Extensions to Proof Planning for Generating Implied Constraints

Alan M. Frisch, Ian Miguel and Toby Walsh

University of York, York, England. frisch,ianm,tw@cs.york.ac.uk

Abstract. We describe how proof planning is being extended to generate implied algebraic constraints. This inference problem introduces a number of challenging problems like deciding a termination condition and evaluating constraint utility. We have implemented a number of methods for reasoning about algebraic constraints. For example, the `eliminate` method performs Gaussian-like elimination of variables and terms. We are also re-using proof methods from the PRESS equation solving system like (variable) isolation.

1 Introduction

Users of computer algebra systems typically have well-defined goals. For example, they might wish to find the solutions to some algebraic equations, or factorize a polynomial. We are interested in an inference problem about algebraic constraints which is less well-defined. We wish to infer some implied constraints that will help a constraint solver. It is difficult to know how many implied constraints to infer, and which will be useful to a particular constraint solver. To tackle this problem, we are using Bundy's proof planning framework [Bun91]. This is one of the most promising inference techniques for dealing with combinatorially explosive search problems. Proof plans are built by methods which come with strong preconditions to limit their applicability. Some of the proof methods we are developing are extensions of those used by the PRESS equation solving system [BW81]. Others of the proof methods are novel, and perform tasks like eliminating variables and linearizing constraints.

The paper is structured as follows. In Sections 2 and 3, we introduce proof planning, and describe how it has been extended to deal with generating implied constraints. In Section 4, we describe the proof methods currently implemented that infer implied constraints. In Section 5, we illustrate their behaviour on two examples taken from the literature. In Section 6, we describe related work. We end in Section 7 with future work and conclusions.

2 Proof Planning

Proof planning is a technique used for guiding the search for a proof in automated theorem proving [Bun91]. Common patterns in proofs are identified and encapsulated in *methods* which are made available to a planner. Methods have strong preconditions which limit their applicability and prevent combinatorially explosive search. Proof planning has often been associated with "rippling" [BSvH+93], a powerful heuristic for guiding search in inductive proof. However, proof planning can easily be adapted to other mathematical tasks like finding closed form sums to series [WNB92] or, as here, generating logical consequences which may make useful implied constraints.

A proof planner like CLAM [BvHHS90] takes a goal to prove, and selects a method from a database of methods which matches this goal. The proof planner checks that the pre-conditions of the method (which are a sequence of statements in a meta-logic) hold. If the pre-conditions hold then the proof planner executes the post-conditions (which are also a sequence of statements in the meta-logic). This constructs the output goal or goals. Associated with each method is a tactic which applies individual inference rules to construct the actual proof. A typical method is the `induction` method, whose input is an universally quantified goal, and whose preconditions then select a suitable induction variable, and induction scheme. The output of the `induction` method are appropriate base and step cases. Proof planning terminates when all the goals have been satisfied.

Proof planning offers several potential advantages over other theorem proving techniques for the task of generating implied constraints automatically. First, methods can be given very strong preconditions to limit the generation of logical consequences to those that are likely to make useful implied constraints. Second, methods can act at a very high level. For example, they can perform complex rewriting, simplifications, and transformations. Such steps might require very long and complex proofs to justify at the level of individual inference rules. And third, the search control in proof planning is cleanly separated from the inference steps. We can therefore easily try out a variety of different search strategies like best-first search or limited discrepancy search.

3 Extensions to Proof Planning

Whilst proof planning has a number of features which make it well suited to the task of generating implied constraints, we have extended it along a number of different dimensions to deal with the following issues:

Non-monotonicity: Our proof methods transform one set of constraints into another. In some cases, they might add a new constraint. In others, they might replace one constraint by a tighter one, or eliminate a redundant constraint. The set of constraints may therefore increase or decrease. To deal with this, we replace the "output" slot in a method by the "add" and "delete" lists used in classical planning.

