Approximating the success set of logic programs using constrained regular types

Lee Naish

Department of Computer Science and Software Engineering
University of Melbourne, Parkville 3010
Email: `lee@cs.mu.oz.au`

Abstract

A key component of many logic programming analysis tasks is defining a safe (superset) approximation to the success set of the program. Our primary application is a declarative mode system which can express polymorphic modes and linearity information. Others include analysis of type and groundness dependencies. Even analysis of procedural properties generally uses declarative information. For example, termination analyses use inter-argument relations, which can be seen as safe approximations to the success set. In this paper we show how the success set can be approximated by *constrained regular types*. This is a domain similar to regular types but can contain constraints to express relationships between multisets of subterms and between well-typedness of subterms. It allows very precise analysis. Even meta interpreters can be analysed with no loss of precision. We define constrained regular types and describe a system which checks if a constrained regular type is a superset of the success set.

Keywords: logic programming, program analysis, mode, type, constrained regular types, set constraint, multiset, meta interpreter, aliasing, linearity

1 Introduction

In a previous paper (Naish 1996) we presented a very high level view of modes, showing how mode information can be captured by a set of ground atoms and defined a notion of well-modedness. We also suggested an expressive mode declaration language for defining such sets. In this paper we present an extension of the underlying formalism, which we call *constrained regular types*. We view modes as (expressive) types; readers may replace "mode" by "type" throughout this paper if desired. One of the attractions of constrained regular types is they are expressive enough to enable analysis of a variety of meta interpreters with no loss of precision and minimal loss of efficiency. In common with most work on analysis of logic programs, our mode system can be seen as an approximation to the success set of the program, hence we believe our work may be of use for a range of program analysis tasks. In particular, our mode declarations can express information about groundness, aliasing and linearity (in the sense that some predicates do not duplicate or discard certain data structures),

In the next two sections we describe our mode system and illustrate the information it captures with several examples. We then discuss the definition and meaning of constrained regular types and mode checking in this framework. After discussing related and further work we conclude.

2 Declarative modes

Our previous work identified a notion of modes with a set of ground atoms M, which is a superset of the success set of a program, $SS(P)$. The restriction to ground atoms simplifies the domain yet is surprisingly expressive. This scheme can be seen as a higher level view of directional types (Boye & Maluszynski 1995) (Bronsard, Lakshman & Reddy 1992). Directional types assign each argument of each procedure a type and directionality (input or output). Type checking ensures that if the inputs of a procedure are well-typed then the outputs will be well-typed for each solution. This is equivalent to taking M to be the set of atoms such that if the inputs are well-typed then the outputs are well-typed and checking $M \supseteq SS(P)$. Consider the normal definition of `append`, where each argument type is *list*, in the "forward" mode, which we declare as follows.

```
:- type list ---> [] ;  any.list.
:- mode append(X=list, Y=list, Z=list):
                  X+Y -> Z.
```

The mode set M is the set of `append` atoms such that $(X \in \mathit{list} \wedge Y \in \mathit{list}) \rightarrow Z \in \mathit{list}$. The "backward" mode can be declared similarly, using `Z -> X+Y`. The combination of these two modes (ad hoc mode polymorphism) can be declared using the conjunction of the two previous constraints or `X+Y <-> Z` and the mode set is the *intersection* of the two modes above. A smaller set means we have more knowledge about the program. The directional types formalism does not allow us to express this more precise approximation to the success set. In order to build more flexible mode systems and better analysis systems in general, we need more precise approximations. It is therefore important to consider expressive languages for defining sets of atoms.

Our previous work (Naish 1996) concentrated on parametric polymorphism, based on polymorphic types such as $\mathit{list}(T)$. It informally introduced constrained regular trees; we now call them constrained regular types and this paper defines them more formally. They allow a non-traditional interpretation of an expression containing type variables. Given an expression such as $\mathit{list}(T1)$ and a list, $T1$ is associated with the set (or more generally multiset) of elements in the list. Mode declarations typically have several type variables and a collection of constraints over the associated (multi)sets of terms. For example, the "forward" polymorphic mode for append associates `T1`, `T2` and `T3` with the sets of elements in each list and states that the set of elements in the third list is a subset of the union of the sets of elements in the first two lists (for simplicity we ignore `->` constraints on the list skeletons):

```
:- type list(T) ---> [] ;  T.list(T).
:- mode append(list(T1),list(T2),list(T3)):
```

```
        T1+T2 => T3.
```

The mode set is the append atoms such that $e(X) \cup e(Y) \supseteq e(Z)$, where $e(L)$ is the set of elements in list L. This mode is polymorphic in the sense that for any type T, if the first two arguments are of type $list(T)$ then the last argument is inferred to be of this type. By separating the constraints involving type variables, stronger constraints can be added independently of the modes of the list skeletons. For example, equality of the three sets can be expressed by the constraint `T1+T2 <=> T3`. This expresses the fact that `append` can be used for "back communication": two lists of variables (or partially instantiated terms) can be appended and the result passed to another procedure which can later bind all the variables. Multiset equality can be expressed by the constraint `T1+T2 <==> T3`. This expresses the fact that no references to elements are created or destroyed by calling `append`, an important fact for analysis of garbage collection and structure reuse. For example, we can infer that if the first two arguments only contain uniquely referenced terms then the last argument will also have this properly — precisely the information needed for one "unique mode" of `append` in Mercury (Somogyi, Henderson & Conway 1995). The declaration can also be seen as a precise description of the aliasing induced by a call to `append`, a groundness dependency, a type dependency, et cetera.

