# Constraint-Based Mode Analysis of Mercury

**David Overton**[*, †] **Zoltan Somogyi**[*]

`dmo@cs.mu.oz.au` `zs@cs.mu.oz.au`

**Peter J. Stuckey**[*]

`pjs@cs.mu.oz.au`

[*] **University of Melbourne**
[†] **Monash University**

# 1 Motivation

Mercury is a strongly-moded logic programming language. This means that the compiler must have precise information about the state of instantiation of each variable at each program point.

The compiler must decide which parts of a procedure produce which variables, and how conjuncts in a predicate body should be re-ordered to ensure that producers come before consumers.

The task of a mode system is to determine this information, either by mode inference or by checking declarations supplied by the programmer.

## 1.1 The Current Mercury Mode System

The current Mercury mode system has several limitations:

- it is not accurate enough to allow partially instantiated data structures to be useful;
- it is not accurate enough to fully support destructive update;
- it doesn’t reorder conjuncts while doing mode inference, in order to avoid combinatorial explosion; and
- the algorithm is quite complicated since it attempts to do several conceptually different things in one pass.

## 1.1 The Current Mercury Mode System

The current Mercury mode system has several limitations:

- it is not accurate enough to allow partially instantiated data structures to be useful;
- it is not accurate enough to fully support destructive update;
- it doesn't reorder conjuncts while doing mode inference, in order to avoid combinatorial explosion; and
- the algorithm is quite complicated since it attempts to do several conceptually different things in one pass.

## 1.1 The Current Mercury Mode System

The current Mercury mode system has several limitations:

- it is not accurate enough to allow partially instantiated data structures to be useful;
- it is not accurate enough to fully support destructive update;
- it doesn't reorder conjuncts while doing mode inference, in order to avoid combinatorial explosion; and
- the algorithm is quite complicated since it attempts to do several conceptually different things in one pass.

## 1.1 The Current Mercury Mode System

The current Mercury mode system has several limitations:

- it is not accurate enough to allow partially instantiated data structures to be useful;
- it is not accurate enough to fully support destructive update;
- it doesn’t reorder conjuncts while doing mode inference, in order to avoid combinatorial explosion; and
- the algorithm is quite complicated since it attempts to do several conceptually different things in one pass.

## 1.1 The Current Mercury Mode System

The current Mercury mode system has several limitations:

- it is not accurate enough to allow partially instantiated data structures to be useful;
- it is not accurate enough to fully support destructive update;
- it doesn't reorder conjuncts while doing mode inference, in order to avoid combinatorial explosion; and
- the algorithm is quite complicated since it attempts to do several conceptually different things in one pass.

```
:- pred length(list(int), int).
:- mode length(out(list_skel(free)), in) is det.
length(L, N) :-
    (   L = [], N = 0
    ;   L = [_ | T], M = N - 1, length(T, 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)
    ).

?- length(L, 10), iota(L, 3).
```

## 1.2 A Constraint-Based Approach

We have developed an alternative two-step algorithm which attempts to solve these problems.

1. Determine producers for each node of the type tree of each variable. In each mode:
   - each node should have at most one producer; and
   - each node that has consumers should have exactly one producer.
2. Find an execution order that ensures that producers are executed before consumers if the implementation doesn't support coroutining.

# 2 Deterministic Regular Tree Grammars

## 2.1 Types

We must be able to talk about each of the individual parts of the terms which a program variable will be able to take as values.

We do this with a regular tree, expressed as a tree grammar.

