TOY(FD): Version 0.8
User Manual

Antonio J. Fernández
Lenguajes y Ciencias de la Computación
Campus Teatinos, University of Málaga,
Málaga 29071, Spain
e-mail: afdez@lcc.uma.es
http://www.lcc.uma.es/~afdez/
Fax-number: +34-952-13 13 97

Teresa Hortalá-González and Fernando Sáenz-Pérez
Sistemas Informáticos y Programación,
Complutense University of Madrid
e-mails: {teresa,fernando}@sip.ucm.es

27th October 2003

1This author has been partially supported by the projects TIC2001-2705-C03-02, and TIC2002-04498-C05-02 funded by both the Spanish Ministry of Science and Technology and FEDER.
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFLP</td>
<td>Constraint Functional Logic Programming</td>
</tr>
<tr>
<td>CFLP(FD)</td>
<td>Constraint Functional Logic Programming over Finite Domains</td>
</tr>
<tr>
<td>CLP</td>
<td>Constraint Logic Programming</td>
</tr>
<tr>
<td>CLP(FD)</td>
<td>Constraint Logic Programming over Finite Domains</td>
</tr>
<tr>
<td>CP</td>
<td>Constraint Programming</td>
</tr>
<tr>
<td>CSP</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>FD</td>
<td>Finite Domain</td>
</tr>
<tr>
<td>FLP</td>
<td>Functional Logic Programming</td>
</tr>
<tr>
<td>FP</td>
<td>Functional Programming</td>
</tr>
<tr>
<td>LP</td>
<td>Logic Programming</td>
</tr>
</tbody>
</table>
Remark

This document is a user manual exclusively developed for TOY(FD) and, as a consequence, does not treat directly about TOY, although it provides an (intuitive) introduction to TOY. In general, we assume the reader is familiarized with the TOY language. If this is not the case, the reader is encouraged to read Caballero et al., 1997; López-Fraguas and Sánchez-Hernández, 1999.

Also, this document does not deal with the underlying theoretical framework of TOY(FD) that will be described further in a TOY(FD) reference manual (currently in preparation). The interested reader in this issue (e.g., type discipline, operational semantics, denotational semantics) is referred to Fernández et al., 2003a; Fernández et al., 2002a; Fernández et al., 2003b.
# Contents

## Acronyms

| Acronyms | iii |

## Remark

| Remark | v |

## The TOY(FD) System

1. The TOY(FD) System

1.1 What is TOY(FD)?

1.1.1 In Brief Words, TOY(FD) ... 1

1.1.2 More.... 1

1.2 The Current Implementation

1.2.1 Where to Find TOY(FD)? 2

1.2.2 Requirements 2

1.2.3 Efficiency 2

1.2.4 Referencing this Software 2

1.2.5 Distribution 3

1.2.6 Runing TOY(FD) 5

1.3 Interpreter Commands

1.3.1 Calls to the Operating System (OS) 7

1.3.2 Compiling and Running TOY(FD) Programs 7

1.3.3 Information about Functions 8

1.3.4 Saving the session 8

1.3.5 Activating the Output for FD Constraint Solving 8

1.4 FD Constraints

1.4.1 Predefined Data Types 9

1.4.2 Membership Constraint 10

1.4.3 Relational Constraints 11

1.4.4 Arithmetic Constraints 12

1.4.5 Combinatorial Constraints 13

1.4.6 Enumeration Constraints 17

1.4.7 Statistics Constraints 19

1.5 Some more constraints..... 21
2 TOY(FD) Programming Examples 23
  2.1 Important Note about Programming with TOY(FD) 23
  2.2 Introductory TOY(FD) Examples 24
    2.2.1 Send+More = Money 24
    2.2.2 N-queens 24
    2.2.3 A Cryptarithmetic Problem 26
    2.2.4 Magic Sequences 27
  2.3 More Complex Examples 29
    2.3.1 A Scheduling Problem 29
    2.3.2 A Hardware Design Problem 32
    2.3.3 A Colour Problem 44
  2.4 Colour Problem 44
  2.5 Lazy Constraint Programs 46
    2.5.1 Hamming Codes 46
    2.5.2 Lazy Magic Sequences 48

Bibliography 51

A TOY(FD) Grammar 53

B Declaration of Primitives (file basic.toy) 59

C Common Functions (file misc.toy) 61
List of Figures

2.1 Precedence Graph. ...................................................... 29
2.2 Basic Modules. ......................................................... 34
2.3 Two-Input Multiplexer Circuit. ..................................... 34
2.4 puzzle ................................................................. 45
2.5 Hamming Codes ......................................................... 47
List of Tables

1.1 Predefined Datatypes for FD Constraints .......................... 9
1.2 The Predefined Set of FD Constraints .............................. 10
Chapter 1

The TOY(FD) System

1.1 What is TOY(FD)?

1.1.1 In Brief Words, TOY(FD) ...

- ... is an implementation of a functional logic language with finite domain (FD) constraints. This language adds the power of constraint programming over finite domains to the characteristics of functional logic programming.

- ... increases the expressiveness and power of constraint logic programming over finite domains (CLP(FD)) by combining functional and relational notation, curried expressions, higher order functions, patterns, partial applications, non-determinism, constraint composition, lazy evaluation, logical variables, types, domain variables, constraints and constraint propagators.

- ... combines both the efficiency and expressiveness of CLP(FD) with new features not existing in CLP(FD) that contribute to increase the expressiveness of constraint declarative languages.

- ... is, to our knowledge, is the first constraint language that provides lazy evaluation.

1.1.2 More....

TOY(FD) is not only a declarative alternative to CLP(FD) but also extends its capabilities with new characteristics unusual or not existing in CLP(FD) such as functional and curried notation, types, curried and higher order functions (e.g., higher order constraints), constraint composition, higher order patterns, lazy evaluation and polymorphism, among others. As a consequence, TOY(FD) provides better tools, when compared to a CLP(FD) language, for a productive declarative programming as, implicitly, it enables more expressivity, due to the combination of functional, relational and curried notation as well as type checking. Moreover, lazy evaluation allows the use of structures impossible to manage in a CLP(FD) language, such as infinite lists.
CHAPTER 1. The TOY(FD) System

In TOY(FD), finite domain (FD) constraints are integrated as functions to make them first-class citizens, which means that they can be used in any place where a data can (e.g., as arguments of functions). This provides a powerful mechanism to define higher order constraints.

One important characteristic of TOY(FD) is that it enables to solve all the CLP(FD) applications as well as another problems closer to the functional setting.

In summary, TOY(FD) has important benefits as it takes functions, higher order patterns, partial applications, non-determinism, lazy evaluation, logical variables, and types from the FLP paradigm and domain variables, constraints, and propagators from the FD constraint community. This leads to a more declarative way of expressing problems which cannot be reached from FP or LP alone.

1.2 The Current Implementation

1.2.1 Where to Find TOY(FD)?

TOY(FD) version 0.8 is freely available as a compressed file toyfd.zip at

http://www.lcc.uma.es/~afdez/cflpfd

1.2.2 Requirements

TOY(FD) is implemented on top of SICStus Prolog and requires SICStus 3#8.2 (or probably higher although this was not tested yet). The current version has been tested under Windows 2000 and is also valid for Unix systems.

Note: The current version has been successfully tested on a PC with Windows 2000 and using SICStus 3.8.2.

1.2.3 Efficiency

In Fernández et al., 2003a we showed that TOY(FD) 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 TOY does not increase significantly the computation time.

Moreover, in the same paper we showed that TOY(FD) is about two and five times faster (and even much more in scalable problems) than another CFLP(FD) implementation M. Hanus (editor), 2002 to come which is said to be efficient in its Web page (http://redstar.cs.pdx.edu/~pakcs/).

1.2.4 Referencing this Software

When referring to this implementation of TOY(FD), please use the reference Fernández et al., 2003a.
1.2.5 Distribution

The current distribution is freely available as a compressed file `toyfd.zip` that contains the files shown below.

**System files.**

- `toyfd.pl` The source code of the system. It is the main file containing both the original TOY implementation and its extension to support FD constraints.
- `toy.ini` Initialization options.
- `token.pl` Module with predicate definitions for the lexicographical analysis.
- `compil.pl` Module with predicates for the generation of intermediate code.
- `transdcg.pl` Module with translation from DCG’s to Prolog clauses.
- `transdebug.pl` Module in charge of transforming a compiled predicate into another ready for debugging.
- `errortoy.pl` Module for error handling during compilation.
- `gramma_dcg.pl` Module with DCG definitions.
- `gramma_toy.pl`, `cabecera_gramma_toy.pl` Modules with syntax definition of the language.
- `transob.pl` Module to interpret commands and/or goals.
- `dds.pl` Module that generates the definitional trees (dds trees) associated to a function.
- `codfun.pl` This module generates the code of a function using the definitional tree.
- `transfun.pl` This module returns the code associated to a function from its name and generates the definitional tree.
- `process.pl` Main module for generating code.
- `toycomm.pl` Module that contains all the common Prolog predicates to each TOY(FD) program.
- `connected.pl` This module searches the connected components of a graph of dependencies, concretely the functional dependencies of the declared functions by the user and the standard dependencies.
- `inferrer.pl` This module implements the type inferrer.
• primFunct.pl, primFunctGra.pl, primFunctIo.pl, primitivCod.pl, primitivCodClpr.pl, primitivCodGra.pl, primitivCodIo.pl
  Primitives modules of the system.

• pVal.pl Module managing the debugging predicate pVal.

• dyn.pl This module contains the dynamic predicates.

• evalexp.pl This module is necessary for the evaluation of expressions. It shows the evaluation in the screen.

• goals.pl This module contains the whole process to show the answer (i.e., solutions to goals).

• initToy.pl Module to recognize goals, commands and expressions to evaluate at prompt-level. It includes the syntax analysis.

• navigating.pl, sailing .pl These modules contain the algorithm to navigate thorough the debugging tree.

• newout.pl This module contains the exportation clause of the predicates needed to import.

• osystem.pl This module contains the predicates with calls to the system.

• tools.pl This module contains general predicates.

• writing.pl Module managing the writing of atoms.

• cflpfd.pl, cflpfdfile.pl These modules support the extension to FD constraints.

• basic.pl, basiccopia.pl, basicGra.pl, misc.pl They contains the predefined functions and types of the system.

• basicIO.pl, ioantonio.pl Basic Input/Output.

• cflpr.pl Module to manage real constraints.

• nada.pl, nada.debug.pl Dummy modules implementation-specific.

