Mode checking using constrained regular trees

Lee Naish

(lee@cs.mu.OZ.AU, http://www.cs.mu.oz.au/~lee)

Technical Report 98/3

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

In a previous paper we presented a high level polymorphic mode system for logic programs. In this paper we present an algorithm which checks if a program is well moded, given that is it well-typed in the sense of Mycroft and O'Keefe. A program is well-moded if the set of ground atoms defined by mode declarations is a superset of the success set. The novelty of the algorithm is the expressiveness of the mode declarations. *Constrained regular trees* are used to define sets of terms and atoms. These are based on polymorphic types but allow set and multiset constraints over type variables. The expressiveness of this domain makes it very promising for many program analysis applications. Complexity is also (exponentially) better than other proposed domains for certain analysis tasks.

Keywords: logic programming, modes, types, constrained regular trees, aliasing, linearity, set constraints, multisets

1 Introduction

In a previous paper [Nai96] we presented a very high level view of modes, showing how mode information can be captured by a set of ground atoms, suggesting an expressive mode declaration language for defining such sets and defining a notion of well-modedness. In this paper we present an algorithm for checking well-modedness for a class of well-typed programs. Because of the high level nature of the mode system and the information which can be captured by the mode declarations, 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, argument sizes and linearity (in the sense that some predicates do not duplicate or discard certain data structures),

In the next section we present our mode system and illustrate the information it captures with several examples. We then discuss *constrained regular trees* which are the basis of our mode declaration language. The mode checking algorithm is then presented, along with some discussion about complexity of mode checking when the modes express linearity information. 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 [BM95][BLR92]. 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 (that is, the well-typed atoms plus all atoms with ill-typed inputs) and checking $M \supseteq SS(P)$. Consider the normal definition of `append`, where each argument type is `list`, in the "forward" mode (input, input, output). In [Nai96] this is declared as follows and the corresponding mode set is given below:

```
:- type list ---> [] ; any.list.
:- mode append(list, list, ~list).
```

$$M_f = \{append(X, Y, Z) | (X \in list \wedge Y \in list) \Rightarrow Z \in list\}$$

One advantage of our declarative set-based approach can be seen when ad hoc polymorphism is considered. The "backward" mode, `append(~list, ~list, list)`, corresponds to the set

$$M_b = \{append(X, Y, Z) | (X \in list \wedge Y \in list) \Leftarrow Z \in list\}$$

The combination of these two modes can be expressed by the *intersection* of the two sets above:

$$M_m = M_f \cap M_b = \{append(X, Y, Z) | (X \in list \wedge Y \in list) \Leftrightarrow Z \in list\}$$

In general, adding more mode information corresponds to making the mode set smaller, obtaining a more precise approximation to the success set. In order to build more flexible

mode systems we need more precise mode information. It is therefore important to consider expressive languages for defining sets of atoms. The previous work concentrated on parametric polymorphism.

The (parametric) polymorphic modes of [Nai96] are based on polymorphic types such as `list(T)`, where `T` is the type of the elements of the list. A formalism called *constrained regular trees* (see below) allows a non-traditional interpretation of an expression containing type variables. Given an expression such as `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:

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

The mode set is as follows, where $e(L)$ is the set of elements in list L:

$$M_{fp} = \{append(X, Y, Z) | (X \in list \wedge Y \in list) \Rightarrow (Z \in list \wedge e(X) \cup e(Y) \supseteq e(Z))\},$$

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 [SHC95]. 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.

In our prototype mode checking system we allow set and multiset inclusion and equality over unions of type variables or the empty set. The `~` notation not supported; instead we use a more expressive notation which we discuss later (we may support the `~` notation as a shorthand in the future). We now illustrate our mode system with some additional examples.

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 verficiation 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 efficient (modulo determinism detection) coding of merge sort preserves the *set* of list elements but removes duplicates. If multiset equalities were used the program would not be correctly moded and our algorithm would identify the third clause of `merge` as the source of the error. Multiset equalities could be used if this clause for `merge` was deleted (`@<` should be changed to `@=<` also). The multiset inequalities we have included express inter-argument size (list length in this case) relationships which are important for termination analysis. We have used `[]` for the last argument of the top level call to `msn`. If this was changed to an underscore the program would still work correctly but our mode checking algorithm would fail (analysis of list lengths and arithmetic would be needed to verify the modes).

```
:- mode mergesort(list(T1), list(T2)): T1<=>T2, T1==>T2.
mergesort(Us, Ss) :- length(Us, N), msn(N, Us, Ss, []).

% Returns first N elements of Us sorted + rest of Us
:- mode msn(int, list(T1), list(T2), list(T3)): T1<=>T2+T3, T1==>T2+T3.
msn(0, Us, [], Us).
msn(1, U.RUs, [U], RUs).
msn(N, Us, Ss, RUs) :- N > 1, N1 is N // 2, N2 is N - N1,
    msn(N1, Us, Ss1, Us2), msn(N2, Us2, Ss2, RUs), merge(Ss1, Ss2, Ss).

% merges two sorted lists; removes duplicates
:- mode merge(list(T1), list(T2), list(T3)): T1+T2<=>T3, T1+T2==>T3.
merge([], Ss, Ss).
merge(S.Ss, [], S.Ss).
merge(A.As, A.Bs, A.Ss) :- merge(As, Bs, Ss).
merge(A.As, B.Bs, A.Ss) :- A @< B, merge(As, B.Bs, Ss).
merge(A.As, B.Bs, B.Ss) :- A @> B, merge(A.As, Bs, Ss).
```