Pattern matching: Existing proof planners like CLAM [BvHHS90] use Prolog's (first-order) unification to match a proof method's input against the current proof goal or subgoal. We use a richer pattern matching language specialized to the task of reasoning about sets of constraints. For example, the input to a proof method is a *set* of constraints, and this is matched against any *subset* of the initial or inferred constraints.

Looping: Unless a proof method deletes one (or more) if the input constraints, the preconditions of the method will typically continue to hold. Proof methods may therefore repeatedly fire, generating identical implied constraints. We therefore added an history mechanism to the proof planner to prevent such repeated method application.

Termination: Previously, proof planning had a clear termination condition. We reduced a goal to subgoals, and when all these had been proven, we finished. It is much less clear when to terminate when using proof planning to infer implied constraints. There are many logical consequences (including the solution to the problem) which could be inferred. At some point, we must decide to stop inferring new constraints and start searching for an answer. At present, our methods have strong enough preconditions that we can run them till exhaustion. However, we may in the future have to add an executive along the lines of Ireland's proof critics [Ire92] which terminates proof planning when future rewards look poor.

Constraint utility: Deciding which implied constraints will be useful to a constraint solver is also very difficult. The proof planner uses measures like constraint arity and tightness to eliminate implied constraints which are obviously useless. However, it remains difficult for the proof planner to decide which of the remaining constraints to keep. We are therefore inventing some heuristics to decide which of the inferred constraints to give to the constraint solver.

Explanation: Methods do not explain what they do. In order for the user to see how an implied constraint was generated, we adapted the tactic mechanism already used within proof planning. Tactics now write out text explaining the application of the methods.

4 Methods

We will illustrate the implemented methods used by our proof planner by means of the following example. This is taken from the implied constraint section of the Oz finite domain constraint programming tutorial[1]. We wish to find 9 distinct non-zero digits, A to I, which satisfy the constraint:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \tag{1}$$

Note that BC is shorthand for $10 * B + C$, EF for $10 * E + F$ and HI for $10 * H + I$.

4.1 Symmetry method

The first method to fire is often the `symmetry` method. The preconditions to the `symmetry` method identify variables or terms which are indistinguishable. In the

[1] `http://www.mozart-oz.org/documentation/fdt/node31.html#section.propagators.fractions`

former case, if swapping the variable x for the variable y (and vice versa) gives the same set of constraints, then x and y are indistinguishable. In the latter case, pairs of variables within two terms are swapped and the same check is made.

To break this symmetry, the `symmetry` method adds an ordering constraint that puts an order on the indistinguishable variables or terms. In the above case, it adds the constraints that $x \leq y$. Note that this is not an *implied* constraint since it does not follow from the initial model. However, symmetry breaking constraints are very useful both for reducing search, and, as we show in the next section, for generating other implied constraints using the `eliminate` method.

Identifying symmetries, especially of non-atomic terms is potentially expensive. We are therefore developing heuristics to identify terms for comparison that are likely to be symmetrical. These heuristics are based primarily on *structural equivalence*. Two terms are said to be structurally equivalent if they are identical when explicit variable names in each are replaced with a common indistinguishable 'marker'. For example,

$$\frac{A}{BC}, \frac{D}{EF}$$

become:

$$\frac{*}{**}, \frac{*}{**}$$

and are therefore structurally equivalent. Each pair of variables, A and D, B and E, and C and F are swapped throughout the problem definition and the indistinguishability test is made.

The `symmetry` method applied to the initial problem definition of the fractions puzzle identifies the fact that the three fractions are indistinguishable and breaks the symmetry by adding the constraints:

$$\frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI} \tag{2}$$

We are currently investigating methods for identifying and breaking other forms of symmetry like rotations and reflections.

4.2 Eliminate method

The next method to fire is often the `eliminate` method. This uses symmetry breaking constraints, as well as other equations and inequalities, to perform Gaussian-like variable elimination. The preconditions to the `eliminate` method identify variables or terms which can be eliminated from a non-linear constraint. This gives an implied constraint of smaller arity than the original non-linear constraint. As constraint solvers typically delay non-linear constraints until their variables are ground, the `eliminate` method generates an implied constraint which may be used by a constraint solver at an earlier point in its search.

