A Calculus for Conjecture Synthesis

Moa Johansson[1], Lucas Dixon[2], and Alan Bundy[2]

[1] Dipartimento di Informatica, Università degli Studi di Verona **
[2] School of Informatics, University of Edinburgh
moakristin.johansson@univr.it, {l.dixon, a.bundy}@ed.ac.uk

Abstract. IsaCoSy is a theory formation system which synthesises and proves conjectures in order to produce a background theory for a new formalisation within a proof assistant. The key idea we employ to make synthesis tractable is to only consider synthesis of terms that are not more complex versions of already known terms. IsaCoSy identifies such undesirable terms as those that match the left-hand sides of rewrite rules. In this paper, we slightly generalise this idea to present a formal language for constraining synthesis such that it does not construct terms that can be matched by a given set of constraint-terms. We give a mathematical account of the algorithms involved, and prove their correctness. In particular, we prove the correctness property for IsaCoSy's approach to synthesis: when given a set of rewrite rules as input, it only produces irreducible terms.

1 Introduction

IsaCoSy is an automated theory formation system for inductive theories [7]. It takes as input a set of constants to be used in synthesis and a set of known terms, called *constraint-terms*, which we want to avoid synthesising 'more complex' variants of. The key idea IsaCoSy uses, to make the synthesis process tractable and the resulting conjectures interesting, is that synthesis is constrained to only construct terms that are not matched by any term in a set of constraint terms. IsaCoSy builds progressively larger conjectures, starting from a given top-level symbol. The system then passes synthesised conjectures to a counter-example checker [1], which filters out obviously false statements. The remaining conjectures are given to the automatic inductive prover in IsaPlanner [4]. Any theorems found may be used to generate additional constraints on the synthesis process and improve proof automation.

IsaCoSy has been applied to generate equations in inductive theories, with the aim of producing results that can be used as intermediate lemmas within a user's, or a proof tool's, subsequent attempts to prove more involved theorems. The implementation and evaluation of IsaCoSy has been described in [7]. IsaCoSy was shown to reduce the size of the synthesis search space by an exponential factor with respect to a naive generate-and-test style algorithm. IsaCoSy was furthermore able to generate most of the relevant inductive lemmas occurring in

** This research was funded by EPSRC grant EPE/005713/1

$a + b = b + a$	$a * b = b * a$
$(a + b) + c = a + (b + c)$	$(a * b) * c = a * (b * c)$
$(a * b) + (c * b) = (a + c) * b$	$(a * b) + (a * c) = (b + c) * a$
$rev(rev\ a) = a$	$(rev\ a)\ @\ (rev\ b) = rev\ (b\ @\ a)$
$rev(map\ a\ b) = map\ a(rev\ b)$	$(map\ a\ b)\ @\ (map\ a\ c) = map\ a\ (b\ @\ c)$

Table 1. Some examples of synthesised theorems about natural numbers and list. These all occur in Isabelle's library. The symbol @ denote append.

Isabelle's libraries for natural number and lists[3], which have been hand-created by a human user. Library-lemmas missed out by IsaCoSy could typically be derived easily from ones that were generated. The number of additional theorems synthesised, not occurring in the libraries, was relatively small. A few sample theorems synthesised by IsaCoSy are shown in Table 1. The complete synthesised theories from these experiments are available online[4].

To complement the algorithmic description given in [7], we here present a higher-level, more succinct and general formal description of IsaCoSy's constraint generation and synthesis machinery. Using this account, we prove the fundamental correctness property for our system: it generates only terms in the language that are not instances of any term in the constraint-term set. In particular, when the constraint-term set consists of the left-hand sides of a set of rewrite rules, we show that only irreducible terms are synthesised.