The following code takes a tree of key—value pairs and swaps the value corresponding to a certain key with another value. The choice of swapping rather than replacing means that the predicate can be used in more modes, many of which preserve resource usage (for example, references to values). The mode declaration captures this information by precise constraints over multisets of keys and values (including the singelton multisets corresponding to the first three arguments). For example, given the key `K`, the new value `V` and the first tree containing `Ks1` and `Vs1`, the three constraints tell us that the predicate might fail (if `K` is not in the tree) the keys in the new tree are the same as the old tree and the values in

the new tree are the same as the old tree except a single occurrence of `V0` has been replaced by `V` (no keys or values are duplicated or discarded).

```
:- mode swap_val(K, V0, V, tree(pair(Ks1, Vs1)), tree(pair(Ks2, Vs2))) :
        Ks1 ==> K, Ks1 <==> Ks2, Vs1 + V <==> Vs2 + V0.
swap_val(K, V0, V, t(L, K-V0, R), t(L, K-V, R)).
swap_val(K, V0, V, t(L, KN-VN, R), t(L1, KN-VN, R)) :-
    K @< KN, swap_val(K, V0, V, L, L1).
swap_val(K, V0, V, t(L, KN-VN, R), t(L, KN-VN, R1)) :-
    K @> KN, swap_val(K, V0, V, R, R1).
```

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 (exactly) one 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 [MS93] (a domain closely related to set constraints) is unable to establish any relationship between `At` and `As` due to the repeated variable `QT`.

```
:- 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).
```

4 Constrained regular trees

Constrained regular trees are defined more formally in [Nai96]. Here we describe them by relating them to the more conventional regular trees which are typically used to define types. The traditional way of defining algebraic types can be reconstructed in the constrained regular tree framework to make the sets of terms associated with type parameters more explicit. We use the definition of the polymorphic type *list(T)* defined above as an example. The definition states that there are two constructors: `[]/0` and `'.'/2`, which has arguments of type T and *list(T)*.

In constrained regular trees we are interested in the (multi)sets of terms denoted by type parameters. In this case the three instances of T denote three different sets. The first instance in the type definition denotes the set of all elements of the list. The next

denotes the singleton set containing the head of the list and the last occurrence denotes the set of elements in the tail of the list. The sets can be made explict by renaming the type parameters in the constructor expression to `T1` and `T2` and adding the constraint `T <==> T1 + T2`. For the nil constructor we can add the constraint that `T` is the empty set since it does not occur within the constructor expression:

```
:- type list(T) --->
    ([]          : T <==> empty) ;
    (T1.list(T2) : T <==> T1 + T2).
```

In general, for each constructor of the type and each type parameter T, the instances of the parameter within the constructor expression are renamed $T_1, T_2, \ldots$ and T is constrained to be the (multiset) union of $T_1, T_2, \ldots$ (if there are no occurrences of T then this is empty). Note that the variables in each constraint all correspond to the same type variable in the algebraic type definition. More conventional type definitions can be reconstructed from this information by unifying all variables occurring in each constraint then discarding the constraints.

Type expressions where arguments of a polymorphic type are non-variables can be seen as a shorthand for a new type. The constraints in the original type definition are duplicated for each type variable in that argument of the expression. For example, `list(pair(K,V))` can be seen as a shorthand for the type `list_pair(K,V)` defined below. `K` (`V`) is the set of terms in the list in the first (second) part of a pair, respectively.

```
:- type list_pair(K,V) --->
    ([]          : K <==> empty, V <==> empty) ;
    (pair(K1,V1).list_pair(K2,V2)
                 : K <==> K1 + K2, V <==> V1 + V2).

:- type pair(K,V) ---> (K1-V1 : K <==> K1, V <==> V1).
```

Constrained regular trees generalise the construction above to allow more flexible use of constraints. For example, a type describing terms which are lists of permutations of the same list (a restriction of `list(list(T))`) can be defined as follows:

```
:- type list_of_perms(L) ---> % L is multiset of terms in each inner list
    [] ; (list(L1).list_of_perms(L2) : L <==> L1, L <==> L2).
    % [] ; list(L).list_of_perms(L). % more concise version
```

In our work on modes we have not performed a detailed investigation of the use of these more flexible types as arguments to predicates. We have concentrated on the use of constraints in mode declarations and the definition of the mode set M.

