Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Conjecture Synthesis for Inductive Theories

Moa Johansson · Lucas Dixon · Alan Bundy

Draft: February 21, 2010

Abstract We have developed a program for inductive theory formation, called *IsaCoSy*, which synthesises conjectures 'bottom-up' from the available constants and free variables. The synthesis process is made tractable by only generating irreducible terms, which are then filtered through counter-example checking and passed to the automatic inductive prover IsaPlanner. The main technical contribution is the presentation of a constraint mechanism for synthesis. As theorems are discovered, this generates additional constraints on the synthesis process.

We evaluate IsaCoSy as a tool for automatically generating the background theories one would expect in a mature proof assistant, such as the Isabelle system. The results show that IsaCoSy produces most, and sometimes all, of the theorems in the Isabelle libraries. The number of additional un-interesting theorems are small enough to be easily pruned by hand.

Keywords Theory Formation · Induction · Synthesis · Theorem Proving · Lemma Discovery

1 Introduction

Discovering unknown theorems and lemmas is a major challenge for automated inductive theorem proving. It has generally been assumed that such discovery requires user intervention. Consequently, most theorem provers rely on the user to supply any additional lemmas that might be needed for a proof. Interactive theorem provers, such as Isabelle [21,20], have a large theory library of formalised mathematics. Such libraries require significant expertise and time to develop. Automating the formation of theory libraries, even just the construction and proof of background lemmas, is an important

Moa Johansson
Università degli Studi di Verona, Dipartimento di Informatica, Strada le Grazie 15, 371 34, Verona, Italy. Tel: +39 45 802 7908
Lucas Dixon and Alan Bundy
University of Edinburgh, School of Informatics, Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, UK.
Tel: +44 131 651 3077. Fax: +44 131 651 1426
E-mail: moakristin.johansson@univr.it, {l.dixon, a.bundy}@ed.ac.uk

challenge. It could help both the development of new theories as well as speed up the formalisation of existing ones.

Given an inductive theory, formed by the definition of recursive datatypes and functions, we show how to automatically synthesise a useful set of theorems. These theorems typically capture the basic properties of the theory and are intended to be useful in further proofs, either by a human or by an automated theorem prover. The set of synthesised theorems can also provide a 'sanity check', ensuring that the theory has been appropriately defined (or axiomatised) by ensuring that no unintended theorems are included.

Our approach is implemented in a program for inductive theory formation, called IsaCoSy (**Isa**belle **Co**njecture **Sy**nthesis), built on top of the proof-planner IsaPlanner [9, 10, 8] and the proof assistant Isabelle. IsaCoSy builds terms, starting from small ones and then building incrementally larger ones, using the set of available constants and function symbols in a given theory. The key idea for making this tractable is to turn rewriting upside down: only irreducible terms, those not matched by the left hand side of any rewrite rule, are synthesised. In terms of the implementation, this restriction turns into constraints on the term-synthesis process, thus avoiding a naive and inefficient generate-and-test style procedure. The main technical contribution of this paper is a representation for the constraints and corresponding algorithms for synthesis of irreducible terms. After conjectures are synthesised, counter-example checking is used to prune out the obviously false ones. The remaining conjectures are given to IsaPlanner, which attempts to prove them automatically by induction using the *rippling* heuristic [4]. Any proved theorems are then used to generate more constraints for synthesis of further terms. An important feature of IsaCoSy is that it is designed to be generic, and may thus be applied to theories about any recursively defined datatypes in Isabelle.

We analyse the efficiency improvement provided by the constraints mechanism, as compared with a naive algorithm that perform exhaustive term synthesis. We show that our constraints machinery enables synthesis to achieve an exponential reduction in the space of conjectures. This allows the synthesis of important theorems, such as distributivity laws, that were previously too large to be found by naive synthesis. We also evaluate IsaCoSy's ability to generate 'useful' theorems by comparing the theorems synthesised by IsaCoSy with those manually formalised by the theory developers of Isabelle. We use precision and recall analysis for this comparison. Precision analysis computes the portion of extra theorems not in the library while recall analysis computes the portion of theorems shared with the library. The domains considered are the equational theorems in a formalisation of natural numbers, a theory of lists, and a theory of binary trees. On an ordinary desktop PC, the synthesis process takes several hours but finds most, and sometimes all, of the theorems in the Isabelle library. Some of the theorems not found are special cases of synthesised theorems, but are useful for configuring proof tools. A relatively small number of additional theorems are also proved. While these are not in the Isabelle library, we note that some of them are also useful for proof automation. We note the main sources of inefficiency in our approach, and highlight how these might be tackled.

Overview

In §2, we introduce the IsaPlanner proof-planner and the Isabelle proof assistant on which IsaCoSy has been developed. In §3 we present some motivating examples intro-

ducing the basic ideas of our constraints for term synthesis. We then, in §4, describe our language for expressing constraints, and in §5, a constraint generator which produces constraints from available theorems. We give examples of typical theorems from which initial constraints are generated in §6. Additional heuristics, for synthesis are described in §7. The synthesis engine itself, including procedures for updating and propagating constraints, is presented in §8.

In §9, we present a small case-study to illustrate how IsaCoSy uses its constraints and heuristics. §10 describes the evaluation and experimental methodology. We show in §10.1, that IsaCoSy manages to exponentially improve on the synthesis-space compared to a naive synthesis. In §10.2, we perform a precision and recall analysis with Isabelle's libraries. Further experiments exploring the effects of allowing constraints to be generated from unfalsified but unproved conjectures and on restricting polymorphism are discussed in §10.3 and §10.4 respectively. The limitations of IsaCoSy and further work are discussed in §11. We compare IsaCoSy to other theory-formation systems in §12, and finally summarise and draw conclusions in §13.

The source code and instructions for IsaCoSy are available on-line at:

`http://dream.inf.ed.ac.uk/projects/isaplanner/`

2 Background

2.1 An Overview of Theory Formation Systems

There are two main approaches to theory formation, *generative* and *deductive*. Theory formation following the generative approach generally produces conjectures according to some set of heuristics, and then checks which of these are theorems by counter-examples and/or proof. Our work falls into this category, together with the early theory formation system AM [15], which managed to conjecture Goldbach's conjecture. However, AM did not have the capability to *prove* any conjectures and its heuristics were domain specific. HR is a more recent generative theory-formation system [6]. It uses the resolution prover Otter to prove its conjectures and the MACE model generator to search for counter-examples. HR has been applied to domains including number theory, graph theory, and group theory. Although the number of interesting conjectures made was rather low, HR did manage to invent some novel integer sequences. Theory exploration based on *knowledge schemes* - essentially higher-order terms - has been proposed for the Theorema system [2]. The knowledge schemes capture prior mathematical knowledge, and are then instantiated with symbols in the current theory in an attempt to produce new concepts or function definitions. A preliminary case-study of the natural number has been undertaken, but the process is not yet automated [12].

Systems using a deductive approach attempt to produce new theorems as logical consequences of known facts. This approach has the advantage of not having to use counter-example checking to filter out non-theorems, but instead it has to apply other forms of filtering to avoid trivial or uninteresting logical consequences. The MATHsAiD system [17, 18] uses this approach, as does the AGInT system for first-order classical logic [22]. AGInT has been applied to axioms about set theory and about logical puzzles from the TPTP library [23], where it finds some theorems.

The IsaCoSy system, described in this paper, is the first system built for a generic higher-order proof assistant. It also implements a novel approach to theory formation

based on the simple idea of generating irreducible terms. We compare it in more detail to related work in §12.

2.2 IsaPlanner and Isabelle

IsaCoSy passes conjectures to the proof-planner IsaPlanner, which attempts to prove them automatically. Proof-planning is a technique used to guide search in automated theorem proving by exploiting the fact that there are families of proofs with a similar structure [3, 5]. One such family is proofs by induction. IsaPlanner employs the rippling heuristic to guide rewriting of the step-case in inductive proofs [4, 8].

Isabelle is a generic, interactive theorem prover which supports a wide range of object logics, such as higher-order logic (HOL), Zermelo-Fraenkel set theory and many others [20]. A large library of theorems for various object logics is available on-line at `http://isabelle.in.tum.de`. Each object logic is formalised in Isabelle's meta-logic, which is an intuitionistic higher-order logic with implication, universal quantifiers and equality [21]. Isabelle follows the LCF approach to theorem proving, where new theorems can only be obtained from previously proved statements through manipulation by a small set of trusted inference rules [11]. More complex tactics are built by combining these rules in different ways, ensuring that the resulting proofs rely only on the fixed set of trusted inference rules.

IsaPlanner is built on top of Isabelle and can also support different object logics and follows the LCF approach. It supports automated inductive proofs using an implementation of the rippling heuristic to guide search. It also has a language for combining simple reasoning techniques into new, more complex ones. The atomic techniques are declarative wrappers for Isabelle's tactics, so that proof-planning and tactic execution can be interleaved, ensuring each step is sound.

2.3 Notation

Functions and Lists: The '@'-symbol denotes the list append function and '#' denotes cons, and both of these are written infix. These and all other functions used in our examples are defined in Appendix A.

Term size: We define the size of a term to be the number of symbols (constants or variables) it contains. For example, the term '$a = 0$' is of size 3 (one variable and the two constants '0' and '=').

Free Variables: Free variables represent arbitrary values. Free variables that are allowed to be instantiated by unification are referred to as *meta-variables*. Meta-variables are written prefixed by '?', e.g. $?f$.

Holes: During synthesis, *holes* are positions in a term-tree that have not yet been fully instantiated by synthesis. Holes are implemented as Isabelle/IsaPlanner meta-variables. Holes will have various constraints associated with them, such as a specified size and restrictions on which constants and variables are allowed to occur inside them.

Naming: Both holes and constraints are identified by unique names. We will use names of the form $?h_i$ for holes and C_i for constraints.

Rewrite rules and equations: We will refer to equational theorems as rewrite rules when they can be used as a rewrite rule from left-to-right. For example, the theorem $0 + y = y$ will be called a rewrite rule, when referring to its use as: $0 + y \Rightarrow y$.

Datatypes: We use ML-notation for datatypes to describe the central data-structures in this paper. For instance, the typical datatype for lists can be written:

```
datatype 'a List =
  Empty
| Cons of 'a * 'a List
```

3 Motivating Examples

Unless we employ heuristics and constraints, the space for conjecture synthesis is intractable. The heuristic we present in this paper avoids generating conjectures that are just 'more complicated versions' of simpler ones. This is effected by the synthesis program only producing irreducible terms - terms that cannot be rewritten to something simpler by any existing rule. We capture the concept of irreducibility by a set of constraints that guide synthesis.

In this section, to illustrate the kinds of constraints used to restrict term synthesis, we shall consider a few examples involving natural numbers. These examples are instances of the kinds of constraints that can be expressed in IsaCoSy's constraint language.

Example 1: Definition of +

Addition can be defined as follows:

$$0 + y = y$$
$$Suc(x) + y = Suc(x + y)$$

The above equations can be used as rewrite rules. The first applies to any term that has 0 in the first argument position of +, while the second applies to any term that has a *Suc* in the first position (regardless of what the *Suc* is applied to). We would like any such terms to be excluded from synthesis. For the first theorem, our constraint generation algorithm produces a constraint stating that synthesis is never allowed to put a 0 in the first argument of +. Similarly, for the second theorem, it generates a constraint disallowing *Suc* to appear in the first argument of +. This ensures that no term, which can be rewritten by the definitions of addition, is ever synthesised.

Example 2: Injectivity of Suc

The fact that *Suc* is injective is expressed in Isabelle by the theorem $(Suc\, n = Suc\, m) = n = m$, which can be seen as a rewrite rule from left to right. To avoid synthesising terms to which this rewrite is applicable, we need a constraint that forbids the two arguments of = to both be instantiated to *Suc* at the same time.

Example 3: Reflexivity

Reflexivity can be expressed as the rewrite rule $(x = x) = True$. The constraint we derive from this theorem is that, in a term we have synthesised, the two arguments of = never should be the same.

Example 4: Conditional Constraints

Imagine we have a partially synthesised term, $Suc\ ?h_3 = ?h_2$. The reflexivity constraint (above) applies and disallows the left- and right-hand side of the equation from being equal. The left-hand side has been partially instantiated to $Suc\ ?h_3$. Synthesis now only needs to consider the inequality constraint *if* the right-hand side hole $?h_2$ also becomes instantiated to *Suc*.

