Inferring Numeric Relationships for Logic Program Transformation: Extended Abstract

Jonathan C. Martin[1] and Andy King[2]

[1] University of Southampton, SO17 1BJ, UK. jcm93r@ecs.soton.ac.uk
[2] University of Kent at Canterbury, CT2 7NF, UK. a.m.king@ukc.ac.uk

1 Introduction

Approximations such as affine [6, 9, 13], interval [1, 12], and polyhedral [2, 4, 5, 10] are able to describe numeric relationships between program variables and, thereby, underpin optimisations such as choice-point removal, the detection of redundant constraints, call graph simplification, and query optimisation [1]. Despite the importance of numeric approximations in the analysis of logic programs, very few analyses are powerful enough to characterise elements that occur deep within a Herbrand term, such as the elements in a list of integers for example. Some optimisations and transformations, however, require this degree of precision. In our control generation work [14], for instance, we have found that the correctness of some transforms depend on how the elements of distinct lists are related. For example, consider the quicksort program in the first column below where the control component has been generated automatically such that the program can be used in reverse mode. Then the query Qsort(c, [1, 2, 2]) gives a succinct, though not very efficient, way of generating all the combinations (rather than the permutations) of the list [1, 2, 2]. A more efficient version of the program can be obtained through (possibly automatic) program transformation by reasoning about the relationships between the integer v and the elements of the lists l and sl. We see that each element of l is strictly less than v and, since the maximal element of l is equal to the maximal element of sl, then each element of sl is also strictly less than v.

```
Qsort([], []).
Qsort([v|vs], s) ←
    Append(sl, [v|sm], s) ∧ Qsort(l, sl)
    Qsort(m, sm) ∧ Partition(vs, v, l, m).

Append([], x, x).
Append([u|x], y, [u|z]) ←
    Append(x, y, z).
```

```
Qsort([], []).
Qsort([v|vs], s) ←
    Append(sl, v, sm, s) ∧ Qsort(l, sl)
    Qsort(m, sm) ∧ Partition(vs, v, l, m).

Append([], v, x, [v|x]).
Append([u|x], v, y, [u|z]) ← u < v
    Append(x, v, y, z).
```

```
Partition([], _, [], []).
Partition([u|x], v, [u|l], m) ← u < v ∧ Partition(x, v, l, m).
Partition([u|x], v, l, [u|m]) ← u ≥ v ∧ Partition(x, v, l, m).
```

This information can be used to transform the original program into the one in the second column. The constraint $u < v$ has been (statically) propagated from the Partition predicate through the intermediate Qsort(l, sl) goal directly into the Append predicate itself. Here, the (redundant) constraint will enable early pruning of the search space when the input contains repeated elements. The extra constraint does not seriously degrade performance for goals such as $\mathsf{Qsort}(\mathsf{c}, [1, 2, 3, 4, 5, 6, 7, 8])$ but gives significant speed up for goals like $\mathsf{Qsort}(\mathsf{c}, [1, 2, 2, 4, 5, 6, 6, 6])$ and $\mathsf{Qsort}(\mathsf{c}, [1, 1, 1, 2, 2, 2, 2, 2])$. The timings for SICStus 3#3 on a 90MHz Pentium PC for all solutions to the three queries are 24.2s/24.3s, 5.40s/1.65s, and 1.30s/0.01s for the first and second versions of the program. This type of transformation lies well beyond the scope of (conjunctive) partial deduction and other techniques based on the fold/unfold methodology. To (semi–)automate this and similar transforms, it would be useful if an analyser could derive the subtle relationships such as exists between v and the elements of l and sl.

This paper contributes an analysis which is precise enough to capture numeric relationships between interpreted functors that occur deep within a Herbrand term. Besides being both useful and practical, the analysis is interesting technically since it extends the domain of polyhedra to a domain of implicitly conjoined, finite sets of linear inequality and minima constraints. The analysis associates each program variable with a (possibly empty) set of numerical values. Since the values themselves are usually unknown, each set is represented by an interval with symbolic upper and lower bounds. The use of symbolic bounds permits relationships among the bounds of different intervals to be expressed using constraints. Another novelty relates to the way empty and non-empty sets are traced. The problem is nicely explained in [1]:

> "Suppose we have three sets of terms: T_1, T_2 and T_3. Suppose also we know that all the numerical leaves that occur in T_1 are numerically less than those of T_2 which, in turn, are less than those of T_3. Can we conclude that all the numerical leaves of T_1 are less than those of T_3? The answer is yes, provided that T_2 has at least one numerical leaf."

In contrast to the solution proposed in [1] which extends abstractions with an explicit cardinality lattice, the non-empty set property is expressed at the level of interpretations. In terms of the s-semantics for CLP, this means that an interpretaion may include several constrained atoms for a single predicate. This simplifies the analysis.

The exposition is structured as follows. Section 2 introduces the notation and theory used throughout the paper. Section 3 describes the abstractions used and gives correctness results. The analysis itself is developed in section 4. Finally, section 7 presents the future work.

