Type Analysis for CHIP*

Włodzimierz Drabent[1] and Paweł Pietrzak[2]

[1] IPI PAN, Polish Academy of Sciences, Ordona 21, Pl - 01-237 Warszawa
and IDA, Linköpings universitet
[2] IDA, Linköpings universitet, S - 581 83 Linköping, Sweden.
{wlodr,pawpi}@ida.liu.se.

Abstract. This paper proposes a tool to support reasoning about (partial) correctness of constraint logic programs. The tool infers a specification that approximates the semantics of a given program. The semantics of interest is an operational "call-success" semantics. The main intended application is program debugging. We consider a restricted class of specifications, which are regular types of constrained atoms.
Our type inference approach is based on bottom-up abstract interpretation, which is used to approximate the declarative semantics (c-semantics). By using "magic transformations" we can describe the call-success semantics of a program by the declarative semantics of another program. We are focused on CLP over finite domains. Our prototype program analyzer works for the programming language CHIP.

1 Introduction and motivation

In this paper we are interested in supporting reasoning about program correctness in the context of CLP (constraint logic programming). Speaking informally, a program is correct if it behaves as expected by the user. But user expectations are seldom well documented. This paper describes an analyzer that for a given CLP program produces a characterization of the form of calls and successes in any execution of the program starting from a given class of goals. The user may inspect the description produced to see whether it conforms to her expectations. We deal with partial correctness, the given program is partially correct w.r.t. the obtained description.

The starting point are well-known verification conditions for partial correctness of logic programs wrt to a specification, which gives a set of procedure calls and a set of procedure successes. (Such verification conditions were proposed in [DM88,Dra88]; a useful special case was given in [BC89,AM94]). We generalize the verification conditions for the case of CLP.

Generally the conditions are undecidable. But they become decidable for a restricted class of specifications. For the case of LP (logic programming) it was shown [Boy96] that it is sufficient to consider specifications describing regular tree sets. In the literature this kind of specifications is often called regular types [YS91,DZ92]. While successes and calls in LP are atoms, their counterpart in

* This work has been supported by the ESPRIT 4 Project 22532 DiSCiPl.

CLP are constrained atoms. Therefore this paper adapts regular types for CLP so that one can describe sets of constrained terms and atoms. This includes adaptation of certain operations on regular types.

To compute semantic approximations of programs, we need static analysis techniques. We show that the verification conditions for a CLP program P constitute another CLP program Q whose declarative semantics describes the calls and successes. (Such approach is often called "magic transformation"). For this purpose we introduce a generalization for CLP of c-semantics [Cla79,FLMP89]; this results in more precise descriptions than using the standard $\mathcal{D}$-model semantics. We adapt then the technique of Gallagher and de Waal [GdW92,GdW94] of bottom-up abstract interpretation to synthesize an approximation of the c-semantics of Q; it also is an approximation of the call-success semantics of P. As a side effect we obtain a tool to approximate the declarative semantics of CLP programs.

We are particularly interested in CLP over finite domains (CLP(FD)) [Hen89], especially the language CHIP [Cos96]. We have implemented a prototype type analysis system for CHIP. It is a major modification of the system described in [GdW92,GdW94]. A preliminary version of our work was presented in [DP98b].

The use of types, as in our work, to approximate the semantics of programs in an untyped language is usually called *descriptive* typing. Another approach is *prescriptive* typing. In that approach the type information, provided by the programmer, influences the semantics of a program. In particular, variables are typed and may only be bound to the values from the respective types. Usually the programmer is required to provide types for function symbols and/or for predicates. Prescriptive typing is a basis of a few programming languages (e.g. TypedProlog [LR91], Gödel [HL94], Mercury [SHC96]).

Experience with languages like Gödel shows that their mechanism of types is able to find numerous errors at compile time. This is an immense advantage in comparison to finding them during testing and run-time debugging. Our work adds a similar potential of static checking to any typeless CLP language (by comparing the types obtained from the analysis with the intended ones).

The paper is organized as follows. The next section summarizes basic concepts of CLP and presents the declarative and the operational semantics. Then we propose a system of regular types for CLP. Section 4 describes the type inference method used in this work. Then we present an example of type analysis for CHIP.

2 Semantics of CLP

In this work we employ two semantics of CLP. We need a semantics providing information about the form of procedure calls and successes during the execution of CLP programs; this is the role of a call-success semantics. The analysis method employs magic transformation, so we also need a declarative semantics. Both semantics are introduced below in this section.

Most of implementations of CLP use syntactic unification[1]. In this paper we are interested in CLP with syntactic unification, we believe however that our work can be adapted to the "standard" CLP.

2.1 Basic concepts

We consider a fixed constraint domain. It is given by fixing a signature and a structure $\mathcal{D}$ over this signature. Predicate symbols of the signature are divided into *constraint predicates* and (non-constraint) *predicates*. The former have a fixed interpretation in $\mathcal{D}$, the interpretation of the latter is defined by programs. Similarly, the function symbols are divided into *interpreted function symbols* and *constructors*. All the function symbols have a fixed interpretation. It is assumed that the interpretations of constructors are bijections with disjoint co-domains. So the elements of structure $\mathcal{D}$ can be seen as terms built from some elementary values by means of constructors[2]. That is why we will often call them $\mathcal{D}$-terms. An *atomic constraint* is an atomic formula with a constraint predicate symbol. Throughout this paper by a *constraint* we will mean an atomic constraint or $c_1 \wedge c_2$ or $c_1 \vee c_2$ or $\exists x c_1$, where c_1 and c_2 are constraints and x is a variable. A CLP clause is of the form: $h \leftarrow c, b_1, \ldots, b_n$ where $h, b_1, \ldots, b_n$ are atoms (i.e. atomic formulae built up from non-constraint predicate symbols) and c is a conjunction of atomic constraints. A CLP program is a finite set of CLP clauses.

