Heuristics for Resolution-Based Set-Theoretic Proofs

J A van der Poll and W A Labuschagne *
*Department of Computer Science and Information Systems, University of South Africa, P.O. Box 392 Pretoria, 0003 email: {vdpolja,labuswa}@alpha.unisa.ac.za

Abstract

*A formal specification language like **Z** permits the specifier to construct proofs which corroborate the aptness of the specification. This process may be facilitated by establishing a partnership between the specifier and a suitable automated reasoner. However, **Z** specifications are essentially set-theoretic and set-theoretic proofs pose demanding challenges to automated reasoners. Taking the OTTER theorem-prover to be representative of resolution-based automated reasoners, we propose a number of heuristics intended to cut down the search space for proofs involving typical set-theoretic structures.*
Keywords: *heuristics, problem frames, resolution, set theory, theorem-proving, **Z**, ZF*
Computing Review Categories: *D.2.4, F.3.1, F.4.1, I.2.3*

1 Introduction

The use of formal methods in software construction ought no longer to be controversial. Among the benefits to be gained by using a formal specification language like **Z** (see e.g. [9], [16], and [20]) is that the specifier can prove things about the specification. The process of constructing proofs can aid in the understanding of the system and may reveal hidden assumptions, as pointed out in [22] by Woodcock and Davies:

A specification without proofs is untested: it may be inconsistent; it may describe properties that were not intended, or omit those that were; it may make inappropriate assumptions. The practice of proof makes for better specifications.

As is well known, the huge cost and inconvenience of detecting and correcting errors only after the system has been released [7], suggests that it is worth the effort to identify and correct errors at an early stage (e.g. the specification phase). However, the readiness with which the information technology industry would accept such a methodology is likely to depend on the availability of environments that ease the burden on the specifier. The more sophisticated the environment (and thus the greater its contribution to the partnership), the more natural becomes the inclusion of an automated reasoning algorithm as one component. The tasks to be performed by such an algorithm are non-trivial, as may be seen by considering the proof obligations that arise from an illustrative application of a typical formalism such as **Z**.

Basically, **Z** specifications are set-theoretic specifications. The **Z** specification languages are based on strongly typed fragments of ZF set theory [6] expressed in suitable first-order languages augmented by schema notation. The schema notation arose, according to Hoare [11], in order to separate visually the formal parts of a specification from the informal documentation around them. A schema is divided into two parts: a declaration part in which the variables to be used in the schema are declared, and a predicate part which constrains the values of the variables. Schemas may be combined, if desired, using the schema calculus (e.g. [20]).

As an example of a small system to be specified in **Z**, consider the following [16]:

A government proposes to introduce a national identity scheme for football fans. Each fan will be allocated a single, unique identity code. The system must keep track of who has been allocated which identity code. It must also keep a list of the identity codes of troublemakers who have been banned from attending matches.

The specification involves two basic types: PERSON, which is some suitable set-theoretic representation of the collection of relevant people, and ID, which is a set of identity codes devised in a manner ensuring adequacy of quantity without gross superfluity.

An abstract state of the system is given by the schema *Fid* below:

$$\begin{array}{|l} \textit{Fid} \\ \hline members : ID \rightarrowtail PERSON \\ banned : \mathbb{P}(ID) \\ \hline banned \subseteq \mathrm{dom}(members) \\ \hline \end{array}$$

The schema above conveys the following information:

- the variable *members* is a partial injective function from *ID* to *PERSON*.
- *banned* is a set of *ID* numbers.
- the set *banned* is further constrained to be a subset of the registered *members*.

Next we can define an operation to add a new member:

AddMember

$members, members' : ID \rightarrowtail PERSON$
$banned, banned' : \mathbb{P}(ID)$
$applicant? : PERSON$
$id! : ID$

$applicant? \notin \mathrm{ran}(members)$
$id! \notin \mathrm{dom}(members)$
$members' = members \cup \{ (id!, applicant?) \}$
$banned' = banned$

The schema *AddMember* above conveys the following information:

- The input is the *applicant*? applying for membership.
- The system must generate the identity code (i.e. *id*!).
- The *applicant*? must not already be a *member*.
- The identity code must not be in use already.
- The function *members* is updated with the new information.
- The set *banned* is not changed.

A number of proof obligations arise from the definition of *AddMember* above and we list some of them below:

- We need to show that the state invariant is preserved, i.e. $banned' \subseteq \mathrm{dom}(members')$ (trivial in this case).
- We need to prove that $members'$ is indeed a partial injective function.
- We need to show that a *banned* member cannot apply and be enrolled with another identity code. Therefore, we need to prove that:
  $(\forall y)(\forall z)((y, z) \in members' \wedge y \in banned \rightarrow z \neq applicant?)$
- Normally one also needs to show that an operation is *functional* (i.e. given the same input, it produces the same consistent output every time). In the case of *AddMember* however, we have to qualify this claim if we think about it in more procedural terms. The reason is that the *id*! is generated (presumably in a random fashion) by the system, and any execution of *AddMember* on the input *applicant*?, followed by a deletion of the same member, followed again by an execution of *AddMember* on the same input, *applicant*? could very well result in the system assigning different identity numbers during the first and second execution of *AddMember*.

  If we insist like in the formula below

  $(((m, b), applicant?, id!), (m', b')) \in AddMember \wedge$
  $(((m, b), applicant?, id!), (m'', b'')) \in AddMember$
  $\rightarrow ((m' = m'') \wedge (b' = b''))$

  that the two identity codes must be the same, then of course the claim holds.

From the *Football* example above it is evident that quite a number of set-theoretic constructs and operations are involved in the specification. This is generally true for **Z** specifications (see e.g. [9]).

Set-theoretic proofs pose demanding challenges to automated reasoning programs (see e.g. [2], [17], and [23]), since unlike number theory or group theory or applications to real systems such as power stations, the denotations of terms in the context of set theory are strongly hierarchical: one object (perhaps at a very fine level of granularity) is a member of another (coarser) object, which in turn may be a member of a higher-level (even coarser) object, and so on. The possibility of moving between levels is a provocation to literally endless irrelevant activity; intelligence would be realised by principles that limit the movement up or down to productive changes of granularity.

For example, in proving that

$$A \subseteq B \rightarrow \mathbb{P}(A) \subseteq \mathbb{P}(B) \qquad (1)$$

the movement from the level of elements of A up to the level of elements of $\mathbb{P}(A)$ should not be iterated to the level of elements of $\mathbb{P}(\mathbb{P}(A))$. Humans understand this intuitively and semantically - the challenge is to enunciate syntactic constraining principles.

Therefore, the resolution process can generate a large number of clauses, many of which are irrelevant to the actual proof. As such, we need a set of heuristics to counteract this combinatorial explosion (see e.g. the discussion on page 107 of [8]).

In this paper we describe a preliminary set of heuristics for finding short proofs of properties that involve typical set-theoretic constructs (e.g. (1) above).

Our heuristics are intended to facilitate the use of automated reasoning algorithms based on resolution. The relevant resolution principles are described below. The form in which results are presented derives from Jackson's work on problem frames in [12]. These frames are linked to the syntactic structure of the set-theoretic definitions appearing in the proof obligations.

