of nested function calls in defining rules (see Section 2.6), CRWL must be able to derive more general facts of the form $e \to t$, where t is a possibly partial pattern and e is a possibly partial expression. These facts are called *approximation statements*. The abstract syntax for partial expressions is like the one introduced for expressions in Section 2.1, extended to allow occurrences of $\bot$. Moreover, due to the occurrence of conditions in defining rules, CRWL must also be able to derive conditions. For the moment we consider only strict equality conditions $e_1 == e_2$; the extension of CRWL to deal with other kinds of conditions will be briefly discussed in Section F.5.
CRWL can be presented as an inference system consisting of the six inference rules displayed below. In each of them, the premises and the conclusion are approximation statements, except in the case of the inference **SE**, where the conclusion is a strict equality condition.

BT Bottom: $\dfrac{}{e \to \bot}$ **VR** Variable: $\dfrac{}{X \to X}$

TD Tuple Decomposition: $\dfrac{e_1 \to t_1,\ \ldots,\ e_n \to t_n}{(e_1,\ \ldots,\ e_n) \to (t_1,\ \ldots,\ t_n)}$

PD Pattern Decomposition: $\dfrac{e_1 \to t_1,\ \ldots,\ \ e_m \to t_m}{h\, e_1\, \ldots\, e_m \to h\, t_1\, \ldots\, t_m}$

FC Function Call: $\dfrac{e_1 \to t_1,\ \ldots,\ e_n \to t_n,\ C,\ r\, a_1\, \ldots\, a_k \to t}{f\, e_1\, \ldots\, e_n\, a_1\, \ldots\, a_k \to t}\ (k \geq 0)$

IF $(f\, t_1\, \ldots\, t_n \to r \Leftarrow C) \in [\mathcal{P}]_\bot$

SE Strict Equality: $\dfrac{e_1 \to t,\ e_2 \to t}{e_1 == e_2}$

IF t is a **total** pattern

In all the rules, t_i stand for possibly partial patterns, while e_i, a_j stand for possibly partial expressions. In rule **FC** the notation $[\mathcal{P}]_\bot$ refers to the set of all the possible instances of program rules, where each particular instance is obtained from some function defining rule in $\mathcal{P}$, by some substitution of (possibly partial) patterns in place of variables. Moreover, the formulation of rule **FC** must be understood assumming that C includes both strict equality conditions $e_1 == e_2$ and approximation statements $d \to s$ in place of the local definitions $s = d$ occurring in the defining rule. This means that CRWL specifies the semantics of local definitions as a particular case of approximation statements.

A derivation in CRWL can be presented as a finite sequence of statements φ_i, $0 \leq i \leq n$, such that each statement φ_i $(0 \leq i \leq n)$ follows from some other statements φ_j $(i < j \leq n)$, by means of some CRWL inference rule. A statement φ is called CRWL-provable from a program $\mathcal{P}$ (in symbols, $\mathcal{P} \vdash_{CRWL} \varphi$) iff there is some CRWL-derivation such that φ_0 is φ, using only rule instances from the set $\mathcal{P}$ at the **FC** steps.
The declarative semantics of a $\mathcal{TOY}$ program $\mathcal{P}$ is modelled by the set of all atomic facts $f\ t_1\ \cdots\ t_n \to t$ such that $\mathcal{P} \vdash_{CRWL} f\ t_1\ \cdots\ t_n \to t$. For example, assume any program $\mathcal{P}$ including the defining rules for `coin` and `coins` given in Section 2.12. Then, $\mathcal{P} \vdash_{CRWL}$ `coins` $\to$ `0:1:`$\bot$ is proved by the following CRWL derivation:

0. `coins` $\to$ `0:1:`$\bot$ by **FC**, 1.
1. `coin:coins` $\to$ `0:1:`$\bot$ by **PD**, 2,3.
2. `coin` $\to$ `0` by **FC**, 4.
3. `coins` $\to$ `1:`$\bot$ by **FC**, 5.
4. `0` $\to$ `0` by **PD**.
5. `coin:coins` $\to$ `1:`$\bot$ by **PD**, 6,7.
6. `coin` $\to$ `1` by **FC**, 8.
7. `coins` $\to \bot$ by **BT**.
8. `1` $\to$ `1` by **PD**.

Note that all the CRWL inference rules, with the exception of **FC**, are program independent. In particular, the rules **BT**, **VR**, **TD**, and **PD** essentially specify the semantic ordering. Considering the empty program $\emptyset$, and any approximation statement $t' \to t$ where t' and t are both patterns, it is easy to check that $\emptyset \vdash_{CRWL} t' \to t$ holds if and only if $t \sqsubseteq t'$. Some other technical properties of CRWL can be found in [11, 13].

F.3 Correctness of Computed Answers

As we said already in Section F.1, CRWL semantics provides no direct information about computed answers. Nevertheless, CRWL can be used to specify the logical correctness of computations. More precisely, assume a substitution σ (of patterns for variables) computed by the $\mathcal{TOY}$ system as answer for a goal G which consists of strict equality conditions, using a program $\mathcal{P}$. Correctness of σ means that $\mathcal{P} \vdash_{CRWL} G\sigma$ (i.e., $\mathcal{P} \vdash_{CRWL} \varphi\sigma$ for each strict equality φ in G). This correctness criterion can be used as a basis to prove soundness and completeness of formally specified goal-solving mechanisms; see [11, 13]. The case of goals including other kinds of conditions can be treated analogously with the extensions of CRWL mentioned in Section F.5.

