Constraint-Based Mode Analysis of Mercury

David Overton[*] [†]
dmo@cs.mu.oz.au

Zoltan Somogyi[*]
zs@cs.mu.oz.au

Peter J. Stuckey[*]
pjs@cs.mu.oz.au

[*] Department of Computer Science and Software Engineering
The University of Melbourne, Victoria, 3010, Australia
[†] School of Computer Science and Software Engineering
Monash University, Victoria, 3800, Australia

ABSTRACT

Recent logic programming languages, such as Mercury and HAL, require type, mode and determinism declarations for predicates. This information allows the generation of efficient target code and the detection of many errors at compile-time. Unfortunately, mode checking in such languages is difficult. One of the main reasons is that, for each predicate mode declaration, the compiler is required to decide which parts of the procedure bind which variables, and how conjuncts in the predicate definition should be re-ordered to enforce this behaviour. Current mode checking systems limit the possible modes that may be used because they do not keep track of aliasing information, and have only a limited ability to infer modes, since inference does not perform re-ordering. In this paper we develop a mode inference system for Mercury based on mapping each predicate to a system of Boolean constraints that describe where its variables can be produced. This allows us handle programs that are not supported by the existing system.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*

General Terms

Algorithms, Languages

Keywords

modes, mode analysis, Boolean constraints

1. INTRODUCTION

Logic programming languages have traditionally been untyped and unmoded. In recent years, languages such as Mercury [19] have shown that strong type and mode systems can provide many advantages. The type, mode and determinism declarations of a Mercury program not only provide useful documentation for developers, they also enable compilers to generate very efficient code, improve program robustness, and facilitate integration with foreign languages. Information gained from these declarations is also very useful in many program analyses and optimisations.

The Mercury mode system, as described in the language reference manual, is very expressive, allowing the programmer to describe very complex patterns of dataflow. However, the implementation of the mode analysis system in the current release of the Melbourne Mercury compiler has some limitations which remove much of this expressiveness (see [8]). In particular, while the current mode analyser allows the construction of partially instantiated data structures, in most cases it does not allow them to be filled in. Another limitation is that mode inference prevents reordering in order to limit the number of possibilities examined.

The effect of these limitations is that it is hard to write Mercury programs that use anything other than the most basic modes and that it is just not possible to write programs with certain types of data flow.

As well as being limited in the ways described above, the current mode analysis algorithm is also very complicated. It combines several conceptually distinct stages of mode analysis into a single pass. This makes modifications to the algorithm (e.g. to include the missing functionality) quite difficult. The algorithm is also quite inefficient when analysing code involving complex modes.

In this paper, we present a new approach to mode analysis of Mercury programs which attempts to solve some of these problems of the current system. We separate mode checking into several distinct stages and use a constraint based approach to naturally express the constraints that arise during mode analysis. We believe that this approach makes it easier to implement an extensible mode analysis system for Mercury that can overcome the limitations of the current system.

We associate with each program variable a set of "positions", which correspond to nodes in its type graph. The key idea to the new mode system is to identify, for each *position* in the type of a variable, where that position is produced, i.e. which goal binds that part of the variable to a function symbol. We associate to each node in the type graph, and

PPDP'02, October 6–8, 2002, Pittsburgh, Pennsylvania, USA.

each goal in which the program variable occurs, a Boolean variable that indicates whether the node in the graph of that program variable is bound within the goal. Given these Boolean variables we can describe the constraints that arise from correct moding in terms of Boolean constraints. We can then represent and manipulate these descriptions using standard data structures such as reduced ordered binary decisions diagrams (ROBDDs). By allowing constraints between individual positions in different data structures, we obtain a much more precise analysis than the current Mercury mode system.

Starting with [11] there has been considerable research into mode inference and checking. However, most of this work is based on assumptions that differ from ours in (at least) one of two significant respects: types and reordering.

Almost all work on mode analysis in logic programming has focused on untyped languages, mainly Prolog. As a consequence, most papers use very simple analysis domains, such as {ground, free, unknown}. One can use patterns from the code to derive more detailed program-specific domains, as in e.g. [3, 10, 12], but such analyses must sacrifice too much precision to achieve acceptable analysis times. In [18], we proposed fixing this problem by requiring type information and using the types of variables as the domains of mode analysis. Several papers since then (e.g. [14, 17]) have been based on similar ideas. Like other papers on mode inference, these also assume that the program is to be analyzed as is, without reordering. They therefore use modes to *describe* program executions, whereas we are interested in using modes to *prescribe* program execution order, and insist that the compiler must have exact information about instantiation states. The only other mode analysis systems that do likewise work with much simpler domains (for example, Ground Prolog [9] recognizes only two instantiation states, free and ground).

Other related work has been on mode checking for concurrent logic programming languages and for logic programming languages with coroutining [1, 5, 7]: there the emphasis has been on detecting communication patterns and possible deadlocks. The only other constraint-based mode analysis we are aware of is that of Moded Flat GHC [4]. Moded Flat GHC relies on position in the clause (in the head or guard versus in the body) to determine if a unification is allowed to bind any variables, which significantly simplifies the problem. The constraints generated are equational, and rely on delaying the complex cases where there are three or more occurrences of a variable in a goal.

There has been other work on constraint-based analysis of Mercury. In particular, the work of [20] on a constraint-based binding-time analysis is notable. It has a similar basic approach to ours of using constraints between the "positions" in the type trees of variables to express data flow dependencies. However, binding-time analysis has different objectives to mode analysis and, in fact, their analysis requires the results of mode analysis to be available.

In Section 2 we give the background information the rest of the paper depends on. In Section 3 we briefly outline the current approach and some of its weaknesses. In Section 4 we give a simplified example of our constraint-based system before presenting the full system in Section 5. In Section 6 we show how the results of the analysis are used to select an execution order for goals in a conjunction. In Section 7 we give some preliminary experimental results.

2. BACKGROUND

Mercury is a purely declarative logic programming language designed for the construction of large, reliable and efficient software systems by teams of programmers [19]. Mercury's syntax is similar to the syntax of Prolog, but Mercury also has strong module, type, mode and determinism systems, which catch a large fraction of programmer errors and enable the compiler to generate fast code. The rest of this section explains the main features of Mercury that are relevant for this paper.

2.1 Programs

The definition of a predicate in Mercury is a goal containing atoms, conjunctions, disjunctions, negations and if-then-elses. To simplify its algorithms, the compiler converts the definition of each predicate into what we call superhomogeneous form [19]. In this form, each predicate is defined by one clause, all variables appearing in a given atom (including the clause head) are distinct, and all atoms are one of the following three forms:

$\mathtt{p(X_1,\ \dots,\ X_n)}$ $\qquad\qquad$ $\mathtt{Y\ =\ X}$ $\qquad\qquad$ $\mathtt{Y\ =\ f(X_1,\ \dots,\ X_n)}$

In this paper, we further assume that in all unifications, neither side is a variable that does not appear outside the unification itself. (Unifications that do not meet this requirement cannot influence the execution of the program and can thus be deleted.) For simplicity, we also assume that all negations have been replaced by if-then-elses (one can replace $\neg G$ with $(G \rightarrow \text{fail} \ ; \text{true})$). The abstract syntax of the language we deal with can therefore be written as

$$
\begin{array}{lll}
\text{atom} & A = & x = y | x = f(y_1, \dots, y_n) | p(x_1, \dots, x_n) \\
\text{goal} & G = & A | (G_1, \cdots, G_n) | (G_1; \dots; G_n) | (G_c \rightarrow G_t; G_e) \\
\text{rule} & R = & p(x_1, \dots, x_n) \leftarrow G
\end{array}
$$

In order to describe where a variable becomes bound, our algorithms need to be able to uniquely identify each subgoal of a predicate body. The code of a subgoal itself cannot serve as its identifier, since a piece of code may appear more than once in a predicate definition. We therefore use *goal paths* for this purpose. A goal path consists of a sequence of *path components*. We use λ to represent the path with zero components, which denotes the entire procedure body.

- If the goal at path p is a conjunction, then the goal path $p.\mathsf{c}_n$ denotes its nth conjunct.
- If the goal at path p is a disjunction, then the goal path $p.\mathsf{d}_n$ denotes its nth disjunct.
- If the goal at path p is an if-then-else, then the goal path $p.\mathsf{c}$ denotes its condition, $p.\mathsf{t}$ denotes its then-part, and $p.\mathsf{e}$ denotes its else-part.

Definition 1. The *parent* of goal path $p.\mathsf{c}_n$, $p.\mathsf{d}_n$, $p.\mathsf{c}$, $p.\mathsf{t}$, or $p.\mathsf{e}$, is goal path p. The function $\mathsf{parent}(p)$ maps a goal path to its parent.

Definition 2. Let $\nu(G)$ be the set of variables that occur within a goal, G. Let $\eta(G) \subseteq \nu(G)$ be the set of variables that are *nonlocal* to goal G, i.e. occur both inside and outside G. For convenience, we also define ν and η for a set of goals, S: $\nu(S) = \bigcup\{\nu(G) \mid G \in S\}$, and $\eta(S) = \bigcup\{\eta(G) \mid G \in S\}$.

Since each goal path uniquely identifies a goal, we sometimes apply operations on goals to goal paths.

2.2 Deterministic Regular Tree Grammars

In order to be able to express all the different useful modes on a program variable, we must be able to talk about each of the individual parts of the terms which that program variable will be able to take as values. To do so in a finite manner, we use regular trees, expressed as tree grammars.

A *signature* Σ is a set of pairs f/n where f is a *function symbol* and $n \geq 0$ is the integer *arity* of f. A function symbol with 0 arity is called a *constant*. Given a signature Σ, the set of all *trees* (the Herbrand Universe), denoted $\tau(\Sigma)$, is defined as the least set satisfying:

$$\tau(\Sigma) = \bigcup_{f/n \in \Sigma} \{f(t_1, \ldots, t_n) \mid \{t_1, \ldots, t_n\} \subseteq \tau(\Sigma)\}.$$

For simplicity, we assume that Σ contains at least one constant.

