Extended Set Constraints and Tree Grammar Abstraction of Programs*

Mads Rosendahl John P. Gallagher
Roskilde University, Building 43.2, DK-4000 Roskilde, Denmark
madsr@ruc.dk jpg@ruc.dk

Set constraints are relations between sets of ground terms or trees. This paper presents two main contributions: firstly we consider an extension of the systems of set constraints to include a tuple constructor, and secondly we construct a simplified solution procedure for set constraints. We illustrate the approach using logic programs and show that we can construct various set-based abstractions of a program including the $\mathscr{T}_P$ abstraction using a system of set constraints derived in a single pass over the program.

1 Introduction

Set constraints are relations between sets of ground terms or trees. In the first part of the paper we consider an extension of the systems of set constraints used by Heintze and Jaffar [17] to analyse logic programs, which can in turn be seen as extensions of the techniques used earlier to analyse LISP-like languages [27], [21]. We introduce a tuple constructor expression in set constraints and show that constraints with such expressions are satisfiable and solvable. We provide a simple yet efficient solution procedure where standard techniques from fixpoint iteration may be used.

With these expressions it is possible in a single pass of the program to generate set constraints whose solution is the $\mathscr{T}_P$ interpretation of logic programs. We also show how to generate constraints that are more precise than the $\mathscr{T}_P$ interpretation and which cannot directly be expressed with membership expressions [9].

2 Set constraints

In this section we describe a version of definite set constraints as used by Heintze and Jaffar [17]. In a later section we extend set constraints with a new expression, called a tuple constructor.

The set of ground terms (or trees) is built from function symbols (or constructors) in a set Σ. Each function symbol f in Σ has a rank or arity greater than or equal to zero. Zero-ary functions are also called constants. The set of ground terms is denoted G and is the least solution to the equation $\mathsf{G} = \{f(t_1,\ldots,t_n) | t_i \in \mathsf{G} \wedge f \in \Sigma\}$.

Set constraints systems are collections of equations of the form

$$cs ::= \; V = e \,, \ldots, V = e$$

where the left hand side is a variable V from a set $\mathscr{V}$ and the right hand side is a set expression. A variable symbol can at most appear on the left hand side of one equation. The syntax of set expressions can be defined as follows.

*Work supported by the Danish Natural Science Research Council (FNU) in the project SAFT: Static Analysis with Finite Tree Automata (FNU-272-06-0574).

Submitted to:
WLPE 2011

$$e ::= \bot \mid \top \mid V \mid e \cap e \mid e \cup e \mid f(V,\ldots,V) \mid f^{-1}_{(i)}(e)$$

In examples variables are written in *italics* and functions in a `typewriter` font. Zero-ary functions will typically be written without extra parentheses.

Expressions are here written in so-called *normalized* form [6] where arguments to functions are set variables. In examples we will allow composite expressions inside functions. They can easily be transformed into the desired form by the introduction of new variables.

A set constraint system is said to be *satisfiable* if there exists an assignment of sets to variables that satisfies the constraints. The set interpretation below will construct such an assignment. If set constraints are extended with complement or general set difference expressions then set constraints are not guaranteed to be satisfiable. Set constraints are said to be *solvable* if they can be transformed into a tree grammar. The tree grammar interpretation in the next section shows they are also solvable.

Set interpretation of set constraints. The meaning of a set constraint system is defined as the least solution to the system. This is here defined as a function M which for a set constraint system returns a mapping of variables to sets of ground terms. Notice that all operations are monotonic and that the least fixpoint is well-defined for all constraint systems.

$$\mathsf{M}[\![cs]\!] : \mathscr{V} \to \wp(\mathsf{G})$$
$$\mathsf{E}[\![e]\!] : (\mathscr{V} \to \wp(\mathsf{G})) \to \wp(\mathsf{G})$$

$$\mathsf{M}[\![V_1 = e_1,\ldots,V_n = e_n]\!] = \mathsf{lfp}\lambda\rho.[V_1 \mapsto \mathsf{E}[\![e_1]\!]\rho,\ldots,V_n \mapsto \mathsf{E}[\![e_n]\!]\rho]$$
$$\mathsf{E}[\![V]\!]\rho = \rho(V)$$
$$\mathsf{E}[\![\top]\!]\rho = \mathsf{G}$$
$$\mathsf{E}[\![\bot]\!]\rho = \emptyset$$
$$\mathsf{E}[\![e_1 \cup e_2]\!]\rho = \mathsf{E}[\![e_1]\!]\rho \cup \mathsf{E}[\![e_2]\!]\rho$$
$$\mathsf{E}[\![e_1 \cap e_2]\!]\rho = \mathsf{E}[\![e_1]\!]\rho \cap \mathsf{E}[\![e_2]\!]\rho$$
$$\mathsf{E}[\![f(V_1,\ldots,V_n)]\!]\rho = \{f(t_1,\ldots,t_n) \mid t_i \in \rho(V_i)\}$$
$$\mathsf{E}[\![f^{-1}_{(i)}(e)]\!]\rho = \{t_i \mid f(t_1,\ldots,t_n) \in \mathsf{E}[\![e]\!]\rho\}$$

For a set constraint system $cs = V_1 = e_1,\ldots,V_n = e_n$ we define

$$\mathsf{M}_{cs}[\![cs]\!] = \bigcup_i \mathsf{M}[\![cs]\!](V_i)$$

so that $\mathsf{M}_{cs}[\![cs]\!]$ has type $\wp(\mathsf{G})$.

3 Tree grammar interpretation

Tree grammars form a sub-language of set constraints where we do not have intersections or projections.

The aim of this section is to show how to translate (or solve) a system of set constraints into a tree grammar. The transformation is here done as an interpretation of the set constraint system over a domain of tree grammar expressions.