E.g. from type declarations

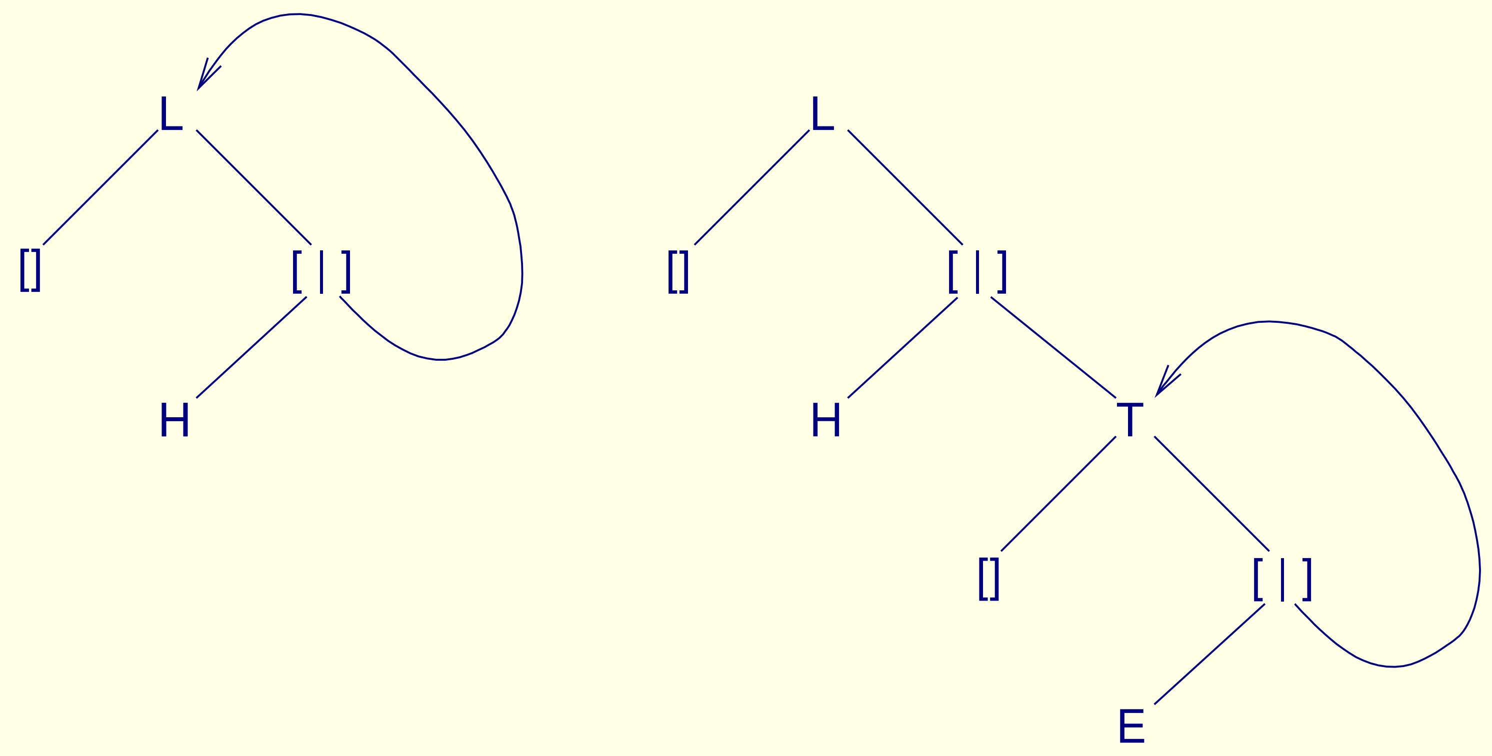
```
:- type list(T) ---> [] ; [T | list(T)].
:- type abc ---> a ; b ; c.
```

and type `list(abc)` we get the grammar

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

## 2.2 Expanded Grammars

Need to expand the grammar (by “unrolling” the type tree) to be able to differentiate, where necessary, between different nodes that share the same non-terminal in the original grammar. It may also be necessary to introduce new unifications and rename variables.



E.g. `append/3` is transformed as follows

```
append(A, B, C) :-                      append(A, B, C) :-
        A = [],                                 A = [],
        B = C.                                  B = C.
append(A, B, C) :-          ==>         append(A, B, C) :-
        A = [H | AT],                           A = [AH | AT],
        C = [H | CT],                           C = [CH | CT],
        append(AT, B, CT).                      AH = CH,
                                                append(AT, B, CT).
```

and the original and the expanded grammars are

$$
\begin{array}{rcl}
A & \longrightarrow & [\,] \; ; \; [H|AT] \\
B & \longrightarrow & [\,] \; ; \; [BE|B] \\
C & \longrightarrow & [\,] \; ; \; [H|CT]
\end{array}
\qquad
\begin{array}{rcl}
A & \longrightarrow & [\,] \; ; \; [AH|AT] \\
AT & \longrightarrow & [\,] \; ; \; [AE|AT] \\
B & \longrightarrow & [\,] \; ; \; [BE|B] \\
C & \longrightarrow & [\,] \; ; \; [CH|CT] \\
CT & \longrightarrow & [\,] \; ; \; [CE|CT]
\end{array}
$$