2 Resolution inferences and strategies

The resolution principle was developed by John Alan Robinson in the early 1960's (see e.g. [18] and [19]) and is closely related to the proof technique of *reductio ad absurdum* (i.e. proof by contradiction).

Resolution works at the level of clauses and the essential idea is to determine whether a given set of clauses (say S) contains the empty clause (which we represent by $\Box$ below). If S contains $\Box$ then S is unsatisfiable. If S does not contain $\Box$, then the resolution mechanism attempts to derive $\Box$ using the clauses of S. The resolution principle is sound and *refutation*

complete in the sense that it can always generate the empty clause from an unsatisfiable set of clauses (see e.g. [3]).

In practice, if a formula G is believed to be provable from a set of formulae, say F, then the procedure is to add the negation of the potential theorem (i.e. $-G$) to F and attempt to derive $\Box$ (i.e. the empty clause). We present an example of this process below:

2.1 Binary resolution

Consider the example database of facts and deductive axioms in *Figure 1* from [5].

$$Parent(Adam, Cain) \quad (2)$$

$$Parent(Cain, Enoch) \quad (3)$$

$$(\forall x)(\forall y)(Parent(x, y) \rightarrow Ancestor(x, y)) \quad (4)$$

$$(\forall x)(\forall y)(\forall z)((Parent(x, y) \wedge Ancestor(y, z)) \rightarrow Ancestor(x, z)) \quad (5)$$

Figure 1. ***Initial database of rules***

Suppose we want to prove that Adam is an ancestor of Enoch, i.e.

$$Ancestor(Adam, Enoch) \quad (6)$$

The first step is to rewrite all formulae in *Figure 1* in clausal form. This step produces the clause set in *Figure 2*.

$$Parent(Adam, Cain) \quad (7)$$

$$Parent(Cain, Enoch) \quad (8)$$

$$-Parent(x, y) \vee Ancestor(x, y) \quad (9)$$

$$-Parent(x, y) \vee -Ancestor(y, z) \vee Ancestor(x, z) \quad (10)$$

Figure 2. ***Clausal form of Figure 1***

The next step is to negate (6) (i.e. the theorem we wish to prove), add such negated form to the clause set (7) to (10) above, and attempt to derive the empty clause. The negated form of (6) is simply:

$$-Ancestor(Adam, Enoch) \quad (11)$$

The final step is to search for a refutation and this is done by resolving complementary literals in the given clauses. In general, we need to unify the variables and constants appearing as arguments of terms. For example: in generating clause $[B1]$ in *Figure 3*, we resolve clause (11) with clause (10). In the process we replace x with *Adam* and z with *Enoch* respectively in (10). The variable y in (10) is then renamed to x and this gives us clause $[B1]$ as the answer. The reader is referred to some good texts, for example [3] or [24] for a thorough treatment of unification.

An example of a proof of the claim in (6), using binary resolution, is presented in *Figure 3*.

$[B1]$ $-Parent(Adam, x) \vee -Ancestor(x, Enoch)$
[Resolvent of (11) and (10)]

$[B2]$ $-Ancestor(Cain, Enoch)$
[Resolvent of $[B1]$ and (7)]

$[B3]$ $-Parent(Cain, Enoch)$
[Resolvent of $[B2]$ and (9)]

$[B4]$ $\Box$
[Resolvent of $[B3]$ and (8)]

Figure 3. ***A binary resolution proof***

The heuristics to be described relate to a subset of the resolution inference rules and strategies treated in [24], whose notation we adopt. Note however that the binary resolution rule which we used in *Figure 3*, and with which students of logic programming are familiar tends toward inefficiency, and so the emphasis in automated theorem-proving is on rules that permit several clauses to participate simultaneously (e.g. unit resulting resolution and hyperresolution). In the light of this we present below further rules of inference with an indication of the resolution process involved for the sample clause set in *Figure 2*.

2.2 Further inference rules

2.2.1 UR-resolution

Unit **R**esulting (UR) resolution is the process whereby n unit clauses are simultaneously resolved with a clause consisting of $(n + 1)$ literals, called the nucleus. The resolvent is a unit clause, i.e. a clause consisting of one literal only. The definition of UR-resolution allows for the resolvent to be either positive or negative.

A UR-resolution proof of (6) using the clause set in *Figure 2* is given in *Figure 4*.

In the next two sections we discuss hyperresolution, a generalization of UR-resolution.

2.2.2 Positive hyperresolution

The object of an application of a *positive hyperresolution* step is to produce a *positive* clause from a set of clauses, one of which contains at least one negative literal, while the remaining are positive clauses.

Formally, positive hyperresolution is that inference rule that applies to a set of clauses, one of which

$[UR1]$ $-\ Ancestor(Cain, Enoch)$
[Resolvent of (7), (10) and (11)]
$[UR2]$ $-\ Parent(Cain, Enoch)$
[Resolvent of $[UR1]$ and (9)]
$[UR3]$ $\square$
[Resolvent of $[UR2]$ and (8)]

Figure 4. ***A UR-resolution proof***

must be negative or mixed (called the nucleus), the remaining (called satellites) must be positive clauses and their number must be equal to the number of negative literals in the nucleus. Every negative literal in the nucleus is resolved with exactly one satellite. These restrictions ensure that any such resolvent will be a purely *positive* clause.

Positive hyperresolution is furthermore refutation complete for arbitrary unsatisfiable clause sets.

A positive hyperresolution proof of (6) using the clause set in *Figure 2* is given in *Figure 5*.

$[PH1]$ $Ancestor(Cain, Enoch)$
[Resolvent of (8) and (9)]
$[PH2]$ $Ancestor(Adam, Enoch)$
[Resolvent of $[PH1]$, (7) and (10)]
$[PH3]$ $\square$
[Resolvent of $[PH2]$ and (11)]

Figure 5. ***A positive hyperresolution proof***

2.2.3 Negative hyperresolution

Negative hyperresolution is like its positive counterpart, but the signs of the nucleus and satellites are reversed, ensuring that all resolvents are purely *negative* clauses. Negative hyperresolution is also refutation complete for any given unsatisfiable set of clauses.

A negative hyperresolution proof of (6) using the clause set in *Figure 2* produces the same refutation as for binary resolution in *Figure 3*.

2.2.4 Binary Paramodulation

Paramodulation is an extension of unification that allows one to make equality substitutions directly at the level of terms in a clause.

2.2.5 Factoring

Factoring is the process of reducing repetition in a clause by unifying two or more literals in the same clause.

Example

$P(a)$ is a factor (i.e. a simplification) of the needlessly repetitive clause $P(x) \vee P(a)$.

2.3 Strategies

2.3.1 Weighting

By assigning different weights to different symbols, the theorem-proving algorithm may be guided towards resolving lighter clauses before heavier, in an attempt to generate lighter resolvents that are closer to the empty clause. The default weight of a clause is the sum of the number of individual constants, variables, function symbols, and predicate symbols. Connectives like | (i.e. *or*) and $-$ (i.e. *not*) are not counted.