4.1 Defining the Mode set

Constrained regular trees also allow a flexibility similar to *existential types*, which are supported in some functional programming languages. Type variables (such as the T_i) can occur locally within constructor expressions. This is the key to defining the "type" *success* which is the superset of the success set we use to describe polymorphic modes. It is a monomorpic type but captures the polymorphism of individual predicates using local

type variables which are suitably constrained. The set of mode declarations in a program, plus some information about builtins, can be seen as a distributed definition of *success* as follows:

```
:- type success --->
    true ;
    plus(int, int, int) ;
    call(success) ;
    (success, success) ;
    (T1 = T2 : T1 <==> T2) ;
    (append(list(T1), list(T2), list(T3)): T1+T2 <==> T3) ;
    (member(T1, list(T2)): T2 => T1) ;
    (mergesort(list(T1), list(T2)): T1<=>T2, T1==>T2) ;
    ...
```

This definition makes no distinction between function symbols, predicate symbols and connectives: a pragmatic choice which simplifies type and mode checking. It does not cause semantic difficulties as the context of a symbol in a program can be used to determine its class. Similarly, predicate and function symbols with multiple arities cause no problem since context information (the number of arguments) can be used to disambiguate them.

5 Mode checking algorithm

We first describe our algorithm for checking modes with parametric polymorphism. We then discuss the less novel case of monomorphic (or ad hoc polymorphic) modes.

5.1 Parametric polymorphic modes

Determining if a set M is a superset of the success set is undecidable in general. Our polymorphic mode checking algorithm is 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 [Llo84]. That is, it checks M (the set of ground atoms defined by the mode declarations) is a *model* of the program. `Mergesort` with `[]` replaced with `_` and the breadth first traversal code with `<==>` replaced by `<=>` are well moded in the sense that $M \supseteq SS(P)$. However, the mode sets are not models: $M \not\supseteq T_P(M)$. This is why our mode checking algorithm fails. M contains atoms such as `msn(2, [a,b], [], [a,b])` and `bfq(s(zero), [nil,t(nil,a,nil)], [t(nil,a,nil)], [a])` which lead to violation of the constraints for `mergesort` and `bf`.

Our algorithm checks one clause at a time. It assumes the clause is well typed in the sense of Mycroft and O'Keefe, and we are given the type for the head of the clause and each body literal. We assume that the type of each literal is identical to the declared type, not just an instance. For example, if a particular call to `append` has arguments which are inferred to be lists of lists then a renamed version of `append`, with a specialised declaration, must be used. A pre-pass which checks types can also perform this specialisation quite simply; it may also be useful for optimisation.

The Mycroft and O'Keefe definition of well-typedness does not allow for the generality of constrained regular trees as argument types. Our algorithm specifically relies on the following properties. A type variable (parameter) is never assigned as the type of a nonvariable term. A nonvariable type is never assigned to a term whose principle functor is not

in that type. All occurrences of the same object variable are assigned the same type, up to renaming of type variables (recall that type variables are renamed if regular type definitions are converted to constrained regular trees).

The algorithm has three main stages: type matching, which collects constraints, constraint simplification and constraint checking. The complexity is dominated by the constraint checking. In the following, our remarks about complexity assume a fixed size for type definitions. In general, the algorithms may depend linearly on the number of parameters in a type definition, the number of variables in a constraint and the number of constraints in a single case within a type.

5.1.1 Type matching

Type matching takes a term X and a type T and returns a set of constraints over type expressions (including type variables) and object variables within the term, assuming Mycroft and O'Keefe type checking of the term with the type would succeed. The type expression may contain type parameters P_i and the term may contain object variables V_i.

Case 1: X is a variable V_i
Note: If T is a type parameter then X must be a variable, otherwise type checking would fail
Return the set containing the single constraint V_i <==> T.

Case 2: X is a term $f(X_1, ...X_n)$, T is a type expression $t(T_1, ...T_p)$ and the definition of type $t(P_1, ...P_p)$ contains the constructor expression $f(E_1, ...E_n)$ with constraints $C_1, ...C_k$
Note: Such a constructor must exist in the type, otherwise type checking would fail
Rename the type definition with type variables not appearing elsewhere;
Substitute $T_1, ...T_p$ for the type parameters in the type definition;
For each constraint over both a non-variable T_i and a type variable TV_j, construct a variant of T_i with new type variables and substitute it for every occurrence of TV_j;
Return the union of the resulting constraint set and the constraint sets returned by recursively matching each X_i with E_i, $i = 1..n$.

Remark: This algorithm is linear in the size of the input term (as is the size of the set of constraints output).