Let V be a set of symbols called *variables*. The set of all *terms* over Σ and V, denoted $\tau(\Sigma, V)$, is similarly defined as the least set satisfying:

$$\tau(\Sigma, V) = V \cup \bigcup_{f/n \in \Sigma} \{f(t_1, \ldots, t_n) \mid \{t_1, \ldots, t_n\} \subseteq \tau(\Sigma, V)\}$$

A *tree grammar* r over signature Σ and *non-terminal set* NT is a finite sequence of *production rules* of the form $x \to t$ where $x \in NT$ and t is of the form $f(x_1, \ldots, x_n)$ where $f/n \in \Sigma$ and $\{x_1, \ldots, x_n\} \subseteq NT$. A tree grammar is *deterministic regular* if for each $x \in NT$ and $f/n \in \Sigma$, there can be at most one rule of the form $x \to f(x_1, \ldots, x_n)$.

For brevity we shall often write tree grammars in a more compressed form. We use $x \to t_1; t_2; \cdots; t_n$ as shorthand for the sequence of production rules: $x \to t_1$, $x \to t_2$, ..., $x \to t_n$.

2.3 Types

Types in Mercury are polymorphic Hindley-Milner types. *Type expressions* (or *types*) are terms in the language $\tau(\Sigma_{type}, V_{type})$ where Σ_{type} are *type constructors* and variables V_{type} are *type parameters*. Each type constructor $f/n \in \Sigma_{type}$ must have a definition.

Definition 3. A *type definition* for f/n is of the form

`:- type` $f(v_1, \ldots, v_n)$ `--->` $f_1(t_1^1, ..., t_{m_1}^1); \cdots; f_k(t_1^k, ..., t_{m_k}^k)$.

where $v_1, \ldots, v_n$ are distinct type parameters, $\{f_1/m_1, \ldots, f_k/m_k\} \subseteq \Sigma_{tree}$ are distinct tree constructor/arity pairs, and $t_1^1, \ldots, t_{m_k}^k$ are type expressions involving at most parameters $v_1, \ldots, v_n$.

Clearly, we can view the type definition for f as simply a sequence of production rules over signature Σ_{tree} and non-terminal set $\tau(\Sigma_{type}, V_{type})$.

In order to avoid type expressions that depend on an infinite number of types, we restrict the type definitions to be *regular* [13]. (Essentially regularity ensures that for any type t, *grammar*(t), defined below, is finite.)

Example 1. Type definitions for lists, and a simple type `abc` which includes constants `a`, `b` and `c` are

```
:- type list(T) ---> [] ; [T|list(T)].
:- type abc ---> a ; b ; c.
```

We can associate with each (non-parameter) type expression the production rules that define the topmost symbol of the type. Let t be a type expression of the form $f(t_1, \ldots, t_n)$ and let f/n have type definition of the form in Definition 3. We define *rules*(t) to be the production rules:

$$\begin{array}{rcl} \theta(f(v_1, \ldots, v_n)) & \to & f_1(\theta(t_1^1), \ldots, \theta(t_{m_1}^1)) \\ & \vdots & \\ \theta(f(v_1, \ldots, v_n)) & \to & f_k(\theta(t_1^k), \ldots, \theta(t_{m_k}^k))) \end{array}$$

where $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ substitutes t_i for v_i. If $t \in V_{type}$ we define *rules*(t) to be the empty sequence.

We can extend this notation to associate a tree grammar with a type expression. Let *grammar*(t) be the sequence of production rules recursively defined by

$$\mathit{rules}(t) \mathbin{++} \mathit{grammar}(\theta(t_1^1)) \mathbin{++} \cdots \mathbin{++} \mathit{grammar}(\theta(t_{m_k}^k))$$

where the ++ operation concatenates sequences of production rules, removing second and later occurrences of duplicate production rules.

We call each nonterminal in a set of production rules a *position*, since we will use them to describe positions in terms; each position is the root of one of the term's subterms. We also sometimes call positions *nodes*, since they correspond to nodes in type trees.

Example 2. Consider the type $\mathsf{list(abc)}$, then the corresponding grammar is

$$\begin{array}{rcl} \mathsf{list(abc)} & \to & [\,] \; ; \; [\mathsf{abc}|\mathsf{list(abc)}] \\ \mathsf{abc} & \to & a \; ; \; b \; ; \; c \end{array}$$

There are two nonterminals and thus two positions in the grammar: $\mathsf{list(abc)}$ and abc.

Mode inference and checking takes place after type checking, so we assume that we know the type of every variable appearing in the program.

2.4 Instantiations and Modes

An *instantiation* describes the binding pattern of a variable at a particular point in the execution of the program. A *mode* is a mapping from one instantiation to another which describes how the instantiation of a variable changes over the execution of a goal.

Instantiations are also defined using tree grammars. The differences are that no instantiation associated with a predicate involves instantiation parameters (no polymorphic modes—although these are a possible extension), and there are two base instantiations: free and ground representing completely uninitialized variables, and completely bound terms. Instantiation expressions are terms in $\tau(\Sigma_{inst}, V_{inst})$.

Definition 4. An *instantiation definition* for $g \in \Sigma_{inst}$ is of the form:

`:- inst` $g(w_1, ..., w_n)$`==`$\mathsf{bound}(g_1(i_1^1, ..., i_{m_1}^1); \cdots; g_k(i_1^k, ..., i_{m_k}^k))$.

where $w_1, \ldots, w_n$ are distinct instantiation parameters, $\{g_1/m_1, \ldots, g_k/m_k\} \subseteq \Sigma_{tree}$ are distinct tree constructors, and $i_1^1, \ldots, i_{m_k}^k$ are instantiation expressions in $\tau(\Sigma_{inst} \cup \{\mathsf{free}, \mathsf{ground}\}, V_{inst})$.

We can associate a set of production rules *rules*(i) with an instantiation expression i just as we do for type expressions. For the base instantiations we define *rules*(free) = *rules*$(\mathsf{ground}) = \emptyset$.

Example 3. For example the instantiation definition

```
:- inst list_skel(I) == bound([] ; [I | list_skel(I)] ).
```

defines an instantiation list_skel(I). A variable with this instantiation must be bound, either to an empty list (`[]`), or to a cons cell whose first argument has the instantiation given by instantiation variable I and whose tail also has the instantiation list_skel(I). For example list_skel(free) describes a list in which the elements are all free.

Each instantiation is usually intended to be used with a specific type, e.g. list_skel(I) with `list(T)`, and it normally lists all the function symbols that variables of that type can be bound to. Instantiations that do not do so, such as

```
:- inst non_empty_skel(I) == bound([I | list_skel(I)] ).
```

represent a kind of subtyping; a variable whose instantiation is non_empty_skel(free) cannot be bound to `[]`.

Definition 5. A *mode* $i \gg f$ is a mapping from an initial instantiation i to a final instantiation f. Common modes have shorthand expressions, e.g. in = ground >> ground and out = free >> ground. A goal that changes the instantiation state of a position from free to ground is said to *produce* or *bind* that position; a goal that requires the initial instantiation state of a position to be ground is said to *consume* that position.

3. CURRENT MODE ANALYSIS SYSTEM

The mode analysis algorithm currently used by the Mercury compiler is based on abstract interpretation. The abstract domain maps each program variable to an instantiation. Before mode analysis, the compiler creates a separate *procedure* for each mode of usage of each predicate. It then analyses each procedure separately.

Starting with the initial instantiation states for each argument given by the mode declaration, the analysis traverses the procedure body goal determining the instantiation state of each variable at each point in the goal. When traversing conjunctions, if a conjunct is not able to be scheduled because it needs as input a variable that is not sufficiently bound, it is delayed and tried again later. Once the goal has been analysed, if there are no unschedulable subgoals and the computed final instantiation states of the arguments match their final instantiation states in the mode declaration, the procedure is determined to be *mode correct*. See [8, 18] for more details (although those papers talk about other languages, the approach in Mercury is similar).

The system is also able to do mode inference for predicates which are not exported from their defining module, using a top down traversal of the module. However, to prevent combinatorial explosion, the mode analysis algorithm does not reorder conjunctions when performing inference; when it arrives at a call, it assumes that the called predicate is supposed to be able to run given only the variables that have been bound to its left.

The mode analysis system has several tasks that it does all at once. It must (1) determine the producer and the consumers of each variable; (2) reorder conjunctions to ensure that all consumers of a variable are executed after its producer; and (3) ensure that sub-typing constraints are met. This leads to a very complicated implementation. One of the aims of our constraint-based approach is to simplify the analysis by splitting these tasks up into distinct phases which can be done separately.

3.1 Limitations

There are two main problems with the above approach.

The first is that it does not keep track of aliasing information. This has two consequences. First, without may-alias information about bound nodes we cannot handle unique data structures in a nontrivial manner; in particular, we cannot implement compile-time garbage collection. Second, without must-alias information about free nodes, we cannot do accurate mode analysis of code that manipulates partially instantiated data structures.

Partially instantiated data structures are data structures that contain free variables. They are useful when one wants different parts of a data structure to be filled in by different parts of a program.

Example 4. Consider the following small program.

```
:- pred length(list(int), int).
:- mode length(free >> list_skel(free), in) is det.
length(L, N) :-
    (   L = [], N = 0
    ;   L = [_ | K], M = N - 1, length(K, M)
    ).
:- pred iota(list(int), int).
:- mode iota(list_skel(free) >> ground, in) is det.
iota(L, X) :-
    (   L = []
    ;   L = [H | T], H = X, Y = X + 1, iota(T, Y)
    ).
```

In the goal `length(L, 10), iota(L, 3)`, `length/2` constructs the skeleton of a list with a specified length and `iota/2` fills in the elements of the list. The current system is unable to verify the mode correctness of the second disjunct of `iota/2`. One problem is that this disjunct sets up an alias between the variable `H` and the first element of `L` (which are both initially free variables), and then instantiates `H` by unifying it with the second argument. Without information about the aliasing between `H` and the first element of `L`, the mode checker is unable to determine that this also instantiates the first element of `L`.