Previously, the inputs to IsaCoSy were a set of datatypes and function symbols, with initial constraints generated from the left-hand sides of rewrite rules about these. While, in this case, the constraints force IsaCoSy to only generate irreducible terms, our generalisation to employing arbitrary constraint-terms allows some additional heuristics. For instance, a sequence of two or more successor symbols can be disallowed by introducing a constraint-term of the form $Suc(Suc(x))$. The generalised formalisation also highlighted some redundancies in the previous version of the language used to express constraints. We plan to implement our revised constraint language in the next version of IsaCoSy. The generalised use of arbitrary terms from which to construct constraints has already been implemented and can be downloaded as part of IsaPlanner/IsaCoSy[5].

2 Related Work

Other theory-formation systems, such as MATHsAiD [9], IsaScheme [10] and Theorema [2, 5] have been applied to inductive theories. However, none of the algorithms used in those systems, or any other theory formation system that we know of, have yet enjoyed a formal analysis.

[3] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/index.html`

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

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

IsaScheme is a theory formation system which generates conjectures by instantiating a set of schemes, which are higher-order terms, by a given set of closed terms in all possible ways [10]. While IsaCoSy will consider all irreducible terms, IsaScheme further restricts its search space by only considering those that are instances of its given schemes instantiated by its input set of closed terms. IsaScheme orients candidate equations, and applies Knuth-Bendix completion to exclude any conjectures that are not valid rewrite rules. Candidate theorems thus form a rewrite system together with the initial background lemmas from the theory. Like IsaCoSy, it employs QuickCheck for counter-example checking, and IsaPlanner for proving remaining conjectures.

The purpose of the MATHsAiD system is to construct theorems that would be consider interesting by a human mathematician [8, 9]. It takes an axiomatic description of the initial theory as input and reason forward to derive logical consequences of these. A range of heuristic measures are then applied to restrict and filter out anything not deemed interesting.

QuickSpec is a tool for automatically deriving algebraic specifications for functional programs written in Haskell or Erlang [3]. Unlike IsaCoSy, QuickSpec simply generates all possible terms up to a given size, and then explore which ones are equal by testing, using a counter-example finder. It then employs filtering to discard equations that are derivable from the remaining set. QuickSpec is not connected to an inductive theorem prover, and thus cannot attempt to prove the conjectures it produces.

3 Background

3.1 Terms

For our purposes, it is convenient to define (possibly partially synthesised) terms as n-ary trees, captured by the following datatype[6]:

Definition 1 (Synthesis Terms)

$$\begin{aligned} \mathit{Atom} &:= \mathit{Const\ of\ k} \ \mid \ \mathit{Hole\ of\ ?h} \\ \mathit{Term} &:= \mathit{App\ of\ (Atom\ *\ Term\ list)} \end{aligned}$$

Atom is either a hole, representing part of a term still to be synthesised, denoted $?h$, or a named constant symbol k. The *App* constructor is used to represent function application with the *Term list* being the functions arguments. A term that consists of a constant, x, with no arguments is represented by $App(x, [\,])$.

During synthesis, only holes are allowed to be instantiated by substitutions as they represent term-positions still to be synthesised. Synthesis may insert variables bound by universal quantifiers in the term, but as these are not allowed to be further instantiated *during synthesis*, they are treated as constants.

[6] We have abstracted away type information as it adds no interesting complexity and clarifies the presentation.

IsaCoSy does not currently consider synthesising terms with lambda-abstractions. This is equivalent to function synthesis and would greatly increase the size of the search space.

We write $hd(t)$ to denote the symbol in the head position of a term, e.g. $hd(App(f, args)) = f$. We use σ to denote substitutions on terms. The symbols $=$ and $\neq$ on terms denote syntactic (dis)equality. We use $\equiv$ to represent instantiations of holes. In addition to this notation, we will use the following definitions.

Definition 2 (Ground Term) *A synthesised term is* ground *if it does not contain any holes.*

Definition 3 (Grouding Substitution) *A substitution σ is a* grounding substitution *on a synthesised term t, if $t\sigma$ is a ground term.*

3.2 Positions in Terms

Positions in terms are expressed as *paths*. These are lists of argument positions within a term, with the empty list being the top of the term. As an example, consider the term $f(x,\ g(y))$. We show a tree-representation of this term in Figure 1 with each position tagged by its path-representation.