2.2 Declarative semantics

The standard least $\mathcal{D}$-model semantics is insufficient for our purposes. We are interested in the actual form of computed answers[3]. Two programs with the same least $\mathcal{D}$-model semantics may have different sets of computed answers. For instance take the following two CLP(FD) programs

$$P_1 = \{\, p(1).;\ p(2).\,\} \qquad P_2 = \{\, p(x) \leftarrow x \in \{1,2\}.\,\}$$

and a goal $p(x)$. Constraint $x \in \{1,2\}$ is an answer for P_2 but not for P_1. In order to describe such differences, we generalize the c-semantics [Cla79,FLMP89]. For logic programs, this semantics is given by the set of (possibly non ground) atomic logical consequences of a program. The c-semantics for CLP will be expressed by means of constrained atoms.

1 In CLP with syntactic unification, function symbols occurring outside of constraints are treated as constructors. So, for instance in CLP over integers, the goal $p(4)$ fails with the program $\{p(2{+}2){\leftarrow}\}$, but the goal $X\#{=}4, p(X)$ succeeds (where $\#{=}$ is the constraint of arithmetical equality).

2 Notice that in many CLP languages function symbols play also the role of constructors. For instance, the interpretation of $2+3$ may be a number, while that of $a+3$ (where a is a 0-ary constructor) is a $\mathcal{D}$-term with the main symbol $+$.

3 $\mathcal{D}$-model semantics can be used to describe CLP with syntactic unification, one has to made $\mathcal{D}$ to be a a Herbrand domain. (No element of the carrier of such a domain is a value of two distinct ground terms).

Definition 1. A *constrained expression* (atom, term, ...) is a pair $c[]E$ of a constraint c and an expression E such that each free variable of c occurs (freely) in E.

If ν is a valuation such that $\mathcal{D} \models \nu(c)$ then $\nu(E)$ is called an $\mathcal{D}$-*instance* of $c[]E$.

A constrained expression $c'[]E'$ is an *instance* of a constrained expression $c[]E$ if c' is satisfiable in $\mathcal{D}$ and there exists a substitution θ such that $E' = E\theta$ and $\mathcal{D} \models c' \to c\theta$ ($c\theta$ means here applying θ to the free variables of c, with a standard renaming of the non-free variables of c if a conflict arises).

If $c[]E$ is an instance of $c'[]E'$ and vice versa then $c[]E$ is a *variant* of $c'[]E'$

By the *instance-closure* $cl(E)$ of a constrained expression E we mean the set of all instances of E. For a set S of constrained expressions, its instance-closure $cl(S)$ is defined as $\bigcup_{E \in S} cl(E)$.

Note that, in particular, $c\theta[]E\theta$ is an instance of $c[]E$ and that $c'[]E$ is an instance of $c[]E$ whenever $\mathcal{D} \models c' \to c$. The relation of being an instance is transitive. (Take an instance $c'[]E\theta$ of $c[]E$ and an instance $c''[]E\theta\sigma$ of $c'[]E\theta$. As $\mathcal{D} \models c'' \to c'\sigma$ and $\mathcal{D} \models c' \to c\theta$, we have $\mathcal{D} \models c'' \to c\theta\sigma$).

Notice also that if c is not satisfiable then $c[]E$ does not have any instance (it is not an instance of itself).

We will often not distinguish E from $true[]E$ and from $c[]E$ where $\mathcal{D} \models \forall c$. Similarly, we will also not distinguish $c[]E$ from $c'[]E$ when c and c' are equivalent constraints ($\mathcal{D} \models c \leftrightarrow c'$).

Example 2. $a + 7$, $Z + 7$, $1{+}7$ are instances of $X + Y$, but 8 is not.

$f(X){>}3[]f(X){+}7$ is an instance of $Z{>}3[]Z{+}7$, which is an instance of $Z + 7$, provided that constraints $f(X){>}3$ and $Z{>}3$, respectively, are satisfiable.

Assume a numerical domain with the standard interpretation of symbols. Then $4 + 7$ is an instance of $X{=}2{+}2[]X{+}7$ (but not vice versa), the latter is an instance of $Z{>}3[]Z{+}7$.

Consider CLP(FD) [Hen89]. A domain variable with the domain S, where S is a finite set of natural numbers, can be represented by a constrained variable $x{\in}S\,[]\,x$ (with the expected meaning of the constraint $x{\in}S$).

If $Vars(c) \not\subseteq Vars(E)$ then $c[]E$ will denote $(\exists_{-Vars(E)}c)[]E$ (where $\exists_{-V}$ stands for quantification over the variables not in V).

Two notions of groundness arise naturally for constrained expressions. $c[]E$ is *syntactically ground* when E contains no variables. $c[]E$ is *semantically ground* if it has exactly one $\mathcal{D}$-instance.

Now we define the c-semantics for CLP with syntactic unification. In the next definition we apply substitutions to program clauses. So let us define $\downarrow\! P$ as $\{\, C\theta \mid C \in P,\ \theta \text{ is a substitution} \,\}$.

