Convex Hull Abstraction in Specialisation of CLP Programs

Peralta, J.C.; Gallagher, John Patrick

Published in:
Lecture Notes in Computer Science

Publication date:
2003

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Peralta, J. C., & Gallagher, J. P. (2003). Convex Hull Abstraction in Specialisation of CLP Programs. *Lecture Notes in Computer Science*, (2664), 90-108.

Convex Hull Abstractions in Specialization of CLP Programs

Julio C. Peralta[1⋆] and John P. Gallagher[2⋆⋆]
jperalta@leibniz.iimas.unam.mx, jpg@ruc.dk

[1] Instituto de Investigación en Matemáticas Aplicadas y en Sistemas
Circuito Escolar s/n, Ciudad Universitaria, México, D.F.
[2] Dept. of Computer Science, Building 42.1, University of Roskilde
P.O. Box 260, DK-4000 Roskilde, Denmark

Abstract. We introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls). This domain is used in an algorithm for specialization of constraint logic programs. The algorithm incorporates in a single phase both top-down goal directed propagation and bottom-up answer propagation, and uses a widening on the convex hull domain to ensure termination. We give examples to show the precision gained by this approach over other methods in the literature for specializing constraint logic programs. The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. Assignments, inequalities and equalities with arithmetic expressions can be interpreted as constraints during specialization, thus increasing the amount of specialization that can be achieved.

1 Introduction

Program specialization is sometimes regarded as being achieved in three phases: *pre-processing of the program, analysis* and *program generation*. During preprocessing the input program may be subject to some minor syntactic analyses or changes, ready for the analysis phase. The analysis computes some data-flow and control-flow information from the program and the specialization query. Finally, at program generation time the result of the analysis is used to produce a residual program reflecting the result of the analysis. In off-line specialization the three phases are consecutive, whereas in on-line specialization and driving the analysis and program generation phases are merged or interleaved.

The use of abstract interpretation techniques to assist program specialization is well-established [9, 8, 16, 17, 24] and goes back to the invention of binding time analysis to compute static-dynamic annotations [15]. More complex and expressive abstract domains have been used such as regular tree structures [22, 10, 12, 18].

⋆ Supported in part by CONACYT Project I39201-A
⋆⋆ Partially supported by European Framework 5 Project ASAP (IST-2001-38059)

In this paper we focus on an abstract domain based on arithmetic constraints. Abstract interpretations based on arithmetic constraints have already been developed [4, 2, 26]. We show how the analysis phase of a specialization algorithm can benefit from advances made in that field. We introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls [4]). The operations on this domain are developed from a standard constraint solver.

We then employ this domain within a generic algorithm for specialization of (constraint) logic programs [12]. The algorithm combines analysis over an abstract domain with partial evaluation. Its distinguishing feature is the analysis of the success constraints (or answer constraints) as well as the call constraints in a computation. This allows us to go beyond the capability of another recent approach to use a linear constraint domain in constraint logic program specialization [6].

The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. We can reason in constraint terms about the arithmetic expressions that occur in logic programs, treating them as constraints (for instance `X is Expr` is treated as {`X = Expr`}). In addition, the algorithm provides a contribution to the growing field of using specialization for model checking infinite state systems [19].

In this paper a constraint domain based on linear arithmetic equalities and inequalities is reviewed (Section 2). The structure of the specialization algorithm is presented (Section 3), along with examples illustrating key aspects. Next, in Section 4 more examples of specialization using the domain of linear arithmetic constraints are given. Then, comparisons with related work are provided in Section 5. Finally, in the last section (Section 6) some final remarks and pointers for future work are considered.

2 A Constraint Domain

Approximation in program analysis is ubiquitous, and so is the concept of a domain of properties. The analysis phase of program specialization is no exception.

2.1 Linear Arithmetic Constraints

Our constraint domain will be based on linear arithmetic constraints, that is, conjunctions of equalities and inequalities between linear arithmetic expressions. The special constraints true and false are also included. This domain has been used in the design of analysers and for model checking infinite state systems. Here we use it for specialization of (constraint) logic programs.

Let Lin be the theory of linear constraints over the real numbers. Let C and D be two linear constraints. We write $C \sqsubseteq D$ iff $\mathsf{Lin} \models \forall(C \rightarrow D)$. C and D are *equivalent*, written $C \equiv D$, iff $C \sqsubseteq D$ and $D \sqsubseteq C$. Let C be a constraint and V be a set of variables. Then $\mathsf{project}_V(C)$ is the projection of constraint C onto the variables V; the defining property of projection is that

$\mathsf{Lin} \models \forall V(\exists V'.C \leftrightarrow \mathsf{project}_V(C))$, where $V' = \mathsf{vars}(C) \setminus V$. Given an expression e let us denote $\mathsf{vars}(e)$ as the set of variables occurring in e. If $\mathsf{vars}(e) = V$, we sometimes refer to $\mathsf{project}_e(C)$ rather than $\mathsf{project}_V(C)$ when speaking of the projection of C onto the variables of e.

Arithmetic constraints can be presented in their simplified form, removing redundant constraints. Constraint simplification serves as a satisfiability check: the result of simplifying a constraint is false if and only if the constraint is unsatisfiable. If a constraint C is satisfiable, we write $\mathsf{sat}(C)$. Because we used the CLP facilities of SICStus Prolog all these operations (projection, simplification and checking for equivalence) are provided for the domain of linear constraints over rationals and reals. We refer the interested reader to a survey on CLP [14] for a thorough discussion on the subject.

Intuitively, a constraint represents a convex polyhedron in cartesian space, namely the set of points that satisfy the constraint. Let S be a set of linear arithmetic constraints. The *convex hull* of S, written $\mathsf{convhull}(S)$, is the least constraint (with respect to the $\sqsubseteq$ ordering on constraints) such that $\forall S_i \in S.S_i \sqsubseteq \mathsf{convhull}(S)$. So $\mathsf{convhull}(S)$ is the smallest polyhedron that encloses all members of S. Further details and algorithms for computing the convex hull can be found in the literature [4].

2.2 Constrained Atoms and Conjunctions

Now we must define our abstract domain. It consists of equivalence classes of *c-atoms*, which are constrained atoms. Each c-atom is composed of two parts, an atom and a linear arithmetic constraint.

Definition 1 (c-atoms and c-conjunctions). *A c-conjunction is a pair $\langle B, C\rangle$; B denotes a conjunction of atomic formulas (atoms) and C a conjunction of arithmetic constraints, where $\mathsf{vars}(C) \subseteq \mathsf{vars}(B)$. If B consists of a single atom the pair is called a c-atom.*

(Note that c-conjunctions are defined as well as c-atoms, since they occur in our algorithm. However, the domain is constructed only of c-atoms).