Conditional constraints may also arise from rewrite rules. For example, if a rule has a left-hand side of the form $f(g\ 0)$, it means that *if* a synthesised term, containing an f, instantiates its argument to g, *then* 0 is not allowed to occur as the argument of g.

4 Constraint Language

Motivated by example rewrite rules, such as those in §3, we have developed a small language for expressing constraints on term synthesis. This constraint language allows us to capture the requirement that no synthesised term should be reducible by an existing rule. In particular, a set of constraints is derived for each rule in order to avoid synthesising the set of terms that match some rule's left hand side.

4.1 Representation of Constraints

The left hand side of each rewrite rule has a symbol at the top of its term tree which we will call its top-level function symbol. When a constraint is constructed, it is given a unique name, and stored in a table associated with the top-level function it was constructed from. When a particular function-symbol is used during synthesis, its associated constraints are attached to the new holes at the argument-positions of the function-symbol.

The constraint language consists of five different kinds of constraints, captured by the constructors of the following datatype:

```
datatype Constr =
  NotAllowed of Arg * ConstantName
| VarNotAllowed of Arg * VarName
| NotSimult of (Arg * ConstraintName)  list
| UnEqual of Arg list
| IfThen of Arg * (ConstantName * ConstraintName)
```

The *arguments* in constraints, the elements of type `Arg`, refer to a particular argument position of the function that the constraint is constructed from.

The first two constructors, *NotAllowed* and *VarNotAllowed*, simply state that the specified argument is not allowed to be instantiated to the specified constant or variable (see Example 1 of §3). The *NotSimult*-constraint captures dependent constraints. Several arguments may not be allowed to have a particular combination of instantiations simultaneously (see Example 2 of §3). The *UnEqual*-constraint specifies a list of arguments that are not all allowed to be instantiated to the same term (some of the arguments may have the same instantiation, but not all). An example of this is the reflexivity theorem in Example 3 of §3. Finally, the *IfThen*-constraint describes a

condition under which a constraint on a future hole should be considered. If the argument of the *IfThen*-constraint is instantiated to the specified constant, the resulting new holes will have to adhere to the named constraint (Example 4 of §3).

The *IfThen* and *NotSimult* constraints refer to other constraints, which we will call their *children*, or *sub-constraints*. Conversely, if a constraint, C_1, is a child of another constraint, C_0, the constraint C_0 is said to be the parent constraint of C_1. An invariant on the stored constraints is that no *NotSimult* constraint has another *NotSimult* constraint as a child - if this occurs we merge the *NotSimult* constraints into one larger *NotSimult* constraint.

In §5.3 we present the important properties of our constraint langauge and algrotihms. This clarifies that the constraint language provides exactly the constraints to prevent synthesis of reducible terms.

The constraint language is not designed to capture constraints from rules with side-conditions and rules containing lambda expressions, these kinds of rules would require additions to the constraint language. Correspondingly, we have not tried to use our synthesis algorithm to synthesise conjectures with side conditions, or conjectures containing lambda-expressions. Monroy-Borja has studied techniques for finding the correct side-condition to make an inductive conjecture true [19], but extending this technique to synthesis is left as further work. Allowing lambda-expressions in synthesised conjectures is equivalent to allowing the synthesis of new functions, which is a difficult problem and would also increase the size of the synthesis space.

4.2 Representation of Argument Positions

Each constraint refers to the argument positions of some function. These positions are represented differently at different stages of the synthesis process by the constructors of the following datatype:

```
datatype ArgPos =
  Hole of HoleName
| Path of int list
| LocalIndex of ConstraintName * int
```

The *Hole* constructor represents positions which are holes during synthesis where the *HoleName* is the name of an Isabelle/IsaPlanner meta-variable.

The *Path* constructor is only used temporarily when analysing a new theorem for constraints. It specifies a position in a term as a path from the top of the term-tree. For example, in the term $(a * b) + c$, the variable b has the path [1, 2], as it is the first (leftmost) argument of plus, and the second argument of multiplication. Variable a has path [1, 1], while the path of c is just [2].

The *LocalIndex* constructor is used to represent future constraints on some hole that does not yet exist, but may in the future. This is also how argument positions are initially represented in constraints generated from rewrite rules, which are computed and stored prior to synthesis, when no holes exist. As synthesis proceeds, positions represented using *LocalIndex* are gradually replaced by holes. The constraint name in a *LocalIndex* indicates the name of the constraint that has to be triggered in order for the *LocalIndex* to be updated to being a hole. This is either the parent-constraint of the

constraint in which the *LocalIndex* occurs[1] or, if the constraint has no parent, itself. Several new holes may be produced at the same time, so the integer-index part of a *LocalIndex* indicates which new hole is intended. We abbreviate an argument position *LocalIndex*(C_i, j) to $C_i.j$ in order to improve readability.

As an example illustrating the use of *LocalIndex*-constraints, assume we are synthesising an equality, and initially have a term with two holes:

$$?h_1 = ?h_2$$

Also suppose there are two constraints, C_1 and C_2, (from the zero-case of the definition of addition) with C_1 attached to $?h_1$:

$$\begin{aligned} C_1: &\quad \mathit{IfThen}(?h_1,\ \text{`plus'},\ C_2) \\ C_2: &\quad \mathit{NotAllowed}(\mathit{LocalIndex}(C_1, 1)\ \text{`zero'}) \end{aligned}$$

The constraints above states that if $?h_1$ is instantiated to plus, the first of the resulting new holes is not allowed to be instantiated to zero. Note that C_2 must use the *LocalIndex*-constructor, as the first argument position of plus does not yet exist as a named hole. Constraint C_1 must be triggered for such a hole to be created. This happens if h_1 is indeed instantiated to plus, resulting in the new term:

$$?h_3 + ?h_4 = ?h_2$$

The new holes are named $?h_3$ and $?h_4$, with $?h_3$ being in the first (leftmost) argument position of plus, thus instantiating constraint C_2:

$$C_2: \quad \mathit{NotAllowed}(?h_3,\ \text{`zero'})$$

5 Generating Constraints

The constraints used during synthesis are automatically inferred from equational theorems. Initial constraints are derived from the definitions of recursively defined functions, as well as from theorems about reflexivity and commutativity of equality and theorems about datatypes that Isabelle's datatype-package proves automatically. This section describes the constraint generation algorithm, and shows how it derives constraints from rewrite rules.

5.1 Constraints and Information about Functions

To initialise synthesis, we compute some relevant information about each function. This includes:

- The type of the function and each of its arguments.
- A domain for each argument position, specifying which constants are allowed to occur in that position. The domain is initially all the symbols with a matching type, and is later restricted by constraints from rewrite rules.

[1] If this constraint is a sub-constraint of a *NotSimult*, its arguments are named after the 'grandparent'-constraints, otherwise a *LocalIndex*-name might not be unique.

- A set of constraints for each of the function's argument positions, arising from the initial rewrite rules.
- Information about whether the function is known to be commutative and/or associative. This is updated as synthesis progresses, as the relevant theorems are discovered. If a function is known to be commutative, we can further restrict synthesis by imposing an order on its arguments. For example, we can require that the first argument is larger than or equal to the second in measure of size.

The above information is stored in a table indexed by the function-symbol's unique name. As synthesis proceeds, and more theorems are proved, these can be fed back into the constraint generation mechanism to produce more constraints on future synthesis attempts.

5.2 Constraint Generation Algorithm

The constraint generation algorithm infers constraints from the left-hand sides of rewrite rules. The algorithm traverses the left-hand side term, producing a set of constraints that will be attached to the top-level symbol of the left-hand side of the rewrite rule. As a running example, consider a rewrite rule based on an equation of the form: $f\ ?a\ ?a\ (g\ 0) = \ldots$. Recall that $?a$ denote a meta-variable that can be instantiated by unification.

Overview of the Algorithm

1. Traverse the term and find its left-hand side (LHS). In the example, the left-hand side of the rule is $f\ ?a\ ?a\ (g\ 0)$.
2. Create equality constraints. Positions of variables that occur several times may not be allowed to be instantiated to the same term. In the example, $f\ ?a\ ?a\ (g\ 0)$, we need to consider disallowing the first and second argument of f to be the same. To find variables, traverse the LHS top down, keeping track of the path taken. On encountering a variable, store its name and path in a table. For those variables that have more than one path, create an *UnEqual*-constraint, e.g.
   $\mathit{UnEqual}(\mathit{Path}(p_1)\ \ldots\ \mathit{Path}(p_n))$.
   For the running example, there are two occurrences of the variable $?a$, in the first and second argument positions of f. Using the *Path*-constructor from §4.2, the two occurrences of $?a$ are represented as $\mathit{Path}[1]$ and $\mathit{Path}[2]$. These are not allowed to be equal, resulting in the constraint: $\mathit{UnEqual}(\mathit{Path}[1], \mathit{Path}[2])$.
3. If the LHS term is a single constant, c, the rewrite rule is applicable whenever a term contains c. To constrain all terms to not contain c, the constant c is removed from the domain of all arguments.
4. If the LHS term is a function application, $f(x_1\ \ldots\ x_n)$, compute the constraints of its argument terms $(x_1\ \ldots\ x_n)$.
   In the running example, the LHS is a function application, so we proceed to compute the constraints for the arguments of f.
   Constraints for an argument-term x_i are computed depending on whether the argument-term is a variable, constant or function application. In the constraint, the argument is referred to by its position, and parent-constraint name, using the *LocalIndex*-constructor.

Returning to the example, assume we give the name C_0 to the top-level constraint we are constructing for f. The arguments of f will thus be in positions named $C_0.1$ (for $?a$), $C_0.2$ (for the second occurrence of $?a$) and $C_0.3$ (for g 0). Computing the constraints for the arguments now proceeds as follows, depending on the form of the argument:

- **Variable** v**:** Look up the variable name v in the table created in step 2, to check if it is involved in any *UnEqual*-constraint. If so, the argument position-type is updated to use a *LocalIndex* instead of a *Path* as we now know the name of its parent constraint.
  In the running example, both the first and second arguments of f are variables, and are involved in the *UnEqual*-constraint created in step 2. We give this constraint the name C_1 and update it to:
  $$C_1 : \quad UnEqual(C_0.1,\ C_0.2)$$
  When the algorithm terminates, all argument positions will be in *LocalIndex*-format.
- **Constant** c**:** Create a new *NotAllowed*-constraint for this argument position and constant: $NotAllowed(C_p.i,\ c)$
- **Function application** $c(y_1\ \ldots\ y_m)$**:** Recursively compute the constraints of the arguments. If the number of constraints are:
  - Greater than 1, i.e. a list $(pos_j,\ C_j)\ \ldots\ (pos_n,\ C_n)$. Create a *NotSimult*-constraint:
    $$NotSimult((pos_j,\ C_j)\ \ldots\ (pos_n,\ C_n))$$
  - Exactly 1, i.e. (pos_j, C_j). Create an *IfThen*-constraint:
    $$IfThen(C_p.i,\ c,\ C_j)$$
  - 0, i.e. all arguments are variables occurring once in the term. Create a *NotAllowed*-constraint for this argument and function-symbol:
    $$NotAllowed(C_p.i,\ c)$$

  In the example, the third argument of f is a function application g 0. To compute a constraint on this, which we shall name C_2, we first compute a constraint for the single argument of g, the constant 0. This gives the constraint $C_3 :\ NotAllowed(C_2.1,\ \text{‘0’})$. Hence we also get $C_2 :\ IfThen(C_0.3,\ \text{‘g’},\ C_3)$. This means that if the position $C_0.3$ is instantiated to g, the first argument of g is not allowed to be 0.

5. Once the constraints on argument positions are computed, we can determine the top-level constraint. As before it becomes a *NotSimult*-constraint if there are several argument-constraints. If there is only one, and that is a *NotAllowed*-constraint, this constant can simply be removed from the domain of the relevant argument position.
   In the example, the complete set of constraints for the three arguments of f are:
   $$C_1 :\ UnEqual(C_0.1,\ C_0.2)$$
   $$C_2 :\ IfThen(C_0.3,\ \text{‘g’},\ C_3)$$
   $$C_3 :\ NotAllowed(C_2.1,\ \text{‘0’})$$

This results in the top-level constraint, C_0, becoming:

$$C_0 : \ \mathit{NotSimult}((C_0.1,\ C_1),\ (C_0.2,\ C_1),\ (C_0.3, C_2))$$

This constraint stops synthesis simultaneously violating both constraints C_1 and C_2 when synthesising a term containing the function f.