The second problem is that the absence of reordering during mode inference prevents many correct modes from being detected.

Example 5. Consider mode inference for the predicate

```
:- pred append3(list(T), list(T), list(T), list(T)).
app3(A,B,C,ABC) :- append(A,B,AB), append(AB,C,ABC).
```

Inference will find only the mode `app3(in,in,in,out)`; it will not find the mode `app3(out,out,out,in)`.

The reordering restriction cannot simply be lifted, because without it, the current mode inference algorithm can explore arbitrarily large numbers of modes, which will in fact be useless, since it does not "look ahead" to see if the modes inferred for a called predicate will be useful in constructing the desired mode for the current predicate.

4. SIMPLIFIED EXAMPLE

The motivation for our constraint based mode analysis system is to avoid the problems in the current system. In order to do, so we break the mode analysis problem into phases. The first phase determines which subgoals produce which variables, while the second uses that information to determine an execution order for the procedure. For now, we will focus on the first task; we will return to the second in Section 6.

For ease of explanation, we will first show a simplified form of our approach. This simplified form requires variables to be instantiated all at once, (i.e. the only two instantiation states it recognizes are free and ground) and requires all variables to eventually reach the ground state. This avoids the complexities that arise when different parts of variables are bound at different times, or some parts are left unbound. We will address those complexities when we give the full algorithm in Section 5.

4.1 Constraint Generation

The algorithm associates several constraint variables with each program variable. With every program variable V, we associate a family of constraint variables of the form V_p; V_p is true iff V is produced by the goal at path p in the predicate body.

We explain the algorithm using `append/3`. The code below is shown after transformation by the compiler into superhomogeneous form as described in Section 2.1. We also ensure that each variable appears in at most one argument of one functor by adding extra unifications if necessary.

```
:- pred append(list(T),list(T),list(T)).
append(A,B,C) :-
    (   A = [], B = C
    ;   A = [AH | AT], C = [CH | CT], AH = CH,
        append(AT, B, CT)
    ).
```

We examine the body and generate constraints from it. The body is a disjunction, so the constraints we get simply specify, for each variable nonlocal to the disjunction, that if the disjunction produces a variable, then all disjuncts must produce that variable, while if the disjunction does not produce a variable, then no disjunct may produce that variable. For `append`, this is expressed by the constraints:

$$(A_\lambda \leftrightarrow A_{\mathtt{d}_1}) \wedge (B_\lambda \leftrightarrow B_{\mathtt{d}_1}) \wedge (C_\lambda \leftrightarrow C_{\mathtt{d}_1}) \wedge \\ (A_\lambda \leftrightarrow A_{\mathtt{d}_2}) \wedge (B_\lambda \leftrightarrow B_{\mathtt{d}_2}) \wedge (C_\lambda \leftrightarrow C_{\mathtt{d}_2})$$

We then process the disjuncts one after another. Both disjuncts are conjunctions. When processing a conjunction, our algorithm considers each variable occurring in the conjunction that has more than one potential producer. If a variable is nonlocal to the conjunction, then it may be produced either inside or outside the conjunction; if a variable is shared by two or more conjuncts, then it may be produced by any one of those conjuncts. The algorithm generates constraints that make sure that each variable has exactly one producer. If the variable is local, the constraints say that exactly one conjunct must produce the variable. If the variable is nonlocal, the constraints say that at most one conjunct may produce the variable.

In the first disjunct, there are no variables shared among the conjuncts, so the only constraints we get are those ones that say that each nonlocal is produced by the conjunction iff it is produced by the only conjunct in which it appears:

$$(A_{\mathtt{d}_1} \leftrightarrow A_{\mathtt{d}_1.\mathtt{c}_1}) \wedge (B_{\mathtt{d}_1} \leftrightarrow B_{\mathtt{d}_1.\mathtt{c}_2}) \wedge (C_{\mathtt{d}_1} \leftrightarrow C_{\mathtt{d}_1.\mathtt{c}_2})$$

The first conjunct in the first disjunct yields no nontrivial constraints. Intuitively, the lack of constraints from this goal reflects the fact that `A = []` can be used both to produce `A` and to test its value.

The second conjunct in the first disjunct yields one constraint, which says that the goal `B = C` can be used to produce at most one of `B` and `C`:

$$\neg(B_{\mathtt{d}_1.\mathtt{c}_2} \wedge C_{\mathtt{d}_1.\mathtt{c}_2})$$

For the second disjunct, we generate constraints analogous to those for the first conjunct for the nonlocal variables. But this disjunct, unlike the first, contains some shared local variables: `AH`, `CH`, `AT` and `CT`, each of which appears in two conjuncts. Constraints for these variables state that each of these variables must be produced by exactly one of the conjuncts in which it appears.

$$(A_{\mathtt{d}_2} \leftrightarrow A_{\mathtt{d}_2.\mathtt{c}_1}) \wedge (B_{\mathtt{d}_2} \leftrightarrow B_{\mathtt{d}_2.\mathtt{c}_4}) \wedge (C_{\mathtt{d}_2} \leftrightarrow C_{\mathtt{d}_2.\mathtt{c}_2}) \wedge \\ \neg(AH_{\mathtt{d}_2.\mathtt{c}_1} \leftrightarrow AH_{\mathtt{d}_2.\mathtt{c}_3}) \wedge \neg(CH_{\mathtt{d}_2.\mathtt{c}_2} \leftrightarrow CH_{\mathtt{d}_2.\mathtt{c}_3}) \wedge \\ \neg(AT_{\mathtt{d}_2.\mathtt{c}_1} \leftrightarrow AT_{\mathtt{d}_2.\mathtt{c}_4}) \wedge \neg(CT_{\mathtt{d}_2.\mathtt{c}_2} \leftrightarrow CT_{\mathtt{d}_2.\mathtt{c}_4})$$

The first conjunct in the second disjunct shows how we handle unifications of the form $X = f(Y_1, ..., Y_n)$ where $n > 0$. The key to understanding the behavior of our algorithm in this case is knowing that it is trying to decide between only two alternatives: either this unification takes all the Y_is as input and produces X, or it takes X as input and produces all the Y_is. This is contrary to most people's experience, because in real programs, unifications of this form can also be used in ways that make no bindings, or that produce only a subset of the Y_i. However, using this unification in a way that requires both X and some or all of the Y_i to be input is possible only if those Y_i have producers outside this unification. When we transform the program into superhomogeneous form, we make sure that each unification of this form has fresh variables on the right hand side. So if a Y_i could have such a producer, it would be replaced on the right hand side of the unification with a new variable Y_i', with the addition of a new unification $Y_i = Y_i'$. That is, we convert unifications that take both X and some or all of the Y_i to be input into unifications that take only X as input, and produce all the variables on the right hand side.

If some of the variables on the right hand side appear only once then those variables must be unbound, and using this unification to produce X would create a nonground term. Since the simplified approach does not consider nonground terms, in such cases it generates an extra constraint that requires X to be input to this goal.

In the case of `A = [AH | AT]`, both `AH` and `AT` appear elsewhere so we get the constraints:

$$(AH_{\mathtt{d}_2.\mathtt{c}_1} \leftrightarrow AT_{\mathtt{d}_2.\mathtt{c}_1}) \wedge \neg(A_{\mathtt{d}_2.\mathtt{c}_1} \wedge AH_{\mathtt{d}_2.\mathtt{c}_1})$$

The first says that either this goal produces all the variables on the right hand side, or it produces none of them. In conjunction with the first, the second constraint says that the goal cannot produce both the variable on the left hand side, and all the variables on the right hand side.

The constraints we get for `C = [CH | CT]` are analogous:

$$(CH_{\mathtt{d}_2.\mathtt{c}_2} \leftrightarrow CT_{\mathtt{d}_2.\mathtt{c}_2}) \wedge \neg(C_{\mathtt{d}_2.\mathtt{c}_2} \wedge CH_{\mathtt{d}_2.\mathtt{c}_2})$$

The equation `AH = CH` acts just as the equation in the first disjunct, generating:

$$\neg(AH_{\mathtt{d}_2.\mathtt{c}_3} \wedge CH_{\mathtt{d}_2.\mathtt{c}_3})$$

The last conjunct is a call, in this case a recursive call. We assume that all calls to a predicate in the same SCC as the caller have the same mode.[1] This means that the call produces its ith argument iff the predicate body produces its ith argument. This leads to one constraint for each argument position:

$$(AT_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow A_\lambda) \wedge (B_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow B_\lambda) \wedge (CT_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow C_\lambda)$$

This concludes the set of constraints generated by our algorithm for `append`.

4.2 Inference and Checking

The constraints we generate for a predicate can be used to infer its modes. Projecting onto the head variables, the constraint set we just built up has five different solutions, so `append` has five modes:

$\neg A_\lambda \wedge \neg B_\lambda \wedge \neg C_\lambda$	`append(in, in, in)`
$\neg A_\lambda \wedge B_\lambda \wedge \neg C_\lambda$	`append(in, out, in)`
$A_\lambda \wedge \neg B_\lambda \wedge \neg C_\lambda$	`append(out, in, in)`
$A_\lambda \wedge B_\lambda \wedge \neg C_\lambda$	`append(out, out, in)`
$\neg A_\lambda \wedge \neg B_\lambda \wedge C_\lambda$	`append(in, in, out)`

Of these five modes, two (`append(in, in, out)` and `append(out, out, in)`) are what we call *principal* modes. The other three are *implied* modes, because their existence is implied by the existence of the principal modes; changing the mode of an argument from `out` to `in` makes the job of a predicate strictly easier. In the rest of the paper, we assume that every predicate's set of modes is *downward closed*, which means that if it contains a mode *pm*, it also contains all the modes implied by *pm*. In practice, the compiler generates code for a mode only if it is declared or it is a principal mode, and synthesizes the other modes in the caller by renaming variables and inserting extra unifications. This synthesis does the superhomogeneous form equivalent of replacing `append([1], [2], [3])` with `append([1], [2], X), X = [3]`.