• tcltk.pl Library for the integration of tcltk in TOY(FD).

A set of TOY(FD) files
There are a number of TOY(FD) files that contains predefined definitions for types and functions.

A set of TOY programs
The distribution also provides a directory (i.e., Examples) that contains a number of examples of TOY(FD) programs that make use of FD constraints. In the following, we enumerate these examples:
1.2. The Current Implementation

- queens.toy The n-queens problem. Example in the FD.
- scheduling.toy A task scheduling problem.
- eq10.toy and eq20.toy Problem of solving of 10 and 20 linear equations respectively.
- smm.toy A solution to the classical SEND+MORE=MONEY problem.
- alpha.toy A cryptoarithmetic problem.
- circuit1.toy and circuitFD.pl Generating circuits with constraining physical factors.
- magicseq.toy and lazymagicseq.toy Solutions to the well-known problem of the magic series. The first one offers a solution based on eager evaluation whereas the second one offers a solution based on lazy evaluation.
- hamming.toy The problem consists of obtaining the ordered sequence of Hamming codes.

Documentation. The distribution also provides a directory (i.e., docs) that contains this user manual and a reference to the on-line manual.

1.2.6 Running TOY(FD)

To run TOY(FD), follow the following process:

- set the environment variable TOYDIR to your working directory;

  Essential note: TOY(FD) can be installed on any directory of any drive as long as the full path of TOYDIR does not contain spaces !!!!

To set environment variables you can follow the following instructions:

(a) In Windows 95/98/ME (TOY(FD) was not tested on these systems!) append next line to the end of your c:\autoexec.bat using notepad, and then reboot your computer (assuming that your working directory is C:\SICSTUS382\bin\toyfd):

```
set TOYDIR=C:\SICSTUS382\bin\toyfd
```

(b) In Windows 2000 (and also NT and XP, although TOY(FD) was not tested in these systems), follow the following steps: Open ‘Control Panel’, click the System icon and the window pops up. Go to the Advanced panel. Click the Environment Variables button. There are two separated windows showing two sets of environment variables. Select the “new” button for the upper window to create a new environment variable. Select the ”Edit” button for the upper window to edit an existing environment variable. Probably you need to create the “TOYDIR” variable, and edit it. The value of “TOYDIR” should be the directory name, containing no spaces, where you want to put TOY(FD).
• Uncompress the file `toyfd.zip` in your working directory (e.g., `C:\SICSTUS382\bin\toyfd`; observe that this path does not contain spaces!!);

• initiate a SICStus session and choose as working directory that identified by `TOYDIR`;

• load TOY(FD) from the command line of SICStus by consulting the file `toyfd`, i.e., from the SICStus prompt

```
| ?- [toyfd].
```

At this moment, SICStus loads all the libraries and archives necessary and will show the prompt of TOY(FD) (i.e., `TOY(FD)>`) and a message for helping, as follows:

```
TYPE "/h." FOR HELP.
TOY(FD)>
```

You are now inside TOY(FD)!!! i.e., you are now in the interpreter command level of TOY(FD).

### 1.3 Interpreter Commands

At the interpreter command level you can type some predefined commands and directives. In the following, we describe some of the most used. For a complete list see Caballero et al., 1997.

All of these commands begins with `/` and ends with `.`. Blank spaces are not allowed and they are interpreted as Prolog terms i.e., the arguments are enclosed between normal brackets `(')`, and separated by commas `,`. Moreover, if the arguments contains blank spaces or symbols like `/` (e.g., to specify a path), then they must be enclosed between simple inverted commas `('`).

For example, the following commands are recognized by TOY(FD):

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

However, the following are not recognized:

```
/compile (my_program).
/cd(home/local/bin).
/cd(../.).
```
1.3.1 Calls to the Operating System (OS)

Some commands calls directly to the operating system, as for example the following:

- `/help. or /h."
  Show a brief help menu;
- `/cd(< dir >)` changes the TOY(FD) working directory to the directory `< dir >`;
- `/system(< comm >)` sends the command `< comm >` to the OS (i.e., to the corresponding shell). This is the most direct way that TOY(FD) provides to communicate with the OS. For example, the command `/cd(< dir >)` can be obtained as `/system('cd < dir >')`. Another example is:
  `/system('cp my_program.toy /home/users/axterix/examples/.').
- `/q, /quit, /e or /exit` terminate the TOY(FD) session.

1.3.2 Compiling and Running TOY(FD) Programs

By default, the extension of TOY(FD) programs is ‘.toy’, that is to say, the same extension than classical TOY programs (remember that TOY is part of TOY(FD)). Any reference to a source program without explicit extension will be completed with the extension ‘.toy’. The compilation of a TOY(FD) program (e.g., `file.toy`) generates a SICStus Prolog program (e.g., `file.pl`).

Before loading a TOY(FD) program, the file containing this TOY(FD) program has to be compiled. Then after a correct compilation, the SICStus file generated from this compilation can be loaded and the user is ready to execute goals. Of course, this process is transparent to the user and is simplified via the following three commands:

- `/compile(< file >)` compiles the archive `< file >` (or `< file >.toy` if no extension is provided) and generates the archive `< file >.pl`. This archive is not compiled by SICStus and, as a consequence, no goal can be solved at the end of this process.
- `/load(< file >)` loads the program `< file >.pl`, previously compiled by TOY(FD). In fact, this command asks SICStus to compile the Prolog program generated by TOY(FD) after executing the previous command.

When, via a new program, a definition for a function previously used in the current TOY(FD) session is redefined, TOY(FD) (via Sicstus) will show the following message:

```
The procedure Name/Arity is being redefined.
Old file: ...
New file: ...
Do you really want to redefine it? (y, n, p, or ?)
```
CHAPTER 1. The TOY(FD) System

To load the new definition choose ‘y’; however, if we have redefined a set of functions is useful to choose the option ‘p’ to avoid the repetitive process of showing multiple messages as that presented above and load all the new definitions.

• /run(< file >) combines both commands showed above and executes in sequence the two operations /compile(< file >) + /load(< file >).

It is not possible, in the current version, to compile simultaneously several programs (observe that compile, load y run admits only one argument), but the same effect can be reached with the directive include explained further in Section 2.1.

During the compilation of a program, TOY(FD) works as TOY and processes the entry (i.e., the file) in several stages: initially it checks for the syntax, functional dependencies and types; then, if there is no error, it generates the Prolog code and during this process it visualizes useful information to the user.

Once a program is compiled, the user can write goals at the TOY(FD) command line in a similar form as for Prolog programs. See for instance Section 1.3.5.

1.3.3 Information about Functions

There are also two commands that allow to get information about functions that has been defined in a program:

• /type(< fun >) shows the type of the function < fun >, if this is defined in the program loaded (currently in memory) in the TOY(FD) session.

• /tree(< file >,< fun >) provides information about the translation that TOY(FD) has generated for the function < fun > defined in the file < file >. Particularly, it shows the definitional tree associated to the function López-Fraguas and Sánchez-Hernández, 1999.

1.3.4 Saving the session

Another useful command is /save(< file >) that allows to keep the current state of a session (with the whole set of predicate definitions, functions, types, etc.) to the file < file >.

To restore the state, we only have to execute the file < file > from a command interpreter level (this have been tested in UNIX).

1.3.5 Activating the Output for FD Constraint Solving

TOY(FD) does not incorporate automatically FD constraints and thus it is necessary to load the extension for FDs and activate the output in order to visualize the results of constraint solving.

To show the current state of the constraint store with respect to the variables in the goal, is necessary to activate the constraint output. This is done with the following command:
1.4. FD Constraints

- `/cflpfd`

This command is necessary to see important information about constraint solving. Below we show an example.

TOY(FD)>domain [X] 1 3
   yes
   Elapsed time: 50 ms.

more solutions (y/n/d) [y]?
   no.
   Elapsed time: 0 ms.

TOY(FD)>/cflpfd. % output activated

TOY(FD)>domain [X] 1 3
   yes
   \( X \) in 1..3 % Information about \( X \) in the constraint store
   Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
   no.
   Elapsed time: 0 ms.

TOY(FD)>

1.4 FD Constraints

1.4.1 Predefined Data Types

In TOY(FD), we have defined a set of predefined datatypes that are used to define the constraints. Table 1.1 shows the set of these datatypes.

<table>
<thead>
<tr>
<th>data labelType = ff</th>
<th>ffc</th>
<th>leftmost</th>
<th>mini</th>
<th>maxi</th>
<th>step</th>
<th>enum</th>
<th>bisect</th>
<th>up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>down</td>
<td>all</td>
<td>toMinimize int</td>
<td>toMaximize int</td>
<td>assumptions int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data statistics = resumptions</td>
<td>entailments</td>
<td>prunings</td>
<td>backtracks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data reasoning = value</td>
<td>domains</td>
<td>range</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data options = on reasoning</td>
<td>complete bool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data typeprecedence = d (int,int,int)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data newOptions = precedences [typeprecedence]</td>
<td>path_consistency bool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>static_sets bool</td>
<td>edge_finder bool</td>
<td>decomposition bool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1. The TOY(FD) System

In the following, we describe one by one all the FD constraints supported by TOY(FD). Table 1.2 shows the whole set of FD constraints currently supported by the system.

