Compiling Formal Specifications to Oz Programs

Tim Wahls

Dickinson College, P.O. Box 1773, Carlisle, PA 17013, USA
wahlst@dickinson.edu

Abstract. Although formal methods have the potential to greatly enhance software development, they have not been widely used in industry (particularly in the United States). We have developed a system for executing specifications by compiling them to Oz programs. Executability is a great aid in developing specifications, and also increases the usefulness of specifications by allowing them to serve as prototypes and test oracles. In this work, we describe how we have used the Oz language both as a translation target and in implementing a library of procedures used by the generated programs. Oz is ideal for our purposes, as it has allowed us to easily use declarative, concurrent constraint and graphical user interface programming together within a single framework.

1 Introduction

Formal specifications of software system functionality have a number of important advantages over specifications expressed in English, such as conciseness, the ability to serve as a basis for proofs of program correctness or of other important system properties, and freedom from ambiguity and implementation bias. A large number of formal specification notations have thus been developed, including VDM [1, 2], Z [3, 4], B [5, 6], JML [7], and SPECS-C++ [8, 9].

However, formal specifications have not been widely adopted in industry, particularly in the United States. The perception is that the cost of using formal methods does not justify the benefits. The use of formal methods is difficult to justify to clients and managers who typically do not understand the notation. Hence, there is a need for tools and techniques that make specifications accessible to nontechnical users, and that reduce the cost of developing formal specifications.

One way to approach these problems is through the use of executable formal specifications. The ability to execute and validate specifications eases their development, as specifiers can immediately check intuition about their specifications. An executable specification can serve as a prototype of the final system, allowing nontechnical users to interact with the specification and to provide feedback on it. Clients and managers can also see the utility of an executable specification as a test oracle. Unsurprisingly, many executable specification languages and execution techniques have been developed [10, 11, 12, 13, 14, 15, 16, 17, 18].

P. Van Roy (Ed.): MOZ 2004, LNCS 3389, pp. 66–77, 2005.
© Springer-Verlag Berlin Heidelberg 2005

We have developed a system for executing SPECS-C++ specifications [9] by compiling them to Oz [19, 20, 21] programs[1]. Our system requires no hand translation, no explicit identification of the range of possible values for variables in most cases, and also introduces almost no implementation bias into specifications. We have also developed a formal semantics [8] and a graphical user interface [22] for our system.

SPECS-C++ is similar to other model-based specification languages such as VDM and Z in that operations are specified using first order pre- and postconditions written over a fixed set of model types. These types include C++ primitive types, as well as tuples, sets, sequences, multisets, functions and maps. SPECS-C++ is designed for specifying the interfaces of C++ classes, and so operation signatures are given as C++ member function prototypes. The model types in SPECS-C++ include references so that interface specifications can handle aliasing and object containment.

Figure 1 presents the specification of a C++ template class `Table`, which allows references to values of any type to be stored and indexed by integer keys. This type would most naturally be modeled as a function from integers to value references, but we have modeled it as a sequence of tuples in order to better demonstrate the kinds of specifications that can be executed. In fact, the specification becomes much easier to execute if a function is used, as there is then no need to use quantifiers and the `sortKeys` operation becomes nonsensical.

The `domains` section of the specification defines types for later use, while the `data members` section gives the abstract data members used to model instances of the class. The `constraints` section specifies any invariants that all instances of the class must satisfy. Here, the invariant is that for any two tuples at different indices within the sequence, the key values must be different (i.e. the sequence of tuples properly represents a function)[2]. The `abstract functions` section defines "specification only" functions that are not part of the interface of the class, but are useful for specification purposes. The abstract functions presented here respectively check that the table values are sorted by key, and that a particular integer value is in the domain of the table.

The member function specifications (following `public:`) describe the interface of the class that is available to client code. Angle brackets are used as sequence constructors, and `||` is sequence concatenation. The `modifies` clause specifies which objects can change from the prestate (before the operation executes) to the poststate. The `^` is used to dereference an object in the prestate, while `'` is used for the poststate value. The notation `theTable^` (for example) is a shorthand for `self^.theTable`. Note that the postconditions of `lookUp` and `sortKeys` are highly implicit – they simply describe the return value (specified as `result`) or poststate resulting from the operation with no indication of how

[1] The system originally generated Agents Kernel Language (AKL) programs, and was later ported to Oz.

