Bigre n°81-82 — September 1992 – Workshop on Static Analysis 92 – WSA92

Static Analysis of CLP Programs over Numeric Domains

Roberto Bagnara, Roberto Giacobazzi, Giorgio Levi

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
{bagnara,giaco,levi}@di.unipi.it

Extended Abstract

Constraint logic programming (CLP) is a generalization of the pure logic programming paradigm, having similar model-theoretic, fixpoint and operational semantics [9]. Since the basic operational step in program execution is a test for solvability of constraints in a given algebraic structure, CLP has in addition an algebraic semantics. CLP is then a general paradigm which may be instantiated on various semantic domains, thus achieving a good expressive power. One relevant feature is the distinction between testing for solvability and computing a solution of a given constraint formula. In the logic programming case, this corresponds to the unification process, which tests for solvability by computing a solution (a set of equations in solved form or *most general unifier*). In CLP, the computation of a solution of a constraint is left to a constraint solver, which does not affect the semantic definition of the language. This allows different computational domains, e.g. real arithmetic, to be considered without requiring complicated encodings of data objects as first order terms. Since the fundamental linguistic aspects of CLP can be separated from the details specific to particular constraint systems, it seems natural to parameterize the semantics of CLP languages with respect to the underlying constraint system [16]. For example, considering a domain of "abstract constraints" instead of the "concrete constraints" that are actually manipulated during program execution, we obtain for free a formal treatment of abstract interpretation of CLP programs: this provides a foundation for dataflow analysis and program manipulation of CLP programs.

The original contribution of the present work is the integration of approximate inference techniques, well known in the field of artificial intelligence (AI), with an appropriate framework for the definition of non-standard semantics of CLP. This integration turns out to be particularly appropriate for the considered case of the abstract interpretation of CLP programs over numeric domains. One notable advantage of this approach is that it allows to close the often existing gap between the formalization of data-flow analysis in terms of abstract interpretation and the possibility of efficient implementations. Towards this aim we identified a class of approximate deduction techniques from AI and a semantic framework general enough to accomodate the corresponding approximate constraint systems.

In [7] a simple and powerful "generalized algebraic semantics" for constraint logic programs is presented that is parameterized with respect to the underlying constraint system. Generalized semantics abstract away from standard semantics objects, by focusing on the general properties of any (possibly non-standard) semantics definition. In constraint logic programming, this corresponds to a suitable definition of the constraint system supporting the semantics definition. An algebraic structure is introduced to formalize the constraint system notion, thus making applicable classical mathematical results and both a top-down and bottom-up semantics are considered. Abstract semantics of CLP can then be formally specified by means of the same techniques used to define the concrete one.

The inference technique we borrow from AI is known, in its generality, under the name of *propagation on constraint networks* [6]. A constraint network is a declarative structure expressing relationships among parameters. It consists of a number of *nodes* connected by a number of *constraints*. Nodes represent individual parameters, and are characterized by a value (attribute, property), known or unknown. A constraint represents a relationship between the values of the nodes it connects. The constraint propagation technique consists of deducing information from a small group of nodes and constraints, and *caching* this new information as changes in the network. This changes will be used in the next inference steps to deduce further information. In such way the deducible consequences of each datum in the the network are gradually propagated through the structure. The main advantages of this technique are: generality, incrementality and good performance degradation when time limitations are imposed.

Based on the above general ideas, here we present a couple of abstract interpretations for CLP languages over numeric domains. The reference language for this work was originally CLP($\mathcal{FD}$) [2], where $\mathcal{FD}$ stands for

Finite numeric Domains, but the static analyses we describe (especially the second one) are applicable and turn out to be very useful also for the compile-time optimization of CLP($\Re$) programs.