2.3.2 Set-of-support (sos)

The set-of-support idea basically involves the following: The set S of clauses is partitioned by taking a subset T comprising one or more clauses to be the sos, and the sos guides the evolution of the reasoning process by requiring one of the parent clauses of a resolvent to come from the sos (i.e. T) and the other parent from $S - T$. Resolvents are placed back into the sos.

Why does this work? If S is an unsatisfiable set of clauses, and if T is a subset of S such that $S - T$ is satisfiable, then taking T as the sos preserves the property of refutation completeness for the resolution mechanism (but see also below). (Typically, as in logic programming, T is taken to be the negation of the desired conclusion.)

So in practice, we normally place the negation of the theorem to be proven in the section called the *sos* and all the other formulae, which we believe will be needed to find a proof, in the section called the *usable* list. The idea is that the proof attempt should start off by using the negation of the theorem for which a proof is sought. In the case of OTTER, the *main loop* for inferring and processing clauses in search of a refutation is presented in Section 3 below.

The resolution mechanism, using the sos idea, is refutation complete using the two inference rules binary resolution and factoring, but is not refutation complete using the sos idea in conjunction with hyperresolution. We can see this in *Figure 3* and *Figure 5* respectively.

For example, in *Figure 3* it was possible to derive $\square$ starting with the negation of the theorem (i.e. clause (11)), but in *Figure 5* we had to start with two other clauses. Nevertheless, the use of a sos very often leads to a quick proof, as opposed to no real time proof at all.

3 The OTTER theorem-prover

In our work we used the **OTTER** (**O**rganized **T**echniques for **T**heorem-proving and **E**ffective **R**esearch) theorem-prover described in [14], [24], and [25]. OTTER is a resolution-based theorem-proving program for first-order logic with equality and includes the inference rules binary resolution, hyperresolution (both positive and negative versions), UR-resolution, and binary paramodulation.

OTTER can convert first-order formulae into sets of clauses, which constitute the input to the resolution algorithm. Of course, OTTER cannot accept formulae in the highly evolved notation of set theory, which is the result of introducing dozens of symbols by meta-level definitions, so the user has to rewrite set-theoretic formulae in terms of a weaker first-order language having the relevant relations and functions as predicate symbols and function symbols in its alphabet. Some other capabilities of OTTER are: forward and backward subsumption, factoring, and weighting. OTTER was written and is distributed by William McCune at the Argonne National Laboratory. OTTER is coded in C, is free software, and is portable to many different kinds of platforms. The version of OTTER which we used in our work is version 3.0.5, which was released in February 1998. [1]

An OTTER program is divided into several sections, each such section made up of first-order formulae or clauses (an exception is the section containing the optional demodulators which must be in clausal form already). The most important sections are the *usable list* and the *set-of-support (sos) list*.

OTTER's main loop for inferring and processing clauses in search of a refutation, as presented in [14], is shown below:

```
While (sos is not empty and no refutation
       has been found)
 1. Let given_clause be the 'lightest clause'
    in sos;
 2. Move given_clause from sos to the usable
    list;
 3. Infer and process new clauses using the
       inference rules in effect; each new
       clause has the given_clause as one of
       its parents and members of usable as
       its other parents; new clauses that
       pass the retention tests are appended
       to sos;
End of While loop.
```

The process above is repeated until a refutation is found, or the sos becomes empty or the user interrupts the proof attempt. (Note: The reason why the sos might become empty can be seen by studying the above algorithm. Sometimes none of the clauses in the sos can resolve with any of those in the usable list. Subsequently in step 2, each such clause is then systematically removed from the sos and appended to the usable list. If this situation prevails with every execution of the above while loop, then the sos will eventually become empty.)

In the proofs reported on below, we placed the hypothesis into the usable list and the negation of the conclusion into the sos. There is one exception, and this is the situation described in the *sos enlargement frame* (see Section 5.10 below).

In the next section we digress briefly in order to discuss problem frames.

[1] The latest version of OTTER is available at the world-wide web address: http://www.mcs.anl.gov/AR/otter/#doc.

4 Problem frames

Jackson mentions in [12] the approach taken by the ancient Greek mathematicians who separated the study of problems from the related study of solutions and solution methods.

A problem has an architecture given by its *principal parts* and *solution task*. Polya [15] and Jackson [12] both discuss the differences between, for instance, *problems to find* or *construct*, such as

> given lengths a, b and c, construct a triangle whose sides are of those lengths.

and *problems to prove*, such as

> prove for any sets A and B that, if $A \subseteq B$, then $\mathbb{P}(A) \subseteq \mathbb{P}(B)$.

in terms of such an architecture. A 'problem to prove' typically involves a sentence of the form **if** p **then** q, having as principal parts the hypothesis (i.e. p) and the conclusion (i.e. q). The solution task is to show that the conclusion follows from the hypothesis. (All the problems that we will consider are of the 'problem to prove' type.)

The essence of the *problem frame* approach is to concentrate on the architecture of the problem instead of immediately concentrating on the solution. Having established the architecture of a problem by determining the principal parts and solution task, one fits a problem frame, or template, for which there is useful information, or even a known method of solution, onto the current problem. The useful information associated with such a problem frame typically consists of an appropriate set of heuristics which have been found generally to facilitate the solution task. (For example, the heuristics might suggest conditions under which contraposition is a good way to prove **if** p **then** q).

We shall introduce variations on the *problem to prove* frame by presenting a number of specialisations in the context of set-theoretic proofs from [6]. All such specialisations are *partial* problem frames, since any num-

ber of such different frames may fit different parts of a particular OTTER program and any particular frame might fit only part of such a program. Our problem frames have the following structure:

- *Problem frame name*: This is the name or description of the specialisation.
- *Principal parts*: These are the various logical or set-theoretic structures which constitute the architecture of the problem.
- *Example*: The set-theoretic example which led to the definition of this particular frame.
- *Heuristic*: A list of the heuristic rules that have been found helpful in problems that fit this particular frame.

The alert reader will notice that we omitted the *solution task* from our structure, but this is merely because the solution task is always the same: prove that the negation of the formula in the *sos* follows from the formulae in the *usable* list.

The default inference rules which we use in the proofs described below are:

set(hyper_res).	*positive hyperresolution*
set(ur_res).	*unit resulting resolution*
set(factor).	*factoring*
set(para_into).	*two forms of binary*
set(para_from).	*paramodulation*

There is one exception to these defaults and that is the occasional use of *negative hyperresolution* (i.e. using set(neg_hyper_res) instead of set(hyper_res)), or the use of binary resolution as described in the *inference rule selection frame* below. All other proofs were found using the default inferences above.

The specialised problem frames are presented next.

5 Specialised problem frames

In analyzing the efficiency of an algorithm one can follow a theoretical approach which involves calculating space and time complexities [1], or one can follow an empirical approach. The theoretical approach is perfectly suited to algorithms in which, say, it is possible to predict roughly how often an assignment statement inside a loop will be executed. However, in the case of automated reasoning algorithms, inefficiency is primarily caused by thrashing, in other words by the exploration of the irrelevant consequences of irrelevant information. The extent of thrashing is difficult to predict theoretically.