Example: $X = append(A.B, [], A.D)$ and $T = success$
At the top level we generate the constraint $T_1 + T_2$ <==> T_3 and recusively match $A.B$ with $list(T_1)$, [] with $list(T_2)$ and $A.D$ with $list(T_3)$. The middle argument results in the constraint T_2 <==> $empty$. The first argument results in the constraint T_1 <==> $T_4 + T_5$ and recursive matching of A with T_4 and B with $list(T_5)$. The third argument matching is similar. The final result is $\{T_1 + T_2$ <==> T_3, T_2 <==> $empty$, T_1 <==> $T_4 + T_5$, A <==> T_4, B <==> $list(T_5)$, T_3 <==> $T_6 + T_7$, A <==> T_6, D <==> $list(T_7)\}$

Example: $X = [A - B, B - A]$ and $T = list(pair(T_1, T_2))$
At the top level we initially generate the constraint $pair(T_1, T_2)$ <==> $T_3 + T_4$, which is converted to $pair(T_1, T_2)$ <==> $pair(T_5, T_6) + pair(T_7, T_8)$ and we recursively match $A - B$ with

$pair(T_5, T_6)$ and $[B - A]$ with $list(pair(T_7, T_8))$. Constraints $A <==> T_5$ and $B <==> T_6$ are generated from the first recursive match. The second results in $pair(T_7, T_8) <==> T_9 + T_{10}$, which is converted to $pair(T_7, T_8) <==> pair(T_{11}, T_{12}) + pair(T_{13}, T_{14})$ and we recursively match $B - A$ with $pair(T_{11}, T_{12})$ and [] with $list(pair(T_{13}, T_{14}))$. The final set of constraints is $\{pair(T_1, T_2) <==> pair(T_5, T_6) + pair(T_7, T_8),\ A <==> T_5,\ B <==> T_6,\ pair(T_7, T_8) <==> pair(T_{11},\ T_{12}) + pair(T_{13}, T_{14}),\ B <==> T_{11},\ A <==> T_{12},\ pair(T_{13}, T_{14}) <==> empty\}$.

5.1.2 Constraint simplification

The constraint simplification algorithm simplifies the set of constraints produced by type matching to remove constraints on object variables and non-variable type expressions.

> For each pair of constraints $V_i <==> T_j$ and $V_i <==> T_k$,
> add the constraint $T_j <==> T_k$;
> Delete all constraints on object variables;
> For each constraint over non-variable types which contain variables at subterm position p, generate a new constraint of the same form with each type expression replaced by its subterm at position p;
> *Note: The types in a constraint are all identical up to renaming of type variables. Constraints over non-variable types are introduced in Case 2 of type matching or from combining two constraints on the same object variable above.*
> Delete all constraints on non-variable types.

Example: $X = append(A.B, [], A.D)$ and $T = success$ (continued)
The two constraints on variable A lead to $T_4 <==> T_6$. The final result is $\{T_1 + T_2 <==> T_3, T_2 <==> empty, T_1 <==> T_4 + T_5, T_3 <==> T_6 + T_7, T_4 <==> T_6\}$.

Example: $X = [A - B, B - A]$ and $T = list(pair(T_1, T_2))$ (continued)
The two occurrences of A and B lead to $T_5 <==> T_{12}$ and $T_6 <==> T_{11}$. The constraint $pair(T_1, T_2) <==> pair(T_5, T_6) + pair(T_7, T_8)$ leads to two constraints: $T_1 <==> T_5 + T_7, T_2 <==> T_6 + T_8$. Similarly for the other two constraints on pairs. The final result is $\{T_1 <==> T_5 + T_7, T_2 <==> T_6 + T_8, T_5 <==> T_{12}, T_6 <==> T_{11}, T_7 <==> T_{11} + T_{13}, T_8 <==> T_{12} + T_{14}, T_{13} <==> empty, T_{14} <==> empty\}$.

5.1.3 Mode checking a clause

To check that a clause $p(...) :- B_1, B_2, ...$ satisfies the declared mode we apply the type matching algorithm to the conjunction $p(...), B_1, B_2, ...$ and the type *success*. The resulting set of constraints is separated into two sets: the constraints which originated from the top level matching of $p(...)$, the *head constraints*, (these are the constraints in the mode declaration for p) and the others, the *body constraints* (these are the constraints from the body of the clause plus matching within the arguments of the head). A clause is well moded if the body constraints entail the head constraints.