F.4 Models

In the pure logic programming field, declarative semantics also includes *models*. A model of a logic program $\mathcal{P}$ is any interpretation of the predicate symbols over a certain domain, so that all the clauses in $\mathcal{P}$ become true when interpreted as universally quantified implications. In particular, models whose domain is the set of all the *data terms* (see Section 2.1) are called *open Herbrand models*; such models can be represented as sets of atomic facts, closed under arbitrary substitutions of data terms for variables. A well-known result of logic programming theory says that the set $\mathcal{M}_\mathcal{P} = \{p(t_1, \ldots, t_n) \mid \mathcal{P} \vdash_{HL} p(t_1, \ldots, t_n)\}$ (i.e., the set of all the atomic facts that can be derived from $\mathcal{P}$ in Horn logic) is an open Herbrand model of $\mathcal{P}$, and even the *least* open Herbrand model (w.r.t. set inclusion).
All this can be generalized to the case of multiparadigm declarative languages like $\mathcal{TOY}$. Here we limit ourselves to introduce the analog of open Herbrand models; results about more general models can be found in [11, 13]. Given a $\mathcal{TOY}$ program $\mathcal{P}$, an open Herbrand interpretation is defined as any set $\mathcal{I}$ of atomic facts $f\, t_1 \ldots t_n \to t$ which satisfies the three following closure properties:

1. $(f\, t_1 \ldots t_n \to \bot) \in \mathcal{I}$, for all $f \in FS^n$ and all (possible partial) patterns t_i $(1 \leq i \leq n)$.
2. IF $(f\, t_1 \ldots t_n \to t) \in \mathcal{I}$ and $t_i \sqsubseteq t'_i$ $(1 \leq i \leq n)$ and $t \sqsupseteq t'$ THEN $(f\, t'_1 \ldots t'_n \to t') \in \mathcal{I}$.
3. IF $(f\, t_1 \ldots t_n \to t) \in \mathcal{I}$ and θ is any substitution of **total** patterns for variables
   THEN $(f\, t_1 \ldots t_n \to t)\theta \in \mathcal{I}$

Given an open Herbrand interpretation $\mathcal{I}$ and any statement φ, let us use the notation $\mathcal{I} \vDash \varphi$ (reads: "φ is *valid* in $\mathcal{I}$") to indicate that φ can be derived in the inference system consisting of all the CRWL inference rules, except **FC**, which is replaced by the following inference rule:

$$\textbf{AF}_\mathcal{I} \text{ Atomic Fact in } \mathcal{I}: \qquad \frac{e_1 \to t_1,\ \ldots,\ e_n \to t_n,\ r\, a_1 \ldots a_k \to t}{f\, e_1 \ldots e_n\, a_1 \ldots a_k \to t}\ (k \geq 0)$$

$$\text{IF } (f\, t_1 \ldots t_n \to r) \in \mathcal{I}$$

An open Herbrand interpretation $\mathcal{I}$ is called a *model* of a program $\mathcal{P}$ iff every program rule instance $(f\, t_1 \ldots t_n \to r \Leftarrow C) \in [\mathcal{P}]_\bot$ is true in $\mathcal{I}$ when interpreted as a conditional implication; more precisely, the following must hold:

IF $\quad \mathcal{I} \vDash C$ (i.e., $\mathcal{I} \vDash \varphi$ for all $\varphi \in C$)
THEN $(f\, t_1 \ldots t_n \to t) \in \mathcal{I}$ for every pattern t such that $\mathcal{I} \vDash r \to t$.

Using this notion of model, it can be shown that $\mathcal{M}_\mathcal{P} = \{f\, t_1 \ldots t_n \to t \mid \mathcal{P} \vdash_{CRWL} f\, t_1 \ldots t_n \to t\}$ (i.e., the set of all the atomic facts that can be derived from $\mathcal{P}$ in CRWL) is an open Herbrand model of $\mathcal{P}$, and even the *least* open Herbrand model (w.r.t. set inclusion). This is a nice generalization of the classical results known for pure logic programs.
Models are a useful tool for various purposes, including program analysis, verification and debugging. In particular, the foundations of $\mathcal{TOY}$'s declarative debugger rely on the comparison between the least open Herbrand model $\mathcal{M}_\mathcal{P}$ and another open Herbrand model $\mathcal{I}$ which plays the rôle of the *intended model* of $\mathcal{P}$. The observation of some wrong computed answer σ for some goal G indicates that $\mathcal{M}_\mathcal{P} \not\subseteq \mathcal{I}$. Since $\mathcal{M}_\mathcal{P}$ is the least open Herbrand model of $\mathcal{P}$, it follows

that $\mathcal{I}$ is not a model of $\mathcal{M}$. Therefore, some program rule instance must be false in $\mathcal{I}$. As explained in Chapter 6, the debugger locates such a wrong rogram rule instance by searching a computation tree. In fact, the computation tree represents the **FC** inference steps in a CRWL derivation which proves $\mathcal{P} \vdash_{CRWL} G\sigma$. All the other CRWL inferences can be ignored by the debugger, since they are program independent and cannot cause logical errors. See [4, 5] for more details.

F.5 Extensions

The presentation of CRWL in Section F.2 and [11, 13] is limited to the derivation of approximation statements and strict equality conditions. However, $\mathcal{TOY}$ programs can also use other kinds of conditions, including disequality and arithmetic constraints (see Sections 2.6 and 2.16). In order to obtain a declarative semantics for such programs, CRWL must be extended to a more powerful inference system (CCRWL, say) with the ability to derive *conditional statements* of the form $\sigma \Rightarrow \varphi$, where σ is a finite conjunction of atomic conditions and φ is either an approximation statement or an atomic condition. The definition of CCRWL is currently ongoing work. Our aim is to characterize the declarative semantics of a program $\mathcal{P}$ as the set of all the *conditional basic facts* of the form $\sigma \Rightarrow f\, t_1 \ldots t_n \to t$ such that $\mathcal{P} \vdash_{CCRWL} \sigma \Rightarrow f\, t_1 \ldots t_n \to t$. This will naturally lead to a logical criterion of correctness for computed answers including constraints. For instance, in Section 2.13 whe have already met the computed answer `L == [2,X,3] {X>=2.0} {X=<3.0}` for the goal `permutationSort [3,2,X] == L`. The logical correctness of this computation means that CCRWL should be able to derive the conditional statement

`L == [2,X,3]` $\wedge$ `X>=2.0` $\wedge$ `X=<3.0` $\Rightarrow$ `permutationSort [3,2,X] == L`.