The study of constraint satisfaction problems has addressed the issue of modifying standard backtracking algorithms so as to eliminate or at least reduce thrashing (see [13] and [21]). However, a typical automated reasoning algorithm offers limited options for the modification of backtracking by incorporating forward-checking or lookahead procedures, so that the art of using an automated theorem-proving assistant is largely a matter of giving it enough information for its task but not too much (rather like a physician administering an addictive drug to a patient). This principle of parsimony is the major intuition underlying the heuristics presented below. From the perspective of efficiency studies, an empirical approach based on considering the practical execution times on a real machine emerges as the most feasible tactic, satisfactory in particular because the aim is to effect comparisons between OTTER programs rather than to assign absolute measures of complexity.

All the proofs reported on below were done on a Pentium II with 32MB RAM, running at a clock speed of 233MHz. The operating system was Linux 2.1.57.

5.1 Exemplification frame

- *Principal parts*: Formulae which define the contents of sets in set-theoretic list notation.
- *Example*: An example involving contents of sets for which OTTER fails to find a quick proof in the absence of the appropriate heuristic is:

  $$\mathbb{P}\{\{1\}\} = \{\emptyset, \{\{1\}\}\}.$$

  We rewrite the above identity in the form **if** p **then** q using a simpler language and making the relevant constructions explicit:

  $$A = \{1\} \wedge B = \{\{1\}\} \wedge$$
  $$C = \mathbb{P}(B) \wedge D = \{\emptyset, \{\{1\}\}\}$$
  $$\longrightarrow C = D$$

  Further decomposition is necessary, in order to define the powerset B. Finally, the formulae constituting the hypothesis are placed in the usable list and the negation of the desired conclusion is placed in the sos.

  OTTER fails to find a proof within 12 minutes despite 127000 clauses generated in the process. However, when we employ the weighting strategy in order to cut down on the amount of paramodulation among the variables in the proof (which happens because of the equality literals), then we find a proof within **0.04 seconds** and 119 clauses generated. This leads to the following heuristic for this problem frame:

- *Heuristic #1*: Use the strategy **weight(x,n)** whenever there is equality involved due to the use of set-theoretic list notation. In the examples we encountered we empirically found that a value of n, where $3 \leq n \leq 5$ generally suffices. (Note: The meaning of **weight(x,n)** is to assign all variables in the proof a weight of n. This is done to avoid too many irrelevant paramodulants being generated.)

5.2 Nested functor frame

It is often natural to use *function symbols* (i.e. *functors*) to encode information in terms, for example when describing an integer as a successor or as a sum. However, terms built up with the aid of functors are more complex, potentially leading to difficulties with unification.

- *Principal parts*: Formulae which contain nested functors, i.e. a function symbol inside the scope of another function symbol.
- *Example*: Suppose we need to prove that set-theoretic symmetric difference is associative:

  $$(A + B) + C = A + (B + C)$$ [2]

  We can attempt this proof in either of two ways:

  Nested functor approach:

  In the OTTER usable list we give the following general definition of + (predictably called *PLUS* below):

  $$(\forall A)(\forall B)(\forall x)(x \in PLUS(A,B) \leftrightarrow (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A))$$

  In the set-of-support we place the negation of:

  $$(\forall x)(x \in PLUS(PLUS(A,B),C) \leftrightarrow x \in PLUS(A,PLUS(B,C)))$$

  With this approach it takes OTTER **7 mins 53 secs** to find a proof.

  Skolem constant approach:

  Let us unfold the nested functors:

  $$D = A + B \wedge E = D + C \wedge F = B + C \wedge G = A + F \rightarrow (\forall x)(x \in E \leftrightarrow x \in G)$$

  Now it takes OTTER only **0.17 secs** to find a proof. This leads us to the next heuristic:

- *Heuristic #2*: Avoid the use of nested functor definitions.

5.3 Equality frame

- *Principal parts*: An equality formula in the set-of-support.
- *Example*: An example of an equality which is hard for OTTER to prove is:

  $$\mathbb{P}\{0,1\} = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}.$$

  As before, we use a simpler language and make the relevant constructions explicit:

  $$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge E = \{\emptyset, \{0\}, \{1\}, \{0,1\}\} \longrightarrow D = E$$

  OTTER finds no proof within 30 minutes, even with the weighting heuristic above included.

[2] The use of a '+' to denote symmetric *difference* is standard set-theoretic notation - see e.g. [6], page 32.

If we however eliminate one level of the *indirection* caused by chunking (i.e. caused by moving from the various elements to a symbol representing the collection of those elements), and specify the principal parts of this problem by

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \longrightarrow D = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$$

and the formula $D = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$ is written as

$$(\forall x)(x \in D \leftrightarrow x = \emptyset \vee x = A \vee x = B \vee x = C)$$

then OTTER finds a proof in **18 mins 3 secs**. This result suggests the following heuristic:

- *Heuristic #3*: Avoid unnecessary levels of elementhood in the formulae (e.g. by using the elements of E directly, as above).

The time of 18 mins 3 secs is still far too long. If we however perform two different proofs, one for each half of the definition of D above, and in the two proofs specify the sos as the negation of:

case 1:

$$(\forall x)(x \in D \rightarrow x = \emptyset \vee x = A \vee x = B \vee x = C)$$

case 2:

$$(\forall x)(x = \emptyset \vee x = A \vee x = B \vee x = C \rightarrow x \in D)$$

respectively, then in **case 1** above OTTER finds a proof in **1 min 8 secs** and in **case 2** above OTTER finds a proof in **2.54 secs**. These results suggest the following heuristic:

- *Heuristic #4*: Perform two separate proofs whenever the formula in the sos requires OTTER to prove the equality of two sets.

5.4 Multivariate functor frame

- *Principal parts*: Formulae involving or giving rise to functors that take *more than one variable* as argument.

  (Remark: Terms involving functors may contain both constants and variables as arguments. The more variables occur, the greater the likelihood of thrashing caused by unification of these variables with other terms. The fewer the variables and the more the constants occurring as arguments, the more limited are the possibilities of unification, since constants may not be replaced by any other values. A subtle way in which functors having multiple variables as arguments may be produced is Skolemisation. Recall that OTTER transforms the formulae of our program into clauses, and that a key step in this transformation involves the elimination of existential quantifiers. If the existential quantifiers are within the scope of universal quan-

tifiers, the existential variable is replaced not by a new Skolem constant but by a term constructed from the universally quantified variables by a new Skolem functor.)

- *Example*: There are two main kinds of example: examples of functors having arity greater than one being produced by *Skolemisation*, and examples of the programmer using functors that take more than one variable as arguments, typically because of *indirect definitions*.

  For an example of the functor problem produced by Skolemisation, we return to a subset half of a proof of a previous equality

  $$\mathbb{P}\{0,1\} \subseteq \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$$

  and consider possible ways in which one can define $D = \mathbb{P}(C)$.

  A possible (indirect) definition of D is

  $$(\forall\, x)(x \in D \leftrightarrow x \subseteq C)$$

  where the preliminary concept $\subseteq$ is first defined by:

  $$(\forall\, A)(\forall\, B) \\ (A \subseteq B \leftrightarrow (\forall\, y)(y \in A \rightarrow y \in B)) \qquad (12)$$

  It takes OTTER **19 mins 42 secs** to find a proof for **case 1** above, using this indirect definition of D.