For example, in the fractions puzzle, `eliminate` can be used to simplify equation (1) by eliminating first $\frac{G}{HI}$ and then $\frac{D}{EF}$ in favour of $\frac{A}{BC}$ as follows:

$$\frac{A}{BC} + 2\frac{D}{EF} \leq 1 \tag{3}$$

$$3\frac{A}{BC} \leq 1 \tag{4}$$

Similarly, by eliminating first $\frac{A}{BC}$ and then $\frac{D}{EF}$ in favour of $\frac{G}{HI}$ produces:

$$2\frac{D}{EF} + \frac{G}{HI} \geq 1 \tag{5}$$

$$3\frac{G}{HI} \geq 1 \tag{6}$$

Equations (4) and (6) are both ternary, as opposed to the original arity 9 constraint and are therefore much more likely to be useful for pruning earlier in the search.

The `eliminate` method uses both equations and inequalities to rewrite constraints. When rewriting with inequalities, it computes the polarity of the terms being rewritten based on the monotonicity properties of the algebraic operators [Sch99]. For instance, addition is monotonic in both of its arguments as replacing either argument with a larger number increases the sum. On the other hand, division is monotonic in the numerator argument but anti-monotonic in the denominator argument; replacing the numerator with a larger number increases the fraction, whilst replacing the denominator with a larger number decreases the fraction.

4.3 Linearise method

As mentioned before, constraint solvers typically delay non-linear constraints until their variables are ground. A `linearise` method therefore attempts to infer linear constraints from non-linear constraints as these will be of more use to a constraint solver. The preconditions to the `linearise` method identify non-linear constraints which can be converted to linear constraints by cross-multiplying terms. We are currently investigating other ways to linearise terms.

The `linearise` method can be applied effectively to equations (4) and (6) to produce:

$$3A \leq 10B + C \tag{7}$$

$$3G \geq 10H + I \tag{8}$$

Bounds consistency can now be employed to start pruning values even before search starts.

4.4 All-different method

In many problems, we have a constraint that certain variables must take distinct values. For example, the times of classes assigned to a particular teacher must all be different from each other. A specialized `all-different` method reasons about constraints containing variables which take distinct values. The preconditions to the `all-different` method identify variables in a summation or product constraint which take distinct values. The method then computes upper and lower bounds based upon the variables taking distinct values. We are again looking at extending the method to other types of constraints.

The `all-different` method produces the following when applied to equations (7) and (8):

$$12 \leq 10B + C \leq 98 \tag{9}$$

$$12 \leq 10H + I \leq 98 \tag{10}$$

A further application of `eliminate` gives a tight lower bound for G:

$$G \geq 4 \tag{11}$$

Note that, on this occasion, bounds consistency establishes the same lower bound on G using equation (8). In general, however, when more all-different variables are involved, this method can be expected to do significantly better than naive bounds consistency.

4.5 Introduce method

The `introduce` method is complimentary to the `eliminate` method as it introduces a new variable into a constraint. The preconditions to the `introduce` method identify a non-atomic subterm common to two (or more) constraints. It then introduces a new variable for this common subterm. As the introduced variable is common to two (or more) constraints, this tightens the constraint graph. This can lead to increased propagation within a constraint solver. For example, in the Golomb rulers problems, Smith et al introduce auxiliary variables for the inter-tick distances as these were common to a large number of constraints [SSW00]. We are also considering methods to introduce new variables equal to the sum of certain other variables. This is a common trick for reasoning about slack or wastage. For example, introducing a new variable equal to sum of some of the existing decision variables on a circular Golomb ruler problem produces a significant reduction in search [SSW00].

5 Other methods

In many examples, we have found that we may have to do some algebraic manipulations in order to eliminate some sub-term. We currently use the `isolate` method from the PRESS equation solving system [BW81] to isolate a variable on one side of an equation or inequality. We also see uses for other methods from PRESS [BW81] like `collect` and `attract`.