Definition 3 (Immediate consequence operator for c-semantics). Let P be a CLP program. $T_P^{\mathcal{C}}$ is a mapping over sets of constrained atoms, defined by

$$\begin{aligned} T_P^{\mathcal{C}}(I) = \{\ c[]h \mid\ & (h{\leftarrow}c', b_1, \ldots, b_n) \in {\downarrow}P,\ n \geq 0, \\ & c_i[]b_i \in I, \ \text{ for } i = 1, \ldots, n, \\ & c = \exists_{-\,Vars(h)}(c', c_1, \ldots, c_n), \\ & \mathcal{D} \models \exists c \quad \} \end{aligned}$$

(where $Vars(E)$ is the set of free variables occurring in E).

Notice that in the definition syntactic unification is used for parameter passing, but terms occurring in constraints are interpreted w.r.t. $\mathcal{D}$.

$T_P^{\mathcal{C}}$ is continuous w.r.t. $\subseteq$. So it has the least fixpoint $T_P^{\mathcal{C}} \uparrow \omega = \bigcup_{i=0}^{\infty} (T_P^{\mathcal{C}})^i(\emptyset)$. By the *declarative semantics* (or *c-semantics*) $M(P)$ of P we mean the instance-closure of the least fixpoint of $T_P^{\mathcal{C}}$:

$$M(P) = cl(T_P^{\mathcal{C}}{\uparrow}\omega).$$

Speaking informally, cl is used here only to add new constraints but not new (non-constraint) atoms: As $T_P^{\mathcal{C}}{\uparrow}\omega$ is closed under substitution, for every $c[]u \in M(P)$ there exists a $c'[]u \in T_P^{\mathcal{C}}{\uparrow}\omega$ such that $\mathcal{D} \models c \rightarrow c'$.

Example 4. Consider programs P_1 and P_2 from the beginning of this section. $M(P_1) = \{p(1), p(2)\}$. $T_{P_2}^{\mathcal{C}}{\uparrow}\omega$ contains $p(1)$, $p(2)$ and $x{\in}\{1,2\}[]p(x)$. (It also contains variants of the latter constrained atom, obtained by renaming variable x). $M(P_2)$ contains additionally all the instances of $x{\in}\{1,2\}[]p(x)$, like $y{=}1[]p(y)$.

The traditional least $\mathcal{D}$-model semantics and the c-semantics are related by the fact that the set of $\mathcal{D}$-instances of the elements of $M(P)$ is a subset of the least $\mathcal{D}$-model of P. If we take a least $\mathcal{D}$-model semantics for CLP with syntactic unification (where $\mathcal{D}$ is a Herbrand domain) then the set of $\mathcal{D}$-instances of the elements of $M(P)$ and the least $\mathcal{D}$-model of P coincide.

2.3 Call-success semantics

We are interested in the actual form of procedure calls and successes that occur during the execution of a program. We assume the Prolog selection rule. Such semantics will be called the *call-success semantics.*

Without loss of generality we can restrict ourselves to atomic initial goals. Given a program and a class of initial goals, we want to provide two sets of constrained atoms corresponding to the calls and to the successes. For technical reasons it is convenient to have just one set. So for each predicate symbol p we introduce two new symbols ${}^{\bullet}p$ and $p^{\bullet}$; we will call them *annotated predicate symbols.* They will be used to represent, respectively, call and success instances of atoms whose predicate symbol is p. For an atom $A = p(\tilde{t})$, we will denote ${}^{\bullet}p(\tilde{t})$ and $p^{\bullet}(\tilde{t})$ by ${}^{\bullet}A$ and $A^{\bullet}$ respectively. We will use analogous notation for constrained atoms. (If $A = c[]p(\tilde{t})$ then ${}^{\bullet}A = c[]{}^{\bullet}p(\tilde{t})$, etc).

The call-success semantics is defined in terms of the computations of the program. For a given operational semantics, which specifies what the computations of a program are, one defines what are the *procedure calls* and the *procedure successes* of these computations. For logic programs and LD-resolution this is done for instance in [DM88]. It is rather obvious how to generalize it to CLP, we omit the details.

Definition 5. Let P be a CLP program and $\mathcal{G}$ a set of constrained atoms. Their *call-success semantics* $CS(P, \mathcal{G})$ is a set of constrained atoms (with annotated predicate symbols) such that

1. $c[]^{\bullet}p(\tilde{t}) \in CS(P, \mathcal{G})$ iff there exists an LD-derivation for P with the initial goal in $\mathcal{G}$ and in which $c[]p(\tilde{t})$ is a procedure call;
2. $c[]p^{\bullet}(\tilde{t}) \in CS(P, \mathcal{G})$ iff there exists an LD-derivation for P with the initial goal in $\mathcal{G}$ and in which $c[]p(\tilde{t})$ is a procedure success.

We will characterize the call-success semantics of a program P as the declarative semantics of some other program P^{CS}. In logic programming this approach is often called "magic transformation". Program P^{CS} can also be viewed as the verification conditions of the proof method of [BC89] or an instance of the verification conditions of the proof method of [DM88].

Proposition 6. Let P be a CLP program and $\mathcal{G}$ a set of constrained atoms. Then

$$cl(CS(P, \mathcal{G})) = cl\left((T^{\mathcal{C}}_{P^{CS}})^{\omega}(\mathcal{G})\right)$$

where P^{CS} is a program that for each clause $H \leftarrow c, B_1, \dots, B_n$ from P contains clauses:

$$\begin{array}{l}
c, {}^{\bullet}H \rightarrow {}^{\bullet}B_1 \\
\dots \\
c, {}^{\bullet}H, B_1^{\bullet}, \dots, B_{i-1}^{\bullet} \rightarrow {}^{\bullet}B_i \\
\dots \\
c, {}^{\bullet}H, B_1^{\bullet}, \dots, B_{n-1}^{\bullet} \rightarrow {}^{\bullet}B_n \\
c, {}^{\bullet}H, B_1^{\bullet}, \dots, B_n^{\bullet} \rightarrow H^{\bullet}
\end{array}$$