The objective of the first abstract interpretation is to derive *spatial approximations* of the success set of program predicates. The concrete interpretation of a constraint, that is, a *shape* in k-dimensional space, is abstracted by an enclosing, though not necessarily minimal, *bounding box*. Bounding boxes are rectangular regions with sides parallel to the axes. Thus, a bounding box is univocally identified by its projections (i.e. intervals) over the axes associated to the variables. Of course bounding boxes are very coarse approximations of general shapes. Finer spatial approximations exist and are well known, such as enclosing convex polyhedra, grid and Z-order approximations [8]. However, the coarseness of bounding boxes is well repaid by the relative facility with which they can be derived, manipulated and used to deduce information relevant to static analysis.

In what follows we stick to linear constraints, even though some extensions are possible. There are several techniques for deriving bounding boxes from a given set of constraints: variants of Fourier-Motzkin variable elimination [5, 15], the *sup-inf* method [3, 17] etc. For our study in CLP($\mathcal{FD}$) we considered a variant of the constraint propagation technique called *label inference*. More precisely we use the Waltz algorithm [13, 14] applying *label refinement*, in which the bounding box corresponding to a set of constraints is gradually restricted [6]. For example, if we have the constraint $X + Y = Z$ and the current bounding box is $X \in [3, +\infty]$, $Y \in [1, 1]$, $Z \in [0, 10]$, then we can refine the label for Z obtaining $Z \in [4, 10]$. The Waltz algorithm entails using refinement on each constraint and each variable over and over until refinement produces no more changes. When this stage is reached, the algorithm is said to have *reached quiescence*. In case the range of variables is a finite set, as is our case for CLP($\mathcal{FD}$), the Waltz algorithm will reach quiescence after performing $O(mc)$ label refinements, where m is the maximum cardinality of variable domains and c is the number of constraints.

The technique of representing integer variables by intervals and deducing theorems about these intervals is common in abstract interpretation [4]. Its application to CLP($\mathcal{FD}$) can possibly bring to the reduction of variable domains at compile time (thus reducing the cost of backtracking search inside the constraint solver at run time), and to the generation of code where some choice point creations have been removed. This is quite important since in

CLP($\mathcal{FD}$) it is not enough to save variables addresses when a choice point is encountered, as in Prolog, but it is necessary to record the current variables domains.

This analysis is applicable also to CLP($\Re$). Even though when infinite domains are concerned the Waltz algorithm is not guaranteed to terminate, it does degrade very well when time limitations are imposed. Since refinement is sound deductively so is the Waltz algorithm which simply iterates refinement. The labels derived after each refinement step are thus sound, no matter if quiescence has been reached or not. This amounts to say that the Waltz algorithm can still be used, provided that its termination is externally forced. Finally, as the abstract domain of intervals (and thus of bounding boxes) over $\Re$ does not satisfy the ascending chain condition, a *widening/narrowing* technique can be used to ensure finiteness of the bottom-up program evaluation [4].

A similar technique has been independently developed in [12], where a correct implementation of CLP($\Re$) is based on constraint interval arithmetic. They present an adaptation of CLP($\mathcal{R}$)[1], called ICLP($\mathcal{R}$), where the results of floating-point computations are approximated by means of intervals. The key difference is in the non-standard semantics construction and in the application purposes. In our approach the non-standard semantics is validated in the more formal framework of abstract interpretation. This makes clear the relationships between the concrete (standard) semantics definition (i.e. CLP($\Re$)) and the abstract (non-standard) one (e.g. ICLP($\mathcal{R}$)). Moreover, a possible use of the ICLP($\mathcal{R}$) interpreter as an abstract interpreter for CLP($\Re$) programs requires additional approximation techniques to ensure finiteness and improve the analysis performances: loop checking, widening/narrowing etc.

Our second analysis is based on constraint inference (another variant of constraint propagation). Its aim is to allow for approximate deduction of consistency/inconsistency and entailment of constraints. These approximations turn out to be useful in triggering some important compiler optimizations such as mutual exclusion exploitation, code motion and simplified treatment of the *future redundant* constraints [11]. This last optimization is reported to give a dramatic efficiency improvement for CLP($\Re$) programs.