2 Preliminaries

2.1 General

Language The CLP framework is defined over a many-sorted first order language. Let Σ_τ (resp. Σ_f) be an alphabet of type constructor (resp. typed function) symbols which includes at least one base (resp. constant) and let $\Pi = \Pi_C \cup \Pi_P$ be an alphabet of typed predicate symbols, where Π_C and Π_P denote respectively the disjoint sets of constraint predicate symbols and programmer defined predicate symbols. Let V denote a countably infinite set of variables. The triple $L = \langle \Pi, \Sigma_f, V \rangle$ defines a many-sorted first-order language. A tuple of objects $\langle o_1, \ldots, o_n \rangle$ is denoted by $\bar{o}$, the arity of the tuple being clear from the context. The projection function π_j selects the jth element in a tuple. The rank function r returns the arity of a predicate. Any finite, non-empty subset S of a chain has a least element and a greatest element denoted $min(S)$ and $max(S)$ respectively. With reference to section 2.2 we define $min(\emptyset) = \infty$ and $max(\emptyset) = -\infty$.

2.2 Symbolic Interval Constraints

We work with a constraint algebra *Int* defined over a many-sorted first order language [3] $L_{\text{Int}} = \langle \Pi, \Sigma_f, V \rangle$. Let $\Sigma_\tau = \{\bar{\mathbb{R}}, \mathsf{interval}\}$ where $\bar{\mathbb{R}}$ represents the domain $\mathbb{R} \cup \{\infty, -\infty\}$. Then we define

$$\Sigma_f = \{0_{\langle \epsilon, \bar{\mathbb{R}} \rangle}, 1_{\langle \epsilon, \bar{\mathbb{R}} \rangle}, \infty_{\langle \epsilon, \bar{\mathbb{R}} \rangle}, -_{\langle \bar{\mathbb{R}}, \bar{\mathbb{R}} \rangle}, +_{\langle \bar{\mathbb{R}}.\bar{\mathbb{R}}, \bar{\mathbb{R}} \rangle}, \mathsf{min}_{\langle \bar{\mathbb{R}}.\bar{\mathbb{R}}, \bar{\mathbb{R}} \rangle}, \langle\rangle_{\langle \bar{\mathbb{R}}.\bar{\mathbb{R}}, \mathsf{interval} \rangle}\}$$
$$\Pi_C = \{=_{\mathsf{interval}.\mathsf{interval}}, =_{\bar{\mathbb{R}}.\bar{\mathbb{R}}}, \neq_{\bar{\mathbb{R}}.\bar{\mathbb{R}}}, \leq_{\bar{\mathbb{R}}.\bar{\mathbb{R}}}\}$$

Moreover, we define for all $x \in \bar{\mathbb{R}}$, $(x \leq \infty) \Leftrightarrow true$, $(-\infty \leq x) \Leftrightarrow true$, $(\infty \leq x) \Leftrightarrow (x = \infty)$ and $(x \leq -\infty) \Leftrightarrow (x = -\infty)$.

The constraints generated by the language L_{Int} are the linear inequality and disequality constraints over the reals together with interval and minima constraints. An interval constraint takes the form $i = \langle l, u \rangle$ and specifies that i is an interval which may be either empty or non-empty. The additional constraint $l \leq u$ determines that the interval i is non-empty and that l and u are the respective minimum and maximum (symbolic) values of the interval. That the interval i is empty, is determined uniquely by the additional constraint $l = \infty \wedge u = -\infty$. Interval constraints are restricted to be of one of these forms only. Constraints of the form $i = \langle l, u \rangle \wedge l > u \wedge (l \neq \infty \vee u \neq -\infty)$ are invalid, thus barring intervals such as $\langle 5, 4 \rangle$ or $\langle 5, -\infty \rangle$. A minima constraint has the form $z = min(x, y)$. Its semantics are defined by the following formula.

$$z = min(x, y) \Leftrightarrow ((x \leq y \Leftrightarrow z = x) \wedge (y \leq x \Leftrightarrow z = y))$$

For notational convenience, we write $z = max(x, y)$ instead of $z = -min(-x, -y)$. *Lin* will denote the set of finite sets of implicitly conjoined inequalities, and *Min* will denote the set of finite sets of minima constraints.

2.3 Semantics

The semantics of the abstract program can be expressed in an s-style semantics for constraint logic programs [3]. The semantics is parameterised over an algebraic structure, C, of constraints. We write $c \models c'$ iff c entails c', and $c \equiv c'$ iff $c \models c'$ and $c' \models c$. The interpretation base B_C for the language defined by a program P is the set of unit clauses of the form $p(\bar{x}) \leftarrow c$ quotiented by equivalence. Equivalence, $\sim$, is defined by: $p(\bar{x}) \leftarrow c \;\sim\; p(\bar{x}') \leftarrow c'$ iff $c \uparrow var(\bar{x}) \equiv (c' \wedge (\bar{x} = \bar{x}')) \uparrow var(\bar{x}')$ where $\uparrow$ denotes projection. When C corresponds to the extended Herbrand universe *EHerb*, for example, $\sim$ is variance. A π-*interpretation* I is a subset of B_C.