6. The final step of the constraint generation algorithm is to store the constraints in the constraint-table. The top-level function symbol on the LHS is associated with the top-level constraint (in the example, $f \mapsto C_0$). This mapping will then be used when synthesising a term containing the function symbol f.

5.3 Properties of Constraint Generation

We now consider properties of the generation of constraints. This brief exposition of the main theoretical properties helps understand the high-level specification of the constraint mechanisms. We first introduce some notation and terminology:

- *Constraints*($l \rightarrow r$) is the set of constraints that are generated from a rewrite rule $l \rightarrow r$.
- A term t *satisfies* a set of constraints when t is allowed to be synthesised by the constraints. For example, the constraint *VarNotAllowed*($?h,\ v$), is satisfied when $?h$ is instantiated to anything other than v).
- A term t *violates* a set of constraint when t is in the set of terms disallowed by the constraints. For example, *VarNotAllowed*($?h,\ v$) is violated if $?h$ is instantiated to v.
- A *redex* of a rewrite rule, $l \rightarrow r$, in a term, t, is a subterm of t that matches l.

The main property of IsaCoSy's constraint language is that the constraints generated from rewrite rules will exclude exactly those terms reducible by at least one of the rewrite rules. This can be decomposed into two sub-properties.

The first property states that any term that violates a constraint (which excludes it from synthesis) is indeed reducible by the rewrite rule from which IsaCoSy constructed the constraint. Thus the constraints generated from a rule never exclude too many terms from synthesis.

Property 1 (No Over-Coverage) *Given a rule $l \rightarrow r$, if t is a term that violates Constraints($l \rightarrow r$), then there is a redex of l within t.*

The second property states the compliment to Property 1, namely that any term that is reducible by some rule will indeed violate the constraints generated from that rule. Thus synthesis will never produce reducible terms.

Property 2 (Sufficient Coverage) *Given a rule $l \rightarrow r$, if a term t contains a redex of l, then t violates Constraints($l \rightarrow r$).*

From the combination of Properties 1 and 2 the final property follows, which is the desired specification for IsaCoSy's constraint generation algorithm:

Property 3 (Exact coverage) *Given a rule $l \rightarrow r$ and a term t, t violates Constraints($l \rightarrow r$) iff there is a redex within t.*

The proofs of these properties can be found in [13]. A more detailed theoretical account of the constraint-mechanism with fully formal proofs, run-time analysis and analysis of the properties of synthesis with respect to term-ordering is interesting further work. In the rest of this article, we focus on the implementation, design choices, and empirical results of the constraints machinery.

6 Sources of Initial Constraints

The initial constraints given to the synthesis machinery are derived from function definitions, datatype theorems and other library theorems. Below, we give examples of each kind, and show the constraints that are derived.

6.1 Constraints From Function Definitions

Function definitions provide an important source of initial constraints for synthesis. Recall the definition of natural numbers:

$$0 + y = y$$
$$(Suc\ x) + y = Suc(x + y)$$

Following the algorithm in §5.2, the left-hand side of the first defining equation is a function application (of 'plus' to 0 and y), so we proceed to compute constraints on the arguments. As 0 is a constant, a *NotAllowed*-constraint on the first argument of + is generated. The variable y only occurs once, so it does not contribute to any constraints. As the *NotAllowed*-constraint for 0 is the only constraint, 0 is removed from the domain of the first argument of plus.

For the left-hand side of the second defining equation, the first argument of + is a function application ($Suc\ x$), but its argument does not produce any constraints. Hence we get only a single *NotAllowed*-constraint which forbids Suc to occur as the first argument of +.

In general, the defining equations of functions defined by structural recursion will restrict the domain of the argument(s) on which the function is recursive.

6.2 Constraints From Datatype Theorems

Isabelle's datatype package will automatically derive a number of useful theorems when a new datatype is defined. These will typically be used to provide constraints on equalities. Our program automatically uses the injectivity and so called distinctness theorems for each datatype to provide synthesis with a useful set of initial constraints.

Returning to our running example of natural numbers, Isabelle derives an injectivity theorem (§3, Example 2):

$$((Suc\ n) = (Suc\ m)) = (n = m)$$

It also derives a so called distinctness theorem[2]:

$$(Suc\ n = 0) = False$$

From injectivity, we derive constraints stating that the first and second arguments of an equality are not simultaneously allowed to be instantiated to Suc:

$$C_1: \quad NotSimult((C_1.1, C_2),\ (C_1.2, C_3))$$
$$C_2: \quad NotAllowed(C_1.1,\ \text{'Suc'})$$
$$C_3: \quad NotAllowed(C_1.2,\ \text{'Suc'})$$

[2] Isabelle actually derives a slightly different variant, of the form $Suc\ n \neq 0$, which IsaCoSy uses to derive an equivalent theorem suitable for our constraint derivation algorithm.

The distinctness theorem forbids the two arguments of an equality being instantiated to opposite constructors:

$$\begin{aligned} C_1: &\quad \mathit{NotSimult}((C_1.1, C_2),\ (C_1.2, C_3)) \\ C_2: &\quad \mathit{NotAllowed}(C_1.1,\ \text{`Suc'}) \\ C_3: &\quad \mathit{NotAllowed}(C_1.2,\ \text{`zero'}) \end{aligned}$$

6.3 Reflexivity: Equality Constraints

Recall the reflexivity theorem from Example 3 of §3: $(x = x) = True$. To avoid this being applicable as a rewrite rule, we do not want to synthesise any terms with identical left and right-hand sides. This results in an equality constraint on the two arguments of the equality:

$$C_1: \quad \mathit{UnEqual}(C_1.1,\ C_1.2)$$

Of course, it might not be possible to establish that the two subterms are indeed different until the whole term is fully synthesised. If the two arguments become instantiated to different top-level symbols, the constraint can be dropped. Otherwise, the equality is broken down into sub-constraints on new holes appearing after instantiation. Unlike the constraints from injectivity and distinctness, we do not know in advance how many levels down the term tree we might have to look before an equality constraint, such as reflexivity, can be dismissed.

6.4 Commutativity: Argument Order Constraints

Commutativity theorems are used to avoid symmetries in synthesis. If we know that a function is commutative, we can impose an order on its arguments. For example, always require the leftmost argument to be of greater or equal size. Initially, IsaCoSy has access to the commutativity theorem for equality:

$$(x = y) = (y = x)$$

Although the ordering used to constrain synthesis of commutative functions is a parameter to our synthesis algorithm, for simplicity, from here on we will use term-size. Thus, for an equality, $?h_1 = ?h_2$, we impose the constraint that the size of $?h_2$ is always smaller or equal to the size of h_1. This allows us to, roughly, cut the synthesis space in half[3].

Size constraints are currently not expressed in the constraint language described above, but attached to holes during synthesis. This lets us take advantage of the fact that our term-synthesis is always given a bounded size. Every time a new hole is introduced, every possible size that the hole can take, up to the bound, is generated. Synthesis of each size is then performed independently.

Commutativity theorems can be provided by the user. By default we provide the commutativity of equality. During synthesis, when a commutativity property is found, it is also used automatically to further constrain synthesis.

[3] Some symmetric terms will be synthesised: when both sides of the equation have the same size. A total order on terms could be used to remove all symmetries. There is a trade-off between the time to compute the ordering and the synthesis space it removes.

7 Additional Heuristics

We also make use of three simple, additional heuristics in order to further constrain synthesis.

7.1 Variable occurrence

A common heuristic for equational rewriting is to only allow rules where the variables in the right-hand side are a subset of the variables on the left. For example, $f(x,\ y) = x$ is a valid rewrite rule, but $x = f(x,\ y)$ is not. As we are interested in synthesising valid rewrite rules, the default settings for IsaCoSy is to only allow holes in the left-hand side to be instantiated to fresh variables, while variables on the right-hand side are only allowed to be picked from those already occurring on the left. For example, if we have the following partially synthesised term, $f(x,\ y) = ?h$, the only variable-instantiation allowed for $?h$ is x or y.

7.2 Number of Variables Allowed

IsaCoSy allows the user to specify how many different variables should be allowed in the synthesised terms. In many theories, such as lists or natural numbers, the theorems found in textbooks and prover libraries often have no more than two or three variables. Studying the theorems in Isabelle's libraries[4] for natural numbers and lists, suggests that a good default heuristic for the number of different variables is 1 + the maximum arity of any function involved. While restricting the number of variables can cause theorems to be missed, it is very useful in reducing the synthesis space.

7.3 Eager Check for Associativity and Commutativity

Another option IsaCoSy supports is to check, prior to synthesis, for associativity and commutativity properties. If the AC-option is switched on, proofs of associativity and commutativity are attempted before the synthesis process starts. This is attempted for all binary function with arguments of the same type. Should the proof of commutativity succeed, we can impose ordering restrictions on the function's argument during synthesis (recall §6.4). Furthermore, if the function is commutative, it is likely that the commuted variants of its definitional theorems will be useful to our prover, so these are also added to the set of synthesised terms. For example, the commuted definitions of plus (defined as in §3) gives us the two theorems:

$$y\ +\ 0 = y$$
$$y\ +\ (Suc\ x) = Suc(y\ +\ x)$$

These theorems are obtained by commuting both the left- and right-hand sides. Commuting only one side is also an option, which would give another two versions, but these were not found to be useful for proof automation in IsaPlanner.

[4] `www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library`

8 Synthesising Conjectures

After the initial constraints of the current theory have been computed, synthesis can start. The synthesis algorithm is given a table of current constraints for the available function symbols and an initial top-level term. The initial term contains meta-variables that are interpretted as the holes to be instantiated. In our experiments, we use the initial term $?h_0 = ?h_1$.

Synthesis is also given a maximum and minimum size limit for the terms being synthesised. At each iteration, non-theorems are filtered out by counter-example checking. The conjectures that can then be proved are fed into the constraint generation mechanism to provide further constraints, thus restricting the synthesis space for future terms. We have also experimented with allowing constraints to be generated from terms that pass counter-example checking, rather than just those that IsaPlanner proves (see §10.3) So far, all the terms that passed counter-example checking were theorems.

8.1 A Data-Structure for Synthesis

The synthesis algorithm uses a data-structure, which we an call *STerm*, to keep track of information related to the current synthesis attempt. An STerm contains:

- The term synthesised so far.
- The name, type and term-size for each uninstantiated hole.
- A table of current constraints, indexed by their unique names.
- The constraints associated with each hole.
- The domain of allowed constants that each hole can be instantiated to.
- Constraint dependencies, keeping track of parent-constraints where relevant.

As synthesis progresses, by instantiating the holes, the constraints on them will be evaluated and either dropped (if they no longer apply) or refined to provide restrictions on new holes.

8.2 Overview of the Algorithm

IsaCoSy starts from some specified minimum size and performs one iteration of the algorithm below for each size, up to the given maximum size.[5]

1. Initialise synthesis by importing the constraints associated with the given top-level function (for example, equality). Also compute the allowed size combinations for the holes.
2. Pick the next hole to be instantiated from the search agenda. The current version of IsaCoSy uses depth-first search, but the synthesis machinery is compatible with other search strategies.
   - If *hole-size = 1*: Instantiate the hole to a constant of size 1 (e.g. the empty list, or the constant 0 for natural numbers), or to a variable. Variables can either be fresh or already exist elsewhere in the term. Existing variables are filtered

[5] The implementation is purely functional ML and provides a more detailed account for the interested reader.

against *VarNotAllowed*-constraints on the hole, in case they are forbidden in this position.

If synthesising an equality, fresh variables are typically only allowed in the left-hand side. Similarly, if we have chosen a maximum number of different variables, fresh variables are only allowed as long as this limit has not been exceeded.

– Else, *hole-size* > 1: Instantiate the hole to a function with arguments providing new holes. Consider all function-symbols in the domain of the hole that have a minimum term size satisfying $1 <$ *min-size* $\leq$ *hole-size*.

3. Update and propagate constraints, given the instantiation and possible new holes (see §8.3).
4. Terminate when there are no more open holes. Filter-out the resulting terms through the counter-example finder. We used Isabelle's quickcheck [1].
5. Attempt to prove the remaining conjectures. We used the inductive theorem prover provided by IsaPlanner [8].
6. Use the constraint generation algorithm on the new theorems to produce additional constraints before the next iteration of synthesis. New theorems are not currently used to prune the set of earlier produced theorems.

