

# Further Inductive Mercury Programming and IMP0.5

Barnaby Fisher and James Cussens

Dept of Computer Science, University of York, York, UK  
{barney|jc}@cs.york.ac.uk

**Abstract.** We explore the use of Mercury for Inductive Logic Programming and present IMP0.5, the product of our research. Mercury is a compiled logic programming language with modern features, which requires the user to write type, mode and determinism declarations for each of their predicates. This information is used by the Mercury compiler to optimise generated code, which, amongst other things, enables Mercury to provide faster execution than Prolog. IMP0.5 is an ILP system, and an ILP software library, which contains re-usable modules of ILP related code. Our aim in creating IMP0.5 as a software library was to significantly reduce the effort required to implement new ILP algorithms. We feel this has been achieved and provide implementations of some typical ILP algorithms, the most notable being the Aleph default algorithm. Since Mercury is a purely declarative language run-time assertion of induced hypotheses is prohibited. Therefore, hypotheses are represented as ground terms, and, to enable fast cover testing, interpreted with a problem specific interpreter, which is generated at compile-time. We also use a cover set representation based on RL-Trees, which is very space efficient and thus beneficial for large searches. Empirical results are generally good, especially for complex background knowledge, and in one case shows a 56 times speed-up compared with Aleph.

## 1 Introduction

The argument for applying ILP to a machine learning problem is at its strongest precisely in those cases where there is a large hypothesis space and where the background knowledge is overwhelmingly intensional. Such situations will inevitably have a high computational overhead so it is no surprise that much work on ILP concerns increasing the efficiency of ILP algorithms. In this paper we present work which addresses this goal by moving from Prolog to an implementation language which allows compilation to native code: the Mercury language.

We have chosen to compare our Mercury implementation (IMP) to Aleph running under YAP Prolog. YAP Prolog is known to be one of the fastest Prolog implementations available and Aleph is a system which has been optimised over many years of development. In previous work in a similar vein [4] a system was presented which, although generally having faster cover testing than Aleph, was slower overall, sometimes much so. In contrast the version of IMP presented here,

IMP0.5, is usually faster than Aleph overall. In a computationally demanding graph-theoretical learning problem IMP is vastly faster than Aleph. As well as presenting such ‘bottom-line’ results we describe the architecture of the IMP system. IMP is not an Aleph clone it is a software library of ILP modules: Aleph is only one algorithm that can be implemented by an appropriate choice of modules.

The paper is organised as follows. Section 2 provides a brief description of the Mercury language. Section 3 presents the IMP system and Section 4 shows how to apply it to a particular ILP problem. Section 5 describes our cover set implementation based on RL-trees which has significantly improved both speed and memory efficiency. Section 6 describes the learning problems we have used to provide a comparison with Aleph and Section 7 presents the results of those comparisons. Section 8 gives our conclusions.

## 2 The Mercury language

We give only a brief description of the Mercury language; a comprehensive account and the Mercury system itself can be found at the Mercury website: <http://www.cs.mu.oz.au/research/mercury/>. From the programmer’s point of view Mercury is similar to Prolog except 1) it is purely declarative and 2) type, mode and determinism information must be supplied for each predicate defined. Fig 1 shows declarations for the familiar list `append` predicate (taken from the source of the Mercury system’s `list` library module). As this example shows, Mercury permits polymorphic types: the use of the variable `T` states that this predicate can append lists of any other type. Consider the goal `append(L1,L2,L3)`. The first mode declaration states that if `L1` and `L2` are given then there is exactly one value for `L3`. The second states that if `L1` and `L3` are inputs then the goal may fail, but if it succeeds it will do so only once. The last mode states that if only `L3` is given the goal will succeed at least once but possibly more times.

```
:- pred list.append(list(T), list(T), list(T)).  
:- mode list.append(in, in, out) is det.  
:- mode list.append(in, out, in) is semidet.  
:- mode list.append(out, out, in) is multi.
```

**Fig. 1.** Mercury declarations for the `append` predicate

Since the Mercury compiler has far more information than a Prolog one and since it does not have to worry about non-logical side-effects substantially more efficient code can be generated. In its standard operation the Mercury compiler generates very low-level C code which can then be compiled to native code. Each mode of a predicate will be translated into a separate C function.

### 3 The IMP system

**Table 1.** Summary of the modules available in the IMP library

Module Name	Short Description
background_knowledge	Enables run-time assertion of hypotheses with an appropriate interpreter. Also, has predicates for the interpretation of compiled background knowledge.
coverset	Contains a space efficient coverset representation based on RL-Trees. See Section 5.
definite_clause	Abstract data type for definite clauses which can be used for hypothesis representation.
entailment_semantics	Cover testing predicates for learning from entailment semantics. Also includes some interpreters for different representations.
example	Abstract data type for examples. Has code to read examples from a file at run-time.
hypothesis_language	Contains data types for the user to declare background literals with types and modes of arguments. Also, has predicates for the manipulation of this knowledge for the rest of the system.
ilp_algorithms	Exports a predicate for each implemented ILP algorithm. See Section 3.2.
optimise_fns	Contains predicates for the typical optimise functions such as accuracy and coverage.
search	Export a predicate for each implemented search strategy. See Section 3.1.
substitution	Abstract data type for a substitution.
successor	Contains useful predicates for the creation of a successor function (or refinement operator).