3 Examples

The mode declaration for the following naive coding of quicksort allows us to show that the multiset of list elements is preserved. This is useful information for verification and optimisation. The precision of the mode information for `part` is important: although `A` is used as an input to `part`, we need to know that it is not copied to `L1` or `L2`. Similarly, we need the multiset equality constraint for `append`.

```
:- mode qsort(list(T1),list(T2)): T1<==>T2.
qsort([], []).
qsort(A.B, C) :-
    part(A, B, L1, L2), qsort(L1, S1),
    qsort(L2, S2), append(S1, A.S2, C).

:- mode part(T0,list(T1),list(T2),
                list(T3)):  T1 <==> T2+T3.
part(A, [], [], []).
part(A, B.C, B.D, E) :-
    A @>= B, part(A, C, D, E).
part(A, B.C, D, B.E) :-
    A @< B, part(A, C, D, E).
```

The following code does a breadth first tree traversal, collecting the elements in the tree to form a list or constructing a tree from the list elements. The traversal is done by `bfq` which uses a queue of trees, represented as a number (the length of the queue) and a difference list. A logic variable representing the tail of the queue, `QT`, is introduced in the second and third arguments of the top level call to `bfq`. It never becomes ground and results in cyclic dataflow and aliasing, making program analysis challenging.

The mode declaration for `bfq` is able to capture the precise relationship between the multisets of elements in the list and the (sub)trees within the front and back of the queue. This, plus the top level call to `bfq`, can be used to verify that `bf` preserves the (multi)set of elements. If we only had information about the *sets* of elements in `bfq` we could not verify this property of `bf`. Mode analysis derives a constraint of the form `At+QT <==> As+QT` and the single occurrence of `QT` can be dropped from both sides. This cannot be done with a set constraint. Similarly, groundness analysis of this program using positive boolean functions (Armstrong, Marriott, Schachte & Søndergaard 1998) (a domain closely related to set constraints) is unable to establish any relationship between `At` and `As` due to the repeated variable `QT`.

```
:- type tree(T) ---> nil ;
            t(tree(T), T, tree(T)).
:- type nat ---> zero ; s(nat).

:- mode bf(tree(At),list(As)): At <==> As.
bf(At, As) :- bfq(s(zero), At.QT, QT, As).

% as above using a queue of trees
% (Length, [Front|Rest], Rest)
:- mode bfq(nat, list(tree(QH)),
    list(tree(QT)), list(L)): L+QT<==>QH.
bfq(zero, Q, Q, []).
bfq(s(N), nil.QH, QT, As) :-
    bfq(N, QH, QT, As).
bfq(s(N), t(L,A,R).QH, L.R.QT, A.As) :-
    bfq(s(s(N)), QH, QT, As).
```

The following code, taken from (Boye & Maluszynski 1995), takes a tree of numbers and builds another tree of numbers of the same shape, where each element is the maximum element of the first tree. Due to cyclic dataflow (through the variable `Max`), checking that a tree of number is indeed created is a challenge. The dependencies are expressed elegantly with `->` constraints. They show the skeleton of the second tree is dependent only on the skeleton of the first tree, the elements are dependent only on the "global" maximum (argument two) and the maximum is dependent only on the elements of the first tree.

```
:- mode maxtree(T0=tree(X=num),
            T=tree(Max=num)) :
        (T0 -> T), (X -> Max).
maxtree(Tree, NewTree) :-
    maxtree1(Tree, Max, Max, NewTree).

:-mode maxtree1(T0=tree(X0=num),GMax=num,
            Max=num, T=tree(X=num)) :
    (T0 -> T), (X0 -> Max), (GMax -> X).
maxtree1(nil, _, 0, nil).
maxtree1(t(L, X, R), GMax, Max,
        t(NewL, GMax, NewR)) :-
    maxtree1(L, GMax, MaxL, NewL),
    maxtree1(R, GMax, MaxR, NewR),
    max3(X, MaxL, MaxR, Max).

:- mode max3(A=num,B=num,C=num,Max=num):
            A+B+C -> Max.
```