[3] The typing is not essential at the level of implementation, but clarifies the domain and co-domain.

Concrete and Abstract Interpretation Bases In this paper, the concrete domain is the power set of the standard Herbrand Base where interpretations are sets of ground atoms. The abstract interpretation base B_{Int} is defined as follows.

$$B_{\text{Int}} = \left\{ [p_n(\bar{x}) \leftarrow \phi]_{\sim} \;\middle|\; \begin{array}{l} p_n \in \Sigma_p, \forall i.x_i = \langle l_i, u_i \rangle \\ \phi \equiv \phi_b \wedge \phi_l \wedge \phi_m, \phi_b \equiv \bigwedge_{i=1}^{n} e_i \\ (e_i \equiv (l_i = \infty \wedge u_i = -\infty) \vee e_i \equiv (l_i \leq u_i)) \\ \phi_l \in Lin, \phi_m \in Min \end{array} \right\}$$

The pre-order on constraints lifts to π-interpretations to define a pre-ordered set $\wp(B_{\text{Int}})(\sqsubseteq)$ where $I \sqsubseteq I'$ iff $\forall [p(\bar{x}) \leftarrow \phi]_{\sim} \in I \;.\; \exists [p(\bar{x}) \leftarrow \phi']_{\sim} \in I' \;.\; \phi \models \phi'$. The pre-order $\sqsubseteq$ defines an equivalence relation: $I \approx I'$ iff $I \sqsubseteq I'$ and $I' \sqsubseteq I$ which, in turn, defines the poset $\wp(B_{\text{Int}})/\!\approx (\sqsubseteq)$ where $[I]_{\approx} \sqsubseteq [I']_{\approx}$ iff $I \sqsubseteq I'$. In fact $\wp(B_{\text{Int}})/\!\approx (\sqsubseteq)$ is a cpo.

Fixpoint Semantics The fixpoint s-semantics $\mathcal{F}_C$ of a program P is defined, as usual [3], in terms of an immediate consequence operator like so: $\mathcal{F}_C[\![P]\!] = lfp(T_P^C)$. T_P^{Int} lifts to $\wp(B_{\text{Int}})/\!\approx (\sqsubseteq)$ by $T_P^{\text{Int}}([I]_{\approx}) = [T_P^{\text{Int}}(I)]_{\approx}$ and is continuous.

3 Abstract Compilation

Through abstract compilation [9, 11] the analysis of program P reduces to evaluating the concrete semantics of an abstract program $\alpha_p(P)$. This reduces the implementation effort.

Given a tuple of terms $\langle t_1, \ldots, t_n \rangle$, we would like to capture relationships among certain subsets of the subterms of $t_1, \ldots, t_n$. Useful information can often be obtained by simply considering the maximal and minimal values of these sets. Thus it is natural to consider the abstraction of a term as a two stage process. First the term is mapped to a set of its subterms and then this set is abstracted by a symbolic interval representing the maximal and minimal values of the set.

Norms can be used to extract the subterms of interest from a term. The typed norms of [16], for example, can be generalised in a straightforward way to obtain norms which, rather than mapping to linear expressions over reals, map instead to expressions involving sets of subterms. It is well known that the derivation of norms for e.g. termination analysis is non-trivial. The norms required for this analysis are much easier to derive since we only need to consider the numeric leaves of terms.

Example 1. The typed norm $|.|_{\mathsf{List}(\mathsf{List}(\mathsf{Int}))}$ collects the set of integers which occur as leaves in a list of list of integers. It is defined in terms of the norm $|.|_{\mathsf{List}(\mathsf{Int})}$ which collects the set of integers in a list of integers.

$$\begin{aligned} |v|_{\mathsf{List}(\mathsf{List}(\mathsf{Int}))} &= v \\ |\mathsf{Nil}|_{\mathsf{List}(\mathsf{List}(\mathsf{Int}))} &= \emptyset \\ |\mathsf{Cons}(t_1, t_2)|_{\mathsf{List}(\mathsf{List}(\mathsf{Int}))} &= |t_1|_{\mathsf{List}(\mathsf{Int})} \cup |t_2|_{\mathsf{List}(\mathsf{List}(\mathsf{Int}))} \end{aligned}$$

$$|v|_{\mathsf{List(Int)}} = v$$
$$|\mathsf{Nil}|_{\mathsf{List(Int)}} = \emptyset \qquad |x|_{\mathsf{Int}} = \{x\}$$
$$|\mathsf{Cons}(t_1, t_2)|_{\mathsf{List(Int)}} = |t_1|_{\mathsf{Int}} \cup |t_2|_{\mathsf{List(Int)}}$$

The Flatten/2 predicate on the right is derived by application of the above norms to the predicate on the left.

Flatten([], []).
Flatten([[]|y], r) ←
    Flatten(y, r).
Flatten([[x|y]|z], [x|r]) ←
    Flatten([y|z], r).
	Flatten(∅, ∅).
	Flatten(y, r) ←
	    Flatten(y, r).
	Flatten({x}∪ y ∪ z, {x}∪ r) ←
	    Flatten(y ∪ z, r).