Given any arithmetic constraint C and atom A, we can form a c-atom $\langle A, C'\rangle$, where $C' = \mathsf{project}_A(C)$. Any atom A can be converted to a c-atom $\langle A', C\rangle$ by replacing each non-variable arithmetic expression occurring in A by a fresh variable[3], obtaining A'. Those expressions which were replaced together with the variables that replace them are added as equality constraints to the constraint part C of the c-atom. For example, the c-atom obtained from $p(f(3), Y + 1)$ is $\langle p(f(X_1), X_2), (X_1 = 3, X_2 = Y + 1)\rangle$.

A c-atom represents a set of concrete atoms. We define the *concretization* function γ as follows.

[3] By parsing the arguments the desired terms can be selected.

Definition 2 (γ).
Let $\mathcal{A} = \langle A, C\rangle$ be a c-atom. Define the concretization function γ as follows.

$$\gamma(\mathcal{A}) = \left\{ A\theta \;\middle|\; \begin{array}{l}\theta \text{ is a substitution } \wedge \\ \forall\varphi.\mathsf{sat}(C\theta\varphi)\end{array} \right\}$$

γ is extended to sets of c-atoms: $\gamma(S) = \bigcup\{\gamma(\mathcal{A}) \mid \mathcal{A} \in S\}$.

There is a partial order on c-atoms defined by $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if and only if $\gamma(\mathcal{A}_1) \subseteq \gamma(\mathcal{A}_2)$. Two c-atoms $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent, written $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ and $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$. Equivalence can also be checked using syntactic comparison of the atoms combined with constraint solving, using the following lemma.

Lemma 1. *Let $\mathcal{A}_1 = \langle A_1, C_1\rangle$ and $\mathcal{A}_2 = \langle A_2, C_2\rangle$ be two c-atoms. Let $\langle \bar{A}_1, \bar{C}_1\rangle$ and $\langle \bar{A}_2, \bar{C}_2\rangle$ be the c-atoms obtained by removing repeated variables from A_1 and A_2 and adding constraints to C_1 and C_2 in the following manner. If a variable X occurs more than once in the atom, then one occurrence is replaced by a fresh variable W and the constraint $X = W$ is added to the corresponding constraint part.*

Then $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if there is a renaming substitution θ such that $\bar{A}_1\theta = \bar{A}_2$ and $\bar{C}_1\theta \equiv \bar{C}_2$.

Now we are in a position to define the domain and the operations on the elements of our domain. The relation $\equiv$ on c-atoms is an equivalence relation. The abstract domain consists of equivalence classes of c-atoms. For practical purposes we consider the domain as consisting of *canonical* constrained atoms, which are standard representative c-atoms, one for each equivalence class. These are obtained by renaming variables using a fixed set of variables, and representing the constraint part in a standard form. Hence we speak of the domain operations as being on c-atoms, whereas technically they are operations on equivalence classes of c-atoms.

Next we define the upper bound of c-atoms which combines the *most specific generalization operator (msg)* [25] on terms and the *convex hull* [4] on arithmetic constraints. The idea is to compute the *msg* of the atoms, and then to rename the constraint parts suitably, relating the variables in the original constraints to those in the *msg*, before applying the convex hull operation.

The following notation is used in the definition. Let θ be a substitution whose range only contains variables; the domain and range of θ are $\mathsf{dom}(\theta)$ and $\mathsf{ran}(\theta)$ respectively. $\mathsf{alias}(\theta)$ is the conjunction of equalities $X = Y$ such that there exist bindings X/Z and Y/Z in θ, for some variables X, Y and Z. Let $\bar{\theta}$ be any substitution such that $\mathsf{dom}(\bar{\theta}) = \mathsf{ran}(\theta)$ and $X\bar{\theta}\theta = X$ for all $X \in \mathsf{ran}(\theta)$. (That is, $\bar{\theta} = \varphi^{-1}$ where φ is some bijective subset of θ with the same range as θ).

The following definition is a reformulation of the corresponding definition given previously [26].

Definition 3 (Upper bound of c-atoms, $\sqcup$). *Let $\mathcal{A}_1 = \langle A_1, C_1\rangle$ and $\mathcal{A}_2 = \langle A_2, C_2\rangle$ be c-atoms. Their upper bound $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is c-atom $\mathcal{A}_3 = \langle A_3, C_3\rangle$ defined as follows.*

1. $A_3 = \mathsf{msg}(A_1, A_2)$, *where* $\mathsf{vars}(A_3)$ *is disjoint from* $\mathsf{vars}(A_1) \cup \mathsf{vars}(A_2)$.
2. *Let* $\theta_i = \{X/U \mid X/U \in \mathsf{mgu}(A_i, A_3), U$ is a variable$\}$, *for* $i = 1, 2$. *Then* $C_3 = \mathsf{project}_{A_3}(\mathsf{convhull}(\{\mathsf{alias}(\theta_i) \cup C_i\bar{\theta}_i \mid i = 1, 2\}))$.

$\sqcup$ *is commutative and associative, and we can thus denote by* $\sqcup(S)$ *the upper bound of the elements of a set of c-atoms* S.

Example 1. Let $\mathcal{A}_1 = \langle p(X, X), X > 0\rangle$ and $A_2 = \langle p(U, V), -U = V\rangle$. Then $\mathcal{A}_1 \sqcup \mathcal{A}_2 = \langle p(Z_1, Z_2), Z_2 \geq -Z_1\rangle$. Here, $\mathsf{mgu}(p(X, X), p(U, V)) = p(Z_1, Z_2)$, $\theta_1 = \{Z_1/X, Z_2/X\}$, $\theta_2 = \{Z_1/U, Z_2/V\}$, $\mathsf{alias}(\theta_1) = \{Z_1 = Z_2\}$, $\mathsf{alias}(\theta_2) = \emptyset$, $\bar{\theta}_1 = \{X/Z_1\}$ (or $\{X/Z_2\}$) and $\bar{\theta}_2 = \{U/Z_1, V/Z_2\}$. Hence we compute the convex hull of the set $\{(Z_1 = Z_2, Z_1 > 0), (-Z_1 = Z_2)\}$, which is $Z_2 \geq -Z_1$.