The following code is a simple interpreter. With appropriate declarations, which we give later, our system is able to analyse this code with no loss of precision — something which is well beyond the capabilities of typical program analysis frameworks. For example, in a call such as `solve(qsort(X,Y))` it can be determined that the multisets of elements in the two lists are the same.

```
solve(true).
solve((G1,G2)) :- solve(G1), solve(G2).
solve(A) :- cl(A, B), solve(B).

cl(append([], As, As), true).
cl(append(A.As, Bs, A.Cs),
            append(As, Bs, Cs)).
...
```

4 Constrained regular types

As well as giving a more formal basis for our previous work (Naish 1996), we present two important improvements: constraints can occur within the definition of types and `->` constraints are supported. We first discuss definitions of *simple* constrained regular types. This is essentially a reconstruction of conventional regular types which makes the association between type parameters and multisets of terms more explicit. We then consider more expressive constraints.

4.1 Simple constrained regular types

We first describe the syntax of simple constrained regular type definitions:

Definition A *simple constrained regular type* is defined as follows:

$$\texttt{:- type } t(P_1, P_2, \ldots, P_n) \texttt{ ---> } C_1; C_2; \ldots$$

t/n is the name of the type, the P_i are distinct parameters and the each C_i is of the form

$$f(T_1, T_2, \ldots, T_k) : P_1 = E_1, P_2 = E_2 \ldots, P_n = E_n$$

f is a functor of arity k, the functors in each C_i are distinct, each T_i is type expression, possibly containing variables V_i (distinct from the parameters) and each E_i is a (possibly empty) set of V_i's denoting the union of these multisets. Each V_i occurs in exactly twice, in an E_i and a T_i.

Examples (we use `empty` to denote the empty set and `+` to denote multiset union):

```
:- type list(Es) ---> [] : Es = empty ;
        E.list(Es1) : Es=E+Es1.
:- type pair(F,S) ---> F1-S1 : F=F1, S=S1.
```

There is a trivial mapping between these definitions and standard discriminated union type definitions. The only difference is that we rename occurrences of parameters on the right hand side of the definition, as specified by the equations. The result is that different multisets of terms are denoted by different variables. In the definition of type `list`, for example, `Es` denotes (the multiset of) all members of the list, while `Es1` denotes the members of the tail of the list and `E` denotes the head.

A type expression denotes a set of functions from terms to booleans and multisets. When an expression such as $list(T)$ is "matched" with a term, we obtain a boolean (true if the term is a list) and a multiset of subterms T (we give a more formal and expressive definition later). Where the type expression contains several type parameters, several multisets are returned. A nested type expression such as $list(pair(F, S))$ associates F (and S) with the multiset subterms "matched" with F (and S, respectively) in a list of pairs. For example, matching with the term `[a-c,b-c]` we would associate F with $\{\texttt{a}, \texttt{b}\}$ and S with $\{\texttt{c}, \texttt{c}\}$. Nested type expressions are not essential for an expressive type system but are very convenient for the programmer. They can be considered a shorthand a new implicitly defined type. For example, $list(pair(F, S))$ can be seen as equivalent to $list\_pair(F, S)$ defined below. Each equation for the single parameter of $list/1$ is duplicated for the two new parameters.

```
:- type list_pair(F,S) --->
     [] : F = empty, S = empty ;
     pair(F1,S1).list(pair(F2,S2)) :
         F=F1+F2, S=S1+S2.
```

Ideally, the expression `list(pair(F2,S2))` in this definition should then be replaced by `list_pair(F2, S2)`. In our algorithm we essentially generate cases of these new definitions dynamically. In Figure 1 we define *matching_case* which returns a new case of a type definition given a (possibly) nested type expression.

Example: Consider the type $list(pair(F, S))$ and the definition of $list(Es)$ in the recursive case, $E.list(Es1) : Es = E + Es1$. We replace Es by $pair(F, S)$. For the equation $pair(F, S) = E + Es1$, we replace E and $Es1$ by new variants of the left hand side to get $pair(F, S) = pair(F1, S1) + pair(F2, S2)$. We then replace the equation by two equations to obtain the result given above in the $list\_pair$ definition. Note the restrictions on variable occurrences in definitions are important for this operation.

4.2 Adding constraints (high level)

We describe the role of `=>`, `==>` and `->` at a high level, in the user-supported syntax. In our implementation this is translated into a lower level form, which simplifies the coding and allows a more detailed formal treatment. Adding extra constraints to constrained regular type definitions allow us to have much finer control over what terms are in the type (the boolean returned); the multisets returned are never affected. Constraints over sets and multisets allow us to exclude more terms from the type, returning false as the boolean in more cases. We allow cases in type definitions of the form:

$$f(T_1, \ldots, T_k) : P_1 = E_1, \ldots, P_n = E_n, S_1, S_2, \ldots$$

where each S_i is a (multi)set constraint over variables. Matching with such a case will only return true if each S_i is true, restricting the type.