We also expect CCRWL to provide a logical basis for the declarative debugging of $\mathcal{TOY}$ programs with constraints, thus eliminating one limitation of the current debugger (see Chapter 6).

Appendix G

Operational Semantics

We define the operational semantics of $\mathcal{TOY}$ by means of a translation function $\mathcal{T}$: **Toy** $\rightarrow$ **Prolog**, where **Toy** and **Prolog** are the sets of $\mathcal{TOY}$ programs and *Prolog* programs, respectively. The translation function, which constitutes the core of the implementation of $\mathcal{TOY}$, is the result of composing three intermediate functions

$$\mathcal{T} = pc \circ sf \circ fo$$

where

- fo (which stands for *first-order*) translates source $\mathcal{TOY}$ programs, which use higher-order syntax, into $\mathcal{TOY}$-like programs written in first-order syntax.
- sf (which stands for *suspended forms*) introduces *suspensions* into first-order $\mathcal{TOY}$ programs, a technicality useful for achieving *sharing* in the execution of $\mathcal{TOY}$ programs.
- pc (which stands for *prolog code*) generates properly the Prolog clauses which are the final result of the translation. The generation of Prolog code is made as to implement a *demand-driven strategy* for lazy narrowing, according to the ideas presented in [22]. A crucial step for the code generation will be the construction of the *definitional trees* of all the functions defined in the $\mathcal{TOY}$ program.

We now give the technical details of all the above mentioned functions.

G.1 First-Order Translation of $\mathcal{TOY}$ Programs

It consists essentially in the three following changes:

- Enhancing the signature with new constructor symbols for expressing partial applications of constructor or function symbols, and adding a new function symbol @ (read *apply*).
- Translating all program rules by writing all applications in first-order syntax, using the new symbols of the enhanced signature whenever necessary.
- Adding program rules for the new function symbol @ .

The details of this translation are given below. The semantic correctness of the translation has been justified in [12].

G.1.1 Enhancing the Signature

Given a signature $\Sigma = DC \cup FS$, the *first-order signature* corresponding to Σ is $\Sigma_{fo} = DC_{fo} \cup FS_{fo}$, where

$$\begin{array}{lcl} FS_{fo} & = & FS \cup \{@\} \\ DC_{fo} & = & \bigcup_{c \in DC^n, n \in \mathbb{N}} \{c_0, \ldots, c_n\} \ \cup \ \bigcup_{f \in FS^n, n \in \mathbb{N}} \{f_0, \ldots, f_{n-1}\} \end{array}$$

where $\{c_0, \ldots, c_n, f_0, \ldots, f_{n-1}\}$ are new symbols. The idea is that in first-order syntax we use the symbol c_i whenever in higher-order syntax we encounter the symbol c applied to i arguments, and similarly for f.

G.1.2 First-Order Translation of Program Rules and Rules for @

Let P be a $\mathcal{TOY}$ program consisting of the rules $R_1, \ldots, R_n$. Then the first-order translation of P, $fo(P)$, is defined as follows

$$\begin{array}{lcl}
fo(P) & = & \{fo(R_1), \ldots, fo(R_n)\} \cup @ - Rules \\
fo(l = e \Leftarrow C) & = & fo(l) = fo(e) \Leftarrow fo(C) \\
fo(X) & = & X \\
fo(n) & = & n \\
fo((e_1, \ldots, e_n)) & = & (fo(e_1), \ldots, fo(e_n)) \\
fo(c\ e_1 \ldots e_k) & = & c_k(fo(e_1), \ldots, fo(e_k)), \text{ if } c \in DC^n, k \leq n \\
fo(f\ e_1 \ldots e_k) & = & f_k(fo(e_1), \ldots, fo(e_k)), \text{ if } f \in FS^n, k < n \\
fo(f\ e_1 \ldots e_n) & = & f(fo(e_1), \ldots, fo(e_n)), \text{ if } f \in FS^n \\
fo(f\ e_1 \ldots e_n\ e_{n+1} \ldots e_{n+k}) & = & @(\ldots @(f(fo(e_1), \ldots, fo(e_n)), fo(e_{n+1})), \ldots), fo(e_{n+k})), \\
 & & \text{if } f \in FS^n, k > 0 \\
fo(X\ e_1 \ldots e_k) & = & @(\ldots @(X, fo(e_1)), \ldots), fo(e_k)) \\
fo(e_1 \diamond e_2) & = & fo(e_1) \diamond fo(e_2), \text{ if } \diamond \in \{==, /=, <, <=, >, >=\} \\
\% \ The\ @ - Rules & & \\
@(c_k(X_1, \ldots, X_k), Y) & = & c_{k+1}(X_1, \ldots, X_k, Y), \text{ if } c \in DC^n, k \leq n \\
@(f_k(X_1, \ldots, X_k), Y) & = & f_{k+1}(X_1, \ldots, X_k, Y), \text{ if } f \in FS^n, k+1 \leq n \\
@(f_{n-1}(X_1, \ldots, X_{n-1}), Y) & = & f(X_1, \ldots, X_{n-1}, Y), \text{ if } f \in FS^n
\end{array}$$

Remarks

1. The real implementation does not need to use different symbols for $c_0, c_1, \ldots, c_n$ (or $f_0, f_1, \ldots, f_n$ resp.). Instead, c (f resp.) is used to represent all of them. This can be done since Prolog allows the use of the same constructor with different arities.