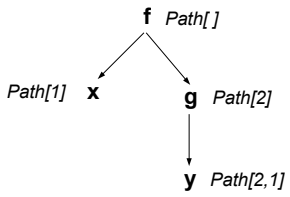

Fig. 1. Term-tree with path-representations of each position highlighted.

We write $t[s]_p$ for a term t with a subterm s in position defined by the path p. The term s can also be referred to by $t|_p$. We write $p_{[i,j]}$ for a path that has the path p_i as a prefix, and is extended by j, where j is an integer. In other words, $p_{[i,j]}$ is the position j immediately below p_i in the term tree. To append two paths to each other we write p_i @ p_j.

4 Overview of IsaCoSy

Figure 2 illustrates the synthesis procedure of IsaCoSy.

The initial input to configure IsaCoSy is a set of constant symbols, with which to synthesise new terms, and a set of constraint-terms. Typically the constants are those from the basic definitions in a theory. For instance, we give an example toy-theory for natural numbers in Figure 3; here the functions symbols are $+$, 0

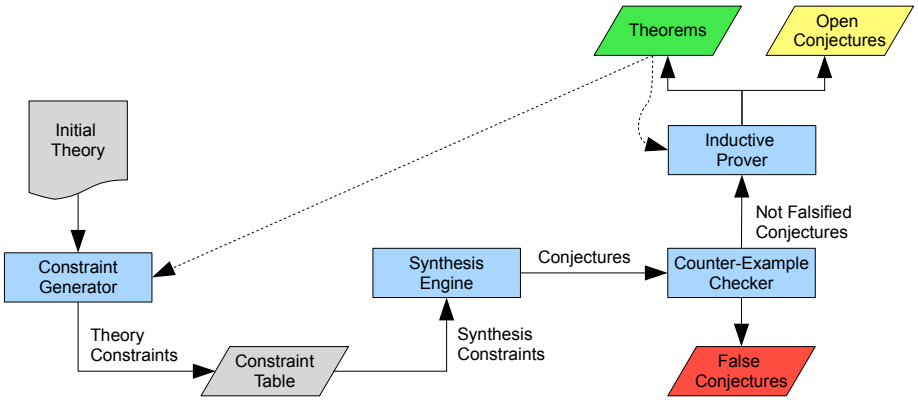

Fig. 2. IsaCoSy's synthesis process.

```
datatype Nat =
      0
   |  Suc of Nat

fun plus : Nat => Nat => Nat
where
       0 + y  =  y
    |  Suc x + y = Suc(x + y)

lemma Suc-Injective:
      (Suc n = Suc m) = (n = m)
```

Fig. 3. An example theory to which IsaCoSy can be applied. It contains the definition of a recursive datatype `Nat`, the definition of a function `plus` and an additional lemmas capturing the injectivity property of `Suc`. This lemma is derived automatically by Isabelle's definitional machinery for datatypes when the `Nat` type is declared.

and *Suc*. The initial constraints are typically generated from the left-hand sides of defining equations, as well as the left-hand side of any additional rewrite rules. In our example, this is the definition of + and the lemma `Suc-Injective`.

The constraint-terms are fed into IsaCoSy's constraint generation machinery, which computes a set of initial constraints for synthesis, referred to as *theory constraints*. Theory constraints are stored in a table, indexed by the head-symbol in the term that generated the constraint. Constraint generation is described in §5.1.

In addition to the constraints from rewrite rules, IsaCoSy may also constrain synthesis by ordering the arguments for functions that are commutative. We refer to [7] for more details about heuristics related to commutativity, as it is not the main focus of this paper.