For example, the following type defines triples of lists where the elements of last list are all elements of the other lists (similar to `append`):

```
:- type app1(P1,P2,P3) --->
       a(list(L1), list(L2), list(L3)) :
           P1=L1, P2=L2, P3=L3, L1+L2=>L3.
```

Matching the term `a([a],[b],[a])` with this type returns $\langle true, P1 = \{\texttt{a}\}, P2 = \{\texttt{b}\}, P3 = \{\texttt{a}\}\rangle$. Matching `a([a],[b],[c])` with it results in *false* being returned (the multisets are also returned, though they are generally not of interest if the boolean is false).

In simple constrained regular types all subterms must be well typed (an implicit conjunctive constraint). Explicit constraints such as `->` allow us to *include* more terms in the type, returning true as the boolean in more cases. In the following type definition, the well-typedness of the three arguments are associated with boolean variables, `B1`, `B2` and `B3` (the booleans returned from the recursive matching of these subterms). The constraint `B1+B2->B3` means `B1` $\wedge$ `B2` implies `B3`.

```
:- type app2 --->
      a(B1=list(L1),B2=list(L2),B3=list(L3)):
           L1+L2=>L3, (B1+B2->B3).
```

Matching the term `a([a],[b],b)` with this type returns $\langle false\rangle$. Matching `a([a],b,b)` with it returns $\langle true\rangle$, since $B2$ is false. To precisely approximate the success set of a program it is vital to have such a method for including "ill-typed" atoms. Note also that we have not included parameters in this type (their values would typically not be very meaningful due to the presence of non-lists). Variables do not have to occur in equations. This allows flexibility similar to *existential types*, which are supported

$matching\_case(T, f, k)$; T is a type expression, $t(A_1, \ldots, A_n)$, A_i containing parameters $P_{i,j}$, the definition of t/n contains case C, $f(T_1, \ldots, T_k) : Q_1 = E_1, \ldots, Q_n = E_n$ (with variables renamed so they are unique):
  replace each Q_i by A_i;
  for each equation $A_i = E_i$
    replace both occurrences of each variable in E_i by a new variant of A_i;
    let the parameters in A_i be (in order) $P_{i,1}, \ldots, P_{i,m}$;
    replace the equation by m equations of the form $P_{i,j} = F_{i,j}, 1 \leq j \leq m$,
    where $F_{i,j}$ is E_i with each type expression replaced by its j^{th} variable;
  return the modified version of C;

Figure 1: matching_case algorithm

in some functional programming languages and recent versions of Mercury (such variables correspond to existentially quantified type variables).

There is an important difference between `->` and (multi)set constraints. Both arguments to (multi)set (in)equalities must be collections of multiset variables. The arguments of `->` can be arbitrary constraints, not only variables, and `+` is just syntactic sugar for conjunction. As well as supporting primitive constraints (either boolean variables or (multi)set constraints), constraints are closed under conjunction, implication and disjunction. Thus each case in a type definition can be seen as having just one constraint, defining the boolean, plus the equations defining the multisets. Recursive matching gives values for all boolean and multiset variables, allowing the constraint to be evaluated to a single boolean, which is returned.

4.3 Adding constraints — the low level

We make several changes to our definitions in order to incorporate these more general constraints. These changes allow us to simplify the matching algorithm, treat the boolean and multisets returned from matching in a more uniform way, allow more flexibility in what multisets of terms can be returned and avoid some overloading of type parameters (so far we have used them to create nested type expressions and also return multisets). We support the same user-level syntax in our system, but convert type definitions into an internal form based on what follows.

Instead of associating booleans and multisets with only some type expressions (those preceded by $B =$ and type variables, respectively), we do so for all type expressions. Each type is given two additional special arguments (a superscript and subscript may be more intuitive but it complicates presentation). The first argument of each type is associated with the multiset of terms the type is matched with. The second argument is associated with a boolean. All other arguments are used as conventional type parameters, allowing nested type expressions. We also introduce the universal type *any*, which replaces some uses of type variables. We only associate a meaning with *closed* type expressions, in which the only variables occur as special arguments.

Definition $t(P_1, P_2, T_1, \ldots, T_n)$ is a *closed type expression* if P_1 and P_2 are parameters and $T_1, \ldots, T_n$ are closed type expressions. All parameters are distinct.

To convert a high level type expression such as *T=tree(list(M))* to an equivalent low level closed type expression two steps are required. The first is to add the special arguments to each non-variable expression. Boolean variables are reused where they exist; otherwise new variables are used, for example, $tree(X_1, T, list(X_2, B_2, M))$. This makes all the booleans and multisets associated with these expressions explicit. The second step replaces each type parameter P by $any(P, B_i)$, for example, $tree(X_1, T, list(X_2, B_2, any(M, B_3)))$. This makes the multisets associated with the type parameters explicit. The booleans introduced in this step will always be true; we keep them for consistency with other types.