Note that IsaCoSy's synthesis algorithm is independent of the order in which holes are instantiated. The term-synthesis machinery and constraint language is designed to allow experimentation with different search strategies. This also allows implementation of additional heuristics to exploit particular search strategies. An example is the variable occurrence heuristic from §7, which was implemented in a way that requires the search strategy to instantiate the left hand side of a rewrite rule before the right hand side.

8.3 Constraint Propagation

The constraint propagation mechanism is crucial for the synthesis algorithm's efficiency. Our constraint language supports expressing 'future' constraints which depend on the instantiations of current holes. These constraints need to be updated and propagated to any new holes. In particular, we need to manage the propagation of equality-constraints, which may break up into several new constraints as holes are instantiated. Constraint propagation also has to take dependencies into account, by checking if constraints are children of a *NotSimult*.

Constraint Propagation Algorithm

The constraint propagation algorithm is called when a hole, $?h$, is instantiated to a symbol s, where $?h$ has constraint C. We use k to stand for some arbitrary constant, and v to stand for a variable.

If the constraint C is a child-constraint of some other constraint, we will call its parent, C_p. Furthermore, during synthesis, all such parent constraints are of the form *NotSimult*, as *IfThen*-constraints only have child-constraints that talk about possible future holes.

Depending on C, the following constraint updates are made:

$C = \mathit{NotSimult}[(pos_1,\ C_1),\ \ldots,\ (pos_i,\ C_i),\ \ldots,\ (pos_n,\ C_n)]$: Assuming $pos_i = ?h$, the constraint propagation function is called on the sub-constraint C_i, associated with

the hole $?h$ that is being instantiated. In processing C_i, the fact that it is part of a *NotSimult* must be taken into account. If the constraint expressed by C_i has been satisfied, it no longer applies. Hence C is also satisfied, and can be dropped. On the other hand, if C_i is violated or replaced by a child-constraint, its sibling constraints and its parent constraint, C, must remain. The constraints are thus updated as follows:

- if C_i is satisfied: Delete C_i, along with its parent *NotSimult*-constraint, and any sibling-constraints.
- if C_i is violated: Delete C_i from its parent. The initial constraint thus becomes:

  $$NotSimult[(pos_1,\ C_1),\ \ldots,\ (pos_{i-1},\ C_{i-1}),\ (pos_{i+1},\ C_{i+1}),\ldots,\ (pos_n,\ C_n)]$$

  If there is now only one child-constraint in C, then there no need for the *NotSimult* constraint, and it can be replaced by the remaining child-constraint. For example, if we get $NotSimult[(pos_j,\ C_j)]$, it is sufficient to keep the constraint C_j on its own.
- if C_i is neither satisfied not violated, then it is replaced by the sub-constraint(s) associated with the symbol s. Each sub-constraint, C_i' is associated with a new hole, $?h'$. We replace C_i by a number of C_i's and let $pos_i' = ?h'$ in C:

  $$NotSimult[(pos_1,\ C_1),\ \ldots,\ (pos_i',\ C_i'),\ \ldots,(pos_n,\ C_n)].$$

  If C_i' also happens to be a *NotSimult*-constraint, it is merged with the parent constraint.

$C = NotAllowed(?h,\ k)$: This constraint only occurs if it is a sub-constraint of a *NotSimult*. Otherwise k would have been removed directly from the domain of $?h$. Hence, C must be a sub-constraint of a *NotSimult*-constraint, C_p.

- If $s \neq k$: C is satisfied. C_p and all its sub-constraints can be dropped (as described above).
- Else, $s = k$: C is violated and removed from C_p, but C_p must remain and is updated as described above.

$C = VarNotAllowed(?h,\ v)$: If considering instantiating a hole with a variable, this constraint is checked at instantiation, ensuring the hole is not instantiated to v. If it is a sub-constraint of a *NotSimult*, the process is analogous to the above case for *NotAllowed*.

$C = IfThen(?h,\ s',\ C_j)$:
- If $s \neq s'$: C is satisfied, and can be dropped along with its sub-constraint, C_j. If any parent constraint exists, this is updated accordingly.
- Else, $s = s'$: the sub-constraint C_j must be considered. C_j should be attached to some new hole(s). To determine which one(s), the argument position-types in the sub-constraint C_j are updated from *LocalIndex*-type to *Hole*-type, as described in §4.1. The *IfThen*-constraint C is then deleted.

  If there is a parent *NotSimult*-constraint C_p then C is replaced by C_j in the parent, as described above. Otherwise, if the sub-constraint C_j is of the form *NotAllowed*, the domain of the relevant hole is updated accordingly, before C_j is deleted.

$C = UnEqual[(pos_1,\ C_1),\ \ldots,\ (pos_i,\ C_i),\ \ldots,\ (pos_n,\ C_n)]$: Assuming $pos_i = ?h$. The other argument positions correspond to other holes. *UnEqual*-constraints always break down into further constraints, until they disallow particular variables or constants. Propagation of equality constraints depends on s:

- $s = v$ (instantiation to a variable): the other arguments of the equality are not simultaneously all allowed to be instantiated to v. The following new constraints are added to express this:

$$\begin{aligned} C_j: &\quad \mathit{NotSimult}((pos_1,\ C_{pos_1})\ \ldots\ (pos_n,\ C_{pos_n})) \\ C_{pos_1}: &\quad \mathit{VarNotAllowed}(pos_1,\ v) \\ &\quad \vdots \\ C_{pos_n}: &\quad \mathit{VarNotAllowed}(pos_n,\ v) \end{aligned}$$

  where the *NotSimult*-constraint is only necessary if there is more than one other argument involved in the equality-constraint. Finally, the original equality-constraint, C, is dropped.
- $s = k$: As for variables but with a $\mathit{NotAllowed}(pos_i,\ k)$ for each of the remaining arguments.
- $s = f(?x_1 \ldots ?x_m)$: Given that $?h$ is instantiated to a function with new holes, $?x_1 \ldots ?x_m$, we must create future-constraints on the potential instantiations for the other arguments of C in positions $pos_1\ \ldots pos_n$.
  The *UnEqual*-constraint can only be violated if the other positions also are instantiated to f, and the argument positions of f are instantiated to the same symbol everywhere. We thus first create the following new equality constraints on the new holes $?x_1 \ldots ?x_m$:

$$\begin{aligned} C_{x_1}: &\quad \mathit{UnEqual}[?x_1,\ C_{pos_1}.1\ \ldots\ C_{pos_n}.1] \\ &\quad \vdots \\ C_{x_m}: &\quad \mathit{UnEqual}[?x_m,\ C_{pos_1}.m\ \ldots\ C_{pos_n}.m] \end{aligned}$$

  Furthermore, we need to create an *IfThen*-constraint for each position $pos_1\ \ldots pos_n$ of the original constraint (the constraints named $C_{pos_1} \ldots\ C_{pos_n}$ above). These need to specify the further constraints applying to potential new holes in the argument positions of f:

$$\begin{aligned} C_{pos_i}: &\quad \mathit{IfThen}(pos_i,\ f,\ C_{pos_{i'}}) \\ C_{pos_{i'}}: &\quad \mathit{NotSimult}((C_{pos_i}.1,\ C_{x_1})\ \ldots\ (C_{pos_i}.m,\ C_{x_m})) \end{aligned}$$

  Finally, all of the *IfThen*-constraints created above are dependent on each other. We thus create a top-level *NotSimult*-constraint to express the dependency between the positions of the original constraint:

$$C': \quad \mathit{NotSimult}((pos_1,\ C_{pos_1})\ \ldots\ (pos_n,\ C_{pos_n}))$$

An *UnEqual*-constraint C, may have several parent constraints for its different argument positions. If there was a parent *NotSimult*-constraint, C_p, for the hole $?h$, then the constraints $C_{x_1} \ldots C_{x_m}$ on the new holes $?x_1\ \ldots\ ?x_m$ replace C in C_p. If a parent constraint exists for any of the other arguments of C ($pos_1\ \ldots pos_n$), then C is replaced by the new constraint C' in C_p.

8.4 After Synthesis

After the synthesised terms have been filtered through the counter-example checker, the remaining conjectures are passed on to IsaPlanner which tries to prove them. IsaPlanner applies induction with rippling, lemma calculation and case-analysis. Lemma calculation is helpful in some proofs, as a conjecture may sometimes need a lemma that has not been synthesised yet. An example is the proof of $rev(rev\ l) = l$, for which IsaPlanner calculates and proves a needed lemma: $rev(l\ @\ [h]) = h\#(rev\ l)$. Without lemma calculation, the proof above would have to wait until synthesis had found the theorem $(rev\ a)\ @\ (rev\ b) = rev(b\ @\ a)$, which is a more general variant of the lemma above. Furthermore, the calculated lemmas are also fed into the next iteration of constraint generation so that they further cut down the synthesis space.

IsaCoSy will occasionally produce theorems that might be considered special cases of other theorems, e.g. $(a\ +\ b)\ +\ a = (b\ +\ a)\ +\ a$ as well as the general version $(a\ +\ b)\ +\ c = (b\ +\ a)\ +\ c$. This is because constraints are only used after synthesis finishes producing all conjectures and theorems of a given size. Thus theorems, such as associativity, will only be used to prune further synthesis at the next round. As the specialised variants are rarely of interest, IsaCoSy has a subsumption check to filter theorems for which a more general variant exists.

9 Case Study: A Small Theory about Natural Numbers

To illustrate how IsaCoSy works, consider a minimal theory about natural numbers, with one recursive function, +, defined in the usual way. In total we have three function symbols: +, Suc and =, as well as the constant 0. To generate initial information about constraints and argument domains, we have the injectivity and distinctness rules for Suc, reflexivity as well as the two rules defining +. Finally, we also assume that the heuristic for only allowing fresh variables in the left-hand side of an equation is used. We do not impose any restrictions on how many different variables are allowed, nor do we attempt to eagerly discover associativity and commutativity theorems. Using this configuration, we compare the number of conjectures synthesised by IsaCoSy with a naive version of synthesis that generates all terms [16]. The naive version constructs all terms that are type-correct combinations of a given set of function symbols and free variables. Like IsaCoSy, it does not create any terms containing lambda-abstractions and never places a free variable at the head of a compound term.

We will use the notation $x \in \{\ldots\}$ to specify the set of constants used to incrementally instantiate x. For instance, $x \in \{0, Suc\}$ means that x can be instantiated to the constant 0 or to a term of the form $Suc\ ?x_2$. Addition will initially have the following associated information about argument domains and constraints:

Name:	$x\ +\ y$
Min size:	3
Argument Domains:	$x \in \{+\}$
	$y \in \{0, Suc, +\}$
Term-Size:	-
Constraints:	-

Recall that the omission of 0 and Suc from the domain of the first argument comes from the defining equations being treated as rewrite rules.

For =, the initial information is:

Name:	$l = r$	
Min size:	3	
Argument Domains:	$l \in \{0, Suc, +\}$	
	$r \in \{0, Suc, +\}$	
Term-Size:	$\|l\| \geq \|r\|$	(Commutativity)
Constraints:	$\mathbf{C_1} : NotSimult((l,\ C_{l_1}),\ (r,\ C_{r_1}))$	(Injectivity)
	$C_{l_1} :\ NotAllowed(l,\ Suc)$	
	$C_{r_1} :\ NotAllowed(r,\ Suc)$	
	$\mathbf{C_2} :\ NotSimult((l,\ C_{l_2}),\ (r,\ C_{r_2}))$	(Distinctness)
	$C_{l_2} :\ NotAllowed(l,\ Suc)$	
	$C_{r_2} :\ NotAllowed(r,\ 0)$	
	$\mathbf{C_3} :\ UnEqual(l,\ r)$	(Reflexivity)

Finally the initial information for *Suc* is:

Name:	$Suc\ n$
Min size:	2
Argument Domains:	$n \in \{0, Suc, +\}$
Term-Size:	-
Constraints:	-

We want to synthesise equations. This means we have to start synthesising terms of size 3, with = as the top level symbol and two holes, each of size 1. The initial term is thus:

$$\underbrace{?h_1}_{size\ 1} = \underbrace{?h_2}_{size\ 1}$$

The holes, represented by the meta-variables $?h_1$ and $?h_2$ will, in addition to their specified size, inherit the restrictions specified for the corresponding arguments of = above.

