FDBG, the CLP(FD) debugger library

Dávid Hanák, Tamás Szeredi

Budapest University of Technology and Economics
{dhanak,tszeredi}@inf.bme.hu

CS^2, Szeged

July 1–4, 2002

1. Introduction

Prolog

- stands for *Programming in Logic*;
- is a declarative, logic programming language.

A Prolog program

- is a set of Horn clauses: *facts* and deduction *rules*;
- is interpreted in order to answer queries (questions) by means of *resolution*.

```
member(H, [H|T]).
member(X, [H|T]) :- member(X, T).
```

```
| ?- member(X, [foo,1]).
X = foo ? ; X = 1 ? ; {no}
```

CLP

- stands for *Constraint Logic Programming*;
- denotes a family of programming languages used for finding values in various domains satisfying a set of relations (*constraints*);
- has several branches: CLP(B), CLP(Q/R), CLP(FD), CHR;
- is usually embedded into a *host language*, like Prolog.

CLP(FD)

- variables are represented by *finite sets of interger values* and
- connected by the constraints propagating changes in their domains;
- solutions can be enumerated by *labeling*;
- constraints can be *global constraints* and *indexicals*.

```
| ?- A in 4..7, B in 0..10, A*2 #= B, labeling([], [A,B]).
A = 4, B = 8 ; A = 5, B = 10 ; {no}
```

SICStus Prolog

- is an implementation of the Prolog language;
- contains full implementation of all the above CLP langauges;
- includes a generic debugger for regular Prolog with a programmable interface.

FDBG

- stands for *Finite Domain deBuGger*;
- enables us to trace CLP(FD) programs;
- uses the *wallpaper trace* technique;
- was written almost entirely in user space;
- shipped with SICStus Prolog from version 3.9.

2. Two simple examples

Loading FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)).
| ?- fdbg_on.
% The clp(fd) debugger is switched on
yes
```

Arithmetic indexicals

```
| ?- fdbg_assign_name(X, x), X #< 5, X #> 3.
<x> #< 5
    x = inf..sup -> inf..4
    Constraint exited.
<x> #> 3
    x = inf..4 -> {4}
    Constraint exited.

X = 4 ? ;
no
```

A built-in global constraint

```
| ?- domain([A,B], 0, 2), exactly(1, [0,A,2], B), B #\= 0.
domain([<fdvar_1>,<fdvar_2>],0,2)
    fdvar_1 = inf..sup -> 0..2
    fdvar_2 = inf..sup -> 0..2
    Constraint exited.
exactly(1, [0,<fdvar_1>,2], <fdvar_2>)
    fdvar_1 = 0..2
    fdvar_2 = 0..2 -> 0..1
<fdvar_2> #\= 0
    fdvar_2 = 0..1 -> {1}
    Constraint exited.
exactly(1, [0,<fdvar_1>,2], 1)
    fdvar_1 = 0..2 -> {1}
    Constraint exited.

A = 1,
B = 1 ? ;
no
```

3. Concepts

Goals

- to be able to follow the narrowing of the domains of FD constraint variables;
- to be informed about the wake-up, exit and effects of (global) constraints, and about the labeling steps and their effects;
- to be able to print terms containing FD variables in a well-readable form.

Terminology

- *CLP(FD) events*
  - a constraint event (when a global constraint is woken)
  - some labeling event (start of labeling, a labeling step or failure of labeling)
- *Visualizer*: a predicate reacting to CLP(FD) events called *before* any changes imposed by the current event can take effect. Two basic types:
  - constraint visualizer
  - labeling visualizer
- *Legend*
  - is a list of variables and the corresponding domains;
  - followed by information about the behaviour of the constraint being examined (exiting, failure, etc.);
  - usually gets printed right after the current constraint.

4. Features

Traceable constraints

- are only the global constraints, indexicals are skipped;
- can be either built-in or user defined;
- after FDBG is loaded, arithmetic constraints are translated into global constraints.

Watching CLP(FD) events

- for each event zero or more visualizers are called;
- these visualizers can be either built-in or user defined.

Tools for writing visualizers. FDBG provides predicates to

- *annotate* terms: replace FD variables by their names;
- print annotated terms in a well-readable form;
- prepare and print a legend.

Term naming. A name can be assigned to a variable or to an arbitary term.

- Each variable in a named term is also assigned a sensible name;
- in some cases names are generated automatically;
- built-in visualizers refer to variables by their names;
- named terms can be queried using their names.

5. Basics

Starting FDBG

- FDBG can be turned on and off any time;
- the following options can be specified when turning FDBG on:
  - trace output can be redirected to a file or a socket to be opened, or to an already opened stream;
  - a set of visualizers may be specified to be called on both constraint and labeling events.

Example 1. Output to file, default built-in visualizer, no labeling trace.

```
| ?- fdbg_on([file('my_log.txt', append), no_labeling_hook]).
% The clp(fd) debugger is switched on
```

Example 2. Output to standard error, user defined and built-in visualizers.