[2] SPECS-C++ specifications are intended for use as C++ header files, so quantifiers (for example) can not be written as ∀ and ∃.

```
template <class Value> class Table {
/* model
  domains
    tuple (int key, Value& val) ElemType;
    sequence of ElemType TableType;
  data members
    TableType theTable;

  constraints
    \forall int i [1 <= i <= length(theTable) =>
      \forall int j [1 <= j <= length(theTable) /\ i != j =>
         theTable[i].key != theTable[j].key]]
  abstract functions
    define sorted(TableType t) as bool such that
       result = tobool(\forall int i [ 1 <= i < length(t) =>
                         t[i].key <= t[i + 1].key]);
    define inDom(int key, TableType t) as bool such that
       result = tobool(\exists int i [1 <= i <= length(t)
                                /\ t[i].key = key]);
*/

public:

Table();
/* modifies: self
   post: theTable' = <>
*/

void addEntry(int key, Value& val);
/* pre: ! inDom(key, theTable^)
   modifies: self
   post: theTable' = <(key, val)> || theTable^
*/

Value& lookUp(int key);
/* pre: inDom(key, theTable^)
   post: \exists ElemType e [e \in theTable^
                         /\ e.key = key
                         /\ result = e.val]
*/

void sortKeys();
/* modifies: self
   post: range(theTable^) = range(theTable') /\ sorted(theTable')
      /\ length(theTable^) = length(theTable')
*/

bool inDomain(int key);
/* post: result = inDom(key, theTable^)
*/

};
```

Fig. 1. The SPECS-C++ specification of class `Table`

this result is to be constructed. In particular, the postcondition of `sortKeys` simply specifies that the poststate value of the calling object is a sorted permutation of the prestate value. The postcondition of `sortKeys` is strong enough to uniquely determine the poststate, assuming that the prestate satisfies the invariant. The `inDomain` member function is provided so that client code can check the preconditions of `addEntry` and `lookUp`.

Our compiler translates SPECS-C++ specifications such as this one to Oz programs. The scanner for the compiler was generated using flex, and the parser using bison. The remaining components of the compiler are implemented in C++. All of the operations on the SPECS-C++ model types are implemented in a library of Oz procedures (as a functor), which are called from the generated programs. The library also includes the code for the graphical user interface (using the Oz embedding of Tk) and various utility procedures. Poststate values and variables bound by existential quantifiers are represented by fresh Oz variables, which are then constrained by the generated code. Additional information about the translation and the library is presented in the following section. A complete description of the translation is contained in [9], and a formal presentation of selected portions of the translation can be found in [8].

2 Compiling to Oz

In this section, we describe how we have used various features of Oz in implementing the specification execution system. We highlight features that have been particularly useful, or that we have used in potentially novel ways.

2.1 Threads

In a formal specification, conjuncts and disjuncts should be ordered in the most natural way for the specifier, or in a way that is intended to increase readability. Of course, the order has no effect on the meaning of the specification. However, in an Oz program, the order of statements in a procedure body and the order of choices in a **`choice`** or **`dis`** statement has a tremendous impact on performance and even on whether or not the program terminates. This issue is well known in the logic and constraint programming communities, where it is referred to as *literal ordering* [23, 24, 25]. Requiring the specifier to write specifications in a particular order is an unacceptable form of implementation bias, so the specification execution system must not rely on ordering properties of specifications.

Our primary approach to this problem has been to make liberal use of threads – the majority of the library procedures that implement SPECS-C++ operators are threaded. When called, these procedures block until their arguments are sufficiently defined to permit execution. This greatly decreases the sensitivity of the system to the order in which the specification is written – if a procedure is called “too early”, it simply suspends and waits until enough of its arguments are available to permit execution. Oz’s data-flow threads are ideal for this purpose, as no code is required for this synchronization. Additionally, Oz threads are sufficiently lightweight so that execution times remain reasonable, even for

programs that create many hundreds of threads (as the programs generated by our system often do).