Definition A *constrained regular type* is defined as follows:

$$\texttt{:- type } t(P_1, P_2, \ldots, P_n) \texttt{ --->} \; C_1; C_2; \ldots$$

t/n is the name of the type, $n \geq 2$, the P_i are distinct parameters and the each C_i is of the form

$$f(T_1, T_2, \ldots, T_k) : P_2 = E_2 \ldots, P_n = E_n$$

f is a functor of arity k, the functors in each C_i are distinct, each T_i is type expression, containing multiset variables MV_i in argument one of (sub)terms, boolean variables BV_i in argument two of (sub)terms and nested type expressions or type variables TV_i in other arguments. Each $E_i, i > 2$ is a (possibly empty) set of TV_i's, denoting a union of multisets and/or a conjunction of booleans (the empty set representing *true*). Each TV_i occurs in exactly twice, in a E_i and a T_i. E_2 is a constraint over the BV_i's and set and multiset constraints over the MV_i's. There is an implicit equation defining P_1 to be the multiset containing the single term which is matched.

Examples (we use underscore for "don't care" variables; note the use of *any* instead of type variables in *app*2):

```
:- type list(X, B, Es) --->
        [] : Es = empty, B=true ;
        E.list(_,B1,Es1): Es=E+Es1, B=B1.
:- type app2(X,B) --->
    a(list(_,B1,any(L1,_)),
              list(_,B2,any(L2,_)),
              list(_,B3,any(L3,_))) :
        B = (L1+L2=>L3, (B1, B2 -> B3)).
```

The meaning of type expressions is given by matching with ground terms. In Figure 2 we give an algorithm which matches a term with a type expression to produce boolean and multiset expressions. The matching algorithm is defined for arbitrary terms and symbolic expressions containing variables may be returned. For ground terms the expressions can be evaluated to booleans and multisets. Matching non-ground terms is discussed later.

Definition A closed type expression containing $2n$ parameters denotes $2n$ functions, $m_1, \ldots, m_n$ and $b_1, \ldots, b_n$. Functions m_i are identified with the parameters appearing as argument one and map ground terms to multisets of (sub)terms. Functions b_i are

$match(X, T)$; X is a term, T is a closed type expression containing parameters $P_1, \ldots, P_n$:
Case 1: T is $any(P_1, P_2)$: return $\langle P_1 = \{X\}, P_2 = true \rangle$;
Case 2: $T = t(P_1, P_2, A_1, \ldots, A_m)$, $X = f(X_1, \ldots X_k)$:
if the definition if t/m has a matching case where
$matching\_case(T, f, k) = f(T_1, \ldots, T_k) : P_2 = E_2, \ldots, P_n = E_n$
recursively match each X_i with T_i to get expressions for each variable;
replace each variable by its assigned expression;
return $\langle P_1 = \{X\}, P_2 = E_2, \ldots, P_n = E_n \rangle$;
otherwise
return $\langle P_1 = \{X\}, P_2 = false, \ldots P_{2i-1} = \phi, P_{2i} = true, \ldots \rangle$;
Case 3: X is a variable, $T \neq any(\_, \_)$:
return $\langle P_1 = \{X\}, \ldots P_i = m(X, T, i), \ldots \rangle$;

Figure 2: matching a type expression with a term

identified with the parameters appearing as argument two and map ground terms to booleans. Ground terms which b_1 (argument two of the top level type expression) maps to *true* are said to be members of the type. The functions are all defined by the function *match*, defined in Figure 2, which takes a type expression and a term and returns an assignment for each parameter.

Example: The following algorithmic trace illustrates matching of the term `a([a],[b],[b])` with *app*2. For brevity we skip returning assignments to underscore variables. Multiset and boolean expressions are left in symbolic form: they could be simplified/evaluated as part of the algorithm or as a separate pass to obtain the value *true*.

call $match(\texttt{a([a],[b],[b])}, app2(P_1, P_2))$
call $match(\texttt{[a]}, list(\_, B_1, any(L_1, \_)))$
call $match(\texttt{a}, any(E_1, B_{E1}))$
return $\langle E_1 = \{\texttt{a}\}, B_{E1} = true \rangle$
call $match(\texttt{[]}, list(\_, B_{L1}, any(E_2, B_{E2})))$
return $\langle B_{L1} = true, E_2 = \phi, B_{E2} = true \rangle$
return $\langle B_1 = true, L_1 = \{\texttt{a}\} \cup \phi \rangle$
call $match(\texttt{[b]}, list(\_, B_2, any(L_2, \_)))$
...
return $\langle B_2 = true, L_2 = \{\texttt{b}\} \cup \phi \rangle$
call $match(\texttt{[b]}, list(\_, B_3, any(L_3, \_)))$
...
return $\langle B_3 = true, L_3 = \{\texttt{b}\} \cup \phi \rangle$
return $\langle P_1 = \{\texttt{a([a],[b],[b])}\}$,
$P_2 = (\{\texttt{a}\} \cup \phi \cup \{\texttt{b}\} \cup \phi \supseteq \{\texttt{b}\} \cup \phi$
$\wedge\ (true \wedge true \rightarrow true)) \rangle$