Like most analysis algorithms, our approach computes a monotonically increasing sequence of abstract descriptions, terminating when the sequence stabilizes at a fixed point. Because infinite ascending chains may arise during specialization it is not enough to have an upper bound operator, in order to reach a fixpoint. An operator called *widening* may be interleaved with the upper bound to accelerate the convergence to a fixpoint and ensure termination of an analysis based on this domain. When widening we assume that the c-atoms can be renamed so that their atomic parts are identical, and the widening is defined solely in terms of widening of arithmetic constraints, ∇_c [4]. This is justified since there are no infinite ascending chains of atoms with strictly increasing generality. Hence the atom part of the c-atoms does not require widening.

Definition 4 (Widening of c-atoms, ∇). *Given two c-atoms* $\mathcal{A}_1 = \langle A_1, C_1\rangle$ *and* $\mathcal{A}_2 = \langle A_2, C_2\rangle$, *where* A_1 *and* A_2 *are variants, say* $A_2 = A_1\rho$. *The widening of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, *denoted as* $\mathcal{A}_1 \nabla \mathcal{A}_2$ *is c-atom* $\mathcal{A}_3 = \langle A_2, C_3\rangle$ *where* $C_3 = C_1\rho \ \nabla_c \ C_2$.

For instance, the widening of $\langle p(X), X \geq 0, X \leq 1\rangle$ and $\langle p(Y), Y \geq 0, Y \leq 2\rangle$ is $\langle p(Y), Y \geq 0\rangle$.

3 An Algorithm for Specialization with Constraints

In this section we describe an algorithm for specialization, incorporating operations on the domain of convex hulls. The algorithm is based on one presented previously [12], where we used a domain of regular trees in place of convex hulls, and the operations named ω, calls and answers are taken from there. The operations ω and aunf^* (which is used in the definition of calls) were taken from Leuschel's top-down abstract specialization framework [17]. The answer propagation aspects of our algorithm are different from Leuschel's answer propagation method, though. There is no counterpart of the answers operation in Leuschel's framework. The differences between the approaches were discussed in our previous work [12].

The structure of the algorithm given in Figure 1 is independent of any particular domain of descriptions such as regular types or convex hulls. The operations

concerning convex hulls appear only within the domain-specific operations calls, ω, ∇ and $\mathsf{answers}$.

INPUT: a program P and a c-atom $\mathcal{A}$
OUTPUT: two sets of c-atoms (calls and answers)

begin
  $S_0 := \{\mathcal{A}\}$
  $T_0 := \{\}$
  $i := 0$
  repeat
    $S_{i+1} := \omega(\mathsf{calls}(S_i, T_i), S_i)$
    $T_{i+1} := T_i \nabla \mathsf{answers}(S_i, T_i)$
    $i := i + 1$
  until $S_i = S_{i-1}$ and $T_i = T_{i-1}$
end

Fig. 1. Partial Evaluation Algorithm with Answer Propagation

3.1 Generation of Calls and Answers

The idea of the algorithm is to accumulate two sets of c-atoms. One set represents the set of *calls* that arise during the computation of the given initial c-atom $\mathcal{A}$. The other set represents the set of *answers* for calls.

At the start, the set of calls S_0 contains only the initial goal c-atom, and the set of answers T_0 is empty. Each iteration of the algorithm extends the current sets S_i and T_i of calls and answers. The diagram in Figure 2 illustrates the process of extending the sets. All existing calls $\mathcal{A} = \langle A, C\rangle \in S_i$ are unfolded according to some unfolding rule. This yields a number of *resultants* of the form $(A, C)\theta \leftarrow B_1, \ldots, B_l, C'$, where $A\theta \leftarrow B_1, \ldots, B_l$ is a result of unfolding A and C' is the accumulated constraint; that is, C' is the conjunction of $C\theta$ and the other constraints introduced during unfolding. If $\mathsf{sat}(C')$ is false then the resultant is discarded. The unfolding process is performed in the algorithm by the operation aunf^*, defined as follows.

Definition 5 ($\mathsf{aunf}, \mathsf{aunf}^*$). *Let P be a definite constraint program and $\mathcal{A} = \langle A, C\rangle$ a c-atom. Let $\{A\theta_1 \leftarrow L_1, C_1, \ldots, A\theta_n \leftarrow L_n, C_n\}$ be some partial evaluation [20] of A in P, where $C_i, L_i (1 \le i \le n)$ are the constraint and non-constraint parts respectively of each resultant body. Then define*

$$\mathsf{aunf}(\mathcal{A}) = \left\{\ A\theta_i \leftarrow L_i, (C_i \wedge C\theta_i) \,\middle|\, 1 \le i \le n, \mathsf{sat}(C_i \wedge C\theta_i)\ \right\}.$$

Let S be a set of c-atoms. We define $\mathsf{aunf}^*(S)$ as:

$$\mathsf{aunf}^*(S) = \left\{\ (L, project_L(C')) \,\middle|\, \begin{array}{l}\langle A, C\rangle \in S \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A})\end{array}\ \right\}$$

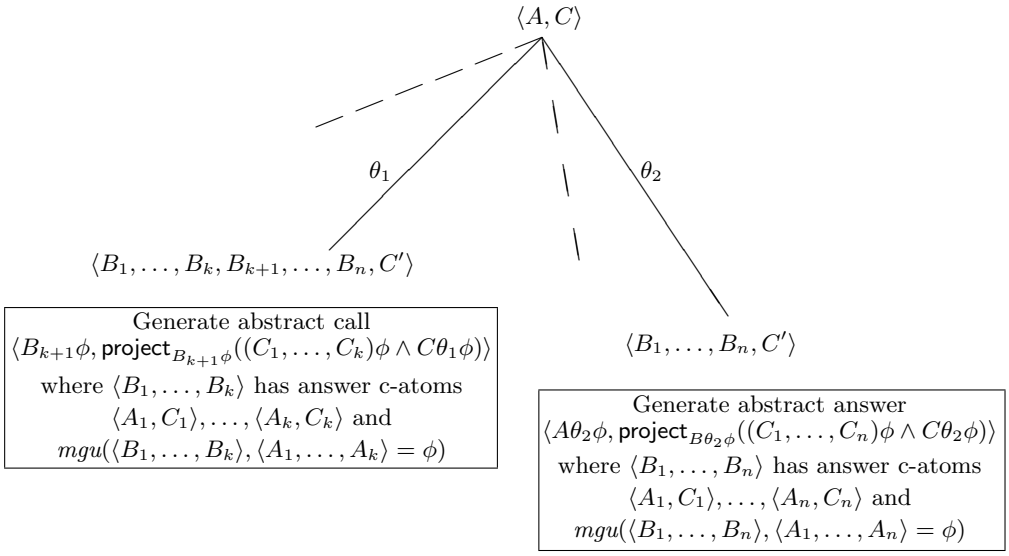

Fig. 2. The generation of calls and answers

In the following examples, assume that the unfolding rule selects the leftmost atom provided that it matches at most one clause (after discarding matches that result in an unsatisfiable constraint), otherwise selects no atom.