Each solution also implicitly assign modes to the primitive goals in the body, by specifying where each variable is produced. For example, the solution that assigns true to the constraint variables C_λ, C_{d_1}, $C_{\mathsf{d}_1.\mathsf{c}_2}$, C_{d_2}, $AH_{\mathsf{d}_2.\mathsf{c}_1}$, $AT_{\mathsf{d}_2.\mathsf{c}_1}$, $C_{\mathsf{d}_2.\mathsf{c}_2}$, $CH_{\mathsf{d}_2.\mathsf{c}_3}$, $CT_{\mathsf{d}_2.\mathsf{c}_4}$ and false to all others, which corresponds to the mode `append(in,in,out)`, also shows that `A = [AH|AT]` is a deconstruction (i.e. uses the fields of `A` to define `AH` and `AT`) while `C = [CH|CT]` is a construction (i.e. it binds `C` to a new heap cell).

In most cases, the values of the constraint variables of the head variables uniquely determine the values of all the other constraint variables. Sometimes, there is more than one set of value assignments to the constraint variables in the body that is consistent with a given value assignment for the constraint variables in the head. In such cases, the compiler can choose the assignment it prefers.

5. FULL MODE INFERENCE

5.1 Expanded Grammars

We now consider the problem of handling programs in which different parts of a variable may be instantiated by different goals. We need to ensure that if two distinct positions in a variable may have different instantiation behaviour, then we have a way of referring to each separately. Hence we need to expand the type grammar associated with that variable.

We begin with an empty grammar and with the original code of the predicate expressed in superhomogeneous normal form. We modify both the grammar and the predicate body during the first stage of mode analysis.

If the unification $X = f(A_1, \ldots, A_n)$ appears in the definition of the predicate, then

- if X has no grammar rule for functor f/n, add a rule $X \rightarrow f(A_1, \ldots, A_n)$, and for each A_i which already occurs in a grammar rule or in the head of the clause, replace each occurrence of A_i in the program by A_i' and add an equation $A_i = A_i'$.
- if X has a grammar rule $X \rightarrow f(B_1, \ldots, B_n)$ replace the equation by $X = f(B_1, \ldots, B_n), A_1 = B_1, \ldots, A_n = B_n$.

This process may add equations of the form $X = Y$ where one of X and Y occurs nowhere else. Such equations can be safely removed.

After processing all such unifications, add a copy of the rules in $rules(t)$ for each grammar variable V of type t which does not have them all.

Example 6. The superhomogeneous form of the usual source code for append has (some variant of)

```
A=[AH | AT], C=[AH | CT], append(AT,B,CT)
```

as its second clause, which our algorithm replaces with

```
A=[AH | AT], C=[CH | CT], AH = CH, append(AT,B,CT)
```

yielding the form we have shown in Section 4. The expanded grammar I computed for append is

$$\begin{array}{lllllll} A & \rightarrow & [] \; ; \; [AH|AT] & \quad AT & \rightarrow & [] \; ; \; [AE|AT] \\ B & \rightarrow & [] \; ; \; [BE|B] & & & \\ C & \rightarrow & [] \; ; \; [CH|CT] & \quad CT & \rightarrow & [] \; ; \; [CE|CT] \end{array}$$

The nonterminals of this grammar constitute the set of positions for which we will create constraint variables when we generate constraints from the predicate body, so from now we will use “nonterminal” and “position” (as well as “node”) interchangeably. Note that there is a nonterminal denoting the top-level functor of every variable, and that some variables (e.g. `A`) have other nonterminals denoting some of their non-top-level functors as well. Note also that a nonterminal can fulfill both these functions, when one variable is *strictly* part of another. For example, the nonterminal AH stands both for the variable `AH` and the first element in any list bound to variable `A`, but the variables `B` and `C`, which are unified on some computation paths, each have their own nonterminal.

A predicate needs three Boolean variables for each position V: (a) V_{in} is true if the position is produced outside the predicate, (b) V_λ is true if the position is produced inside the predicate, and (c) V_{out} is true if the position is produced *somewhere* (either inside or outside the predicate). Note that $V_{\mathsf{out}} \leftrightarrow V_{\mathsf{in}} \vee V_\lambda$.

Definition 6. Let $\alpha(p/n)$ be the tuple $\langle H_1, \ldots, H_n \rangle$ of head variables (i.e. formal parameters) for predicate p/n.

[1]This assumption somewhat restricts the set of allowable well-moded programs. However, we have not found this to be an unreasonable restriction in practice. We have not come across any cases in typical Mercury programs where one would want to make a recursive call in a different mode.

Definition 7. For an expanded grammar I and position X, we define the *immediate descendants* of X as

$$\delta_I(X) = \bigcup_{X \to f(Y_1,\ldots,Y_n) \,\in\, I} \{Y_1, \ldots Y_n\}$$

and the set of positions *reachable* from X as

$$\rho_I(X) = \{X\} \cup \bigcup_{X \to f(Y_1,\ldots,Y_n) \,\in\, I} \big\{\rho_I(Y) \;\big|\; Y \in \{Y_1, \ldots, Y_n\}\big\}$$

When we are generating constraints between two variables (which will always be of the same type) we will need to be able to refer to positions within the two variables which *correspond* to each other, as e.g. AH and CH denote corresponding positions inside `A` and `C`. The notion of correspondence which we use allows for the two variables to have been expanded to different extents in the expanded grammar. For example, `A` has more descendant nonterminals in `append`'s grammar than `B`, even though they have the same type. In the unification `A = B`, the nonterminal B would correspond to AT as well as A.

Definition 8. For expanded grammar I and positions X and Y, we define the set of pairs of *corresponding nodes* in X and Y as

$$\chi_I(X,Y) = \big\{\langle X, Y\rangle\big\} \;\cup\; \bigcup_{\langle V,W\rangle \,\in\, \chi'_I(X,Y)} \chi_I(V,W)$$

where

$$\chi'_I(X,Y) = \begin{cases} \bigcup_{\substack{X \to f(V_1,\ldots,V_n) \,\in\, I \\ Y \to f(W_1,\ldots,W_n) \,\in\, I}} \big\{\langle V_1, W_1\rangle, \ldots, \langle V_n, W_n\rangle\big\} & \text{if } X \in \mathsf{lhs}(I) \text{ and } Y \in \mathsf{lhs}(I), \\ \bigcup_{X \to f(V_1,\ldots,V_n) \,\in\, I} \big\{\langle V_1, Y\rangle, \ldots, \langle V_n, Y\rangle\big\} & \text{if } X \in \mathsf{lhs}(I) \text{ and } Y \notin \mathsf{lhs}(I), \\ \bigcup_{Y \to f(W_1,\ldots,W_n) \,\in\, I} \big\{\langle X, W_1\rangle, \ldots, \langle X, W_n\rangle\big\} & \text{if } X \notin \mathsf{lhs}(I) \text{ and } Y \in \mathsf{lhs}(I), \\ \emptyset & \text{otherwise} \end{cases}$$

and $\mathsf{lhs}(I) = \{X \mid \exists f/n \in \Sigma_{tree}.X \to f(Y_1,\ldots,Y_n) \in I\}$ For convenience, we also define χ for a pair of n-tuples:

$$\chi_I(\langle X_1, \ldots, X_n\rangle, \langle Y_1, \ldots, Y_n\rangle) = \bigcup_{i=1}^{n} \chi_I(X_i, Y_i)$$

Definition 9. Given an expanded grammar I and a rule $X \to f(Y_1, \ldots, Y_n) \in I$ we say that X is the *parent node* of each of the nodes $Y_1, \ldots, Y_n$.

5.2 Mode Inference Constraints

We ensure that no variable occurs in more than one predicate, renaming them as necessary. We construct an expanded grammar I for P, the program module being compiled. We next group the predicates in the module into strongly-connected components (SCCs), and process these SCCs bottom up, creating a function C_{SCC} for each SCC. This represents the Boolean constraints that we generate for the predicates in that SCC. The remainder of this section defines C_{SCC}.

The constraint function $C_{\mathsf{SCC}}(I, S)$ for an SCC S is the conjunction of the constraint functions $C_{\mathsf{Pred}}(I, p/n)$ we generate for all predicates p/n in that SCC, i.e. $C_{\mathsf{SCC}}(I, S)$ is

$$\bigwedge_{p/n \in S, p(H_1,\ldots,H_n)\text{:-}G \in P} C_{\mathsf{Pred}}(I, p(H_1, \ldots, H_n)\text{:-}G)$$

The constraint function we *infer* for a predicate p/n is the constraint function of its SCC, i.e. $C_{\mathsf{Inf}}(I, p/n) = C_{\mathsf{SCC}}(I, S)$ for each $p/n \in S$. $C_{\mathsf{Inf}}(I, p/n)$ may be stricter than $C_{\mathsf{Pred}}(I, p/n)$ if p/n is not alone in its SCC. For predicates defined in other modules, we derive their C_{Inf} from their mode declarations using the mechanism we will describe in Section 5.3.

C_{Pred} itself is the conjunction of two functions: C_{Struct}, the structural constraints relating `in` and `out` variables, and C_{Goal}, the constraints for the predicate body goal:

$$C_{\mathsf{Pred}}(I, (p(H_1, \ldots, H_n)\text{ :- }G)) \;=\; C_{\mathsf{Struct}}(I, \{H_1, \ldots, H_n\}, \nu(G)) \;\wedge\; C_{\mathsf{Goal}}(I, \lambda, G)$$

We define C_{Struct} and C_{Goal} below.

5.2.1 Structural Constraints

V_{in} is the proposition that V is bound at call. V_{out} is the proposition that V is bound at return. V_λ is the proposition that V is bound by this predicate. These constraints relate the relationships between the above variables and relationships of boundedness at different times.

If a node is not reachable from one of the predicate's argument variables, then it cannot be bound at call.

A node is bound at return if it is bound at call or it is produced within the predicate body. A node may not be both bound at call and produced in the predicate body.