4.4 Approximating the success set

The key to our declarative view of modes (and much analysis of logic programming) is a set of atoms which is a safe approximation to (superset of) the success set. In most approaches to program analysis there is a (relatively inexpressive) type domain which is used to describe terms but not formulas. The approximation to the success set is based on additional constraints over the types of arguments of atomic formulas, such as the non-parametric modes of `append` we discussed in Section 2. Constrained regular types are much more expressive and we use them to define an approximation to the success set directly. We call this type *success*. It is a monomorphic type but captures the polymorphism (and other information) of individual predicates using constraints within the different cases of the type. The set of mode declarations in a program, including builtins, can be seen as a distributed definition of *success* as follows:

```
:- type success --->
    true:                              true ;
    call(A=success):                      A ;
    (A=success,B=success):              A,B ;
    T1=T2 :                         T1<==>T2 ;
    qsort(list(T1),list(T2)):       T1<==>T2 ;
    append(A=list(T1), B=list(T2),
       C=list(T3)):    T1+T2=>T3, A+B<->C;
    ...
```

The first case states that the atom `true` is in *success*. The next states that `call(X)` is in *success* if (and only if) `X` is in *success*. This is more precise than most analysis systems, which must assume *any* atom of the form `call(X)` may succeed. The next case states that a conjunction of two atoms (or formulas) is in *success* if both are in *success*. The next case states that `X=Y` is in *success* if the multiset containing `X` equals the set containing `Y`. No precision is lost for any of these predicates. For predicates such as `qsort` we lose some precision, as expected. The constraints given for `append` are chosen for an example we give later, and could be strengthened.

Mode checking must verify that this implicitly declared type *success* is indeed a superset of the success set. Checking if a given set is a superset of $SS(P)$ is undecidable so no complete (and sound) algorithm is possible. Our earlier mode checking system (like many other analysis systems) was based on the sufficient condition that $M \supseteq SS(P)$ if $M \supseteq T_P(M)$, where T_P is the immediate consequence operator for the program (Lloyd 1984). That is, it checks *success* is a *model* of the program. Resorting to model checking/inference is the typical (partial) solution to the undecidability of program analysis. The `bf` program provides an example of incompleteness of model checking. If we replace the multiset constraints by weaker set constraints (or even weaker groundness dependencies) *success* becomes larger but is no longer a model so analysis fails.

Our current mode checker does not use T_P. We use a more direct approach which is possible due to the expressiveness of constrained regular types. Models are normally characterised in a declarative way: M is a model if for every ground clause instance `H:-B`, if B is true in M then H is true in M. If we think of M as a type (*success*) this condition can be seen as a type dependency between B and H and can be expressed by a constrained regular type. Type *success* is a model, and hence a superset of the success set, if every ground clause instance is in type *clause*, defined as follows:

```
:- type clause --->
    (H=success :- B=success): B->H.
```

This expressiveness is what allows us to analyse meta interpreters with no loss of precision. There is no special treatment of clauses in our current system, except for the definition of the type above. The same functionality (and more) can also be user-defined. For example, the mode declarations for the meta interpreter given earlier are as follows:

```
:- mode solve(A=success) : A.
:- mode cl(H=success,B=success): (B->H).
```

Analysing the meta program is (almost) identical to analysing the object program. If matching a clause H:-B with *clause* results in the expression $S_B \rightarrow S_H$ then matching cl(H,B):-true with *clause* results in $true \rightarrow (S_B \rightarrow S_H)$. Other representations of clauses can be dealt with in similar ways. For example, we could use a difference list:

```
:- mode cl2(H=success, list(B=success),
        list(C=success)) : (B -> H + C).
cl2(append(A.As, Bs, A.Cs),
    [append(As, Bs, Cs) | Cont], Cont).
...
```

Although the data flow/modes/instantiation states associated with predicates such as cl2/2 can be extremely complicated, we have a very simple yet precise abstraction. One enabling factor is the very high level declarative approach. Another is the complexity which is encapsulated by the type *success*. All flexible analysis frameworks allow the expression of dependencies, such as our -> constraints, groundness dependencies and other type dependencies (implicit in directional types). However, these dependencies are normally only expressible at the top level. Constrained regular types allow dependencies to be expressed *within* a (recursively defined) type such as *success*.

At this point we have an expressive language for defining sets of terms, a condition for determining if such a set is a superset of the success set and a matching algorithm, which can check if a ground clause instance satisfies this condition. However, a practical analysis system requires algorithms for checking if *all* ground instances of a clause satisfy the condition. This is what we discuss next.