Example 2. Consider the following simple program P.

```
s(X,Y,Z) <-
  p(X,Y,Z), q(X,Y,Z)
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,Y1,Z1)
```

```
q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

Let S be $\{\langle \mathtt{s(X,Y,Z)}, \mathtt{X} > 2\rangle\}$. Then $\mathsf{aunf}^*(S) = \{(\mathtt{p(X1,Y,Z1)}, \mathtt{q(X,Y,Z)}, (\mathtt{X} > 2, \mathtt{X} = \mathtt{X1} + 3, \mathtt{Z} = \mathtt{Z1} + 3)\}$. The unfolding rule results in four steps: the unfolding of the atom `s(X,Y,Z)` followed by three unfoldings of `p`, since the initial constraint $\mathtt{X} > 2$ implies that the base case `p(0,0,0)` cannot be matched so long as the first argument of `p` is greater than zero.

Note that the range of the function aunf^* is the set of c-conjunctions. The current answers from T_i are then applied, from left to right, to the c-conjunctions generated by aunf^*. If there is some prefix $B_1 \ldots, B_k$ $(k < l)$ in a c-conjunction, having a solution in T_i, then a call to an instance of B_{k+1} is generated. More precisely, we define a function calls as follows. We first define the notion of a "solution" of a conjunction with respect to a set of c-atoms.

Definition 6 (solution of a conjunction). *Let $(B_1, \ldots, B_l)$ be a conjunction of atoms and T be a set of c-atoms. Then $\langle \varphi, \bar{C}\rangle$ is a solution for $(B_1, \ldots, B_l)$*

in T if there is a sequence of c-atoms $\langle \mathcal{A}_1, \ldots, \mathcal{A}_l \rangle$ where $\mathcal{A}_j = \langle A_j, C_j \rangle \in T$, $1 \leq j \leq l$, such that $\mathsf{mgu}((B_1, \ldots, B_l), (A_1, \ldots, A_l)) = \varphi$, *and* $\mathsf{sat}(\bar{C})$ *(where* $\bar{C} = (C_1 \wedge \cdots \wedge C_l)\varphi$*).*

Definition 7 (calls). *Let S_i be a set of call c-atoms and T_i be a set of answer c-atoms. Define* $\mathsf{calls}(S_i, T_i)$ *to be the set of c-atoms* $\langle B_{k+1}\varphi, \mathsf{project}_{B_{k+1}\varphi}(\bar{C} \wedge C'\varphi) \rangle$ *where*

1. $\langle B_1, \ldots, B_l, C' \rangle \in \mathsf{aunf}^*(S_i)$, *and*
2. *there is a conjunction $(B_1, \ldots, B_k)$ $(k < l)$ which has a solution $\langle \varphi, \bar{C} \rangle$ in T_i, and* $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 3. Let P be the program from Example 2 and let S be $\{\langle \mathtt{s(X,Y,Z)}, \mathtt{X} > \mathtt{2} \rangle\}$. Let $T = \{\langle \mathtt{p(X1,Y1,Z1)}, \mathtt{X1} = \mathtt{0}, \mathtt{Y1} = \mathtt{0}, \mathtt{Z1} = \mathtt{0} \rangle\}$. Then $\mathsf{calls}(S, T) = \{\langle \mathtt{p(X1,Y,Z1)}, \mathsf{true} \rangle, \langle \mathtt{q(X,Y,Z)}, \mathtt{X} = \mathtt{3}, \mathtt{Y} = \mathtt{3}, \mathtt{Z} = \mathtt{3} \rangle\}$. Note that the call to `q` arises from applying the solution for `p` and simplifying the accumulated constraints.

An answer is derived by finding a resultant $A\theta \leftarrow B_1, \ldots, B_k, C'$ whose body has a solution in the current set of answers. The function answers is defined as follows.

Definition 8 (answers). *Let S_i be a set of call c-atoms and T_i be a set of c-atoms. Define* $\mathsf{answers}(S_i, T_i)$ *to be the set of answer c-atoms* $\langle A\theta\varphi, \mathsf{project}_{A\theta\varphi}(\bar{C} \wedge C'\varphi) \rangle$ *where*

1. $\mathcal{A} = \langle A, C \rangle \in S_i$, *and*
2. $A\theta \leftarrow B_1, \ldots, B_l, C' \in \mathsf{aunf}(\mathcal{A})$, *and*
3. $(B_1, \ldots, B_l)$ *has a solution* $\langle \varphi, \bar{C} \rangle$ *in* T_i, *and* $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 4. Let P be the program from Example 2 and let S be $\{\langle \mathtt{p(X,Y,Z)}, \mathsf{true} \rangle\}$. Let $T = \{\langle \mathtt{p(X1,Y1,Z1)}, \mathtt{X1} = \mathtt{0}, \mathtt{Y1} = \mathtt{0}, \mathtt{Z1} = \mathtt{0} \rangle\}$. Then $\mathsf{answers}(S, T) = \{\langle \mathtt{p(X,Y,Z)}, \mathtt{X} = \mathtt{1}, \mathtt{Y} = \mathtt{1}, \mathtt{Z} = \mathtt{1} \rangle\}$.

An important feature of the algorithm is that no call to a body atom is generated until the conjunction of atoms to its left has an answer. One effect of this is to increase specialization because the propagation of answers for some atom restricts the calls to its right. Secondly, answers can only be generated for called atoms, and no answer to an atom is generated until there is an answer to the whole body of some resultant for that atom. There can exist abstract calls that have no corresponding answers; these represent concrete calls that either fail or loop. In fact, infinitely failed computations are not distinguished from finitely failed computations, with the result that programs that produce infinitely failing computations can be specialized to ones that fail finitely. The examples later in this section illustrate this point.

3.2 Approximation Using Convex Hulls and Widening

Call and answer c-atoms derived using the calls and answers functions are added to the sets S_i and T_i respectively. There is usually an infinite number of c-atoms that can be generated in this way. The purpose of the ω and ∇ functions in the algorithm is to force termination. The ω function computes a safe approximation of the calls and answers, using the *convex hull* and *widening* operations, both of which are standard in analyses based on linear arithmetic constraints.