Size 3

We can generate two terms of size 1, the constant 0 or a variable a. Putting these together, IsaCoSy synthesises only one term: $a = 0$ (out of a possible five for the naive version of synthesis). The synthesised term is not a theorem, so it is discarded after counter-example checking. Note that IsaCoSy does not synthesise $a = a$ or $0 = 0$ thanks to the equality constraint from reflexivity. Neither does it synthesise $0 = a$ or $a = b$, as both of these have variables in the right-hand side that do not occur on the left.

Size 4

For terms of size 4, IsaCoSy start from the template $\underbrace{?h_1}_{size\ 2} = \underbrace{?h_2}_{size\ 1}$. Note that we do not consider terms where the right-hand side is larger than the left, due to the constraint arising from the commutativity of equality.

IsaCoSy only synthesises one non-theorem (which is caught by counter-example checking) for this size:

$$Suc\ a = a$$

Note that conjectures of the form $Suc(\ldots) = 0$ are not generated as this can be written to *False* by the distinctness theorem.

The naive version produces ten non-theorem conjectures of size 4 (we use / to separate alternative right-hand sides):

$$\begin{array}{cc} a = Suc\ 0\ /\ Suc\ a\ /\ Suc\ b & 0 = Suc\ 0\ /\ Suc\ a \\ Suc\ a = 0\ /\ a\ /\ b & Suc\ 0 = 0\ /\ a \end{array}$$

Size 5

For terms of size 5, we get two possibilities to start from:

$$\underbrace{?x_1}_{size\ 2} = \underbrace{?x_2}_{size\ 2} \text{ and } \underbrace{?y_1}_{size\ 3} = \underbrace{?y_2}_{size\ 1}$$

From the size restriction, the former can only attempt to generate terms of the form $Suc\ ?x = Suc\ ?y$, but this is disallowed due to the injectivity of Suc, so IsaCoSy will not generate any conjectures in this case.

Using the second template, IsaCoSy produces 8 conjectures:

$$\begin{array}{cc} a + b = 0\ /\ a\ /\ b & a + 0 = 0\ /\ a \\ a + a = 0\ /\ a & Suc(Suc\ a) = a \end{array}$$

The list above includes only one theorem: $a\ +\ 0 = a$. The remaining conjectures are filtered out by counter-example checking. The theorem found can be proved automatically and is then given to the constraint generator, which will conclude that we should no longer generate terms where 0 is the second argument to +. The naive version of synthesis generates a 45 conjectures of size 5.

We will revisit this case-study in §10.1.1, where we evaluate IsaCoSy and compare its performance with the naive version on several larger theories. We will also discuss the effect of additional heuristics on the synthesis space.

10 Evaluation and Methodology

Our evaluation is based on inductive theories involving natural numbers, lists and binary trees. IsaCoSy is given the task of synthesising equational theorems. We consider the following main hypotheses:

- IsaCoSy has an exponentially smaller synthesis-space than naive synthesis. This makes synthesis of many useful and interesting theorems computationally feasible.
- IsaCoSy produces useful theorems. In particular, the kind of theorems found in Isabelle's libraries and which can be used by automatic proof tools.

In order to select the other heuristic parameters with which to explore these hypotheses, we performed a preliminary case-study, detailed in §10.1.1, which extends the basic theory of addition we started in §9. In this study, we compare the effects of IsaCoSy's two optional heuristics (presented in §7) on the size of the synthesis space. Recall that the optional heuristics concern whether to attempt to prove associativity and commutativity properties prior to synthesis, as well as restrictions on the number of different variables allowed in terms. These experiments provide evidence that the heuristics are beneficial to synthesis.

We also carried out preliminary experiments to check the maximum size of terms that IsaCoSy can synthesise before it runs out of memory. For the theories involving append and reverse, memory ran out most quickly and conjectures greater than size 14 could not be synthesised with a memory limit of 500MB. Having established this, and observed that all theorems in the comparable theories in the Isabelle library are smaller, all further experiments were limited to size 14.

With these basic settings, we then perform the main analysis using six larger theories with IsaCoSy's optional heuristics switched on. Here, we recorded the run-time for the different tasks in synthesis, as well as the size of the synthesis space, and which conjectures were synthesised, and which of these were then proved. The theories used are:

- Natural Numbers:
  - addition and multiplication
- Lists:
  - append, reverse, length
  - append, reverse, map
  - append, reverse, quick-reverse (qrev)
  - append, foldl, foldr
- Binary Trees:
  - mirror, height, nodes, max.

The definitions of the functions above can be found in Appendix A. We found that most theorems in Isabelle's library contain around three function symbols, which is why we limit our evaluation-theories to either three or four function symbols each.

The experiments were run on a computer with a 2 GHz Intel Xenon processor. Full results from the experiments are available on-line[6], including all theorems, conjectures, proofs and run-time statistics for each theory. The synthesised theorems from the experiments are listed in Appendix B.

10.1 Evaluating the Size of the Synthesis Space

The main difficulty with term synthesis is that the number of possible terms grows exponentially as term size is increased. As a result, many interesting conjectures are too large to be synthesised naively. In this section we examine the size of the synthesis space in IsaCoSy. We first present a brief evaluation of IsaCoSy's optional heuristics in §10.1.1, illustrating their effect on the synthesis space size. Having established the benefits of these heuristics, in §10.1.2, we present a comparison of the synthesis space size for IsaCoSy and the naive version of synthesis on the larger theories listed above. As we see in §10.1.3, the overall run-time of IsaCoSy is proportional to how many terms are generated and have to be counter-example checked. Hence, we do not provide any run-times for the naive version but only compare how many terms are generated by each algorithm, bearing in mind that this is proportional to the overall run-time in most cases.

[6] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/synth_results.php`

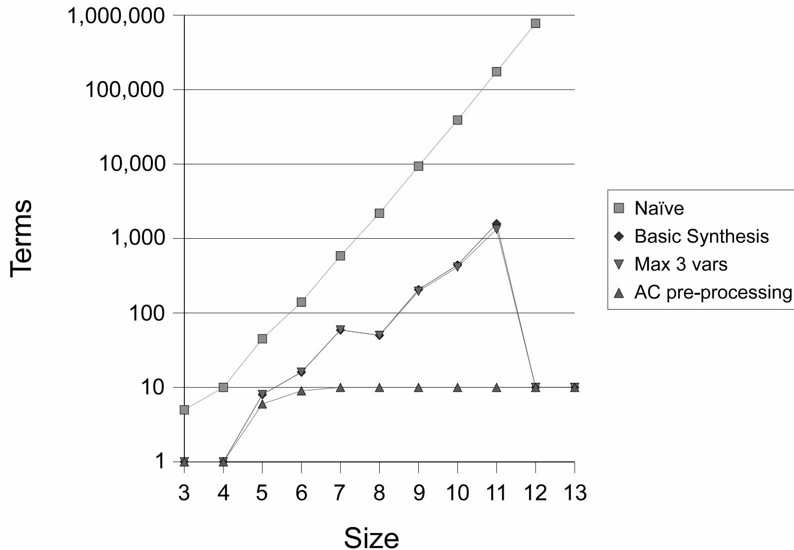

Fig. 1 The number of equational terms generated up to size 13 about natural numbers with addition for a naive version of synthesis, IsaCoSy's basic version of synthesis and synthesis with additional heuristics. The graph shows the synthesis space reduction achieved by adding a pre-processing step to look for associativity and commutativity properties, as well as restricting the number of different variables allowed in synthesised terms. Note that the y-axis scale is logarithmic for better visibility.

10.1.1 Effect of Heuristics

We now continue where we left the case-study about addition on natural numbers from §9. Figure 1 summarises the number of terms synthesised for increasingly large term sizes. As well as the naive variant and IsaCoSy's basic version of synthesis, which was described in §9, results are also included for two versions of synthesis using the optional heuristics described in §7. The first of these is restricted to only allow three different variables in the synthesised terms (the arity of $+$ is 2, so we allow 3 variables). Because synthesis is highly constrained in this example, this heuristic only has a noticeable effect when large numbers of terms are generated. The other version of synthesis has the variable occurrence restriction as well as eagerly attempting to synthesise associativity and commutativity properties prior to commencing synthesis. As we shall see, this version of synthesis performs the best and is thus used in further experiments.

The number of terms increases exponentially for the naive version, until it runs out of memory when reaching size 13 (given a limit of 500MB). IsaCoSy's basic synthesis version performs considerably better. When reaching size 7, the commutativity of addition is discovered, which allows the arguments to be ordered, cutting out many symmetries. In fact, IsaCoSy synthesises fewer terms of size 8 than size 7. The largest number of conjectures are synthesised for size 11. At this point, the associativity of addition is discovered. All theorems in our small theory (adhering to the constraints of

Size	Theorem
5	$a + 0 = a$
7	$a + Suc\, b = Suc(a + b)$
7	$a + b = b + a$
11	$(a + b) + c = (b + a) + c$
11	$(a + b) + c = (a + c) + b$
11	$(a + b) + c = (c + b) + a$

Table 1 Theorems about addition discovered by IsaCoSy without optional heuristics.

the synthesis algorithm) have now been discovered, and all corresponding constraints are available. The domains for the arguments of addition are now empty, meaning no function symbols is allowed occur in these positions. If asked to continue, only 10 terms are considered of size 12 and 13. These are 'silly' non-theorems of the form $Suc(\ldots Suc(Suc(a + b))) = a + b$, stacking up lots of successor functions.

When restricting IsaCoSy to only 3 different variables (each of which may occur several times), even fewer terms are generated for larger sizes. The same theorems are still discovered. The greatest difference comes from the AC pre-processing heuristic which, in this toy theory, manages to discover all relevant theorems as consequences of associativity or commutativity. Table 1 shows the theorems discovered and proved by IsaCoSy using the basic setting. These are, as expected, the commuted variants of the definition of plus, as well as commutativity and associativity. Associativity appears in a few different variants, but not in the common form: $(x + y) + z = x + (y + z)$. This is because the constraint imposed after discovering the commutativity of addition requires that the first argument of addition has greater or equal size compared to the second argument.

10.1.2 Synthesis Space Reduction over Naive Synthesis

We now consider slightly larger theories. IsaCoSy is configured to using all the optional heuristics. Thus the number of different variables allowed is restricted to one more than the maximum arity of any function in the theory. IsaCoSy is also set to perform pre-processing steps looking for associativity and commutativity theorems prior to the start of synthesis.

Figure 2 compares the total number of terms, up to size 11, generated by IsaCoSy and by the naive version of synthesis. The six evaluation theories are combinations of functions on natural numbers, lists and trees. For term-sizes larger than 11, the naive version runs out of memory.

The large difference between IsaCoSy and the naive approach on the natural number theory is due to both addition and multiplication being associative and commutative. These properties produce useful constraints which significantly cut down the synthesis space. The more structure a theory has, the more efficient synthesis is.

IsaCoSy's synthesis space is largest for the quick-reverse theory. This is because many of these theorems require generalisation of an accumulator variable, which is beyond the current capabilities IsaPlanner. As the theorems cannot be proved, they are not used to generate additional constraints, so the synthesis space remains relatively large. We discuss this in more detail in §10.3.

Figure 3 shows the ratio in the size of the synthesis space between the naive version and IsaCoSy. This is computed on term sizes from 3 up to 11. Beyond size 11 the naive

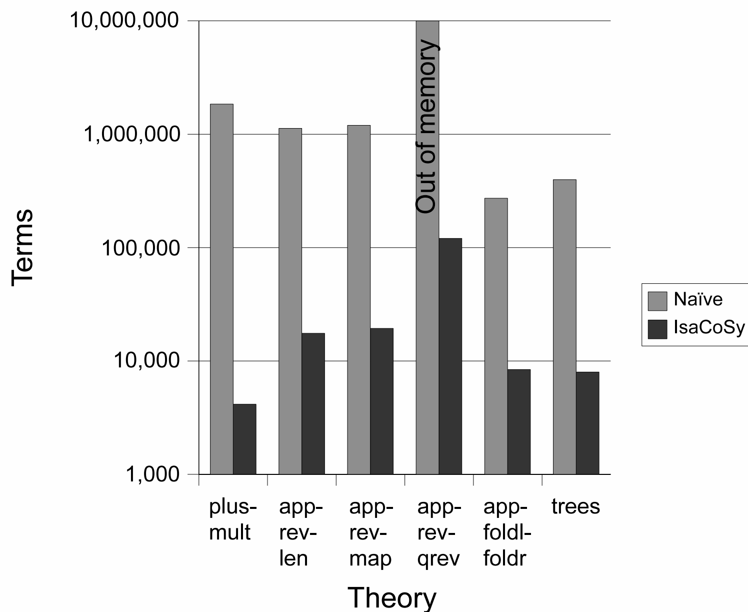

Fig. 2 Number of terms generated up to size 11 for IsaCoSy and for a naive version of synthesis. The different theories are combinations of functions on natural numbers, lists and trees. The synthesis-space for IsaCoSy is considerably smaller: the y-axis scale is logarithmic to make the diagram more readable.

algorithm runs out of memory. The factor by which IsaCoSy's synthesis space is smaller grows as the term size increases.