4.5 Matching non-ground terms

Matching a non-ground clause with *success* returns an expression which can contain program variables. Instead of being able to simply check the expression evaluates to *true*, we need to check if it is true for all possible values of the variables. It is similar to checking a purely boolean expression is *valid* (and we reduce it to this problem for some classes of constraints).

When the matching algorithm encounters a variable (Case 3) it returns symbolic expressions of the form $m(X, T, i)$, where X the is the program variable, T is a type expression and i is a parameter number. The expression denotes the value of the i^{th} parameter which would have been returned for some instance of X. For example, if $T = list(P_1, P_2, any(P_3, P_4))$, $m(\texttt{As}, T, 2)$ is a boolean representing whether (an instance of) As is a list and $m(\texttt{As}, T, 3)$ denotes $e(\texttt{As})$ — the multiset of elements in (the instance of) As. Matching our previously defined version of *success* with append(As,Bs,Cs) returns a symbolic boolean expression B which denotes

$$(e(\texttt{As}) \cup e(\texttt{Bs}) \supseteq e(\texttt{Cs})) \wedge(\texttt{As} \in list \wedge \texttt{Bs} \in list \leftrightarrow \texttt{Cs} \in list)$$

and matching with append(A.As,Bs,A.Cs) returns an expression H denoting

$$(\{\texttt{A}\} \cup e(\texttt{As}) \cup e(\texttt{Bs}) \supseteq \{\texttt{A}\} \cup e(\texttt{Cs})) \wedge(\texttt{As} \in list \wedge \texttt{Bs} \in list \leftrightarrow \texttt{Cs} \in list)$$

Thus to check all ground instances of the recursive clause for append are in type *clause*, we need to check $B \rightarrow H$ is true for all instances of the program variables.

The algorithm we use sacrifices completeness at some points. This does not overly concern us, since the fact that we can only detect models of the program (rather than arbitrary supersets of the success set) leads to incompleteness anyway. We have concentrated on getting the system to support a wide class of typical programs. Space limitations prevent us from presenting the algorithm in full, but we describe how the example above is dealt with.

We first substitute variables for some parts of the expression to make the domain simpler. Each distinct basic type constraint, such as $\texttt{As} \in list$ (actually the equivalent $m(\ldots)$ expression returned from *match*) is replaced by a new boolean variable such as L_{As}. Basic multiset expressions, such as $e(\texttt{As})$ and $\{\texttt{A}\}$, are also replaced by variables. Set (but not multiset) constraints are reduced to boolean constraints: the empty set is replaced by *true*, $\cup$ by $\wedge$ and $\supseteq$ by $\rightarrow$ (this preserves validity). This gives the following purely boolean expression:

$$\begin{array}{ll} & (E_{As} \wedge E_{Bs} \rightarrow E_{Cs}) \\ & \quad \wedge(L_{As} \wedge L_{Bs} \leftrightarrow L_{Cs}) \\ \rightarrow & (S_A \wedge E_{As} \wedge E_{Bs} \rightarrow S_A \wedge E_{Cs}) \\ & \quad \wedge(L_{As} \wedge L_{Bs} \leftrightarrow L_{Cs}) \end{array}$$

We then check this formula is *valid*, which is a sufficient condition for all instances of the clause to be in type *clause*.

There are several properties of this example which result in simple and efficient analysis. First, we have avoided multiset constraints, which are significantly harder to deal with than set constraints. Second, append is well typed in the Hindly-Milner style, so each variable is matched with only one type. If a variable is matched with more than one type (such as *num* and *int*) then analysis of sub-types may be needed, and that can also be expensive. Third, there are no disjunctive constraints, which makes checking validity easier. For this (very useful) class of programs our procedure runs in time polynomial in the size of the clause. For other classes of programs our procedure can have exponential complexity and it may be that further tradeoff between completeness and complexity is desirable.

Our implementation has rudimentary support for subtypes. If program variable N is matched with types *num* and *int* a constraint denoting $\texttt{N} \in int \rightarrow \texttt{N} \in num$ is added. Multisets constraints cannot be reduced to booleans in the way we have done for set constraints. For example, for multiset inclusion $e(\texttt{As}) \not\supseteq e(\texttt{As}) \cup e(\texttt{As})$ but $E_{As} \rightarrow E_{As} \wedge E_{As}$. We use a specialised algorithm to check if a collection of multiset constraints is valid and have limited interaction between set and multiset constraints. We have shown that multiset mode checking is NP hard in general but we believe data structures which are *single threaded* should not cause exponential complexity. Devising improved algorithms which take advantage of this is an area for future work.

5 Related work

Layered modes (Etalle & Gabbrielli 1996) are another extension of directional types and are also less expressive than -> constraints. The mode system of Mercury has recently been made more flexible using constraints (Overton, Somogyi & Stuckey 2002). Similarly, the mode system of Hal (de la Banda, Stuckey, Harvey & Marriott 2000), which is implemented on top of Mercury. The mode constraints for Mercury are lower level than ours, for example, constraints that each part of each variable has only a single producer. While we hope to apply the ideas behind our mode system to languages such as Mercury, there is still a significant gap between our high level description of modes and the level of Mercury code generation.

