CHR for Prototyping Abstract Interpretation

Fred Mesnard and Ulrich Neumerkel*

Iremia, Université de la Réunion
15, avenue René Cassin - BP 7151 -
97 715 Saint Denis Messag. Cedex 9 France
E-mail: fred@univ-reunion.fr URL: www.univ-reunion.fr/~gcc

Abstract

We have implemented with CHR an abstract interpretation for constraint logic programs of the instance CLP($\mathcal{R}$). Following the approach proposed by Hanus to analyze non linear constraints in CLP($\mathcal{R}$), we were able to obtain with CHR an implementation that is very close to the original specification. The CHR are used within a framework in the spirit of Bruynooghe with a common tabulation algorithm that uses abstract operators. Our experience suggests that CHR are an effective tool for rapid prototyping of abstract interpretations.

1 Introduction

CHR have been used to implement various applications [4, 5], their major area being dedicated constraint solvers. We show how CHR can be used to model abstract domains needed for program analysis techniques like abstract interpretation [2, 3]. We implemented an analysis to determine those CLP($\mathcal{R}$) programs that do not produce nonlinear constraints as answers following an approach proposed by Hanus [6]. With the help of CHR the specification of the abstract domain was easily translated directly into a working implementation. Our experience suggests that CHR are a convenient tool to allow rapid development in implementing abstract domains and abstract interpreters.

We organize the paper as follows. After a short motivation for Hanus' analysis we present the abstract domain, its normalization rules and our implementation with CHR. We then discuss aspects concerning the interface of our CHR set with a tabulated interpreter.

2 An abstract domain for nonlinear constraints

Current implementations of CLP($\mathcal{R}$) must face a tradeoff in the design of their constraint solvers when dealing with nonlinear constraints, choosing between completeness and efficiency. Most systems prefer to delay nonlinear constraints until they become linear, therefore accepting conditional answers that may not have any solution at all.

To ameliorate the situation of such implementations, Hanus [6] proposed an analysis to determine those programs where no nonlinear constraint remains delayed. If such a program produces an answer it is ensured that a solution exists. The heart of the analysis proposed by Hanus is based on an abstract domain that consists of two components. The first describes the dependencies between variables; the second represents conditional delays.

Dependencies between variables are represented with $Vs \Rightarrow X$ describing the fact that the set of variables Vs uniquely determine the value X. If all variables in Vs are ground, also X will be ground. Therefore the implication $\{\} \Rightarrow X$ denotes ground values for X. For the constraint $X = Y + Z$ the following dependencies are described: $\{Y, Z\} \Rightarrow X$, $\{X, Y\} \Rightarrow Z$, and $\{X, Z\} \Rightarrow Y$. For multiplication in $X = Y * Z$ only $\{Y, Z\} \Rightarrow X$ can be stated, due to the rôle of zero in multiplication.

*On leave from: Technische Universität Wien, Institut für Computersprachen

The other part models the condition that a nonlinear constraint will become linear: *delay* (X *or* Y) means that X or Y must become ground, whereas *delay* states that there is a nonlinear constraint that cannot be made linear. For the constraint $X = Y * Z$ the abstract element describes the condition for nonlinearity with $delay(Y\ or\ Z)$, i.e. one of Y and Z have to be known to ensure linearity.

These three elements used to describe any abstract substitution can be directly mapped into CHR constraints. We represent variable dependencies by `Vs => X`, where the set `Vs` is represented by a list of variables. Delays are mapped into `delay([X,Y])` and `delay` respectively.

In order to eliminate redundancies in abstractions the following normalization rules are proposed by Hanus. These rules can be almost directly implemented. In fact, the implementation is actually slightly simpler than the specification: there is no need to refer to the remaining set of unrelated elements (denoted by the set A) and simpagation rules prevent the double occurrence of elements on both sides of the rules. The predicates `member_of/2`, `included_in/2` and `not_member_of/2` are defined as usual. `member_of_(X,Ys,Zs)` is true iff `X` is a member of `Ys`, and `Zs` is `Ys` without `X`. `union_of_(V1sV2s,V1s,V2s)` is true iff V1sV2s is the union of V1s and V2s.

Hanus' normalization rules for abstractions

(N1)	$A \cup \{\{\} \Rightarrow Z, V \cup \{Z\} \Rightarrow X\}$	$\longrightarrow$	$A \cup \{\{\} \Rightarrow Z, V \Rightarrow X\}$
(N2)	$A \cup \{\{\} \Rightarrow X, delay(X or Y)\}$	$\longrightarrow$	$A \cup \{\{\} \Rightarrow X\}$
(N3)	$A \cup \{V_1 \Rightarrow X, V_2 \Rightarrow X\}$	$\longrightarrow$	$A \cup \{V_1 \Rightarrow X\}$ if $V_1 \subseteq V_2$
(N4)	$A \cup \{V \cup \{X\} \Rightarrow X\}$	$\longrightarrow$	A
(N5)	$A \cup \{delay(X or Y), delay\}$	$\longrightarrow$	$A \cup \{delay\}$
(N6)	$A \cup \{V_1 \Rightarrow X, V_2 \cup \{X\} \Rightarrow Y\}$	$\longrightarrow$	$A \cup \{V_1 \Rightarrow X, V_2 \cup \{X\} \Rightarrow Y, V_1 \cup V_2 \Rightarrow Y\}$ if $Y \notin V_1$