The design of IMP aims to take full advantage of Mercury's benefits such as support for the creation of modular and re-usable code and fast execution speed of compiled binaries. To achieve re-usability of code we use higher-order predicate terms, abstract types, polymorphic types and modules. We take advantage of the generated fast binaries by having the users' background knowledge as a module of normal Mercury code.

IMP treats examples as data which is read at run-time from a specified file. The alternative would be to have the examples compiled into the system, but we choose run-time because it is much faster reading examples as data rather than compiling as code. Another advantage of reading examples at run-time is that different example sets can be used without recompilation.

As stated before IMP is essentially an ILP software library. By this we mean it is a collection of Mercury modules containing ILP related code. Table 1 gives a

brief description of these modules. The following few sections describe how these modules are related to each other; thus, giving a high-level view of the design.

### 3.1 Search module interface

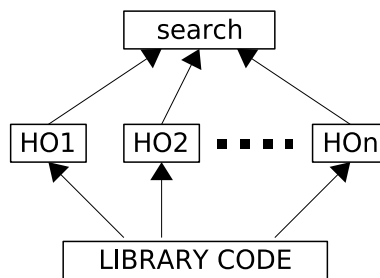


Fig. 2. Organisation of IMP's code with respect to the search module.

Figure 2 shows the basic organisation of IMP's code with respect to the search module. In the figure `search` is the search module, `HO1`, `HO2`,  $\dots$ , `HOn` are higher order predicate terms, and `LIBRARY CODE` is a set of modules as described in Table 1.

The interface for the search module exports predicates for different search strategies. Each search strategy is made abstract, and thus re-usable, by accepting as input higher order predicate terms for things unrelated to the search strategy *per se*, such as the successor, optimise and semantics functions. In Fig 2 these are represented by `HO1`, `HO2`,  $\dots$ , `HOn`. For an example consider how the breadth-first strategy is exported:

```

:- interface.

    % Using a breadth first search strategy return the first hypothesis with
    % correct semantics found in the hypothesis language. It's semidet b/c
    % it might be the case that there is no solution.
    %
:- pred bf(list(H),hypothesis_literals(V),background_knowledge(H,V),
           examples(V),successors(H,V,SS,successors(H)),
           solution_semantics(H,V,SS),SS,SS,H).
:- mode bf(in,in,in(background_knowledge),
           in,in(successors),in(solution_semantics),in,out,out)
           is semidet.

:- implementation.

bf(StartHs,HLs,BK,Es,Successor,Semantics,!SS,Solution) :- ...

```

where **StartHs** are the starting hypotheses, **HLs** is a description of literals which can appear in a hypothesis, **BK** is a description of the background knowledge, **Es** are all examples, **Successor** is a higher order predicate term representing a successor function, **Semantics** is a higher order predicate term which gives the semantics of a hypothesis, **!SS** is a state variable which is passed to the higher order predicates, and, finally, **Solution** is the first solution found.

The search is independent of hypothesis representation by using the **H** type variable, and independent of a particular problem by using type **V** for problem values. The **SS** type variable stands for *search state*, and enables data of the callers choice to be threaded through the **successor** and **solution\_semantics** higher order terms via the state variable **!SS**. This enables the caller of the search to store any information in the search state without it having to be represented explicitly as an argument. So, here we have achieved abstraction by using higher order predicate terms and polymorphic types.

When writing the higher order predicates terms for input to a search we can make use of the many modules available as described in Table 1.

### 3.2 Writing an ILP algorithm with IMP

An important module supplied with IMP is the `ilp_algorithms` module, which exports predicates representing ILP algorithms. Currently implemented algorithms are a top-down breadth-first search (with or without a bottom clause), branch and bound breadth-first search (with or without a bottom clause), an Aleph sat and reduce algorithm and the Aleph induce cover algorithm. The Aleph sat and reduce algorithm has the interface,

```
:- interface.
:- pred aleph_sat_reduce(int,hypothesis_literals(V),
    background_knowledge(definite_clause(V),V),examples(V),int,
    int,int,int,int,io,io).
:- mode aleph_sat_reduce(in,in,in(background_knowledge),in,in,in,in,
    in,in,di,uo) is det.

:- implementation.

aleph_sat_reduce(EId,HLs,BK,Es,CL,Noise,NodeLimit,VarDepth,RecallBound,!IO) :- ...
```