□

A typed norm can be derived for each argument position in an atom where the type of the norm corresponds to the type of the argument [16]. Since the types are not essential to this work, we omit the types from norms, writing $|t|$ to denote the application of a norm of the correct type to the term t. Note that typed norms can also be derived for non-typed programs [7]. The result of norm application to a term is a set expression which can be approximated to obtain a representation of the set in terms of its maximal and minimal values. Each set is approximated by an interval constraint and a conjunction of inequality and minima constraints.

The norms can be defined to map to set expressions in a normal form. This simplifies the subsequent interval abstraction. Thus we may assume that a set is either a variable (nothing is known about its contents or its cardinality), empty or of the form $\{e\}$ or $x \cup y$, where x and y are sets. The abstraction can then be described as a function mapping an interval $\langle l, u\rangle$, where $l, u \in V$, and a set expression S to a conjunction of constraints.

Definition 1 interval abstraction.

$$\text{int}(\langle l, u\rangle, S) \equiv \begin{cases} (l = \nu_l(v)) \wedge (u = \nu_u(v)) & \text{if } S = v \in V \\ (l = \infty) \wedge (u = -\infty) & \text{if } S = \emptyset \\ (l = e) \wedge (u = e) & \text{if } S = \{e\} \\ C & \text{if } S = x \cup y \end{cases}$$

where

$$C \equiv \left(\begin{array}{l} l = \text{min}(xl, yl) \quad \wedge \text{int}(\langle xl, xu\rangle, x) \wedge \\ u = \text{max}(xu, yu) \wedge \text{int}(\langle yl, yu\rangle, y) \end{array} \right)$$

where xl, xu, yl and yu are fresh distinct variables and $\nu_l : V \mapsto V_l$ and $\nu_u : V \mapsto V_u$ are bijective mappings to denumerable sets of variables V_l and V_u with $V_l \cap V_u = \emptyset$. □

It is now possible to formulate a program transformation in terms of a norm $|.|$ (implicitly defined over all types) and the interval abstraction function *int*.

Definition 2 program abstraction.

$$\alpha_p(\{w_1, \ldots, w_m\}) = \{\alpha_{\text{clause}}(w_1), \ldots, \alpha_{\text{clause}}(w_m)\}$$

$$\begin{array}{l} \alpha_{\text{clause}}(\ p_0(\bar{t}_0) \ \leftarrow c \wedge p_1(\bar{t}_1) \ \wedge \ldots \wedge p_n(\bar{t}_n) \ \ ) = \\ \qquad p_0(\bar{x}_0) \leftarrow c \wedge p_1(\bar{x}_1) \wedge \ldots \wedge p_n(\bar{x}_n) \bigwedge_{i=0}^{n}(\bigwedge_{j=1}^{r(p_i)} c_{ij}) \end{array}$$

where

$$\forall i \in [0,n].\forall j \in [1, r(p_i)].c_{ij} \equiv (\pi_j(\bar{x}_i) = \langle l_{ij}, u_{ij}\rangle \wedge \text{int}(\langle l_{ij}, u_{ij}\rangle, |\pi_j(\bar{t}_i)|))$$

c is a constraint and the $\pi_j(\bar{x}_i)$, l_{ij} and u_{ij} are fresh distinct variables. □

Example 2. The Flatten/2 predicate above is abstracted by α_p to give the following (simplified) abstract program.

$\mathsf{Flatten}(\infty, -\infty, \infty, -\infty).$

$\mathsf{Flatten}(\mathsf{y_l}, \mathsf{y_u}, \mathsf{r_l}, \mathsf{r_u}) \leftarrow \mathsf{Flatten}(\mathsf{y_l}, \mathsf{y_u}, \mathsf{r_l}, \mathsf{r_u}).$

$\mathsf{Flatten}(\mathsf{l_{01}}, \mathsf{u_{01}}, \mathsf{l_{02}}, \mathsf{u_{02}}) \leftarrow \mathsf{Flatten}(\mathsf{l_{11}}, \mathsf{u_{11}}, \mathsf{r_l}, \mathsf{r_u}) \wedge$
$\quad \mathsf{l_{01}} = \mathsf{min}(\mathsf{x}, \mathsf{yl}) \wedge \mathsf{u_{01}} = \mathsf{max}(\mathsf{x}, \mathsf{yu}) \wedge \mathsf{yl} = \mathsf{min}(\mathsf{y_l}, \mathsf{z_l}) \wedge \mathsf{yu} = \mathsf{max}(\mathsf{y_u}, \mathsf{z_u}) \wedge$
$\quad \mathsf{l_{02}} = \mathsf{min}(\mathsf{x}, \mathsf{r_l}) \wedge \mathsf{u_{02}} = \mathsf{max}(\mathsf{x}, \mathsf{r_u}) \wedge \mathsf{l_{11}} = \mathsf{min}(\mathsf{y_l}, \mathsf{z_l}) \wedge \mathsf{u_{11}} = \mathsf{max}(\mathsf{y_u}, \mathsf{z_u}).$

Notice that the intervals have been lifted into the heads of the clauses. This allows the desired relationships among the bounds of the intervals to be obtained by considering inter-argument relationships.