If a node is bound at call then its parent node must also be bound at call. Similarly, if a node is bound at return then its parent node must also be bound at return.

$$\begin{aligned} C_{\mathsf{Struct}}(I, HV, NL) \;=\; & \bigwedge_{V \in \rho_I(NL) \setminus \rho_I(HV)} \neg V_{\mathsf{in}} \\ \wedge\; & \bigwedge_{V \in NL} \big((V_{\mathsf{out}} \leftrightarrow V_{\mathsf{in}} \vee V_\lambda) \;\wedge\; \neg(V_{\mathsf{in}} \wedge V_\lambda)\big) \\ \wedge\; & \bigwedge_{V \in \rho_I(NL)} \bigwedge_{D \in \delta_I(V)} \big\{(D_{\mathsf{in}} \to V_{\mathsf{in}}) \;\wedge\; (D_{\mathsf{out}} \to V_{\mathsf{out}})\big\} \end{aligned}$$

Example 7. For `append`, the structural constraints are:

$$\begin{array}{l} A_{\mathsf{out}} \leftrightarrow A_{\mathsf{in}} \vee A_\lambda, \neg(A_{\mathsf{in}} \wedge A_\lambda), \\ AH_{\mathsf{out}} \leftrightarrow AH_{\mathsf{in}} \vee AH_\lambda, \neg(AH_{\mathsf{in}} \wedge AH_\lambda), \\ AT_{\mathsf{out}} \leftrightarrow AT_{\mathsf{in}} \vee AT_\lambda, \neg(AT_{\mathsf{in}} \wedge AT_\lambda), \\ AE_{\mathsf{out}} \leftrightarrow AE_{\mathsf{in}} \vee AE_\lambda, \neg(AE_{\mathsf{in}} \wedge AE_\lambda), \\ B_{\mathsf{out}} \leftrightarrow B_{\mathsf{in}} \vee B_\lambda, \neg(B_{\mathsf{in}} \wedge B_\lambda), \\ BE_{\mathsf{out}} \leftrightarrow BE_{\mathsf{in}} \vee BE_\lambda, \neg(BE_{\mathsf{in}} \wedge BE_\lambda), \\ C_{\mathsf{out}} \leftrightarrow C_{\mathsf{in}} \vee C_\lambda, \neg(C_{\mathsf{in}} \wedge C_\lambda), \\ CH_{\mathsf{out}} \leftrightarrow CH_{\mathsf{in}} \vee CH_\lambda, \neg(CH_{\mathsf{in}} \wedge CH_\lambda), \\ CT_{\mathsf{out}} \leftrightarrow CT_{\mathsf{in}} \vee CT_\lambda, \neg(CT_{\mathsf{in}} \wedge CT_\lambda), \\ CE_{\mathsf{out}} \leftrightarrow CE_{\mathsf{in}} \vee CE_\lambda, \neg(CE_{\mathsf{in}} \wedge CE_\lambda), \\ AH_{\mathsf{in}} \to A_{\mathsf{in}}, AH_{\mathsf{out}} \to A_{\mathsf{out}}, \\ AT_{\mathsf{in}} \to A_{\mathsf{in}}, AT_{\mathsf{out}} \to A_{\mathsf{out}}, \\ AE_{\mathsf{in}} \to AT_{\mathsf{in}}, AE_{\mathsf{out}} \to AT_{\mathsf{out}}, \\ BE_{\mathsf{in}} \to B_{\mathsf{in}}, BE_{\mathsf{out}} \to B_{\mathsf{out}}, \\ CH_{\mathsf{in}} \to C_{\mathsf{in}}, CH_{\mathsf{out}} \to C_{\mathsf{out}}, \\ CT_{\mathsf{in}} \to C_{\mathsf{in}}, CT_{\mathsf{out}} \to C_{\mathsf{out}}, \\ CE_{\mathsf{in}} \to CT_{\mathsf{in}}, CE_{\mathsf{out}} \to CT_{\mathsf{out}} \end{array}$$

5.2.2 Goal Constraints

There is a Boolean variable V_p for each path p which contains a program variable X such that $V \in \rho_I(X)$. This variable represents the proposition that position V is produced in the goal referred to by the path p.

The constraints we generate for each goal fall into two categories: general constraints that apply to all goal types (C_{Gen}), and constraints specific to each goal type (C_{Goal}). The complete set of constraints for a goal (C_{Comp}) is the conjunction of these two sets.

The general constraints have two components. The first, C_{Local}, says that any node reachable from a variable that is local to a goal will be bound at return iff it is produced by that goal. The second, C_{Ext}, says that a node reachable from a variable V that is external to the goal G (i.e. does not occur in G) cannot be produced by G. The conjunction in the definition of C_{Ext} could be over all the variables in the predicate that do not occur in G. However, if a variable V does not occur in G's parent goal, then the parent's C_{Goal} constraints won't mention V, so there is no point in creating constraint variables for V for G.

$$\begin{array}{lcl} C_{\mathsf{Comp}}(I,p,G) &=& C_{\mathsf{Gen}}(I,p,G) \ \wedge \ C_{\mathsf{Goal}}(I,p,G) \\ C_{\mathsf{Gen}}(I,p,G) &=& C_{\mathsf{Local}}(I,P,G) \ \wedge \ C_{\mathsf{Ext}}(I,p,G) \\ C_{\mathsf{Local}}(I,p,G) &=& \bigwedge_{V \in \rho_I(\nu(G)\setminus\eta(G))} (V_p \leftrightarrow V_{\mathsf{out}}) \\ C_{\mathsf{Ext}}(I,p,G) &=& \bigwedge_{V \in \rho_I(\eta(\mathsf{parent}(p))\setminus\nu(G))} \neg V_p \end{array}$$

5.2.3 Compound Goals

The constraints we generate for each kind of compound goal (conjunction, disjunction and if-then-else) are shown in Figure 1. In each case, the goal-type-specific constraints are conjoined with the complete set of constraints from all the subgoals.

In each conjunctive goal a position can be produced by at most one conjunct.

In each disjunctive goal a node either must be produced in each disjunct or not produced in each disjunct.

For an if-then-else goal a node is produced by the if-then-else if and only if it is produced in either the condition, the then branch or the else branch. A node may not be produced by both the condition and the then branch. Nodes reachable from variables that are nonlocal to the if-then-else must not be produced by the condition. If a node reachable from a nonlocal variable is produced by the then branch then it must also be produced by the else branch, and vice versa.

5.2.4 Atomic Goals

Due to space considerations, we leave out discussion of higher-order terms which may be handled by a simple extension to the mode-checking algorithm.

We consider three kinds of atomic goals:

1. Unifications of the form $X = Y$.
2. Unifications of the form $X = f(Y_1, \ldots, Y_n)$ where $X \rightarrow f(Y_1, \ldots, Y_n) \in I$.
3. Calls of the form $q(Y_1, \ldots, Y_n)$.

A unification of the form $X = Y$ may produce at most one of each pair of corresponding nodes. Mercury does not allow aliases to exist between unbound nodes so each node reachable from a variable involved in a unification must be produced somewhere.[2] For a unification $X = Y$ at goal path p, the constraints $C_{\mathsf{Goal}}(I,\ p,\ X = Y)$ are

$$\bigwedge_{(V,W) \in \chi_I(X,Y)} \big(V_{\mathsf{out}} \ \wedge \ W_{\mathsf{out}} \ \wedge \ \neg(V_p \ \wedge \ W_p)\big)$$

Example 8. For the unification `B = C` in `append` at goal path $\mathsf{d}_1.\mathsf{c}_2$ the constraints generated are:

$$\begin{array}{l} B_{\mathsf{out}} \wedge C_{\mathsf{out}} \wedge \neg(B_{\mathsf{d}_1.\mathsf{c}_2} \wedge C_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ BE_{\mathsf{out}} \wedge CH_{\mathsf{out}} \wedge \neg(BE_{\mathsf{d}_1.\mathsf{c}_2} \wedge CH_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ B_{\mathsf{out}} \wedge CT_{\mathsf{out}} \wedge \neg(B_{\mathsf{d}_1.\mathsf{c}_2} \wedge CT_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ BE_{\mathsf{out}} \wedge CE_{\mathsf{out}} \wedge \neg(BE_{\mathsf{d}_1.\mathsf{c}_2} \wedge CE_{\mathsf{d}_1.\mathsf{c}_2}) \end{array}$$

[2] During the goal scheduling phase, we further require that a node must be produced before it is aliased to another node. These two restrictions together disallow most uses of partially instantiated data structures. In the future, when the Mercury implementation can handle the consequences, we would like to lift both restrictions.

A unification of the form $X = f(Y_1, \ldots, Y_n)$ at path p does not produce any of the arguments $Y_1, \ldots, Y_n$. X must be produced somewhere (either at p or somewhere else). The constraints $C_{\mathsf{Goal}}(I,\ p,\ X = f(Y_1, \ldots, Y_n))$ are

$$X_{\mathsf{out}} \ \wedge \ \bigwedge_{V \in \rho_I(\{Y_1,\ldots,Y_n\})\setminus\{X\}} \neg V_p$$

Example 9. For the unification `A = [AH|AT]` in `append` at goal path $\mathsf{d}_2.\mathsf{c}_1$ the constraints generated are:

$$A_{\mathsf{out}} \wedge \neg AH_{\mathsf{d}_2.\mathsf{c}_1} \wedge \neg AT_{\mathsf{d}_2.\mathsf{c}_1} \wedge \neg AE_{\mathsf{d}_2.\mathsf{c}_1}$$

A call $q(Y_1, \ldots, Y_n)$ will constrain the nodes reachable from the arguments of the call. For predicates in the current SCC, we only allow recursive calls that are in the same mode as the caller. The constraints $C_{\mathsf{Goal}}(I,\ p,\ q(Y_1, \ldots, Y_n))$ are

$$\bigwedge_{\langle V,W\rangle \in \chi_I(\alpha(q/n),\langle Y_1,\ldots,Y_n\rangle)} \big((V_\lambda \leftrightarrow W_p) \ \wedge \ (V_{\mathsf{in}} \rightarrow W_{\mathsf{out}})\big)$$

The first part ensures that the call produces the position if the position is produced by the predicate in the SCC. The second part ensures that call variable W is produced somewhere if it is required to be bound at call to the call (V_{in}). Since if V_{in} is true V_λ will not be true, we can't mistakenly use this call site to produce W.