PROOF (outline) One shows that all the procedure calls and successes occurring in (a prefix of) an SLD-derivation of length j are in $(T^{\mathcal{C}}_{P^{CS}})^{j}(\mathcal{G})$. Conversely, for any member of $(T^{\mathcal{C}}_{P^{CS}})^{j}(\mathcal{G})$ the corresponding call/success occurs in a derivation. Both proofs are by induction on j. $\square$

Assume that the set of initial constrained goals is characterized by a CLP program P': $\mathcal{G} = \{\, A \mid {}^{\bullet}A \in M(P')\,\}$. Assume that no predicate $p^{\bullet}$ occurs in P'. From the last proposition it follows that the declarative semantics of $P^{CS} \cup P'$ describes the call-success semantics of P:

$$cl(CS(P, \mathcal{G})) = M(P^{CS} \cup P') \cap \mathcal{A}$$

where $\mathcal{A}$ is the set of all constrained atoms with annotated predicate symbols. (The role of the intersection with $\mathcal{A}$ is to remove auxiliary predicates that may originate from P').

3 Types

We are interested in computing approximations of the call-success semantics of programs. A program's semantics is an instance closed set of constrained atoms, an approximation is its superset. The approximations are to be manipulated by an analysis algorithm and communicated to the user.

We need a suitable class of approximations and a language to specify them. We extend for that purpose the formalism of regular unary logic programs [YS91] used in LP to describe regular sets of terms/atoms.[4] We call such sets regular (constraint) types. So we use (a restricted class of) CLP programs and their declarative c-semantics to describe approximations of the call-success semantics of CLP programs.

3.1 Regular unary programs

Our approach to defining types is a generalization of canonical regular unary logic (RUL) programs [YS91]. We begin this section with presenting RUL programs. Then we introduce our generalization, called RULC programs. We conclude with several examples.

To define types we will use a restricted kind of programs, with unary predicates only. In such a program R a predicate symbol t is considered to be a *name* of a type and $[\![t]\!]_R := \{\, c[]u \mid c[]t(u) \in M(R) \,\}$ is the corresponding type.

Definition 7. A (canonical) regular unary logic program (**RUL program**) is a finite set of clauses of the form:

$$t_0(f(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n). \tag{1}$$

(where $n \geq 0$ and $x_1,\ldots,x_n$ are distinct variables) such that no two clause heads have a common instance.

Notice that the types defined by a RUL program are sets of ground terms. (For such programs there is no difference between the c-semantics and the least Herbrand model semantics).

RUL programs were introduced in [YS91]. In [FSVY91] they are called reduced regular unary-predicate programs. The formalism defines tuple distributive [Mis84,YS91] sets of terms. So if $f(u_1,u_2)$ and $f(u_1',u_2')$ are members of such a set then also $f(u_1,u_2')$ and $f(u_1',u_2)$ are. (For exact definitions the reader is referred to [Mis84,YS91]).

We will write $F[x_1,\ldots,x_n]$ to stress that F is a formula such that $Vars(F) \subseteq \{x_1,\ldots,x_n\}$. $F[u_1,\ldots,u_n]$ will denote F with each x_i replaced by the term u_i.

[4] The formalism is equivalent to deterministic root-to-frontier tree automata [GS97], to (deterministic) regular term grammars (see e.g. [DZ92] and references therein) and to type graphs of [JB92,HCC95].

Definition 8. A constraint $c[x]$ in a constraint domain $\mathcal{D}'$ will be called a **regular constraint** if there exists a RUL program R and a predicate symbol t such that for any ground term u, $\mathcal{D}' \models c[u]$ iff $u \in [\![t]\!]_R$. Constraint c will be called the **corresponding constraint** for t and R.

Notice that a constraint corresponding to a RUL program may be not regular (if $\mathcal{D}'$ is a non Herbrand domain). For instance consider domain $\mathcal{D}'$ of integers, where $+$ is an interpreted function symbol. Take a program $R = \{t(4).\}$. The set of terms satisfying the corresponding constraint contains for instance $1+3$ and $3+1$ but not $3+3$. So it cannot be described by a RUL program.

The next definition provides a CLP generalization of RUL programs. From now on we assume that the constraint domain $\mathcal{D}$ contains the regular constraints.

Definition 9. By an *instance of the head* of a clause $h \leftarrow c, b_1, \ldots, b_n$ (where c is a constraint and $b_1, \ldots, b_n$ are non constraint atoms) we mean an instance of $c[]h$. A regular unary constraint logic program (**RULC program**) is a finite set of clauses of the form (1) or of the form

$$t_0(x) \leftarrow c[x]. \tag{2}$$

(where $c[x]$ is a regular constraint) such that no two clause heads have a common instance.

Example 10. The type t described by the RUL program $\{\, t(2).,\ t(3).,\ t(4).\,\}$ is the set $\{2, 3, 4\}$ of ground terms.

Consider CLP(FD) [Hen89]. To describe type t extended by a domain variable, with $\{2, 3, 4\}$ as its domain, we use a regular constraint $x{\in}\{2, 3, 4\}$ in a RULC program $R' = \{\, t'(x) \leftarrow x{\in}\{2, 3, 4\}\,\}$. Indeed, $[\![t']\!]_{R'} = cl(x{\in}\{2, 3, 4\}\,[]\,x)$.