Let us focus our attention to arithmetic domains, where the constraints

[1] CLP($\mathcal{R}$) denotes the CLP($\Re$) implementation described in [10].

are binary relations over expressions. We abstract them by means of labelled digraphs. Nodes are called *quantities* and are labeled with the corrisponding arithmetic formula (and possibly a variation interval). Arcs are labelled with relation symbols. More precisely, let

$$C = \{(e_{11} \bowtie_1 e_{12}), \ldots, (e_{n1} \bowtie_n e_{n2})\},$$

where the e_{ij} are terms of the constraint language (assume linear expressions for simplicity) and $\bowtie_i \in \{=, \neq, \leq, >, \geq, <\}$, denote a conjuction of constraints. Let $T = \{e_{ij} \mid i = 1, \ldots n,\ j = 1, 2\}$ be the set (modulo syntactical identity) of terms appearing in C, and let $k = |T|$, where $|\,.\,|$ denotes set cardinality. The abstraction of C is $G = (N, l_n, E, l_e)$, where:

$$\begin{array}{rcl} N & = & \{n_1, \ldots, n_k\} \\ l_n & \text{is} & \text{any bijection between } N \text{ and } T \\ E & = & \{(n1, n2) \mid \exists m \in \{1, \ldots n\} \,.\, (l_n(n_1) \bowtie_m l_n(n_2)) \in C\} \\ l_e((n_1, n_2)) & = & \{\bowtie_m \mid (l_n(n_1) \bowtie_m l_n(n_2)) \in C\} \end{array}$$

Disjunctions of constraints are represented by unions of digraphs, while conjuction is handled by connecting digraphs in the obvious way, merging the nodes having equal labels.

Let us consider a digraph representing a conjunction of constraints. We can *enrich* it by either adding new arcs to it or by adding (stronger) relations to the labels of existing arcs (these two operations collapse into just one if you consider that a missing arc is equivalent with one labelled with the always-true relation). Of course, since we are interested in approximate but *sound* deduction, we are interested only in correct enrichments. The addition of an arc (n_1, n_2) with label $\bowtie$ to the digraph abstracting C is correct iff $C \models_{\mathcal{R}} l_n(n_1) \bowtie l_n(n_2)$[2].

Now we must answer the question: how do we infer new constraints at a reasonable computational cost? An answer comes (as before) from AI where approximate deduction holds the spotlight since the origins. In [18, 19] an arithmetic reasoning system, called the *Quantity Lattice*, is described. The Quantity Lattices supports, in a computationally efficient way, various qualitative and quantitative reasoning techniques. All these techniques are integrated, that is, inference made with one technique can trigger further inferences by the other ones. As a result the range of arithmetic inferences

[2]This means $\mathcal{R} \models \big(C \Rightarrow l_n(n_1) \bowtie l_n(n_2)\big)$, where $\mathcal{R}$ is the structure that *interprets* the constraints.

the system is able to perform is quite wide and suitable for our application. Given one of the above mentioned digraphs the Quantity Lattice can apply five different inference techniques:

1. Determining new relationships using graph search.
2. Determining new relationships using numeric constraint propagation.
3. Constraining the value of arithmetic expressions using interval arithmetic.
4. Constraining the value of arithmetic expressions using relational arithmetic.
5. Constraining the value of arithmetic expressions using constant elimination arithmetic.

Previous inference results are *cached* by adding arcs or refining their labels (and possibly by restricting the intervals associated with quantities). In our work we currently use only the inference techniques 1 (but computing the full transitive closure, as in [1]) and 4. The resulting system is still powerful enough to perform useful deductions for our purposes. All the inferences made are recorded along with their justifications (i.e. sets of arcs). This is important, as we will see shortly.

Consider the following example:

$$\begin{array}{lll} mg(T) & :- & T = 1. \\ mg(T) & :- & T > 1, T1 = T - 1 \,\square\, mg(T1). \end{array}$$

The *behaviour* of the above program with respect to T is the same as the *mortgage* well known example.