10.1.3 Run-time and Space Usage

We found that the run-time and space usage of IsaCoSy increases exponentially with the size of the terms synthesised. Furthermore, the time taken per iteration is proportional to the number of synthesised terms of that size. Figure 4 illustrates this on the theory about natural numbers. The number of terms generated has been plotted together with the run-time for each size. The graphs increase exponentially, and clearly mirror each other.

The reason for the correlation between the number of terms and the run-time is that IsaCoSy spends most of its time doing counter-example checking on the terms it generates. Proof attempts make up a considerably smaller proportion of the total time. This is because the vast majority of terms generated are non-theorems and filtered out by counter-example checking. The total run-times, as well as timings for the different tasks making up the synthesis process are summarised in Table 2. This shows that for all theories in the experiments, IsaCoSy spends the majority of time performing counter-example checking. Comparatively little time is spent on proof attempts.

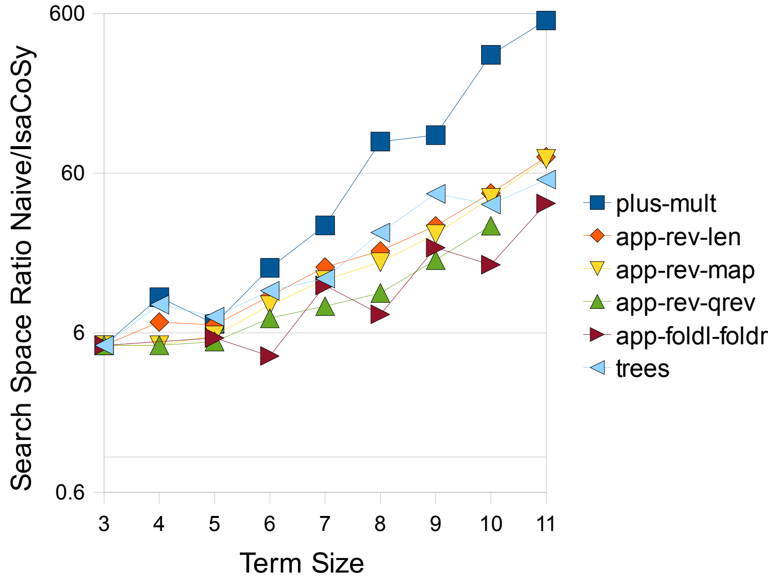

Fig. 3 The ratio, in terms of the size of the synthesis space, between the naive version of synthesis and IsaCoSy. Term sizes 3 - 11 are shown. The number of conjectures is calculated on six theories involving functions on natural numbers, lists and binary trees. The factor by which IsaCoSy's synthesis space is smaller grows with increasing term size. Note that the y-axis scale is logarithmic.

Theory	Total time	Counter-examples	Proof Search	Synthesis
plus-mult	1h 22 min	1h 20 min	1 min	48 sec
app-rev-len	15h 56 min	15h 30 min	17 min	9 min
app-rev-map	17h 21 min	16h 51 min	17 min	11 min
app-rev-qrev	18h 43 min	17h 20 min	1h 11 min	12 min
app-foldl-foldr	6h 8 min	6h 5 min	3 sec	3 min
trees	9h 46 min	9h 41 min	3 min	2 min

Table 2 Total run-time for IsaCoSy on six theories up to size 14, along with a breakdown of how much time was spent on each sub-task during the synthesis process. Times are rounded up to nearest hours and minutes. Note that for all theories, the largest proportion of time is spent on counter-example checking.

10.2 Precision/Recall Analysis

To assess the quality of the theorems produced by IsaCoSy we performed a precision/recall analysis using Isabelle's library as the reference. However, Isabelle does not have a standard library for binary trees, so this theory could not be analysed here. The quick-reverse function is also not included in the library, and was thus excluded.

10.2.1 Natural Numbers

The theorems synthesised in the theory natural numbers, involving addition and multiplication, are shown in Table 3 in Appendix B. The standard commutativity and

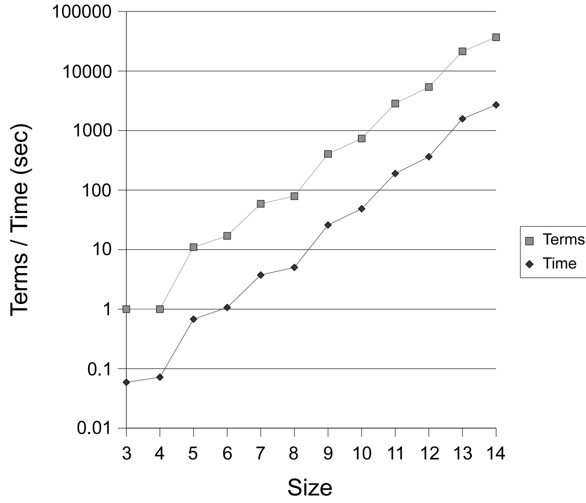

Fig. 4 The two graphs show the time taken and the number of synthesised terms for each iteration from size 3 - 14 of synthesis. This is for the theory about natural numbers with addition and multiplication. The graphs mirror each other, hence we can conclude that the run-time is approximately proportional to the number of terms synthesised for each size. The y-axis uses a logarithmic scale, which means that the growth in synthesis space size is exponential as the size of terms synthesised is increased.

associativity theorems are synthesised, along with commuted versions of the function definitions. IsaCoSy also synthesises theorems for the distributivity of multiplication over addition.

Isabelle's library contains 12 equational theorems about addition and multiplication, 10 of which are synthesised by IsaCoSy:

$$\begin{array}{rcl} a + 0 = a & \quad & a + Suc\, b = Suc(a + b) \\ a * 0 = 0 & & a * Suc\, b = a + (a * b) \\ a + b = b + a & & a * b = b * a \\ (a + b) + c = a + (b + c) & & (a * b) * c = a * (b * c) \\ (a * b) + (c * b) = (a + c) * b & & (a * b) + (a * c) = (b + c) * a \end{array}$$

Using Isabelle's 12 theorems as a benchmark for 'interestingness' we can calculate precision and recall for IsaCoSy. With ten of our theorems included in the library, this gives recall of 83%. IsaCoSy synthesised a total of 16 theorems for this theory, which gives precision of 63%.

The two theorems from Isabelle's library that are not synthesised are:

$$\begin{array}{rl} \textit{add_suc_shift:} & (Suc\ m) + n = m + (Suc\ n) \\ \textit{nat_left_commute:} & x + (y + z) = y + (x + z) \end{array}$$

These theorems are trivially derivable by simplification from theorems we do synthesise. The theorem *add_suc_shift* is not synthesised as its left-hand side is identical to the left-hand side of the definition of addition. Should we wish to derive theorems of this form, one solution would be to add them to the set of theorems IsaCoSy tries to prove when discovering that a function is commutative. Currently, it derives the commuted versions of the function's definition. For example the theorem $b + (Suc\ a) = Suc(b + a)$.

This could easily be extended to also include trying to prove theorems of the same form as *add_suc_shift*, without having to relax constraints on synthesis. The theorem *nat_left_commute* is not synthesised because of the size-constraint coming from commutativity requires the first argument of addition to be greater than or equal to the second argument in size.

10.2.2 Lists

The theorems synthesised for the list theories are shown in Tables 5, 6 and 7 in Appendix B. IsaCoSy produces a total of 24 theorems in these theories, while Isabelle's list theory contains 9 relevant theorems. All 9 of these are synthesised, giving a recall of 100%. These 'interesting' theorems are:

$$
\begin{array}{cc}
a \mathbin{@} [\,] = a & (a \mathbin{@} b) \mathbin{@} c = a \mathbin{@} (b \mathbin{@} c) \\
rev(rev\ a) = a & (rev\ a) \mathbin{@} (rev\ b) = rev\ (b \mathbin{@} a) \\
rev(map\ a\ b) = map\ a(rev\ b) & (map\ a\ b) \mathbin{@} (map\ a\ c) = map\ a\ (b \mathbin{@} c) \\
foldl\ a\ (foldl\ a\ b\ c)\ d = foldl\ a\ b\ (c \mathbin{@} d) & foldr\ a\ b\ (foldr\ a\ c\ d) = foldr\ a\ (b \mathbin{@} c)\ d \\
len(rev\ a) = len\ a &
\end{array}
$$

Because of the 14 extra synthesised theorems which are not in Isabelle's library, the precision is just 38%. Most of these extra theorems concern reverse and append.

10.3 Allowing Unfalsified Conjectures to Generate Constraints

Quick-reverse (*qrev*) is the tail recursive version of reverse. The proof of many theorems about tail recursive functions, such as *qrev*, require generalisation of the accumulator variable. IsaPlanner does not currently have the capabilities to discover such generalisations, which is why it fails to prove many synthesised theorems in this theory. One example is the theorem $qrev\ a\ [\,] = rev\ a$, which needs the more general theorem $(rev\ a) \mathbin{@} b = qrev\ a\ b$ to complete the proof. In total IsaPlanner proves 19 theorems about *qrev* (see table 8 in Appendix B), while a further 46 conjectures pass counter-example checking, but fail to be proved. As a consequence of failing to prove many theorems, few constraints are generated and the synthesis space remains relatively large compared to the other theories, as shown in figure 2 on page 25. Another consequence is that some of the synthesised theorems that IsaPlanner manages to prove are rather contrived; but if the generalised case could be proved they would not be synthesised.

A solution that results in IsaCoSy generating fewer terms when IsaPlanner fails to prove conjectures, is to allow conjectures that have passed counter-example checking, but not been proved, to also generate constraints. This would benefit synthesis in many theories, including the theory of quick-reverse. We observed that many of the conjectures in this theory which pass counter-example checking, but are not proved by IsaPlanner, appear to be theorems. A random selection of 20 out of the 46 unproved conjectures were proved by hand, and no non-theorems were found, which supports our confidence in Isabelle's counter-example checker for simple equational theories.

We repeated the experiment on the theory about quick-reverse, this time allowing unproved conjectures to generate constraints. The run-time for generating terms up to size 14 was reduced by 11 hours, now only taking 7 hours and 40 minutes. 11 fewer

theorems were generated, but it was only the larger, more contrived ones that were cut out. Theorems L23-24 and L26-31 in Table 8 (Appendix B) were still generated. As an example, theorem L25: $qrev\ (qrev\ a\ b)[\,] = qrev\ b\ a$, is no longer generated as the simpler (but unproved) theorem $qrev\ a\ [\,] = rev\ a$ is now allowed to generate constraints. There were also significantly fewer unproven conjectures, only 8 as opposed to 46, as larger variants of previous unproven conjectures were no longer generated.

Allowing unproved conjectures to generate constraints makes IsaCoSy less dependent on the underlying theorem prover and lets it operate efficiently even when many conjectures remain unproved. There is of course a risk of missing desired theorems if any non-theorems slip through counter example checking.

10.4 Restricting Polymorphic Types

Large gains in run-time can also be obtained by disallowing instantiations of polymorphic type variables to another polymorphic type. For instance, this would disallow nested lists (lists of lists) and nested trees. We noticed that many non-theorems in the list domain contained highly nested lists. As an example of terms that it would be beneficial to prune from the term space, consider a term $?h_0\ \#\ l$, which is of type $?\alpha\ list$, with $?h_0 :: ?\alpha$. Suppose $?h_0$ gets instantiated to another cons-operator, which also instantiates the type variable $?\alpha$ to $?\beta\ list$. The original term is now $(?h_1\ \#\ ?h_2)\#\ l :: ?\beta\ list\ list$. Subsequent instantiations may cause even deeper nesting of lists. The restriction we suggest is to limit type-variables, such as $?\alpha$ above, from being instantiated with types beyond a certain size.

