Inferring call and success types for CLP programs*

Włodzimierz Drabent†, Paweł Pietrzak‡

September 28, 1998

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 main intended application is program debugging. We deal with a "call-success" semantics of CLP. We consider a restricted class of specifications, which are regular types of constrained atoms. The call-success semantics of a CLP program is characterized by the declarative semantics of another CLP program ("magic transformation"). Then bottom-up abstract interpretation is used to approximate the latter.

We study the theoretical background of this approach. We are mainly interested in applying it to CLP over finite domains. Our prototype program analyzer works for the programming language CHIP.

1 Introduction and motivation

The work reported addresses the problem of correctness of CLP programs. Intuitively, 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

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

†IPI PAN, Polish Academy of Sciences, Ordona 21, Pl - 01-237 Warszawa and IDA, Linköping University.

‡IDA, Linköping University, S - 581 83 Linköping, Sweden. E-mail: {wlodr,pawpi}@ida.liu.se.

it conforms to her expectations. The language of description has thus to be relatively simple in order to allow effective automatic inference and to be easily understood by the user. This paper presents a concretization of this idea and a tool based on it. This is an extension of our previous work [DP98].

The approach proposed is an adaptation of the existing static analysis techniques for logic programs to the needs of CLP. The results adapted and the contributions of this paper are as follows.

The starting point are well-known verification conditions for partial correctness of LP 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]). The question is what is the semantics suitable for description of calls and successes in CLP, and whether the verification conditions of LP can be adapted for this semantics. This paper introduces the call-success semantics for CLP and provides verification conditions for it.

Generally the conditions are undecidable. But they become decidable for a restricted class of specifications. As shown by Boye [Boy96], in the case of LP 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 CLP are constrained atoms. Therefore this paper adapts regular types for CLP so that one can describe sets of constrained terms and atoms. This requires adaptation of the operations on regular types. We define these operations and we prove their properties.

Finally we need static analysis techniques. We show that the verification conditions for a CLP program constitute another CLP program 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 [FLMP89]. This results in more precise descriptions than using the standard $\mathcal{D}$-model semantics. We adopt then the technique of Gallagher and de Wall [GdW92, GdW94] of bottom-up abstract interpretation to synthesize approximations of the call-success semantics of a given program. As a side effect we obtain a tool to approximate the declarative semantics of CLP programs. We deal with partial correctness, the approximations are supersets of the actual semantics of programs. (A wider class of approximations is discussed in [BDD$^+$97]).

Using 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 influences the semantics of a program. In particular, variables are typed and may only be bound to the values from the respective types. Prescriptive typing is

a basis of a few programming languages (e.g. TypedProlog [LR91], Gödel [HL94], Mercury [SHC96]). In contrast to prescriptive typing, our approach is applicable to any (C)LP language.

We implemented a type inference tool which is a major modification of the LP analyzer of Gallagher and de Waal [GdW92, GdW94]. Our first prototype has been implemented in CHIP for analysis of CHIP programs. Subsequently it has been ported in cooperation with UPM Madrid and with COSYTEC to other platforms (SICSTUS, Ciao) for off-line analysis of CHIP programs. It is possible to modify it for analysis of other CLP languages. The type-inference tool is also used as a part of our type-based diagnoser for CHIP which automatizes localization of type errors [CDP98, CDMP98].

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 section we present the approach to CLP semantics used in our work. We want to express groundness information in our semantics. We have to take into account that most of implementations of CLP use a semantics with syntactic unification[1]. As the declarative semantics we will use a generalization of c-semantics [FLMP89] to CLP. The operational semantics is given by sets of call and success instances of atomic goals.

After recalling some basic notions of constraint logic programming we introduce the declarative and operational semantics used in this work.

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

[1]Many actual implementations of CLP use syntactic unification: function symbols occurring outside of constraints are treated as constructors. So, for instance, $5-1$ and $2+2$ are not unifiable.

bijections with disjoint co-domains. So the elements of structure $\mathcal{D}$ can be seen as (finite) terms built from some elementary values by means of constructors. That is why we will often call them $\mathcal{D}$-terms[2].

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

A *primitive constraint* is an atomic formula with a constraint predicate symbol. Throughout this paper by a *constraint* we will mean a primitive 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 primitive constraints. A CLP program is a finite set of CLP clauses.

2.2 CLP with syntactic unification

We want to point out an important semantic feature of CLP languages. There exist two substantially different variants of their semantics. Many actual implementations use syntactic unification: function symbols occurring outside of constraints are treated as constructors. This results in a semantics different from the standard one that is given by the least $\mathcal{D}$-models of programs. For instance, consider $\mathcal{D}$ being integers and take the program

$$\begin{array}{l} p(X) \leftarrow q(X), r(X). \\ q(2+3). \\ r(6-1). \end{array}$$

$p(5)$ is true in the least $\mathcal{D}$-model of the program. However in languages like CHIP, query $p(X)$ fails, as $2+3$ and $6-1$ are not unifiable. We will refer to CLP with the second kind of semantics as to *CLP with syntactic unification*. To characterize this semantics by the least $\mathcal{D}$-model of a program one has to use a Herbrand domain as $\mathcal{D}$. (No element of the carrier of such a domain is a value of two distinct ground terms).

In this paper we are mainly interested in CLP with syntactic unification. We believe however that our approach can be adapted to the standard semantics of CLP.

[2]Sometimes we will slightly abuse the notation and use $\mathcal{D}$ to denote the set of $\mathcal{D}$-terms.

2.3 Declarative semantics

The standard least $\mathcal{D}$-model semantics is insufficient for our purposes. We are interested in the actual form of computed answers. 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

Definition 2.1 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 $\mathcal{D} \models c' \rightarrow c\theta$ ($c\theta$ 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 $\mathcal{D} \models c' \rightarrow c$. 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 $\mathcal{D} \models c'' \rightarrow c'\sigma$ and $\mathcal{D} \models c' \rightarrow c\theta$, we have $\mathcal{D} \models c'' \rightarrow c\theta\sigma$).

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 2.2 $a+7$, $Z+7$, 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 2.3 [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

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

(where $Vars(E)$ is the set of free variables occurring in E, and $\exists_{-V}$ stands for quantification over the variables not in V). □

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 2.4 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. We expect that c-semantics for CLP without syntactic unification can be described in a similar way. This topic is however outside of the scope of this paper.

2.4 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*. In this section we will also refer to a more detailed operational semantics, given by LD-resolution (SLD-resolution with the Prolog selection rule).