For example, the procedure implementing the `length` operation on sequences is threaded. The procedure takes two arguments (the sequence and the length) and unblocks when either becomes known. If the sequence becomes known, the procedure constrains the length. (If only some prefix of the sequence becomes known, the procedure finds the length of the prefix and resuspends.) If the length becomes known, the procedure constrains the sequence to contain that number of free variables. Those variables can then be constrained by other parts of the specification that refer to index positions within the sequence. Note that implementing the length procedure in this manner makes it a constraint propagator in the Oz sense.

It is advantageous to call such threaded library procedures as soon as they could possibly perform some propagation. Calling such an operation too early does no harm (it simply suspends), while calling it too late may drastically hurt performance if unnecessary search is done. Hence, the specification compiler does some explicit reordering to move up calls to threaded procedures. An alternative to using threading in this manner is to reorder (unthreaded) statements/literals based on data flow analysis (to ensure that a procedure can always execute at the time that it is called). That has been implemented within the Mercury project [25], but is considerably more complex than our approach.

2.2 Choice Points

In Oz, choice points are explicitly created (by the programmer) when search is needed, and are later explored via backtracking. This contrasts sharply with traditional logic programming languages such as Prolog in which any conditional is expressed by creating an implicit choice point. Several of the procedures in the library of the specification execution system create choice points by using **`dis`** statements, and disjunction in a specification is translated to a **`choice`** statement. The library procedures that create choice points correspond to SPECS-C++ operators that are frequently used in underdetermined or nondeterministic computations. For example, the procedure implementing sequence indexing is implemented using **`dis`** to allow all indices where a particular element occurs within a sequence to be found, or to allow a range of possibilities for where an element is to be inserted into a sequence (depending on the mode of use).

The reordering done by the specification compiler moves calls to library procedures that create choice points as late as possible in the generated programs. The idea is to delay search until variables are as constrained as possible, so that a search path that can not lead to success fails as soon as possible. This reordering, in conjunction with judicious selection as to which library procedures create choice points, seems to be effective in controlling the amount of search done by the generated programs, and in reducing the sensitivity of the system to the order in which the specification is written. We have tested the system extensively with specifications that were intentionally written to cause search

(for example, the `sortKeys` member function specification of Figure 1, the specification of a maximum clique member function for a graph class, ...), and have systematically permuted the order of conjuncts, the order of disjuncts, the order of arguments where possible (i.e. to the `=` operator), and so on. In every case, the system could execute specifications over large enough inputs for reasonable testing purposes.

2.3 Computation Spaces

An Oz space is a complete encapsulation of a computation. A space consists of a constraint store and one or more threads that operate on the store. In Oz, the programmer can create and execute a space explicitly. Once the space becomes stable (can no longer execute), the programmer can query it to determine whether it succeeded, failed or suspended (contains at least one blocked thread), and the result of a succeeded space can be extracted. The idea of spaces in implicit in other languages – for example, backtracking search in Prolog can be thought of as creating one space for each of a set of rules with the same head, and then executing the space corresponding to the first rule. If that space fails, then it is discarded and the space for the second rule is executed (and so on). However, Oz is unique in that spaces can be explicitly created and manipulated by the programmer.

We have used explicit spaces in several ways in the specification execution system. In one (possibly novel) use, spaces are used to determine whether all threads resulting from a member function call have terminated. When a member function is called, a new space is created, and the Oz procedure representing that member function is executed within the space. If the space becomes stable and suspended, at least one thread blocked and could not be resumed. In this case, the system reports to the user that the specification did not contain sufficient information to permit execution. If spaces were not used in this manner, explicit synchronization would be necessary to determine if some threads remained blocked. This synchronization is not difficult in Oz (especially in programs written by an experienced human), but would have added some unnecessary complication to both the specification compiler and the library.

A nice property of spaces is that any binding of nonlocal variables is not visible outside of the space (and its child spaces). The specification execution system takes advantage of this property for executing assertions simply for their boolean value (rather than as constraints). For example, the antecedent of an implication is executed in a new space. If the space succeeds, the consequent is treated as a constraint. If the space fails, the consequent is ignored. If the space is stuck, the system reports that the specification is not executable. If the antecedent were just treated as a constraint (not executed in a new space), an explicit choice point would be required so that any variable bindings that resulted could be "backed out" in case the antecedent were false (i.e. treating $P \Rightarrow Q$ as $\neg P \vee Q$). We have experimented with both approaches and found creating a

choice point for this purpose to be considerably less efficient in practice. Similarly, a negated assertion is executed within a new space and simply succeeds if the space fails and vice versa with no danger of determining nonlocal variables (that correspond to poststate values of the specification). This is quite similar to the behavior of the Oz **not** statement, but seems to be more general. We have found cases where this approach allows a specification to be executed, while using a **not** statement causes the thread to block.