On each iteration, the sets of call c-atoms are partitioned into sets of "similar" c-atoms. The notion of "similar" is heuristic: the only requirements are that the definition of similarity should yield a finite partition, and that all c-atoms in one subset should have the same predicate name. In our implementation we partitioned based on the *trace terms* or "unfolding patterns" of the c-atoms [11]. We assume a function that partitions a set S of c-atoms into a finite set $\{S_1, \ldots, S_m\}$ of disjoint subsets of S, and computes the upper bound of each subset. The function $\mathsf{partition}(S)$ is defined as $\mathsf{partition}(S) = \{\sqcup(S_1), \ldots, \sqcup(S_m)\}$. It is desirable though not essential that $\sqcup(S)$ belongs to the same set as S.

Even if the partition is finite, a widening is required to enforce termination. The widening step is defined between the sets of c-atoms on two successive iterations of the algorithm. Let S, S' be two sets of c-atoms, where we assume that both S and S' are the result of a partition operation. Define $S' \nabla S$ to be

$$\left\{ \mathcal{A}' \nabla \mathcal{A} \,\middle|\, \begin{array}{l} \mathcal{A}' \in S', \mathcal{A} \in S, \\ \mathcal{A}', \mathcal{A} \text{ are in the same set} \end{array} \right\}$$

$$\bigcup$$

$$\left\{ \mathcal{A} \,\middle|\, \begin{array}{l} \mathcal{A} \in S, \\ \nexists \mathcal{A}' \in S' \text{ in the same set as } \mathcal{A} \end{array} \right\}$$

Finally the operation ω can be defined as $\omega(S, S') = S' \nabla \mathsf{partition}(S)$. This ensures termination if the number of sets returned by partition is bounded. The definition states that each element $\mathcal{A}$ of S is replaced by the result of widening $\mathcal{A}$ with the element from S' from the same set, if such an element exists.

3.3 Generation of the Specialized Program

After termination of the algorithm, the specialized program is produced from the final sets of calls and answers S and T respectively. It consists of the following set of clauses.

$$\left\{ \mathsf{rename}(A\theta\varphi \leftarrow L\varphi, C'\varphi) \,\middle|\, \begin{array}{l} \mathcal{A} = \langle A, C \rangle \in S, \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A}), \\ L \text{ has solution } \langle \varphi, \bar{C} \rangle \text{ in } T, \\ \mathsf{sat}(\bar{C} \wedge C'\varphi) \end{array} \right\}$$

That is, each of the calls is unfolded, and the answers are applied to the bodies of the resultants. Note that we do not add the solution constraints $\bar{C}$ to the generated clause, so as not to introduce redundant constraints. The rename function

is a standard renaming to ensure independence of different specialized versions of the same predicate, as used in most logic program specialization systems (see for example [8] for a description of the technique).

Example 5. Consider again the example from Example 2. We specialize this program with respect to the c-atom $\langle \mathtt{s(X,Y,Z)}, \mathsf{true} \rangle$ assuming the usual left-to-right computation rule. Note that the concrete goal `s(X,Y,Z)` does not have any solutions, although with the standard computation rule the computation is infinite.

After the first few iterations of the algorithm the answer for `p(X,Y,Z)` is computed, after widening the successive answers `p(0,0,0)`, `p(1,1,1)`, `p(2,2,2)`, ... This in turn generates a call to `q(X,Y,Z)`. The c-atom describing the answers for `p(X,Y,Z)` is $\langle \mathtt{p(X,X,X)}, \mathtt{X} \geq 0 \rangle$ and thus the call $\langle \mathtt{q(X,X,X)}, \mathtt{X} \geq 0 \rangle$ generated. Further iterations of the algorithm show that this call to `q` has no answers. Concretely, the call would generate an infinite failed computation.

When the algorithm terminates, the complete set of calls obtained is

$$\{\langle \mathtt{s(X,Y,Z)}, \mathsf{true} \rangle, \langle \mathtt{p(X,Y,Z)}, \mathsf{true} \rangle, \langle \mathtt{q(X,X,X)}, \mathtt{X} \geq 0 \rangle\}.$$

The set of answers is $\{\langle \mathtt{p(X,X,X)}, \mathtt{X} \geq 0 \rangle\}$. Thus we can see that there are some calls (namely, to `q` and `s`) that have no answers.

To generate the specialized program from this set of calls and answers, we generate resultants for the calls, and apply the answers to the bodies. Since there is no answer for `q(X,Y,Z)` in the resultant for `s(X,Y,Z)`, `s(X,Y,Z)` fails and the specialized program is empty. The specialized program thus consists only of the resultants `p(0,0,0)` and `p(X,X,X) <- {X = Y+1}, p(Y,Y,Y)`. The failure of the original goal is immediately apparent since there are no clauses for predicate `s`.

Example 6. More insight into the nature of the approximation can be gained by considering the same program as in the previous example, except that the body goals are reversed in the clause for `s`. In this case `q(X,Y,Z)` is called first, and the answers for `q` constrain the calls to `p`. The call $\langle \mathtt{q(X,Y,Z)}, \mathsf{true} \rangle$ results in the abstract answer c-atom $\langle \mathtt{q(X,Y,Z)}, \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y} \rangle$. Again, widening is essential to derive this answer. Note that the solution `q(0,0,0)` is included as a result of the convex hull approximation, even though this is not a concrete solution.

This answer is then propagated to the call to `p`, hence there is a call c-atom $\langle \mathtt{p(X,Y,Z)}, \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y} \rangle$. Specialization of this call to `p` gives the abstract answer $\langle \mathtt{p(X,X,X)}, \mathtt{X} \geq 0 \rangle$.

The specialized program corresponding to this set of calls and answers is the following.