2. Goals must also be translated to first-order syntax, in the same way as conditions of program rules are translated, that is

$$fo(\ldots, e_1 \diamond e_2, \ldots) = \ldots, fo(e_1) \diamond fo(e_2), \ldots, \quad \text{if } \diamond \in \{==, /=, <, <=, >, >=\}$$

3. The first-order translation fo determines indeed a way of representing as Prolog terms $\mathcal{TOY}$ programs (program rules, more precisely), expressions, and goals.

G.2 Introduction of Suspensions

We assume a $\mathcal{TOY}$ program P, translated to first-order syntax, consisting of the rules $R_1, \ldots, R_n$. The idea of *suspensions* is to replace each subexpression in P with the shape of a function call $f(e_1, \ldots, e_n)$ by a Prolog term of the form $susp(f(e_1, \ldots, e_n), R, S)$ (called a suspension) where R and S are initially (i.e., at the time of translation) new Prolog variables. If during execution the computation of a head normal form for $f(e_1, \ldots, e_n)$ is required, the variable R will be used to get the obtained value, and we say that the suspension has been evaluated. The argument S in a suspension is a flag to indicate if the suspension has been evaluated or not. Initially S is a variable (indicating a non-evaluated suspension), which is set to the value hnf once the suspension is evaluated (to head normal form).
The result of introducing suspensions into first-order $\mathcal{TOY}$ programs is formally defined by means of the following function fs:

$$\begin{array}{lcl}
fs(P) & = & \{fs(R_1), \ldots, fs(R_n)\} \\
fs(f(t_1, \ldots, t_n)) = e \Leftarrow C) & = & f(t_1, \ldots, t_n) = fs(e) \Leftarrow fs(C) \\
fs(X) & = & X \\
fs(n) & = & n \\
fs((e_1, \ldots, e_n)) & = & (fs(e_1), \ldots, fs(e_n)) \\
fs(c(e_1, \ldots, e_n)) & = & c(fs(e_1), \ldots, fs(e_n)) \\
fs(f(e_1, \ldots, e_n)) & = & susp(f(fs(e_1), \ldots, fs(e_n)), R, S) \\
fs(e_1 \diamond e_2) & = & fs(e_1) \diamond fs(e_2)
\end{array}$$

G.3 Prolog Code Generation

We now define the function pc which, given a $\mathcal{TOY}$ program (translated to first-order and with suspensions), returns a Prolog program which implements (equality and disequality) constraint solving by means of lazy narrowing with the demand-driven strategy described in [22].
We assume a (first-order with suspensions) TOY program P for a signature $\Sigma = DC \cup FS$. The Prolog code for P, $pc(P)$, is defined as

$$pc(P) = \mathsf{Solve} \cup \mathsf{Equal}_{DC} \cup \mathsf{NotEqual}_{DC} \cup \mathsf{Hnf}_{\Sigma} \cup \mathsf{Prolog}_{FS}$$

where $\mathsf{Solve}, \mathsf{Equal}_{DC}, \mathsf{NotEqual}_{DC}, \mathsf{Hnf}_{\Sigma}, \mathsf{Prolog}_{FS}$ are sets of Prolog clauses to be defined below. For the sake of clarity, we present here in most occasions a simplification of the more optimized sets of clauses used in the actual implementation. For a more detailed presentation, we refer to [32] or $\mathcal{TOY}$'s source code, specially the file toycomm.pl.
It will be apparent below that in some cases the code depends on the signature, having different but similar clauses for different constructor or function symbols. Whenever we write c of f in the clauses below, it indicates that a different clause must be considered for each constructor

symbol $c \in DC$ or function symbol $f \in FS$. The actual code might be more compact by using metapredicates to recognize and decompose the structure of expressions.

G.3.1 Clauses for Goal Solving (Solve)

A goal (or constraint) takes the form $e_1 \diamondsuit e'_1, \ldots, e_n \diamondsuit e'_n$, where $\diamondsuit$ can be $==$ or $/=$. As answer for a goal the system produces a collection of constraints in solved form.
Solved forms for equality constraints can be handled as substitutions, and the system indeed does it. Substitutions are managed by means of Prolog unification, and therefore there is no need of explicit management of substitutions in the Prolog code.
Things are different for the case of disequality constraints, which are explicitly managed in the code by means of a *disequality constraint store* with the form of a list $[X : [t_1, \ldots, t_n], Y : [s_1, \ldots, s_m], \ldots]$, representing the collection of constraints in solved form $X \neq t_1, \ldots, X \neq t_n, Y \neq s_1, \ldots, Y \neq s_m, \ldots$. This constraint store must be consistently kept and taken into account throughout the computation. This is why many Prolog procedures below have two arguments Cin, $Cout$ containing the constraint store as it is before and after the procedure acts.
These are the top-level clauses for goal solving:

```
solve(G,OutStore) :- solve(G,[],OutStore).
solve((G,Gs),Cin,Cout) :- !, solve(G,Cin,Cout1), solve(Gs,Cout1,Cout).
solve(E1==E2,Cin,Cout) :- equal(E1,E2,Cin,Cout).
solve(E1/=E2,Cin,Cout) :- notEqual(E1,E2,Cin,Cout).
```

It is clear that the predicate *solve* can be eliminated by partial evaluation, resulting in direct calls to *equal* and *notEqual*. This is done in the system.