Without lack 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 that will become clear later, it is convenient to have just one set. 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). If M is a set of constrained atoms then ${}^\bullet M$ is $\{{}^\bullet A \mid A \in M\}$ and $M^\bullet$ is $\{A^\bullet \mid A \in M\}$.

We assume a natural generalization of LD-resolution, with constrained goals of the form $c[]A_1, \ldots, A_n$ (where A_i are atoms) and with derivations that are sequences of constrained goals, mgu's and input clauses (similarly to [Llo87]). For a constrained goal $G_i = c[]A_1, \ldots, A_n$ and a clause $C_{i+1} = H \leftarrow c', B_1, \ldots, B_m$ the next goal in an LD-derivation is $G_{i+1} = c''[](B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$, provided that θ is an mgu of A and H, constraint c'' is equivalent to $(c \wedge c')\theta$ and is satisfiable and $Vars(G_i) \cap Vars(C_{i+1}) = \emptyset$.

We adapt the definition of procedure call and success from [DM88]. If $G_i = c[]A_1, \ldots, A_n$ is a goal then $c[]A_1$ is the corresponding *procedure call*. (Remember that $c[]A_1$ is an abbreviation for $\exists_{-Vars(A_1)} c[]A_1$). If in an LD-derivation $G_0, G_1, \ldots$ with the mgu's $\theta_1, \theta_2, \ldots$ we have $G_i = c[]A_1, \ldots, A_n$ and j is the least number such that $j > i$ and $G_j = c'[](A_2, \ldots, A_n)\theta$, where $\theta = \theta_{i+1} \cdots \theta_j$, then $c'[]A_1\theta$ is the *procedure success* corresponding to the

procedure call in G_i (in this LD-derivation).

Definition 2.5 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 2.6 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 $H \leftarrow c, B_1, \ldots, B_n$ from P contains clauses:

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

□

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.

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[3]. Following the terminology of LP 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.

For communication with the user we use a more terse formalism of regular term grammars with constraints (Section 3.6). The formalism provides additionally a parametric mechanism. One can define a family of types, like $list(t)$; such a definition describes the set of list with elements of any type t.

Section 3.1 describes the class of types we use and a way of specifying them by a certain class of CLP programs. The next section discusses some technical properties of the constraints used in these programs. These properties are used in Section 3.3, where we present algorithms for operations on types. The operations are needed in computing semantic approximations of programs. Then we discuss our class of types from the point of view of abstract interpretation. In Section 3.5 we apply the ideas of the former sections to construct a system of types for CLP(FD). Finally we generalize the notion of regular term grammars to the case of constrained terms.

3.1 Regular unary programs

Our approach to defining types is a generalization of canonical regular unary logic (RUL) programs [YS91]. We use (a restricted class of) CLP programs to approximate the semantics of (arbitrary) CLP programs.

We begin with presenting RUL programs. Then we introduce our generalization, called RULC programs. We conclude this section with several examples.

The next definition combines a success set of a given unary predicate and a corresponding set of constrained terms.

Definition 3.1 Let P be a CLP program. Let p be a unary predicate. Then $[\![p]\!]_P := \{\, c[]u \mid c[]p(u) \in M(P) \,\}$. □

[3]The formalism is equivalent to deterministic root-to-frontier tree automata [GS97] and to (non parametric) regular term grammars (see e.g. [DZ92] and references therein).

A type will be determined by a unary predicate in a (restricted kind of a) constraint logic program. All the predicates of a program R (which defines types) are unary; a predicate symbol t is considered to be a *name* of a type and $[\![t]\!]_R$ is the corresponding type.

Definition 3.2 A (canonical) regular unary logic program (**RUL program**) is a finite set of clauses of the form:

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n).$$

(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]).

Before introducing a CLP generalization of RUL programs we need some definitions. A clause with predicate symbol p in its head will be called a clause *defining* p. A predicate p in a program P *depends* on a predicate q if $p = q$ or in the r.h.s. of a clause defining p there occurs a predicate p' which depends on q. (Formally, relation "depends on" is the least relation satisfying these conditions). A clause is *relevant* for p if it defines q and p depends on q. 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.

Definition 3.3 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. Conversely, program R will be called a **corresponding program** for c and t, provided that all the clauses of R are relevant for t.

□

Notice that if $\mathcal{D}'$ is a Herbrand domain then the corresponding constraint for a RUL program is regular. This may not be the case for a non Herbrand domain. For instance consider domain $\mathcal{D}'$ of integers, where $+$ is an

interpreted function symbol. So terms $1+3$ and 4 denote the same value in $\mathcal{D}'$. Take a program $R = \{t(4).\}$. The corresponding constraint should be satisfied by $1+3$ and by $3+1$ but not by $3+3$. So the set of terms for which it is satisfied 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 3.4 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:

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

(where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables) 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 3.5 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]. Let S be a finite set of integers. Assume that we want to describe a type containing a domain variable with S as its domain. (It is natural that the type also contains all possible instantiations of the variable). To do this, we use a regular constraint $x{\in}S$ in a RULC program $R' = \{\, t'(x) \leftarrow x{\in}S \,\}$. Indeed, $[\![t']\!]_{R'} = cl(x{\in}S \,[]\, x)$ contains the constants corresponding to the elements of S and the constrained terms of the form $x{\in}S' \,[]\, x$, where $S' \subseteq S$.

□

Example 3.6 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 corresponding (RUL) program for $c[x]$ and *elem*. 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 c-semantics of this program is $M(R'') = cl(\, c_{list}[x][]list(x)\,)$. Notice that the type *list* (i.e. the set $[\![list]\!]_{R''} = cl(c_{list}[x][]x)$) contains unbound variables whose further bindings are restricted to be lists.[4] Thus our approach makes it possible to express prescriptive types like those of programming language Goedel [HL94].

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

□

Example 3.7 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 $\{\, any(x) \leftarrow true \,\}$. Notice that the RUL program defining *ground* is the corresponding program for constraint *true*. □

3.2 Regular constraints

In this section we discuss some properties of regular constraints and corresponding RUL programs, which will be used later.