In addition, some methods may produce output which could be usefully simplified before being added to the current set of constraints. A `simplify` method is therefore used to perform any possible simplification, again using methods such as `isolate` and, in the future, `collect` and `attract`. A `normalise` method is also applied to the output of other methods in order to maintain a normal form on the set of equations the proof planner is working on. This helps to avoid the same constraint being added twice (in slightly re-arranged form).

6 Results

We illustrate the behaviour of these methods on two examples taken from the literature. In addition to listing the implied constraints generated, we show the reductions in runtimes that result with the SICSTUS finite domain constraint solver.

6.1 Fractions puzzle

The proof planner takes as input a set of constraints specifying the problem. For example, the input for the fractions puzzle is given below:

```
prob('Fractions Puzzle',
     [domain(var('A'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('B'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('C'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('D'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('E'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('F'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('G'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('H'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      domain(var('I'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),
      all_different([var('A'),var('B'),var('C'),var('D'),var('E'),
                     var('F'),var('G'),var('H'),var('I')]),
      eq(var('A')/(10*var('B')+var('C')) +
         var('D')/(10*var('E')+var('F')) +
         var('G')/(10*var('H')+var('I')),
         1)]).
```

In the future, we intend to accept input in a high level constraint modelling language like OPL or ESRA [FH01]. The proof planner outputs a set of implied constraints. In addition, associated with each method is a tactic which explains how the implied constraint generated by the method is inferred. In the future, we intend to output LaTeX and HTML as well as plain ascii text. Part of the proof planner's output on the fractions puzzle is given below:

```
Using var(A)/(10*var(B)+var(C)) =< var(D)/(10*var(E)+var(F)),
we eliminate var(A)/(10*var(B)+var(C)) in favour of
var(D)/(10*var(E)+var(F)) in
var(A)/(10*var(B)+var(C))+var(D)/(10*var(E)+var(F))+
var(G)/(10*var(H)+var(I)) = 1.
This gives: 1 =< 2*(var(D)/(10*var(E)+var(F)))+
                  var(G)/(10*var(H)+var(I)).

Using var(D)/(10*var(E)+var(F)) =< var(G)/(10*var(H)+var(I)),
we eliminate var(D)/(10*var(E)+var(F)) in favour of
```

```
var(G)/(10*var(H)+var(I)) in
1 =< 2*(var(D)/(10*var(E)+var(F)))+var(G)/(10*var(H)+var(I)).
This gives: 1 =< 3*(var(G)/(10*var(H)+var(I))).

Linearising 1 =< 3*(var(G)/(10*var(H)+var(I)))
Gives: 10*var(H)+var(I) =< 3*var(G)

Since we know that the variables in 10*var(H)+var(I)
are all-different, the lower bound of this summation is 12,
and the upper bound 98.

Using 12 =< 10*var(H)+var(I),
we eliminate 10*var(H)+var(I) in favour of 12 in
10*var(H)+var(I) =< 3*var(G).
This gives: 4 =< var(G).
```

The proof planner generates some 22 implied constraints in total in this manner. We are currently developing heuristics to prune this set. One of the simplest heuristic is the constraint arity. If we delete all implied constraints of arity 4 or greater, we get the following five implied constraints to add to the problem definition:

$$\begin{aligned}
\frac{A}{BC} &\leq \frac{D}{EF} \\
\frac{D}{EF} &\leq \frac{G}{HI} \\
G &\geq 4 \\
3G &\geq 10H + I \\
3A &\leq 10B + C
\end{aligned}$$

Table 1 presents the results obtained when solving the fractions problem with the SICSTUS finite domain constraint library. The first column gives the results using the basic model only, the second shows the result of adding the small set of implied constraints described above, and the third shows the results of adding all the implied constraints generated by the proof planner. In both cases, the implied constraints provide a significant reduction in search.