Clauses for Solving Equalities (Equal_{DC})

```
equal(E1,E2,,Cin,Cout) :-
   hnf(E1,H1,Cin,Cout1),
   hnf(E2,H2,Cout1,Cout2),
   equalHnf(H1,H2,Cout2,Cout).
equalHnf(X,H,Cin,Cout) :-
   var(X),!, binding(X,H,Cin,Cout).
equalHnf(H,X,Cin,Cout) :-
   var(X),!, binding(X,H,Cin,Cout).
equalHnf(c(X1,...,Xn),c(Y1,...,Yn),Cin,Cout) :-     % a clause for each c ∈ DC^n
   equal(X1,Y1,Cin,Cout1),
   ...,
   equal(Xn,Yn,Coutn-1,Coutn).
binding(X,Y,Cin,Cin) :-
   var(Y),X==Y,!.
binding(X,Y,Cin,Cout) :-
   var(Y),!,
   extractCtr(Cin,X,Cout1,CX),
```

```
    not member(Y,CX),
    extractCtr(Cout1,Y,Cout2,CY),
    append(CX,CY,CXY),
    X=Y,
    Cout = [X:CXY—Cout2].
binding(X,H,Cin,Cout) :-
    occursNot(X,H,SkelH,Res),
    extractCtr(Cin,X,Cout1,CX),
    X=SkelH,
    propagate(X,CX,Cout1,Cout2),
    continueBind(Res,Cout2,Cout).
occursNot(X,Y,SkY,Res) :- occursNot(X,Y,SkY,[],Res).
occursNot(X,Y,Y,Res,Res) :-
    var(Y),!,X ≐ Y.
occursNot(X,c(Y1,...,Yn),c(Sk1,...,Skn),Res0,Resn) :-
    occursNot(X,Y1,Sk1,Res0,Res1),
    ...,
    occursNot(X,Yn,Skn,Resn−1,Resn).
occursNot(X,susp(E,R,S),U,Res,[U==susp(E,R,S)—Res]) :- var(S),!.
occursNot(X,susp(E,R,S),Skel,Res,Res1) :-
    occursNot(X,R,Skel,Res,Res1).
propagate(X,[],Cin,Cin).
propagate(X,[T—Terms],Cin,Cout) :-
    notEqual(X,T,Cin,Cout1),
    propagate(X,Terms,Cout1,Cout).
continueBind([],Cin,Cin).
continueBind([X==Y—Res],Cin,Cout) :-
    equal(X,Y,Cin,Cout1),
    continueBind(Res,Cout1,Cout).
extractCtr([],X,[],[]).
extractCtr([Y:Terms—Ctrs],X,Ctrs,Terms) :- X==Y,!.
extractCtr([Y:CY—Ctrs],X,[Y:CY—Ctrs1],CX) :-
    extractCtr(Ctrs,X,Ctrs1,CX).
```

Clauses for Solving Disequalities (NotEqual$_{DC}$)

```
notEqual(X,Y,Cin,Cout) :-
    hnf(X,HX,Cin,Cout1),
    hnf(Y,HY,Cout1,Cout2),
    notEqualHnf(HX,HY,Cout2,Cout).
notEqualHnf(X,Y,Cin,Cout) :-
    var(X),!,notEqualVar(X,Y,Cin,Cout).
notEqualHnf(Y,X,Cin,Cout) :-
    var(X),!,notEqualVar(X,Y,Cin,Cout).
```

```
notEqualHnf(c(X1,...,Xn),d(Y1,...,Ym),Cin,Cin). % if c ≠ d
notEqualHnf(c(X1,...,Xn),c(Y1,...,Yn),Cin,Cout) :-
    notEqual(X1,Y1,Cin,Cout)
        ;
        ... % don't know nondeterminism
        ;
    notEqual(Xn,Yn,Cin,Cout).
notEqualVar(X,Y,Cin,Cout) :-
    var(Y),!,X =Y,
    addCtr(X,Y,Cin,Cout1),
    addCtr(Y,X,Cout1,Cout).
notEqualVar(X,Y,Cin,Cout) :-
    occursNot(X,Y,ShY,Res),
    !,
    contNotEqual(X,Y,ShY,Res,Cin,Cout).
notEqualVar(X,Y,Cin,Cin). % The occur-check failed, then X ≠ Y is valid
contNotEqual(X,_,ShY,[],Cin,Cout) :- % No residual; Y was a cterm
    !,addCtr(X,ShY,Cin,Cout).
contNotEqual(X,c(X1,...,Xn),Cin,Cout) :-
    hnf(X,d(Y1,...,Ym),Cin,Cout)
        ; % don't know nondeterminism
    hnf(X,c(Y1,...,Yn),Cin,Cout1),
    notEqual(X,c(X1,...,Xn),Cout1,Cout).
addCtr(X,Term,Cin,Cout) :-
    extractCtr(Cin,X,Cout1,CX),
    Cout = [X:[Term—CX]—Cout1]
```

G.3.2 Clauses for Computing Head Normal Forms (Hnf_Σ)

```
hnf(X,H,Cin,Cin) :-
    var(X),var(H),!,X=H.
hnf(X,H,Cin,Cout) :-
    var(X),!,
    extractCtr(Cin,X,Cout1,CX),
    X=H,
    propagate(X,CX,Cout1,Cout).
hnf(c(X1,...,Xn),H,Cin,Cin) :-
    !, H=c(X1,...,Xn).
hnf(susp(f(X1,...,Xn),R,S),H,Cin,Cout) :-
    var(S),!,
    f(X1,...,Xn,H,Cin,Cout),
    R=H,
    S=hnf.
hnf(susp(E,R,S),H,Cin,Cout) :-
```

hnf(R,H,Cin,Cout).