In the model obtained by our abstract interpretation for the $mg/2$ predicate there are two four–nodes digraphs associated with the second clause (see Figure 1). The nodes (quantities) are labelled with the formulas T, $T-1$, $T1$ and 1. Both the digraphs contain the *textual* arcs (that is arcs coming from the program's text) $T > 1$ and $T1 = T - 1$. The first digraph contains also the textual arc $T1 = 1$, and the *induced* arcs $T > T - 1$ (inferred by technique 4. above), $T > T1$ (inferred by 1. and justified by $\{T > T-1, T1 = T-1\}$), $T - 1 = 1$ (inferred by 1. and justified by $\{T1 = T - 1, T1 = 1\}$). Now by technique 1. we are able to parallel the textual arc $T > 1$ with an induced arc $T > 1$ with justification

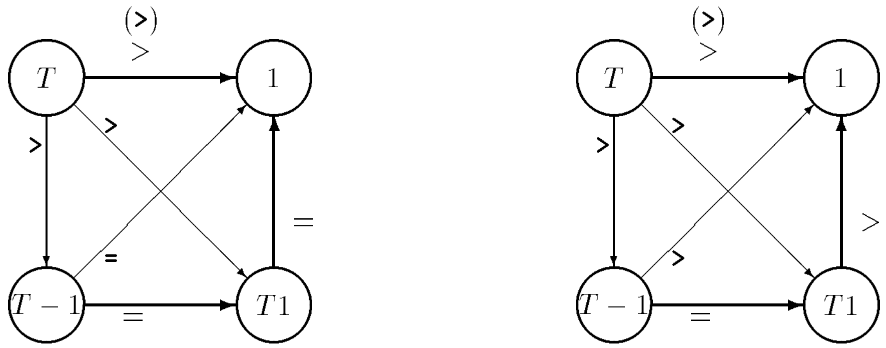

Figure 1: Portions of the two digraphs for the second clause of $mg/2$. Textual arcs have thick lines and are labelled with $=$ and $>$. Induced arcs have thin lines and are labelled with = and >.

$\{T > T-1, T-1 = 1\}$. Similarly, considering the second digraph, we end up with an induced arc $T > 1$ with justification $\{T > T-1, T-1 > 1\}$. Since in both cases the induced arc $T > 1$ does not have the textual arc $T > 1$ into its justification, the textual arc $T > 1$ is future redundant [11]. In general a textual arc (constraint) is future redundant if it can be doubled by an induced arc labelled with an equal or stronger relation and if this induced arc does not have the textual arc into its justification.

References

[1] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. of the ACM*, 26(11):832–843, 1983.

[2] R. Bagnara. *Interpretazione Astratta di Linguaggi Logici con Vincoli su Domini Finiti.* M.Sc. thesis, Università di Pisa, July 1992.

[3] W. W. Bledsoe. The Sup-Inf Method in Presburger Arithmetic. Memo ATP-18, Math. Dept., University of Texas at Austin, Austin, 1974.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[5] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[6] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32:281–331, 1987.

[7] R. Giacobazzi, S. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.

[8] R. Helm, K. Marriot, and M. Odersky. Spatial Query Optimization: from Boolean Constraints to Range Queries. Technical Report RC 17231, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, 1991.

[9] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[10] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 1991. To appear.

[11] N. Jørgensen, K. Marriot, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\Re$). In *Proc. 1991 Int'l Symposium on Logic Programming*, pages 420–434, 1991.

[12] J. H. M. Lee and M. H. van Emden. Adapting CLP($\mathcal{R}$) to Floating-Point Arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 996–1003, 1992.

[13] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[14] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[15] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing '91*, 1991. To appear in Commun. of the ACM.

[16] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM, 1991.

[17] R. Shostak. On The Sup-Inf Method in for Proving Presburger Formulas. *Journal of the ACM*, 24(4):529–543, 1977.

[18] R. Simmons. Representing and Reasoning About Change in Geologic Interpretation. Technical Report 749, MIT AI Laboratory, 1983.

[19] R. Simmons. Commonsense Arithmetic Reasoning. In *Proceedings AAAI-86*, pages 118–124, 1986.