Let $c[x]$ be a regular constraint and R be a corresponding program for c and t. Consider the constraint $c[f(x_1, \ldots, x_n)]$ (obtained by replacing in c the variable x by the term $f(x_1, \ldots, x_n)$), where $x_1, \ldots, x_n$ are distinct variables. Assume that $c[f(x_1, \ldots, x_n)]$ is satisfiable. So there must exist a clause $t(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$ in R, and the clause is unique. Consider constraints

$$c_i[x_i] := \exists_{-\{x_i\}}\, c[f(x_1, \ldots, x_n)]$$

[4] It also contains, for instance, open lists with such variables as their tails (i.e. terms of the form $c_{list}[y][][u_1, \ldots, u_n|y]$ where y is a variable and $u_1, \ldots, u_n$ are possibly nonground terms of type *elem*).

$(1 \leq i \leq n)$. We have $u_1 \in [\![t_1]\!]_R$ iff $f(u_1, \ldots, u_n) \in [\![t]\!]_R$ for some $u_2, \ldots, u_n$ iff $\mathcal{D} \models c[f(u_1, \ldots, u_n)]$ for some $u_2, \ldots, u_n$ iff $\mathcal{D} \models c_1[u_1]$. (Here $u_1, \ldots, u_n$ are ground terms). The same reasoning holds for $u_2, \ldots, u_n$. Thus c_i is the corresponding constraint for t_i and R, for $i = 1, \ldots, n$.

Now consider an arbitrary term u and the constraint $c[u]$. Assume that $c[u]$ is satisfiable and that $y \in \mathit{Vars}(u)$. From the previous property, by induction on the depth of u, we obtain that

$$\exists_{-\{y\}}\, c[u]$$

is the corresponding constraint for some predicate t' in R. So for a given y, the class of constraints of this form modulo equivalence is finite. (Constraints c_1 and c_2 are equivalent iff $\mathcal{D} \models c_1 \leftrightarrow c_2$). Moreover, the corresponding constraint for any t' in R is (equivalent to one) of this form.

3.3 Operations on RULC programs

Now we discuss basic algorithms for types described by RULC programs. We discuss membership check, checks for type emptiness and type inclusion, and computing the intersection and (an approximation of) the union of types. These operations will be employed in type analysis in section 4.

Let R be an RULC program. A method of checking whether a ground term u is in a type $[\![t]\!]_R$ is obvious (and is linear time w.r.t. the size of the term), provided that an algorithm to check whether a ground term satisfies a regular constraint is given (and is linear time).

Checking if a constrained term is in $[\![t]\!]_R$ can be based on the following property. A constrained variable $c[]x \in [\![t]\!]_R$ iff there exists a clause $t(y) \leftarrow c'[y]$ in R such that $\mathcal{D} \models c \rightarrow c'[x]$. For non variable constrained terms, $c[]f(u_1, \ldots, u_n) \in [\![t]\!]_R$ iff there exists a clause $t(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$ in R and $c[]u_i \in [\![t_i]\!]_R$ for $i = 1, \ldots, n$, or there exists $t(y) \leftarrow c'[y]$ in R such that $\mathcal{D} \models c \rightarrow c'[f(u_1, \ldots, u_n)]$. The resulting algorithm is also linear time provided that the check for $\mathcal{D} \models c \rightarrow c'$ is linear time.

The set of empty types in an RULC program R can be computed as follows. Mark as empty every predicate t which contains no clauses in its definition or contains only clauses with unsatisfiable constraints. Then mark as empty each predicate t such that all clauses defining t contain at least one predicate marked as empty; until no new marks can be added. Now a predicate t is marked iff $[\![t]\!]_R = \emptyset$.

Consider a RULC program R. We show how to construct a RUL program $\mathit{ground}(R)$ that defines ground types corresponding to the possibly non ground types defined by R.

Definition 3.8 Let R be a RULC program. Let $R_c = \{ t_i(x) \leftarrow c_i[x] \mid i = 1, \ldots, n \}$ be those clauses of R that contain constraints. Let (for $i = 1, \ldots, n$) R_i be a corresponding program for c_i and t_i, such that any predicate symbol distinct from $t_1, \ldots, t_n$ occurs in at most one of the programs $R, R_1, \ldots, R_n$ (and that t_i does not occur in R_j, for $i \neq j$). Then

$$ground(R) := (R \setminus R_c) \cup R_1 \cup \cdots \cup R_n.$$

□

It is easy to see that $ground(R)$ is a RUL program and that $[\![t]\!]_{ground(R)}$ is the set of ground terms from $[\![t]\!]_R$, for any predicate t.

Definition 3.9 [Inclusion] Let R_1, R_2 be RULC programs (not necessarily distinct). Relation $\sqsubseteq$ is defined as the greatest[5] relation such that for any predicate symbols t_1, t_2 from respectively R_1, R_2, $(t_1, R_1) \sqsubseteq (t_2, R_2)$ iff

1. for every clause $t_1(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_1$, $n \geq 0$,
   (a) there is a clause $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_2$ such that $(r_i, R_1) \sqsubseteq (s_i, R_2)$ for $1 \leq i \leq n$, or
   (b) there is a clause $t_2(x) \leftarrow c[x] \in R_2$ and a clause $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_c$, where R_c is a corresponding program for $c[x]$ and t_2, and $(r_i, ground(R_1)) \sqsubseteq (s_i, R_c)$ for $1 \leq i \leq n$.
2. For every clause of the form $t_1(x) \leftarrow c_1[x] \in R_1$ there is a clause $t_2(x) \leftarrow c_2[x] \in R_2$ such that $(t_1, R_{c_1}) \sqsubseteq (t_2, R_{c_2})$, where R_{c_j} is a corresponding program for c_j and t_j $(j = 1, 2)$.

□

We will sometimes abbreviate $(t_1, R_1) \sqsubseteq (t_2, R_2)$ to $t_1 \sqsubseteq t_2$. The next proposition shows that relation $\sqsubseteq$ indeed corresponds to relation $\subseteq$ between types:

Proposition 3.10 For RULC programs R, R' and predicates t, t', if $(t, R) \sqsubseteq (t', R')$ then $[\![t]\!]_R \subseteq [\![t']\!]_{R'}$ (i.e. t is a subtype of t').

If $[\![t]\!]_R \subseteq [\![t']\!]_{R'}$ and $[\![t]\!]_R \neq \emptyset$ then $(t, R) \sqsubseteq (t', R')$. □

[5] $\sqsubseteq$ is a relation on $PP \times RR$ where RR is a finite set of programs, containing $R_1, R_2, ground(R_1), ground(R_2)$ and the programs (corresponding to the constraints of R_1, R_2) used in the construction of $ground(R_1), ground(R_2)$. PP is the set of predicates occurring in RR.