  Suppose we try to reduce the effect of Skolemisation by eliminating one of the universally quantified variables in (12) above, and use the following definition of $\subseteq$ (C is now a constant):

  $$(\forall\, A)(A \subseteq C \leftrightarrow (\forall\, y)(y \in A \rightarrow y \in C)) \qquad (13)$$

  Now it takes OTTER **3 mins 40 secs** to find a proof for **case 1** above, using the definition of $\subseteq$ in (13) above.

  An examination of the clausal form of (12) above reveals that as a result of Skolemisation the variable y becomes $\$f1(x1, x2)$ (i.e. a functor of two variables), while in (13) y becomes $\$f1(x1)$ (i.e. a functor of only one variable). The use of (13) instead of (12) therefore limits the number of variable arguments introduced by Skolemisation and so the possibilities of irrelevant unifications.

  Recall however that OTTER previously found a proof for this same problem in **1 min 8 secs**. This was achieved with the following direct definition of D:

  $$(\forall\, x)(x \in D \leftrightarrow (\forall\, y)(y \in x \rightarrow y \in C)) \qquad (14)$$

  The variable y in (14) above clausifies into $\$f1(x1)$ (i.e. again a functor of one variable).

  Why is there a significant difference in the times for the two definitions which both Skolemise y as a function of one variable? One might be tempted to claim that it is the indirect nature of (13) which makes it slower than (14). However, if we use the following half of the definition of $\subseteq$ (The *search-guiding frame* below explains how to determine which half of a formula to use.)

  $$(\forall\, A)(A \subseteq C \rightarrow (\forall\, y)(y \in A \rightarrow y \in C)) \qquad (15)$$

  then OTTER finds a proof in just **6.44 secs**.

  In the same way using half of the direct definition of D in the following way

  $$(\forall\, x)(x \in D \rightarrow (\forall\, y)(y \in x \rightarrow y \in C)) \qquad (16)$$

  results in a proof taking **6.45 secs** which is basically the same as for the indirect definition.

  Therefore, we notice that it is not the indirection which causes the problems, but the presence of redundant clauses (which, as a result of Skolemisation, contain functors). By using (15) and (16) instead of (13) and (14) respectively, we eliminate the redundant parts of the formulae (i.e. the parts giving rise to the functor in each case). This elimination of functors may be viewed as a special case of minimising the number of variable arguments taken by a functor, leading to the following heuristic.

- *Heuristic* #5: Always attempt to give definitions of sets in which terms are as simple as possible — either not involving functors at all, or else involving functors that take at most one variable as argument. Not only does this apply to the functors explicitly appearing in the unclausified formulae, but also to the functors arising from Skolemisation. (Note: This heuristic is closely related to a heuristic concerning the half-definition of a formula, which will be discussed in the *search-guiding frame* below.)

5.5 Intermediate structure frame

This frame may be regarded as supplementary to the previous frame; it focuses on a type of situation in which multivariate functors tend to be associated with direct rather than indirect definitions.

- *Principal parts*: Formulae involving or giving rise to intermediate structures, the direct definition of which produces complex functor expressions.
- *Example* : Suppose we need to prove the following subset relationship:

  $$A \times \bigcup(B) \subseteq \bigcup\{A \times X \mid X \in B\}$$

  A possible attempt at unfolding the sophisticated set-theoretic notation is:

  $$C = \bigcup(B) \wedge D = A \times C \wedge \\ E = \{A \times X \mid X \in B\} \wedge F = \bigcup(E) \\ \rightarrow D \subseteq F$$

This is not yet sufficient; the intermediate structure E remains to be defined in more useful detail.

If we attempt the following direct definition of E

$$(\forall x)(x \in E \leftrightarrow (\exists X)(X \in B \wedge (\forall y)(\forall z)((y,z) \in x \leftrightarrow y \in A \wedge z \in X))) \quad (17)$$

then OTTER fails to find a proof. An investigation of the clausal form of definition (17) above reveals that the ordered pair (y,z) (we use the name ORD for an ordered pair below) clausifies into the form $ORD(\$f4(x,x2),\$f3(x,x2))$, which shows us that each of the simple variables in $ORD(y,z)$ is transformed into a complex term not only involving a functor of two variables but leading to functor nesting. This complicates the resolution process as pointed out in the multivariate functor frame, in the nested functor frame, and also in [17].

If, however, we define the set E above as

$$(\forall x)(x \in E \leftrightarrow (\exists X)(X \in B \wedge (\forall y)(\forall z)((y,z) \in x \leftrightarrow (y,z) \in PROD(A,X)))) \quad (18)$$

where $PROD(A,X)$ is defined by

$$(\forall X)(\forall y)(\forall z)((y,z) \in PROD(A,X) \leftrightarrow y \in A \wedge z \in X) \quad (19)$$

then OTTER finds a proof in just **0.06 secs**.

The clausal form of the ordered pair (y,z) in (19) above simply remains $ORD(y,z)$. This facilitates the unification process and allows the theorem prover rapidly to find a proof.

There is a possible further simplification to (18) above:

If we use *equality* to define the set $A \times X$ and define E as

$$(\forall x)(x \in E \leftrightarrow (\exists X)(X \in B \wedge x = PROD(A,X)))$$

together with the existing definition of $PROD$, then OTTER finds a short proof for the subset relationship $D \subseteq F$ above in just **0.02 secs**. This short proof is the result of the power of paramodulation applied to a *single* occurrence of an equality literal in the proof. In fact, the automated reasoner's bible [24] claims on page 619 that:

Paramodulation can focus directly on the appropriate term, even when that term is found deep within a literal.

Note however that with this approach we introduce equality into the proof mechanism. The equality relation needs the axiom of reflexivity as well as paramodulation in order to make the search complete (see e.g. [17]). Paramodulation in turn often generates many redundant clauses, and it would therefore be a good idea to use a weight strategy as described in *Heuristic* #1 above.

These results lead us to the following heuristic:

- *Heuristic* #6: Use an indirect definition for an intermediate structure whenever this appears less likely to produce complex functor expressions than the direct definition, and particularly when such indirect definition involves a single equality literal.

5.6 Element structure frame

Like the intermediate structure frame, this problem frame is supplementary to the multivariate functor frame and focuses on another type of situation which tends to generate complex functor expressions.

- *Principal parts*: Formulae containing a description of the structure and properties of elements of relations and functions.
- *Example*: Suppose we need to show that:

  $$F = \{(\emptyset, a), (\{\emptyset\}, b)\} \rightarrow F^{-1} = \{(a,\emptyset), (b,\{\emptyset\})\}$$

  We can define F and F^{-1} in at least two possible ways:

  First format

  Suppose we define F in the usable list as follows ($ORD(x,y)$ represents the ordered pair (x,y))

  $$(\forall x)(x \in F \leftrightarrow x = ORD(\emptyset, a) \vee x = ORD(\{\emptyset\}, b))$$

  together with the following two facts about F and F^{-1}:

  $$(\forall y)(\forall z)(ORD(y,z) \in F \leftrightarrow ORD(z,y) \in F^{-1}) \quad (20)$$

  $$(\forall x)(x \in F \rightarrow (\exists y)(\exists z)(x = ORD(y,z))) \quad (21)$$

  We also need (for without it OTTER finds no proof) the following theorem from [6]:

  $$(\forall u)(\forall v)(\forall w)(\forall x)((u,v) = (w,x) \leftrightarrow (u = w \wedge v = x)) \quad (22)$$

  In the sos we put the negation of:

  $$(\forall x)(x \in F^{-1} \rightarrow (x = ORD(a,\emptyset)) \vee (x = ORD(b,\{\emptyset\})))$$

  With these definitions OTTER finds a proof after **2 mins 7 secs**.