Example 11. A type of lists with (possibly nonground) elements satisfying a constraint c can be expressed by the following RULC program R:

$$\begin{array}{l} list([\,]) \leftarrow . \\ list([x|xs]) \leftarrow elem(x), list(xs). \\ elem(x) \leftarrow c[x] \end{array}$$

The c-semantics of this program is
$M(R) = cl\left(\{\, c[x_1], \ldots, c[x_n]\,[]\,list([x_1, \ldots, x_n]) \mid n \geq 0\,\} \cup \{\, c[x][]elem(x)\,\}\right)$.

Let Q be a RUL program such that $c[x]$ is the corresponding constraint for $elem$ and Q. Replacing in R the last clause by (the clauses of) Q results in a RUL program R' describing the set of ground lists from the previous type.

Let $c_{list}[x]$ be the corresponding constraint for $list$ and R'. A type of possibly non-ground lists with elements of the type $elem$ can be defined by a one clause RULC program R''

$$list(x) \leftarrow c_{list}[x].$$

The type contains unbound variables whose further bindings are restricted to be lists (i.e. constrained variables of the form $c_{list}[y][]y$). It also contains all their instances. Thus our approach makes it possible to express prescriptive types like those of programming language Gödel [HL94].

Comparing the three list types presented here, we obtain $[\![list]\!]_{R'} \subseteq [\![list]\!]_R \subseteq [\![list]\!]_{R''}$.

Example 12. The type of all ground terms (over the given signature) is defined by predicate *ground* and a (RUL) program containing the clause $ground(f(x_1,\ldots,x_n)) \leftarrow ground(x_1),\ldots,ground(x_n)$ for each function symbol f of arity $n \geq 0$.

The type of all constrained terms is defined by predicate *any* and program $\{\, any(x) \leftarrow true \,\}$.

3.2 Operations on types

In type analysis some basic operations on types are employed. One has to perform a check for type emptiness and inclusion. One has to compute the intersection and (an approximation of) the union of two types[5]. One has to find type $\{\, c_1,\ldots,c_n[\!]f(u_1,\ldots,u_n) \mid c_i[\!]u_i \in [\![t_i]\!],\ i=1,\ldots,n \,\}$ for given types $t_1,\ldots,t_n$, and for a given type t and an i find type $\{\, (\exists_{-Vars(u_i)}c)[\!]u_i \mid c[\!]f(u_1,\ldots,u_n) \in [\![t]\!] \,\}$. These operations for RULC are generalization of those for RUL [GdW94], and are described in [DP98a]. Here we present only an example. To find the intersection of the types t_1, t_2 defined by

$$\begin{array}{l} t_1(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n) \\ t_2(x) \leftarrow c[x] \end{array}$$

we construct clauses

$$\begin{array}{l} (t_1 \sqcap t_2)(f(x_1,\ldots,x_n)) \leftarrow (r_1 \sqcap s_1)(x_1),\ldots,(r_n \sqcap s_n)(x_n). \\ s_1(x_1) \leftarrow \exists_{-\{x_1\}}\, c[f(x_1,\ldots,x_n)]. \\ \ldots \\ s_n(x_n) \leftarrow \exists_{-\{x_n\}}\, c[f(x_1,\ldots,x_n)]. \end{array}$$

Here $r \sqcap s$ is a new type, it is the intersection of types r, s. $s_1,\ldots,s_n$ are new types. Notice that $\exists_{-\{x_i\}}\, c[f(x_1,\ldots,x_n)]$ is a regular constraint.

3.3 Regular programs as an abstract domain

In this section we present how RULC programs are used to approximate the semantics of CLP programs. We also show that it is a rather unusual case of abstract interpretation, as most of the commonly required conditions [CC92] are not satisfied.

In our approach, the concrete domain **C** is that of the semantics of programs. So **C** is the set of sets of constrained atoms over the given language. (We do not

[5] The union of two types defined by RULC programs may be not definable by RULC programs.

need to make the domain more sophisticated by removing from $\mathbf{C}$ those elements that are not the meaning of any program). $(\mathbf{C}, \subseteq)$ is a complete lattice.

We want to approximate sets of constrained atoms by RULC programs. Following [GdW92,GdW94] we introduce a distinguished (unary) predicate symbol *approx*. The type corresponding to *approx* in a RULC program R is understood as the set of constrained atoms specified by R. Notice that the arguments of *approx* are treated both as atoms and as terms, we use here the ambivalent syntax [AB96]. So R *approximates* a set I of constrained atoms iff $I \subseteq [\![approx]\!]_R$. We will call such a program R a *regular approximation* of I.

Example 13. Let P be the following CLP(R) program

$rev([\,], Y, Y).$
$rev([f(V,X)|T],\ Y, Z) \leftarrow V{*}V + X{*}X < 9,\ rev(T, Y,\ [f(V,X)|Z]).$

Then the following program is a regular approximation of $M(P)$.

$approx(rev(X,Y,Z)) \leftarrow t1(X), any(Y), any(Z).$
$t1([\,]).$
$t1([X|Xs]) \leftarrow t2(X),\ t1(Xs).$
$t2(f(X,Y)) \leftarrow t3(X),\ t3(Y).$
$t3(X) \leftarrow -3 < X,\ X < 3.$

So the abstract domain $\mathbf{A}$ is the set of RULC programs (over the given language). The concretization function $\gamma : \mathbf{A} \rightarrow \mathbf{C}$ is defined as the meaning of *approx*:

$$\gamma(R) := [\![approx]\!]_R.$$

The ordering of the concrete domain induces the relation $\preceq$ on $\mathbf{A}$:

$$R \preceq R' \text{ iff } \gamma(R) \subseteq \gamma(R').$$

$\preceq$ is a pre-order but not a partial order.