## 2.3 Reachable and Corresponding Nodes

Reachable:

$\rho_I(X) =$ the set of nodes reachable from $X$ in the grammar $I$

Corresponding:

$\chi_I(X, Y) =$ the set of pairs of corresponding nodes reachable from $X$ and $Y$ in the grammar $I$

E.g. if $I$ is the expanded grammar for `append/3`:

$$\rho_I(\mathsf{C}) = \{\mathsf{C}, \mathsf{CH}, \mathsf{CT}, \mathsf{CE}\}$$

and

$$\chi_I(\mathsf{B}, \mathsf{C}) = \{\langle \mathsf{B}, \mathsf{C}\rangle, \langle \mathsf{BE}, \mathsf{CH}\rangle, \langle \mathsf{B}, \mathsf{CT}\rangle, \langle \mathsf{BE}, \mathsf{CE}\rangle\}$$

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathsf{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathsf{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathsf{c}$ denotes its condition, $p.\mathtt{t}$ denotes its then-part, and $p.\mathsf{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathsf{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathsf{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathsf{c}$ denotes its condition, $p.\mathsf{t}$ denotes its then-part, and $p.\mathsf{e}$ denotes its else-part.

## 3 Goal Paths

In order to describe where a part of a variable becomes bound, we need to be able to uniquely identify each subgoal of a predicate body.

- A goal path consists of a sequence of path components.
- The empty goal path $\lambda$ denotes the entire procedure body.
- If the goal at path $p$ is a conjunction, then $p.\mathrm{c}_n$ denotes its $n$th conjunct.
- If the goal at path $p$ is a disjunction, then $p.\mathrm{d}_n$ denotes its $n$th disjunct.
- If the goal at path $p$ is an if-then-else, then $p.\mathrm{c}$ denotes its condition, $p.\mathrm{t}$ denotes its then-part, and $p.\mathrm{e}$ denotes its else-part.

## 4 Constraint Domain

We associate a set of constraint variables with each non-terminal $V$ in the expanded grammar for the predicate.

- $V_{\mathsf{in}}$ is the proposition that $V$ is produced outside the predicate.
- $V_{\mathsf{out}}$ is the proposition that $V$ is produced somewhere (either inside or outside the predicate).
- $V_p$ is the proposition that $V$ is produced by the goal at goal path $p$.

## 4 Constraint Domain

We associate a set of constraint variables with each non-terminal $V$ in the expanded grammar for the predicate.

- $V_{\text{in}}$ is the proposition that $V$ is produced outside the predicate.
- $V_{\text{out}}$ is the proposition that $V$ is produced somewhere (either inside or outside the predicate).
- $V_p$ is the proposition that $V$ is produced by the goal at goal path $p$.

# 4 Constraint Domain

We associate a set of constraint variables with each non-terminal $V$ in the expanded grammar for the predicate.

- $V_{\mathsf{in}}$ is the proposition that $V$ is produced outside the predicate.
- $V_{\mathsf{out}}$ is the proposition that $V$ is produced somewhere (either inside or outside the predicate).
- $V_p$ is the proposition that $V$ is produced by the goal at goal path $p$.

## 4 Constraint Domain

We associate a set of constraint variables with each non-terminal $V$ in the expanded grammar for the predicate.

- $V_{\mathsf{in}}$ is the proposition that $V$ is produced outside the predicate.
- $V_{\mathsf{out}}$ is the proposition that $V$ is produced somewhere (either inside or outside the predicate).
- $V_p$ is the proposition that $V$ is produced by the goal at goal path $p$.

# 5 Mode Inference Constraints

## 5.1 Structural Constraints

For all nodes $V$ in the grammar:

$$(V_{\mathsf{out}} \leftrightarrow V_{\mathsf{in}} \vee V_{\lambda}) \ \wedge \ \neg(V_{\mathsf{in}} \wedge V_{\lambda})$$

For all nodes $V$ in the grammar which are not reachable from the head variables:

$$\neg V_{\mathsf{in}}$$

For all nodes $D$, $V$ such that $D \in \rho_I(V)$

$$(D_{\mathsf{in}} \rightarrow V_{\mathsf{in}}) \ \wedge \ (D_{\mathsf{out}} \rightarrow V_{\mathsf{out}})$$