G.3.3 Clauses for Function Definitions (Prolog_{FS})

Construction of Definitional Trees

Definition G.1 (Demanded positions)

- *A position u is* demanded by a rule $f(t_1,\ldots,t_n) = e \Leftarrow C$ *if $f(t_1,\ldots,t_n)$ has a constructor symbol at position u.*
- *A position u is* demanded by a set of rules S *if it is demanded by at least one rule of S.*
- *A position u is* uniformly demanded by a set of rules S *if it is demanded by all the rules of S.*

Definition G.2 (Call patterns)

- *A* call pattern *for f is $f(s_1,\ldots,s_n)$, where $(s_1,\ldots,s_n)$ is a linear tuple of patterns.*
- *A call pattern $f(s_1,\ldots,s_n)$ is* compatible *with a set of rules S if $f(s_1,\ldots,s_n)$ matches all the left-hand sides of the rules in S, that is, $f(s_1,\ldots,s_n) \leq f(t_1,\ldots,t_n)$ for any $f(t_1,\ldots,t_n)$ which is a left-hand side of a rule of S.*

In the following definition we write $vpos(e)$ for the set of positions in e occupied by variables.

Definition G.3 (Definitional trees) *Let $f \in FS^n$ be a function symbol defined by a set of rules R_f in a program P. Then:*

- $dt(f) = dt(f(X_1,\ldots,X_n), R_f)$
- *Let S be a nonempty subset of R_f, and pat a call pattern for f compatible with S. The* definitional tree *for pat corresponding to S, written $dt(pat,S)$, is defined recursively according to the following cases (exactly must hold, and the second case involves a non-deterministic choice):*

  1. *Some position in $vpos(pat)$ is uniformly demanded by S.*
     Let u be one of them (non-determinism)
     X the variable in pat at u
     $c_1,\ldots,c_m$ the constructor symbols at u in the rules of S
     $S_i = \{R \in S \mid R$ has c_i at $u\}$ $(i \in \{1,\ldots,m\})$
     $pat_i = pat[u \leftarrow c_i(U_1,\ldots,U_k)]$ $(c_i \in DC^k, U_1,\ldots,U_k$ news$)$
     Then $dt(pat,S) = pat \rightarrow$ **case** X **of**
     $\langle c_1 : dt(pat_1,S_1)$
     $\ldots$
     $c_m : dt(pat_m,S_m)\rangle$

2. *Some position in vpos(pat) is demanded by S, but no one is uniformly demanded.*
   Let $u_1, \ldots, u_m$ *the demanded positions, in any order (non-determinism)*
   $S_{u_1} = \{R \in S \mid R \text{ demands } u_1\}$; $Q_1 = S - S_{u_1}$
   $S_{u_2} = \{R \in Q_1 \mid R \text{ demands } u_2\}$; $Q_2 = Q_1 - S_{u_2}$
   $\ldots$
   $S_{u_m} = \{R \in Q_{m-1} \mid R \text{ demands } u_m\}$
   $S_0 = Q_{m-1} - S_{u_m}$ *(if* $S_0 = \emptyset$*, it is ignored)*
   Then $dt(pat, S) = pat \rightarrow$ **or** $\langle dt(pat, S_0)$
   $\mid dt(pat, S_{u_1})$
   $\ldots$
   $\mid dt(pat, S_{u_m})\rangle$
3. *No position in vpos(pat) is demanded by S (therefore, since pat is compatible with S, the left-handsides of S are variants of pat).*
   Let $R_1 \equiv pat = e_1 \Leftarrow C_1$
   $\ldots$
   $R_n \equiv pat = e_n \Leftarrow C_n$
   be variants of the rules of S
   Then $dt(pat, S) = pat \rightarrow$ **try** $\langle e_1 \Leftarrow C_1$
   $\mid e_2 \Leftarrow C_2$
   $\ldots$
   $\mid e_n \Leftarrow C_n\rangle$

Prolog Code associated to Definitional Trees

For each function symbol $f \in FS$, a set of Prolog clauses is generated, which is responsible of computing head normal forms for calls to f of the form $f(e_1, \ldots, e_n)$. At the top-level, the Prolog code for $f \in FS$ defines a Prolog predicate f [1] with three more arguments than the original function f (one for the computed head normal form H, two more for the initial and final states (*Cin*, *Cout*) of the disequality constraints store). Thus the top-level clause for f has the shape

$$f(E_1, \ldots, E_n, H, Cin, Cout) :- \ldots$$

The complete code for f, called here $prolog(f)$, is extracted from the definitional tree of f, and is defined as

$$prolog(f) = prolog(f, dt(f))$$

where the auxiliary function $prolog/2$ takes a definitional tree and a function symbol (possibly different to the function of the definitional tree), returning as value a set of Prolog clauses, as defined below.

- $prolog(f(\bar{s})) \rightarrow$ **case** X **of** $\langle c_1 : dt_1 \ldots c_m : dt_m\rangle$, $g)$

[1] In the actual implementation, the predicate is named $\$f$ in order to avoid conflicts between 'user-defined' predicates coming from the Prolog translation of a $\mathcal{TOY}$ program and built-in predicates or predicates belonging to the Prolog translator itself.

$=$ let $\sigma = X/HX$ in
$\{g(\overline{s}, H, Cin, Cout) :- hnf(X, HX, Cin, Cout1), g'(\overline{s}\sigma, H, Cout1, Cout)\}$
$\cup\ prolog(dt_1, g')$
$\ldots$
$\cup\ prolog(dt_m, g')$

where g' is a new function symbol.

- $prolog(f(\overline{s})) \rightarrow \textbf{or}\ \langle dt_1 \mid \ldots \mid dt_m \rangle,\ g)$

  $=$ $\{g(\overline{s}, H, Cin, Cout) :- g_1(\overline{s}, H, Cin, Cout),$
  $\ldots,$
  $g(\overline{s}, H, Cin, Cout) :- g_m(\overline{s}, H, Cin, Cout)\}$
  $\cup\ prolog(dt_1, g_1)$
  $\ldots$
  $\cup\ prolog(dt_m, g_m)$

  where $g_1, \ldots, g_m$ are new function symbols.