We repeated the synthesis experiments on some of the list theories with a specialised lists datatypes over natural numbers, rather than polymorphic lists. The same theorems as before were still found, but the number of non-theorems decreased significantly and IsaCoSy ran considerably faster. However, the figures for precision and recall from §10.2 were not affected: the same theorems were still discovered. The results are summarised below:

Theory	Run-time		Non-theorems	
	polymorphic	non-polymorphic	polymorphic	non-polymorphic
app-rev-len	15h 56 min	5h 8 min	601 405	229 104
app-rev-map	17h 21 min	4h 31 min	636 361	195 800
app-foldl-foldr	6h 8 min	17 min	249 404	14 503

11 Limitations and Further Work

IsaCoSy can be applied to generate theorems about any recursively defined datatype in Isabelle. However, as IsaCoSy relies on the counter-example checker and automated prover to be able to produce theorems efficiently, the performance of these tools is an important limitations. For example, both the current prover and counter-example checker perform poorly when applied to conjectures with many nested assumptions. IsaCoSy itself could be applied to, for example, inductively defined relations, or even non-inductive theories, without major implementational changes, as long as an efficient counter-example checker and prover is available.

The automated prover in IsaPlanner is an active research project whose improvements will contribute to IsaCoSy. Combining IsaCoSy with other proof techniques might benefit from slightly different constraint configurations. For example, Isabelle's simplifier relies on its rewrite rules being oriented in such a way that rewriting terminates, while rippling terminates regardless of the directions of the rules. However, rippling requires skeleton preservation, and will sometimes need lemmas in a different, and sometimes even more specialised form, to those that IsaCoSy produces. For simplification, it would be interesting to attempt to configure IsaCoSy to produce a confluent set of rewrite rules by, for example, combining it with Knuth-Bendix completion [14]. In addition to rippling and simplification, another technique to consider is rewriting modulo associativity and commutativity. With an AC-rewriting technique, IsaCoSy would not have to consider synthesising commuted versions of theorems.

A limitation to IsaCoSy's constraint for commutative operations is the use of a very simple ordering on terms based on their size. As this is not a total order, the measure will sometimes allow symmetric equations and multiple commutative variants of the same theorem to be synthesised (see theorems L19/L20 in Table 6 and theorems N9-N16 in Table 3 in Appendix B). Furthermore, with the exception of simple commutativity theorems, which can be identified and are used to generate size constraints, IsaCoSy's constraint mechanism cannot determine if a given theorem is a valid rewrite rule. This may result in overly restrictive constraints. An example is the theorem $len(a \ @ \ b) = len(b \ @ \ a)$. This is not a valid rewrite rule, but IsaCoSy treats it as such and generates a constraint disallowing @ to occur as an argument to *len*, which subsequently would exclude a theorem such as $len(a \ @ \ b) = (len\ a) \ + \ (len\ b)$. A solution to the problem would be to only allow constraints to be generated if the theorem is a valid rewrite rule according to some appropriate ordering, such as *recursive path ordering* [7].

Future Applications

We believe the IsaCoSy program has the potential to be useful for assisting theory development as well as for generating challenge problems.

In the theory development setting, a synthesis tool could be applied to functions and datatypes defined by a user. It could then either be left to run to some finite level of completion (e.g. a specified maximum term size), or possibly left to run in the background, finding and proving routine lemmas that may be of use for later proofs. Alternatively, IsaCoSy could be called when the user is stuck in some proof. In this scenario, synthesis can be further restricted to build lemmas from a schematic term using the constants and function symbols in the goal.

Secondly, a synthesis tool could be used to automatically generate test-sets for inductive theorem provers, perhaps for inclusion in a library such as TPTP [23]. Here unfalsified conjectures might also provide interesting challenge problems for the inductive theorem proving community.

12 Related Work

From a foundational point of view, IsaCoSy is the only theory formation system based on a proof assistant with a small fixed logical kernel. This ensures that the soundness of the theory produced by IsaCoSy rests only on the implementation of the logical kernel

and the consistency of HOL. The other feature that distinguishes IsaCoSy from other theory formation systems is that it has the potential to be complete. If the constraints avoid synthesis of any rule that could be rewritten by a confluent rewrite system, such as that produced by the Knuth-Bendix procedure, and the number of variables is not limited, then an instance of every conjecture up to the memory limit will be produced.

Some systems, such as HR [6], and the scheme-based theory exploration model [2], include the capabilities for forming new concepts. IsaCoSy's pre-processing step, looking for associativity and commutativity properties, is similar to how the scheme-based method discovers theorems of a certain form. More generally, IsaCoSy is not concerned with this side of theory formation. It is designed to produce conjectures and theorems about existing functions and datatypes, not to invent new ones. Considering the use of IsaCoSy for concept definition is left as further work.

Thus far, the only other theory formation system that has been applied to discover inductive theorems is the MATHsAiD system [17,18]. Its aim is to generate theorems that a human mathematician would consider interesting, for example, theorems that occur in mathematics textbooks. MATHsAiD has been applied to inductive theorems about natural numbers [18], as well as theorems in non-inductive domains such as basic group theory.

The main general difference is that MATHsAiD is a deductive system in the sense that it deduces new theorems in a forward directed way rather than making conjectures. However, for finding inductive theorems, MATHsAiD also uses some synthesis based techniques. Unlike IsaCoSy, which generates whole terms at once and then discards most after counter-example checking, MATHsAiD first produces a set of potential left-hand sides, called *terms of interest*. Smaller terms of interest can be used to build larger ones. The generation of interesting terms is guided by heuristics, which include rules for producing terms about associativity, commutativity and distributivity for relevant functions. Our system implements a similar idea in the pre-processing step which searches for AC-properties. However, we do not currently employ any special heuristics to look for distributivity rules, which might further decrease our the size of the synthesis space.

After generating the terms of interest, MATHsAiD proceeds to inductively generate theorems by replacing a variable in the term with 'TWO' (corresponding to $Suc(Suc\,0)$ for natural numbers or $[a,\ b]$ for lists) and reasoning forward to find an appropriate right hand side of the equation. This forward reasoning may have a large search space, which is why MATHsAiD imposes a size limit on potential right-hand sides, computed as *(number of function-symbols in LHS) + number of function symbols in 'TWO' + 2*. IsaCoSy's use of ordering commutativity also imposes restrictions on the size of left and right-hand sides, requiring the left-hand side to be larger than or equal to the right-hand side. However, while IsaCoSy's restriction is more general - applying to any commutative statement, it is less strict for equality. Thus it may generate equations with big differences in the size of the left-and right-hand side.

As IsaCoSy works on Isabelle-theories, properties such as well-definedness of functions are proved automatically by Isabelle's function package. This makes it easy to apply it to different theories. We have experimented with natural numbers, lists and binary trees. MATHsAiD cannot be extended to new domains so easily, as much of the configuration has to be done by adding axioms by hand.

MATHsAiD has been applied to the domain of natural numbers with addition and multiplication [18]. It generates the common associativity, commutativity and dis-

tributivity theorems which IsaCoSy also finds (although MATHsAiD produces fewer variants of distributivity). It also produces the following three extra theorems:

$$a + (Suc\ 0) = Suc\ a, \quad a * (Suc\ 0) = a, \quad (Suc\ 0) * a = a$$

IsaCoSy does not generate these theorems as they are subsumed by more general ones. MATHsAiD was designed to aid human mathematicians and has specially designed heuristic for what an 'interesting' theorem is, which includes the above identities about '*Suc* 0'.

MATHsAiD is considerably faster than IsaCoSy, the theorems for the natural number theory are generated in just 84 seconds. This is not surprising as MATHsAiD has more heuristics encoded, including a specific heuristic for distributivity theorems. However, MATHsAiD has not been applied to higher-order theories (such as lists with *map* and *fold*). Our system can deal with such theories without modifications. The use of IsaPlanner allows IsaCoSy to prove harder theorems than MATHsAiD, including higher-order ones. In particular, IsaPlanner uses a lemma calculation critic to prove theorems that MATHsAiD has to return to later, once an appropriate lemma has been generated.

13 Conclusions

Automating theory formation is a challenging problem. We have developed a program for inductive theory formation by conjecture synthesis, called IsaCoSy. To make synthesis computationally feasible, we turn rewriting upside down, and only allow the synthesis of terms that do not contain subterms that can be rewritten by a set of rewrite rules. This is enforced by generating constraints from terms assumed to be the left hand side of a rewrite rule. The constraints restrict the context in which constants and variables are allowed to occur. We also use the size of terms to restrict the number of commutative variants produced. Additional heuristics to further constrain synthesis by restricting the number and location of free variables were also presented.

Once conjectures have been produced, the algorithm filters out non-theorems by counter-example checking and passes the remaining conjectures to IsaPlanner which then tries to prove them. As new theorems are derived, more constraints are generated to restrict further synthesis.

IsaCoSy has been evaluated on several inductive theories about natural numbers, lists and binary trees. Interestingly, we found that the run-times are proportional to the number of false conjectures synthesised, as the majority of time is spent in counter-example checking.

We showed that the system produces 'good' theorems in the sense that it produces most of the theorems found in the corresponding Isabelle libraries. To evaluate the quality of theorems found by IsaCoSy, we compared them with those in Isabelle's libraries. IsaCoSy produces many of the theorems, resulting in high recall of 83% for natural numbers and 100% for lists. However, it produces a number of less interesting theorems too, so precision is lower, 63% for natural numbers and 38% for lists. The extra theorems are mostly variants of distributivity laws and can be removed without significant difficulty by hand.

We also verified the hypothesis that IsaCoSy is exponentially more efficient than a naive version of synthesis, but it is still exponential. This was verified by comparing IsaCoSy to a naive version of synthesis on several different inductive theories. IsaCoSy

is not only faster, but also able to explore larger term-sizes before running out of memory.

Furthermore, the performance of synthesis can be significantly improved by allowing conjectures that IsaPlanner is not able to prove, but which counter-example finding also cannot disprove, to generate constraints. For the equational theories considered, this optimisation still produces all the 'good' theorems, as Isabelle's counter-example finder rarely misses a non-theorem. We also showed that restricting nesting of polymorphic datatypes, such as lists, can also give large benefits. A range of other improvements were also suggested as further work.

Given the positive experimental results, and the range of further improvements, conjecture synthesis seems a promising approach to automated theory exploration, with important applications to automated reasoning.

Acknowledgements. The main part of this work was done while the first author was at the University of Edinburgh. It was funded by EPSRC grants EPE/005713/1 and EP/P501407/1. The reserach has also been supported by grant 2007-9E5KM8 of the Ministero dell'Istruzione Università e Ricerca.

A Function Definitions

A.1 Natural Numbers

$$\text{datatype} \;\; nat = 0 \mid \text{Suc } of \; nat$$

fun $+$ :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{add-zero:} \quad 0 + y = y$$
$$\text{add-suc:} \quad (Suc\ x) + y = Suc(x + y)$$

fun $*$ :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{mult-zero:} \quad 0 * y = 0$$
$$\text{mult-suc:} \quad (Suc\ x) * y = y + (x * y)$$

fun exp :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{exp-zero:} \quad x^0 = Suc\ 0$$
$$\text{exp-suc:} \quad x^{Suc\ y} = x * x^y$$

fun max :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{max-zero:} \quad max\ 0\ y = y$$
$$\text{max-suc:} \quad max\ (Suc\ x)\ y = case\ y\ of\ 0 \Rightarrow (Suc\ x) \mid Suc\ z \Rightarrow Suc(max\ z\ y)$$

A.2 Lists

$$\text{datatype } \alpha\ list = [\,] \mid \#\ of\ \alpha * \alpha\ list$$

fun @ :: $\alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

$$\text{app-nil:} \quad [\,]\ @\ l = l$$
$$\text{app-cons:} \quad (h \# t)\ @\ l = h \# (t\ @\ l)$$

fun len :: $\alpha\ list \Rightarrow nat$

$$\text{len-nil:} \quad len\ [\,] = 0$$
$$\text{len-cons:} \quad len\ (h \# t) = Suc\ (len\ t)$$

fun rev :: $\alpha\ list \Rightarrow \alpha\ list$

$$\text{rev-nil:} \quad rev\ [\,] = [\,]$$
$$\text{rev-cons:} \quad rev(h \# t) = (rev\ t)\ @\ [h]$$

fun $qrev$ :: $\alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

$$\text{qrev-nil:} \quad qrev\ [\,]\ l = l$$
$$\text{qrev-cons:} \quad qrev(h \# t)\ l = qrev\ t\ (h \# l)$$

fun map :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$

$$\text{map-nil:} \quad map\ f\ [\,] = [\,]$$
$$\text{map-cons:} \quad map\ f\ (h \# t) = (f\ h) \# t$$

fun $foldl$ :: $(\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \beta\ list \Rightarrow \alpha$

$$\text{foldl-nil:} \quad foldl\ f\ a\ [\,] = a$$
$$\text{foldl-cons:} \quad foldl\ f\ a\ (h \# t) = foldl\ f\ (f\ a\ h)\ t$$

fun $foldr$ :: $(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta \Rightarrow \beta$

$$\text{foldr-nil:} \quad foldr\ f\ [\,]\ a = a$$
$$\text{foldr-cons:} \quad foldr\ f\ (h \# t)\ a = f\ h\ (foldr\ f\ t\ a)$$