4 Symbolic Interval Analysis

The analysis of a program P proceeds by computing the concrete bottom-up semantics of the abstract program $\alpha_p(P)$. In order to derive meaningful relationships, empty and non-empty intervals must be traced separately. This is achieved by maintaining multiple constraint sets for a single predicate. These constraint sets are 'mutually exclusive' cardinality-wise. That is, given a constraint which specifies an argument to be empty and another constraint that specifies the same argument to be non-empty, these two constraints will appear in different constraint sets. It is the union of these sets that forms the complete interpretation for the predicate.

The number of atoms in an abstract interpretation can increase on each iteration. Thus to keep the interpretations manageable, several atoms can be approximated by a single atom. In [2], a convex hull approximation is used so that only one set of inequalities is needed per predicate. When dealing with sets of terms however, inaccurate information is obtained when empty and non-empty sets are over-approximated together. Hence we adapt the approach of [2] only approximating into a single set, those sets which share the same cardinality properties.

The analysis relies on the manipulation of minima constraints. These constraints have not received much attention in the literature and as far as we know, no constraint system available at present directly supports the constraint handling needed for the analysis. Thus we adopt a glass box approach based on the constraint handling rules of [8] to implement the necessary constraint manipulation.

We now sketch the abstract interpretation procedure through the analysis of the predicate Partition/4 from the introduction. Applying the program abstraction function α_p gives

$$
\begin{array}{l}
\mathsf{Partition}(\infty, -\infty, \mathsf{v}, \mathsf{v}, \infty, -\infty, \infty, -\infty).\\
\mathsf{Partition}(\mathsf{x_l}, \mathsf{x_u}, \mathsf{v}, \mathsf{v}, \mathsf{l_l}, \mathsf{l_u}, \mathsf{m_l}, \mathsf{m_u}) \leftarrow\\
\quad \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_l'}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge\\
\quad \mathsf{l_l} = \mathsf{min}(\mathsf{u}, \mathsf{l_l'}) \wedge \mathsf{l_u} = \mathsf{max}(\mathsf{u}, \mathsf{l_u'}) \wedge\\
\quad \mathsf{u} < \mathsf{v} \wedge\\
\quad \mathsf{Partition}(\mathsf{x_l'}, \mathsf{x_u'}, \mathsf{v}, \mathsf{v}, \mathsf{l_l'}, \mathsf{l_u'}, \mathsf{m_l}, \mathsf{m_u}).
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Partition}(\mathsf{x_l}, \mathsf{x_u}, \mathsf{v}, \mathsf{v}, \mathsf{l_l}, \mathsf{l_u}, \mathsf{m_l}, \mathsf{m_u}) \leftarrow\\
\quad \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_l'}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge\\
\quad \mathsf{m_l} = \mathsf{min}(\mathsf{u}, \mathsf{m_l'}) \wedge \mathsf{m_u} = \mathsf{max}(\mathsf{u}, \mathsf{m_u'}) \wedge\\
\quad \mathsf{u} \geq \mathsf{v} \wedge\\
\quad \mathsf{Partition}(\mathsf{x_l'}, \mathsf{x_u'}, \mathsf{v}, \mathsf{v}, \mathsf{l_l}, \mathsf{l_u}, \mathsf{m_l'}, \mathsf{m_u'}).
\end{array}
$$

To save space, we denote the tuple $\langle \mathsf{x_l}, \mathsf{x_u}, \mathsf{v}, \mathsf{v}, \mathsf{l_l}, \mathsf{l_u}, \mathsf{m_l}, \mathsf{m_u} \rangle$ by $\bar{t}$. We also perform trivial simplifications both to save space and also to draw attention to the more unusual constraint handling that is necessary. The ith iterate in the fixpoint computation is denoted by I_i, where $I_0 = \emptyset$ and I_{i+1} is obtained by application of the fixpoint operator to I_i.

Step 1 The calculation of each iterate begins with the application of the fixpoint operator to the current interpretation. Below, I_3 has just been obtained by application of the fixpoint operator to I_2 and has yet to be simplified. The iterates I_1 and I_2 are expressed in their simplest form. Their derivation is similar to that of I_3 which we now describe.