This is a case of abstract interpretation, in which an abstraction function *does not exist*. The reason is, roughly speaking, that there may exist an infinite decreasing sequence of regular approximations (of some $I \in \mathbf{C}$) which does not have a g.l.b. in $\mathbf{A}$[6] [DP98a].

We want also to mention that the abstract immediate consequence function $T_P^{\mathcal{A}}$, defined later on and used in type inference, may be not monotonic. So its least fixpoint may not exist. The properties outlined above hold already for the approach of [GdW92,GdW94]; this contradicts some claims of [GdW92,GdW94].

[6] This property also holds when the pre-order $(\mathbf{A}, \preceq)$ is replaced by the induced partial order on the set $\mathbf{A}/_{\preceq}$. Also, using another natural pre-order on $\mathbf{C}$ ($R \sqsubseteq R'$ iff $M(R) \subseteq M(R')$) does not improve the properties discussed in this section.

3.4 Types for CLP(FD)

The concept of *finite domains* was introduced to logic programming by [Hen89]. We will basically follow this framework, including the terminology. So within this section "domain" stands for a finite domain in the sense of [Hen89]. We assume that a domain is a finite set of natural numbers (including 0). This is the case in most of CLP(FD) languages. To any domain S there corresponds a *domain constraint* $x \in S$, with the expected meaning. Usually a variable involved in such a constraint is called a domain variable.

In our type analysis for CHIP we use some types that correspond to restrictions on the form of arguments of finite domain constraint predicates. We need the type of natural numbers, the type of integers, the type of finite domains (the l.u.b. of the types of the form $cl(x{\in}S[]x)$), the type of arithmetical expressions and its subset of so called linear terms.

Defining the first three of them by a RULC program would require an infinite set of clauses. So we extend RULC programs by three "built-in" types[7]. We introduce unary predicate symbols nat, neg and $anyfd$, which cannot occur in the left hand side of a RULC clause. We assume that (independently from a RULC program) $[\![nat]\!]$ is the set of all non-negative integer constants, $[\![neg]\!]$ is the set of all negative integer constants and $[\![anyfd]\!]$ is $cl(\{\, x{\in}S[]x \mid S \subseteq \mathrm{N},\ S \text{ is finite} \,\})$.[8] We allow clauses of the form $t(x) \leftarrow builtin(x)$ to occur in RULC programs (where $builtin$ is one of the three symbols). By an instance of the head of such clause we mean any element of $[\![builtin]\!]$.

The type int of integers and the type of arithmetical expressions are defined by means of these special types by a RULC program. The type of linear terms cannot be defined by a RULC program. (For instance, for domain variables x, y and a natural number n, it contains $x * n$ and $n * y$ but not $x * y$). So we use a RULC description of a superset of it.

4 Type inference

The core of our method is computing a regular approximation of the c-semantics of a program. It is described in [DP98a], here we present an outline. Our approach is based on [GdW92,GdW94], it can be seen as a bottom-up abstract interpretation. We use a function $T_P^{\mathcal{A}} : \mathbf{A} \to \mathbf{A}$, which approximates the immediate consequence operator $T_P^{\mathcal{C}}$. The program semantics $M(P)$ is approximated by a fixpoint of $T_P^{\mathcal{A}}$. A technique of widening, similar to that of [CC92], is applied to assure that a fixpoint is reached in a finite number of steps.

For a CLP program P and an RULC program R, $T_P^{\mathcal{A}}(R)$ is defined as

$$T_P^{\mathcal{A}}(R) = norm\left(R \amalg \coprod_{C \in P} solve(C, R) \right).$$

[7] Alternatively we can assume that the type of integers is finite. A similar solution is taken in constructing a semantics for CLP with interval constraints [BO97].

[8] If all the finite domains are the subset of some maximal domain $0..max$, then this type may be defined by a RULC clause $anyfd(x) \leftarrow x \in 0..max$.

Here *norm* [GdW94,DP98a] is a widening function; $R \preceq norm(R)$ for any R. For RULC programs Q and Q', $Q \amalg Q'$ is a RULC program such that $Q \preceq Q \amalg Q'$ and $Q' \preceq Q \amalg Q'$. It is computed using the type union operation of Sect. 3.2.

The main function is *solve*, which gives a regular approximation of $T^{\mathcal{C}}_{\{C\}}(\gamma(R))$: $T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C,R))$. Due to lack of space we only briefly outline its definition. It is based on that of [GdW92,GdW94]. The main difference is that we take into account the constraints occurring in clause C. Let $C = h \leftarrow c, b_1, \ldots, b_m$, where $c[x_1, \ldots, x_n]$ is a conjunction of elementary constraints. We approximate c by computing a "projection" of c. The projection consists of one argument constraints $c_1[x_1], \ldots, c_n[x_n]$ such that

$$\mathcal{D} \models c[x_1, \ldots, x_n] \rightarrow c_1[x_1], \ldots, c_n[x_n].$$

It is computed using the constraint solver of the underlying CLP implementation (or possibly some more powerful solver). So the types defined in the RULC program $R' = \{\, t_i(x_i) \leftarrow c_i[x_i] \mid i = 1, \ldots, n \,\}$ approximate the sets of possible values of the variables in c. Now clause $C' = h \leftarrow t_1(x_1), \ldots, t_n(x_n), b_1, \ldots, b_m$ is submitted as an argument to the function **solve** of [GdW92,GdW94], together with $R \cup R'$ as the second argument. It computes an approximation of $T^{\mathcal{C}}_{\{C'\}}(\gamma(R \cup R'))$, thus of $T^{\mathcal{C}}_{\{C\}}(\gamma(R))$.