  Second format

  Suppose we use the following more direct definition for F

$$(\forall y)(\forall z)(ORD(y,z) \in F \leftrightarrow$$
$$(y = \emptyset \wedge z = a) \vee (y = \{\emptyset\} \wedge z = b))$$

together with definition (20) above for F^{-1}.

The formula in the sos becomes the negation of:

$$(\forall y)(\forall z)(ORD(y,z) \in F^{-1} \rightarrow$$
$$(y = a \wedge z = \emptyset) \vee (y = b \wedge z = \{\emptyset\}))$$

OTTER now finds a proof in just **0.02 secs** which is a remarkable improvement.

Analysis reveals that the first format above needs formula (22). The clausified version of this formula contains numerous occurrences of a functor (in this case the ordered pair) with 2 variables (contradicting the heuristic of the multivariate functor frame above).

The second format does not need formula (22) above, since the proof attempt is specified at the level of ordered pairs already.

These observations lead to the following heuristic:

- *Heuristic* #7: Define the elements of relations and functions directly in terms of ordered pairs or ordered n-tuples.

5.7 Redundant information frame

This problem frame largely stresses the obvious, but when the obvious is crucially important it may be no bad thing to stress it. Moreover, this frame prepares the way for the widely useful search-guiding frame.

- *Principal parts*: Formulae in the usable list which are not obviously necessary for the proof.
- *Example*: Suppose we need to prove the following (where $Fun(f)$ indicates that f is a function):

  $$[Fun(f) \wedge Fun(g) \wedge$$
  $$(\forall x)(x \in \mathrm{dom}(f) \cap \mathrm{dom}(g) \rightarrow f(x) = g(x)]$$
  $$\rightarrow Fun(f \cup g)$$

  The detailed first-order reformulation of the above makes use of the ordered pair notation and as a result of this one may reasonably be tempted again to include the following fact about ordered pairs in the usable list:

  $$(\forall u)(\forall v)(\forall w)(\forall x)$$
  $$((u,v) = (w,x) \leftrightarrow (u = w \wedge v = x))$$

  However, the inclusion of this fact prevents OTTER from finding any proof within **30 mins**. If we now simply remove the offending formula, then we quickly find a proof in just **0.53 secs**.

  As previously seen, the above characterisation of ordered pairs produces functors of 2 variables as an artifact of clausification, and the theorem-prover spends most of its time unifying these functor variables with other terms — a process which does not further the construction of a proof, since the proof has no need for this formula.

  This result suggests the following (unsurprising) heuristic.

- *Heuristic* #8: Avoid the inclusion of information that is not obviously necessary in the input to the theorem-prover.

5.8 Search-guiding frame

Heuristic #8 suggests that one should include in the input only those formulae that actually contribute to the construction of the desired proof. Any extra information can potentially lead the theorem-prover astray. However, it is in general difficult to decide whether a formula may be needed or not.

In the light of this problem we propose in this section a technique called *resolution by inspection*, which can be used to determine which part of a formula (called the 'half-definition' below) is unquestionably relevant to the construction of the desired proof, and which part is less clearly relevant and can therefore provisionally be eliminated.

- *Principal parts*: A set-of-support comprising either a simple *conditional* formula or a single *literal*, together with biconditional formulae in the usable list.
- *Example*: Consider proving the following:

  $$\bigcap(A \cup B) \subseteq \bigcap(A) \cap \bigcap(B) \qquad (23)$$

  A possible preliminary unfolding is:

  $$C = A \cup B \wedge D = \bigcap(C) \wedge$$
  $$E = \bigcap(A) \wedge F = \bigcap(B) \wedge G = E \cap F$$
  $$\rightarrow D \subseteq G$$

  We shall trace a small, initial part of the proof with the aid of the simplified first-order reformulations of some of the definitions above, specifically those of D and G.

  The question we ask in the sos is whether D is a subset of G, which in unabbreviated form is a conditional formula:

  $$(\forall x)(x \in D \rightarrow x \in G)$$

  Negated, as all goals must be, this clausifies into two clauses (where the Skolem constants arise because the negation and the universal quantifier interact to produce an existential quantifier):

  $$\$c1 \quad \in \quad D \qquad (24)$$
  $$\$c1 \quad \notin \quad G \qquad (25)$$

  An unabbreviated first-order definition of G is the biconditional formula:

  $$(\forall x)(x \in E \wedge x \in F \leftrightarrow x \in G) \qquad (26)$$

  Since OTTER needs to resolve (25) and (26), we can see by *inspection* that the literal $\$c1 \notin G$ in

(25) can resolve *only* with a literal of the form $x \in G$ in (26).

Therefore, we need just one half of the definition in (26) and the half which we need is the *only-if* direction:

$$(\forall x)(x \in E \wedge x \in F \rightarrow x \in G)$$

The same procedure can be followed for the literal $\$c1 \in D$ in (24) above, because the unabbreviated definition of D as the generalised intersection of C is a biconditional formula, and the half we need is:

$$(\forall x)(x \in D \rightarrow (\forall b)(b \in C \rightarrow x \in b))$$

A proof for (23) which does not employ this technique of half-definitions takes **8.47 secs**, while the half-definition counterpart takes only **0.11 secs**. This is about 77 times faster.

We have found this technique of determining the appropriate half-definitions of formulae via resolution by inspection to be a useful tool in cutting down on the search space. (Recall the result with the half definition of $\subseteq$, in the multivariate functor frame, where we cut down the execution time from 1 min 8 secs to 6.45 secs). In essence we *guide the search* along the path of the half-definitions.

These results lead us to the following heuristic:

- *Heuristic* #9: Generate and use half-definitions, via the technique of resolution by inspection, for biconditional formulae in the usable list whenever the sos consists of a conditional formula or a single literal.

5.9 Inference rule selection frame

As noted earlier, the default OTTER settings used were those for UR-resolution and positive hyperresolution. There are, however, occasions when these settings do not suffice, occasions that may be recognised by the sos rapidly becoming empty. (The reason why the sos might become empty during the search for a proof is explained in Section 3 above.) Often in these circumstances negative hyperresolution will succeed, and if not the option of invoking binary resolution may be exercised.

- *Principal parts*: A set-of-support consisting of a formula which, when clausified, consists either of a single negative literal or of two literals, one positive and one negative, together with a paucity of positive clauses in the usable list. Typically the sos will be observed, under the default settings, rapidly to become empty.
- *Example*: Consider again a previous problem:

$$\bigcap(A \cup B) \subseteq \bigcap(A) \cap \bigcap(B)$$

As before a possible approach is to unfold the formula thus:

$$C = A \cup B \wedge D = \bigcap(C) \wedge E = \bigcap(A) \wedge$$
$$F = \bigcap(B) \wedge G = E \cap F$$
$$\rightarrow D \subseteq G$$

Suppose we use the default inference rule positive hyperresolution and we furthermore employ the technique of half-definitions discussed in the search-guiding frame.