Definition 3.11 [Intersection] Let R_1, R_2 be RULC programs. We construct a RULC program R where for each pair t_1, t_2 of predicates of respectively R_1, R_2 a new predicate $t_1 \sqcap t_2$ is defined.

Take a variable y. For each constraint $c[x]$ occurring in R_i $(i = 1, 2)$ and any term u such that $y \in \mathit{Vars}(u)$ consider the constraint $\exists_{-\{y\}}\, c[u]$. As shown in Section 3.2, the set of such constraints is finite (when we do not distinguish between equivalent constraints). For each such constraint add to R_i a clause $t_u(y) \leftarrow \exists_{-\{y\}}\, c[u]$, where t_u is a new predicate symbol, obtaining a RULC program R_i'.

1. if $t_1(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_1$ and
   $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_2$ then R contains a clause
   $(t_1 \sqcap t_2)(f(x_1, \ldots, x_n)) \leftarrow (r_1 \sqcap s_1)(x_1), \ldots, (r_n \sqcap s_n)(x_n)$.

2. If
   $$t_i(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_i$$
   $$t_j(x) \leftarrow c[x] \in R_j'$$
   (where $\{i, j\} = \{1, 2\}$) and if $c[f(x_1, \ldots, x_n)]$ is satisfiable then R contains clause
   $$(t_1 \sqcap t_2)(f(x_1, \ldots, x_n)) \leftarrow (r_1 \sqcap s_1)(x_1), \ldots, (r_n \sqcap s_n)(x_n).$$
   where clauses
   $$s_1(x_1) \leftarrow \exists_{-\{x_1\}}\, c[f(x_1, \ldots, x_n)].$$
   $$\ldots$$
   $$s_n(x_n) \leftarrow \exists_{-\{x_n\}}\, c[f(x_1, \ldots, x_n)].$$
   occur in $R_j' \setminus R_j$.

3. If
   $$t_1(x) \leftarrow c_1[x] \in R_1'$$
   $$t_2(x) \leftarrow c_2[x] \in R_2'$$
   then R contains a clause
   $$(t_1 \sqcap t_2)(x) \leftarrow c_1[x], c_2[x].$$

□

The obtained program is a RULC program, as the conjunction $c_1[x], c_2[x]$ of regular constraints is a regular constraint. (This follows for instance from applying the following proposition to RUL programs).

Proposition 3.12 For R_1, R_2, R, t_1, t_2 and $t_1 \sqcap t_2$ as in the last definition,

$$[\![t_1 \sqcap t_2]\!]_R = [\![t_1]\!]_{R_1} \cap [\![t_2]\!]_{R_2}$$

□

Definition 3.13 [Upper bound] Let R_1, R_2 be RULC programs. We construct a new RULC program R where, for each pair t_1, t_2 of predicates defined in R_1 and R_2 respectively, a new predicate $t_1 \sqcup t_2$ is defined. We will sometimes write $(t_1 \sqcup t_2, R) = (t_1, R_1) \sqcup (t_2, R_2)$ to make it explicit to which program each of the predicates belongs.

We say that RULC clauses C, C' (defining t and t' respectively) *overlap* if, for some term w, $t(w)$ and $t'(w)$ are instances of the heads of C and C' respectively, in the sense of Definition 3.4.

1. If
$$t_i(u) \leftarrow B \in R_i$$
and this clause does not overlap with any clause defining t_j (where $\{i, j\} = \{1, 2\}$) then
$$(t_1 \sqcup t_2)(u) \leftarrow B \in R$$
and each clause of R_i relevant for a predicate occurring in B is in R.

2. If
$$t_1(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n) \in R_1$$
$$t_2(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_2$$
then
$$(t_1 \sqcup t_2)(f(x_1, \ldots, x_n)) \leftarrow (q_1 \sqcup r_1)(x_1), \ldots, (q_n \sqcup r_n)(x_n) \in R.$$

3. - Collect in $R_1' \subseteq R_1$ and $R_2' \subseteq R_2$ all the remaining clauses defining respectively t_1 and t_2 (i.e. those that do not satisfy the conditions of case 1 and 2 above). Thus for each clause $C \in R_1'$ there exits at least one clause $C' \in R_2'$ (and vice versa) such that C and C' overlap and at least one of C and C' is a clause with a constraint.
   - for each predicate q occurring in a clause body in R_i' add to R_i' the clauses of R_i relevant for q, obtaining R_i'' $(i = 1, 2)$,
   - compute $(t_1 \sqcup t_2, R'') = (t_1, ground(R_1'')) \sqcup (t_2, ground(R_2''))$ and let c be the constraint corresponding to $(t_1 \sqcup t_2)$ in R'' (notice that it is a RUL program),

- then
$$(t_1 \sqcup t_2)(x) \leftarrow c[x] \in R.$$

□

Proposition 3.14 If $(t_1 \sqcup t_2, R) = (t_1, R_1) \sqcup (t_2, R_2)$ then

$$[\![t_1]\!]_{R_1} \cup [\![t_2]\!]_{R_2} \subseteq [\![t_1 \sqcup t_2]\!]_R$$

□

3.4 Regular programs as an abstract domain

In this section we present how RULC programs can be treated as approximations of CLP programs. This is a straightforward extension of the ideas of [GdW92, GdW94]. It seems natural to view such an approach as abstract interpretation. This is however a rather unusual case of abstract interpretation. We show that most of the usually required conditions [CC92a] are not satisfied (neither by the approach of [GdW92, GdW94] nor by our generalization). This contradicts some claims of [GdW92, GdW94]. In particular, the abstract domain is not partially ordered, the abstraction function does not exist and the abstract semantics function is not monotonic, hence (although it has a fixpoint) a least fixpoint may not exist.

In our approach, the concrete domain $\mathbf{C}$ is used by the c-semantics of programs. So $\mathbf{C}$ is the set of sets of constrained atoms over the given language. (We do not 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.

For a given CLP program we want to approximate its semantics by a RULC program. So we have to relate somehow the semantics of both programs. Following [GdW92, GdW94] we introduce a distinguished (unary) predicate symbol *approx*. The set (unary relation) corresponding to *approx* is understood as the set of constrained atoms specified by the RULC program.

Definition 3.15 Let P be a CLP program and R a RULC program. Let I be a set of constrained atoms. Then R is a *regular approximation* of I if $I \subseteq [\![approx]\!]_R$. R is a *regular approximation* of P if it is a regular approximation of $M(P)$. □

Notice that the arguments of *approx* are treated both as atoms and as terms, we use here the ambivalent syntax [AB96].