Table 1.2: The Predefined Set of FD Constraints

<table>
<thead>
<tr>
<th>RELATIONAL CONSTRAINTS</th>
<th>(&gt;, &lt;, &gt;=, &lt;=) :: int → int → bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETICAL CONSTRAINTS</td>
<td>(*, /) :: int → int → int</td>
</tr>
<tr>
<td></td>
<td>sum :: [int] → (int → int → bool) → int → bool</td>
</tr>
<tr>
<td></td>
<td>scalar_product :: [int] → [int] → (int → int → bool) → int → bool</td>
</tr>
<tr>
<td>COMBINATORIAL CONSTRAINTS</td>
<td>assignment, circuit’, serialized :: [int] → [int] → bool</td>
</tr>
<tr>
<td></td>
<td>all_different, all_distinct, circuit :: [int] → bool</td>
</tr>
<tr>
<td></td>
<td>all_different’, all_distinct’ :: [int] → [options] → bool</td>
</tr>
<tr>
<td></td>
<td>serialized’ :: [int] → [int] → [newOptions] → bool</td>
</tr>
<tr>
<td></td>
<td>count :: int → [int] → (int → int → bool) → int → bool</td>
</tr>
<tr>
<td></td>
<td>element, exactly :: int → [int] → int → bool</td>
</tr>
<tr>
<td></td>
<td>cumulative :: [int] → [int] → [int] → int → bool</td>
</tr>
<tr>
<td></td>
<td>cumulative’ :: [int] → [int] → [int] → [int] → int → [newOptions] → bool</td>
</tr>
<tr>
<td>STATISTICS CONSTRAINTS</td>
<td>fd_statistics :: statistics → int → bool</td>
</tr>
<tr>
<td></td>
<td>fd_statistics’ :: bool</td>
</tr>
<tr>
<td>ENUMERATION CONSTRAINTS</td>
<td>indomain :: int → bool</td>
</tr>
<tr>
<td></td>
<td>labeling :: [labelType] → [int] → bool</td>
</tr>
<tr>
<td>MEMBERSHIP CONSTRAINTS</td>
<td>domain :: [int] → int → int → bool</td>
</tr>
<tr>
<td>PROPOSITIONAL CONSTRAINTS</td>
<td>#&lt;=&gt; :: bool → bool → bool</td>
</tr>
</tbody>
</table>

1.4.2 Membership Constraint

It restricts a list of FD variables to have values in an integer interval.

domain/3

- Type declaration:
  domain :: [int] → int → int → bool
- Definition: `domain L A B` returns true if each element in the list L (with integers and/or FD variables) belongs to the closed interval [A,B] and also constrains each FD variable in L to have values in the integer interval [A,B].
1.4. FD Constraints

- **Example in the TOY(FD) command level:**
  
  Next goal returns true and constrains X and Y to have values in the range [1,10].

  ```
  TOY(FD)> domain [X,Y] 1 10
  yes
  Y in 1..10
  X in 1..10
  Elapsed time: 0 ms.
  ```

1.4.3 Relational Constraints

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

- **Type declaration:**

  ```
  (>) :: int \rightarrow int \rightarrow bool
  (<) :: int \rightarrow int \rightarrow bool
  (>=) :: int \rightarrow int \rightarrow bool
  (<=) :: int \rightarrow int \rightarrow bool
  (==) :: int \rightarrow int \rightarrow bool
  (\neq) :: int \rightarrow int \rightarrow bool
  ```

- **Definition:** 
  'A #Op B', also written in infix notation as 'A #Op B', where Op $\in \{>, <, \geq, \leq, \geq, \leq\}$, returns true if the imposition of the relation 'A #Op B' entails the constraint store.

- **Remarks:** Infix notation allowed.

- **Priorities:**

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

- **Example at the TOY(FD) command level:**

  Next goal returns true and restricts X and Y to have values in the intervals [2,10] and [1,9] respectively (as a consequence of range narrowing).
CHAPTER 1. The TOY(FD) System

TOY(FD)> domain [X,Y] 1 10, X #> Y
   yes
   Y in 1..9
   X in 2..10
Elapsed time: 0 ms.

• Type declaration:

- (#*) :: int → int → int
- (#+) :: int → int → int
- (#-) :: int → int → int
- (#/) :: int → int → int

• 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.

• Remarks: Infix notation allowed.

• Priorities:

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

• Example at the TOY(FD) command level:

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
   yes
   Z in 1..100
   Y in 1..10
   X in 1..10
Elapsed time: 0 ms.

1.4.4 Arithmetic Constraints

The following constraints express a relation between a sum or scalar product and a value, using a dedicated algorithm that avoids creating any temporary variables holding intermediate values.

sum/3
1.4. FD Constraints

- **Type declaration:**
  \[ \text{sum :: } [\text{int}] \rightarrow (\text{int} \rightarrow \text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool} \]

- **Definition:** \( \text{sum L Op V} \) is true if the sum of all elements in \( L \) is related with \( V \) via the relational operator \( \text{Op} \). i.e., if \( \sum_{e \in L} \text{Op V} \).

- **Example at the TOY(FD) command level:**
  Next goal returns true and constrains \( X, Y \) and \( Z \) to be 1.
  \[
  \text{TOY(FD)} > \text{L == [X,Y,Z], domain L 1 3, sum L (#<) 4} \]
  yes
  L == [ 1, 1, 1 ]
  Z == 1
  Y == 1
  X == 1
  Elapsed time: 0 ms.

- **Type declaration:**
  \[ \text{scalar_product :: } [\text{int}] \rightarrow [\text{int}] \rightarrow (\text{int} \rightarrow \text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool} \]

- **Definition:** ‘\( \text{scalar_product L1 L2 Op V} \)’ is true if the scalar product (in the sense of FD constraint solving) of the integers or FD variables in \( L_1 \) and \( L_2 \) is related with the value \( V \) via the operator \( \text{Op} \), i.e., if ‘\( (L_1 \ast \text{s} L_2) \text{ Op V} \)’ is satisfied with \( \ast \text{s} \) defined as the usual scalar product of integer vectors.

- **Example at the TOY(FD) command level:**
  \[
  \text{TOY(FD)} > \text{domain [X,Y,Z] 1 10, scalar_product [1,2,3] [X,Y,Z] (#<) 10} \]
  yes
  Z in 1..2
  Y in 1..2
  X in 1..4
  Elapsed time: 0 ms.

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

1.4.5 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** Beldiceanu, 2000.
• **Type declaration:**

\[
\text{assignment :: [int] \rightarrow [int] \rightarrow bool}
\]

• **Definition:** 'assignment' 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, 'assignment \(L_1 \leftrightarrow L_2\)' is true if for all \(i, j \in \{1, \ldots, n\}\), and \(X_i \in L_1, Y_j \in L_2\), then \(X_i = j\) if and only if \(Y_j = i\).

• **Example at the TOY(FD) command level:**

Next goal returns true and constrains \(X, Y\) and \(Z\) to be 3, 1 and 2 respectively.

\[
\text{TOY(FD)> domain [X,Y,Z] 1 3, assignment [X,Y,Z] [2,3,D]}
\]

\[
\text{yes}
\]

\[
\begin{align*}
X & = 3 \\
Y & = 1 \\
Z & = 2 \\
D & = 1
\end{align*}
\]

Elapsed time: 0 ms.

• **Type declaration:**

\[
\text{circuit :: [int] \rightarrow bool}
\]

\[
\text{circuit' :: [int] \rightarrow [int] \rightarrow bool}
\]

• **Definition:** 'circuit \(L_1\)' and 'circuit \(L_1 \leftrightarrow L_2\)' are true if the values in \(L_1\) forms 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. \(L_1\) and \(L_2\) are lists of FD variables or integers of length \(N\), where the \(i\)-th element of \(L_1\) (resp. \(L_2\)) is the successor (resp. predecessor) of \(i\) in the graph.

• **Example at the TOY(FD) command level:**

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.

\[
\text{count/4}
\]

• **Type declaration:**

\[
\text{count :: int \rightarrow [int] \rightarrow (int \rightarrow int \rightarrow bool) \rightarrow int \rightarrow bool}
\]

• **Definition:** 'Count \(V \ L \ Op \ Y\)' is true if the number of elements of \(L\) that are equal (in the sense of 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).

• **Example at the TOY(FD) command level:**

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.
1.4. FD Constraints

TOY(FD)> L == [X,1,2], domain [X] 1 2, count 1 L (#=) 2
   yes
   L == [ 1, 1, 2 ]
   X == 1
   Elapsed time: 0 ms.

exactly/3

• Type declaration:
  exactly :: int → [int] → int → bool

• Definition: ‘exactly X L N’ is true if X occurs exactly N times in the list L.

• Example at the TOY(FD) command level:
  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
   yes
   L == [ 2, 2 ]
   X == 2
   Y == 2
   Elapsed time: 0 ms.

element/3

• Type declaration:
  element :: int → [int] → int → bool

• Definition: ‘element X L Y’ is true if the X-th element in the list L is Y (in the sense of FD).

• Example at the TOY(FD) command level:

TOY> L == [X,Y,Z], domain L 1 10, element 2 L 7
   yes
   L == [ X, 7, Z ]
   Y == 7
   Z in 1..10
   X in 1..10
   Elapsed time: 0 ms.

serialized/2
serialized'/3
cumulative/4
cumulative/5
CHAPTER 1. The TOY(FD) System

- **Type declaration:**
  
  \[
  \begin{align*}
  \text{serialized} & : \quad [\text{int}] \to [\text{int}] \to \text{bool} \\
  \text{serialized’} & : \quad [\text{int}] \to [\text{int}] \to [\text{newOptions}] \to \text{bool} \\
  \text{cumulative} & : \quad [\text{int}] \to [\text{int}] \to [\text{int}] \to \text{int} \to \text{bool} \\
  \text{cumulative’} & : \quad [\text{int}] \to [\text{int}] \to [\text{int}] \to [\text{int}] \to \text{newOptions} \to \text{bool}
  \end{align*}
  \]

- **Definition:** cumulative/5, cumulative/6, serialized/3 and serialized’/4 are useful to solve scheduling and placements problems. In general,

  - 'cumulative \([S1,Sn]\) \([D1,Dn]\) \([R1,Rn]\) Limit’
  - 'cumulative \([S1,Sn]\) \([D1,Dn]\) \([R1,Rn]\) Limit Options’
  - 'serialized \([S1,Sn]\) \([D1,Dn]\)’
  - 'serialized \([S1,Sn]\) \([D1,Dn]\) Options’

  are 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. 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 \((V1,V2,I)\), and I is the value superior (to denote a fictitious -lifted- top element) or lift I with \(I \in \text{Integers}\). Each element imposes the constraint \(SV_{i+1} \leq SV_2\), if I is an integer; and \(SV_2 \leq SV_i\) otherwise.

- **Example at the TOY(FD) command level:**

  \[
  \begin{align*}
  \text{TOY(FD) > cumulative} & \quad [0,5,8] \quad [1,1,2] \quad == \quad U, \quad U \quad [1,1,1] \quad 10 \quad == \quad I \\
  & \quad \text{yes} \\
  & \quad U == \quad (\text{cumulative} \quad [ \quad 0, \quad 5, \quad 8 \quad ] \quad [ \quad 1, \quad 1, \quad 2 \quad ]) \\
  & \quad I == \quad \text{true} \\
  & \quad \text{Elapsed time: 0 ms.}
  \end{align*}
  \]
1.4. FD Constraints

- **Type declaration:**
  ```
  all_different :: [int] → bool
  all_different' :: [int] → [options] → bool
  all_distinct :: [int] → bool
  all_distinct' :: [int] → [options] → bool
  ```

- **Definition:**
  - ‘all_different $L$’ and ‘all_distinct $L$’ are true if each variable 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 difference between both constraints is that all_different/1 uses a complete algorithm that maintains the domain consistency Régin, 1994 whereas all_distinct/1 uses an incomplete one.