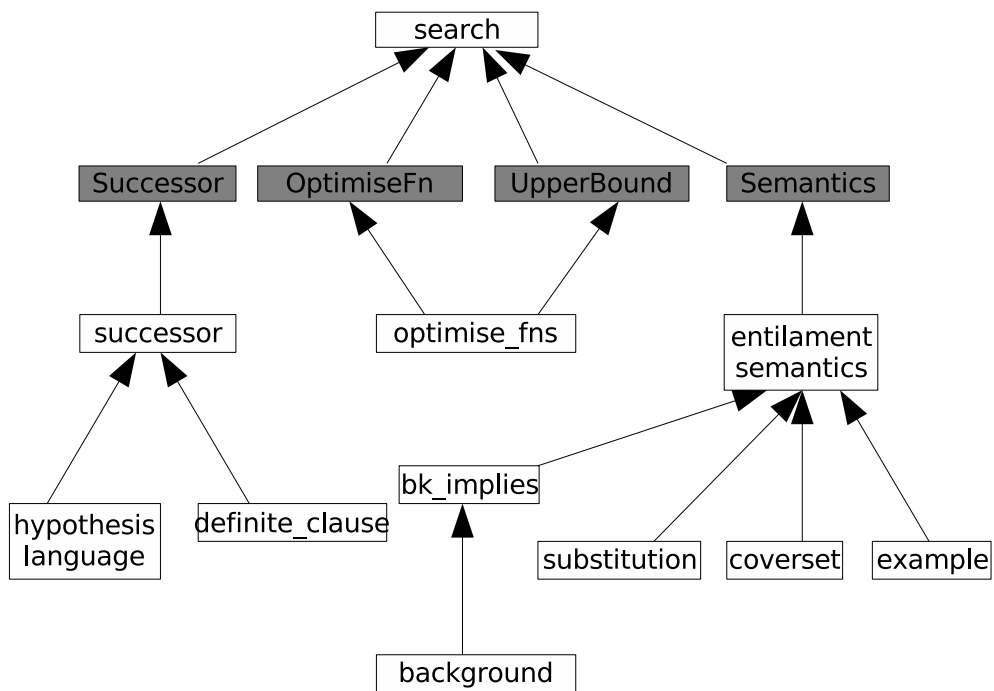
The `aleph_sat_reduce` algorithm is independent of problem and so uses the type variable **V** but uses definite clauses for hypothesis representation and so has `definite_clause(V)` for the hypothesis type.

At the moment recursion is not implemented for algorithms involving a bottom clause. We can cover test recursive clauses already and are working on producing bottom clauses involving recursive clauses.

All inputs and settings of the algorithm are given as inputs to the algorithm predicate. The alternative would be to define predicates for the settings and the system code to call the predicate. This is not possible, however, if we want to

separately compile the system code as a library, because then the settings would need to be known at compile time. We also feel that passing all settings as parameters to the search makes explicit exactly how each search can be customised, and is thus clearer for the user.

As stated previously one of the goals of this research is to maximise reusability of code and thus minimise effort to write ILP algorithms. To write an ILP algorithm we can use any code, but much work can be alleviated by using the library code. If we wish our algorithm to do a search we can use a search strategy from the search module and prepare higher order predicates using the library code. An example of how the `aleph_sat_reduce` algorithm uses the search module is shown in Figure 3.



**Fig. 3.** Diagram illustrating how `aleph_sat_reduce` uses the search module.

### 3.3 Representation specific interpreters

We follow the work of [4] and represent hypotheses as ground terms and use an interpreter for cover testing. Due to hypotheses being represented as ground terms, hypothesis interpretation speed will be crucial for any algorithm which relies on repeated cover testing against examples. To maximise efficiency of interpretation, at compile-time we generate a problem specific predicate to interpret

each background literal. Again, this is as in [4]. The difference of this work comes at the top-level of the interpreter. When cover testing, as well as passing the hypothesis to be tested, we also pass a higher order predicate term to interpret the hypothesis. This abstraction enables us to re-use the cover testing code with different hypothesis representations. It also allows us to optimise interpretation, because now we can write specialised interpreters for certain cases. The following is an interpreter for any definite clause as specified by `definite_clause.m`,

```
definite_clause_implies(BK,H,E) :-
    definite_clause.head(H,Head),
    unify(Head,E,S), % S is a substitution
    definite_clause.body(H,Body),
    definite_clause_implies_conj(BK,Body,S).

:- pred definite_clause_implies_conj(background_knowledge(H,V),
    list(literal(V)),substitution(V)).
:- mode definite_clause_implies_conj(in(background_knowledge),in,in) is semidet.
definite_clause_implies_conj(_,[],_).
definite_clause_implies_conj(BK,[L|Ls],SO) :-
    ( definite_clause.literal_is_dynamic(L) ->
        background_knowledge.dynamic_bk_implies_literal(BK,L,SO,S)
    ; % literal is static
        background_knowledge.static_bk_implies_literal(BK,L,SO,S)
    ),
    definite_clause_implies_conj(BK,Ls,S).
```