Example 3.16 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 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 $\gamma : \mathbf{A} \to \mathbf{C}$ is defined as the meaning of *approx*:

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

This and 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. A pre-order generates in a standard way an equivalence relation $\cong$ and a partial ordering of the quotient set. However, even taking quotient set $\mathbf{A}/_{\cong}$ as the abstract domain we do not avoid the next problem. Namely, an abstraction function *does not exist*. (This holds already in the case of logic programs and the approach of [GdW92, GdW94]).

In the example below we show that there does not exist an abstraction function $\alpha : \mathbf{C} \to \mathbf{A}$, which is monotonic and $\alpha\gamma(Q) = Q$, for any Q. It is sufficient to consider logic programs.

Example 3.17 Consider a program P.

$$p(s(0), 0).$$
$$p(s(s(N)), NM) \leftarrow p(N,M), plus(N,M,NM).$$
$$plus(0, Y, Y).$$
$$plus(s(X), Y, s(Z)) \leftarrow plus(X,Y,Z).$$

The semantics of p in P is $[\![p]\!]_P = \{\, p(s^{2i+1}(0), s^{i^2}(0)) \mid i \geq 0 \,\}$. So the type of the second argument of p is not regular. There exists an infinite sequence

$Q_1 \succeq Q_2 \succeq \cdots$ of pairwise non-equivalent regular approximations of P, where Q_i is:

$$approx(p(X,Y)) \leftarrow t(X), u(Y).$$
$$approx(plus(X,Y,Z)) \leftarrow \ldots.$$
$$\ldots$$

such that the semantics of u is $[\![u]\!]_{Q_i} = \{\, 0, s^{1^2}(0), \ldots, s^{i^2}(0) \,\} \cup \{\, s^j(r) \mid j > i^2 \,\}$ So there does not exist a best regular approximation of $M(P)$, hence the abstraction function does not exists.

More precisely: Assume that there exists a monotonic abstraction function $\alpha : \mathbf{C} \to \mathbf{A}$ such that $\alpha\gamma(Q) = Q$, for any Q. Let $\alpha(M(P)) = R$. Program R specifies a regular set of terms as the type of the second argument of p. Thus this set contains a term r, which is not of the form $s^{i^2}(0)$. On the other hand, there exists a k such that $r \notin [\![u_k]\!]_{Q_k}$. So there exists a regular program R' such that $\gamma(R') = \gamma(R) \cap \gamma(Q_k)$, moreover $R' \preceq R$ and $R' \not\cong R$. Now, as $M(P) \subseteq \gamma(R')$, we have $R = \alpha(M(P)) \preceq \alpha\gamma(R') = R'$, contradiction. □

We want to mention that the abstract immediate consequence function $T_P^{\mathcal{A}}$, defined later on and used in type inference, may be not monotonic and that $\cong$ may be not a congruence w.r.t. $T_P^{\mathcal{A}}$.[6] The same holds for the analogical function of [GdW92, GdW94]. 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.

3.5 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.

[6] Relation $\cong$ is a congruence w.r.t. a function f iff $f(x) \cong f(y)$ whenever $x \cong y$.

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 $cl(\{\, x{\in}S[]x \mid S \subseteq \mathbb{N},\ S \text{ is finite} \,\})$.[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.

3.6 Regular term grammars

The formalism of RULC programs is not very convenient as a tool of communication between the user and the analysis system. We believe that a formalism of regular term grammars with constraints is more suitable for this purpose. It additionally provides some form of parametric types. If the parameters are not used, the formalism is equivalent to RULC programs. The user may provide to the system some (possibly parametric) type declarations. Whenever possible, the system uses the declared types in its output. For instance assume that the user has described a type $list(\alpha)$, with the expected meaning. (The details are given in Example 3.19 below). Assume also that the system derives a type t with the corresponding fragment of a RULC program

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

Then, instead of displaying the RULC clauses (or actually a corresponding grammar) the system informs that the type is $list(nat)$.

In this section we describe a restricted class of regular term grammars with constraints. The class is suitable for CLP(FD) and is used in our analysis system. It corresponds to the class of RULC programs described in the previous subsection.

[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 $anyfd(x) \leftarrow x \in 0..max$.

We build our specifications over the alphabet including a set F of function symbols, a set V of variables, a set of type symbols $\mathcal{T}$ (each of certain arity) and type variables $\mathcal{TV}$. $\mathcal{T}_0 = \{ any, anyfd, nat, neg \} \subseteq \mathcal{T}$ is a set of distinguished type symbols of arity 0. As we are going to specify sets of constrained terms the alphabet includes also constraint predicate $\in$.

We denote by $Term(S_1, S_2)$ the universe of terms built from function symbols from S_1 and variables from S_2. Elements of $Term(\mathcal{T}, \emptyset)$, are called *ground type terms*. Below we describe a notion of a grammar that from a given ground type term t generates a set of constrained terms over F and V.

A grammatical rule may include type variables. Such a rule is considered a shorthand for a possibly infinite set of its instances, where each occurrence of a type variable is replaced by a ground type term.

Definition 3.18 We consider *rules* of the form

$$t(\alpha_1, \ldots, \alpha_n) \to f(t_1, \ldots, t_k)$$
$$t(\alpha_1, \ldots, \alpha_n) \to c[x]$$

where $t \in \mathcal{T} \setminus \mathcal{T}_0$ and is of arity $n \geq 0$, $\alpha_1, \ldots, \alpha_n$ are distinct type variables, $f \in F$ and is of arity $k \geq 0$ and $t_1, \ldots, t_k \in Term(\mathcal{T}, \{\alpha_1, \ldots, \alpha_n\})$.

A *ground type substitution* is a mapping from type variables to ground type terms. A *ground rule* is the instance of a rule under a ground type substitution.

The RULC clause *corresponding* to ground rule $s \to f(s_1, \ldots, s_k)$ is $s(f(x_1, \ldots, x_k)) \leftarrow s_1(x_1), \ldots, s_k(x_k)$. The RULC clause *corresponding* to ground rule $s \to c[x]$ is $s(x) \leftarrow c[x]$. (Ground type terms play here the role of predicate symbols).

Let R be a finite set of rules. Let R' be the set of ground rules that are instances of those from R. Consider the set Q of RULC clauses corresponding to the rules of R'. R is a parametric **regular term grammar** with constraints (in normal form) if

1. any finite subset of Q is a RULC program and

2. for any ground type term s the set Q_s of those clauses of Q that are relevant for s is finite.