  There are extended versions that allow one more argument which is a list of options, where each option may have one of the following values

  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 a variable changes.

  2. ‘complete true’ or ‘complete false’ to specify if the propagation algorithm to apply is complete or incomplete.

- **Example at the TOY(FD) command level:**

  ```
  TOY(FD)> L == [X,1,Z], domain L 1 3, all_different' L [complete true, on range] yes L == [ X, 1, Z ] Z in 2..3 X in 2..3 Elapsed time: 0 ms.
  ```

1.4.6 Enumeration Constraints

Enumeration constraints reactivate the search process when no more constraint propagation is possible. TOY(FD) provides two enumeration constraints:

```
CHAPTER 1. The TOY(FD) System

- **Type declaration:**
  
  \[
  \texttt{indomain} :: \texttt{int} \to \texttt{bool}
  \]

- **Definition:** ‘\texttt{indomain }X\texttt{’ assigns a value, from the minimum to the maximum in its domain, to }X\texttt{.}

- **Example at the TOY(FD) command level:**
  
  Next goal assigns (by backtracking and from minimum to maximum in its range) each value to a variable.

  \[
  \text{TOY(FD)} > \text{domain }[X] \ 1 \ 2, \text{indomain }X
  \]

  \[
  \begin{align*}
  \text{yes} \\
  X & = 1 \\
  \text{Elapsed time: } & 0 \ms
  \end{align*}
  \]

  \[
  \text{more solutions (y/n/d) }[y]? \\
  \begin{align*}
  \text{yes} \\
  X & = 2 \\
  \text{Elapsed time: } & 0 \ms
  \end{align*}
  \]

- **Type declaration:**
  
  \[
  \texttt{labeling} :: [\texttt{labelType}] \to [\texttt{int}] \to \texttt{bool}
  \]

- **Definition:** ‘\texttt{labeling Options }L\texttt{’ is true if an assignment of the variables in }L\texttt{ can be found such that all of the constraints already presented in the constraint store are satisfied. }Options\texttt{ is a list of at most four elements that allow to specify the nature of the search.}

  Each element in this list may 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\texttt{ (\texttt{leftmost}), the variable with the smallest lower bound (\texttt{mini}), the variable the greatest upper bound (\texttt{maxi}) or the variable with the smallest domain (\texttt{ff}). The value \texttt{ffc} extends the option \texttt{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 (\texttt{enum}), by selecting the minimum or maximum (\texttt{step}) or by dividing the domain in two choices by the mid point (\texttt{bisect}). Also the domain of a variable can be explored in ascending order (\texttt{up}) or in descending order (\texttt{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 of assumptions $K$ (choices) made during the search (assumptions $K$).

- **Example at the TOY(FD) command level:**

Next goal looks (by backtracking) for assignments to all variables in list $L$ and follows a first fail strategy.

```
TOY> L == [X,Y], domain L 1 3 == T,
    labeling [ff] L == true
    yes
    L == [ 1, 1 ]
    X == 1
    Y == 1
    T == true
    Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
    yes
    L == [ 1, 2 ]
    X == 1
    Y == 2
    T == true
    Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
    .......
```

1.4.7 **Statistics Constraints**

TOY(FD) also provides a set of constraints that allow to recover information about constrained FD variables and their associated domains during the solving of a goal. In this group we have the *statistics constraints* to generate execution statistics that are useful in many forms as, for example, to provide information about the solving of a goal.

**A remark about statistics constraints.** Statistics functions pose no problem for our functional logic programming framework since they behave as usual TOY functions, returning values that, in general, depend on the computation state.

```
fd_statistics'/0
```

- **Type declaration:**
fd_statistics' :: bool

- **Definition**: \( fd_{\text{statistics}}' \) always returns true and displays a set of useful statistics (usually in the console).

- **Example at the TOY(FD) command level**:

```plaintext
TOY(FD)> L= [X,Y,Z], domain L 1 4,
    all_different' L [complete true],
    labeling [] L, fd_statistics'
    
    Resumptions: 4  
    Entailments: 1  
    Prunings: 9  
    Backtracks: 0  
    Constraints created: 1
    
    yes
    L == [ 1, 2, 3 ]
    X == 1
    Y == 2
    Z == 3
    Elapsed time: 0 ms.

    more solutions (y/n/d) [y]? ...........
```

- **Type declaration**:

```plaintext
fd_statistics :: statistics → int → bool
```

- **Definition**: ‘\( fd_{\text{statistics}} \) Key V’ returns true if either
  - V unifies with the number of constraints created and also Key == constraints’ or
  - V unifies with the number of times that
    1. a constraint is resumed and ‘Key == resumptions’,
    2. a (dis) entailment was detected by a constraint and ‘Key == entailments’,
    3. a domain was pruned and Key == prunings”,
    4. a backtracking was executed because a domain becomes empty and Key == backtracks’.
1.5 Some more constraints.....

TOY(FD) also provides a set of constraints (not shown in Table 1.2), called reflection constraints, that allow to recover information about constrained FD variables and their associated domains during the solving of a goal.

**EXAMPLE 1.1** *The reflection constraints*

\[
\text{fd_min}, \text{fd_max} :: \text{int} \rightarrow \text{int}
\]

applied to a FD variable return respectively the minimum and maximum value of this FD variable in its current domain).

These constraints are the issue of a further paper and for this reason they will not be described here in detail. The interested reader is encouraged to visit the link proposed in Fernández et al., 2002b for a more detailed explanation (when available as this will information be published as new constraints appear).
CHAPTER 1. The TOY(FD) System
Chapter 2

TOY(FD) Programming Examples

2.1 Important Note about Programming with TOY(FD)

In many cases, it can be useful to modularize a program and divide it in smaller pieces of code. As many other programming languages, TOY(FD) provides a directive that allows to use code written in a TOY(FD) file in another TOY(FD) program without rewriting it. This directive is \texttt{include} and follows the following syntax\footnote{Observe that this directive does not begin with / as it is not a directive to be used at the command level, but directly inserted in the program –see examples in this chapter.}:

- \texttt{include "file.toy"}

For example, if we are writing a program in a file \texttt{file.toy}, we can use the directive \texttt{include "file.toy"} in any part of the program (usually in the beginning) and, as a consequence, our program can make use of any function or data type defined in \texttt{file.toy}.

Two important remarks in this sense:

- If a program requires the use of functions for finite domain constraint solving, it must contain the directive

  \texttt{include "cflpfd.toy"}

  as the file \texttt{cflpfd.toy} contains the definitions of data types and functions needed to execute constraint solving in the finite domain (see examples in the rest of the chapter).

- The file “misc.toy” is distributed with the system and provides many functions frequently used (see Appendix C). As a consequence, many programs include in their codes the directive

  \texttt{include "misc.toy"}
2.2 Introductory TOY(FD) Examples

2.2.1 Send+More = Money

Below, a TOY(FD) program (included in the distribution in the directory Examples in the file smm.toy) to solve the classical arithmetic puzzle “send more money” is shown. Observe that TOY(FD) 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 1.1 and 1.2, and explanations given in Section 1.4.

```toy
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
    #*
    100 ** 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 1.3.2), we can solve goals as follows from the command line of TOY(FD):²:

```toy
TOY(FD)> smm L [ff]
yes
L == [ 9, 5, 6, 7, 1, 0, 8, 2 ]

Elapsed time: 0 ms.
more solutions (y/n/d) [y]? ; \% Do not look for more solutions!
```

2.2.2 N-queens

Here we present a TOY(FD) solution (included in the distribution in the directory Examples in the file queens.toy), for the classical problem of the \( N \) queens problem: place

²Remember that to activate the constraint output, the directive /cflpfd must be typed at the command line before compiling the program.
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 both the FD constraint functions and the function length/1 (predefined in the file misc.toy).

```toy
include "cflpfd.toy"
include "misc.toy"  %%To use length/1

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 the following:

TOY(FD)> queens 4 L [ff]
yes
  L == [ 2, 4, 1, 3 ]

  Elapsed time: 0 ms.
more solutions (y/n/d) [y]?
yes
  L == [ 3, 1, 4, 2 ]
```
2.2.3 A Cryptoarithmetic Problem

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,
PUGUE = 50,
OPERA = 65,
SOLO = 37,
WALTZ = 34

A TOY(FD) 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 <==
   domain LD 1 26,
   all_different LD,
   B #+ A #+ L #+ L #+ E #+ T 
   #= 45,
```
2.2. Introductory TOY(FD) Examples

Solving in the command level is shown below:

TOY(FD)> alpha [ff] L
  yes
  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.

2.2.4 Magic Sequences

We present another simple (and well-known) example, the magic series problem Van Hentenryck, 1989. 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 TOY(FD) solution, included in the distribution in the directory Examples in the file magicseq.toy, to this problem.

include "cflpfd.toy"
include "misc.toy"  %% To use length/1

magic :: int -> [int] -> [labelingType] -> bool
magic N L Label = true <==
  length L == N,
  domain L 0 (N-1),
labeling Label LD

Solving in the command level is shown below:

TOY(FD)> alpha [ff] L
  yes
  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.
constrain L L 0 Cs, % essential
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 % Functional syntax (I+1)

Below we show an example of solving including compiling and loading process:

TOY(FD)> /compile(magicseq)
Checking syntax Including File cflpfd.toy...............
...........................................................................
PROCESS COMPLETE

TOY(FD)> /load(magicseq)
{compiling c:/sicstus382/bin/toyfd/magicseq.pl...}
{module plgenerated imported into initToy}
{module toycomm imported into plgenerated}
{module primFunct imported into plgenerated}
{module primitivCod imported into plgenerated}
{module clpfd imported into plgenerated}
{compiled c:/sicstus382/bin/toyfd/magicseq.pl in
module plgenerated, 610 msec -32 bytes}

TOY(FD)> magic 10 L []
yes
L == [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ]

Elapsed time: 0 ms.

more solutions (y/n/d) [y]? ; % Do not look for more solutions
2.3 More Complex Examples

2.3.1 A Scheduling Problem

Here, we consider the problem of scheduling tasks that require resources to complete, and have to fulfill precedence constraints\(^3\). Figure 2.1 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 2.1: Precedence Graph.](image)

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.