```
| ?- fdbg_on([stream(user_error), constraint_hook(fdbg_show),
              constraint_hook(my_show)]).
% The clp(fd) debugger is switched on
```

6. Built-in visualizers

- `fdbg_show(+`*Constraint*`, +`*Actions*`)`
  A built-in visualizer displaying the current global constraint and the corresponding legend.

```
exactly(1,[<a>,<b>,<c>],2)
    a = 0..2 -> {1}
    b = {0}\/{2}
    c = 0..2 -> {1}
    Constraint exited.
```

- `fdbg_label_show(+`*Event*`, +`*ID*`, +`*Variable*`)`
  A built-in visualizer displaying labeling events.

```
Labeling [13, <c>]: starting in range {0}\/{2}.
Labeling [13, <c>]: dual: <c> = 0
[...]
Labeling [13, <c>]: dual: <c> = 2
[...]
Labeling [13, <c>]: failed.
```

7. Term naming

When naming a term

- the specified name is assigned to the whole term;
- all variables appearing in the term are assigned a dervied name – this name is generated from the specified atom and the selector of the variable;
- names are kept in a global store;
- a separate name store belongs to each toplevel call (the store is *volatile*).

Derived names

$$\text{derived name} = \text{base name} + \text{selector}$$

For example the call `fdbg_assign_name(bar(A, [B, C]), foo)` generates the following names:

name	term	remark
`foo`	`bar(A, [B, C])`	the whole term
`foo_1`	`A`	1^{st} argument of `bar`
`foo_2_1`	`B`	1^{st} element of the 2^{nd} argument of `bar`
`foo_2_2`	`C`	2^{nd} element of the 2^{nd} argument of `bar`

Predicates

- `fdbg_assign_name(+`*Term*`, ?`*Name*`)`
  Assigns name *Name* to term *Term* for the scope of the current toplevel call. If *Name* is a variable, uses an autogenerated name and returns that.
- `fdbg_current_name(?`*Term*`, ?`*Name*`)`
  - recalls a term (variable) from the global store by its name;
  - enumerates every name-term pair in the store.
- `fdbg_get_name(+`*Term*`, -`*Name*`)`
  Returns the name *Name* that is assigned to term *Term*.

8. Magic sequences

```
:- use_module(library(fdbg)).
:- use_module(library(clpfd)).
:- use_module(library(lists)).

magic(N, L) :-
    length(L, N),
    fdbg_assign_name(L, list),
    N1 is N-1,
    domain(L, 0, N1),
    occurrences(L, 0, L),
    labeling([ff], L).

occurrences([], _, _).
occurrences([O|Os], I, List) :-
    exactly(I, List, O),
    J is I+1,
    occurrences(Os, J, List).
```

The `exactly/3` constraint

The global constraint `exactly(`*I*`,`*List*`,`*O*`)` succeeds if *I* occurs in *List* exactly *O* times.

Sample run

```
| ?- magic(4, L).
L = [1,2,1,0] ? ;
L = [2,0,2,0] ? ;
no

| ?- magic(10, L).
L = [6,2,1,0,0,0,1,0,0,0] ? ;
no
```

9. Sample trace

```
| ?- [magic].

| ?- fdbg_on(file('fdbg.log',
                  write)).
% FDBG is switched on
yes
f| ?- magic(4, L).
L = [1,2,1,0] ?
yes
| ?- fdbg_off.
% FDBG is switched off
yes
```

The end of `fdbg.log`

```
exactly(2,[1,2,<list_3>,
              <list_4>],<list_3>)
    list_3 = 1..3
    list_4 = 0..2
exactly(0,[1,2,<list_3>,<list_4>],1)
    list_3 = 1..3
    list_4 = 0..2 -> {0}
    Constraint exited.

exactly(1,[1,2,<list_3>,0],2)
    list_3 = 1..3 -> {1}
    Constraint exited.

exactly(2,[1,2,1,0],1)
    Constraint exited.

exactly(3,[1,2,1,0],0)
    Constraint exited.
```

10. Advanced feature highlights

Fine tuning `fdbg_show/2`

- it is possible to tune the output by writing *hook predicates*;
- change the appearance of variables;
- change the appearance of legend lines.

```
exactly(1,[<a>,2],1)
    a = 0..2 -> {1}
    Constraint exited.
```

```
exactly(1,[<a = 0..2>,2],1)
    a = [0,1,2] -> [1]
    Constraint exited.
```

Writing your own visualizers

- for deeper changes you have to write your own visualizer predicates;
- these can exploit problem specific knowledge;
- e.g., "eight queens" problem, draw the complete board.

Support for writing visualizers

- a set of predicates provided by FDBG;
- *annotation:* replacing variables in a term by a descriptive compound;
- built-in legend printer;
- predicate to simplify action list to prepare a fully customized legend.