With this scenario OTTER simply makes the sos empty and the proof fails. Analysis reveals that positive hyperresolution is unable to produce even a single hyperresolvent. Recall that positive hyperresolution must identify a clause consisting either solely of negative literals or of a mixture of negative and positive literals as the nucleus, and must then find one electron (positive clause) for each negative literal in the nucleus. It is quite possible that the input clauses do not permit sufficient electrons to be found. Unable to apply positive hyperresolution, the theorem-prover then applies UR-resolution, a rule generally useful because it produces short clauses (single literals). OTTER manages in the example to produce three new unit clauses, using UR-resolution, before it fails because the sos becomes empty (UR-resolution is sound, but not refutation complete).

This problem is well-known: [24] mentions that (positive) hyperresolution is sometimes not refutation complete when used in conjunction with the set-of-support strategy of OTTER. (In fact, we observed this very fact in the proof attempt in *Figure 5*.)

To illustrate what goes wrong, let us examine the first few resolution steps that take place in the context of our example:

The complete clausal form of the sos (i.e. the negation of $D \subseteq G$) is:

$$\$c1 \quad \in \quad D \tag{27}$$
$$\$c1 \quad \notin \quad G \tag{28}$$

In view of the discussion of half-definitions above we know that $\$c1 \in D$ must resolve with the formula:

$$(\forall x)(x \in D \rightarrow (\forall b)(b \in C \rightarrow x \in b))$$

However, positive hyperresolution cannot produce a resolvent, since such a resolvent would include one negative literal and therefore cannot be a valid positive hyperresolvent. There is furthermore no clause in either the usable list or sos which can help to remove the offending negative literal (i.e. $b \notin C$) during this step.

Turning now to $\$c1 \notin G$ and the half-definition

$$(\forall x)(x \in E \wedge x \in F \rightarrow x \in G)$$

we observe that in this case there are no electrons (i.e. positive clauses) present, since both clauses

contain at least one negative literal and again this blocks any possible attempt at generating a positive hyperresolvent. Switching to UR-resolution now makes the sos empty and the proof fails.

However, it is possible to generate a negative hyperresolvent (i.e. a purely negative clause); changing to set(neg_hyper_res) allows OTTER to find a quick proof for $D \subseteq G$ in just **0.12 secs**.

A special case of this problem arises when the sos consists of a single negative literal only. Consider the following example:

If A is a set of functions such that for any f and g in A it is the case that either $f \subseteq g$ or $g \subseteq f$, then $\bigcup(A)$ is a function.

A first-order formulation of this statement is:

$$(\forall f)(f \in A \rightarrow Fun(f)) \wedge$$
$$(\forall f)(\forall g)(f \in A \wedge g \in A \rightarrow f \subseteq g \vee g \subseteq f) \wedge$$
$$B = \bigcup(A)$$
$$\rightarrow Fun(B)$$

The theorem posed in the sos is simply: $-Fun(B)$.

Under the default setting set(hyper_res), OTTER simply makes the sos empty and the proof attempt fails. If the inference rule is changed to negative hyperresolution (i.e. set(neg_hyper_res)), then OTTER finds a proof after only **0.07 secs**.

There is good reason why a negative resolution strategy is to be preferred when the sos contains a single negative literal.

Negative hyperresolution can use the negative literal as an electron and then needs another clause with at least one positive literal in order to produce a negative resolvent (or the empty clause). Such (mixed) clauses are normally readily available in the usable list, for instance because of the clausification of conditional formulae.

Positive hyperresolution on the other hand has to use the negative literal as a nucleus and it then needs a purely positive clause to act as an electron in order to generate a positive resolvent (or the empty clause). This is much harder to achieve, such positive clauses not being so readily available in the usable list.

These results lead us to the following heuristic, which, for simplicity, is stated in operational terms, i.e. in terms of what may be observed when using the default settings:

- *Heuristic* # 10: Use set(neg_hyper_res) whenever the combination of set(hyper_res) and set(ur_res) rapidly make the sos empty. If no rapid proof results, try binary resolution.

5.10 Sos enlargement frame

It is customary for the sos to consist of the negated goal. As we saw with the previous frame, the default positive hyperresolution setting (and even negative hyperresolution) may at times fail when the sos is small. One always has the option of switching to binary resolution in such a case. But this is not the only option, for an alternative is to enlarge the sos in a useful way. Experience suggests that when the proof is sought in the context of a restriction stating that some set is nonempty, then that restriction may usefully be added to the sos.

- *Principal parts*: A small sos and, in the usable list, the restriction that a set is nonempty.
- *Example*: Suppose we need to prove

  $$\bigcap\{C - X \mid X \in A\} \subseteq C - \bigcup(A), \text{ for } A \neq \emptyset$$

  and we place the restriction that A is nonempty (i.e. $(\exists x)(x \in A)$) into the usable list only.

  A possible unfolding is:

  $$D = \bigcup(A) \wedge E = C - D \wedge$$
  $$F = \{C - X \mid X \in A\} \wedge G = \bigcap(F)$$
  $$\rightarrow G \subseteq E$$

  The formula we put into the sos is (when negated as all goals must be):

  $$-(\forall x)(x \in G \rightarrow x \in E) \tag{29}$$

  OTTER fails to find a proof with either positive or negative hyperresolution (we always switch on the set(ur_res) rule). Let us next trace some initial steps in the proof.

  The sos in (29) clausifies into:

  $$\$c1 \in G \tag{30}$$
  $$\$c1 \notin E \tag{31}$$

  Following the discussion on half-definitions above, we notice that we need the following half-definitions for G and E respectively:

  $$(\forall x)(x \in G \rightarrow (\forall b)(b \in F \rightarrow x \in b)) \tag{32}$$

  $$(\forall x)(x \in C \wedge x \notin D \rightarrow x \in E) \tag{33}$$

  However, (30) and (32) fail to produce either a valid positive or negative hyperresolvent, since any such possible resolvent would in both cases produce a mixed clause. The same problem occurs between (31) and (33). There are also no further clauses in the usable list which can resolve this conflict.

  If we now move the fact $A \neq \emptyset$ from the usable list to the sos, then positive hyperresolution is able to start off and find a proof in just **0.02 secs**. (The fact $\$c1 \in A$ in the sos is able to produce a positive hyperresolvent from the definition of $C - X$ in the set F above and thereby start the proof attempt

off.)

This leads us to the following heuristic, stated in operational terms:

- *Heuristic* #11: Place a constraint requiring that a set be nonempty in the sos instead of the usable list, whenever neither positive nor negative hyperresolution is capable of rapidly finding a proof.

For ease of reference we summarise all the heuristics below:

6 Summary

(1) Use the strategy weight(x,n), for $n \in \{3,4,5\}$, whenever there is equality involved due to the use of set-theoretic list notation.

(2) Avoid, if possible, the use of nested functor symbols in definitions.

(3) Avoid unnecessary levels of elementhood in formulae by using the elements of sets directly.