Example: $member(A, B.C) :- member(A, C)$
Matching $member(A, B.C), member(A, C)$ with *success* results in the constraints $\{T_2 => T_1, A <==> T_1, T_2 <==> T_3 + T_4, B <==> T_3, C <==> T_4\}$ for the first literal (the

first constraint is from the top level matching) and $\{T_6 => T_5, C <==> T_6, A <==> T_5\}$ for the second literal. Taking the union of these two sets, simplifying and separating out the head constraint we obtain $\{T_2 => T_1\}$ as the head constraints and $\{T_2 <==> T_3 + T_4, T_4 <==> T_6, T_6 => T_5, T_5 <==> T_1\}$ as the body constraints. Interpreted as set or multiset constraints, the head constraints are indeed entailed by the body constraints so the clause is considered well moded.

5.1.4 Checking constraint entailment

Algorithms for checking constraint entailment depend on the allowed constraints. We consider our main contribution to be showing how polymorphic mode checking can be reduced to constraint entailment in relatively simple domains. We have not yet studied or devised efficient entailment algorithms but have implemented (very) naive algorithms for our prototype. We have separate set and multiset entailment algorithms: to show a set constraint is entailed we treat multiset constraints as set constraints and to show a multiset constraint is entailed we only consider multiset constraints. Both these restrictions can lead to incompleteness of the entailment algorithm for the combined domain. We discuss this below. If only set constraints are used our prototype performs satisfactorily. For multiset constraints we have experienced efficiency and/or completeness problems even for some reasonably simple code (though all examples in this paper and its predecessor are handled).

Set constraint entailment is equivalent to entailment of (positive) boolean functions. The constraint $A + B => C + D$ can be interpreted as a boolean function where the variables are propositions, $+$ is conjunction and $=>$ is implication. It is likely that previously proposed algorithms for checking boolean functions will perform well for problems of this form. For example, the complexity of algorithms based on reduced ordered binary decision diagrams [Bry92] [Sch96] are likely to be almost linear in the size of clauses if the size of each declaration is considered constant, and quadratic otherwise. We have implemented a very naive algorithm which has cubic complexity for constant declaration size: to check that a head constraint $L => R$ is entailed we find everything that can be proved from L (using a fixedpoint calculation) and check this is a superset of R.

It is not surprising that multiset constraint entailment is significantly more difficult to implement: the number of non-equivalent expressions involving union over N variables is 2^N for sets but infinite for multisets, since $A + A \not\equiv A$. Multiset constraints are related to *linear* logic rather than classical logic. Even devising methods to prune the search space so it is finite (without compromising completeness) are non-trivial. From a constraint of the form $B ==> B + A$ we can derive $B ==> B + A + A$, $B ==> B + A + A + A$, et cetera. Constraints such as this generally only arise if programs contain cyclic modes. For example, the fact that `append(A.B,[],B)` has no (finite) solution can be concluded from the multiset mode constraint declared for `append`. Any clause containing such a call cannot contribute to the success set so we could conclude that the clause is well moded according to our definition. Cyclic modes are strongly linked to nontermination of programs [Nai93] and hence it would also be reasonable to give an error message if such constraints were encountered, or mode checking could simply fail. Our current algorithm simply concludes that (the type variable associated with) A is empty — a valid conclusion in the domain of (finite) multisets.

For our prototype we implemented an algorithm similar to a naive Prolog-style theorem prover with loop trapping which maintains a list of variables which are empty sets. The

algorithm has exponential complexity, which we believe is unavoidable (see later). To make this version practical we were forced to use an additional simplification phase to reduce the number of variables and constraints. The algorithm is also imcomplete. Even in subsequent versions we have implemented, which avoid much of the (huge) redundancy in the search space, we have had to compromise completeness in order to get acceptable performance for checking programs which are not well moded (multiset entailment fails but can take a very long time to do so). Our current version handles all examples in this paper in a fraction of a second and fails within a second or two for code of similar complexity which is not well moded.

There are two ways in which separating the set and multiset domains leads to incompleteness. The first is that treating cyclic multiset constraints as set constraints prevents us from concluding that some sets must be empty. The second is ignoring set constraints completely when dealing with multiset entailment. From a set constraint $A => B$, no multiset information can be concluded if A is not known to be empty. However, if A is known to be empty then we can conclude that B is also empty. Our multiset algorithm could be modified to use set constraints to infer that additional variables are empty sets and this seems desirable. It is unclear how much multiset information can be used for set constraint entailment without compromising efficiency.