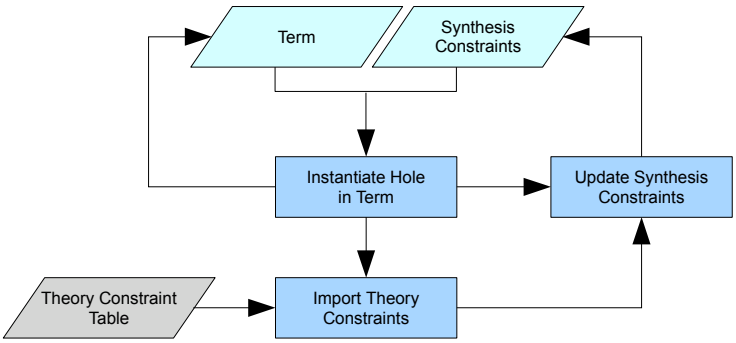

Fig. 4. The synthesis engine. While there are still open holes in the term, IsaCoSy picks a hole and a symbol to instantiate it with, in accordance with the constraints. New constraints for new holes are imported for the relevant symbol, and old constraints are updated to take the instantiation into account.

The input to the synthesis engine is a term containing *holes*, standing for the parts yet to be synthesised. At each step of synthesis, a hole is picked and instantiated with a symbol. If this is a function symbol, new holes are also introduced, corresponding to the arguments of the function. The symbol chosen to instantiate a hole is picked from the set of symbols allowed by the synthesis constraints.

During the synthesis process, IsaCoSy imports theory constraints for the constants which are used to instantiate a hole (typically these constants are function symbols). The set of constraints applicable to a particular synthesis attempt are referred to as *synthesis constraints*. Synthesis constraints are updated and modified as the term becomes instantiated, while theory constraints remains static. Figure 4 shows the steps of the synthesis engine in more detail. IsaCoSy will thus synthesise a set of terms adhering to the relevant constraints.

Example 1. Consider a partially synthesised term $?h_1 + ?h_2 = ?h_3$, and suppose we pick $?h_1$ to be instantiated next. The synthesis algorithm now has to pick a symbol to instantiate $?h_1$.

The synthesis constraints on the term will forbid picking 0 or Suc as either of these would produce a term that matches one of the two rewrite rules from the definition of addition in Figure 3.

The algorithm may however choose to instantiate $?h_1$ with $+$, resulting in an updated term with two new holes: $(?h_4 + ?h_5) + ?h_2 = ?h_3$. After instantiation, any new applicable constraints are imported from the theory constraints of the newly introduced symbol. Here, we import constraints about $+$, which will restrict instantiations of $?h_4$ and $?h_5$.

The synthesis algorithm is described in more detail in §6.

IsaCoSy divides synthesis into iterations, starting from a given smallest term size, and incrementally increasing the size. For each term size, a set of terms

are synthesised. After each iteration the set of synthesised terms are filtered through counter-example checking and then passed on to the prover. Any theorems proved are used to generate additional theory constraints, which can be used in the next iteration to further constrain the synthesis of larger terms. Proved theorems are also used by the prover in subsequent proofs. This makes the prover more powerful as more theorems are discovered.

5 Constraint Language

The purpose of the constraint language is to express restrictions on synthesis in order to avoid generating any terms that match known terms from the constraint-term set. For instance, when the constraints are generated from rewrite rules, no terms containing a redex matching a known rewrite rule should be synthesised. This keeps the synthesis search space size manageable and avoids the generation of more complex versions of already known theorems. The constraints specify which positions are not allowed to be instantiated to certain constants, as well as which positions are not allowed to be instantiated to equal terms. They may also restrict certain symbols to ensure that they do not occur in certain positions at the same time.

A term t *satisfies* a constraint c, if it cannot be unified with the term from which c was generated. In the case where the constraint was generated from the left-hand side of a rewrite rule, this corresponds to t not having a redex for that rule. Otherwise, we say that t *violates* the constraint. Definition (7) in §5.2 specifies the *Satisfies* relation.

Referring to positions as paths from the top of the term tree allows us to simplify and revise the constraint language compared to that presented in [7]. In previous work, the constraint language was unnecessarily complicated as the constraints were built as a tree-like structure reflecting the underlying term. The simplified definition for the constraint language is:

Definition 4 (Constraint Language)

$$Constr := NotConst(p, k) \mid UnEqual(p_1, \ldots, p_n) \mid c_1 \vee c_2 \mid \top \mid \bot$$

The *NotConst* constraints express that a constant symbol is not allowed to occur in position p. The *UnEqual* constraints specify a list of positions not allowed to be instantiated to equal terms. In addition, the language allows disjunctions of constraints, $c_1 \vee c_2$, and contains the two constant constraints $\top$ and $\bot$, which are trivially satisfied and violated respectively[7].