```
s(0,0,0) <-
  q(0,0,0), p(0,0,0).
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,X1,X1)

q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

The instance of the clause for `s` is obtained by conjoining the answers for the body goals `q(X,Y,Z), p(X,Y,Z)`, that is, $\mathtt{X} \geq 0, \mathtt{X} = \mathtt{Y}, \mathtt{X} = \mathtt{Z}, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y}$, which simplifies to the constraint $\mathtt{X} = 0, \mathtt{Y} = 0, \mathtt{Z} = 0$. The above program does not make the failure of `s(X,Y,Z)` explicit; a non-trivial post-processing such as another run of the specialization algorithm would be needed to discover the failure of the call `q(0,0,0)`. The general point here is that the convex hull approximation loses the information that `q(0,0,0)` is not a solution for `q(X,Y,Z)`.

The two examples taken together show that the direction of propagation of answers affects precision. It would be possible to design an algorithm incorporating more sophisticated propagation, but post-processing or re-specialization is a practical alternative for experimental studies.

Note that the above presentation of the algorithm is naive in the sense that the sets of calls and answers need not be totally recomputed on each iteration. We use standard techniques to optimize the algorithm, focusing on the "new" calls and answers on each iteration. We can also use the recursive structure of the target program to optimize the iterative structure of the algorithm. Instead of one global fixpoint computation, we compute a series of fixpoints, one for each group of mutually recursive predicates.

3.4 Correctness of the Specialization

A program that has been specialized with respect to a c-atom $\mathcal{A} = \langle A, C \rangle$ produces the same answers as the original program for any terminating computation for any query in $\gamma(\mathcal{A})$. Note that the proposition below states nothing about the preservation of looping computations in the original program. A goal that loops in the original program can finitely fail in the specialized program.

Proposition 1. *Let P be a definite CLP program and $\mathcal{A}$ a c-atom. Let P' be the specialized program derived by the algorithm described above, with initial c-atom $\mathcal{A}$. Let S and T be the sets of call and answer c-atoms returned by the algorithm. Then for any goal $G = \leftarrow B_1, \ldots, B_k$ such that $i = 1 \cdots k$ and $B_i \in \gamma(\mathcal{A}')$ for some $\mathcal{A}' \in S$, $P \cup \{G\}$ has an answer ρ if and only if $P' \cup \{G\}$ has an answer ρ. Also, if $P \cup \{G\}$ fails finitely then $P' \cup \{G\}$ fails finitely.*

Proof. Suppose there is a terminating (possibly failed) derivation of $P' \cup \{G\}$. We argue by induction on the length of the derivation. If the derivation has length 0, then G fails immediately. We know that there is some $\mathcal{A}' = \langle A', C' \rangle$ in S such that $B_1 \in \gamma(\mathcal{A}')$, since the first call c-atom is $\langle B_1, \mathsf{true} \rangle$, and so S should contain an element $\mathcal{A}'$ such that $\langle B_1, \mathsf{true} \rangle \sqsubseteq \mathcal{A}'$. So a failure means that (i) there are no resultants for A', or (ii) that no resultant body has an answer, or (iii) that there is a resultant $A'\theta \leftarrow L$ with an answer φ for L given by the set of answer c-atoms, but B_1 does not unify with $A'\theta\varphi$. In the case of (i) there is a finitely failed computation tree of $P \cup \{G\}$. In the case of (ii) or (iii) there is either a finitely failed computation tree of $P \cup \{G\}$, or the computation tree for $P \cup \{G\}$ is infinitely failed.

If the derivation has length 1, with answer substitution ρ, then $G =\leftarrow B_1$ and there is some unit clause in P' whose head unifies with B_1 with substitution ρ. Now, unit clauses in P' may come from two sources: either they are already in P or they are the result of successfully unfolding the body of a non-unit clause, also in P. Hence by definition of the residual program construction the mgus are equivalent modulo variable renaming.

If all derivations of length at most m in P' have a corresponding derivation in P, then we show that all derivations of length $m+1$ in P' do as well. Suppose the first clause used in the derivation is $A'\theta \leftarrow L$, $\mathsf{mgu}(B_1, A'\theta) = \varphi$ and $(L, B_2, \ldots, B_k)\varphi$ has a derivation in P' of length at most m. By the induction hypothesis there is a corresponding derivation for $(L, B_2, \ldots, B_k)\varphi$ in P. Then clearly there is a derivation in P corresponding to the $m+1$ step derivation in P', obtained by concatenating the steps corresponding to the clause $A'\theta \leftarrow L$.

The above argument establishes soundness. For completeness, a sketch of a proof is provided. For each terminating derivation of $P \cup \{G\}$ we can construct a terminating derivation in $P' \cup \{G\}$. The clauses in P' that are needed to construct such a derivation exist by virtue of the closedness of the sets of calls and answers. That is $S = \omega(\mathsf{calls}(S,T), S)$ and $T = T\nabla\mathsf{answers}(S,T)$. Furthermore, the answers produced by successful derivations in P can be reproduced by derivations in P' by virtue of the correctness of the unfolding function aunf, and the procedure for computing the solution of a conjunction with respect to a set of answer c-atoms.

4 Examples

We implemented the algorithm described in the previous section, using the SICStus Prolog linear arithmetic constraint solver. Next we present some examples where on-line specialization as presented here is used for verifying some formulas in CTL [3].

Specialization can be seen as an approach to model-checking infinite systems [19, 7] and in this context our more powerful specialization techniques are highly relevant. We used the CTL metainterpreter shown in Fig. 3 (also used by M. Leuschel et al. [19]).

The set of transitions[4](predicate `trans/3` in the figure) of the system to be verified in the form of a (C)LP program is appended to this metainterpreter. Also, the property (predicate `prop/2` in the CTL metainterpreter) with respect to which verification is to be carried out should be specified. Finally, the specialization query provides the initial state and the CTL formula which is to be verified for the given system and initial state.

4.1 Specialization Strategy

Before applying the convex hull specialization, we performed a trivial top-down specialization with respect to the given goal. The main effect of this stage was

[4] A transition system may be that of a Kripke structure or a Petri Net, for instance.

```
sat(_,true) <-
sat(_,false) <- fail
sat(E,P) <- prop(E,P)
sat(E,and(F,G)) <-
        sat(E,F),
        sat(E,G)
sat(E,or(_F,G)) <-
        sat(E,G)
sat(E,or(F,_G)) <-
        sat(E,F)
sat(E,not(F)) <-
        not(sat(E,F))
sat(E,en(F)) <-
        trans(_Act,E,Ei),
        sat(Ei,F)
sat(E,an(F)) <-
        not(sat(E,en(not(F))))
sat(E,eu(F,G)) <-
        sat_eu(E,F,G)
```

```
sat(E,au(F,G)) <-
      sat(E,not(eu(not(G),
                   and(not(F),not(G))))),
      sat_noteg(E,not(G))
sat(E,ef(F)) <-
      sat(E,eu(true,F))
sat(E,af(F)) <-
      sat_noteg(E,not(F))
sat(E,eg(F)) <-
      not(sat_noteg(E,F))
sat(E,ag(F)) <-
      sat(E,not(ef(not(F))))
sat_eu(E,_F,G) <-
      sat(E,G)
sat_eu(E,F,G) <-
      sat(E,F),
      trans(_Act,E,Ei),
      sat_eu(Ei,F,G)
sat_noteg(E,F) <-
      sat(E,not(F))
sat_noteg(E,F) <-
      not((trans(_Act,E,Ei),
           not(sat_noteg(Ei,F))))
```

Fig. 3. CTL metainterpreter

to unfold the calls to the transition relation `trans/3`. In principle, this unfolding could be performed during the execution of the main specialization algorithm. However, the overall process is faster and easier to control when doing the specialization in two stages.