$$
\begin{array}{rl}
I_1 = I_0 \cup & \left\{ \bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} = \mathsf{m_l} = \infty \wedge \mathsf{x_u} = \mathsf{l_u} = \mathsf{m_u} = -\infty \right\}\\
I_2 = I_1 \cup & \left\{ \bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{x_u} = \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{x_u} < \mathsf{v} \wedge \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \right\}\\
\cup & \left\{ \bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{x_u} = \mathsf{m_l} = \mathsf{m_u} \wedge \mathsf{x_u} \geq \mathsf{v} \wedge \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \right\}\\
I_3 = I_2 \cup & \left\{ \bar{t} \,\middle|\, \begin{array}{l} \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{u} < \mathsf{v} \wedge \\ \mathsf{l_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{l_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{x_u'} < \mathsf{v} \wedge \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \end{array} \right\}\\
\cup & \left\{ \bar{t} \,\middle|\, \begin{array}{l} \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{l_u}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{l_u}) \wedge \mathsf{u} \geq \mathsf{v} \wedge \\ \mathsf{m_l} = \mathsf{min}(\mathsf{u}, \infty) \wedge \mathsf{m_u} = \mathsf{max}(\mathsf{u}, -\infty) \wedge \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{l_u} < \mathsf{v} \end{array} \right\}\\
\cup & \left\{ \bar{t} \,\middle|\, \begin{array}{l} \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{m_l}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{m_l}) \wedge \mathsf{u} < \mathsf{v} \wedge \\ \mathsf{l_l} = \mathsf{min}(\mathsf{u}, \infty) \wedge \mathsf{l_u} = \mathsf{max}(\mathsf{u}, -\infty) \wedge \mathsf{m_l} = \mathsf{m_u} \wedge \mathsf{m_u} \geq \mathsf{v} \end{array} \right\}\\
\cup & \left\{ \bar{t} \,\middle|\, \begin{array}{l} \mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{x_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{u} \geq \mathsf{v} \wedge \\ \mathsf{m_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{m_u} = \mathsf{max}(\mathsf{u}, \mathsf{x_u'}) \wedge \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \wedge \mathsf{x_u'} \geq \mathsf{v} \end{array} \right\}
\end{array}
$$

Observe that I_2 consists of 3 disjoint sets of constraints where empty and non-empty intervals are distinguished - the first set specifies that the first argument of the $\mathsf{Partition/4}$ predicate is empty ($\mathsf{x_l} = \infty \wedge \mathsf{x_u} = -\infty$) whilst in the other two sets it is seen to be non-empty ($\mathsf{x_l} = \mathsf{x_u}$). These last two sets are distinguished by the constraints over the fifth and sixth, and the seventh and eighth arguments.

Step 2 Following the application of the fixpoint operator, the first immediate task is to simplify the minima constraints within the individual sets, reducing their number and establishing some preliminary relationships. Notice in the first set of I_3 that the two constraints $\mathsf{x_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'})$ and $\mathsf{l_l} = \mathsf{min}(\mathsf{u}, \mathsf{x_u'})$ clearly imply that $\mathsf{x_l} = \mathsf{l_l}$. Such relationships can be discovered using constraint handling rules. Applying CHR rules to the second set of I_3 result in the following constraint set

$$\left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_u} \wedge \mathsf{x_u} = \mathsf{u} \wedge \mathsf{m_l} = \mathsf{u} \wedge \mathsf{m_u} = \mathsf{u} \wedge \mathsf{u} \geq \mathsf{v} \wedge \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{l_u} < \mathsf{v} \right\}$$

Step 3 The next step is to group together those sets of constraints where the emptiness/non-emptiness of intervals for every argument position coincide. Redundant constraints are also added to allow useful information to be inferred later. In addition, at this point we can project over the variables in $\bar{t}$ retaining a minima constraint $a = min(b, c)$ iff $\exists x, y, z$ such that $a = x$, $b = y$, $c = z$ and x, y and z occur in $\bar{t}$. In our example, we now have

$$\begin{array}{rl} I_3 = & \left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} = \mathsf{m_l} = \infty \wedge \mathsf{x_u} = \mathsf{l_u} = \mathsf{m_u} = -\infty \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \wedge \mathsf{x_l} = \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{x_u} = \mathsf{l_u} \wedge \mathsf{x_u} < \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{x_u} = \mathsf{l_u} \wedge \mathsf{x_u} < \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \wedge \mathsf{x_l} = \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{m_l} \wedge \mathsf{x_u} = \mathsf{m_u} \wedge \mathsf{x_l} \geq \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{m_l} \wedge \mathsf{x_u} = \mathsf{m_u} \wedge \mathsf{x_l} \geq \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{m_l} = \mathsf{m_u} \wedge \mathsf{x_u} = \mathsf{m_l} \wedge \mathsf{m_l} \geq \mathsf{v} \wedge \mathsf{l_u} < \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{m_l} = \mathsf{m_u} \wedge \mathsf{x_u} = \mathsf{m_l} \wedge \mathsf{m_l} \geq \mathsf{v} \wedge \mathsf{l_u} < \mathsf{v} \right\} \end{array}$$

Step 4 The aim now is to approximate each group of sets by a single set. This can be accomplished in our example by simply taking the convex hull of the sets within a single group. More generally, however, each set within a group may contain minima constraints which cannot be dealt with directly by the convex hull operation. Relaxing a minima constraint of the form $a = min(b, c)$ to $a \leq b \wedge a \leq c$ allows the convex hull operation to be performed, but may lose valuable information contained in the minima constraints. Thus, we proceed by first collecting those minima constraints which are common in some sense to all sets within a group. These minima constraints can then be added back into the final set resulting from the convex hull calculation. Commonality of minima constraints is naturally defined using entailment. Given sets $S_1, \ldots S_n$, suppose $a = min(b, c) \in S_i$. If $\forall i.S_i \models a = min(b, c)$ then $a = min(b, c)$ is said to be common to $S_1, \ldots S_n$.