`definite_clause_implies` succeeds when the background knowledge BK and hypothesis H implies the example E. Each literal stores whether it is dynamic or static. Dynamic literals can be in the head of a predicate which is asserted to the background knowledge at run-time. Static literals cannot be asserted and therefore only consist of those clauses compiled. If the literal is dynamic then we have to check the dynamic background knowledge for a clause matching the literal; otherwise, to interpret a static literal we only need call the problem specific interpreter. In the case where we are doing a single clause search this check is redundant because there will always be no dynamic background knowledge. The following interpreter takes advantage of this special case:

```
% Specialised interpreter for definite clauses with no constants in
% the head, and where we only wish to learn a single clause
%
definite_clause_implies_single_clause_noconstants_norecursion(BK,H,E) :-
    unify_single_predicate_no_constants(E,S),
    definite_clause.body(H,Body),
    definite_clause_implies_static_conj(BK,Body,S).

% Succeeds if the static background knowledge implies the list of literals.
%
```

```

:- pred definite_clause_implies_static_conj(background_knowledge(H,V),
      list(literal(V)),substitution(V)).
:- mode definite_clause_implies_static_conj(in(background_knowledge),
      in,in) is semidet.
definite_clause_implies_static_conj(_,[],_).
definite_clause_implies_static_conj(BK,[L|Ls],S0) :-
      background_knowledge.static_bk_implies_literal(BK,L,S0,S),
      definite_clause_implies_static_conj(BK,Ls,S).

```

This interpreter is also specialised for the case where there are no constants in the head of the hypothesis clause, because then unify need not do any checking of E against H and so can simply just return the example as a substitution.

## 4 Using IMP as an ILP system

To solve an ILP problem with IMP the user must begin by specifying the background knowledge and examples of the problem. Background knowledge must be in the form of a Mercury module with all predicates and types available in the hypothesis language exported in the interface. Examples are given to the system as data whose type is specified in the example module. Each example represents a ground atom and has the form: `example(PredName,PredArgs,PosOrNeg)`. So, for the trains problem the first pos and neg examples would be,

```

example("eastbound",[train(east1)],pos).
example("eastbound",[train(west6)],neg).

```

When the user has written their background knowledge module they can use the program `imputgen` to generate the problem specific literal interpreter, `bk_implies.m`, and a run file. The run file is a Mercury module containing the 'main' predicate with a call to an `ilp` algorithm predicate and default settings for all the algorithm parameters. `imputgen` guesses the hypothesis language description by putting a literal for each exported predicate mode of the user's background knowledge. After checking and modifying the run file with appropriate parameters all that is left is to compile. We only need put,

```

EXTRA_LIB_DIR=/path/to/imp/lib/
EXTRA_LIBRARIES=imp

```

in an `Mmakefile` and run the commands,

```

mmake run_file_module.depend
mmake run_file_module

```

We should then have an executable file which will run the specified algorithm with the specified parameters.

So to summarise, there are four steps to use IMP to solve a problem,

1. Specify the background knowledge and examples in an appropriate format

2. Run `imputgen` on the background knowledge file
  3. Check and modify the generated run file
  4. Compile the run file and link with the IMP library
- Changes from previous system.
  - It's a software library.

## 5 Implementing efficient cover sets

The *cover set* for an induced clause is the set of examples that it *covers*, i.e. the set of examples which logically follow from the clause, together with background knowledge and any previously induced clauses. Rather than store the examples themselves indices to the examples are stored so that cover sets are sets of non-negative integers. Generally, for each induced clause there are two cover sets, one for positive examples and one for negatives: since they are both implemented in the same way we draw no distinction between them in what follows. The process of determining the cover set for a clause is called *cover testing* and generally this is the most time-consuming part of the ILP process.

Effective implementation of cover sets is crucial to the overall efficiency of any ILP algorithm. In a top-down search, by storing the cover set of any induced clause  $C$  the cover testing of its specialisations can be greatly sped up by restricting attention to only those examples in  $C$ 's cover set. In some situations, the cover set of a clause can be computed directly as the intersection of the cover sets of two parent clauses; in other cases such an intersection only determines a superset of the desired cover set so some cover testing is still required. When learning multiple clauses it is also possible to 'cache' cover sets generated in one top-down search to speed up a subsequent one [1]: this can lead to significantly faster computation—as long as it does not exhaust the machine's memory! All of these operations are implemented in the ILP system Aleph [9], for example.

A cover set data structure should thus have the following properties:

1. It should allow rapid cover testing.
2. It should be compact.
3. It should allow rapid intersection of cover sets.

### 5.1 The IMP coverset data type

The data structure implemented in the current version of IMP is a modification of the RL-tree data structure introduced by [5]. We proceed by defining the IMP data structure in this section and compare with RL-trees later on in Section 5.4. In IMP there are four different ways of representing a cover set so the Mercury type used to represent cover sets is a discriminated union of 4 subtypes. Having no more than 4 subtypes is particularly efficient in Mercury; "Since the low-order bits of pointers to aligned words are always zero, we can use these bits as a tag, giving us four different tag values ... For types with up to four different

alternatives, the two tag bits are sufficient to distinguish them; the remainder of the word is a pointer to a sequence of words on the heap containing the arguments of the function symbol, if any.” [7].

Fig 4 shows the Mercury source defining the `coverset` type. We have chosen to implement this as an *abstract* type in the cover set module (`coverset.m`) so that the rest of the IMP system cannot know how it is implemented. This provides a useful guarantee of modularity. An explanation of the four `coverset` subtypes follows.