- $prolog(f(\overline{s})) \rightarrow \textbf{try}\ \langle e_1 \Leftarrow C_1 \mid \ldots \mid e_m \Leftarrow C_m \rangle,\ g)$

  $=$ $\{g(\overline{s}, H, Cin, Cout) :- solve(C_1, Cin, Cout1), hnf(e_1, H, Cout1, Cout),$
  $\ldots,$
  $\{g(\overline{s}, H, Cin, Cout) :- solve(C_m, Cin, Cout1), hnf(e_m, H, Cout1, Cout)\},$

Appendix H

Release Notes History

This chapter lists in reverse chronological order the release notes history from the first maintained versioning system (Toy system version 2.0. February, 2002).

H.1 Version 2.3. Launched on December, 2006

Enhancements

- communication function `bridge` has beed added: `(#==) :: int -> real -> bool`.
- The libraries CLP(R) and CLP(FD) can be activated together.
- Implementation of the propagation for the cooperation between the real and finite domain solvers (see 4)
- New commands:

```
/prop          Enables propagation
/noprop        Disables propagation
```

- More examples have been added.

```
examples/.             Uncataloged examples
```

- New chapter in the manual:
  - **??**hapter 4. Cooperation of Solvers.

Deprecations
Changes
Fixed Bugs

H.2 Version 2.2.3. Launched on July, 2006

Enhancements

- Optimization functions have beed added to the library CFLPR: `minimize, maximize :: real -> real`, for linear optimization `bb_minimize, bb_maximize :: real -> [real] -> real`, for linear optimization with integral constraints
- When the library CLP(R) is activated, the function `log/2` is now handled by the real solver
- The file `miscfd.toy` is added to contain useful finite domain functions
- More run-time errors due to demandness are handled by means of exceptions, instead of failures (e.g., `belongs`, `sum`, `scalar_product`, ...)
- The implementation of several finite domain functions has been modified, also adding some missing usage mode information
- New commands:

```
/prolog(Goal) Executes the Prolog goal Goal
/status       Informs about the loaded libraries
/version      Displays the Toy software version
```

- More examples have been added. Moreover, the directory `examples` becomes more structured:

```
examples/.            Uncataloged examples
examples/apifd/.      Program examples for the API FD
examples/cflpfp/.     Program examples for CFLP(FD)
examples/cflpr/.      Program examples for CFLP(R)
examples/debug/.      Debugging examples
examples/failure/.    Programming with failure examples
examples/flp/.        Functional logic programming examples
examples/fp/.         Functional programming examples
examples/h_constr/.   Program examples for Herbrand constraints
                      (equality and disequality)
examples/io_file/.    Program examples for the library IO File
examples/io_graphic/. Program examples for the library IO Graphic
examples/lp/.         Logic programming examples
examples/tads/.       Program examples for abstract data types
```

- New sections in the manual:

– Release Notes History
– Known Bugs

Deprecations

- `substract` (in `misc.toy`). See Section Changes below
- `wakeoptions`. See Section Changes below
- `options`. See Section Changes below
- `newOptions`. See Section Changes below

Changes

- `fd_statistics` and `fd_statistics'` have interchanged their role in order to adhere to the convention that the prime indicates that options can be used. The type of `fd_statistics'` has changed to `statistics -> int`
- `substract` (in `misc.toy`) becomes `subtract`
- `wakeoptions` becomes `wakeOptions`
- `options` becomes `allDiffOptions`
- `newOptions` becomes `serialOptions`

Fixed Bugs

- The command `type` did not work on expressions. Fixed
- User-defined commands as described in section 1.5.5 of User Manual for version 2.2.2 did not work. Fixed
- `sum` and `scalar_product` were not correctly implemented, giving rise to type error exceptions. Fixed
- `fd_statistics` delivered duplicate solutions. Fixed
- `/nocflpfd` did not completely unload the library CFLPFD. Fixed
- The libraries IO File and IO Graphic could not be unloaded. Fixed
- When first loading the libraries IO File and IO Graphic, and depending on the previous state of the system, their functions were not available at command prompt until a program were compiled. Fixed
- The data type ioMode was documented as:

```
data ioMode = read | write | append
```

but the actual declaration is:

```
data ioMode = readMode | writeMode | appendMode
```

H.3 Version 2.2.2. Launched on March, 2006

Enhancements

- Dynamic cut optimization. It can be activated using the `/cut` command before compiling a program.
- The debugger now allows programs/goals with constraints over real numbers. See the example examples/debug/ladder.toy
- New finite domain functions and constraints:
  - Membership constraints:

```
subset :: int -> int -> bool          % Domain subset
setcomplement :: int -> int -> bool   % Domain complement
inset :: int -> int -> bool           % Domain member
intersect :: int -> int -> int        % Domain intersection
belongs :: int -> [int] -> bool       % Domain definition
```

  - Propositional constraints:

```
(#\/) :: bool -> bool -> bool   % Disjunction
(#<=>) :: bool -> bool -> bool  % Equivalence
(#=>) :: bool -> bool -> bool   % Implication
```

  - Arithmetic constraint operators:

```
(#&) :: int -> int -> int       % Integer remainder
```

- More constraints are now reifiable (e.g., `sum`, `scalar_product`, ...)
- Some constraints become more declarative as they do not demand arguments with known finite types. This is the case, for instance, of labeling. Now, it produces all the admissible options for labeling. Consider the following goal:

```
Toy(FD)> domain [X] 0 1, labeling L [X]
      {  X -> 0,
         L -> []   }
      Elapsed time: 110 ms.
```

```
more solutions (y/n/d) [y]?
      {  X -> 1,
         L -> []   }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      {  X -> 0,
         L ->  [ bisect ]   }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      {  X -> 1,
         L ->  [ bisect ]   }
      Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
      {  X -> 0,
         L ->  [ bisect, ff ]   }
      Elapsed time: 0 ms.
```

...