Although the 22 implied constraint model takes longer to solve than the 5 implied constraint model, it also significantly reduces the size of the search tree. This suggests that there is a mid-point between the 2 which out-performs them both. In [SS] Schulte and Smolka report that the adding the symmetry breaking and implied constraints to their model of the fractions puzzle reduces the size of Oz's search tree by one order of magnitude.

	Basic Model	Basic Model+ 5 Implied Constraints	Basic Model+ 22 Implied Constraints
Backtracks (1st solution)	3203	2689	1529
Time Taken(ms) (1st solution)	1450	1280	2460
Backtracks (all solutions)	13359	3556	2059
Time Taken(ms) (all solutions)	5470	1690	3310

Table 1. The Fractions Problem: Results

6.2 Professor Smart's Safe

This example problem is also taken from the Oz finite domain constraint programming tutorial[2]. The code of Professor Smart's safe is a sequence of 9 non-zero digits $x_1, ..., x_9$ such that the following constraints are satisfied:

$$x_4 - x_6 = x_7 \tag{12}$$

$$x_1 x_2 x_3 = x_8 + x_9 \tag{13}$$

$$x_2 + x_3 + x_6 < x_8 \tag{14}$$

$$x_9 < x_8 \tag{15}$$

$$x_i \neq i \tag{16}$$

$$\text{all-different}(x_1, ..., x_9) \tag{17}$$

The proof planner produces the following set of constraints:

$$2x_9 < x_1 x_2 x_3 < 2x_8$$
$$6 \leq x_1 x_2 x_3 \leq 504$$
$$6 \leq x_2 + x_3 + x_6 \leq 24$$
$$3 \leq x_6 + x_7 \leq 17$$
$$3 \leq x_8 + x_9 \leq 17$$
$$6 < x_8$$
$$3 < x_4$$

Table 2 presents the results of comparing the basic model and the basic model with implied constraints. Run-times are negligible and so are not reported. Even though this problem in its basic state is very easy to solve, the addition of the implied constraints still gives some improvement. It is also important to note that the addition of implied constraints to this easy problem does not *degrade* the performance of the solver. For example, the implied constraints reduce the problem of deciding a value for variable x_8 to choosing either the value 7 or 9.

[2] `http://www.mozart-oz.org/documentation/fdt/node19.html#section.problem.safe`

	Basic Model	Basic Model + Implied Constraints
Backtracks (1st solution)	5	5
Backtracks (all solutions)	20	17

Table 2. Professor Smart's Safe: Results

7 Related Work

Proof planning has been implemented in a number of systems. The CLAM proof planner developed in Edinburgh controls search in the Oyster proof checker [BvHHS90]. CLAM has also been linked to the HOL theorem prover [BSBG98]. Whilst much of the development of CLAM has been for inductive proof [BSvH+93], several other domains have been explored including finding closed form sums to series [WNB92]. Prior to CLAM, the PRESS system used a meta-level representation of proof methods to solve algebraic equations [BW81]. Despite the lack of a planner to put its methods together, PRESS was competitive with computer algebra systems of its era. The Ω system developed in Saarbrücken also implements proof planning, but in this case for a a higher order natural deduction style logic [HKK+94]. Recently, the third author has built a simple proof planning shell, CLAM-Lite, on top of the Maple computer algebra system [Wal00]. This system allows us to explore how proof and computation can be mixed together. It can, for example, find a closed form sum to a series, and prove by induction that the answer is correct.

As exemplified by [SSW99], several recent studies show that implied constraints added by hand to a problem representation can lead to significant reductions in search. However, outside a highly focused domain like planning (see, for example, [EMW97]), there has been little research on how to generate such implied constraints automatically. One exception is [Fri99], which generalises resolution to multi-valued clauses in which variables can take more than just the two values *True* and *False*, and proves that implied constraints generated by the closure of this operation will eliminate search.