**white** This term represents the empty set.

**black**(Min,Max) A term of this form represents the interval of integers  $\{i : \text{Min} \leq i < \text{Max}\}$

**bitmap**(Min,N,I) In a term of this form Min, N and I are all integers and, in particular, I is assumed to be a 32-bit integer. (At present all our cover set code is hard-coded for 32-bit integers.) Such a cover set represents a subset of the interval  $\{i : \text{Min} \leq i < \text{Min} + 32\}$ .  $i$  is included in this subset iff the  $(i - \text{Min})$ th bit of I is set, where the least significant bit is the 0th bit. N records the number of bits set—the size of the cover set.

**grey**(Min,Max,N,C1,C2,C3,C4) In a term of this form Min, Max and N are integers and C1, C2, C3 and C4 are cover sets. Such a cover set represents a set of integers Grey which is a subset of  $\{i : \text{Min} \leq i < \text{Max}\}$ . Let BlockSize be the smallest integer such that  $\text{BlockSize} \times 4 \geq \text{Max} - \text{Min}$  and define S1, S2, S3 and S4 as follows:

$$\begin{aligned} S1 &= \text{Grey} \cap \{i : \text{Min} \leq i < \text{Min} + \text{BlockSize}\} \\ S2 &= \text{Grey} \cap \{i : \text{Min} + \text{BlockSize} \leq i < \text{Min} + 2 \times \text{BlockSize}\} \\ S3 &= \text{Grey} \cap \{i : \text{Min} + 2 \times \text{BlockSize} \leq i < \text{Min} + 3 \times \text{BlockSize}\} \\ S4 &= \text{Grey} \cap \{i : \text{Min} + 3 \times \text{BlockSize} \leq i < \text{Max}\} \end{aligned}$$

then C1, C2, C3 and C4 are cover set representations of S1, S2, S3 and S4, respectively. N records the size of Grey.

It is not difficult to see that any set of integers can be represented by the `coverset` type; in fact, only the `grey` and `bitmap` types are needed. It is also not difficult to see that many sets can be represented in several ways. For example, the empty set can be represented by any of the 4 subtypes. Our code effects the following normalisation: represent a set by the `white` subtype if possible, failing that the `black` subtype, failing that the `bitmap` subtype and, as a last resort the `grey` subtype. Table 2 has some example cover sets matched with their `coverset` type representations.

## 5.2 Cover testing in IMP

Basic cover testing in IMP is effected by the function `covertest.filter` which has the declaration shown in Fig 5. The second argument of `filter` is a higher-order term representing a predicate which takes an example id as input and

```

:- interface.
:- type coverset.

:- implementation.
:- type coverset --->
    white;
    black(int,int);
    grey(int,int,int,coverset,coverset,coverset,coverset);
    bitmap(int,int,int).

```

Fig. 4. Mercury source defining the coverset type

$\{i : 0 \leq i < 507\}$	black(0, 507)
$\emptyset$	white
$\{1, 2, 3\}$	bitmap(0, 3, 14)
$\{3, 400\}$	grey(0, 507, 2, bitmap(0, 1, 8), white, white, bitmap(381, 1, 524288))
$\{i : 0 \leq i < 507 \text{ and } i \text{ is even}\}$	grey(0, 507, 254, grey(0, 127, 64, bitmap(0, 16, 1431655765), bitmap(32, 16, 1431655765), bitmap(64, 16, 1431655765), bitmap(96, 16, 1431655765)), grey(127, 254, 63, bitmap(127, 16, -1431655766), bitmap(159, 16, -1431655766), bitmap(191, 16, -1431655766), bitmap(223, 15, 715827882)), grey(254, 381, 64, bitmap(254, 16, 1431655765), bitmap(286, 16, 1431655765), bitmap(318, 16, 1431655765), bitmap(350, 16, 1431655765)), grey(381, 507, 63, bitmap(381, 16, -1431655766), bitmap(413, 16, -1431655766), bitmap(445, 16, -1431655766), bitmap(477, 15, 715827882)))

Table 2. Some cover sets with coverset representation

which succeeds iff the induced clause under consideration (the *current clause*) covers the example. The first argument is just a `coverset` representation of the examples to test. The result is the cover set of examples covered by the current clause. We have chosen to implement cover testing as a function. This is equivalent to using a deterministic predicate but allows slightly more compact source code.

```
:- func filter(coverset,pred(int)) = coverset.
:- mode filter(in,pred(in) is semidet) = out is det.
```

**Fig. 5.** Declaration for the cover testing function `filter`

The key step in cover testing is processing `bitmap` cover sets. Our code creates a bitmap for the output cover set ‘directly’ without, say, translating back and forth to a list. The predicate `filter_bitmap_acc` in Fig 6 shows the basic idea (where  $\vee$  is bitwise or). In `filter_bitmap_acc`, `I` is a bitmap of the examples to be cover tested and `Min` is an offset: it is the example id represented by bit 0 of `I`. `HO` is the higher-order term previously discussed and `Acc` and `NAcc` are accumulators which are both 0 when `filter_bitmap_acc` is first called. The code is straightforward. If `I=0` then there are no examples to cover test and so the process terminates. Otherwise the position of the first set bit in `I` is found, added to the offset `Min` and `HO` is called to see if the example `Min+Bit` is covered. If it is then the appropriate bit in the accumulator is set. In the recursive call the bit just processed is unset by the function `unset_first_bit`.

```
:- pred filter_bitmap_acc(int,int,pred(int),int,int,int,int).
:- mode filter_bitmap_acc(in,in,pred(in) is semidet,in,in,out,out) is det.

filter_bitmap_acc(I,Min,HO,Acc,NAcc,J,NOut) :-
(
  I = 0 ->
  J = Acc, NOut = NAcc;
  Bit = first_bit(I),
  (
    HO(Min+Bit) ->
    Acc0 = Acc \vee bitmask(Bit), NN = NAcc+1;
    Acc0 = Acc, NN = NAcc
  ),
  filter_bitmap_acc(unset_first_bit(I),Min,HO,Acc0,NN,J,NOut)
).
```