The transformations may generate new variables - in the worst case one for every subset of variables in the constraint system. Instead of working with dynamically generated variable names we will generate a tree grammar as a mapping of sets of variables to tree grammar expressions. A set of variables represents the intersection of the meaning of each variable in the set. A tree grammar expression is either the top symbol or a set of constructors with sets of variables as arguments.

$$\mathsf{D} : \{\top\} \cup \{f_j(s_1,\ldots,s_n) \mid s_i \subseteq \mathscr{V}, s_i \neq \emptyset\}$$

Traditionally the transformation to tree grammars is done as a sequence of rewrites of the system where one adds new expressions and a final removal of expressions with projections and intersections. Here we will present the transformation as direct interpretation of set expressions as tree grammar expressions. This interpretation is expressed as fixpoint over a cpo with finite height and this means that standard techniques from implementation of abstract interpretation can be utilized.

The transformation of set constraints to tree grammars is done using the S function. In the interpretation we only include constructor expressions as results if all arguments are non-empty. This means that test of non-emptiness is included in intersection and constructor expressions. In the Heintze and Jaffar implementation the test is made in the projection where one only projects arguments out of constructor expressions if all other arguments in the expression are non-empty.

$$\mathsf{S}[\![V_1 = e_1, \ldots, V_n = e_n]\!] : \wp(\mathscr{V}) \to \wp(\mathsf{D})$$
$$\mathsf{R}[\![e]\!] : (\wp(\mathscr{V}) \to \wp(\mathsf{D})) \to \wp(\mathsf{D})$$

$$\begin{array}{l}\mathsf{S}[\![V_1 = e_1, \ldots, V_n = e_n]\!] = \\ \mathsf{lfp}\lambda\rho.[\emptyset \mapsto \emptyset, \{V_1\} \mapsto \mathsf{R}[\![e_1]\!]\rho, \ldots, \{V_n\} \mapsto \mathsf{R}[\![e_n]\!]\rho, \\ \quad \{V_1, V_2\} \mapsto \mathsf{intersect}(\rho(V_1), \rho(V_2)), \{V_1, V_3\} \mapsto \mathsf{intersect}(\rho(V_1), \rho(V_3)), \ldots]\end{array}$$

where

$$\begin{array}{l}\mathsf{R}[\![V]\!]\rho = \rho(\{V\}) \\ \mathsf{R}[\![\top]\!]\rho = \{\top\} \\ \mathsf{R}[\![\bot]\!]\rho = \emptyset \\ \mathsf{R}[\![e_1 \cup e_2]\!]\rho = \mathsf{R}[\![e_1]\!]\rho \cup \mathsf{R}[\![e_2]\!]\rho \\ \mathsf{R}[\![e_1 \cap e_2]\!]\rho = \mathsf{intersect}(\mathsf{R}[\![e_1]\!]\rho, \mathsf{R}[\![e_2]\!]\rho) \\ \mathsf{R}[\![f(V_1, \ldots, V_n)]\!]\rho = \{f(\{V_1\}, \ldots, \{V_n\}) \mid \forall j.\rho(\{V_j\}) \neq \emptyset\} \\ \mathsf{R}[\![f^{-1}_{(i)}(e)]\!]\rho = \mathsf{project}(f, i, \mathsf{R}[\![e]\!]\rho)\end{array}$$

and

$$\begin{array}{l}\mathsf{intersect}(\{\top\}, d) = d \\ \mathsf{intersect}(d, \{\top\}) = d \\ \mathsf{intersect}(\{d_1, d_2 \ldots\}, d) = \mathsf{intersect}(\{d_1\}, d) \cup \cdots \cup \mathsf{intersect}(\{d_n\}, d) \\ \mathsf{intersect}(d, \{d_1, d_2 \ldots\}) = \mathsf{intersect}(d, \{d_1\}) \cup \cdots \cup \mathsf{intersect}(d, \{d_n\}) \\ \mathsf{intersect}(\{f(u_1, \ldots, u_n)\}, \{g(w_1, \ldots, w_n)\}) \\ \quad = \{f(u_1 \cup w_1, \ldots, u_n \cup w_n) \mid \forall i : \rho(u_i \cup w_i) \neq \emptyset \wedge f = g\}\end{array}$$

$$\mathsf{project}(f, i, d) = (\{\top\} \textbf{ if } \top \in d) \cup \{\rho(u_i) \mid f(u_1, \ldots, u_n) \in d\}$$

Correctness of solver. We will use the set interpretation M of set constraint systems on tree grammars since tree grammars form a sublanguage of set constraints. In this case M will have type $\wp(\mathscr{V}) \to \wp(\mathsf{G})$.

$$\mathsf{M}_{cs}[\![cs]\!] = \mathsf{M}_{cs}[\![\mathsf{S}[\![cs]\!]]\!]$$

4 Extending set constraints

Certain set operations are not directly expressible as set constraints. Assume we have the environment $\rho = X \mapsto \{\mathtt{f}(\mathtt{a},\mathtt{b}), \mathtt{f}(\mathtt{b},\mathtt{a})\}$ and want to write a set constraint that restricts another variable Y to be the same as X except that the function symbol $\mathtt{f}$ is changed to $\mathtt{g}$. The obvious constraint would be

$$Y = \mathtt{g}(\mathtt{f}^{-1}_{(1)}(X), \mathtt{f}^{-1}_{(2)}(X))$$

This, however, will have the solution $\{\mathtt{g(a,b)},\mathtt{g(b,a)},\mathtt{g(a,a)},\mathtt{g(b,b)}\}$ and the relationship between the arguments is lost.