Implementation in CHR

```
n1@   [] => Z \ Vs => X          <=>  member_of_(Z,Vs,Ws) | Ws => X.
n2@   [] => X \ delay(Vs)        <=>  member_of(X,Vs) | true.
n3@   V1s => X \ V2s => X        <=>  included_in(V1s,V2s) | true.
n4@   Vs => X                    <=>  member_of(X,Vs) | true.
n5@   delay\delay(_)             <=>  true.
n6@   V1s => X, V2sX => Y        ==>
                                 member_of_(X,V2sX,V2s), not_member_of(Y,V1s) |
                                 union_of_(V1sV2s,V1s,V2s), V1sV2s => Y.
```

Let us run this solver. The goal `Z = X*Y, X = 2, Y = 3` is abstracted to `delay([X,Y]), [X,Y] => Z, [] => X, [] => Y` which is normalized to `[]=>X, []=> Y, [] => Z`. For the goal `A*C=X, B*X=Y, C = 3` the abstraction is `delay([A,C]), [A,C] => X, delay([B,X]), [B,X] => Y, [] => C` which normalizes to `delay([B,X]), [] => C, [A] => X, [A,B] => Y, [B,X] => Y`.

3 Plugging the CHR solver into an abstract interpreter

A tabulating interpreter similar to [8] is used as the engine of Bruynooghe's framework for abstract interpretation [1]. Our interpreter, described in details in [7], was designed to meet the specific needs of CLP programs.

The interface of our solver relies mainly on the following predicates. `equals(Var1,Var2,C)` iff `C` denotes the constraint `Var1=Var2`. `conjunction(C1,C2,C3)` iff `C3=C1`$\wedge$`C2`. `satisfiable(C)` iff `C` is satisfiable. `entail(C1,C2)` iff `C1` entails `C2`. `call_restrict(C1,Vars,C2)` iff `C2` is the constraint computed from `C1` and `Vars` when entering a clause. `exit_restrict(C1,Vars,C2)` iff `C2` is the constraint computed from `C1` and `Vars` when exiting a clause. `lub(C1,C2,C)` iff `C3` is the least upper bound of `C1` and `C2`.

Since we represent an abstract substitution as a list of the domain elements `delay`, `delay/1`, `=>/2`, the code for the first three predicates is trivial. `Var1=Var2` is coded as `[Var1] => Var2, [Var2] => Var1`. The conjunction of two constraints is the concatenation of their representations. The predicate `satisfiable/1` is always true, because in our domain any constraint is satisfiable. Based on the description given in Hanus' paper, the last predicates mainly rely on a normalized representation of the constraints.

We present the main example given in [6] running with our implementation. The following CLP($\mathcal{R}$) program (in flat form) describes the product of all elements of a list:

```
prod(A,B) :- A=[],B=1.
prod(A,B) :- A=[E|R],B=E*P,prod(R,P).
```

We precompile it only once for the complete analysis:

```
prod(A,B) :- [] => A,[] => B.
prod(A,B) :- [E,R] => A,[A] => E,[A] => R,[E,P] => B,delay([E,P]),prod(R,P).
```

In order to determine whether there is a delayed non-linear arithmetic constraint in a computed answer for the query: `prod([2,3,4],P)`, we execute the corresponding abstract goal `tabInt( ([] => L, prod(L,P)), Sol)` with our tabulating interpreter which computes the answer: `[] => L, [] => P`. Hence, from the correctness and completeness results of delay abstractions given in [6], one concludes that *no* answer to the original goal contains a delayed non-linear arithmetic constraint, i.e. all non-linear constraints generated in all derivations become linear at run time.

For using `prod/2` the other way round: `prod(L,24)` leads to `delay, [] => P`. Here, the only thing one can conclude is that there *may be* at least one computed answer for the original goal including a delayed non-linear arithmetic constraint.

4 Conclusion

We have presented an application of CHR in the area of abstract interpretation. With the help of CHR the actual implementation of the abstract domain turned out to be very close to its specification. We believe that also other domains can profit from the ease of implementing abstract domains with CHR. In particular it seems that for other domains, the component that describes variable dependencies is reusable. Besides extending our approach to other domains of analysis, another path for further research concerns the tighter integration of CHR and the abstract interpreter, leaving a larger part of the work for CHR.

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming*, 10:91–124, 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *J. Logic Programming*, 13:103–180, 1992.

[4] T. Frühwirth. Introducing simplification rules. *Workshop on Rewriting and Constraints,Dagstuhl, Germany*, 1991.

[5] T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic programming*, 37:95–138, 1998.

[6] M. Hanus. Compile-time analysis of nonlinear constraints in clp(r). *New Generation Computing*, 13(2):155–186, 1995.

[7] F. Mesnard and S. Hoarau. A tabulation algorithm for clp. *Proc. of the 1st International Workshop on Tabling in Logic Programming, Leuven*, 1997. Revised report available at `www.univ-reunion.fr/~gcc`.

[8] H. Tamaki and T. Sato. Old resolution with tabulation. *Proc. of the 3rd ICLP*, pages 84–98, 1986.