**Fig. 6.** Cover testing cover sets represented as bitmaps

For `filter_bitmap_acc` to be efficient, calls to `first_bit(I)` and `unset_first_bit(I)` must be fast. We do this by using an array to store the values of these functions for all integers in the range  $0 < i < 2^8$ . This is a simple C static integer array wrapped inside a Mercury function. It is trivial to just drop C source directly into Mercury source. The function `bitmask/1` is similarly implemented using an array of size 32.

Since most bitmaps do not represent integers in the range  $0 < i < 2^8$  this approach cannot be used directly. So instead we process bitmaps in 4 blocks of 8. This is done by right shifting the input bitmap by 0, 8, 16 and 24 bits successively and then bitwise anding with  $11111111_2 = 255$  to produce a number in the range  $0 \leq i < 2^8$

In some cases cover testing can be aborted early before a new cover set is constructed. This can happen when processing negatives: if at any point more negatives are known to be covered than some noise limit, then there is no point finding the entire set of negatives since the clause is now known not to be acceptable. In IMP this is effected by inputting a higher-order term which both cover tests and checks for early stopping; cover testing simply fails if and as soon as too many examples are covered.

### 5.3 Intersecting cover sets

Although our code computes the intersection of any two cover sets, those actually produced during top-down searches are dealt with particularly efficiently. To see why this is the case note that parentless clauses in a top-down search (and clauses whose parents have full cover) have to check the set of all examples, which is `black(0,Max)` where `Max` is the number of examples in the data. Typically the cover set produced will be a `grey` one of the form `grey(0,Max,N,C1,C2,C3,C4)`. Intersecting with another similar cover set of the form `grey(0,Max,M,D1,D2,D3,D4)` is quick because ‘child’ coverset `Ci` need only be intersected with `Di`. An inductive argument show that cover sets produced further down in the search tend to be these ‘in synch’ `grey` cover sets.

Cover sets which are not `grey` are even easier to intersect. `white` cover sets and disjoint pairs of cover sets can obviously be dispensed with quickly. Non-disjoint pairs of bitmaps are, like `grey` cover sets, ‘in synch’ and so can be intersected with a single bitwise and operation.

### 5.4 Comparison with RL-trees

As stated in [5], “The basic idea behind RL-Trees is to represent [a] disjunct set of intervals in a domain by recursively partition[ing] the domain interval into equal subintervals”. This recursive process stops when the subinterval is white (empty) or black (full) or is represented by a *list node* which represents an interval of a fixed size. In the RL-tree implementation of [5], the presence of an example within the interval represented by a list node is represented by a set bit.

Our `coverset` type is thus just an RL-tree with the minor modification that each coverset term (of whichever subtype) is a standalone representation of a set. This allows the coverset  $\{0, 3\}$  to be represented as `bitmap(0,2,9)` irrespective of the size of the ‘domain’ (i.e. the set of all possible examples) whereas the RL-Tree representation of  $\{0, 3\}$  as a subset of  $0 \dots 65$  is something more like `grey(grey(bitmap(9),white,white,white),white,white,white)`

The bitmap is more compact since the interval it covers can be inferred from its parent node. Our goal in this alteration was to simplify and compress the representation of small cover sets in big domains. It also allows simpler intersection. Note that we also (redundantly) store the size of all cover sets: this is a space for time tradeoff.

## 6 Benchmarking tests

### 6.1 The benchmark ILP learning problems

In our benchmark test both existing well-known ILP problems and our own synthetic problems were used. The former comprised the Michalski’s trains problem [6] (Trains), learning morpho-syntactic descriptions for Slovene (MSD) [2] and the mutagenicity learning problem (Mutagenesis) [8]. The `float_*` problems are synthetic problems where all background predicates were restricted to computations and comparisons between floats. In each case a target predicate was invented and positive and negative examples for this target were generated. We considered a number of target predicates which differed in their determinacy. For example, `float_det_target1` is a learning problem where examples are generated from a deterministic target. `int_det_target1` is similar except that integers were used.

Examples for the graphs problem were generated by a random graph generator and classified according to this target predicate:

```
target1(G) :-
    delete(_,G,G1),
    delete(_,G1,G2),
    triangulated(G2).
```

In other words, examples are graphs and are positive iff it is possible to construct a triangulated graph by deleting two vertices. A simple list based representation of graphs were used together with a simple test for being triangulated based on vertex elimination.