5.2 Non-parametric modes

Checking of non-parametric modes, for example the input and output modes associated with the outer list skeletons in `append` can be done using the same framework of constraints. Rather than constraints only on parameters (leaf nodes) of the trees describing types, constraints can be on all nodes. We introduce a syntax to label nodes in a type expression, *Var* `=` *Expr*, and two new forms of constraint, *Vars* `->` *Vars* and *Vars* `<->` *Vars*. Collections of label variables in constraints are interpreted as conjunctions of booleans expressing dependencies between the well-typedness of different terms. Labels are implicitly inherited to outer expressions but not inner expressions. In the labelled expression `L=list(P=pair(K,V))`, `L` is true if the associated term is a list of anything and `P` is true if it is a list of pairs. The following declaration for `append` expresses the fact that the first two arguments are lists (of anything) if and only if the last argument is a list.

```
:- mode append(X=list(T1), Y=list(T2), Z=list(T3)): T1+T2<==>T3, X+Y <-> Z.
```

Our use of `~` in mode declarations previously can be considered a shorthand for a single `->` constraint involving all (non-variable) types of the predicate. Multiple mode declarations using `~` can be expressed using a single mode declaration with multiple `->` constraints. By using `->` constraints with only a subset of the labels we can gain even more expressive power. For example, in `append`, the fact that the first argument is always a list and the second argument is a list if and only if the third argument is a list can be expressed by the constraints `(empty -> X), Y <-> Z` (or more simply `X, Y <-> Z`). *Layered modes* have been proposed as a more flexible alternative to directional types [EG96]. Our approach is more flexible still, and similar to the *mode segments* proposed for Ptah [Som89].

Mode checking of `->` constraints is done in a very similar way to `=>` constraints. It is simplest to consider all non-variable nodes to be labelled. For the type matching algorithm, in Case 1 if the type T is labelled L an additional constraint is added to the set: V_i`<->`L. In Case 2, the renaming/substitution steps introduce new label variables as well as type

variables and an additional `<->` constraint is added to capture the fact that the term is well-typed if and only if all the arguments are well-typed (such constraints could also be added to all type definitions implicitly). In the constraint simplification algorithm, `<->` constraints over object variables and non-variable types are eliminated in the same way as <==> constraints. Boolean constraint entailment can use the same algorithm as set constraint entailment.

6 Complexity of multiset mode checking

The mode checking algorithms we have investigated have exponential complexity for multiset constraints. In this section we show that, unsurprisingly, alternative algorithms will still have exponential complexity[1]. This does not preclude the possibility algorithms which work adequately in practice.

We show how the NP-hard 3-satisfiability problem can be reduced to the multiset mode checking problem in polynomial time. Given a set of propositional variables $P_1, P_2, \ldots, P_k$, consider a boolean formula C of the form

$$D_1 \wedge D_2 \wedge \ldots \wedge D_N$$

where each D_i is a disjunction of three literals of the form P_i or $\neg P_i$. The problem is to find an assignment of *true* or $false$ to each P_i such that C is true (that is, each D_i is true). Our construction uses the following constraint to encode the condition that C is true:

$$P_1 + P_2 + \ldots + P_k \Longrightarrow D_1 + D_2 + \ldots + D_N$$

We construct a single clause procedure `c` such that mode checking of `c` has this constraint as the head constraint:

```
:- mode c(list(P1),list(P2),...,list(Pk),list(D1),list(D2),...,list(DN)) :
        P1+P2+...+Pk ==> D1+D2+...+DN.
c(P1,P2,...,Pk,D1,D2,...,DN) :- ...
```

The body of `c` is constructed so as to encode the choice of true or false for each P_i and the structure of C. For each positive (negative) occurrence of P_i in C we introduce a new variable $T_{i,j}$ ($F_{i,j}$). The choice of true or false for each P_i is encoded by two body constraints:

$$P_i \Longrightarrow T_{i,1} + T_{i,2} + \ldots, \qquad P_i \Longrightarrow F_{i,1} + F_{i,2} + \ldots$$

These are derived from two calls to procedure `ss`, which is assumed to be well moded:

```
    ss(Pi, [Ti_1,Ti_2,...]), ss(Pi, [Fi_1,Fi_2,...]), % body of c

:- mode ss(list(X), list(list(Y))) : X ==> Y. % mode declaration for ss
```

The structure of each D_i is encoded by three body constraints; the right hand side of each is D_i and the left hand sides are the T (F) variables corresponding to the positive (negative) occurrences of the P variables in D_i. For example, if $D_6 = P_1 \vee P_2 \vee \neg P_3$, where

[1] Our argument is somewhat informal; in particular, we don't prove that $P \neq NP$.

these are fourth occurrences of P_1 and P_2 and the fifth occurrence of $\neg P_3$, it would be encoded as follows:

$$T_{1,4} ==> D_6, \quad T_{2,4} ==> D_6, \quad F_{3,5} ==> D_6$$

These constraints are derived from three additional calls to `ss` in the body of procedure `c`. This completes the definition of `c`, which can clearly be constructed from C in polynomial time.

Due to the multiplicity of variables in the constraints in this construction, a proof procedure must choose between the constraints $P_i ==> T_{i,1} + T_{i,2} + \ldots$ and $P_i ==> F_{i,1} + F_{i,2} + \ldots$ for each P_i in order to find a proof. This is equivalent to choosing true or false for each P_i to show that C is satisfiable.