In this section we introduce a new set expression, called a tuple constructor: $f(e_1,\ldots,e_n)[V]$. It behaves essentially as an ordinary constructor expression except that values of a selected variable (V) are treated independently. We will show that set constraint systems extended with tuple constructor expressions are still satisfiable and solvable. The set interpretation of the tuple constructor expression is fairly straightforward:

$$\mathsf{E}[\![f(e_1,\ldots,e_n)[V]]\!]\rho = \bigcup_{s\in\rho(V)} \{f(t_1,\ldots,t_n) \mid t_i \in \mathsf{E}[\![e_i]\!]\rho[V\mapsto\{s\}]\}$$

In the remainder of the paper we will assume that the set interpretation M of set constraints is extended with this interpretation of tuple constructor expressions.

If we consider the example above then we could write the constraint as

$$Y = \mathtt{g}(\mathtt{f}^{-1}_{(1)}(X),\mathtt{f}^{-1}_{(2)}(X))[X]$$

This constraint will have the solution $\{\mathtt{g(a,b)},\mathtt{g(b,a)}\}$. The tuple constructor will in general return a result which is a subset of the ordinary constructor expression.

We will impose certain restrictions on the possible expressions that can appear inside tuple constructors. The expressions should be intersections of simple variables and projections of the selected variable. We further assume that the expressions only project disjoint parts of the selected variable. The context free part of this restriction can be expressed as follows:

$$e ::= f(t,\ldots,t)[V] \qquad t ::= t\cap t \mid u \qquad u ::= V \mid f^{-1}_{(i)}(u)$$

Constraint solver. The basic idea in the constraint solver is to expand out the value of the selected variable so that the projections in the arguments can be evaluated as sets of variables. Each argument in the tuple constructor expression can then be represented as an intersection of simple variables. This makes it possible to represent the result in the domain D of tree grammar expressions. We extend the solver S to the extended set constraints by extending the R function to tuple constructor expressions.

$$\begin{array}{l}
\mathsf{R}[\![f(e_1,\ldots,e_n)[V]]\!]\rho = \\
\quad \mathsf{tuple}(f,[e_1,\ldots,e_n],\rho,V,\mathsf{expand}(e_1\cap\ldots\cap e_n,\rho(V),V,\rho))
\end{array}$$

$$\begin{array}{l}
\mathsf{expand}(e_1\cap e_2,d,V,\rho) = \mathsf{expand}(e_1,\mathsf{expand}(e_2,d,V,\rho),V,\rho) \\
\mathsf{expand}(e,e_1\cup e_2,d,V,\rho) = \mathsf{expand}(e,e_1,V,\rho)\cup\mathsf{expand}(e,e_2,V,\rho) \\
\mathsf{expand}(f^{-1}_{(i)}(V),d,V,\rho) = d \\
\mathsf{expand}(W,d,V,\rho) = d \\
\mathsf{expand}(f^{-1}_{(i)}(e),\{f(t_1,\ldots,t_{i-1},s,t_{i+1},\ldots,t_n)\},V,\rho) \\
\quad = \{f(t_1,\ldots,t_{i-1},d,t_{i+1},\ldots,t_n) \mid d\in\mathsf{expand}(e,\rho(s),V,\rho)\} \\
\mathsf{tuple}(f,[e_1,\ldots,e_n],\rho,V,d_1\cup d_2) \\
\quad = \mathsf{tuple}(f,[e_1,\ldots,e_n],\rho,V,d_1)\cup\mathsf{tuple}(f,[e_1,\ldots,e_n],\rho,V,d_2) \\
\quad \mathsf{tuple}(f,[e_1,\ldots,e_n],\rho,V,d) = \textbf{let}\, s_i = \mathsf{select}(e_i,V,d)\ \textbf{in} \\
\qquad \textbf{if}\,\forall i.\rho(s_i)\neq\emptyset\ \textbf{then}\,\{f(s_1,\ldots,s_n)\}\ \textbf{else}\,\emptyset \\
\quad \mathsf{select}(W\cap e,V,d) = \{W\}\cup\mathsf{select}(e,V,d) \\
\quad \mathsf{select}(e\cap W,V,d) = \{W\}\cup\mathsf{select}(e,V,d) \\
\quad \mathsf{select}(f^{-1}_{(i)}(e),f(t_1,\ldots,t_n)) = \mathsf{select}(\{e\},t_i) \\
\quad \mathsf{select}(W,V,d) = \{W\}
\end{array}$$

The auxiliary function expand expands the value of the selected variable so that each projection in the argument to the tuple constructor represents a set of variables. The select function takes an argument to the tuple constructor and the expanded version of the selected variable and returns a set of variables.

As an example consider the expression

$$\mathtt{f}(\mathtt{h}^{-1}_{(1)}(\mathtt{g}^{-1}_{(1)}(V)) \cap \mathtt{g}^{-1}_{(2)}(V))[V]$$

in an environment where $\{V\} \mapsto \{\mathtt{g}(X,Y), \mathtt{g}(X,Z)\}$ and $\{X\} \mapsto \{\mathtt{h}(A), \mathtt{h}(B)\}$. We assume further that all other variables and their intersections have non-empty values. The expand function will expand the value of V so that the projected values are represented as simple sets of variables. In this case we obtain:

$$\{V\} \mapsto \{\mathtt{g}(\mathtt{h}(A),Y), \mathtt{g}(\mathtt{h}(B),Y), \mathtt{g}(\mathtt{h}(A),Z), \mathtt{g}(\mathtt{h}(A),Z)\}$$

The value of the tuple constructor expression is then

$$\{\mathtt{f}(\{A,Y\}), \mathtt{f}(\{B,Y\}), \mathtt{f}(\{A,Z\}), \mathtt{f}(\{B,Z\})\}$$

Correctness. Constraints that contain the new tuple constructor expressions are still satisfiable and solvable: $\mathsf{M}_{cs}[\![cs]\!] = \mathsf{M}_{cs}[\![\mathsf{S}[\![cs]\!]]\!]$

5 Set constraint solver implementation

The tree grammar interpretations above have been implemented essentially as described here. We have also implemented the set constraint interpretations described later and the efficiency when applied to constraints derived from logic programs seems comparable to [13].