Reification is another option for executing assertions for their value only, as the value of the assertion would then simply be the value of the boolean variable associated with it. However, this approach would require implementing reified versions of all of the SPECS-C++ predicates (operations that return boolean) and major changes to the structure of the generated programs. Additionally, it is not clear that this effort would lead to significant performance improvements.

2.4 The User Interface

The specification execution system can run specifications from the command line or via a graphical user interface (GUI). When running from the command line, the desired variable declarations and member function invocations are placed directly in the SPECS-C++ specification (`.h`) file. After the specification is compiled, running it will display the state resulting from the sequence of member function invocations, and the return value from the last member function invocation (if the return type of the member function is not `void`). The default is to display only the first solution, but all solutions can be generated by specifying appropriate arguments to the specification compiler.

The GUI can be used to declare variables (including those that instantiate template classes), to choose member functions to execute, to specify the actual arguments in a member function invocation, and to step through all post states that satisfy the member function specification. Figure 2 is a screen shot of the interface being used to execute the `Table` specification of Figure 1. The specification has been instantiated with `string` as the parameter type (note that strings are represented as sequences of characters), and after adding several entries, the `sortKeys` operation has been invoked.

SPECS-C++ references are displayed as arrows (i.e. pointers) in the interface using a canvas widget. This allows *aliasing* (multiple references to the same object) to be indicated by multiple arrows pointing to the same object [22]. Aliasing is a common source of errors and confusion in specifications (and programs!), so indicating aliases in this graphical fashion is extremely useful for developing specifications. In Figure 2, it is immediately apparent that each of the string objects `s1` and `s2` have been added to the table three times, and so that a great deal of aliasing is present. The interface allows allows the user to directly edit specification states, including adding and removing aliases using the mouse. New objects (targets of references) can also be added directly. If this functionality were not provided, building complex states for use in testing specifications would be much more tedious.

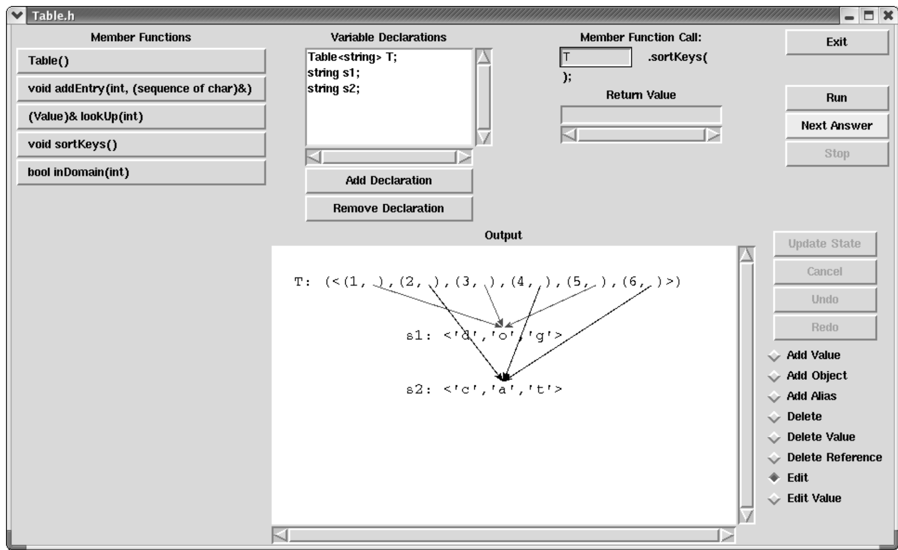

Fig. 2. The Graphical User Interface

2.5 Miscellaneous Features

Each specification is compiled to an Oz functor, and only the procedures implementing public member function specifications are exported. This enforces SPECS-C++ accessibility rules, as only these procedures are visible from client test code and the GUI.