The set (of constrained terms over $Term(F, V)$) *generated* by a ground type term s in the grammar R is the set $[\![s]\!]_R := [\![s]\!]_{Q_s}$. □

The first of the two conditions means that for no two ground instances of rules of R the corresponding RULC clauses have a common instance of their

heads. The second condition means that any s depends on a finite set of predicates of Q. We may point out similarity of the second condition to the "reflexive condition" used by [SHK97] (in the context of prescriptive types). The sets generated by the grammar may be also defined in a natural way using a notion of a derivation. This is however outside of the scope of the present paper.

Example 3.19

Let $t1$ be a type constant, $list$ a unary type constructor and let α be a type variable. Let $a, b, [\,], | \in F$. We may define the following term grammar (where we use the standard Prolog list notation).

$$\begin{array}{l} list(\alpha) \rightarrow [\,] \\ list(\alpha) \rightarrow [\alpha | list(\alpha)] \\ t1 \rightarrow a \\ t1 \rightarrow b \end{array}$$

The RULC clauses corresponding to the last two rules are $t1(a)\leftarrow$ and $t1(b)\leftarrow$. The clauses corresponding to (the ground instances of) the first two rules are of the form $list(s)([\,])\leftarrow$ and $list(s)([x|y]) \leftarrow s(x)$, $list(s)(y)$, where s is an arbitrary ground type term. □

4 Type inference

Inferring descriptive types means computing an approximation of the semantics of a given program. In our approach the approximations are expressed as RULC programs. We are mainly interested in the call-success semantics. However we use its characterization by means of the declarative semantics of another program (Section 2.4). So the core of the method is computing a regular approximation of the declarative semantics.

Computing a regular approximation can be seen as a bottom-up abstract interpretation which for a given program P gives an over-approximation of $M(P)$. Our approach is based on [GdW92, GdW94]. We use a function $T_P^{\mathcal{A}} : \mathbf{A} \rightarrow \mathbf{A}$, 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 similar to widening [CC92a] is applied to assure that a fixpoint is reached in a finite number of steps. (It is called normalization and shortening in [GdW92] and [GdW94] respectively).

To describe the algorithm, we first present approximating a constraint, then approximating a clause and then computing $T_P^{\mathcal{A}}$.

4.1 Approximating a constraint

A constraint occurring in a program clause is a conjunction of atomic constraints. We use a constraint solver to find a kind of Cartesian projection of such constraint. (Another constraint solvers may also be used).

Example 4.1 Consider CLP(FD) and a constraint $c = 0{<}x,\ x{<}y,\ y{<}7$. A CLP(FD) solver finds that $c \rightarrow x{\in}\{1..5\}, y{\in}\{2..6\}$. Types t, s defined by a RULC program $\{\, t(x) \leftarrow x{\in}\{1..5\};\ s(y) \leftarrow y{\in}\{2..6\}\,\}$ will be used to approximate c, by the conjunction $t(x), s(y)$. □

Let $c[x_1, \ldots, x_n]$ be a constraint (with the free variables $\{x_1, \ldots, x_n\}$). If $c_1[x_1], \ldots, c_n[x_n]$ are regular constraints such that $\mathcal{D} \models c[x_1, \ldots, x_n] \rightarrow c_1[x_1], \ldots, c_n[x_n]$ then $t_1(x_1), \ldots, t_n(x_n)$ is called a **regular approximation** of c in a RULC program $\{\, t_i(x_i) \leftarrow c_i[x_i] \mid i = 1, \ldots, n\,\}$.

4.2 Approximating a clause

Now we give an algorithm that computes a function *solve*. Its arguments are a CLP clause and a RULC program and its value is a RULC program. It approximates the semantics $T^{\mathcal{C}}_{\{C\}}$ of a clause C:

$$T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C, R)).$$

Let C be $h \leftarrow c_0, b_1, \ldots, b_n$. Then $solve(C, R)$ is computed in the following steps. The data structures of the algorithm are a clause D and a RULC program Q. An invariant

$$T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq T^{\mathcal{C}}_{\{D\}}(M(Q))$$

is maintained.

1. Compute a regular approximation $t_1(x_1), \ldots, t_m(x_m)$ of c_0, in a RULC program R' (such that R and R' are predicate-disjoint). Set D as

   $$h \leftarrow t_1(x_1), \ldots, t_m(x_m), approx(b_1), \ldots, approx(b_n)$$

   and Q as $R \cup R'$.

   Since $t_1(x_1), \ldots, t_m(x_m)$ approximates c_0 and since $c[]b \in \gamma(R)$ iff $c[]approx(b) \in M(R)$, this step established the invariant.

2. Unfold clause D w.r.t. Q, by iteratively applying the following until all the body atoms are of the form $t(x)$ where x is a variable.

   Choose an atom $t(f(u_1, \ldots, u_l))$ from the body ($l \geq 0$).

   i. If there is a clause $t(f(y_1, \ldots, y_l)) \leftarrow \vec{b}$ in Q then replace $t(f(u_1, \ldots, u_l))$ by $\vec{b}\theta$, where θ is the substitution $\{ y_i/u_i \mid i = 1, \ldots l \}$.

   ii. If there is a clause $t(x) \leftarrow c'[x]$ in Q such that $c'[f(u_1, \ldots, u_l)]$ is satisfiable then let $\{z_1, \ldots, z_k\} := \mathit{Vars}(f(u_1, \ldots, u_l))$. Take new predicate symbols $s_1, \ldots, s_k$ and add to Q the clauses $\{ s_i(z_i) \leftarrow \exists_{-\{z_i\}} c'[f(u_1, \ldots, u_l)] \mid i = 1, \ldots, k \}$. Replace $t(f(u_1, \ldots, u_l))$ in the body of D by $s_1(z_1), \ldots, s_k(z_k)$.

   iii. Otherwise halt with $\mathit{solve}(C, R) = \emptyset$.

   Remove from Q all the clauses containing *approx* in the heads.

   To show that the invariant is preserved, first notice that there is at most one clause in Q which satisfies the conditions of steps (i) and (ii) above. Unfolding (i) does not change $T^{\mathcal{C}}_{\{D\}}(M(Q))$. In (ii) we have $\models c'[f(u_1, \ldots, u_l)] \rightarrow \bigwedge_{i=1}^{k} \exists_{-\{z_i\}} c'[f(u_1, \ldots, u_l)]$. So step (ii) replaces $T^{\mathcal{C}}_{\{D\}}(M(Q))$ by its superset.