It is worth mentioning that in an initial experiment with this problem, Aleph was so slow we had to terminate the process. Upon investigation we realised that Aleph’s background knowledge (which was the same as IMP’s without the declarations) could be made much more efficient by writing two predicates to define a connection between graph vertices; one with lots of cuts. The point is that we did not have to worry about cuts for IMP since Mercury does not allow them! It is generally much easier to write correct and efficient background knowledge with IMP, due to declarative nature of Mercury.

## 6.2 Fixing modes for mutagenesis

In initial comparisons between IMP and Aleph on the mutagenesis problem IMP was found to be far slower. One problem we found was that predicates in the mutagenesis background knowledge were being called with the ‘wrong’ modes. There were cases where a predicate was being called with one of its arguments ground but with a mode where that argument had mode `out` (despite the fact that appropriate modes were available). This meant that no indexing for that argument was being effected, causing a drastic slowdown. This was happening because the ‘wrong’ mode appeared before the right one in the Mercury source file and so could be solved by re-ordering mode declarations.

Although this brought about improvements IMP was still slower. We suspect this is due to the way IMP is obliged to handle the mutagenesis background knowledge predicates `atm/5` and `bond/4` which are defined by 5894 and 6309 ground facts, respectively. Compiling these as normal predicates caused memory problems, so we took the obvious step of using Mercury *fact tables* which provide a specialised efficient way of compiling such predicates.

As well as having many clauses both these predicates require many different modes. Fig 7 shows the 5 modes for `atm/5`. `bond/4` has 6 modes. Programs needing only one of these modes showed acceptable efficiency but when all were declared, execution was slow, presumably due to remaining cases of the ‘wrong’ mode being called. We suspect this problem may be due to the use of fact tables.

```
:- pred atm(drug,atomid,element,int,charge).
:- mode atm(in,in,in,in,out) is semidet.
:- mode atm(in,in,out,out,out) is semidet.
:- mode atm(in,out,in,in,out) is nondet.
:- mode atm(in,out,in,out,out) is nondet.
:- mode atm(in,out,out,out,out) is nondet.
```

Fig. 7. Declaration for the mutagenesis background predicate `atm/5`

A workaround for this problem was possible because Mercury allows the user to write (potentially impure) code defining different clauses for different modes of a predicate. This allowed us to define `atm/5` as shown in Fig 8. Each of the 5 predicates `atm_in_in_in_in_out`, ... `atm_in_out_out_out_out` has a single mode and is defined by a separate fact table containing the same information (modulo a change in predicate name). This workaround is what was used to produce our comparison with Aleph. Essentially we bullied Mercury into producing the code that it should be producing anyway.

## 7 Results

Table 7 gives most of our results, which were done with the sat-reduce algorithm. The time  $t$  is the total run-time excluding  $t_{Ex}$ , the time to read the examples.

```

:- pragma promise_pure(atm/5).
atm(Drug::in,Atom::in,Type::in,N::in,Charge::out) :-
atm_in_in_in_in_out(Drug,Atom,Type,N,Charge).
atm(Drug::in,Atom::in,Type::out,N::out,Charge::out) :-
atm_in_in_out_out_out(Drug,Atom,Type,N,Charge).
atm(Drug::in,Atom::out,Type::in,N::in,Charge::out) :-
atm_in_out_in_in_out(Drug,Atom,Type,N,Charge).
atm(Drug::in,Atom::out,Type::in,N::out,Charge::out) :-
atm_in_out_in_out_out(Drug,Atom,Type,N,Charge).
atm(Drug::in,Atom::out,Type::out,N::out,Charge::out) :-
atm_in_out_out_out_out_out(Drug,Atom,Type,N,Charge).

```

**Fig. 8.** Ensuring the right modes are used for atm/5

**Table 3.** Results comparing IMP with Aleph.  $t_{Ex}$  is the time to read the examples from a file.  $n$  the number of clauses generated and tested.  $t$  the total search time excluding time  $t_{Ex}$ .  $n/t$  is clauses per second.

System	Problem	$ Ex $	$t_{Ex}$	$n$	$t$	$n/t$
IMP	float_det_target1	50000	3	175	11	15.91
Aleph	float_det_target1	50000	1	177	12	14.75
IMP	float_det_target2	100000	7	175	20	8.75
Aleph	float_det_target2	100000	2	382	47	8.13
IMP	float_multi_target1	100000	7	5681	144	39.45
Aleph	float_multi_target1	100000	2	3691	100	36.91
IMP	float_multi_target2	100	0	100000	49	2040.82
Aleph	float_multi_target2	100	0	100000	32	3125
IMP	float_semidet_target1	11426	1	349	4	87.25
Aleph	float_semidet_target1	11426	0	179	3	59.67
IMP	graphs	50	0	24119	3084	7.82
Aleph	graphs	50	0	12104	88751	0.14
IMP	int_det_target1	200000	13	40	8	5
Aleph	int_det_target1	200000	4	42	12	3.5
IMP	MSD	2815	1	30000	34	882.35
Aleph	MSD	2815	0	30000	76	394.74
IMP	Mutagenesis	230	0	823	46	17.89
Aleph	Mutagenesis	230	0	438	26	16.85
IMP	Mutagenesis	230	0	13951	64	217.98
Aleph	Mutagenesis	230	0	4684	28	167.29
IMP	Trains	30000	1	38	2	19
Aleph	Trains	30000	0	73	2	36.5

IMP takes longer to read the examples. This is because to Mercury performing type check for each term read when performing IO. A benefit of this is that our examples are confirmed to be 'well-formed' and type correct, but a disadvantage is the extra time taken. In order to get a fair comparison of search and cover testing speed we exclude  $t_{Ex}$  from  $t$ .