Example 7. Consider for instance the following transition system, where `trans(t,[X,Y],[Z,W])` holds iff state `[Z,W]` may be obtained from state `[X,Y]` using transition `t`.

```
trans(t1,[P1,P2],[X,P3]) <-
        X is 0,
        P1>=1,
        P2>=0,
        P3>=0,
        P3 is P2+1
```

```
trans(t2,[P1,P3],[P4,P2]) <-
        P1>=0,
        P2>=0,
        P4 is P1+2,
        P3 is P2+1
```

The encoding of an unsafe state property `[X,Y]` with `X>=3` is added as another clause in the CTL metainterpreter.

```
prop([X,Y],p(unsafe)) <- X>=3
```

The specialization query from initial state `[X,Y]` with `X=1,Y=0` for CTL formula `ef(p(unsafe))`[5] is `<- sat([1,0],ef(p(unsafe)))`. As a result of spe-

[5] Meaning that there exists a state in the future such that state property `unsafe` holds.

cializing the CTL metainterpreter with a description (transition system) of the system and a state property with respect to the query above, we obtained the empty program. This is equivalent to saying that there is no residual program in which state `[1,0]` may reach state `[X,Y]` with `X>=3`. Had we obtained a residual program we would have interpreted the residual program as the set of traces which lead from the initial state to the unsafe state, as above.

This behaviour may be regarded as that of a model checker, hence we argue that our specializer may be used as a model checker for some infinite state systems. The only requirement is that those systems may be expressed as definite (constraint) logic programs and the CTL formulas does not use negation.

Example 8. Figure 4 depicts a Petri net modeling one process with its critical section (`cs`) and a semaphore (`sema`) controlling access to it. The definition of

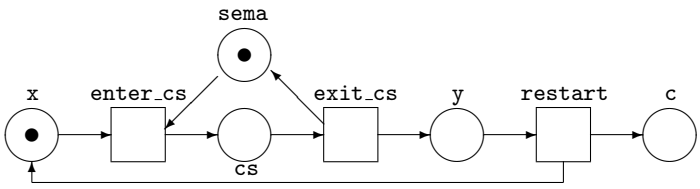

Fig. 4. Petri Net with one semaphore

predicate `trans/3` corresponding to the transition relation of the Petri net above, follows.

```
trans(enter_cs,[X,Sema,Cs,Y,C],[X1,Sema1,Cs1,Y,C]) <-
        X>=1, X1 is X-1,
        Sema>=1, Sema1 is Sema-1,
        Cs>=0, Cs1 is Cs+1
trans(exit_cs,[X,Sema,Cs,Y,C],[X,Sema1,Cs1,Y1,C]) <-
        Sema>=0, Sema1 is Sema+1,
        Cs>=1, Cs1 is Cs-1, Y>=0, Y1 is Y+1
trans(restart,[X,Sema,Cs,Y,C],[X1,Sema,Cs,Y1,C1]) <-
        X>=0, X1 is X+1,
        Y>=1, Y1 is Y-1,
        C>=0, C1 is C+1
```

Next, we may specify with the following clause the unsafe property of more than two processes being in their critical section (`cs`) at the same time:

```
prop([_X,_Sema,Cs,_Y,_C],p(unsafe)) <-  Cs>=2.
```

Now, for the specialization query, with constraint[6] `X>=1`:

[6] For every token in the place with name `X` we associate a process, thus the constraint `X>=1`.

```
<- sat([X,1,0,0,0],ef(p(unsafe)))
```

we obtained the empty program, thus denoting that there is no path from the initial state (`[X,1,0,0,0]`), with `X>=1` leading to a state where property `p(unsafe)` holds.

Example 9. Another way of specifying concurrent systems was proposed by U. A. Shankar [28]. Delzanno and Podelski [5], in turn, propose a systematic method to translate such specifications into CLP programs. Our translation is similar to theirs, differing only in the form of the clauses produced, mainly due to the meaning of the predicate employed.

Figure 5 below contains a specification of the bakery algorithm for two processes using the technique above cited.

Control variables $p_1, p_2 : \{think,\ wait,\ use\}$
Data variables $turn_1, turn_2$: int
Initial condition $p_1 = think \wedge p_2 = think \wedge turn_1 = turn_2 = 0$

Events	**cond** $p_1 = think$	**action** $p_1' = wait \wedge turn_1' = turn_2 + 1$
	cond $p_1 = wait \wedge turn_1 < turn_2$	**action** $p_1' = use$
	cond $p_1 = wait \wedge turn_2 = 0$	**action** $p_1' = use$
	cond $p_1 = use$	**action** $p_1' = think \wedge turn_1' = 0$
	... symmetrically for Process 2	

Fig. 5. The bakery algorithm

Such a specification may be readily translated into the following definition of the `trans` predicate:

```
trans(f,[think,A,P2,B],[wait,A1,P2,B]) <-  A>=0, A1 is B+1
trans(f,[P1,A,think,B],[P1,A,wait,B1]) <-  B>=0, B1 is A+1
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  A>=0, A<B
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  B>=0, B<A
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  B=0
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  A=0
trans(t,[use,A,P2,B],[think,A1,P2,B]) <-  A>=0, A1=0
trans(t,[P1,A,use,B],[P1,A,think,B1]) <-  B>=0, B1=0
```

Consequently, an unsafe property for the previous system would be a state where the two processes are in their critical section (denoted as `use`) at the same time. This property is denoted as the clause:

```
prop([use,A,use,B],p(unsafe)) <-
```

Furthermore, verifying that there is no state of the above mentioned system where such an unsafe state holds amounts to obtaining an empty program for the following query:

```
<-sat([think,0,think,0],ef(p(unsafe)))
```

where the variables denoting the turn of each process, namely `A,B`, are initially constrained by `A=B=0`. As a result of the specialization we obtained the empty program thus verifying that there is no unsafe state in any path beginning from the initial state described in figure 5.

In a similar way we verified some correctness property [19] of the producers and consumers algorithm [1] for one producer, one consumer and a buffer of size one. The authors [19] could not successfully specialize this last example.