We also clarify the difference between *theory constraints* and *synthesis constraints*:

[7] The disjunction subsumes the *IfThen* and *NotSimult* constructors from [7]. Also note that *NotConst* subsumes both the constraints *NotAllowed* and *VarNotAllowed* in [7].

– Theory constraints are generic constraints associated with particular constant symbols. They arise from terms given to IsaCoSy's constraint generation algorithm.
– Synthesis constraints are associated with a particular synthesis attempt and are updated during the synthesis process, as the term is built.

To disambiguate, we write theory constraints with a subscript $_T$ and synthesis constraints with a subscript $_S$, e.g. $NotConst_T$ and $NotConst_S$.

5.1 Constructing Constraints

Suppose we want to generate the constraints from a term t:

– For each position p in t containing a constant symbol k we produce a constraint $NotConst_T(p,\ k)$.
– If there are several distinct positions $p_j, \ldots\ p_m$ in t, that contain the same variable, we produce a constraint $UnEqual_T(p_j, \ldots, p_m)$.

A term will often give rise to a set of constraints for different positions. We call these *dependent constraints*. They express instantiations not simultaneously allowed. Synthesis may violate some of those constraints, but not all of them, so the final step of constraint generation is to create a disjunction of all the constraints for the rule. This is formally expressed as:

Definition 5 (Theory Constraints for a term t)

$$\begin{aligned} &ThyConstrs(t) := \\ &\qquad \bigvee \big(\{NotConst_T(p,k) \ \mid\ t|_p = k \wedge IsConst(k)\} \cup \\ &\{UnEqual_T(p_j, \ldots, p_m) \ \mid\ t|_{p_j} = \ldots = t|_{p_m} \wedge p_j \neq p_m \\ &\qquad\qquad \wedge IsVar(t|_{p_j}) \wedge \ldots \wedge IsVar(t|_{p_m})\}\big) \end{aligned}$$

Here we let $\bigvee$ stand for the disjunction of the constraints in a given set. The predicate *IsConst*, is true iff a term is a constant, while *IsVar* is true iff the term is a variable.

Example 2. Suppose IsaCoSy generates a constraint from the term $0 + y$ (from the LHS of the rewrite rule $0 + y = y$, in the definition of addition in Figure 3). As there are no variables occurring more than once, IsaCoSy generates the constraint:

$$NotConst_T(Path[\,], +) \vee NotConst_T(Path[1], 0)$$

This specifies that is a position contains the symbol +, then it is forbidden to instantiate the first argument of + to be 0. A similar constraint is generated for the *Suc*-case.

Example 3. As a slightly more complex example, consider a term $f(x,\ g(x))$. The position $Path[\,]$ contains the symbol f, while position $Path[2]$ contains the symbol g and positions $Path[1]$ and $[2,1]$ both contain the variable x. This produce the constraint:

$$NotConst_T(Path[\,], f) \vee NotConst_T(Path[2], g) \vee UnEqual_T(Path[1],\ Path[2,1])$$

The theory constraints are stored in mapping from a function symbol f to sets of constraints where f occurs in the head position (i.e. where it has a theory constraint with $Path[\,]$).

Definition 6 (Theory constraints of a function f)

$$ThyConstrs(f) = \{c \mid NotConst_T(Path[\,], f) \in c\}$$

We use the notation $NotConst_T(Path[\,], f) \in c$ to specify any constraint where $NotConst_T(Path[\,], f)$ is one of the disjuncts in c.

5.2 Semantics of Constraints

We define a function $Satisfies(t,\ c)$ defined below, which takes a ground term t and a constraint c and returns returns *True* iff the term *satisfies* the constraint. Otherwise t *violates* the constraint. Recall that as t is ground it does not contain any holes.