An existentially quantified variable of type `int` is translated to an Oz finite domain variable if its domain is explicitly specified as a range of integers, i.e.:

```
\exists int x [1 <= x <= 10 /\ ...]
```

and the range is within the possible values of an Oz finite domain variable. As each such variable is found, it is added to a single instance of an Oz class. This instance is then used to *distribute* over all of these variables at at the same time (after all constraints have been seen). The distribution strategy used determines the order in which these variables are considered during search. In particular, the execution system uses the built-in first fail distribution strategy, which means that the variable with the smallest domain is considered first, then the variable with the next smallest domain and so on. This strategy often explores a much smaller search tree than a naive distribution strategy (simply considering the variables in order of appearance in the program) would. For the first fail strategy to be most effective, all finite domain variables should be distributed at the same time. Hence, it is convenient to collect all such finite domain variables within an instance of a class, as instances have state and can be updated. For similar reasons, several data structures used by the GUI are implemented as classes.

Currently, the execution system does not take advantage of *propagators* for finite domains. Propagators are operators or procedures that reduce the domains

of finite domain variables without performing search, and thus often dramatically decrease running times. For example, if the domain of variable `X` is {3, 5, 7} and of variable `Y` is {1,2,3,4,5}, then the constraint `X <: Y` (using the propagator `<:` in place of a less than comparison) immediately reduces the domain of `X` to {3} and of `Y` to {4,5}. The search tree now has 2 leaves (rather than 15), and so can be explored much more quickly. The execution system does not use these propagators because they can only be used with finite domain variables and constants, and we have found it difficult to test whether or not a variable is a finite domain variable without introducing additional problems. However, this is an important area for future work.

3 Conclusion

3.1 Future Work

Currently, the execution system does not handle inheritance in general and inheritance of specifications in particular. Additionally, the system does not check invariants – they are parsed and typechecked, but never executed. We are currently working on a version of the system for the Java Modeling Language (JML) [7] that will address these deficiencies.

We are aware that the use of **dis** statements has fallen out of favor within some parts of the Oz community, and we are investigating how the **dis** statements used in our system could be replaced by **choice** or **or** statements, or (preferably) how choice points could be removed altogether. However, because we are executing programs generated from specifications, we frequently do not know and can not control the mode of use of library operations. For example, the sequence index operation is essentially implemented in the library as:

```
proc {Index S N V}
   dis N = 1 then
         S = V | _
    [] SR N1 in S = _ | SR
         {Greater N 1}
      then
         {Plus N1 1 N}
         {Index SR N1 V}
   end
end
```

where `Plus` and `Greater` are threaded library procedures with the obvious functionality. If the index parameter `N` is known, this procedure creates no choice points, and so is much more efficient than a version using **choice**. If `N` is not known, this procedure can find all indices where the value `V` occurs in the sequence `S`, or can insert `V` at any index within `S` (via backtracking). The problem is how to achieve this combination of flexibility and efficiency without using **dis**.

We are currently attempting to reduce or eliminate the use of **dis** by taking advantage of the search that is implicit for finite domain variables. For example,

the index operation can mimic the standard `FD.element` constraint propagator in a naive manner [3], i.e.:

```
proc {Index S N V}
   thread
      local L in
         {List.length S L}
         N::1#L
         {List.nth S N V}
      end
   end
end
```

providing that the variable `N` is then included in distribution. `FD.element` can not be used directly because the elements of the list are not finite domain variables. Preliminary performance results for this approach are encouraging, but considerable work remains to be done to extend it to all library procedures that currently use **dis**.

In general, we are interested in any technique that increases the range of specifications that can be executed, or that increases the efficiency of the generated Oz programs and library code (without introducing implementation bias into specifications, of course). Specific areas for further work include making more sophisticated use of finite domain variables, incorporating finite set constraints, eliminating explicit choice points whenever possible, and improving the constraint propagation done by the library procedures. We are also interested in testing the system on a wider range of practical specifications in order to determine what kinds of additional performance improvements would be most beneficial.

3.2 Summary

We have developed a system that allows many formal specifications to be executed directly. Our system can execute specifications that are written at a high level of abstraction, so that executability does not compromise other uses (documentation, proof, etc.) of specifications. The generated programs and library procedures take advantage of many features of Oz. Threads, constraint propagation, computation spaces and search are used heavily in finding post states that satisfy specifications. The graphical user interface capabilities of Oz are critical for enabling users to freely interact with specifications, and for displaying references and aliasing directly. We have found Oz to be nearly ideal as a translation target language for formal specifications.