The construction gives some insight into a reason for the algorithmic complexity. Each P_i variable appears as an input to two calls and is a superset of the outputs of these calls (the $T_{i,j}$ and $F_{i,j}$, respectively). P_i is therefore a superset of the set union of the two but might not be a superset of the multiset union of the two. The proof procedure must choose between using either the $T_{i,j}$ or the $F_{i,j}$ for multisets whereas for sets the choice can be avoided by just using their union. Even if we do not have the full power of multisets but just have information about *linearity* (whether an element occurs just once or multiple times) the same exponential behaviour is exhibited.

However, the most important practical use for multiset constraints in modes is likely to be the analysis of *single threaded* data structures which are amenable to destructive update or more efficient garbage collection. Variables representing such data structures typically appear in the input of a call exactly once. The unique mode analysis in Mercury breaks down when there is more than one occurrence of an input variable which can be copied to an output. This conservatism avoids exponential behaviour. We are hopeful that using heuristics based on this class of programs will lead to practical algorithms for mode checking with multiset constraints.

7 Related work

In terms of the application area, mode systems for logic programming, the most closely related work is that on directional types referred to earlier. This work does not support polymorphism. Even the work on layered modes [EG96] is less flexible than our use of `->` constraints. Layered modes assign a directionality (input or output) and a *timing* to each argument. Output arguments depend on the input arguments with a smaller timing. The timings define a total order for the arguments. Using `->` defines a *partial* order, allowing us to specify depedencies more precisely (thus leading to more flexible modes). Using `<->` constraints allows ad hoc polymorphic, further increasing flexibility.

In terms of the abstract domain we use for analysis, the most closely related work is that of Codish et al. In [CL96] a domain based on associative commutative idempotent (ACI) unification of polymorphic type expressions is presented. Given definitions of types, terms are mapped to type expressions. For example, given polymorphic type definitions which associate `'.'/2` with $list/1$ and `[]` with nil, the arguments of a well-typed call to `append` may be mapped to $list(T_1) \oplus nil$ (representing a nil-terminated list whose elements are of type T_1), $list(T_2) \oplus nil$ and $list(T_1) \oplus list(T_2) \oplus nil$ (a nil-terminated list whose elements are of type T_1 or T_2). This captures essentially the same information as the set constraint we would use for `append`.

Due to idempotence ($X \oplus X = X$) no equivalent of multiset constraints can be expressed and types with more than one parameter are not supported so, for example, in code dealing with key—value pairs the keys and values cannot be distinguished. Another advantage of our domain is that the use of constraints can avoid exponential complexity for set-based analysis. Consider the following program:

```
:- mode p(list(A), list(B), list(C)): A + B <=> C.
p(A, B, [C_1,C_2,...,C_N]) :- append(A, B, [C_1,C_2,...,C_N]).
```

Analysis based on ACI-unification would unify $list(T_1) \oplus list(T_2) \oplus nil$ with the abstraction of the last argument of `append`: $list(C_1) \oplus list(C_2) \oplus \ldots \oplus list(C_N) \oplus nil$. This results in $3^N - 2$ unifiers (each C_i can be T_1, T_2 or $T_1 \oplus T_2$, as long as they are not all T_1 or all T_2). Our method would result in a single set of constraints, equivalent to $T_1 + T_2 <=> C_1 + C_2 + \ldots + C_N$, making (this part of) the analysis linear instead of exponential.

An advantage of the ACI-unification domain is that no restrictions are placed on programs. The type expressions for `append` given above are actually instances of more general expressions which can be inferred. These describe all successful instances of `append`, including those in which some arguments are non-lists. By restricting our attention to well-typed programs we have made it easier to support a very expressive type language while retaining relatively simple algorithms (for example, we avoid the need for complicated intersection or unification algorithms).

In [CLB97] another domain is proposed for sharing analysis which is isomorphic to the set sharing domain [JL92]. A domain based on sets of variables and ACI1-unification is presented then extended to incorporate linearity information (whether a set can contain multiple occurrences of variables). This goes some way towards the multiset information our constraints express: there is a distinction between one occurrence and (possibly) multiple occurrences. There may be some efficiency advantages in weakening our domain so less information about the multiplicity of variables is maintained. However, as we mentioned earlier, if the constraints can express linearity then constraint entailment will have exponential complexity.

8 Further work

There are many extensions to our work which may be worthwhile persuing. First, to make our prototype more useful, further investigation of set and multiset entailment algorithms needs to be done. Research may be needed to combine the set and multiset domains and (hopefully) devise methods which can avoid exponential complexity in typical cases.

Second, dropping the restriction to well-typed programs could be considered. Our mode checking algorithm could be combined with some form of type checking, preferably supporting subtypes (which are closely related to mode information [DH88] [RNP92]). Alternatively, arbitrary programs could be allowed. The matching phase of our algorithm would then become substantially more complex, requiring instersection of types in general. The work of Ueda [Ued95] may be useful here. This work infers modes in GHC, which is not strongly typed but has in-built constraint on modes due to guards. This mode inference algorithm is almost linear when variables occur at most twice (corresponding to data structures being single threaded). Supporting arbitrary constrained regular trees as argument types could be considered, further complicating these algorithms.