```prolog
include "cflpfld.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
```

\(^3\)Adapted from Marriot and Stuckey, 1998.
CHAPTER 2. TOY(FD) Programming Examples

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 <= belongs (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,
    belongs 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

belongs :: A -> [A] -> bool
belongs R [] = false
belongs R [R|Rs] = true
belongs R [R1|Rs] = belongs R Rs

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.

\[
\text{sched} :: \text{startType} \to \text{startType} \to \text{startType} \to \text{startType} \to \text{startType} \to \text{bool} \\
\text{sched} S1 S2 S3 S4 S5 S6 :- \\
\text{Tasks} == [(t1,3,[t5,t6],[m1],S1), \\
\quad (t2,8,[], [m1], S2), \\
\quad (t3,8,[], [m1], S3), \\
\quad (t4,6,[t3],[m2],S4), \\
\quad (t5,3,[t3],[m2],S5), \\
\quad (t6,4,[], [m2], S6)], \\
\text{schedule Tasks} 1 20, \\
\text{labeling [ff]} [S1,S2,S3,S4,S5,S6]
\]

where Tasks defines the set of tasks. Observe that the problem for a possible scheduling is limited to time window (1,20) by the sub-goal schedule Tasks 1 20. A solving in the command line:

TOY(FD)> sched S1 S2 S3 S4 S5 S6

yes
S1 == 1
S2 == 4
S3 == 12
S4 == 1
S5 == 7
S6 == 10

Elapsed time: 0 ms.

more solutions (y/n/d) [y]?

yes
S1 == 1
S2 == 4
S3 == 12
S4 == 1
S5 == 7
S6 == 11

Elapsed time: 0 ms.

more solutions (y/n/d) [y]? ; % Do not look for more solutions

2.3.2 A Hardware Design Problem

A more interesting example comes from the hardware area. 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 Zhou, 1996.

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 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:

<table>
<thead>
<tr>
<th>Gate</th>
<th>Area</th>
<th>Power</th>
<th>Cost</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

In the sequel we will introduce the problem by first considering the features TOY offers
for specifying logical circuits, what are its weaknesses, and how they can effectively be solved with the integration of constraints in TOY(FD).

**FLP Simple Circuits**

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, i1, i2 :: behavior
i0 I2 I1 I0 = I0
i1 I2 I1 I0 = I1
i2 I2 I1 I0 = I2

notGate :: behavior → behavior
notGate B I2 I1 I0 = not (B I2 I1 I0)

andGate :: behavior → behavior → behavior
andGate B1 B2 I2 I1 I0 = and (B1 I2 I1 I0) (B2 I2 I1 I0)

orGate :: behavior → behavior → behavior
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 2.2. 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 inputs-circuits (see Figure 2.2).

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 2.3 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:
CHAPTER 2. TOY(FD) Programming Examples

Figure 2.2: Basic Modules.

Figure 2.3: Two-Input Multiplexer Circuit.

TPY(FD) > P == orGate (andGate i0 (notGate i2)) (andGate i1 i2),
P true false true == 0
is evaluated to true and produces the substitution 0 == 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.

\[
\begin{align*}
\text{power} & : \text{behavior} \to \text{int} \\
\text{power} \ i0 & = 0 \\
\text{power} \ i1 & = 0 \\
\text{power} \ i2 & = 0 \\
\text{power} \ (\text{notGate} \ \text{C}) & = \text{notGatePower} + (\text{power} \ \text{C}) \\
\text{power} \ (\text{andGate} \ \text{C1} \ \text{C2}) & = \text{andGatePower} + (\text{power} \ \text{C1}) + (\text{power} \ \text{C2}) \\
\text{power} \ (\text{orGate} \ \text{C1} \ \text{C2}) & = \text{orGatePower} + (\text{power} \ \text{C1}) + (\text{power} \ \text{C2})
\end{align*}
\]

Then, we can submit the goal \( \text{power} \ \text{B} == \text{P}, \ \text{P} < \maxPower \) (provided the function \maxPower acts as a problem parameter that returns just the maximum power allowed for the circuit) in which the function \text{power} is used as a behavior generator\(^4\). As outcome, we get several solutions

\(^4\)Equivalently and more concisely, \( \text{power} \ \text{B} < \maxPower \) could be submitted, but doing so we make the power unobservable.
2.3. More Complex Examples

\[
\begin{align*}
B &= i_0, P = 0, \\
B &= i_1, P = 0, \\
B &= i_2, P = 0, \\
B &= \neg i_0, P = 1, \\
&\ldots \\
B &= \neg (\neg i_0), P = 2, \ldots
\end{align*}
\]

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 \(\neg (\ldots(\neg i_0)\ldots)\), because the application of the fourth rule of \texttt{power} yields to an infinite computation. This drawback is solved by recursing to Peano’s arithmetics, that is:

\begin{verbatim}
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))
\end{verbatim}

So, we can submit the goal \texttt{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.

CFLP(FD) Simple Circuits

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 \(\langle \text{area}, \text{power}, \text{cost}, \text{delay} \rangle\). The idea to formulate the problem consists of attaching this state to an ongoing circuit so that state variables
CHAPTER 2. TOY(FD) Programming Examples

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) == (B, (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) = (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)),

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:

```haskell
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))
  domain [Din1, Din2] ((fd_min Dout) + andGateDelay)(fd_max Dout),
genCirDelay (A, P, C, D) Din1 == (B1, (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)),
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 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]

The whole TOY(FD) code, included in the distribution file circuit3.toy, is shown below

include "cflpfd.toy"

% Type Definitions

type area = int
type power = int
type cost = int
type delay = int

type state = (area, power, cost, delay)
2.3. More Complex Examples

type behaviour = bool -> bool -> bool -> bool
type circuit = (behaviour, state)
type functionality = [bool]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Problem Parameters  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Gate Area  
notGateArea, andGateArea, orGateArea :: int  
notGateArea = 1  
andGateArea = 2  
orGateArea = 2

% Gate Power Dissipation  
notGatePower, andGatePower, orGatePower :: int  
notGatePower = 1  
andGatePower = 2  
orGatePower = 2

% Gate Cost  
notGateCost, andGateCost, orGateCost :: int  
notGateCost = 1  
andGateCost = 1  
orGateCost = 2

% Gate Delay  
notGateDelay, andGateDelay, orGateDelay :: int  
notGateDelay = 1  
andGateDelay = 1  
orGateDelay = 2

% Circuit Constraints  
maxArea, maxPower, maxCost, maxDelay :: int  
maxArea = 20  
maxPower = 20  
maxCost = 20  
maxDelay = 20

% Functionality  
circuitOutput :: functionality  
circuitOutput = [false,false,false,false,false,false,false,false]  % FALSE  
circuitOutput1 = [true,true,true,true,true,true,true,true] % TRUE  
circuitOutput2 = [true,false,true,false,true,false,true,false] % NOT BIT0  
circuitOutput3 = [false,true,true,false,true,false,false,true] % XOR
circuitOutput4 = [true,true,true,true,true,true,true,false] % NAND

% Inputs
i2 :: behaviour
i2 I2 I1 I0 = I2

i1 :: behaviour
i1 I2 I1 I0 = I1

i0 :: behaviour
i0 I2 I1 I0 = I0

% Gate Definitions
notGate :: behaviour -> behaviour
notGate B I2 I1 I0 = not (B I2 I1 I0)

andGate :: behaviour -> behaviour
andGate B1 B2 I2 I1 I0 = and (B1 I2 I1 I0) (B2 I2 I1 I0)

orGate :: behaviour -> behaviour
orGate B1 B2 I2 I1 I0 = or (B1 I2 I1 I0) (B2 I2 I1 I0)

% Behaviour Generation
genBeh :: state -> behaviour
genBeh (A, P, C, D) = i0
genBeh (A, P, C, D) = i1
genBeh (A, P, C, D) = i2

% Boolean Functions
not :: bool -> bool
not true = false
not false = true

or :: bool -> bool -> bool
or true X = true
or false X = X

and :: bool -> bool -> bool
and true X = X
and false X = false
2.3. More Complex Examples

domain [C] ((fd_min C) + notGateCost) (fd_max C)
genBeh (A, P, C, D) = andGate (genBeh (A, P, C, D)) (genBeh (A, P, C, D)) ==
domain [A] ((fd_min A) + andGateArea) (fd_max A),
domain [P] ((fd_min P) + andGatePower) (fd_max P),
domain [C] ((fd_min C) + andGateCost) (fd_max C)
genBeh (A, P, C, D) = orGate (genBeh (A, P, C, D)) (genBeh (A, P, C, D)) ==
domain [A] ((fd_min A) + orGateArea) (fd_max A),
domain [P] ((fd_min P) + orGatePower) (fd_max P),
domain [C] ((fd_min C) + orGateCost) (fd_max C)

% Circuit Generation
genCir :: state -> circuit
genCir (A,P,C,D) =
genCirDelay (A,P,C,D) D

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 [A] ((fd_min A) + notGateArea) (fd_max A),
domain [P] ((fd_min P) + notGatePower) (fd_max P),
domain [C] ((fd_min C) + notGateCost) (fd_max C),
domain [Dout] ((fd_min Dout) + notGateDelay) (fd_max Dout),
genCirDelay (A, P, C, D) Dout = (B, (A, P, C, D))
domain [A] ((fd_min A) + andGateArea) (fd_max A),
domain [P] ((fd_min P) + andGatePower) (fd_max P),
domain [C] ((fd_min C) + andGateCost) (fd_max C),
domain [Din1, Din2] ((fd_min Din1) + andGateDelay) (fd_max Din1),
genCirDelay (A, P, C, D) Din1 = (B1, (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 [A] ((fd_min A) + orGateArea) (fd_max A),
domain [P] ((fd_min P) + orGatePower) (fd_max P),
domain [C] ((fd_min C) + orGateCost) (fd_max C),
domain [Din1, Din2] ((fd_min Din1) + orGateDelay) (fd_max Din1),
genCirDelay (A, P, C, D) Din1 = (B1, (A, P, C, D)),
domain [Dout] (maximum (fd_min Din1) (fd_min Din2)) (fd_max Dout)