As $T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C,R))$ and $R \preceq norm(R)$, we have that $T^{\mathcal{A}}_P$ approximates the concrete semantic function $T^{\mathcal{C}}_P$:

$$T^{\mathcal{C}}_P(\gamma(R)) \subseteq \gamma(T^{\mathcal{A}}_P(R))$$

and thus $\forall n\; T^{\mathcal{C}}_P \uparrow n \subseteq \gamma(T^{\mathcal{A}}_P \uparrow n)$.

Due to widening, a fixed point of $T^{\mathcal{A}}_P$ is found in a finite number of iterations (conf. [GdW94]); $T^{\mathcal{A}}_P \uparrow n = T^{\mathcal{A}}_P \uparrow \omega$, for some n. We call it the *computed fixpoint*. Function $T^{\mathcal{A}}_P$ is in general not monotonic w.r.t. $\preceq$ (as *norm* is not monotonic [DP98a] and $\amalg$ is not required to be). Thus we cannot claim that the computed fixpoint is the least fixpoint.

The result $T^{\mathcal{A}}_P \uparrow \omega$ of the computation approximates $M(P)$ as $M(P) = lfp(T^{\mathcal{C}}_P) \subseteq \gamma(T^{\mathcal{A}}_P \uparrow \omega) = [\![approx]\!]_{T^{\mathcal{A}}_P \uparrow \omega}$

5 Examples

This section presents a type analysis of two example programs. The user interface of our prototype analyser employs, instead of RULC programs, a more convenient formalism. So we explain it before coming to the examples.

To provide a more compact and more readable notation, we use *regular term grammars with constraints*. They can be seen as an abbreviation for RULC programs. A clause $t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$ is represented by the *grammar rule* $t_0 \rightarrow f(t_1, \ldots, t_n)$, a clause $t(x) \leftarrow c[x]$ by the rule $t \rightarrow c[x]$.

The formalism includes parametric types. It uses type symbols of arity ≥ 0 and type variables; terms built out of them are called *type terms*. A *parametric grammar rule* is of the form $t(\alpha_1, \ldots, \alpha_k) \rightarrow f(t_1, \ldots, t_n)$ where t is a k-ary type symbol, t_j are type terms and α_i are type variables. (One requires that $Vars(t_1, \ldots, t_n) \subseteq \{\alpha_1, \ldots, \alpha_k\}$). Such a rule stands for a family of RULC clauses represented by the (non parametric) rules $t(s_1, \ldots, s_k) \to f(t_1, \ldots, t_n)\theta$, where s_i are arbitrary types and θ is the substitution $\{\alpha_i/s_i \mid i = 1, \ldots, k\}$. [9]

For example, rules

$$list(\alpha) \to [\,] \qquad\qquad list(\alpha) \to [\alpha|list(\alpha)]$$

correspond to a family of RULC programs

$$list(t)([\,]). \qquad\qquad list(t)([x_1|x_2]) \leftarrow t(x_1), list(t)(x_2).$$

which for any type term t define the type $list(t)$ of lists of elements of type t.

The user may declare some types by providing (possibly parametric) grammar rules.[10] Whenever possible, the system uses the declared types in its output. Thus the output may be expressed (partially) in terms of types familiar to the user; this can substantially improve the readability of the results of the analysis. For instance, assume that the system derives a type t with the corresponding fragment of a RULC program:

$$t([\,]). \qquad\qquad t([x|y]) \leftarrow nat(x), t(y).$$

Then, instead of displaying the RULC clauses (or actually the corresponding grammar) the system informs that the type is $list(nat)$. Notice that the system does not infer parametric polymorphic types, the polymorphism comes only from user declarations.

As the first example we use the following program, which solves the well-known N-queens problem. The current version of our analyzer treats all the finite domains in a uniform way, namely as *anyfd* (the types of the form $cl(x \in S[\,]x)$ are not yet implemented).

```
:- entry nqueens(nat,any).
nqueens(N,List) :- length(List,N), List::1..N,
                   constraint_queens(List), labeling(List).
labeling([]).
labeling([X|Y]) :- indomain(X), labeling(Y).
constraint_queens([]).
constraint_queens([X|Y]) :- safe(X,Y,1), constraint_queens(Y).
safe(_,[],_).
safe(X,[Y|T],K) :- noattack(X,Y,K), K1 is K+1, safe(X,T,K1).
```

[9] So now the predicate symbols of RULC are type terms. We allow only such grammars for which no two corresponding clauses have a common head instance (conf. Def. 9). We should deal with finite RULC programs. But the program corresponding to a set of parametric rules may be infinite. So a condition on grammars is imposed: in the obtained RULC program any type should depend on a finite set of types. For details see [DP98a,DZ92].

[10] The widely used type $list(\alpha)$, declared as above, is predefined in the system.

The `entry` declaration indicates the top goal and its call patterns for the call-success analysis. Types inferred by the system are presented below.

```
call    : nqueens(nat,any)
success : nqueens(nat,list(nat))
-------------------------
call    : labeling(list(anyfd))
success : labeling(list(nat))
-------------------------
call    : constraint_queens(list(anyfd))
success : constraint_queens(list(anyfd))
-------------------------
call    : safe(anyfd,list(anyfd),int)
success : safe(anyfd,list(anyfd),int)
-------------------------
call    : noattack(anyfd,anyfd,int)
success : noattack(anyfd,anyfd,int)
```

Assume now that the second clause defining `safe/3` contains a bug:

```
safe(X,[Y|T],K):-noattack(X,Y,K),K1 is K+1,safe(X,t,K1).  % bug here
```

Types inferred by the analyzer look like follows (we show only those which differ from ones generated previously):

```
success : nqueens(nat,t102)
t102 --> [nat|t78]
t102 --> []
t78 --> []
-------------------------
call    : labeling(t90)
t90 --> []
t90 --> [anyfd|t78]
success : labeling(t102)
-------------------------
success : constraint_queens(t90)
-------------------------
call    : safe(anyfd,t71,int)
t71 --> []
t71 --> [anyfd|list(anyfd)]
t71 --> t
success : safe(anyfd,t78,int).
```

The types inferred are obviously suspicious and should be helpful in localizing the bug in the program. For instance, the second argument of success of `nqueens/2` (type `t102`) is an empty list or a one-element list of naturals. A similar problem is with `constraint_queens`. The problem may be traced down to `safe/3` which succeeds with the empty list as the second argument.