A deeper understanding of how well-typedness simplifies program analysis would also be useful — it may be that costly operations performed using other abstract domains can also be simplified or avoided for this class of programs. Historically, there have been two types camps in the logic programming community: "accept arbitrary programs and infer types" and "restrict programs and declare types". Program analysis has naturally been aligned more with the first camp but the second camp also provides fertile ground for analysis tasks. How these restrictions on programs affect analysis is poorly understood.

Third, inference rather than checking is an obvious topic. There are several levels of information which could be inferred. The simplest is to assume that types have been defined and declared for all argument positions. The dependencies or constraints over the types for predicates could then be inferred. Given type definitions, the argument types of predicates could also be inferred or everything, including the type definitions, could be inferred. Last, our work could be adapted for more concrete applications, either language design (for example, supporting polymorphic modes in Mercury or using mode declarations for coroutining) or analysis (for example, groundness and sharing).

9 Conclusion

We have presented a very high level mode system which can be given a purely declarative reading. It is very flexible, supporting both ad hoc and parametric polymorphism, and can express linearity information. We have emphasised the variety of information which can be expressed by the mode declarations rather than just concentrating on describing the data flow within programs. The language for expressing the information is based on constraints or dependencies between sets and multisets of terms. In this paper we have improved the treatment of non-parametric modes, showing how they can be supported in the same framework of constraints and thus allowing greater flexibility than other proposals.

Our main contribution here is a mode checking algorithm. It reduces the mode checking problem to the problem of constraint entailment over boolean, set and multiset inequalities. In the absence of linearity information the overall complexity is polynomial and almost linear algorithms should be possible in practice. The use of constraints in the analysis is important for achieving this efficiency: similar analysis which does not use constraints takes exponential time. When linearity information is present we have argued that the problem is NP-complete in general but may not be when linearity information concerns only single-threaded data structures, which is likely to be the most important application for this information.

For pragmatic reasons we restricted attention to well-typed programs. This significantly simplifies the algorithms needed. Specifically, it avoids the need for intersection or unification of the sets and multisets of terms described by our domain. Understanding how this restriction may affect other program analysis may be an interesting area for further research. Similarly, there are many opportunities for making our prototype implementation more practical, further improving its flexibility, investigating inference rather than checking and application of this work to specific problems.

References

[BLR92] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In *Proceedings of the Ninth Joint*

International Conference and Symposium on Logic Programming, pages 321–335, 1992.

[BM95] J. Boye and J. Maluszynski. Two aspects of directional types. In Leon Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 747–761, Kanagawa, Japan, June 1995.

[Bry92] Randal Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[CL96] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996.

[CLB97] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In *Proceedings of the Fourth International Static Analysis Symposium (SAS'97)*, volume 1302 of *LNCS*, pages 68–82. Springer-Verlag, Sept 1997.

[DH88] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for prolog. In H. Ganzinger, editor, *Proceedings of the Second European Symposium on Programming*, pages 79–93, Nancy, France, March 1988. published as Lecture Notes in Computer Science 300 by Springer-Verlag.

[EG96] S. Etalle and M. Gabbrielli. Layered modes. In F. de Boer and M. Gabbrielli, editors, *Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs*, 1996. Tehcnical Report TR-96-31, Dipartimento di Informatica di Pisa.

[JL92] Dean Jacobs and Anno Langen. Static analysis of logic programs for independent and-parallelism. *Journal of Logic Programming*, 13(2,3):291–314, 1992.

[Llo84] John W. Lloyd. *Foundations of logic programming.* Springer series in symbolic computation. Springer-Verlag, New York, 1984.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[Nai93] Lee Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.

[Nai96] Lee Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, September 1996.

[RNP92] Yann Rouzaud and Lan Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In *Proceedings of the Ninth Joint International Conference and Symposium on Logic Programming*, pages 85–97, 1992.

[Sch96] Peter Schachte. Efficient ROBDD operations for program analysis. In Kotagiri Ramamohanarao, editor, *ACSC'96: Proceedings of the 19th Australasian Computer Science Conference*, pages 347–356. Australian Computer Science Communications, 1996.

[SHC95] Zoltan Somogyi, Fergus J. Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.

[Som89] Z. Somogyi. A parallel logic programming system based on strong and precise modes. Technical Report 89/4, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1989. Ph.D. thesis.

[Ued95] K. Ueda. I/O mode analysis in concurrent logic programming. In T. Ito and Yonezawa, editors, *Theory and practice of parallel programming*, number 907 in Lecture Notes in Computer Science. Springer-Verlag, 1995.