The problem of solving set constraints is here formulated as a fairly standard second-order fixpoint iteration problem. The fixpoint operation used in the solver has type:

$$\mathsf{lfp} : (\wp(\mathscr{V}) \to \wp(\mathsf{D})) \to (\wp(\mathscr{V}) \to \wp(\mathsf{D}))$$

In the worst-case scenario one would have to evaluate and iterate this function for all subsets of $\mathscr{V}$. In practice we compute the minimal function graph [22] reachable from the set of reachable calls consisting of all the singleton sets of variables.

Demand-driven versions of the solver. The fact that the fixpoint is found as a minimal function graph also means that we can create demand driven versions of the problem. If we only are interested in a single variable then a minimal function graph implementation of the fixpoint operation will create the tree grammar describing that variable and the variables it may depend on.

Evaluation strategy. The main problem with second-order fixpoint iteration is that function application $\rho(x)$ is not monotonic when the environment ρ is not monotonic. A straightforward selective reevaluation of the environment will then not necessarily form an ascending sequence. There are two main approaches to secure correctness and termination of second-order iteration:

- Chaotic iteration sequence [7]. This is based on the principle that a fair reevaluation strategy will reach the fixpoint.
- Semi-monotonic iteration [10]. Reevaluation of the environment from a sub-fixpoint of an argument and all its needed arguments of a semi-monotonic functional is stable if and only if it is the fixpoint for those arguments.

The literature contains a number of different approaches to scheduling reevaluation during fixpoint iteration. Internally such strategies may use worklists or neededness graphs to find arguments for which the functional should be reevaluated. It is a trade-off between the cost of reevaluation and the cost of maintaining dependencies so that one only reevaluates when dependent values have changed. In the examples we have considered, we found that the most efficient algorithm was a truncated depth first evaluation strategy [28] where in each iteration one evaluates depth first, but uses the result from the last iteration or evaluation if the evaluation of the same argument is attempted more than once.

6 Interpretations of logic programs

We will now present a small language of logic programs and two interpretations: T_P and $\mathscr{T}_P$.

The language of logic programs can be described with the following grammar

$$\begin{array}{lll} prg & : cl_1 \cdots cl_n & \\ cl & : pred :\!\text{-}\ pred_1, \ldots, pred_n. & n \geq 0 \\ pred & : p(term_1, \ldots, term_n) & n > 0 \\ term & : V_i \mid f(term_1, \ldots, term_n) & n \geq 0 \end{array}$$

we assume that variables are not shared between clauses. We will further assume that clauses with non-empty right hand side contains at least one variable on the left hand side. It is fairly easy to transform a program so that these conditions is satisfied. A clause of the form "`p(a):-q(b).`" can be transformed to "`p(X):-q(b),isA(X).  isA(a).`" where the rank of each predicate symbol has been retained.

Standard semantics T_P. The set G is the Herbrand universe and GP the Herbrand base. Substitutions are ordered with $\mho$ (fail) as bottom element and $\lambda x.\top$ as top element. The standard interpretation returns the success set as a subset of GP

$$\begin{array}{l} \mathsf{T}[\![cl_1 \cdots cl_n]\!] = \mathsf{lfp}\lambda\sigma.\mathsf{U}[\![cl_1]\!]\sigma \cup \cdots \cup \mathsf{U}[\![cl_n]\!]\sigma \\ \mathsf{U}[\![H :\!\text{-}\ B_1, \ldots B_n]\!]\sigma = (\mathsf{mgu}(H, \mathsf{GP}) \sqcap \mathsf{mgu}(B_1, \sigma) \sqcap \cdots \sqcap \mathsf{mgu}(B_n, \sigma))(H) \end{array}$$

$$\begin{array}{l} \mathsf{mgu}(t, \sigma) = \{\mathsf{mgu}'(t, s) \mid s \in \sigma\} \backslash \{\mho\} \\ \mathsf{mgu}'(V, s) = \lambda x.\ \textbf{if}\, x = V\ \textbf{then}\, s\ \textbf{else}\, \top \\ \mathsf{mgu}'(c, s) = \ \textbf{if}\, c = s\ \textbf{then}\ \lambda x.\top\ \textbf{else}\, \mho \\ \mathsf{mgu}'(f(t_1, \ldots, t_n), g(s_1, \ldots, s_n)) = \\ \qquad \textbf{if}\, f = g\ \textbf{then}\ \mathsf{mgu}'(t_1, s_1) \sqcap \cdots \sqcap \mathsf{mgu}'(t_n, s_n)\ \textbf{else}\, \mho \end{array}$$

Set-based semantics $\mathscr{T}_P$. This is the least set based model [16] of a logic program. The meaning of the right hand sides are abstracted as set substitutions before being applied to the left hand side.

$$\begin{array}{l} \mathscr{T}[\![cl_1 \cdots cl_n]\!] = \mathsf{lfp}\lambda\sigma.\mathsf{U}^\tau[\![cl_1]\!]\sigma \cup \cdots \cup \mathsf{U}^\tau[\![c_n]\!]\sigma \\ \mathsf{U}^\tau[\![H :\!\text{-}\ B_1, \ldots, B_n]\!]\sigma = (\mathsf{topmap}(H) \sqcap \alpha(\mathsf{mgu}(B_1, \sigma) \sqcap \cdots \sqcap \mathsf{mgu}(B_n, \sigma))(H) \end{array}$$

$$\begin{array}{l} \mathsf{topmap}(H) = \lambda x.\ \textbf{if}\, x \in \mathsf{freevar}(H)\ \textbf{then}\, \mathsf{G}\ \textbf{else}\, \top \\ \alpha(\emptyset) = \mho \\ \alpha(\phi) = \lambda x.\ \textbf{if}\, x \in \mathsf{dom}\phi\ \textbf{then}\, \{\theta(x) \mid \theta \in \phi\}\ \textbf{else}\, \top \end{array}$$

Relation. The set-based interpretation gives a safe approximation of the success sets of the program: $\mathsf{T}[\![prg]\!] \subseteq \mathscr{T}[\![prg]\!]$

7 Set constraint construction

In this section we will describe three different ways of constructing set constraints from a logic program. They will not be defined as a fixpoint but as a direct construction and collection of constraints from terms.