% genCir without delay. Only area
genCir' :: state -> circuit


\[ \text{CHAPTER 2. TOY(FD) Programming Examples} \]

\[ \text{genCir'} (A, P, C, D) = (i0, (A, P, C, D)) \]
\[ \text{genCir'} (A, P, C, D) = (i1, (A, P, C, D)) \]
\[ \text{genCir'} (A, P, C, D) = (i2, (A, P, C, D)) \]
\[ \text{genCir'} (A, P, C, D) = (\text{notGate } B, (A, P, C, D)) \implies \]
\[ \begin{align*}
\text{domain } [A] &\text{ ((fd}\text{min } A) + \text{notGateArea) (fd}\text{max } A), \\
\text{genCir}' & (A, P, C, D) == (B, (A, P, C, D)) \\
\end{align*} \]
\[ \text{genCir'} (A, P, C, D) = (\text{andGate } B1 B2, (A, P, C, D)) \implies \]
\[ \begin{align*}
\text{domain } [A] &\text{ ((fd}\text{min } A) + \text{andGateArea) (fd}\text{max } A), \\
\text{genCir}' & (A, P, C, D) == (B1, (A, P, C, D)), \\
\text{genCir}' & (A, P, C, D) == (B2, (A, P, C, D)) \\
\end{align*} \]
\[ \text{genCir'} (A, P, C, D) = (\text{orGate } B1 B2, (A, P, C, D)) \implies \]
\[ \begin{align*}
\text{domain } [A] &\text{ ((fd}\text{min } A) + \text{orGateArea) (fd}\text{max } A), \\
\text{genCir}' & (A, P, C, D) == (B1, (A, P, C, D)), \\
\text{genCir}' & (A, P, C, D) == (B2, (A, P, C, D)) \\
\end{align*} \]

% Delay

\[ \text{circuitDelay :: circuit }\rightarrow\text{ bool} \]
\[ \text{circuitDelay } (B, (A, P, C, D)) :=- \\
\text{ delay } (B, (A, P, C, D)) D \]
\[ \text{delay :: circuit }\rightarrow\text{ delay }\rightarrow\text{ bool} \]
\[ \text{delay } (i0, (A, P, C, D)) Dout = true \]
\[ \text{delay } (i1, (A, P, C, D)) Dout = true \]
\[ \text{delay } (i2, (A, P, C, D)) Dout = true \]
\[ \text{delay } ((\text{notGate } B), (A, P, C, D)) Dout :- \\
\text{ domain } [Dout] \text{ ((fd}\text{min } Dout) + \text{notGateDelay) (fd}\text{max } Dout), \\
\text{ delay } (B, (A, P, C, D)) Dout \]
\[ \text{delay } ((\text{andGate } B1 B2), (A, P, C, D)) Dout :- \\
\text{ domain } [Din1, Din2] \text{ ((fd}\text{min } Dout) + \text{andGateDelay) (fd}\text{max } Dout), \\
\text{ delay } (B1, (A, P, C, D)) Din1, \\
\text{ delay } (B2, (A, P, C, D)) Din2, \\
\text{ domain } [Dout] \text{ (maximum (fd}\text{min Din1) (fd}\text{min Din2)) (fd}\text{max } Dout) \]
\[ \text{delay } ((\text{orGate } B1 B2), (A, P, C, D)) Dout :- \\
\text{ domain } [Din1, Din2] \text{ ((fd}\text{min } Dout) + \text{orGateDelay) (fd}\text{max } Dout), \\
\text{ delay } (B1, (A, P, C, D)) Din1, \\
\text{ delay } (B2, (A, P, C, D)) Din2, \\
\text{ domain } [Dout] \text{ (maximum (fd}\text{min Din1) (fd}\text{min Din2)) (fd}\text{max } Dout) \]

% Switching Function

\[ \text{switchingFunction :: behaviour }\rightarrow\text{ functionality} \]
\[ \text{switchingFunction } \text{Behaviour} = [\text{Outcome0, Outcome1, Outcome2, Outcome3,} \\
\text{ Outcome4, Outcome5, Outcome6, Outcome7}] \implies \\
\text{ (Behaviour false false false) == Outcome0,} \]
2.3. More Complex Examples

(Behaviour false false true) == Outcome1,
(Behaviour false true false) == Outcome2,
(Behaviour false true true) == Outcome3,
(Behaviour true false false) == Outcome4,
(Behaviour true false true) == Outcome5,
(Behaviour true true false) == Outcome6,
(Behaviour true true true) == Outcome7

% Circuit Generation

\[
\text{genCircuit :: area} \rightarrow \text{power} \rightarrow \text{cost} \rightarrow \text{delay} \rightarrow \text{functionality} \rightarrow \text{circuit}
\]
\[
\text{genCircuit MA MP MC MD F} = (B, S) \leftrightarrow
\]
\[
\text{domain [A] 0 MA,}
\text{domain [P] 0 MP,}
\text{domain [C] 0 MC,}
\text{domain [D] 0 MD,}
\text{genCir (A,P,C,D) == (B, S),}
\text{switchingFunction B == F}
\]

% Auxiliary Predicates

\[
\text{maximum :: int} \rightarrow \text{int} \rightarrow \text{int}
\]
\[
\text{max} X Y = X \leftrightarrow X \geq Y
\]
\[
\text{max} X Y = Y \leftrightarrow X < Y
\]

An example of generating a NOR circuit from the TOY(FD) command level is shown above:

\[
\text{TOY(FD)> F=[true,false,false,false,false,false,false,false],}
\]
\[
\text{genCircuit 6 6 6 6 F} = \text{CIRCUIT}
\]
\[
\text{yes}
\]
\[
F == [true, false, false, false, false, false, false, false]
\]
\[
\text{CIRCUIT} == ((\text{notGate (orGate i0 (orGate i1 i2))}), (_A, _B, _C, _D))
\]
\[
_A \in 5..6
\]
\[
_B \in 5..6
\]
\[
_C \in 5..6
\]
\[
_D \in 5..6
\]

Elapsed time: 1828 ms.

more solutions (y/n/d) [y]?
CHAPTER 2. TOY(FD) Programming Examples

yes

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: 0 ms

more solutions (y/n/d) [y]?

......24 SOLUTIONS!!

Note: The solutions shown above to the problem are included in the distribution in the directory Examples in the files circuit1.toy and circuitFD.toy.

2.3.3 A Colour Problem

2.4 Colour Problem

We want to solve the classical map coloring problem. Consider the simple map shown in Figure 2.4: To solve this problem, we have to specify that some countries have different colors by using the constraint all_different L.

The TOY(FD) code to solve this problem is shown below:

include "cflpfd.toy"

color::[int] -> bool

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],
2.4. Colour Problem

![Puzzle](image)

**Figure 2.4: puzzle**

all_different [I6, I7], all_different [I6, I13],
all_different [I6, I14], all_different [I6, I15],
all_different [I7, I8], all_different [I7, I16],
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],

Below we show an example of constraint solving with two different solution to this
problem:

T OY(FD)> color L
    yes
    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: 16 ms.

more solutions (y/n/d) [y]?
    yes
    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]? T OY(FD)>

2.5 Lazy Constraint Programs

A very powerful characteristic of TOY(FD) is lazy evaluation of goals (to our knowledge, TOY(FD) is the first constraint programming language providing laziness in the solving of goals). In this Section we show some examples of programs that combines FD constraint solving and lazy evaluation.

2.5.1 Hamming Codes

Let H be the smallest subset of N satisfying the following axioms:

- 1 ∈ H
- ∀x.x ∈ H ⇔ 2x, 3x, 5x ∈ H

The problem consists of obtaining the ordered sequence of elements in H, that is to say:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, ...

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

$$\text{map } (2\#) \ h$$
$$\text{map } (3\#) \ h$$
$$\text{map } (5\#) \ h$$

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:

$$\text{merge}3 \ U \ V \ W = (U \setminus V) \cup W$$
where the operation $\alpha/2$ is defined as follows:

\[
\begin{align*}
[] \alpha V &= V : [] \\
U \alpha [] &= U : [] \\
[X|Xs] \alpha [Y|Ys] &= X : (Xs \alpha Ys) \iff X \geq Y \\
[X|Xs] \alpha [Y|Ys] &= X : ([X|Xs] \alpha Ys) \iff X < Y \\
[X|Xs] \alpha [Y|Ys] &= Y : ([X|Xs] \alpha Ys) \iff X > Y
\end{align*}
\]

and finally to add the element 1 as initial element. A schematic picture is shown in Figure 2.5.

A solution in TOY(FD) is shown above where \texttt{merge2} defines the operator $\alpha$.

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


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))
```

Note: The solution shown above is included in the distribution in the directory Examples in the files \texttt{hamming.toy}.
2.5.2 Lazy Magic Sequences

Now we present a lazy solution (included in the distribution in the directory Examples in the file `lazymagicseq.toy`) for the problem of the magic series problem that was already treated in Section 2.2.4. With this new solution we illustrate some of the extra capabilities of the CFLP(FD) approach of TOY(FD) with respect the traditional CLP(FD) approach.

```prolog
include "misc.toy" %% To use take/2, map/2 and ./2 include "cflpfd.toy"

%% Next function generates an infinite list of FD variables ranging in 0..N-1
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
```

A goal `lazymagic N`, for some natural 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). After compiling and loading the code shown above we can solve goals as follows:

```
TOY(FD)> lazymagic 10 == L
yes
L == [ 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 ]
```

Alternatives solutions and more flexibility can be reached in TOY(FD). 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 function5.

5Observe that TOY(FD) uses the typical Prolog notation for lists.
Now we generate an infinite list of lists containing the magic sequences from a number \( N \), i.e., the magic sequences from \( N, N+1, N+2 \), etc.

\[
\text{magicfrom} :: \text{int} \rightarrow [[\text{int}]]
\]

\[
\text{magicfrom} \; N = [\text{lazymagic} \; N | \text{magicfrom}(N+1)]
\]

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:

\[
\text{TOY(FD)} > \text{take 3 (magicfrom 7)} == L
\]

\[
\text{L} == [ [ 3, 2, 1, 1, 0, 0, 0 ], [ 4, 2, 1, 0, 1, 0, 0, 0 ], [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ] ]
\]

Elapsed time: 20 ms.

More expressiveness is shown by mixing curried functions, HO functions, infinite lists and function composition (another nice feature from the functional component of TOY(FD)). For example, consider the TOY(FD) code below:

\[
\text{from} :: \text{int} \rightarrow [\text{int}]
\]

\[
\text{from} \; N = [N | \text{from} \; (N+1)]
\]

\[
\text{lazysseries} :: \text{int} \rightarrow [[\text{int}]]
\]

\[
\text{lazysseries} = \text{map lazymagic.from}
\]

where the operator ‘\( . \)’ defines the composition of functions as follows (again, function \( ./2 \) is predefined in the file \textit{misc.toy}): \[
(\cdot) :: (\text{B} \rightarrow \text{C}) \rightarrow (\text{A} \rightarrow \text{B}) \rightarrow (\text{A} \rightarrow \text{C}) \; (\text{F} \; . \; \text{G}) \; X = \text{F} \; (\text{G} \; X)
\]

Observe that \textit{lazysseries} curries the composition (\textit{map lazymagic}).\textit{from}. Then, it is easy to generate the 3-element list shown above by just typing the goal:

\[
\text{TOY(FD)} > \text{take 3 (lazysseries 7)} == L
\]

\[
\text{L} == [ [ 3, 2, 1, 1, 0, 0, 0 ], [ 4, 2, 1, 0, 1, 0, 0, 0 ], [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ] ]
\]

This goal is equivalent to the following:

\[
\text{TOY(FD)} > \text{take 3 (map lazymagic (from 7))} == L
\]

\( ^6 \)The function \textit{take}/2 is predefined in the file \textit{misc.toy} (see Appendix C).
This simple example gives an idea of the nice features of TOY(FD) that combines
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(FD).


Appendix A

TOY(FD) Grammar

FD constraints are declared as classical TOY functions and, as a consequence, the grammar of TOY(FD) is equivalent to the grammar of TOY. This appendix, mainly borrowed from Sánchez-Hernández, 1998, contains the basic rules of this grammar. We adopt the following notation:

1. Terminal symbols appear in standard typography.
2. Reserved terminal symbols appear enclosed in boxes.
3. Vertical bars ‘|’ are used to separate different alternatives.
4. Items between ‘[’ and ‘]’ are optional.
5. The notation \((\text{item})^*\) represents zero or more occurrences of \(\text{item}\).
6. The notation \((\text{item})^+\) represents one or more occurrences of \(\text{item}\).

We distinguish the following classes of tokens:

- **Constructor symbols** (\textit{consym}) and **Function Symbols** (\textit{funsym}): Identifiers starting with a lowercase letter and followed by an arbitrary number of letters, digits, apostrophes (’) and/or underlines (_). For example, \textit{f}, \textit{reverse} and \textit{nat} are valid constructor and function names. There are also some reserved words for the language that cannot be used as identifiers. These are:

  data,    type,    infixl,   infixr,  infix,    primitive, 
  if,      then,    else,     include, where,  let, 
  in,      subtype, int,      real,    char,     bool
The identifiers `let`, `where`, `in` and `subtype` correspond to non-implemented constructions in the current version, but they are reserved for future versions of the system.

- **Variable Symbols** (`varsym`): Identifiers starting with a uppercase letter and followed by an arbitrary number of letters, digits, apostrophes (`'`) and/or underlines (`_`). *Anonymous* variables are allowed and begins with a underline (`_`) (we remind that two anonymous variables with the same name correspond to different variables). For example,

  \[ I\_don't\_Know\_What\_To\_Do, X, Nothing, _anything \]

  are valid variable names, whereas

  \[ Mr.Mr , 9Days, waiting\_for\_you \]

  are not.

- **Predefined Type Symbols** (`typesym`): Identifiers of predefined type symbols:
  - `[bool]` Booleans defined as `data bool = true | false`.
  - `[int]` and `[real]` for integers and reals, respectively.
  - `[char]` for single characters.

- **(Function) Operator Symbols** (`funopsym`): Infix operators, when used as functions, are written using one or more of the following symbols:

  \[ ! \# \& * \backslash + - . < = > ? @ ^ | \]

  Also, the symbols

  \[ \% \$ : \]

  are allowed except when they are used as the first symbol (i.e., they are in the first position) of the whole name. The following operator identifiers

  \[ i\- :: = .. : * + - / <=== /* ^ := \Rightarrow \]

  are reserved.

- **(Constructor) Operator Symbols** (`conopsym`): Infix operators, when used as constructors, follow the same rules than those for `funopsym`, with the exception that they have to begin with the symbol `[`.
Also, it is allowed to use two different kinds of comments inserted in the program text:

- *Comments in one sentence* (i.e., in one line in a text) starts with the character $\%$ and ends in the same line.

- *Delimited comments* starts with the symbol $\ast$ and ends with $\ast\backslash$. As well, nested comments are allowed and are useful in some contexts, e.g., when one wants to remove a function without deleting its code.
**BASIC GRAMMAR**

program \(\rightarrow\) \((\{\text{topdecl}\})^+\)  

**topdecl**  

- **data** typeLhs \(\equiv\) constrs  
- **type** typeLhs \(\equiv\) type  
- **operdecl**  
- **primitive** prims \(\equiv\) type  
- **decls**  
- **include** string  

**prims**  

- \(\rightarrow\) prim \((\quad\text{prim})^*\)  

**prim**  

- \(\rightarrow\) fun  

**decls**  

- \(\rightarrow\) decl \((\quad\text{decl})^*\)  

**decl**  

- **typedecl**  
- **funrule**  
- **clause**  

**operdecl**  

- \(\rightarrow\) infix integer oplist  
- \(\rightarrow\) infixl integer oplist  
- \(\rightarrow\) infixr integer oplist  

**oplist**  

- \(\rightarrow\) op \((\quad\text{op})^*\)  

**TYPE EXPRESSIONS**

**typeLhs**  

- \(\rightarrow\).typesym \((\quad\text{varsym})^*\)  

**constrs**  

- \(\rightarrow\) constr \((\quad\text{constr})^*\)  

**constr**  

- \(\rightarrow\) type conop type  
- \(\rightarrow\) con \((\quad\text{atype})^*\)  

**type**  

- \(\rightarrow\) ctype \((\quad\text{ctype})^*\)  
- \(\rightarrow\) typesym \((\quad\text{atype})^*\)  

**ctype**  

- \(\rightarrow\) atype  

**atype**  

- \(\rightarrow\) varsym  
- \(\rightarrow\)\(\quad\)type  
- \(\rightarrow\)\(\quad\)\(\quad\)type  

**typedecl**  

- \(\rightarrow\) fun \((\quad\text{fun})^*\) \(\equiv\) type  

After **data** or **type**
FUNCTION RULES AND CLAUSES

funrule → ruleLhs \[=\] exp [conditionrule] \textit{function rule}
ruleLhs → \textit{function rule left-hand side}
| pat funop pat \textit{rule for infix operator}
conditionrule → \textit{condition}
condition → exp (exp)* \textit{conditional expressions}
clause → ruleLhs \[\textit{condition}] \textit{Prolog Clause (\[\textit{-}\] mandatory)}

NAMES

fun → \textit{function name}
| funsym \textit{function operator}
con → \textit{constructor name}
| consym \textit{infix constructor operator}
op → \textit{infix operators}
| funop \textit{infix function}
| conop \textit{infix constructor}
funop → \textit{infix function}
| funopsym \textit{binary operator}
| \textit{fun}′ \textit{function as a binary operator}
conop → \textit{infix constructor}
| conopsym \textit{binary constructor}
| \textit{con}′ \textit{constructor as a infix constructor}

EXPRESSIONS

exp → \textit{expression}
| if exp then exp else exp \textit{if then else expression}
| if exp then exp \textit{if then expression}
| opExp \textit{operator expression}
opExp → \textit{infix operator or constructor}
| opExp op opExp
| pfxExp \textit{prefix expression}
pfxExp → [ ] appExp \textit{prefix expression}
appExp → ( atomic )+ \textit{function application}
atomic → varsym \textit{atomic expression}
| fun \textit{variable}
| \textit{function name}
CHAPTER A. TOY(FD) Grammar

| con              | constructor name |
| integer         | integer number   |
| real            | real number      |
| ( )             | unit             |
| exp             | parenthesised expression |
| ( atomic op )   | left-section     |
| ( op atomic )   | right-section    |
| ( exp ( exp )* )| tuple            |
| list            | list             |

list $\rightarrow$ 
- [ exp ( exp )* ] ]
- [ exp ] list ]

PATTERNS

pat $\rightarrow$
- pat conop pat
- ( apat )+

apat $\rightarrow$
- varsym
- con
- fun
- integer
- real
- ( )
- ( pat op )
- ( op pat )
- ( pat ( exp )* )

listapat $\rightarrow$
- [ apat ( apat )* ] ]
- [ apat ] listpat ]

patterns
constructor operator
application
application pattern
variable
constructor
function
integer number
real number
unit
left-section
right-section
tuple
list
list of patterns
enumerated list
Prolog list
Appendix B

Declaration of Primitives (file basic.toy)

/*** THIS FILE DEFINES THE PREDEFINED FUNCTIONS AND TYPES OF THE SYSTEM ***/

data bool = true | false

% primitive functions source code can be found in 'primitives.pl' where
% its actual type is also declared.
% Type definitions in this file are just informative

% unary integer and real functions
primitive uminus, % unary minus operator
   abs   % absolute value
      :: real -> real

% unary real functions
primitive sqrt,
   ln, exp, % natural logarithm and exponential
