Constraint Logic Programming and Mu-Calculus

Serge Colin, Fred Mesnard and Antoine Rauzy

Iremia, Université de la Réunion
15, avenue René Cassin - BP 7151 -
97715 Saint Denis Messag. Cedex 9 France
E-mail : colin@univ-reunion.fr

1 Context

A wide range of problems such as static program analysis and symbolic verification induces the need for an evaluation mechanism of fixed point, and thus, a certain form of quantification over relations. Many specialized tools (MVL [7], MEC [1], ...) exist, mainly based upon the mu-calculus formalism [9]. On the one hand, while generally offering good performances, these tools use their own syntaxes, and are restricted to particular applications. On the other hand, Constraint Logic Programming languages are easy to use, give compact source code, provide friendly development platforms with lots of useful libraries, but often suffer from poor performances and lack of features such as second order quantification and explicit manipulation of relations.

The purpose of the work described in this paper is to take the best of the two approaches, more precisely, to integrate a solver of fixed point equations, based on the μ-calculus, into a well known and open Prolog environment : SICStus Prolog.

This work originate in the Toupie [10] project. Toupie is a constraint language designed on an extension of μ-calculus over symbolic finite domain, which provide full universal and existential quantification and allow the definition of relations as fixed point of equations.

We will first describe the module which has been implemented, then give some results obtained for groundness analysis of logic programs and finally propose promising extensions to this work.

2 The module

We chose to implement this solver of fixed point equations as a SICStus Prolog module. The SICStus Prolog environment provides a lot of libraries (terms, AVL trees, ...). Especially, it has a boolean constraints solver [4], coded as a module, and using the ordered binary decision diagram (OBBD) [2, 3] data structure. Techniques of representation of boolean relations and formulas of μ-calculus using BDDs (and their extensions) have proved to be efficient [6, 10]. It was thus convenient for us to use the BDD primitive of the boolean constraint solver in order to represent our formulas.

This boolean solver has been partially written in C language, compiled and linked with the Prolog executable, and thus offer good performances. We tried to reuse as much as possible of this module. The BDDs are coded as ground terms, and we have a few primitives for manipulating this BDDs [4] : `$bdd_build/4` for creating a BDD, `$bdd_equiv/2` for testing equivalence of two BDDs, `$bdd_negate/2` for the negation, `bdd_univ/4` for applying operator (e.g. or, and) and `$bdd_parts/4` for getting the subcomponents of a BDD.

Our module exports [1] three predicates :

- `mu(Rel,Args,Formula)` create the constraint

  $$Rel = \mu Z.\lambda(Args).Formula$$

  of the μ-calculus, that is to say Rel is the relation defined as the least fixed point of the operator $\lambda(Args).Formula$.

- `nu(Rel,Args,Formula)` create the same constraint, but with the greatest fixed point.

- `fp(Rel,Params,Formula)` compute (evaluate) the value of the relation `Rel`. Params are the effective parameters and `Formula` is the a boolean formula equivalent to the result of the computation.

For instance, one may want to calculate the set of states that are in a dead-lock (analysis of finite states systems). Provided that the relation *edge* (the transition relation) has been defined, one may write this constraint :

```
mu(dead,[S1,S2,S3],
  (T1 v T2 v T3 v (
   fp(edge,[S1,S2,S3,T1,T2,T3]) =< fp(dead,[T1,T2,T3]))),
```

i.e. a state s (coded with three bits) is in the dead-lock if for all state t ('v' is the universal quantifier), if there exists a transition from s to t, then t belongs to the dead-lock.

The way the expressions are evaluated is the following : the formula defining a relation is parsed and the graph of its dependencies is built. Then, at the first call to the relation, the BDD representing the relation is computed recursively, *i.e.* the BDDs corresponding to the sub formulas are computed and the adequate operator is applied to obtain the BDD. The module manages explicit universal quantification (as seen before) and mutually dependencies (via the graph of dependencies).