Step 5 Having identified the common minima, all minima constraints are now relaxed. The convex hull operation is then performed as usual [2].

$$\begin{array}{rl} I_3 = & \left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} = \mathsf{m_l} = \infty \wedge \mathsf{x_u} = \mathsf{l_u} = \mathsf{m_u} = -\infty \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{x_u} = \mathsf{l_u} \wedge \mathsf{x_u} < \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{m_l} \wedge \mathsf{x_u} = \mathsf{m_u} \wedge \mathsf{x_l} \geq \mathsf{v} \right\} \\ \cup & \left\{\bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{l_l} = \mathsf{l_u} \wedge \mathsf{m_l} = \mathsf{m_u} \wedge \mathsf{x_u} = \mathsf{m_l} \wedge \mathsf{m_l} \geq \mathsf{v} \wedge \mathsf{l_u} < \mathsf{v} \right\} \end{array}$$

Step 6 At this point, the common minima can be added back into the relevant sets giving the final form for the iterate. In our example, there were no such minima so the final form is that above. To complete our example, we note that the next iteration gives the interpretation I_4 below which is, in fact the fixpoint. This information can be used to underpin the transformation described in the introduction.

$$
\begin{aligned}
I_4 = & \left\{ \bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} = \mathsf{m_l} = \infty \wedge \mathsf{x_u} = \mathsf{l_u} = \mathsf{m_u} = -\infty \right\} \\
\cup & \left\{ \bar{t} \,\middle|\, \mathsf{m_l} = \infty \wedge \mathsf{m_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{x_u} = \mathsf{l_u} \wedge \mathsf{x_u} < \mathsf{v} \right\} \\
\cup & \left\{ \bar{t} \,\middle|\, \mathsf{l_l} = \infty \wedge \mathsf{l_u} = -\infty \wedge \mathsf{x_l} \leq \mathsf{x_u} \wedge \mathsf{x_l} = \mathsf{m_l} \wedge \mathsf{x_u} = \mathsf{m_u} \wedge \mathsf{x_l} \geq \mathsf{v} \right\} \\
\cup & \left\{ \bar{t} \,\middle|\, \mathsf{x_l} = \mathsf{l_l} \wedge \mathsf{l_l} \leq \mathsf{l_u} \wedge \mathsf{m_l} \leq \mathsf{m_u} \wedge \mathsf{x_u} = \mathsf{m_u} \wedge \mathsf{m_l} \geq \mathsf{v} \wedge \mathsf{l_u} < \mathsf{v} \right\}
\end{aligned}
$$

We have identified some general constraint handling rules which are useful in the symbolic interval analysis of a variety of programs. In general, widening is required to enforce convergence of the iterates [15] [4].

5 Transformation

In this section we present a general technique for program specialisation which uses interargument relationships (in the form of constraints) as its basis.

The technique is based on the following idea. Given an interargument relationship for each atom in the body of a clause, a specialised interargument relationship may be derived for each atom by taking the conjunction of all interargument relationships projected over each individual atom. More generally, we may consider the case where each individual atom may have multiple interargument relationships as occurs in the symbolic interval analysis, and multiple specialised interargument relationships may be derived. These specialised relationships (in the form of constraints) can be used as the basis for introducing redundant constraints into the program, the aim being to move constraints "forward in time".

Definition 3 specialised interargument relationship. Let $c : H \leftarrow B_1, \ldots, B_n$ be a clause, and for all $i \in [1, n]$, let C_i be a set of valid interargument relationships for B_i. Then the set of specialised interargument relationships S_i for B_i wrt c is given by

$$\{c_1 \wedge \ldots \wedge c_n \uparrow vars(B_i) \mid \text{ for all } i \in [1, n], c_i \in C_i \text{ and } c_1 \wedge \ldots \wedge c_n \not\equiv \text{false}\}$$

Observe that each interargument relationship $s_i \in S_i$, is specialised in the sense that for some $c_i \in C_i$, $s_i \models c_i$, though the converse is not necessarily true. Our interest in the specialised constraint s_i is that it captures some new information about the specific call B_i. The problem is how best to identify this information. Restricting ourselves to a single interargument relationship per atom for the moment, as a first approximation, we may begin by noting which of the primitive constraints of s_i are entailed by c_i. More specifically, suppose that s_i is of the form $a_1 \wedge \ldots \wedge a_k$, where for all $j \in [1, k]$, a_j is an atomic (or primitive) constraint. Then if $c_i \not\models a_j$, then a_j is 'new information' which may be used to specialise the call B_i. A refinement of this approach would be to consider which atomic constraints of s_i are entailed by $c_1 \wedge \ldots \wedge c_i$ or even entailed by $s_1 \wedge \ldots \wedge s_{i-1} \wedge c_i$.

[4] Submitted to ILPS'97

6 Related Work

In broad terms, analyses for inferring numeric relationships have either been formulated in terms of affine sub-spaces, intervals or polyhedra.