(4) Perform two separate subset proofs whenever the sos requires one to prove the equality of two sets.

(5) Ensure that definitions involve either no functors or functors taking at most one variable as argument.

(6) Avoid complex functor expressions by using indirect definitions of internal structures.

(7) Define the elements of relations and functions directly in terms of ordered pairs or ordered n-tuples.

(8) Avoid the inclusion of any information that is not obviously necessary in the input to the theorem prover.

(9) Generate and use half-definitions, via the technique of resolution by inspection, for biconditional formulae in the usable list whenever the sos consists of a conditional formula or a single literal.

(10) Use set(neg_hyper_res) whenever the combination of set(hyper_res) and set(ur_res) rapidly makes the sos empty. If no rapid proof results, try binary resolution.

(11) Place a constraint requiring that a set be nonempty in the sos instead of the usable list, whenever neither positive nor negative hyperresolution is capable of rapidly finding a proof.

7 Future research

There are further set-theoretic operations and structures which sometimes appear in **Z** specifications. We briefly discuss some of these below and mention possible problems with each of these.

- The concept of a mathematical *bag* is sometimes used in **Z** specifications (see e.g. [9]). A bag is often called a 'multi-set', since it is like a set, but it may contain duplicate elements. It differs from a traditional list in the sense that the elements are unordered.

  Below is an example of a bag (called $bagA$) in **Z** notation:

  $$bagA = [\![a, c, b, c, a, c]\!] \tag{34}$$

  In set-theoretic notation we represent $bagA$ as:

  $$bagA = \{ (a,2), (b,1), (c,3) \} \tag{35}$$

  Definition (35) tells us that element a occurs twice, b occurs once, and c occurs three times in $bagA$.

  It might be necessary to develop additional heuristics for bags, but since a bag is basically a function, the current set of heuristics could suffice.

- Another structure which often appears in **Z** specifications is a sequence. In **Z** a sequence of elements of type X is defined in [20], page 115 as an element of $\mathrm{seq}\, X$ where,

  $$\mathrm{seq}\, X = \{ f : \mathbb{N} \pfun X \mid \mathrm{dom}(f) = 1 \,..\, \#f \} \tag{36}$$

  In **Z** the symbol $\pfun$ is used to denote a *finite* partial function. The definition above also contains the *cardinality operator* (i.e. $\#$) and this could bring about some of the problems mentioned in conjunction with cardinality below.

- One often needs to prove aspects regarding cardinality. For example, for a set A, if A is nonempty and $\#(A) = n$ and we remove an element, say x from A, then a proof obligation would be to show that $\#(A - \{x\}) = n - 1$.

  A possible first-order definition of cardinality could be:

  (1) $(\forall A)(\#(A) = 0 \leftrightarrow A = \varnothing)$

  (2) $(\forall A)(\forall n)$
  $(\#(A) = n + 1 \leftrightarrow$
  $(\exists x)(x \in A \wedge \#(A - \{x\}) = n))$

  Due to the recursive nature of definition (2) above, one would expect resolution-based theorem provers (with a *breadth-first* search strategy like OTTER) to experience some problems. An alternative definition of cardinality augmented with a set of heuristics could be called for.

- From the examples given in this paper it is evident that a resolution-based theorem prover experiences problems in calculating an answer in the case of a concrete example. This problem is aggravated when the underlying definition is highly declarative and 'non-deterministic' (e.g. the definition of a powerset). An example of such a calculation which OTTER has difficulty with is:

  $\mathbb{P}(\{0,1,2\}) =$
  $\{ \varnothing, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{1,2,3\} \}$

In all fairness we should note that the resolution mechanism is probably not intended for a problem with the complexity like the one above. OTTER however has an *equational programming mode*, resembling the style and algorithm used in logic programming languages like Prolog (see e.g. [4]) or Gödel, described in [10]).

It is possible that the equational programming mode of OTTER could provide a neat solution to the 'powerset-problem' above.

- OTTER also has difficulty in constructing a so-called 'existence proof', i.e. to find and show that a value satisfies some set-theoretic relationship. An example of such a problem from [6] is:

  > Give an example of sets a and B such that $a \in B$, but $\mathbb{P}(a) \notin \mathbb{P}(B)$.

  However, these sort of problems often require a considerable amount of human ingenuity [3] and furthermore falls in the class of *problems to find* or *construct* as pointed out in the discussion of Jackson's problem frames in Section 4 above. The technique of answer variables described in [14] could however be useful in this regard.

  Quaife [17, p. 82 - 83] gives an example of a proof in which OTTER derived the formula for calculating the next prime number after a given one, but it is presumably still left to the user to substitute actual values in the formula and perform the calculation.

8 Acknowledgement and final remark

The authors express their gratitude to the referees who made valuable suggestions regarding this article. Finally, readers who would like to see some of the detailed proofs mentioned in this paper are welcome to contact the authors at the above addresses.

References

1. S. Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 2nd edition, 1988.
2. R. Boyer et al. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2(3):287 – 327, 1986.
3. C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.
4. W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 4th edition, 1994.
5. C.J. Date. *An Introduction to Database Systems.* The System Programming Series. Addison-Wesley, 6th edition, 1995.
6. H.B. Enderton. *Elements of Set Theory.* Academic Press, Inc., 1977.
7. M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182 – 211, 1976.
8. M. Fitting. *First-order Logic and Automated Theorem Proving.* Graduate Texts in Computer Science. Springer, 2nd edition, 1996.
9. I. Hayes. *Specification Case Studies.* Prentice Hall, 2nd edition, 1993.
10. P. Hill and J. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.
11. C.A.R. Hoare. Preface. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and **Z** - Formal Methods in Software Development*, number 428 in LNCS, 1990.
12. M. Jackson. Problems, methods and specialization. *Software Engineering Journal*, 9(6):249 – 255, November 1994.
13. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99 – 118, 1977.
14. W.W. McCune. *OTTER 3.0 Reference Manual and Guide.* Argonne National Laboratory, August 1995. ANL-94/6.
15. G. Polya. *How To Solve It.* Princeton University Press, 2nd edition, 1973.
16. B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and **Z**.* Prentice Hall, 2nd edition, 1996.
17. A. Quaife. *Automated Development of Fundamental Mathematical Theories.* Kluwer Academic Publishers, 1992.
18. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23 – 41, January 1965.
19. J.A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227 – 234, 1965.
20. J.M. Spivey. *The **Z** Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.
21. P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
22. J.C.P. Woodcock and J Davies. *Using **Z**: Specification, Refinement, and Proof.* Prentice Hall, 1996.
23. L. Wos. Research problem #8: An inference rule for set theory. In *Automated Reasoning: 33 Basic Research problems*, pages 137 – 138. Prentice-Hall, 1988.
24. L. Wos and R. Overbeek and E. Lusk and J. Boyle. *Automated Reasoning: Introduction and Applications.* McGraw-Hill, 2nd edition, 1992.
25. L. Wos. Programs that offer fast, flawless, logical reasoning. *Communications of the ACM*, 41(6):87 – 95, 1998.

[3] An example of such sets could be $a = \{1\}$ and $B = \{\{1\}, \{2\}\}$.