## 5.2 Goal Constraints

For each goal path $p$:

For each node reachable from a variable local to $p$:

$$V_p \leftrightarrow V_{\text{out}}$$

For each node reachable from a variable that is non-local to the parent of $p$ but does not occur in $p$:

$$\neg V_p$$

## 5.3 Compound Goals

If the goal at $p$ is a conjunction $(G_1, \ldots, G_n)$, for all nodes reachable from a variable in the goal:

$$(V_p \leftrightarrow V_{p.\mathsf{c}_1} \vee \ldots \vee V_{p.\mathsf{c}_n}) \;\wedge\; \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{i-1} \neg(V_{p.\mathsf{c}_i} \wedge V_{p.\mathsf{c}_j})$$

If the goal at $p$ is a disjunction $(G_1; \ldots; G_n)$, for all nodes reachable from a variable in the goal:

$$V_p \leftrightarrow V_{p.\mathsf{d}_i}$$

An if-then-else $(G_c \rightarrow G_t; G_e)$ is semantically equivalent to $(G_c, G_t); (\neg\exists G_c, G_e)$.

If the goal at $p$ is an if-then-else $(G_c \rightarrow G_t; G_e)$, for all nodes reachable from a variable in the goal:

$$(V_p \leftrightarrow V_{p.\mathsf{c}} \vee V_{p.\mathsf{t}} \vee V_{p.\mathsf{e}}) \;\wedge\; \neg(V_{p.\mathsf{c}} \wedge V_{p.\mathsf{t}})$$

For nodes reachable from variables non-local to the if-then-else we also need:

$$\neg V_{p.\mathsf{c}} \;\wedge\; (V_{p.\mathsf{t}} \leftrightarrow V_{p.\mathsf{e}})$$

## 5.4 Atomic Goals

For a unification $X = Y$ for each pair $\langle V, W \rangle \in \chi_I(X, Y)$

$$V_{\mathsf{out}} \wedge W_{\mathsf{out}} \wedge \neg(V_p \wedge W_p)$$

For a unification $X = f(Y_1, \ldots, Y_n)$:

$$X_{\mathsf{out}}$$

and for each node $V$ reachable from $Y_i$:

$$\neg V_p$$

For a call $q(Y_1, \ldots, Y_n)$ where $\langle X_1, \ldots, X_n \rangle$ are the formal parameters of $q/n$.

Recursive call.

$$\bigwedge_{\langle V,W\rangle \in \chi_I(\langle X_1,\ldots,X_n\rangle,\langle Y_1,\ldots,Y_n\rangle)} (V_\lambda \leftrightarrow W_p) \ \wedge \ (V_{\mathsf{in}} \rightarrow W_{\mathsf{out}})$$

(assumes that the recursive call is in the same mode).

Non-recursive call.

$$\exists \rho_I(\{X_1, \ldots, X_n\}).\ C_{\mathsf{Inf}}(I, q/n)$$

$$\wedge \quad \bigwedge_{\langle V,W\rangle \in \chi_I(\langle X_1,\ldots,X_n\rangle,\langle Y_1,\ldots,Y_n\rangle)} (V_\lambda \leftrightarrow W_p) \ \wedge \ (V_{\mathsf{in}} \rightarrow W_{\mathsf{out}})$$

where $C_{\mathsf{Inf}}(I, q/n)$ is the constraint inferred for $q/n$.

## 5.5 Example

Each of the following rows gives one solution of the constraints we build for `append/3`, projected onto $\{V_{\mathsf{in}} | V \in \rho_I(\{\mathsf{A}, \mathsf{B}, \mathsf{C}\})\}$:

$$\begin{array}{cccccccccc}
\mathsf{A}_{\mathsf{in}} & \mathsf{AH}_{\mathsf{in}} & \mathsf{AT}_{\mathsf{in}} & \mathsf{AE}_{\mathsf{in}} & \mathsf{B}_{\mathsf{in}} & \mathsf{BE}_{\mathsf{in}} & \neg\mathsf{C}_{\mathsf{in}} & \neg\mathsf{CH}_{\mathsf{in}} & \neg\mathsf{CT}_{\mathsf{in}} & \neg\mathsf{CE}_{\mathsf{in}} \\
\neg\mathsf{A}_{\mathsf{in}} & \neg\mathsf{AH}_{\mathsf{in}} & \neg\mathsf{AT}_{\mathsf{in}} & \neg\mathsf{AE}_{\mathsf{in}} & \neg\mathsf{B}_{\mathsf{in}} & \neg\mathsf{BE}_{\mathsf{in}} & \mathsf{C}_{\mathsf{in}} & \mathsf{CH}_{\mathsf{in}} & \mathsf{CT}_{\mathsf{in}} & \mathsf{CE}_{\mathsf{in}} \\
\mathsf{A}_{\mathsf{in}} & \mathsf{AH}_{\mathsf{in}} & \mathsf{AT}_{\mathsf{in}} & \mathsf{AE}_{\mathsf{in}} & \mathsf{B}_{\mathsf{in}} & \mathsf{BE}_{\mathsf{in}} & \mathsf{C}_{\mathsf{in}} & \neg\mathsf{CH}_{\mathsf{in}} & \mathsf{CT}_{\mathsf{in}} & \neg\mathsf{CE}_{\mathsf{in}} \\
\mathsf{A}_{\mathsf{in}} & \neg\mathsf{AH}_{\mathsf{in}} & \mathsf{AT}_{\mathsf{in}} & \neg\mathsf{AE}_{\mathsf{in}} & \mathsf{B}_{\mathsf{in}} & \neg\mathsf{BE}_{\mathsf{in}} & \mathsf{C}_{\mathsf{in}} & \mathsf{CH}_{\mathsf{in}} & \mathsf{CT}_{\mathsf{in}} & \mathsf{CE}_{\mathsf{in}} \\
\mathsf{A}_{\mathsf{in}} & \mathsf{AH}_{\mathsf{in}} & \mathsf{AT}_{\mathsf{in}} & \mathsf{AE}_{\mathsf{in}} & \mathsf{B}_{\mathsf{in}} & \mathsf{BE}_{\mathsf{in}} & \mathsf{C}_{\mathsf{in}} & \mathsf{CH}_{\mathsf{in}} & \mathsf{CT}_{\mathsf{in}} & \mathsf{CE}_{\mathsf{in}}
\end{array}$$

Each solution corresponds to a different mode of `append/3`. The first two solutions are the principal modes. The other solutions are implied modes.

## 6 Selecting Execution Order

- Each solution of the constraints specifies which goals produce which nodes and which goals consume which nodes and thus specifies the modes of the arguments.
- We need to ensure that each node is produced before it is consumed so we build a directed graph and do a topological sort.
- If the graph has a cycle, there is no viable sequential execution order.

## 6 Selecting Execution Order

- Each solution of the constraints specifies which goals produce which nodes and which goals consume which nodes and thus specifies the modes of the arguments.
- We need to ensure that each node is produced before it is consumed so we build a directed graph and do a topological sort.
- If the graph has a cycle, there is no viable sequential execution order.

## 6 Selecting Execution Order

- Each solution of the constraints specifies which goals produce which nodes and which goals consume which nodes and thus specifies the modes of the arguments.
- We need to ensure that each node is produced before it is consumed so we build a directed graph and do a topological sort.
- If the graph has a cycle, there is no viable sequential execution order.

## 6 Selecting Execution Order

- Each solution of the constraints specifies which goals produce which nodes and which goals consume which nodes and thus specifies the modes of the arguments.
- We need to ensure that each node is produced before it is consumed so we build a directed graph and do a topological sort.
- If the graph has a cycle, there is no viable sequential execution order.

## 7 Experimental Evaluation

- Our implementation uses Reduced Ordered Binary Decision Diagrams (ROBDDs).
- Preliminary results suggest that analysis time is considerably slower (typically between 7 and 50 times slower) than the current system.
- For this reason we anticipate only using it for predicates for which the current system doesn't work.
- It may also be worth trying alternative constraint solvers.

## 8 Conclusion and Future Work

- Our new system is not as efficient as the current system, but it is able to check and infer more complex modes, and decouples reordering of conjuncts from determining producers.
- The implementation handles all Mercury constructs, including higher-order.

Future work:

- Support for sub-typing and uniqueness modes, including more complicated uniqueness modes than the current system can handle.
- Polymorphic modes, where Boolean variables represent a pattern of mode usage.
- Circular modes needed for coroutining.