References

1. Jones, C.B.: Systematic Software Development Using VDM. Second edn. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J. (1990)

[3] Thanks to Christian Schulte for suggesting this idea at the conference.

2. Fitzgerald, J.S., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press (1998) ISBN 0521623480.
3. Spivey, J.M.: An introduction to Z and formal specifications. Software Engineering Journal **4** (1989) 40 – 50
4. Davies, J., Woodcock, J.C.P.: Using Z: Specification, Refinement and Proof. International Series in Computer Science. Prentice Hall (1996)
5. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996) ISBN 0 521 49619 5.
6. B-Core(UK) Ltd: B-Core website (2004) http://www.B-core.com/.
7. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota, ACM (2000) 105–106
8. Wahls, T., Leavens, G.T.: Formal semantics of an algorithm for translating model-based specifications to concurrent constraint programs. In: Proceedings of the 16th ACM Symposium on Applied Computing, Las Vega, Nevada (2001) 567 – 575
9. Wahls, T., Leavens, G.T., Baker, A.L.: Executing formal specifications with concurrent constraint programming. The Automated Software Engineering Journal **7** (2000)
10. West, M.M., Eaglestone, B.M.: Software development: Two approaches to animation of Z specifications using Prolog. IEE/BCS Software Engineering Journal **7** (1992) 264–276
11. Fuchs, N.: Specifications are (preferably) executable. Software Engineering Journal **7** (1992) 323 – 334
12. Gray, J.G., Schach, S.R.: Constraint animation using an object-oriented declarative language. In: Proceedings of the 38th Annual ACM SE Conference, Clemson, SC (2000) 1 – 10
13. O'Neill, G.: Automatic translation of VDM specifications into Standard ML programs. The Computer Journal **35** (1992) 623–624
14. Elmstrøm, R., Larsen, P.G., Lassen, P.B.: The IFAD VDM-SL toolbox: A practical approach to formal specifications. ACM Sigplan Notices **29** (1994) 77 – 80
15. Fröhlich, B.: Program Generation Based on Implicit Definitions in a VDM-like Language. PhD thesis, Technical University of Graz (1998)
16. Jackson, D., Damon, C.: Semi-executable specifications. Technical Report CMU-CS-95-216, School of Computer Science, Carnegie Mellon University (1995)
17. Breuer, P.T., Bowen, J.P.: Towards correct executable semantics for Z. In Bowen, J.P., Hall, J.A., eds.: Z User Workshop, Cambridge 1994. Workshops in Computing, Springer-Verlag (1994)
18. Grieskamp, W.: A computation model for Z based on concurrent constraint resolution. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: ZB 2000: Formal Specification and Development in Z and B, First International Conference of Z and B Users. Volume 1878 of Lecture Notes in Computer Science., York, UK, Springer-Verlag (2000) 414 – 432
19. Mehl, M., Müller, T., Popov, K., Scheidhauer, R., Schulte, C.: DFKI Oz User's Manual. Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI) and Universität des Saarlandes, Postfach 15 11 50, D-66041 Saarbrücken, Germany. (1998)
20. Mozart Consortium: Mozart Programming System website (2004) http://www.mozart-oz.org.
21. Van Roy, P., Haridi, S.: Concepts, Techniques and Models of Computer Programming. The MIT Press, Cambridge, Massachusetts (2004)

22. Wu, D., Cheng, Y., Wahls, T.: A graphical user interface for executing formal specifications. The Journal of Computing in Small Colleges **17** (2002) 79 – 86
23. Marriott, K., Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press, Cambridge, Massachusetts (1998)
24. Struyf, J., Blockeel, H.: Query optimization in inductive logic programming by reordering literals. In Horváth, T., Yamamoto, A., eds.: Proceedings of the 13th International Conference on Inductive Logic Programming. Volume 2835 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2003) 329 – 346
25. Overton, D., Somogyi, Z., Stuckey, P.J.: Constraint-based mode analysis of Mercury. In: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, Pittsburgh, PA, USA, ACM Press (2002) 109 – 120