Operations on set constraints. Simple set constraints are formed using the notation $\ulcorner V = e \urcorner$ where e is a set constraint expression. For two constraint systems cs_1 and cs_2 we can construct new constraint systems $cs_1 \sqcup cs_2$ and $cs_1 \sqcap cs_2$ where we create the union or the intersection of the expressions for the shared variables and preserve constraints that only appear in one of the constraint systems. For a term t we let p_t denote the outer predicate/functor symbol. E.g. $p_{rev(nil,nil)}$ is rev.

Heintze and Jaffar style constraints. We here show the construction of constraints described in [15].

$$\mathsf{C}^h[\![cl_1 \cdots cl_n]\!] = \mathsf{R}^h[\![cl_1]\!] \sqcup \cdots \sqcup \mathsf{R}^h[\![c_n]\!]$$
$$\mathsf{R}^h[\![H :\!- B_1, \ldots, B_n]\!] = \ulcorner p_H = H \urcorner \sqcap \mathsf{sc}(H, \top) \sqcap \mathsf{sc}(B_1, p_{B_1}) \sqcap \cdots \sqcap \mathsf{sc}(B_n, p_{B_n})$$

$$\mathsf{sc}(V, u) = \ulcorner V = u \urcorner$$
$$\mathsf{sc}(f(t_1, \ldots, t_n, u) = \mathsf{sc}(t_1, f^{-1}_{(1)}(u)) \sqcap \cdots \sqcap \mathsf{sc}(t_n, f^{-1}_{(n)}(u))$$

Relation. For a logic program P, let $\mathsf{M}_P[\![cs]\!]$ denote the application of $\mathsf{M}[\![cs]\!]$ to all set variables corresponding to predicate symbols in the program P. We have $\mathscr{T}[\![P]\!] \subseteq \mathsf{M}_P[\![\mathsf{C}^h[\![P]\!]]\!]$

$\mathscr{T}_P$ equivalent constraints. In the $\mathscr{T}_P$ equivalent constraints we create constraints that describe the set of substitutions that satisfy the right-hand side of the clause. A set of substitutions is represented as a terms with an argument for each variable and constant in the clause.

$$\mathsf{C}^\tau[\![cl_1 \cdots cl_n]\!] = \mathsf{R}^\tau[\![cl_1]\!] \sqcup \cdots \sqcup \mathsf{R}^\tau[\![c_n]\!]$$
$$\mathsf{R}^\tau[\![H :\!- B_1, \ldots, B_n]\!] = \ulcorner p_H = H \urcorner \sqcap \mathsf{prjvars}(vl, hb) \sqcap$$
$$\quad \mathsf{mgu}^\tau(B_1, p_{B_1}, vl, hb) \sqcap \cdots \sqcap \mathsf{mgu}^\tau(B_n, p_{B_n}, vl, hb)$$
$$\mathsf{mgu}^\tau(t, u, [V_1, \ldots, V_n, c_1, \ldots, c_m], hb) =$$
$$\quad \ulcorner hb = hb(\mathsf{mgu}^\tau_{V_1}(t, u), \ldots, \mathsf{mgu}^\tau_{c_m}(t, u))[u] \urcorner$$

$$\mathsf{mgu}^\tau_w(f(t_1, \ldots, t_n), u) = \mathsf{mgu}^\tau_w(t_1, f^{-1}_{(1)}(u)) \sqcap \cdots \sqcap \mathsf{mgu}^\tau_w(t_n, f^{-1}_{(n)}(u))$$
$$\mathsf{mgu}^\tau_w(c, u) = \textbf{ if } w = c \textbf{ then } u \sqcap w \textbf{ else } \top$$
$$\mathsf{mgu}^\tau_w(V, u) = \textbf{ if } w = V \textbf{ then } u \textbf{ else } \top$$
$$\mathsf{prjvars}([V_1, \ldots, V_n, c_1, \ldots, c_n], hb) = \ulcorner V_1 = hb^{-1}_{(1)}(hb), \ldots, V_n = hb^{-1}_{(n)}(hb) \urcorner$$

where hb is a fresh variable for each clause and vl is a list of all variables and constants in the clause. The function mgu^τ generates a constraint that describes the set of substitutions from the right hand side. The function prjvars projects each variable and thus corresponds to the α function in the set-based interpretation.