Example 10. For the recursive call `append(AT,B,CT)` at goal path $\mathsf{d}_2.\mathsf{c}_4$ in `append` the constraints generated on the first argument are:

$$\begin{array}{l} (A_\lambda \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (A_{\mathsf{in}} \rightarrow AT_{\mathsf{out}}) \ \wedge \\ (AH_\lambda \leftrightarrow AE_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AH_{\mathsf{in}} \rightarrow AE_{\mathsf{out}}) \ \wedge \\ (AT_\lambda \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AT_{\mathsf{in}} \rightarrow AT_{\mathsf{out}}) \ \wedge \\ (AE_\lambda \leftrightarrow AE_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AE_{\mathsf{in}} \rightarrow AE_{\mathsf{out}}) \end{array}$$

For calls to predicates in lower SCCs the constraints are similar, but we must existentially quantify the head variables so that it is possible to call the predicate in different modes from different places within the current SCC:

$$\begin{array}{l} C_{\mathsf{Goal}}(I,\ p,\ q(Y_1,\ldots,Y_n)) \ = \ \exists \rho_I(\alpha(q/n)) \ : \ C_{\mathsf{Inf}}(I,\ q/n) \\ \wedge \bigwedge_{\langle V,W\rangle \in \chi_I(\alpha(q/n),\langle Y_1,\ldots,Y_n\rangle)} \big((V_\lambda \leftrightarrow W_p) \ \wedge \ (V_{\mathsf{in}} \rightarrow W_{\mathsf{out}})\big) \end{array}$$

5.3 Mode Declaration Constraints

For any predicate which has modes declared, the mode analysis system should check these declarations against the inferred mode information. This involves generating a set of constraints for the mode declarations and ensuring that they are consistent with the constraints generated from the predicate body.

The declaration constraint $C_{\mathsf{Decls}}(I,\mathcal{D})$ for a predicate with a set of mode declarations $\mathcal{D}$ is the disjunction of the constraints $C_{\mathsf{Decl}}(I,d)$ for each mode declaration d:

$$C_{\mathsf{Decls}}(I,\mathcal{D}) = \bigvee_{d \in \mathcal{D}} C_{\mathsf{Decl}}(I,d)$$

The constraint $C_{\mathsf{Decl}}(I,d)$ for a mode declaration $d = p(m_1, \ldots, m_n)$ for a predicate $p(H_1, \ldots , H_n)$ is the conjunction of the constraints $C_{\mathsf{Arg}}(I,m,H)$ for each argument mode m with corresponding head variable H:

$$\bigwedge_{i=1,\ldots,n} \Big\{ C_{\mathsf{Arg}}(I,m_i,H_i) \Big\} \wedge C_{\mathsf{Struct}}(I,\{H_1,\ldots,H_n\},\emptyset)$$

The structural constraints are used to determine the H_λ variables from H_{in} and H_{out}.

The constraint $C_{\mathsf{Arg}}(I,m,H)$ for an argument mode $m = i \mathbin{>\!>} f$ of head variable H is the conjunction of the constraint

$$
\begin{array}{ll}
C_{\mathsf{Goal}}(I,\ p,\ (G_1,\ldots,G_n)) & = \bigwedge_{i=1}^{n} C_{\mathsf{Comp}}(I,\ p.\mathsf{c}_i,\ G_i) \quad \wedge \quad C_{\mathsf{Conj}}(I,\ p,\ (G_1,\ldots,G_n)) \\
C_{\mathsf{Goal}}(I,\ p,\ (G_1;\ldots;G_n)) & = \bigwedge_{i=1}^{n} C_{\mathsf{Comp}}(I,\ p.\mathsf{d}_i,\ G_i) \quad \wedge \quad C_{\mathsf{Disj}}(I,\ p,\ (G_1;\ldots;G_n)) \\
C_{\mathsf{Goal}}(I,\ p,\ (G_c \to G_t;G_e)) & = C_{\mathsf{Comp}}(I,\ p.\mathsf{c},\ G_c) \ \wedge\ C_{\mathsf{Comp}}(I,\ p.\mathsf{t},\ G_t) \ \wedge\ C_{\mathsf{Comp}}(I,\ p.\mathsf{e},\ G_e) \ \wedge\ C_{\mathsf{Ite}}(I,\ p,\ (G_c \to G_t;G_e)) \\
C_{\mathsf{Conj}}(I,\ p,\ (G_1,\ldots,G_n)) & = \bigwedge_{V\in\rho_I(\nu(\{G_1,\ldots G_n\}))} \Big(\big(V_p \leftrightarrow V_{p.\mathsf{c}_j} \vee \cdots \vee V_{p.\mathsf{c}_n}\big) \quad \wedge \quad \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{i-1} \neg \big(V_{p.\mathsf{c}_i} \wedge V_{p.\mathsf{c}_j}\big)\Big) \\
C_{\mathsf{Disj}}(I,\ p,\ (G_1;\ldots;G_n)) & = \bigwedge_{V\in\rho_I(\nu(\{G_1,\ldots G_n\}))} \bigwedge_{i=1}^{n} (V_p \leftrightarrow V_{p.\mathsf{d}_1}) \\
C_{\mathsf{Ite}}(I,\ p,\ (G_c \to G_t;G_e)) & = \bigwedge_{V\in\rho_I(\nu(G))} \big((V_p \leftrightarrow V_{p.\mathsf{c}} \vee V_{p.\mathsf{t}} \vee V_{p.\mathsf{e}}) \ \wedge\ \neg(V_{p.\mathsf{c}} \wedge V_{p.\mathsf{t}})\big) \ \wedge\ \bigwedge_{V\in\rho_I(\eta(G))} \big(\neg V_{p.\mathsf{c}} \ \wedge\ (V_{p.\mathsf{t}} \leftrightarrow V_{p.\mathsf{e}})\big)
\end{array}
$$

Figure 1: Constraints for conjunctions, disjunctions and if-then-elses.

$C_{\mathsf{Init}}(I,i,H)$ for the initial instantiation state i, and the constraint $C_{\mathsf{Fin}}(I,f,H)$ for the final instantiation state f:

$$C_{\mathsf{Arg}}(I, i \gg f, H) = C_{\mathsf{Init}}(I,i,H) \ \wedge\ C_{\mathsf{Fin}}(I,f,H)$$

The constraint $C_{\mathsf{Init}}(I,i,H)$ for an initial instantiation state i of a head variable H is given below:

$$
\begin{array}{rcl}
C_{\mathsf{Init}}(I,\mathsf{free},H) & = & \neg H_{\mathsf{in}} \\
C_{\mathsf{Init}}(I,\mathsf{ground},H) & = & \bigwedge_{W\in\rho_I(H)} W_{\mathsf{in}} \\
C_{\mathsf{Init}}(I,i,H) & = & H_{\mathsf{in}} \quad \wedge \\
\multicolumn{3}{l}{\bigwedge_{\substack{i\to f(i_1,\ldots,i_n)\in rules(i)\\ H\to f(Y_1,\ldots,Y_n)\in I}} \big(C_{\mathsf{Init}}(I,i_1,Y_1) \wedge \cdots \wedge C_{\mathsf{Init}}(I,i_n,Y_n)\big)}
\end{array}
$$

The constraint $C_{\mathsf{Fin}}(I,i,H)$ for a final instantiation state i of a head variable H is given below:

$$
\begin{array}{rcl}
C_{\mathsf{Fin}}(I,\mathsf{free},H) & = & \neg H_{\mathsf{out}} \\
C_{\mathsf{Fin}}(I,\mathsf{ground},H) & = & \bigwedge_{W\in\rho_I(H)} W_{\mathsf{out}} \\
C_{\mathsf{Fin}}(I,i,H) & = & H_{\mathsf{out}} \quad \wedge \\
\multicolumn{3}{l}{\bigwedge_{\substack{i\to f(i_1,\ldots,i_n)\in rules(i)\\ H\to f(Y_1,\ldots,Y_n)\in I}} \big(C_{\mathsf{Fin}}(I,i_1,Y_1) \wedge \cdots \wedge C_{\mathsf{Fin}}(I,i_n,Y_n)\big)}
\end{array}
$$

Mode checking is simply determining if the declared modes are at least as strong as the inferred modes. For each declared mode d of predicate p/n we check whether the implication

$$C_{\mathsf{Decl}}(I,d) \to C_{\mathsf{Inf}}(I,p/n)$$

holds or not. If it doesn't, the declared mode is incorrect.

If we are given declared modes $\mathcal{D}$ for a predicate p/n, they can be used to shortcircuit the calculation of SCCs, since we can use $C_{\mathsf{Decls}}(I,\mathcal{D})$ in mode inference for predicates q/m that call p/n.

Example 11. Given the mode definition:

```
:- mode lsg == (listskel(free) >> ground).
```

the mode declaration $d_1 =$ `append(lsg,lsg,in)` for append gives $C_{\mathsf{Decl}}(I,d_1)$: (ignoring V_{out} variables)

$$
\begin{array}{l}
A_{\mathsf{in}} \wedge \neg AH_{\mathsf{in}} \wedge AT_{\mathsf{in}} \wedge \neg AE_{\mathsf{in}} \wedge B_{\mathsf{in}} \wedge \neg BE_{\mathsf{in}} \ \wedge \\
C_{\mathsf{in}} \wedge CH_{\mathsf{in}} \wedge CT_{\mathsf{in}} \wedge CE_{\mathsf{in}} \ \wedge \\
\neg A_{\lambda} \wedge AH_{\lambda} \wedge \neg AT_{\lambda} \wedge AE_{\lambda} \wedge \neg B_{\lambda} \wedge BE_{\lambda} \ \wedge \\
\neg C_{\lambda} \wedge \neg CH_{\lambda} \wedge \neg CT_{\lambda} \wedge \neg CE_{\lambda}
\end{array}
$$

We can show that $C_{\mathsf{Decl}}(I,d_1) \to C_{\mathsf{Inf}}(I,$ `append/3`$)$.