   sin, cos, tan, cot,
   asin, acos, atan, acot,
   sinh, cosh, tanh, coth,
   asinh, acosh, atanh, acoth
      :: real -> real

% binary arithmetic operators and functions for reals and integers
primitive (+),(-),(*),min,max :: real -> real -> real

% binary real functions
primitive (/),
   (**), log   % Exponentiation and logarithm
% integer powers
primitive (^) :: real -> int -> real

% binary integer functions
primitive div, mod, gcd :: int -> int -> int

% rounding and truncating functions
primitive round, trunc, floor, ceiling :: real -> int

% integer to real conversion
primitive toReal :: int -> real

% relational operators
primitive (<), (<=), (>), (>=) :: real -> real -> bool

% equality and disequality functions
primitive (==), (/=) :: A -> A -> bool

% infix operator precedences
infix 90 ^, **
infix 80 /
infixl 80 *
infixl 70 +, -

infix 50 <, <=, >, >=
infix 20 ==, /=

% 'if_then_else' and 'if_then' functions are equivalent to the 'sugar syntax'
% functions if .. then .. else and if .. then, but useful as partial functions
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' function is necessary for syntax sections management.
flip :: (A -> B -> C) -> B -> A -> C
flip F X Y = F Y X
Appendix C

Common Functions (file \textit{misc.toy})

\begin{verbatim}
\% FILE: misc.toy
\% A collection of useful functions and type declarations, 
\% many of them taken from Gofer'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 \rightarrow bool \rightarrow bool
\not :: bool \rightarrow bool

\% Parallel and
false 'and' _ = false
_ 'and' false = false
true 'and' true = true

\% Parallel or
true 'or' _ = true
_ 'or' true = true
\end{verbatim}
false \text{ ‘or’ } false = false

\% Sequential and
false \text{ \(\land\) } _ = false
ttrue \text{ \(\land\) } X = X

\% Sequential or
ttrue \text{ \(\lor\) } X = true
false \text{ \(\lor\) } X = X

\% Negation
not true = false
not false = true

\begin{verbatim}
andL, orL, orL' :: [bool] -> bool
andL = foldr (\text{\(\land\)}) true
orL = foldr or false
orL' = foldr (\text{\(\lor\)}) false
\end{verbatim}

\% 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)

\begin{verbatim}
undefined :: A
undefined = if false then undefined
\end{verbatim}

\% (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 /= _

\begin{verbatim}
nf X = Y <=< X==Y
\end{verbatim}

\% (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.

% (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.
hnf X = X <=< X /= _

% (strict F) is the restriction of F to finite, totally defined arguments.  
% Operationally, 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.  
% Operationally, 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 \texttt{foldl} and \texttt{scanl} functions, variants \texttt{foldl1} and
%% \texttt{scanl1} for non-empty lists, and strict variants \texttt{foldl'} \texttt{scanl'} describe
%% common patterns of recursion over lists. Informally:
%%
%% \texttt{foldl} \textit{F} \textit{a} \textit{[x1, x2, ..., xn]} = \textit{F} \ldots (\textit{f} \,(\textit{f} \,\textit{a} \,\textit{x1}) \,\textit{x2})\ldots \textit{xn}
%%
%% \texttt{foldl'} \textit{F} \textit{A} \textit{[X|Xs]} = \text{strict} \,\texttt{foldl} \textit{F} \,(\text{\textit{F} \,\textit{A} \,\textit{X}}) \,\textit{Xs}
%%
%% \texttt{foldr} \textit{F} \textit{a} \textit{Xs} = \texttt{foldl} \,(\textit{flip} \,\textit{f}) \,\textit{a} \,(\text{\textit{reverse} \textit{Xs}}) \quad \text{for finite lists} \textit{Xs}.

\texttt{foldl} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{A}) \rightarrow \textit{A} \rightarrow \texttt{[B]} \rightarrow \textit{A}
\texttt{foldl} \textit{F} \textit{Z} \texttt{[]} = \textit{Z}
\texttt{foldl} \textit{F} \textit{Z} \texttt{[X|Xs]} = \texttt{foldl} \textit{F} \,(\textit{F} \,\textit{Z} \,\textit{X}) \,\textit{Xs}

\texttt{foldl1} :: (\textit{A} \rightarrow \textit{A} \rightarrow \textit{A}) \rightarrow \texttt{[A]} \rightarrow \textit{A}
\texttt{foldl1} \textit{F} \textit{X} \texttt{[X|Xs]} = \texttt{foldl} \textit{F} \,\textit{F} \,\textit{X} \,\textit{Xs}

\texttt{foldl'} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{A}) \rightarrow \textit{A} \rightarrow \texttt{[B]} \rightarrow \textit{A}
\texttt{foldl'} \textit{F} \textit{A} \texttt{[]} = \textit{A}
\texttt{foldl'} \textit{F} \textit{A} \texttt{[X|Xs]} = \text{strict} \,(\texttt{foldl} \textit{F} \,(\textit{F} \,\textit{A} \,\textit{X}) \,\textit{Xs})

\texttt{scanl} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{B}) \rightarrow \textit{B} \rightarrow \texttt{[A]} \rightarrow \texttt{[B]}
\texttt{scanl} \textit{F} \textit{Q} \texttt{[]} = \texttt{[Q]}
\texttt{scanl} \textit{F} \textit{Q} \texttt{[X|Xs]} = \texttt{[Q|scanl} \textit{F} \,(\textit{F} \,\textit{Q} \,\textit{X}) \,\textit{Xs}]

\texttt{scanl1} :: (\textit{A} \rightarrow \textit{A} \rightarrow \textit{A}) \rightarrow \textit{A} \rightarrow \texttt{[A]} \rightarrow \textit{[A]}
\texttt{scanl1} \textit{F} \textit{X} \texttt{[X|Xs]} = \texttt{scanl} \textit{F} \,\textit{X} \,\textit{Xs}

\texttt{scanl'} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{B}) \rightarrow \textit{A} \rightarrow \texttt{[B]} \rightarrow \textit{[A]}
\texttt{scanl'} \textit{F} \textit{Q} \texttt{[]} = \texttt{[Q]}
\texttt{scanl'} \textit{F} \textit{Q} \texttt{[X|Xs]} = \texttt{[Q|strict (scanl} \textit{F} \,(\textit{F} \,\textit{Q} \,\textit{X}) \,\textit{Xs}]

\texttt{foldr} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{B}) \rightarrow \textit{B} \rightarrow \texttt{[A]} \rightarrow \textit{B}
\texttt{foldr} \textit{F} \textit{Z} \texttt{[]} = \textit{Z}
\texttt{foldr} \textit{F} \textit{Z} \texttt{[X|Xs]} = \textit{F} \,\textit{X} \,(\texttt{foldr} \textit{F} \,\textit{Z} \,\textit{Xs})

\texttt{foldr1} :: (\textit{A} \rightarrow \textit{A} \rightarrow \textit{A}) \rightarrow \textit{A} \rightarrow \texttt{[A]} \rightarrow \textit{A}
\texttt{foldr1} \textit{F} \textit{X} \texttt{[]} = \textit{X}
\texttt{foldr1} \textit{F} \textit{X} \texttt{[Y|Xs]} = \textit{F} \,\textit{X} \,(\texttt{foldr1} \textit{F} \,\texttt{[Y|Xs]})

\texttt{scanr} :: (\textit{A} \rightarrow \textit{B} \rightarrow \textit{B}) \rightarrow \textit{B} \rightarrow \texttt{[A]} \rightarrow \textit{[B]}
\texttt{scanr} \textit{F} \textit{Q0} \texttt{[]} = \texttt{[Q0]}

\texttt{foldr} \textit{F} \textit{a} \textit{Xs} = \texttt{foldl} \,(\textit{flip} \,\textit{f}) \,\textit{a} \,(\text{\textit{reverse} \textit{Xs}}) \quad \text{for finite lists} \textit{Xs}.
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 up to 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 \([X|Xs]\) else []

takeUntil :: \((A \rightarrow \text{bool}) \rightarrow [A] \rightarrow [A]\)
takeUntil \( P \) [] = []
takeUntil \( P \) \([X|Xs]\) = if \( P X \) then \([X]\) else \([X|Xs]\)

dropWhile :: \((A \rightarrow \text{bool}) \rightarrow [A] \rightarrow [A]\)
dropWhile \( P \) [] = []
dropWhile \( P \) \([X|Xs]\) = if \( P X \) then dropWhile \( P Xs \) else \([X|Xs]\)

span, break :: \((A \rightarrow \text{bool}) \rightarrow [A] \rightarrow ([A], [A])\)
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 :: \((A \rightarrow B \rightarrow C) \rightarrow [A] \rightarrow [B] \rightarrow [C]\)
zipWith \( Z \) [] \( Bs \) = []
zipWith \( Z \) \([A|As]\) [] = []
zipWith \( Z \) \([A|As]\) \([B|Bs]\) = \([Z A B | zipWith \( Z As Bs\])\)

zip :: \([A] \rightarrow [B] \rightarrow [(A,B)]\)
zip \( Xs \) \( Ys \) = zipWith \( mkpair \) \( Xs \) \( Ys \)
%where
mkpair :: A \rightarrow B \rightarrow (A,B)
mkpair \( X \) \( Y \) = (X,Y)

unzip :: \([(A,B)] \rightarrow ([A],[B])\)
unzip [] = ([], [])
unzip \([X|XsYs]\) = auxForUnzip \( X \) \( Ys \) (unzip \( XsYs\))

auxForUnzip \( X \) \( Ys \) \( (Xs,Ys) = ([X|Xs],[Y|Ys])\)

until :: \((A \rightarrow \text{bool}) \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A\)
until \( P \) \( F \) \( X \) = if \( P X \) then \( X \) else until \( P \) \( F \) \( F X \)

until' :: \((A \rightarrow \text{bool}) \rightarrow (A \rightarrow A) \rightarrow A \rightarrow [A]\)
until' \( P \) \( F \) = (takeUntil \( P \) . (iterate \( F \)))
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

substract :: real -> real -> real  
substract = flip (-)

even, odd :: int -> bool  
even X = (X 'mod' 2) == 0  
odd = not . even

lcm :: int -> int -> int
\[
\text{lcm } X \ Y = \begin{cases}
    0 & \text{if } (X = 0) \lor (Y = 0) \\
    \text{else } \text{abs} \left( (X \div \text{gcd } X \ Y) \times Y \right) & 
\end{cases}
\]

%%%% Standard list processing functions: %% %%% %%% %%% %%% %%%

**head** :: \([A] \rightarrow A\)

\[
\text{head } [X|\_] = X
\]

**last** :: \([A] \rightarrow A\)

\[
\text{last } [X] = X \\
\text{last } [\_,Y|Xs] = \text{last } [Y|Xs]
\]

**tail** :: \([A] \rightarrow [A]\)

\[
\text{tail } [\_|Xs] = Xs
\]

**init** :: \([A] \rightarrow [A]\)

\[
\text{init } [X] = [] \\
\text{init } [X,Y|Xs] = [X|\text{init } [Y|Xs]]
\]

**nub** :: \([A] \rightarrow [A]\) \quad \%\% \text{ remove duplicates from list}

\[
\text{nub } [] = [] \\
\text{nub } [X|Xs] = [X|\text{nub } (\text{filter } (X \neq) Xs)]
\]

**length** :: \([A] \rightarrow \text{int}\)

\[
\text{length } [] = 0 \\
\text{length } [\_|Xs] = 1 + \text{length } Xs
\]

**size** :: \([A] \rightarrow \text{int}\)

\[
\text{size } = \text{length } \cdot \text{nub}
\]

**reverse** :: \([A] \rightarrow [A]\) \quad \%\% \text{ reverse elements of list}

\[
\text{reverse } = \text{foldl } (\text{flip } (:)) []
\]

**member,notMember** :: \(A \rightarrow [A] \rightarrow \text{bool}\)

\[
\text{member } = \text{any’ } \cdot \text{ (=)} \\
\text{notMember } = \text{all’ } \cdot \text{ (/=)}
\]

**concat** :: \([[A]] \rightarrow [A]\) \quad \%\% \text{ concatenate list of lists}

\[
\text{concat } = \text{foldr } (++) []
\]

**transpose** :: \([[A]] \rightarrow [[A]]\) \quad \%\% \text{ 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