A.3 Binary Trees

$$\text{datatype } \alpha\ Tree = \text{Leaf} \mid \text{Node } of\ \alpha\ Tree * \alpha\ * \alpha\ Tree$$

fun $mirror\ ::\ \alpha\ Tree \Rightarrow \alpha\ Tree$

mirror-leaf: $mirror\ Leaf = Leaf$
mirror-node: $mirror\ (Node\ l\ data\ r) = Node\ (mirror\ r)\ data\ (mirror\ l)$

fun $nodes\ ::\ \alpha\ Tree \Rightarrow\ nat$

nodes-leaf: $nodes\ Leaf = 0$
nodes-node: $nodes\ (Node\ l\ data\ r) = (Suc\ 0)\ +\ (nodes\ l)\ +\ (nodes\ r)$

fun $height\ ::\ \alpha\ Tree \Rightarrow\ nat$

height-leaf: $height\ Leaf = 0$
height-node: $height\ (Node\ l\ data\ r) = Suc(max\ (height\ l)\ (height\ r))$

B Experimental Results for Conjecture Synthesis

The tables below show the theorems synthesised by IsaCoSy for the six evaluation theories about natural numbers, lists and binary trees. The theorems marked 'Pre-Processing' in the tables below have been discovered by IsaCoSy's heuristic which attempts to find associativity and commutativity properties prior to synthesis.

Label	Size	Theorem
N1*	Pre-processing	$a + 0 = a$
N2*	Pre-processing	$a + Suc\, b = Suc(a + b)$
N3*	Pre-processing	$a * 0 = 0$
N4*	Pre-processing	$a * Suc\, b = a + (a * b)$
N5*	Pre-processing	$a + b = b + a$
N6*	Pre-processing	$a * b = b * a$
N7*	Pre-processing	$(a + b) + c = a + (b + c)$
N8*	Pre-processing	$(a * b) * c = a * (b * c)$
N9*	13	$(a * b) + (c * b) = (a + c) * b$
N10	13	$(a * b) + (c * a) = (b + c) * a$
N11	13	$(a * b) + (c * a) = (c + b) * a$
N12	13	$(a * b) + (c * b) = (c + a) * b$
N13*	13	$(a * b) + (a * c) = (b + c) * a$
N14	13	$(a * b) + (a * c) = (c + b) * a$
N15	13	$(a * b) + (b * c) = (a + c) * b$
N16	13	$(a * b) + (b * c) = (c + a) * b$

Table 3 Theorems discovered about addition and multiplication on the natural numbers. Theorems marked with * are included in Isabelle's library for natural numbers. Note that Isabelle has the equations orientated in the opposite direction for N9 and N13. N13 is furthermore commuted over multiplication, e.g. the RHS of N13 is $(b + c) * a$, while in Isabelle it is $a * (b + c)$.

Label	Size	Theorem
T1	Pre-processing	$max\ a\ b = max\ b\ a$
T2	Pre-processing	$max\ (max\ a\ b)\ c = max\ a\ (max\ b\ c)$
T3	5	$max\ a\ a = a$
T4	5	$mirror\ (mirror\ a) = a$
T5	6	$nodes\ (mirror\ a) = nodes\ a$
T6	6	$height\ (mirror\ a) = height\ a$

Table 4 Theorems found about binary trees with functions max, mirror, nodes and height. Isabelle has no library for binary trees.

Label	Size	Theorem
L1*	Pre-processing	$(a\ @\ b)\ @\ c = a\ @\ (b\ @\ c)$
L2*	5	$rev(rev\ a) = a$
L3*	5	$a\ @\ [\,] = a$
L4*	6	$len(rev\ a) = len\ a$
L5	9	$len(a\ @\ b) = len(b\ @\ a)$
L6	10	$rev(a\ @\ (rev\ b)) = b\ @\ (rev\ a)$
L7*	10	$(rev\ a)\ @\ (rev\ b) = rev\ (b\ @\ a)$
L8	10	$rev((rev\ a)\ @\ (rev\ b)) = b\ @\ a$
L9	11	$rev(a\ @\ [b]) = b\ \#\ (rev\ a)$
L10	11	$rev((rev\ a)\ @\ [b]) = b\ \#\ a$
L11	14	$rev(a\ @\ (b\ @\ (rev\ c))) = c\ @\ (rev(a\ @\ b))$
L12	14	$rev(a\ @\ (b\ \#\ (rev\ c))) = c\ @\ (b\ \#\ (rev\ a))$
L13	14	$rev((rev\ a)\ @\ (b\ \#\ c)) = (rev\ c)\ @\ (b\ \#\ a)$
L14	14	$(rev\ a)\ @\ ((rev\ b)\ @\ c) = (rev\ (b\ @\ a))\ @\ c$
L15	14	$(rev\ a)\ @\ (b\ \#\ (rev\ c)) = rev(c\ @\ (b\ \#\ a))$
L16	14	$rev((rev\ a)\ @\ b)\ @\ c = (rev\ b)\ @\ (a\ @\ c)$
L17	14	$rev((rev\ a)\ @\ (b\ \#\ (rev\ c))) = c\ @\ (b\ \#\ a)$
L18	14	$a\ @\ (rev((rev\ b)\ @\ c)) = a\ @\ ((rev\ c)\ @\ b)$

Table 5 Theorems discovered about append, reverse and length on lists. Theorems marked with * are included in Isabelle's list library. Note that L18 is allowed to be synthesised as its simpler version, $rev(rev\ b)\ @\ c = (rev\ c)\ @\ b$, could not be proved when it was first synthesised, and thus did not generate constraints. Its proof require L7 as a lemma, which was not yet available. The lemma is however available when attempting to prove L18, so this succeeds.

Label	Size	Theorem
L19	9	$map\ a\ (rev\ b) = rev(map\ a\ b)$
L20*	9	$rev(map\ a\ b) = map\ a(rev\ b)$
L21	9	$rev(map\ a\ (rev\ b)) = map\ a\ b$
L22*	13	$(map\ a\ b)\ @\ (map\ a\ c) = map\ a\ (b\ @\ c)$

Table 6 Theorems discovered about append, reverse and map on lists. The theorems about append and reverse from table 5 (theorems L1 - L3 and L6 - L18) were also discovered. Theorems marked with * are included in Isabelle's list library.

Label	Size	Theorem
L42*	14	*foldl a (foldl a b c) d = foldl a b (c @ d)*
L43*	14	*foldr a b (foldr a c d) = foldr a (b @ c) d*

Table 7 Theorems discovered about foldl and foldr. Both theorems are included in Isabelle's list library. The theorems about append and reverse from table 5 (theorems L1 - L3 and L6 - L18) were also discovered.

Label	Size	Theorem
L23	8	$qrev\ (rev\ a)\ b = a\ @\ b$
L24	8	$(rev\ a)\ @\ b = qrev\ a\ b$
L25	9	$qrev\ (qrev\ a\ b)[\,] = qrev\ b\ a$
L26	11	$qrev\ a\ (qrev\ b\ c) = qrev\ (b\ @\ a)\ c$
L27	11	$qrev\ a\ (b\ @\ c) = qrev\ (qrev\ b\ a)\ c$
L28	11	$qrev\ a\ (b\ @\ c) = (qrev\ a\ b)\ @\ c$
L29	11	$qrev\ (qrev\ a\ b)\ c = qrev\ b\ (a\ @\ c)$
L30	11	$qrev\ (a\ @\ b)\ c = qrev\ b\ (qrev\ a\ c)$
L31	11	$(qrev\ a\ b)\ @\ c = qrev\ a\ (b\ @\ c)$
L32	12	$rev\ (qrev\ a\ (b\ \#\ c)) = qrev\ c\ (b\ \#\ a)$
L33	12	$a\ @\ (rev\ (qrev\ b\ [\,])) = rev\ (qrev\ b\ (rev\ a))$
L34	13	$rev\ (a\ @\ (b\ @\ c)) = qrev\ c\ (rev\ (a\ @\ b))$
L35	13	$rev\ (a\ @\ (b\ \#\ c)) = qrev\ c\ (b\ \#\ (rev\ a))$
L36	13	$qrev\ a\ (rev\ (b\ @\ c)) = rev\ (b\ @\ (c\ @\ a))$
L37	13	$qrev\ a\ (b\ \#\ (rev\ c)) = rev\ (c\ @\ (b\ \#\ a))$
L38	13	$rev\ (qrev\ a\ (rev\ (b\ @\ c))) = b\ @\ (c\ @\ a)$
L39	13	$a\ @\ (rev\ (b\ @\ c)) = a\ @\ qrev\ c\ (rev\ b)$
L40	13	$a\ @\ qrev\ b\ (rev\ c) = a\ @\ (rev\ (c\ @\ b))$
L41	13	$a\ @\ rev\ (qrev\ b\ (rev\ c)) = a\ @\ (c\ @\ b)$

Table 8 Theorems discovered about append, reverse and qrev on lists. The theorems about append and reverse from table 5 (theorems L1 - L3 and L6 - L18) were also discovered. The qrev-function is not defined in Isabelle's list library, so no comparison can be made. Note that theorems L39 - L41 are allowed to be synthesised as their simpler versions (excluding the $a\ @ \ldots$ on both sides) could not be proved. However, other proofs will later discover the required lemmas by lemma calculation. These lemmas are stored, so IsaPlanner will be able to re-use them to prove L39 - L41.

References

1. S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 230–239, Washington, DC, USA, 2004. IEEE Computer Society.
2. B. Buchberger, A. Craciun, T Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkrantz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction*, pages 111–120, 1988.
4. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
5. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1992.
6. S. Colton. *Automated Theory formation in pure mathematics*. Springer-Verlag, 2002.
7. N. Dershowitz. Orderings for term-rewriting systems. In *20th Annual Symposium on Foundations of Computer Science*, pages 123–131, 1979.
8. L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.
9. L. Dixon and J. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, pages 279–283, 2003.
10. L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *Proceedings of TPHOLs'04*, pages 83–98, 2004.
11. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
12. M. Hodorog and A. Craciun. Scheme-based systematic exploration of natural numbers. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 26–34, 2006.
13. M. Johansson, L. Dixon, and A. Bundy. Properties of IsaCoSy's constraint generation algorithm. Available online: http://homepages.inf.ed.ac.uk/v1mjoha1/IsaCoSyProperties.pdf.
14. D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, 1967.
15. D.B. Lenat. AM: Discovery in mathematics as heuristic search. In D. Heiberg and D.A. Damstra, editors, *Knoledge-based systems in Artificial Intelligence*, chapter 1. McGraw-Hill, 1982.
16. E. Maclean, A. Ireland, R. Atkey, and L. Dixon. Refinement and term synthesis in loop invariant generation. In *WING 09: Workshop on Invariant Generation*, 2009.
17. R. McCasland and A. Bundy. MATHsAiD: a mathematical theorem discovery tool. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 17–22. IEEE Computer Society Press, 2006.
18. R. McCasland, A. Bundy, and S. Autexier. Automated discovery of inductive theorems. *Special Issue of Studies in Logic, Grammar and Rhetoric on Computer Reconstruction of the Body of Mathematics: From Insight to Proof: Festschrift in Honor of A. Trybulec*, 10(23):135–149, 2007.
19. R. Monroy-Borja. The use of abduction to correct faulty conjectures. Master's thesis, University of Edinburgh, 1993.
20. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
21. L. C. Paulson. Isabelle: The next seven hundred theorem provers. In *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 772–773. Springer-Verlag, 1988.
22. Y. Puzis, Y. Gao, and G. Sutcliffe. Automated generation of interesting theorems. In *Proceedings of the 19th International FLAIRS Conference*, pages 49–54, 2006.
23. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.