6. SELECTING PROCEDURES AND EXECUTION ORDER

Once we have generated the constraints for an SCC, we can solve those constraints. If the constraints have no solution, then some position has consumers but no producer, so we report a mode error. If the constraints have some solutions, then each solution gives a mode for each predicate in the SCC; the set of solutions thus defines the set of modes of each predicate. We then need to find a feasible execution order for each mode of each predicate in the SCC. The algorithm for finding feasible execution orders takes a solution as its input. If a given mode of a predicate corresponds to several solutions, it is sufficient for *one* of them to have a feasible ordering.

The main problem in finding a feasible schedule is that the mode analyser and the code generator have distinct views of what it means to produce a (position in a) variable. In the grammar we generate for `append`, for example, the nonterminal AH represents both the value of the variable `AH` and the value of the first element of the variable `A`. In the forward mode of `append`, AH_{in} is true, so the mode analyser considers `AH` to have been produced by the caller even *before* execution enters `append`. However, as far as the code generator is concerned, the producer of `AH` is the unification `A = [AH|AT]`. It is to cater for these divergent views that we separate the notion of a variable being *produced* from the notion of a variable being *visible*.

Definition 10. Given an expanded grammar I, an assignment M of boolean values to the constraint variables of a predicate p/n that makes the constraint $C_{\mathsf{Inf}}(I,p/n)$ true is a *model* of $C_{\mathsf{Inf}}(I,p/n)$. We write $M \models C_{\mathsf{Inf}}(I,p/n)$.

Definition 11. For a given model $M \models C_{\mathsf{Inf}}(I,p/n)$ we define the set of nodes *produced* at a goal path p by

$$\mathsf{produced}_M(I,p) = \{V \mid M(V_p) = 1\}$$

Definition 12. For a given model $M \models C_{\mathsf{Inf}}(I,p/n)$ we define the set of nodes *consumed* by the goal G at a goal path p by the formula shown in Figure 2.

For a unification of the form $X = Y$, we say that a node on one side of the equation is consumed iff a corresponding node on the other side is produced. (Due to the symmetric nature of this relationship, if $V1$ and $V2$ both correspond to W, then $V1$ is consumed iff $V2$ is consumed, and $V1$ is produced iff $V2$ is produced.) It is also possible for a pair of corresponding nodes to be neither produced nor consumed by the unification. This can mean one of two things. If the subterms of X and Y at that node are already bound, then the unification will test the equality of those subterms; if they are still free, then it will create an alias between them. Note that if the unification produces either of the top level nodes X or Y, then we call it an *assignment* unification.

For a unification of the form $X = f(Y_1,\ldots,Y_n)$ we say that the node X is consumed iff it is not produced, and that no other nodes are ever consumed. The reason for the latter half of that rule is that our grammar will use the same nonterminal for e.g. Y_1 as for the first subterm of X. Since this unification merely creates aliases between the Y_i and the corresponding subterms of X, the nonterminals

$$
\begin{array}{rcl}
\mathsf{consumed}_M(I,p,X=Y) & = & \big\{V \;\big|\; \langle V,W\rangle \in \chi_I(X,Y) \wedge W \in \mathsf{produced}_M(I,p)\big\} \\
& & \cup \big\{W \;\big|\; \langle V,W\rangle \in \chi_I(X,Y) \wedge V \in \mathsf{produced}_M(I,p)\big\} \\
\mathsf{consumed}_M(I,p,X=f(Y_1,\ldots,Y_n)) & = & \{X\} \setminus \mathsf{produced}_M(I,p) \\
\mathsf{consumed}_M(I,p,q(Y_1,\ldots,Y_n)) & = & \big\{V \;\big|\; \langle V,W\rangle \in \chi \;\wedge\; M'(W_{\mathsf{in}}) = 1\big\} \\
& & \quad \text{where } \;\chi = \chi_I(\langle Y_1,\ldots,Y_n\rangle, \alpha(q/n)) \text{ and } M' \models C_{\mathsf{Inf}}(I,q/n) \\
& & \qquad \text{such that } \forall\langle V,W\rangle \in \chi \,.\, M(V_d) \leftrightarrow M'(W_\lambda) \\
\mathsf{consumed}_M(I,p,(G_1,\ldots,G_n)) & = & \bigcup_{i=1}^n \mathsf{consumed}_M(I,p.\mathsf{c}_i,G_i) \setminus \mathsf{produced}_M(I,p) \\
\mathsf{consumed}_M(I,p,(G_1;\ldots;G_n)) & = & \bigcup_{i=1}^n \mathsf{consumed}_M(I,p.\mathsf{d}_i,G_i) \setminus \mathsf{produced}_M(I,p) \\
\mathsf{consumed}_M(I,p,(G_c \to G_t;G_e)) & = & \big(\mathsf{consumed}_M(I,p.\mathsf{c},G_c) \cup \mathsf{consumed}_M(I,p.\mathsf{t},G_t) \\
& & \quad \cup\, \mathsf{consumed}_M(I,p.\mathsf{e},G_e)\big) \setminus \mathsf{produced}_M(I,p)
\end{array}
$$

Figure 2: Calculating which nodes are "consumed" at which positions.

of the Y_i cannot be produced by this unification; if they are produced at all, they have to be produced elsewhere. Note that if the unification produces X, then we call it a *construction* unification; if it consumes X, then we call it a *deconstruction* unification.

For a call to a predicate q, we know which nodes of the actual parameters of the call the model M of the predicate we are analyzing says should be produced by the call. We need to find a model M' of the constraints of q that causes the corresponding nodes in the actual parameters of q to be output. Since the first stage of the analysis succeeded we know such a model M' exists. The consumed nodes of the call are then the nodes of the actual parameters that correspond to the nodes of the formal parameters of q that M' requires to be input.

For compound goals, the consumed nodes are the union of the consumed nodes of the subgoals, minus the nodes that are produced within the compound goal.

Example 12. In the `(in, in, out)` mode of `append`, the produced and consumed sets of the conjuncts are:

Path	produced	consumed
$\mathsf{d}_1.\mathsf{c}_1$	$\{\}$	$\{A\}$
$\mathsf{d}_1.\mathsf{c}_2$	$\{C, CH, CT, CE\}$	$\{B, BE\}$
$\mathsf{d}_2.\mathsf{c}_1$	$\{\}$	$\{A\}$
$\mathsf{d}_2.\mathsf{c}_2$	$\{C\}$	$\{\}$
$\mathsf{d}_2.\mathsf{c}_3$	$\{CH\}$	$\{AH\}$
$\mathsf{d}_2.\mathsf{c}_4$	$\{CT, CE\}$	$\{AT, AE, B, BE\}$

Neither disjunct produces any position that it also consumes. Therefore, if our ordering algorithm only required a node to be produced before it is consumed, it would find any order acceptable. On the other hand, the code generator is more fussy; for example, before it can emit code for the recursive call, it needs to know where variables `AH` and `AT` are stored, even if they have not been bound yet. This is why we need the concept of *visibility*.

Definition 13. A variable is *visible* at a goal path p if the variable is a head variable or has appeared in the predicate body somewhere to the left of p. The functions make_visible and need_visible defined in Figure 3 respectively determine whether a goal makes a variable visible or requires it to be visible.

Example 13. Given the `(in, in, out)` mode of `append` the make_visible and need_visible sets of the conjuncts are:

Path	make_visible	need_visible
$\mathsf{d}_1.\mathsf{c}_1$	$\{\}$	$\{A\}$
$\mathsf{d}_1.\mathsf{c}_2$	$\{C\}$	$\{B\}$
$\mathsf{d}_2.\mathsf{c}_1$	$\{AH, AT\}$	$\{A\}$
$\mathsf{d}_2.\mathsf{c}_2$	$\{C, CH, CT\}$	$\{\}$
$\mathsf{d}_2.\mathsf{c}_3$	$\{CH\}$	$\{AH\}$
$\mathsf{d}_2.\mathsf{c}_4$	$\{CT\}$	$\{AT, B\}$

Our algorithm needs to find, in each conjunction in the body, an ordering of the conjuncts such that the producer of each node comes before any of its consumers, and each variable is made visible before any point where it needs to be visible. We do this by traversing the predicate body top down. At each conjunction, we construct a directed graph whose nodes are the conjuncts. The initial graph has an edge from c_i to c_j iff c_i produces a node that c_j consumes. If this graph is cyclic, then mode ordering fails. If it isn't, we try to add more edges while keeping the graph acyclic.

We sort the variables that need to be visible anywhere in the conjunction that are also made visible in the conjunction into two classes: those where it is clear which conjunct should make them visible and those where it isn't. A variable falls into the first class iff it is made visible in only one conjunct, or if a conjunct that makes it visible is also the producer of its top level node. (In the forward mode of `append`, all variables fall into the first class; the only variable that is made visible in more than one conjunct, `CH`, does not need to be visible in any conjunct in that conjunction.) For each of these variables, we add an edge from the conjunct that makes the variable visible to all the conjuncts c_j need it to be visible. If the graph is still acyclic, we then start searching the space of mappings that map each variable in the second class to a conjunct that makes that variable visible, looking for a map that results in an acyclic graph when we add links from the selected make_visible conjunct of each variable to all the corresponding need_visible conjuncts.

It can also happen that some of the conjuncts need a variable visible that none of the goals in the conjunction make visible. If this variable is made visible by a goal to the left of the whole conjunction, by another conjunction that encloses this one, then everything is fine. If it isn't, then the ordering of the enclosing conjunction would have already failed, because if no conjunct makes the variable visible, then the conjunction as a whole needs it visible.