A number of other projects have combined tools for performing inference and algebraic reasoning. For example, the Theorema project [BJK+97] is extending the Mathematica computer algebra system with theorem proving capabilities. The system consists of a collection of special purpose provers. These include a prover for induction over the natural numbers, another for induction over lists, as well as an interface to external theorem provers. The Analytica prover [BCZ96] also adds theorem proving capabilities to the Mathematica computer algebra system. The system is able to prove some complex theorems in analysis about sums and limits, as well as some simple inductive theorems.

8 Future Work and Conclusions

We have described a new application for proof planning, the generation of implied (algebraic) constraints. This required a number of extensions to proof planning like the inclusion in methods of add and delete lists (as in classical planning). We have implemented a number of proof methods for generating implied constraints automatically including: the `symmetry` method which breaks symmetries between indistinguishable variables and terms, the `eliminate` method which eliminates variables and terms from non-linear constraints, the `introduce` method which introduces new variables, the `linearize` method which linearizes non-linear constraints, and the `all-different` method which reasons about constraints containing variables which take distinct values. In future work, we will test these methods on a larger corpus of examples. In addition, we intend to develop a number of specialized methods. For instance, in many scheduling, partitioning and cutting problems, new variables are introduced equal to sums of certain decision variables (for example, to compute the tardiness or wastage) and implied constraints inferred about these variables. We will implement methods to perform such variable introduction and inference.

Acknowledgements

This project and the second author are supported by EPSRC Grant GR/N16129[3]. The third author is supported by an EPSRC advanced research fellowship. The authors wish to thank Brahim Hnich for discussions about generating implied constraints automatically. Julian Richardson's adaptation of PRESS to manipulate inequalities and its success at generating a number of implied constraints automatically helped inspire the search for the methods presented here.

References

[BCZ96] A. Bauer, E. Clarke, and X. Zhao. Analytica: an experiment in combining theorem proving and symbolic computation. In *Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation, AISMC-3*, 1996.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the Theorema project. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC'97*, 1997.

[3] `http://www.cs.york.ac.uk/aig/projects/implied/index.html`

[BSBG98] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between Clam and HOL. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*. Springer Verlag, Lecture Notes in Computer Science, 1998.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Bun91] A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[BW81] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available as DAI Research Paper 121, Dept. Artificial Intelligence, Edinburgh.

[EMW97] M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th IJCAI*, pages 1169–1176. International Joint Conference on Artificial Intelligence, 1997.

[FH01] P. Flener and B. Hnich. Compiling high-level type constructors in constraint programming. In Ramakrishnan. I.V., editor, *Proceedings of PADL-01*, pages 229–244. Springer Verlag, 2001. LNCS 1990.

[Fri99] A.M. Frisch. Solving constraint satisfaction problems with nb-resolution. *Electronic Transactions in Artificial Intelligence*, 3, Section B:105–120, 1999.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP: A proof development environment. In Alan Bundy, editor, *Automated Deduction — CADE-12*, Proceedings of the 12th International Conference on Automated Deduction, pages 788–792, Nancy, France, 1994. Springer-Verlag, Berlin, Germany. LNAI 814.

[Ire92] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proof. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence 624*. Springer-Verlag, 1992. Also available as Research Report 592, Dept of AI, Edinburgh University.

[Sch99] W.M. Schorlemmer. Term rewriting in a logic of special relations. In *Proceedings of the Seventh International AMAST Conference, LNAI 1548*, pages 178–195. Springer Verlag, 1999.

[SS] C. Schulte and G. Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial.* http://www.mozart-oz.org/documentation/fdt/index.html.

[SSW99] B.M. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem. In *Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints*. International Joint Conference on Artificial Intelligence, 1999. Also available as APES report, APES-11-1999 from http://apes.cs.strath.ac.uk/reports/apes-11-1999.ps.gz.

[SSW00] B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the 16th National Conference on AI*, pages 182–187. American Association for Artificial Intelligence, 2000.

[Wal00] T. Walsh. Proof planning in Maple. In *Proceedings of the Workshop on Automated Deduction in the Context of Mathematics*. 17th Conference on Automated Deduction, 2000.

[WNB92] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.