3. If some variable y occurs more than once in the body of D, say as an argument of s and s', then compute $t = s \sqcap s'$, together with a program R_t, which is predicate disjoint from Q. Delete $s(y)$ and $s'(y)$ from the body of D, and put $t(y)$ instead. Add to Q the clauses of R_t. Repeat this step until each variable occurs exactly once in the body. If some intersection is empty then halt with $\mathit{solve}(C, R) = \emptyset$.

   Computing intersections of types is exact (see Proposition 3.12), hence the invariant is obviously preserved.

4. For each variable z in D, that occurs in the head but not in the body, add the atom $\mathit{any}(z)$ to the body (and clause $\mathit{any}(x) \leftarrow .$ to Q, if it is not there).

5. If some variable z occurs more than once in the head of D, where $t(z)$ occurs in the body, then replace one occurrence of z in the head by a fresh variable y, and add $t(y)$ to the body. Repeat this operation until each variable in the head occurs once.

   The current $T^{\mathcal{C}}_{\{D\}}(M(Q))$ is a superset of the previous one, therefore the invariant also holds. Now D is of the form $h' \leftarrow s_1(y_1), \ldots, s_l(y_l)$, where each y_i occurs exactly once in h' and exactly once in the body.

6. Repeat the following step until impossible.

   Choose a non variable proper subterm u of h', where $u = f(\vec{z})$, the arity of f is ≥ 0 and $\vec{z}$ is a tuple of variables. Let s be a new predicate symbol and y a new variable. Replace the clause $D = h'[u] \leftarrow B_{\vec{z}}, B$ (where $B_{\vec{z}}$ contains those atoms from the body which contain variables $\vec{z}$) by $h'[y] \leftarrow s(y), B$. Add clause $s(u) \leftarrow B_{\vec{z}}$ to Q.

   After this step the set of constrained atoms generated by the clause D and the program Q (augmented with new clauses) remains the same as after the previous step.

The result is $solve(C, R) = \{approx(h') \leftarrow B\} \cup Q$, where $D = h' \leftarrow B$.

The invariant implies the required property $T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C, R))$, because $\gamma(solve(C, R)) = [\![approx]\!]_{solve(C,R)} = T^{\mathcal{C}}_{\{D\}}(M(Q))$ (as $approx$ occurs only once in $solve(C, R)$).

4.3 Approximating a program

Definition 4.2 Let R_1 and R_2 be RULC programs and let each of them contain a definition of $approx$. We define $R_1 \coprod R_2$ as a RULC program such that $(approx, R_1) \sqcup (approx, R_2) = (approx, R_1 \coprod R_2)$. □

Notice that $\gamma(R_1) \cup \gamma(R_2) \subseteq \gamma(R_1 \coprod R_2)$.

The function $T^{\mathcal{A}}_P : \mathbf{A} \to \mathbf{A}$, which approximates the function $T^{\mathcal{C}}_P$ characterizing the c-semantics of program P, is defined by

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

The function $norm$ is used to obtain a fixpoint in a finite number of steps and will be defined below. The approach is similar to the widening of Cousot [CC92a, CC92b] and will be also called widening. The function is not monotonic but it has a property that $R \preceq norm(R)$ (see the next section). Hence $R \preceq T^{\mathcal{A}}_P(R)$.

As $T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C, R))$, we have that $T^{\mathcal{A}}_P$ indeed approximates $T^{\mathcal{C}}_P$:

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

Hence $\forall n\ T^{\mathcal{C}}_P \uparrow n \subseteq \gamma(T^{\mathcal{A}}_P \uparrow n)$. Due to widening, a fixed point of $T^{\mathcal{A}}_P$ is found in a finite number of iterations. More precisely, it is a fixpoint up to equivalence $\cong$. There exists an n such that $T^{\mathcal{A}}_P \uparrow (i+1) \cong T^{\mathcal{A}}_P \uparrow i$ for all $i > n$. We call this fixpoint the *computed fixpoint* and denote it by $T^{\mathcal{A}}_P \uparrow \omega$.

Function $T_P^{\mathcal{A}}$ is in general not monotonic w.r.t. $\preceq$ (as *norm* is not monotonic). Thus we cannot claim that the computed fixpoint is the least fixpoint.

The computed fixpoint of $T_P^{\mathcal{A}}$ approximates the c-semantics of P, as

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

4.4 Avoiding infinite loops

This section presents the widening function used in our approach to assure termination. We follow Gallagher and de Waal, however we apply the widening function used in their implementation of regular approximation tool, not the one described in [GdW92, GdW94][9].

Let $Fun(p)$ denote the set all function symbols occurring in the heads of clauses defining a predicate p.

Definition 4.3 [Relation $D(t,s)$] Let R be an RULC program containing predicates t and s $(t \neq s)$. $D(t,s)$ is true if t depends on s and $Fun(t) = Fun(s)$. □

Definition 4.4 [Widening function] Let R be an RULC program and let t and s be predicates defined in R s.t. $D(t,s)$ and $s \sqsubseteq t$ holds.

Then the program $N(R)$ is obtained from R by replacing all the occurrences of s by t in the bodies of all clauses relevant for t.

Operation N is repeated until inapplicable (i.e. no predicates t and s such that $D(t,s)$ and $s \sqsubseteq t$ occur in $N^n(R)$). Then $norm(R) = N^n(R)$. □

The following example illustrates how the widening is used during the analysis.

Example 4.5 Let P be the following program:

$$even(0).$$
$$even(s(s(X))) \leftarrow even(X).$$

First iteration of analysis results in the RULC program:

$$approx(even(X1)) \leftarrow t1(X1).$$
$$t1(0).$$

Second iteration gives:

[9] This operation is called *shortening* in [GdW94].