The proposed program analysis domain based on ACI unification of polymorphic type expressions (Codish & Lagoon 1996) has some similarities with our set constraints. Two advantages of our approach are that types can have multiple parameters and our use of constraints can avoid (an exponential number of) intermediate results during analysis. Another domain based on ACI1-unification (Codish, Lagoon & Bueno 1997) captures linearity information, whether an element occurs just once or multiple times, thus being closer to our multiset constraints.

6 Further work

In a separate paper (submitted for publication) we discuss application to floundering analysis of coroutining programs. Perhaps surprisingly, purely declarative analysis (ignoring the computation rule) can obtain quite precise information. We have built into our mode checker the types *ground* and *nonvar*. Precise floundering analysis relies heavily on these types, analysis of subtypes and well-typedness constraints (`->`, conjunction and disjunction). Other applications of our work related to groundness analysis (particularly inference) may also be of interest. More general mode inference is also an obvious area to investigate. An initial approach would be to start with type information, either declared or inferred in some way. The types would not have to be a superset of the success set but constraints could be inferred (particularly `->` constraints) so the resulting constrained type has this property.

It is straightforward to generalise our framework to support additional domains. For each domain we need a unit value ($true$, ϕ) and an operator for combining values ($\wedge$, $\cup$). We have done some initial work on domains which measure term size or depth for termination analysis, using 1 and + or *maximum*. Rather than (multi)set inequalities we can use numeric inequalities to map the domain onto boolean constraints.

Our work may also be helpful in the design of languages, particularly strongly typed ones. Its gives a basis for more flexible type system, including subtypes, and mode systems which support parametric polymorphism and linearity/uniqueness. Finally, more work is needed on algorithms for checking the validity of constraints. The applications we have suggested above may benefit from specialised algorithms which take advantage of the particular domains.

7 Conclusion

We have introduced the constrained regular type formalism for defining sets of terms or atoms. Its great flexibility has been demonstrated by describing a mode system for logic programs. Parametric polymorphism using set and multiset constraints has been discussed in previous work, but here we have given a more formal treatment. We have also shown how to incorporate ad hoc polymorphism using boolean constraints and how constraints can be incorporated into type definitions. This allows the success set to be precisely approximated by a single constrained regular type, a more direct way of checking the approximation is a model and analysis of meta interpreters with no loss of precision. We are hopeful the formalism will be found useful in other applications.

References

Armstrong, T., Marriott, K., Schachte, P. & Søndergaard, H. (1998), 'Two classes of Boolean functions for dependency analysis', *Science of Computer Programming* **31**(1), 3–45.

Boye, J. & Maluszynski, J. (1995), Two aspects of directional types, *in* L. Sterling, ed., 'Proceedings of the Twelfth International Conference on Logic Programming', Kanagawa, Japan, pp. 747–761.

Bronsard, F., Lakshman, T. K. & Reddy, U. S. (1992), A framework of directionality for proving termination of logic programs, *in* 'Proceedings of the Ninth Joint International Conference and Symposium on Logic Programming', pp. 321–335.

Codish, M. & Lagoon, V. (1996), Type dependencies for logic programs using ACI-unification, *in* 'Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems', IEEE Press, pp. 136–145.

Codish, M., Lagoon, V. & Bueno, F. (1997), An algebraic approach to sharing analysis of logic programs, *in* 'Proceedings of the Fourth International Static Analysis Symposium (SAS'97)', Vol. 1302 of *LNCS*, Springer-Verlag, pp. 68–82.

de la Banda, M. G., Stuckey, P., Harvey, W. & Marriott, K. (2000), Mode checking in HAL, *in* J. Lloyd, ed., 'Proceedings of the First International Conference on Computational Logic', LNCS 1861, Springer-Verlag, pp. 1270–1284.

Etalle, S. & Gabbrielli, M. (1996), Layered modes, *in* F. de Boer & M. Gabbrielli, eds, 'Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs'. Tehcnical Report TR-96-31, Dipartimento di Informatica di Pisa.

Lloyd, J. W. (1984), *Foundations of logic programming*, Springer series in symbolic computation, Springer-Verlag, New York.

Naish, L. (1996), A declarative view of modes, *in* 'Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming', MIT Press, pp. 185–199.

Overton, D., Somogyi, Z. & Stuckey, P. (2002), Constraint-based mode analysis of Mercury, *in* C. Kirchner, ed., 'International Conference on Principles and Practice of Declarative Programming', ACM Press, p. (to appear).

Somogyi, Z., Henderson, F. J. & Conway, T. (1995), Mercury: an efficient purely declarative logic programming language, *in* 'Proceedings of the Australian Computer Science Conference', Glenelg, Australia, pp. 499–512.