4.2 Assessment

Here we have shown some applications of our specialization strategy to infinite state model checking. Compared to other approaches using specialization for the same purpose, we believe our approach sheds some insight into the field. The example of the bakery protocol was also verified by Fioravanti et al. [7]. As opposed to their approach we show the actual specialization strategy and its use in other related examples. We depart from a general CTL metainterpreter whereas Fioravanti et al. present a somehow specialized version of a CTL metainterpreter.

For the other examples of this section M. Leuschel et al. [19] have a four stage model checker, as opposed to ours which is just one specialization step. That is, M. Leuschel et al. first pass through an off-line specializer, then one or more specialization passes of their on-line specializer and finally one pass through a most-specific-version analyser.

Admittedly examples 8 and 9 in this section do not propagate answers, and require a simple unfolding prior to specialization with answers. By contrast, example 7 and the producer-consumer of [19] do not need any prior unfolding and have some limited answer propagation. That is, specialization with answers could be applied directly to the metainterpreter (together with the transition definition and the property), to yield the expected verification results. The running example of Section 3 does indeed need and use answer propagation.

5 Related Work

Despite the fact that unfold-fold approaches to program transformation and program specialization based on a fixpoint calculation are not directly comparable, there are some unfold-fold methods related to our techniques.

In [21] the authors propose the use of convex-hull analysis to enable optimization/specialization of CLP programs. Their removal, refinement and reordering may be rendered as transformation rules. The fairness of comparing our technique with theirs is dubious because theirs is used for compilation and ours for specialization, and potentially the former is a special case of the latter one. A weak form of their method was later dubbed by Fioravanti et al. [6] as contextual specialization.

Peralta and Gallagher [23] use arithmetic constraints (convex hulls) to specialize CLP programs, especially an interpreter for imperative programs.

Their specialization abstract domain is the same as that one used here, but the specializer only propagates information top-down and cannot achieve the effects of answer propagation.

Fioravanti et al. [6] (without reference to [13, 23]) argue an automatic specialization method based on folding and unfolding among other transformation techniques. They use a domain of atomic formulas constrained by arithmetic expressions with upper bound based on widening alone, rather than the combination of convex hull and widening which is known to give better approximations. The aspects of their method concerned with specialization resemble a top-down on-line specializer with a subsequent "contextual specialization", and thus does not in general achieve the effects of answer propagation.

Another application of specialization using abstract interpretation over polyhedral descriptions followed by a contextual specialization was given by Howe *et al.* [13]. This approach is similar in being based on abstract interpretation over a domain of polyhedra. Its bottom-up analysis of answers is not as powerful as ours, which combines top-down and bottom-up propagation.

Conjunctive Partial Deduction (CPD) [27] aims to solve the answer propagation problem in a different way. The approach is to preserve shared information between subgoals by specializing conjunctions rather than atoms. It is not yet clear whether CPD or answer propagation via atoms, or some combination of both, will be most effective. In the extreme case of CPD, no resolvent is ever split, and no answer propagation is needed. However in general resolvents can be of unbounded size, some splitting is therefore needed, and answer propagation is required to preserve shared information between conjunctions.

6 Final Remarks

We have presented an algorithm for specialization of definite (C)LP programs. Its main novelty is the propagation of calls and answers described by atoms whose arguments are described by convex hulls. The use of answer propagation with an expressive domain like convex hulls gives increased specialization. By interpreting Prolog arithmetic as constraints we can also apply the algorithm to "non-constraint" programs.

6.1 Future Work

At the moment we can only specialize definite (constraint) logic programs. Because negation in CTL is interpreted as negation in (constraint) logic programs, this restricts us to model checking of safety properties, as opposed to liveness properties. Extending the presented techniques to include negation is the focus of our current research.

Scalability of our specialization method is one avenue into which we plan to extend the current proposal. Thus making our specialization techniques applicable to larger systems.

Also, in order to improve precision of our specialization with answers more sophisticated domains are sought.

References

1. Krysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.
2. F. Benoy and A. King. Inferring argument size relations in CLP($\mathcal{R}$). In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, pages 204–223, Sweden, 1996. Springer-Verlag, LNCS 1207.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 2000.
4. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978. Also in "http://www.di.ens.fr/ cousot/COUSOTpapers/POPL78.shtml".
5. G. Delzanno and A. Podelski. Model Checking in CLP. In W. R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–239. Springer-Verlag, LNCS 1579, 1999.
6. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Automated strategies for specialising constraint logic programs. In Kung-Kiu Lau, editor, *10th International Workshop on Logic-based Program Synthesis and Transformation*, pages 125–146. Springer-Verlag, LNCS 2042, 2000.
7. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. Technical Report DSSE-TR-2001-3, Department of Electronics and Computer Science, University of Southampton, 2001. Proceedings of the Second International Workshop on Verification and Computational Logic (VCL'01).
8. J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.
9. J. Gallagher, M. Codish, and E.Y. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.
10. J. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.
11. John P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemman, editors, *Partial Evaluation*, pages 115–136. Springer-Verlag, LNCS 1110, 1996.
12. John P. Gallagher and Julio C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3):143–172, 2001.
13. J.M. Howe and A. King. Specialising finite domain programs using polyhedra. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *Springer-Verlag Lecture Notes in Computer Science*, pages 118–135, April 2000.
14. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19(20):503–581, 1994.
15. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

16. Neil D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pages 396–405, 1997.
17. M. Leuschel. Program specialisation and abstract interpretation reconciled. In Joxan Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press.
18. M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In A. Pettorossi, editor, *(Pre)Proceedings of LOPSTR-2001 11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR-2001), Paphos, Cyprus*, December 2001.
19. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 62–81. Springer Verlag, LNCS 1817, 1999.
20. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.
21. K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *Proceedings of the Twentieth Symposium on Principles of Programming Languages*, pages 334–344, Charleston, South Carolina, 1993. ACM Press.
22. Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.
23. Julio C. Peralta and John P. Gallagher. Imperative program specialisation: An approach using CLP. In Annalisa Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 102–117. Springer Verlag, LNCS 1817, 1999.
24. G. Puebla, M. Hermenegildo, and J. P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 75–84, San Antonio, Texas, January 1999.
25. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, pages 135–151, 1970.
26. Hüseyin Sağlam and John P. Gallagher. Constrained regular approximations of logic programs. In N. Fuchs, editor, *LOPSTR'97*, pages 282–299. Springer-Verlag, LNCS 1463, 1997.
27. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2-3):231–277, 1999. Erratum appeared in JLP 43(3): 265(2000).
28. U. A. Shankar. An Introduction to Assertional Reasoning for Concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.