$approx(even(X1)) \leftarrow t2(X1).$
$t2(0).$
$t2(s(X1)) \leftarrow t3(X1).$
$t3(s(X1)) \leftarrow t1(X1).$
$t1(0).$

Next iteration produces:

$approx(even(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t2(X1).$
$t2(0).$
$t2(s(X1)) \leftarrow t3(X1).$
$t3(s(X1)) \leftarrow t1(X1).$
$t1(0).$

and after applying *norm* (as $D(t4, t2)$ and $t2 \sqsubseteq t4$) we get:

$approx(even(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t4(X1).$

Now observe, that to detect the fixpoint we need yet another iteration:

$approx(even(X1)) \leftarrow t6(X1).$
$t6(0).$
$t6(s(X1)) \leftarrow t7(X1).$
$t7(s(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t4(X1).$

Since $t6 \sqsubseteq t4$ (and $t4 \sqsubseteq t6$) widening is applied again and finally we obtain:

$approx(even(X1)) \leftarrow t6(X1).$
$t6(0).$
$t6(s(X1)) \leftarrow t7(X1).$
$t7(s(X1)) \leftarrow t6(X1).$

□

Proposition 4.6 Let R be an RULC program. Then $R \preceq norm(R)$.
PROOF: We present a proof for the case of RUL. It suffices to show that $R \preceq N(R)$. Let t and s be those predicates of R that are selected in computing $N(R)$. So $D(t,s)$ and $s \sqsubseteq t$ in R. It is sufficient to show that if $p(u) \in M(R)$ then $p(u) \in M(N(R))$ for any predicate p and any ground term u. The proof is by induction on depth m of u. If $m = 1$ then the property trivially holds.

Let depth of u be m and let the result hold for each term of depth less than m. Let u be of the form $f(u_1, \ldots, u_n)$. Assume that $p(f(u_1, \ldots, u_n)) \in M(R)$. Then there exists in R a clause $C = p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$ and $p_i(u_i) \in M(R)$, for $1 \leq i \leq n$. If $p_i = s$ then $t(u_i) \in M(R)$. By the inductive assumption, each $p_i(u_i) \in M(N(R))$ and if $p_i = s$ then $t(u_i) \in M(N(R))$. As $N(R)$ contains the clause obtained from C by replacing s in the r.h.s. by t, we have $p(f(u_1, \ldots, u_n)) \in M(N(R))$. □

Now we provide informal justification that *norm* makes the iteration of $T_P^{\mathcal{A}}{\uparrow}i$, $i = 0, 1, \ldots$, finite. Let us begin with introducing a notion of a position. A *position* is a sequence of the form $(p, i_0), (f_1, i_1), \ldots, (f_k, i_k)$, where p is a predicate name (from the analyzed program P), i_0 a number of its argument, f_j is a function symbol (from P) and i_j a number of its argument (for $j = 1, \ldots, k$). A position describes a path from a predicate (of P) to a type of a subterm of one its arguments. In other words it describes a path in a RULC program, from *approx* to a type. Any position corresponds to exactly one type (and any type relevant for *approx* has a corresponding position, possibly not unique).

Assume that we use function $T_P^{\mathcal{A}}$ without *norm*, let us call it $T'^{\mathcal{A}}_P$. Notice that programs $Q_n := T'^{\mathcal{A}}_P \uparrow n$ do not contain recursion (no predicate depends on itself). Recursion in P leads to infinite iteration of $T'^{\mathcal{A}}_P$: each Q_n is not equivalent to Q_{n+1} (for $n = 1, 2, \ldots$).

Recursion in P results in existing in Q_n (for a sufficiently large n) of types with the following property. The type t at a position π in Q_n depends on the type t' at position π in Q_m, $m < n$. This means that there is a type s in Q_n (at a position π') such that $[\![s]\!]_{Q_n} = [\![t']\!]_{Q_m}$ and t depends on s. As $Q_1 \preceq Q_2 \preceq \ldots$, we have $[\![t']\!]_{Q_m} \subseteq [\![t]\!]_{Q_n}$. Such pairs of types can be seen as the reason of the infinite iteration, at some further $Q_{n'}$, $n < n'$, there appears a type r which depends on a type equivalent to t (which depends on a type equivalent to s), etc.

Function *norm* destroys all such infinite chains, replacing such dependency chains of increasing length by recursion. Notice that for a sufficiently large n $Fun(t_n) = Fun(s_n)$, where t_n and s_n are the types at the positions π and π' respectively in Q_n. So at some stage N becomes applicable to t_n and s_n.

We conclude this section by showing that *norm* is not monotonic (and therefore $T_P^{\mathcal{A}}$ is not).

Example 4.7 Consider CLP over the Herbrand domain and RUL programs R and R':

$$R = \{\ approx(a).\quad approx(f(X)) \leftarrow s(X).\quad s(a).\quad s(f(X)) \leftarrow t(X).\quad t(a).\ \}$$

$$R' = \{\ approx(a).\quad approx(f(X)) \leftarrow s(X).\quad approx(b).\quad s(a).\quad s(f(X)) \leftarrow t(X).\quad t(a).\ \}$$

We have $R \preceq R'$. Applying *norm* to R results in

$$norm(R) = \{\ approx(a).\quad approx(f(X)) \leftarrow approx(X).\quad s(a).\quad s(f(X)) \leftarrow t(X).\quad t(a).\ \}$$

while $norm(R') = R'$. Now $norm(R) \not\preceq norm(R')$ (as $\gamma(norm(R))$ is infinite, while $\gamma(norm(R'))$ is finite). So *norm* is not monotonic. □

5 An example

A prototype of the type analyzer has been implemented. The 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).

The program below solves the well-known N-queens problem.

```
:- 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).

noattack(X,Y,K):-X #\= Y,Y #\= X+K,X #\= Y+K.
```

The `entry` declaration indicates the top goal and its call patterns for the call-success analysis. Types inferred by the system are presented below. The actual implementation of the analyzer provides a more user-friendly syntax for types, namely regular term grammars (see Section 3.6).

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

Types `natlist` and `fdlist` denote lists of `nat` and `anyfd` respectively. 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|fdlist]
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.

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 is implemented for the programming language CHIP. We expect it to be easily portable to analyze 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 system presents types to the user as regular term grammars, which are more easily comprehensible than RULC programs. It provides a restricted but useful kind of polymorphism (conf. Section 3.6)

A subject for future work is obtaining more precise analysis by using a more sophisticated treatment of constraints and implementing a richer class of types (in the present implementation a restricted class of constraints is allowed in RULC programs).

Another direction of further work is relating our technique to abstract debugging [CLV94]. A clear relationship between these two techniques should be established. The first step is presented in [CDP98, CDMP98]. We also plan to develop a tool combining the two approaches.

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.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In M. Kamkar, editor, *Proc. of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

[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.

[CC92a] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programming. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

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

[CLV94] M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.

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

[DP98] 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, 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.

[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.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.

[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.

[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.

[SHK97] J.-G. Smaus, P. Hill, and A. King. Domain constrauction for mode analysis of typed logic programs. Technical report, November 1997.

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