3 Applications

At this time, our main application is abstract interpretation of logic program over the prop domain [5]. We have compared the computation time of groundness analysis of well-known logic programs using our module with the results obtained using an efficient implementation of the fix point of the T_p operator based on an article of M. Codish [5].

Here are some representative results :

[1] it may evolve, since the module is not achieved yet

Language: Sicstus prolog 3.5 - compiled code
Machine: ULTRA 1 SUN Station
Frequency: 167MHz
RAM: 128Mo
Time unit: ms

Name	#ClP	#VarP	Trans	Scc	#ClB	#VarB	Bu	Mu
qsortapp	15	43	30	0	15	72	60	20
zebra	19	44	30	0	19	81	200	65
money	16	59	30	0	16	96	200	100
grammar	18	44	15	0	18	78	25	0
progeom	20	68	30	10	20	110	110	50
bid	50	111	50	0	50	196	130	50
credit	58	105	50	0	58	218	110	45
warplan	102	242	110	10	109	415	230	90
peephole	135	379	290	10	159	783	690	37
tictactoe	72	515	150	0	74	714	240	50
qplan	150	471	210	15	152	841	1060	3000

Name: name of the program,
#ClP: number of clauses of the initial program,
#VarP: number of variables of the initial program,
Trans: time to translate the program into its boolean version,
Scc: time to build, reduce and sort the graph of calls,
#ClB: number of clauses of the boolean program,
#VarB: number of variables of the boolean program,
Bu: time to compute the boolean model with Bottom-up iteration,
Mu: same with our module.

We have also used this module for analysis of finite state system and the computation of boolean criteria of termination as described in an article of Fred Mesnard [8].

4 Future Works

Wee see three directions to extend this work.

The first and immediate is to improve the module we have implemented. This module was developed as a master thesis project, and lacks of certain features : memory management, cache mechanism, variable ordering, ... We hope to have a "clean" version of this module available at the end of the year via our web : `<http://www.univ-reunion.fr/~gcc/>`

The second and main is to explore the interaction between the relations (seen as predicates) we can define with these "μ-constraints" and the predicates of the logic programs. The dynamic construction (or modification) of predicates induces a lot of questions in terms of semantic and termination. The logical first order variables that appear in a constraint defining a relation could be themselves constrained (integer, rational trees, ...). What would be the effects of mixing these types of constraints ... Moreover, with the ensemblist interpretation of the relations (as sets of tuples), one may want to perform ensemblist operations upon theses relations such as intersection and union. In that sense, "μ-constraints" can be seen as a kind of set constraints.

Last, we think there are other domains of computation of fixed points that could be interesting. For instance, we know we can extend the μ-calculus over finite domains [10]. One can also imagine to compute fixed point over useful domains such as rational languages or databases.

References

[1] A. Arnold. MEC : a System for Constructing and analysing Transition System. In *Automatic verification Methods for Finite States Systems*, volume 407 of *LNCS*. Springer-Verlag, 1989.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 1035–1044, 1986.

[3] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, pages 293–318, 1992.

[4] M. Carlsson. Boolean constraints in sicstus prolog. Technical Report T91:09, Swedish Institute of Computer Science, 1994.

[5] M. Codish and B. Demoen. Analyzing logic programs using prop-ositional logic programs and a magic wand. *Journal of Logic Programming*, pages 249–274, 1995.

[6] K.L. McMillan D.L. Dill J.R. Burch, E.M. Clarke and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, pages 142–170, 1992.

[7] K.L. McMillan. *Symbolic Model Checking : An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[8] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. *Proc. of JICSLP'96*, pages 7–21, 1996.

[9] D. Park. Finiteness is mu-ineffable. In *Theory of Computation Report No. 3*. The University of Warwick, 1974.

[10] A. Rauzy. Toupie : Technical Report. Technical report, LaBRI - CNRS - Universite Bordeaux I, July 92.