Both systems construct an equivalent bottom clause for each test; however, in general the literals are ordered differently and so we get different searches. Different searches have different pruning opportunities and so we get a discrepancy in  $n$ . For these reasons we will mainly be comparing  $n/t$ , clauses per second.

Overall results are good for IMP, and in some cases they are excellent. The graphs problem is our best result with a 56 times speed-up compared with Aleph. This problem has very computationally demanding background knowledge, makes use of constrained determinisms and the basic 'char' type. The more demanding the background knowledge the greater the time spent executing the background knowledge compared to search and interpretation code. If the background knowledge also makes use of constrained determinisms and basic types then Mercury can use this information for optimisation and therefore has a faster run-time than Prolog. See [7] for details on the Mercury execution algorithm.

To perform cover testing Aleph asserts clauses at run-time into the Prolog database. To determine how efficient our interpreter is relative to this we need to look at small searches with simple background knowledge and large numbers of examples. These attributes are displayed in the float\_det\_target1, float\_det\_target2, int\_det\_target1 and Trains problems. Here we see that IMP's interpretation overhead is, in general, slightly less than Aleph's.

In the float\_multi\_target2 test Aleph noticeably outperforms IMP. This test consists of small numbers of examples and a very large search. IMP slightly outperforms Aleph in the float\_multi\_target1 test which uses the same background knowledge but has more examples and a smaller search. This leads one to believe that IMP's search code is less efficient than Aleph's.

We have also tested the induce-cover algorithm with the MSD problem. Here both IMP and Aleph find the same solutions. The total search time for IMP was 8 minutes and 39 seconds compared to 15 minutes and 32 seconds for Aleph.

## 8 Conclusion

Our central conclusion is that IMP outperforms Aleph significantly when there are computationally expensive predicates in the background knowledge. This is simply because Mercury is substantially faster than Prolog so that any overhead associated with interpreting the ground terms representing induced clauses is more than overcome.

In other cases the systems are more closely matched with IMP usually ahead. Once a user is familiar with the Mercury language it is easier to 'get the background knowledge right' than with Prolog. On the other hand, there are clearly problems for IMP (see Section 6.2) when there are many ground facts in the back-

ground knowledge. We expect that recent work on *exo-compilation* in Mercury [3] will improve performance in such cases.

In this paper IMP's modular design (see Section 3) has not been fully exploited since we have focused on a comparison with Aleph. In future work, we intend to implement other ILP approaches by an appropriate choice of modular components. Our new cover set implementation is a significant improvement over our former simpler approach and we will continue work on optimising code. Because IMP's induced clauses are ground terms (i.e. on the object level) it is possible to reason about them logically. We hope to exploit this in future optimisation work.

## References

1. James Cussens. Part-of-speech tagging using Progol. In *Inductive Logic Programming: Proceedings of the 7th International Workshop (ILP-97)*. LNAI 1297, pages 93–108. Springer, 1997.
2. James Cussens, Sašo Džeroski, and Tomaž Erjavec. Morphosyntactic tagging of Slovene using Progol. In Sašo Džeroski and Peter Flach, editors, *Inductive Logic Programming: Proc. of the 9th International Workshop (ILP-99)*, Bled, Slovenia, June 1999. Springer-Verlag.
3. Bart Demoen, Phuong-Lan Nguyen, Vítor Santos Costa, and Zoltan Somogyi. Dealing with large predicates: exo-compilation in the WAM and in Mercury. In *Proceedings of the Seventh International Colloquium on Implementation of Constraint and Logic Programming Systems*, Porto, Portugal, September 2007.
4. Barnaby Fisher and James Cussens. Inductive Mercury programming. In Stephen Muggleton and Ramon Otero, editors, *Inductive Logic Programming: Proceedings of the 16th International Conference (ILP-06)*, Santiago de Compostela, 2007. Springer.
5. Nuno A. Fonseca, Ricardo Rocha, Rui Camacho, and Fernando Silva. Efficient Data Structures for Inductive Logic Programming. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming (ILP 2003)*, volume 2835 of *LNCS (Lecture Notes in Artificial Intelligence)*, pages 130–145, Szeged, Hungary, September 2003. Springer-Verlag.
6. Yves Kodratoff and Ryszard S. Michalski, editors. *Machine Learning: An Artificial Intelligence Approach*, volume 3. Morgan Kaufmann, 1990.
7. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.
8. A. Srinivasan, S.H. Muggleton, M.J.E Sternberg, and R.D. King. Theories for mutagenicity: a study of first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
9. Ashwin Srinivasan. *The Aleph Manual*, version 4 and above edition, 2004.