The next example illustrates inferring non-trivial constraints in the approximation of a program. The predicate `split5(Xs,Ls,Gs)` splits an input list `Xs` of finite domain variables (or natural numbers) into lists of elements less and greater or equal to 5 (`Ls` and `Gs` respectively).

```
:-entry split5(list(anyfd),any,any).
split5([],[],[]).
split5([X|Xs],[X|Ls],Gs) :- X #< 5, split5(Xs,Ls,Gs).
split5([X|Xs],Ls,[X|Gs]) :- X #>= 5, split5(Xs,Ls,Gs).
```

The inferred types are presented below.

```
call    : split5(list(anyfd),any,any)
success : split5(list(anyfd),list(t1),list(t2))
t1 --> X #< 5
t2 --> X #>= 5
```

6 Conclusions and future work

In this paper we propose a method of computing semantic approximations for CLP programs. Our aim is a practical tool that would be helpful in debugging. We are mainly interested in CLP(FD), particularly in the language CHIP. Our approach considers the (operational) call-success semantics and the (declarative) c-semantics.

As a specification language to express the semantic approximations we propose a system of regular types for CLP, which is an extension of an approach used for logic programs. The types are defined by (a restricted class of) CLP programs, called RULC programs. We present an algorithm for computing regular approximations of the declarative semantics. This algorithm can also be used for approximating the call-success semantics, due to a characterization of this semantics by the c-semantics of a transformed program.

We have adopted a regular approximation system (described in [GdW92,GdW94]) to constraint logic programming over finite domains. The current version analyzes programs in the language CHIP. We expect it to be easily portable to work with other CLP languages, as we have isolated its parts responsible for the built-ins of CHIP. The prototype has been implemented in CHIP and has been ported to SICStus Prolog and CIAO [CLI97]. The latter implementation is a part of an assertion-based framework for debugging in CLP [PBM98].

The system presents types to the user as regular term grammars, which are more easily comprehensible than RULC programs. This provides a restricted but useful kind of polymorphism (conf. Section 5)

A subject for future work is obtaining more precise analysis by using a more sophisticated treatment of constraints. We also plan to evaluate the method experimentally by applying it to non-toy programs.

Another direction of further work is relating our technique to abstract debugging [CLMV98]. A clear relationship between these two techniques should be established. The first step is a diagnosis method [CDP98,CDMP98] which finds the clauses responsible for a program being incorrect w.r.t. a type specification. That work uses the type system presented here as the class of specifications. Computing an approximation of $T_C^{\mathcal{C}}$, as discussed in Sect. 4, is at the core of the diagnosis algorithm.

ACKNOWLEDGMENT
The authors want to thank Jan Małuszyński for discussions and suggestions.

References

[AB96] K.R. Apt and R. Ben-Eliyahu. Meta-variables in Logic Programming, or in Praise of Ambivalent Syntax. *Fundamenta Informaticae*, 28(1-2):22–36, 1996.

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–764, 1994.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. Lecture Notes in Computer Science.

[BO97] F. Benhamou and W. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997.

[Boy96] J. Boye. *Directional Types in Logic Programming.* Linköping studies in science and technology, dissertation no. 437, Linköping University, 1996.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programming. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CDMP98] M. Comini, W. Drabent, J. Małuszyński, and P. Pietrzak. A type-based diagnoser for CHIP. ESPRIT DiSCiPl deliverable, September 1998.

[CDP98] M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP programs using type information. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[Cla79] K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

[CLI97] The CLIP Group. *CIAO System Reference Manual.* Facultad de Informática, UPM, Madrid, August 1997. CLIP3/97.1.

[CLMV98] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 1998. To appear.

[Cos96] Cosytec SA. *CHIP System Documentation*, 1996.

[DM88] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59:133–155, 1988.

[DP98a] W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, September 1998.

[DP98b] W. Drabent and P. Pietrzak. Type analysis for CHIP. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[Dra88] W. Drabent. On completeness of the inductive assertion method for logic programs. Unpublished note (available from `www.ipipan.waw.pl/~drabent`), Institute of Computer Science, Polish Academy of Sciences, May 1988.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[FSVY91] T. Fruewirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In G. Kahn, editor, *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 300–309, Amsterdam, July 1991. IEEE Computer Society Press. Corrected version available from `http://WWW.pst.informatik.uni-muenchen.de/~fruehwir`.

[GdW92] J. Gallagher and D. A. de Waal. Regular Approximations of Logic Programs and Their Uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

[GdW94] J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, 1997.

[HCC95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, March 1995.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[LR91] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proc. of the 8th International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.

[PBM98] G. Puebla, F. Bueno, and Hermenegildo M. A framework for assertion-based debugging in constraint logic programming. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[SHC96] Z. Somogyi, F. Hederson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):14–64, 1996.

[YS91] E. Yardeni and E. Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.