Up to all the correct answers

- More examples added to the distribution
- User Manual updated (draft published)
- New formatted output
- Nonlinear real functions revisited. Arguments of nonlinear real functions do not need to be instantiated when the library CLP(R) is loaded: `ln`, `exp`, `cot`, `asin`, `acos`, `atan`, `acot`, `sinh`, `cosh`, `tanh`, `coth`, `asinh`, `acosh`, `atanh`, `acoth`

Deprecations

- `all_distinct` is deprecated since its behaviour is the same as `all_different`
- `all_distinct'` is deprecated since its behaviour is the same as `all_different'`

Changes

- `[fdset]` becomes simply `fdset`. See type declaration in User Manual

Fixed Bugs

- The goal:

  ```
  labeling X [Y]
  ```

  ran into an infinite loop. Fixed

- The goal:

  ```
  domain [(id X)] 0 1
  ```

  with the program:

  ```
  id :: A -> A id A = A
  ```

  shows the following error:

  ```
  SYSTEM ERROR:
  type_error(domain([$$susp($id,[_116],_121,_122)],0,1),1,integer,
             $$susp($id,[_116],_121,_122))
  ```

  The same situation held for labeling. Fixed

- Reification. Missing answer:

  ```
  Toy(FD)> domain [X] 0 1, B #<=> (X#=0)
        {  X -> 0,
           B -> true   }
        Elapsed time: 0 ms.
  more solutions (y/n/d) [y]?
        no.
        Elapsed time: 0 ms.
  ```

  With the current version:

  ```
  Toy(FD)> domain [X] 0 1, B #<=> (X#=0)
        {  X -> 0,
           B -> true   }
        Elapsed time: 0 ms.
  more solutions (y/n/d) [y]?
        {  X -> 1,
           B -> false   }
        Elapsed time: 0 ms.
  more solutions (y/n/d) [y]?
        no.
        Elapsed time: 0 ms.
  ```

- Equality and inequality constraints missed during processing of scheduling constraints. Fixed

- Duplicate solutions with scheduling constraints. Fixed

- Non ground, non FD variables were output as `inf..sup`. Fixed

H.4 Version 2.2.1. Launched on November, 2005

Enhancements

- The set of examples has been extended with a new folder `examples/debug` containing examples of buggy programs that can be used for testing the declarative debugger
- The computation tree generated by the declarative debugger is now stored in the directory `javaTree`. This avoids the creation of auxiliary files in the directory containing the program to be debugged

Fixed Bugs

- A program as

  ```
  data d = c int int
  f (c 3) = 4
  ```

  yielded an error during compilation (error code 24). Fixed

H.5 Version 2.2.0. Launched on August, 2005

Enhancements

- The system variable `TOYDIR` is no longer necessary
- The answers for a goal are now displayed as two sets, the first one representing a substitution and the second one a conjunction of constraints. For instance, a goal as `X==5, X/=Y` displays the answer:

  ```
  { X -> 5}
  { Y /= 5}
  ```

  meaning that if `X` is substituted by 5 and `Y` is any value different from 5 the goal holds. However, not all finite domain constraints are shown up to now
- Totality constraints allowed. They can be activated by typing `/tot` and disabled with `/notot`
- The declarative debugger handles programs including equality and disequality constraints
- Four new (non-declarative) primitives, `collect`, `collectN`, `once`, and `fails` have been included

- `collect e` collects in a list all the values obtained when evaluating the expression e. For instance, a goal as `collect (3 // 4 // 5) == R` has one answer { `R -> [3,4,5]` }. The elements of the list correspond to all the values `V` computed for the goal `e==V`. If `e==V` has no answer then `collect e` returns `[]`
- `collectN n e` collects in a list the `n` first values obtained when evaluating `e`. For instance a goal as `collectN 2 (3 // 4 // 5) == R` has one answer { `R -> [3,4]` }. If `e` cannot be evaluated to `n` values, the primitive fails
- `once e` is equivalent to `head (collectN 1 e)`
- `fails e` returns `true` if the goal `e==V` fails, or false if the goal `e==V` has some answer. For instance, the goal `fails (head [])==R` has the single answer `{R -> true}`. In contrast, `fails (head [X|Xs]) == R` has the single answer `{R -> false}`

- The system can display statistics about the number of head normal forms, the system memory or the number of the choice points required by the underlying Prolog program during a certain computation. See `/statistics` in the help command (displayed by typing `/help`) to check all the possibilities

Fixed Bugs

- The finite domain library was not working properly if no program was compiled after typing `/cflpfd`. Fixed
- Fixed an error which involved the combination of disequality and equality with real numbers. For example, after loading the library CLP(R) (by typing `/cflpr`), a goal as `X/=1, X==2` was throwing an exception
- Fixed an error that occurred when combining disequality and equality constraints of composed terms. For example, `X/=(1,1), X==(2,2)` was throwing an exception

H.6 Version 2.1.0. Launched on May, 2005

Enhancements

- Finite Domain Constraints
- Enhanced Declarative Debugging
- Multiplatform Support (including Windows, UNIX, and Linux)
- Executables for Windows 98 and later, and SunOS54

H.7 Version 2.0. Launched on February, 2002

Undocummented

Appendix I

Known Bugs

- The concurrent execution of several Toy interpreter sessions is not guaranteed to work properly when loading and downloading libraries.
- 0-arity tuples are not allowed.
- You can compile and load a program with FD constraints without loading the library CFLP(FD), but the correct implementation of primitive FD constraints and functions is not available. So, before "running" a CFLP(FD) program, load this library by means of the command `/cflpfd`.
- If CFLP(FD) is loaded and a program is compiled and run without the directive `include "cflpfd.toy"`, then the definitions of the FD constraint and functions are lost.
- When Ctrl-C is pressed, the stand-alone $\mathcal{TOY}$ application exits.