If no mapping yields an acyclic graph, the procedure has a mode error. If some mappings do, then the algorithm in general has two choices to make: it can pick any acyclic graph, and it can pick any order for the conjuncts that is consistent with that graph.

$$
\mathsf{make\_visible}_M(I,p,G) = \begin{cases} \nu(G) \setminus \mathsf{consumed}_M(I,p,G) & \text{if } G \text{ is atomic;} \\ \bigcup_{i=1}^{n} \mathsf{make\_visible}_M(I, p.\mathsf{c}_i, G_i) & \text{if } G = (G_1, \ldots, G_n); \\ \bigcap_{i=1}^{n} \mathsf{make\_visible}_M(I, p.\mathsf{d}_i, G_i) & \text{if } G = (G_1; \ldots; G_n); \\ (\mathsf{make\_visible}_M(I, p.\mathsf{c}, G_c) \cup \mathsf{make\_visible}_M(I, p.\mathsf{t}, G_t)) & \text{if } G = (G_c \rightarrow G_t; G_e). \\ \qquad\qquad \cap\ \mathsf{make\_visible}_M(I, p.\mathsf{e}, G_e) & \end{cases}
$$

$$
\mathsf{need\_visible}_M(I,p,G) = \begin{cases} \nu(G) \cap \mathsf{consumed}_M(I,p,G) & \text{if } G \text{ is atomic;} \\ \bigcup_{i=1}^{n} \mathsf{need\_visible}_M(I, p.\mathsf{c}_i, G_i) \setminus \mathsf{make\_visible}_M(I,p,G) & \text{if } G = (G_1, \ldots, G_n); \\ \bigcup_{i=1}^{n} \mathsf{need\_visible}_M(I, p.\mathsf{d}_i, G_i) & \text{if } G = (G_1; \ldots; G_n); \\ \bigcup_{pc \in \{c,t,e\}} \mathsf{need\_visible}_M(I, p.pc, G_{pc}) \setminus \mathsf{make\_visible}_M(I,p,G) & \text{if } G = (G_c \rightarrow G_t; G_e). \end{cases}
$$

Figure 3: Calculating make_visible and need_visible.

All the nodes the forward mode of `append` consumes are input to the predicate, so there are no ordering constraints between producers and consumers. The first disjunct has no visibility constraints either, so it can be given any order. In the second disjunct, visibility requirements dictate that `A = [AH|AT]` must occur before both `AH = CH` and `append`(AT, B, CT), to make AH and AT visible where required. This leaves the compiler with this graph:

$$
\begin{tikzcd}
\texttt{A=[AH|AT]} \arrow[rr, bend left] \arrow[dr] & \texttt{C=[CH|CT]} & \texttt{AH=CH} \\
& \texttt{append}(AT, B, CT) &
\end{tikzcd}
$$

This graph does not completely fix the order of the conjuncts. A parallel implementation may choose to execute several conjuncts in parallel, although it this case that would not be worth while. More likely, an implementation may choose to schedule the recursive call last to ensure tail recursion. (With the old mode analyser, we needed a program transformation separate from mode analysis [15] to introduce tail recursion in predicates like this.)

7. EXPERIMENTAL EVALUATION

Our analysis is implemented within the Melbourne Mercury compiler. We represent the Boolean constraints as reduced ordered binary decision diagrams (ROBDDs) [2] using a highly-optimised implementation by Schachte [16] who has shown that ROBDDs provide a very efficient representation for other logic program analyses based on Boolean domains.

ROBDDs are directed acyclic graphs with common-subexpressions merged. They provide an efficient, canonical representation for Boolean functions.

In the worst case, the size of an ROBDD can be exponential in the number of variables. In practice, however, with a bit of care this worst-case behaviour can usually be avoided. We use a number of techniques to keep the ROBDDs as small and efficient as possible.

We now present some preliminary results to show the feasibility of our analysis. The timings are all taken from tests run on a Gateway Select 950 PC with a 950MHz AMD Athlon CPU, 256KB of L2 cache and 256MB of memory, running Linux kernel 2.4.16.

Table 7 compares times for mode checking some simple benchmarks. The column labelled "simple" is the time for the simple constraint-based system for ground variables presented in Section 4. The column labelled "full" is for the full constraint-based system presented in Section 5. The column labelled "old" is the time for mode checking in the current Mercury mode checker. The final two columns show the ratios between the new and old systems. All times are in milliseconds and are averaged over 10 runs.

	simple	full	old	simple/old	full/old
cqueens	407	405	17	23	23
crypt	1032	1335	38	27	35
deriv	13166	32541	59	223	551
nrev	520	569	42	12	13
poly	1348	5245	63	21	83
primes	356	358	12	29	29
qsort	847	1084	112	7	9
queens	386	381	9	42	42
query	270	282	11	24	25
tak	204	201	2	102	100

Table 1: Mode checking: ground.

The constraint-based analyses are significantly slower than the current system. This is partly because they are obtaining much more information about the program and thus doing a lot more work. For example, the current system selects a fixed sequential order for conjunctions during the mode analysis — an order that disallows partially instantiated data structures — whereas the constraint-based approaches allow all possible orderings to be considered while building up the constraints. The most appropriate scheduling can then be selected based on the execution model considering, for example, argument passing conventions (e.g. for the possibility of introducing tail calls) and whether the execution is sequential or parallel.

Profiling shows that much of the execution time is spent in building and manipulating the ROBDDs. It may be worth investigating different constraints solvers, such as propagation-based solvers. Another possible method for improving overall analysis time would be to run the old mode analysis first and only use the new analysis for predicates for which the old analysis fails.

It is interesting to observe the differences between the simple constraint system and the full system. None of these benchmarks require partially instantiated data structures so they are all able to be analysed by the simple system. In some cases, the simple system is not very different to the full system, but in others—particularly in the bigger benchmarks—it is significantly faster. We speculate that

	check	infer	infer/check
iota	384	472	1.22
append	327	488	1.49
copytree	150	6174	41.16

Table 2: Mode checking: partially instantiated.

this is because the bigger benchmarks benefit more from the reduced number of constraint variables in the simple analysis.

Table 7 shows some timings for programs that make use of partially instantiated modes, which the current Mercury system (and the simple constraint-based system) is unable to analyse. Again the times are in milliseconds averaged over 10 runs.

The "iota" benchmark is the program from Example 4. The "append" benchmark is the classic `append/3` (however, the checking version has all valid combinations of `in`, `out` and `lsg` modes declared). The "copytree" benchmark is a small program that makes a structural copy of a binary tree skeleton, with all elements in the copy being new free variables.

The times in the "check" columns are for checking programs with mode declarations whereas the "infer" column shows times for doing mode inference with all mode declarations removed. It is interesting to note the saving in analysis time achieved by adding mode declarations. This is particularly notable for the "copytree" benchmark where mode inference is able to infer many more modes than the one we have declared. (We can similarly declare only the `(in, in, out)` mode of `append` and reduce the analysis time for that to 210ms.)

8. CONCLUSION

We have defined a constraint based approach to mode analysis of Mercury. While it is not as efficient as the current system for mode checking, it is able to check and infer more complex modes than the current system, and decouples reordering of conjuncts from determining producers. Although not described here the implementation handles all Mercury constructs such as higher-order.

The constraint-based mode analysis does not yet handle subtyping or unique modes. We plan to extend it to handle these features as well as explore more advanced mode systems: complicated uniqueness modes, where unique objects are stored in and recovered from data structures; polymorphic modes, where Boolean variables represent a pattern of mode usage; and the circular modes needed by client-server programs, where client and server processes (modelled as recursive loops) cooperate to instantiate different parts of a data structure in a coroutining manner.

We would like to thank the Australian Research Council for their support.

9. REFERENCES

[1] J. Boye and J. Małuszyński. Directional types and the annotation method. *JLP*, 33(3):179–220, 1997.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.

[4] K. Cho and K. Ueda. Diagnosing non-well-moded concurrent logic programs. In *Procs. of ICLP96*, 215–229. MIT Press, 1996.

[5] C. Codognet, P. Codognet, and M. Corsini. Abstract interpretation of concurrent logic languages. In *Procs. of NACLP 1990*, 215–232, MIT Press 1990.

[6] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM TOPLAS*, 11(3):418–450, 1989.

[7] S. Etalle and M. Gabbrielli. Layered Modes. *JLP*, 39:225–244, 1999.

[8] M. García de la Banda, P. Stuckey, W. Harvey, and K. Marriott. Mode checking in HAL. In *Procs. of CL2000*, LNCS 1861, 1270–1284, 2000.

[9] F. Kluzniak. Type synthesis for ground Prolog. In *Procs. of ICLP87*. 788–816, MIT Press, 1987.

[10] G. Janssens and M. Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *JLP*, 13:205–258, 1993.

[11] C. Mellish. The automatic derivation of mode declarations for Prolog programs. Research paper 163, Dept. of AI, University of Edinburgh, 1981.

[12] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *Procs. of ICLP95*. 781–796, MIT Press, 1995.

[13] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *AI*, 23:295–307, 1984.

[14] O. Ridoux, P. Boizumault, and F. Malesieux. Typed static analysis: Application to groundness analysis of Prolog and lambda-Prolog. In *Procs. of FLOPS99*, LNCS 1722, 267–283, 1999.

[15] P. Ross, D. Overton, and Z. Somogyi. Making Mercury programs tail recursive. In *Procs. of LOPSTR99*, LNCS 1817, 196–215, 1999.

[16] P. Schachte. Efficient ROBDD operations for program analysis. In *Procs. of the Nineteenth Australasian Computer Science Conference*, 347–356. Australian Computer Science Communications, 1996.

[17] J.-G. Smaus, P. Hill, and A. King. Mode analysis domains for typed logic programs. In *Procs. of LOPSTR99*, LNCS 1817, 82–101, 2000.

[18] Z. Somogyi. A system of precise modes for logic programs. In *Procs. of ICLP87*, 769–787, MIT Press, 1987.

[19] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *JLP*, 26(1-3):17–64, 1996.

[20] W. Vanhoof. Binding-time analysis by constraint solving: A modular and higher-order approach for Mercury. In *Procs. of LPAR2000*, LNAI 1955, 399–416. Springer-Verlag, 2000.