Affine Approximations In the affine approach [6, 9, 13], numeric relationships are represented by affine subspaces, basically points, lines or hyper-planes in $\mathbf{R}^n$, which can be represented and manipulated using matrices. The affine approach is attractive because the approximation is Noetherian and therefore termination is not an issue, but inequalities and minima relationships cannot be inferred.

Interval Approximations CHINA is an analyser for CLP which infers numeric relationships between variables using intervals. In [1, Chapter 4] a "structural information analysis" is described for handling interpreted functors that occur as the leaves of Herbrand terms. The depth-k abstractions, however, though useful, cannot capture properties of interpreted functors that occur deep in recursive data types such as lists. In [1, Chapter 5] a "range and relations analysis" is presented for approximating sets of numbers with interval abstractions. A cardinality lattice $\{\perp, 0, 1, 01, 1, 1^+, 0^+\}$ is used to infer which sets of numbers are non-empty. In our approach, however, the non-empty property is expressed at the level of interpretations. This removes the need for a cardinality lattice thereby simplifying the analysis. To be fair, [1, Chapter 5] explains how some interesting interval abstractions, like bounded and quotiented differences, can be applied to CLP. The intervals used, however, are not symbolic and minima relationships cannot be expressed.

Polyhedral Approximations Polyhedral approximations require widening to obtain stability of the iterates. The widenings of [2, 4, 10, 17] all refine the basic widening of [5] comparing successive polyhedra, P_i and P_{i+1}, to basically throw away the inequalities of P_i that are not satisfied by P_{i+1}. To reduce the loss of information, the widening of [10] reformulates the representation of P_i in order to maximise the number of common constraints. The widening of [4], on the other hand, first selects those inequalities of P_i that entail P_{i+1}. This gives $P_i{}'$. Second, it finds those inequalities which hold for both P_i and P_{i+1} and yet are not explicit in $P_i{}'$. This gives $P_{i+1}{}'$ and the widening $P_i{}' \cup P_{i+1}{}'$. The so-called affine widening of [17] is simple but compromises the expressiveness of polyhedra since it "cannot infer relationships in the form of inequality" [17]. Finally, the widening of [2] improves precision by delaying the application of widening for a few, typically one or two, iterations.

The widening used in this paper is a natural extension of the delay widening of [2] modified to deal with multiple constraint sets for individual predicates. Interestingly, the analyses of [2, 4, 5, 10, 17] all derive invariants among program variables – relationships between tuples of numbers. Prior to our work, polyhedral approximations have not been applied to infer relationships between tuples of sets of numbers.

7 Conclusion and Future Work

We have presented a new analysis which is precise enough to capture numeric relationships that occur deep within Herbrand terms. The novelties of the analysis include the use of symbolic intervals, the way sets are traced and the ability to express maxima and minima relationships.

Future work will focus on implementation, the ECLiPSe system being a likely target for initial prototyping due to its support of constraint handling rules. The analysis, as currently formulated, is rather complex and it is not clear where the performance bottlenecks will occur in the implementation. Profiling experiments are required. Also, recent experimental work on a related analysis [2], has shown that polyhedral abstractions can often infer deep and unexpected numeric relationships. Indeed, a handworking of the analysis presented in this paper on a finite domain constraint program discovered a surprising numeric relationship that, when added to the program (as a redundant constraint), led to a five fold speedup. Another role of the implementation will therefore be to find whether useful redundant constraint can be inferred for other programs.

References

1. R. Bagnara. *Dataflow Analysis for Constraint Logic Based Languages*. PhD thesis, Università di Pisa, 1997.
2. F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathcal{R}$). In *LOPSTR'96*. Springer-Verlag, 1996.
3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *Journal of Logic Programming*, 1991.
4. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP'92*. Springer-Verlag, 1992.
5. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL'78*, pages 84–97, 1978.
6. D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(2):117–154, 1995.
7. S. Decorte, D. de Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In *ILPS'93*, pages 420–436, 1993.
8. T. Frühwirth. Constraint simplification rules. Technical Report ECRC-92-18, ECRC, 1992.
9. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 3(25):191–248, 1995.
10. N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Universit'e scientifique et médicale de Grenoble, 1979. Thèse de 3 ème d'informatique.
11. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *JLP*, 13(4):349–366, 1992.
12. G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numeric values in CLP(H,N). In *PLILP'94*, pages 400–414. Springer-Verlag, 1994.
13. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
14. J. Martin and A. King. Generating Efficient, Terminating Logic Programs. In *TAPSOFT'97*. Springer-Verlag, 1997.
15. J. Martin and A. King. Symbolic Interval Analysis for Logic Programs. Technical Report DSSE-TR-97-1, University of Southampton, UK, 1997.
16. J. Martin, A. King, and P. Soper. Typed Norms for Typed Logic Programs. In *LOPSTR'96*, pages 224–238. Springer-Verlag, 1996.

17. K. Sohn. Constraints among Argument Sizes in Logic Programs. In *PODS'94*, pages 68–74. ACM Press, 1994.

This article was processed using the LaTeX macro package with LLNCS style