Relation. $\mathscr{T}[\![P]\!] = \mathsf{M}_P[\![\mathsf{C}^\tau[\![P]\!]]\!]$

Improved constraints. Further precision can be gained by preserving some dependencies between "brother variables" in the clause head instead of projecting each variable separately.

$$\begin{array}{l}
\mathsf{C}^b[\![cl_1 \cdots cl_n]\!] = \mathsf{R}^b[\![cl_1]\!] \sqcup \cdots \sqcup \mathsf{R}^b[\![c_n]\!] \\
\mathsf{R}^b[\![H \text{ :- } B_1, \ldots, B_n]\!] = \\
\quad \mathsf{apply}(H, vl, hb) \sqcap \mathsf{mgu}^\tau(B_1, p_{B_1}, vl, hb) \sqcap \cdots \sqcap \mathsf{mgu}^\tau(B_n, p_{B_n}, vl, hb) \\
\mathsf{apply}(p(e_1, \ldots, e_n), vl, hb) = \textbf{let}\,(s_i, cs_i) = \mathsf{apply}'(e_i, vl, hb)\ \textbf{in} \\
\quad \ulcorner p = p(\mathsf{prj}(s_1, vl, hb), \ldots, \mathsf{prj}(s_n, vl, hb))[hb]\urcorner \sqcap cs_1 \sqcap \cdots \sqcap cs_n \\
\mathsf{apply}'(V, vl, hb) = (V, \ulcorner\urcorner) \qquad \text{where } \ulcorner\urcorner \text{ is the empty constraint system} \\
\mathsf{apply}'(f(e_1, \ldots, e_n), vl, hb) = \textbf{let}\,(s_i, cs_i) = \mathsf{apply}'(e_i, vl, hb)\ \textbf{in} \\
\quad (h, \ulcorner h = f(\mathsf{prj}(s_1, vl, hb), \ldots, \mathsf{prj}(s_n, vl, hb))[hb]\urcorner \sqcap cs_1 \sqcap \cdots \sqcap cs_n) \\
\mathsf{prj}(V_i, [V_1, \ldots, V_n, c_1, \ldots, c_m], hb) = hb^{-1}_{(i)}(hb) \\
\mathsf{prj}(h, [V_1, \ldots, V_n, c_1, \ldots, c_m], hb) = h
\end{array}$$

In the function apply' we generate a new variable h for each sub-expression in the head. The function mgu^τ was defined previously and it creates constraints that describe the set of substitutions. We then use the tuple constructor to apply the substitutions independently to the head. This is only possible if all arguments are variable. It can, however, be performed at all levels of the head and thus retain relations between arguments at the same level in the head.

Relation. $\mathsf{T}[\![P]\!] \subseteq \mathsf{M}_P[\![\mathsf{C}^b[\![P]\!]]\!] \subseteq \mathscr{T}[\![P]\!]$

8 Example

The following example shows a small program where the three set constraint interpretations generate different results. We show the set constraints and their solutions as found by the interpretation presented in sections 3 and 4.

```
p(f(a,b)).
p(f(b,a)).
p(f(c,c)).
q(f(X,Y)) :- p(f(X,Y))
r(V) :- q(f(V,V)).
s(W) :- p(f(W,W)).
```

Heintze and Jaffar style set constraints. To the left we show the constraint system generated from the program above using the C^h function. To the right we show the constraints in solved form.

$$\begin{array}{l}
V = \mathtt{f}^{-1}_{(1)}(\mathtt{q}^{-1}_{(1)}(q)) \cap \mathtt{f}^{-1}_{(2)}(\mathtt{q}^{-1}_{(1)}(q)) \\
W = \mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p)) \cap \mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)) \\
X = \mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p)) \\
Y = \mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)) \\
p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b})) \cup \mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a})) \cup \mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) \\
q = \mathtt{q}(\mathtt{f}(X,Y)) \\
r = \mathtt{r}(V) \\
s = \mathtt{s}(W)
\end{array}
\qquad
\begin{array}{l}
p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b})) \cup \\
\qquad \mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a})) \cup \mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) \\
q = \mathtt{q}(\mathtt{f}(V,V)) \\
r = \mathtt{r}(V) \\
s = \mathtt{s}(V) \\
V = \mathtt{a} \cup \mathtt{b} \cup \mathtt{c}
\end{array}$$

Tuple based set constraints. In the tuple based constraints we can describe the sets of substitutions but the values are projected out before being applied to the head. This means that precision is lost in q and r, but the description of s is correct since only one variable needs to be extracted.

$$
\begin{array}{ll}
V = \mathtt{rbody}^{-1}_{(1)}(rbody) & p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b}))\cup \\
X = \mathtt{qbody}^{-1}_{(1)}(qbody) & \quad \mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a}))\cup\mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) \\
Y = \mathtt{qbody}^{-1}_{(2)}(qbody) & q = \mathtt{q}(\mathtt{f}(V,V)) \\
p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b}))\cup\mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a}))\cup\mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) & r = \mathtt{r}(V) \\
q = \mathtt{q}(\mathtt{f}(X,Y)) & s = \mathtt{s}(\mathtt{c}) \\
qbody = \mathtt{qbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p)),\mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)))[p] & V = \mathtt{a}\cup\mathtt{b}\cup\mathtt{c} \\
r = \mathtt{r}(V) & \\
rbody = \mathtt{rbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{q}^{-1}_{(1)}(q))\cap\mathtt{f}^{-1}_{(2)}(\mathtt{q}^{-1}_{(1)}(q)))[q] & \\
s = \mathtt{s}(W) & \\
W = \mathtt{sbody}^{-1}_{(1)}(sbody) & \\
sbody = \mathtt{sbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p))\cap\mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)))[p] &
\end{array}
$$

Improved constraints. In the last example we can keep the relationship between the arguments to `f` in the `q` clause and consequently we obtain a precise description of all the variables.

$$
\begin{array}{ll}
h = \mathtt{f}(\mathtt{qbody}^{-1}_{(1)}(\mathtt{qbody}),\mathtt{qbody}^{-1}_{(2)}(\mathtt{qbody}))[\mathtt{qbody}] & p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b}))\cup \\
p = \mathtt{p}(\mathtt{f}(\mathtt{a},\mathtt{b}))\cup\mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a}))\cup\mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) & \quad \mathtt{p}(\mathtt{f}(\mathtt{b},\mathtt{a}))\cup\mathtt{p}(\mathtt{f}(\mathtt{c},\mathtt{c})) \\
q = \mathtt{q}(h) & q = \mathtt{q}(h) \\
qbody = \mathtt{qbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p)),\mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)))[p] & r = \mathtt{r}(\mathtt{c}) \\
r = \mathtt{r}(\mathtt{rbody}^{-1}_{(1)}(rbody))[rbody] & s = \mathtt{s}(\mathtt{c}) \\
rbody = \mathtt{rbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{q}^{-1}_{(1)}(q))\cap\mathtt{f}^{-1}_{(2)}(\mathtt{q}^{-1}_{(1)}(q)))[q] & h = \mathtt{f}(\mathtt{a},\mathtt{b})\cup \\
s = \mathtt{s}(\mathtt{sbody}^{-1}_{(1)}(sbody))[sbody] & \quad \mathtt{f}(\mathtt{b},\mathtt{a})\cup\mathtt{f}(\mathtt{c},\mathtt{c}) \\
sbody = \mathtt{sbody}(\mathtt{f}^{-1}_{(1)}(\mathtt{p}^{-1}_{(1)}(p))\cap\mathtt{f}^{-1}_{(2)}(\mathtt{p}^{-1}_{(1)}(p)))[p] &
\end{array}
$$

The new constraints will in this example give the same solution as T_P and in general the solution will be a subset of $\mathscr{T}_P$ and a superset of T_P.

9 Related Work and Discussion

Related Work. Automatic derivation of descriptions of sets of terms capturing the "shape" of computational entities such as data structures, computation trees and proof trees has been the subject of much research in static analysis. Following Reynolds' early work [27], further advances were made by Jones and Muchnick [21, 20]. The explicit study of set constraints was started by Heintze and Jaffar [17, 15]. This led to a series of publications exploring various classes of set constraints and their solution algorithms (e.g. [2, 1, 23, 5, 25, 9]).

In parallel with this stream of research, abstract interpretation over domains of tree descriptions were being explored, especially in static analysis of logic programs [19, 30, 14, 24, 4]. Cousot and Cousot [8] showed that the two approaches were related: the solution of a set of set constraints derived from a program could be seen as an abstract interpretation of the program over a domain of tree grammars.

Another approach was pursued by Frühwirth *et al.* [12]. In this work a logic program was first approximated by a program containing only unary predicates. This approach has some similarity to the

set constraint method; the derivation of the unary program followed by its normalisation corresponds to the extraction of set constraints followed by their solution. It was also noted in [12] that the computation of the model of the unary program model mirrored the construction of the abstract $\mathscr{T}_P$ abstraction of the original program, but the abstract domain in this case was an infinite height domain consisting of infinite sets of ground terms and so this interpretation was not particularly useful in practice. Gallagher and Puebla [13] constructed and implemented an abstract interpretation for logic programs over a domain of non-deterministic tree grammars, yielding a tree-grammar approximation of a program P equivalent to the $\mathscr{T}_P$ abstraction.

Tree grammar approximations of the set of reachable terms in rewrite systems has also been studied, dating back to the work of Jones [20]. Recent work in this area is [11, 3].

Discussion. The first aspect of this work concerns the extraction of more precise set constraints *directly* from programs. Previous presentations of set constraint analyses have shown existentially quantified set expressions having a similar purpose to our tuple constructor, and are used to express the $\mathscr{T}_P$-model of a logic program [17]. However, a clear syntactic characterisation of a class of solvable constraints comparable to our extended constraints, along with a precisely defined solution procedure, seems to be absent from previous work. The presentation in [17] uses existentially quantified set constraints to define an abstraction of a logic program. It is pointed out by Heintze [16], where a procedure handling quantified constraints is discussed, that unrestricted use of quantified expressions leads to undecidability.

The second aspect concerns different program abstractions and their relation to each other. Firstly, set constraints themselves are regarded as a semantic domain; this has not previously been done. A function S was constructed, assigning a tree grammar as the meaning of a set of (extended) set constraints. S is both a semantic function for set constraints and a practical solution procedure when executed. Tree grammars thus form another domain of description which is more abstract than set constraints, since many sets of set constraints can denote the same tree grammar. A program P in some source language (we used logic programs as a case study) can be given an abstract interpretation $\mathsf{C}[\![P]\!]$ as a set of set constraints. The function C can be chosen to give different approximations (such as τ, Y and Z for logic programs [18]). By composing the set constraint extraction function C with the S interpretation we obtain a tree grammar abstraction of P, namely $\mathsf{S}(\mathsf{C}[\![P]\!])$. Finally the term interpretation M returns an abstract meaning $\mathsf{M}(\mathsf{S}(\mathsf{C}[\![P]\!]))$ over the domain of sets of ground expressions, which is in turn a more abstract meaning than tree grammars.

Other extensions of set constraints. Constraints may be extended with quantified constraints [16], membership expressions [9] and generalized definite set constraints [29]. The central aspect of such extensions are expressions of the form $\{x \mid t_1 \in V_1 \wedge \cdots \wedge t_n \in V_n\}$ where t_i are terms built from constructors, the variable x and other unbound variables. Such expressions can be rewritten using the tuple constructor by constructing an expression that describe the set of environments for the unbound variables in the expression. From this set one can extract the variable x as the meaning of the whole expression. The example from section 4, however, cannot be expressed directly as membership expressions. In [16] there are extensions with atomic complement expressions and a disjointness expression and in [29] quantified expressions are extended with universally quantifiers. These extensions seems orthogonal to the extension discussed here.

Future Work. Our construction is step in developing a framework in which both set constraint solving and abstract interpretation over tree grammars can be explored. This in turn will allow connections to be

explored with a wider class of analyses. Firstly, tree grammar interpretations over infinite sets of states (with a widening operator) are inherently more precise than those over program-specific finite domains. The corresponding development in set constraints would seem to be the dynamic generation of set constraints during program execution or partial evaluation. Secondly, tree grammars (tree automata) can be enriched with constraints [6, 26] and this points to the possibility of more precise abstract domains. Again, a synthesis would allow this to be related to extension of set constraints with, say, arithmetic constraints. Finally, as pointed out by Cousot and Cousot, standard constructions from abstract interpretation such as reduced product can be exploited to combine tree grammar abstractions with other abstractions, such as term size, instantiation modes, term sharing and so on.

10 Conclusion

We introduced a tuple constructor set expression and showed that set constraints using this construct can be solved yielding tree grammars. Extended set constraints extracted in a single pass from a logic program are able to express the $\mathscr{T}_P$ abstraction, which previous set constraint analyses for logic programs were unable to achieve as far as we are aware. We also formulated the derivation and solution of set constraints in a form that emphasized that set constraints, tree grammars and sets of ground terms are closely related domains of abstraction.

References

[1] A. Aiken (1994): *Set Constraints: Results, Applications, and Future Directions.* In: *PPCP'94*, LNCS vol. 874, pp. 326–335.

[2] A. Aiken & E. L. Wimmers (1992): *Solving Systems of Set Constraints (Extended Abstract).* pp. 329–340.

[3] Y. Boichut et al. (2007): *Rewriting Approximations for Fast Prototyping of Static Analyzers.* In: *RTA 2007*, LNCS vol. 4533, pp. 48–62.

[4] F. Bueno, J. A. Navas & M. V. Hermenegildo (2010): *Towards Parameterized Regular Type Inference Using Set Constraints*. Technical Report, CoRR abs/1002.1836.

[5] W. Charatonik, A. Podelski & J. Talbot (2000): *Paths vs. Trees in Set-Based Program Analysis.* In: *POPL 2000*, pp. 330–337.

[6] H. Comon et al. (1999): *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata.

[7] P. Cousot & R. Cousot (1978): *Static determination of dynamic properties of recursive procedures*. In E. J. Neuhold, editor: *Formal Description of Programming Concepts*, North-Holland.

[8] P. Cousot & R. Cousot (1995): *Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation.* In: *FPCA'95*, pp. 170–181.

[9] P. Devienne, J. Talbot & S. Tison (1997): *Solving Classes of Set Constraints with Tree Automata.* In: *CP97*, LNCS vol. 1330, pp. 62–76.

[10] A. Dix (1988): *Finding Fixed Points in Non-Trivial Domains: Proofs of Pending Analysis and Related Algorithms*. Technical Report 107, Univ. of York.

[11] G. Feuillade, T. Genet & V. V. T. Tong (2004): *Reachability Analysis over Term Rewriting Systems. J. Autom. Reasoning* 33(3-4), pp. 341–383.

[12] T. Frühwirth et al. (1991): *Logic Programs as Types for Logic Programs*. In: *IEEE Symposium on Logic in Computer Science, Amsterdam.*

[13] J. P. Gallagher & G. Puebla (2002): *Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs.* In: *PADL 2002*, LNCS vol. 2257, pp. 243–261.

[14] J. P. Gallagher & D. A. de Waal (1994): *Fast and Precise Regular Approximations of Logic Programs.* In: *ICLP'94*, pp. 599–613.

[15] N. Heintze (1992): *Practical Aspects of Set Based Analysis.* In: *ICLP'92*, pp. 765–779.

[16] N. Heintze (1992): *Set Based Program Analysis*. Ph.D. Thesis, CMU.

[17] N. Heintze & J. Jaffar (1990): *A Finite Presentation Theorem for Approximating Logic Programs.* In: *POPL'90*, pp. 197–209.

[18] N. Heintze & J. Jaffar (1992): *Semantic Types for Logic Programs.* In: *Types in Logic Programming*, pp. 141–155.

[19] G. Janssen & M. Bruynooghe (1992): *Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. J. Log. Program.* 13(2& 3), pp. 205–258.

[20] N. D. Jones (1987): *Flow Analysis of Lazy Higher Order Functional Programs*. In S. Abramsky & C. Hankin, editors: *Abstract Interpretation of Declarative Languages*, Ellis-Horwood, pp. 103–122.

[21] N. D. Jones & S. S. Muchnick (1982): *A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures.* In: *POPL'82*, pp. 66–74.

[22] N. D. Jones & A. Mycroft (1986): *Data Flow Analysis of Applicative Programs Using Minimal Function Graphs.* In: *POPL'86*, pp. 296–306.

[23] D. Kozen (1998): *Set Constraints and Logic Programming. Inf. Comput.* 142(1), pp. 2–25.

[24] P. Mildner (1999): *Type Domains for Abstract Interpretation: A Critical Study*. Ph.D. thesis, Department of Computer Science, Uppsala University.

[25] A. Podelski, W. Charatonik & M. Müller (1999): *Set-Based Failure Analysis for Logic Programs and Concurrent Constraint Programs.* In: *ESOP'99*, LNCS vol. 1576, pp. 177–192.

[26] A. Reuß & H. Seidl (2010): *Bottom-Up Tree Automata with Term Constraints*. In Christian G. Fermüller & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, *Lecture Notes in Computer Science* 6397, Springer, pp. 581–593.

[27] J. C. Reynolds (1968): *Automatic computation of data set definitions.* In: *IFIP Congress (1)*, pp. 456–461.

[28] M. Rosendahl (2004): *Demand-Driven Higher-Order Fixpoint Iteration*. Technical Report, Univ. of Roskilde.

[29] J. Talbot, P. Devienne & S. Tison (2000): *Generalized Definite Set Constraints. Constraints* 5(1/2), pp. 161–202.

[30] P. Van Hentenryck, A. Cortesi & B. Le Charlier (1995): *Type Analysis of Prolog Using Type Graphs. J. Log. Program.* 22(3), pp. 179–209.