Definition 7 (Semantics for Constraints) *The Satisfies function is defined below for the constructs of the constraint language:*

NotConst:

$$Satisfies(t,\ NotConst(p,k)) \Longrightarrow t|_p \neq k$$

UnEqual:

$$Satisfies(t, UnEqual(p_1, \ldots, p_n)) \Longrightarrow \\ \forall i \in \{1 \ldots n\}.\ t|_{p_1} \neq t|_{p_i}\ \vee \ldots \vee\ t|_{p_n} \neq t|_{p_i}$$

Or:

$$Satisfies(t,\ c_1 \vee c_2) \Longrightarrow Satisfies(t,\ c_1) \vee Satisfies(t,\ c_2)$$

Top:

$$Satisfies(t,\ \top) \Longrightarrow True$$

Bottom:

$$Satisfies(t,\ \bot) \Longrightarrow False$$

If the constraint refers to paths longer than is possible in t, the constraint is trivially satisfied.

The constraint update mechanism (see Def. (9) in §7), is a lazy unfolding of *Satisfies*, operating over the terms in the process of being synthesised, which may contain holes.

5.3 Correctness of the Constraint Generation Algorithm

We will now prove the constraint generation mechanism is correct, in the sense that it produces exactly those constraints which exclude terms matching any of those in the constraint-term set. In the case of constraints from rewrite rules, this means excluding any reducible terms.

The correctness properties below were stated in [7], but not proved. Using the *Satisfies* function (Def. 7) allows us to prove this theorem. Our correctness proof consists of two parts. We first show the *sufficient coverage* property: that the constraints generated cover all instances of the term they were generated from. We then show that the constraints only correspond to the terms they were generated from, the *no over-coverage* property.

We refer to terms from which constraints have been generated as *constraint terms* and use the notation $\mathit{Constraints}(l)$ for the disjunction of constraints the algorithm generates for the term l. We say that a term s is an *instance* of l if there is a substitution σ such that $s = l\sigma$.

Lemma 1 (Sufficent coverage). *Given a term t and a constraint-term l, if t contains a subterm s, which is an instance of l, then t violates $\mathit{Constraints}(l)$.*

Proof. $\mathit{Constraints}(l)$ is a disjunction: $c_1 \vee \ldots \vee c_n$. The constraint is violated when $\mathit{Satisfies}(s,\ c_1 \vee \ldots \vee c_n)$ evaluates to false.

There are two cases, depending on the type of each disjunct:

NotConst: By construction, each position p_i in l containing a constant symbol k, will have contributed a constraint $\mathit{NotConst}_T(p_i,\ k)$. However, as s is assumed to be an instance of l, the position p_i in s must contain k, or else $s \neq l\sigma$.
Hence, $\mathit{Satisfies}(s,\ \mathit{NotConst}_T(p_i,\ k))$ evaluates to false for all disjuncts that are $\mathit{NotConst}_T$ constraints.

UnEqual: By construction, each set of positions $p_j, \ldots p_m$ in l containing the same variable x, will contribute a constraint:

$$\mathit{UnEqual}_T(p_j, \ldots, p_m)$$

By assumption $s = l\sigma$ and the substitution σ must map the variable x to the same term everywhere it occurs in l, namely the term represented by the sub-trees starting at $p_j, \ldots,\ p_m$ in s, which must be identical.
By the semantics for $\mathit{UnEqual}_T$ in definition (7):

$$\mathit{Satisfies}(s,\ \mathit{UnEqual}_T(p_j, \ldots,\ p_m))$$

will evaluate to false when the sub-trees rooted at $s|_{p_j}, \ldots s|_{p_m}$ are identical.

Thus $\mathit{Satisfies}(s,\ c_1 \vee \ldots \vee c_n)$ evaluates to false, as s violates$\mathit{Constraints}(l)$. Hence also $t[s]$ violates the constraint.

Lemma 2 (No over-coverage). *Given a constraint-term l, if t is a term that violates $\mathit{Constraints}(l)$, then there is a subterm in t that is an instance of l.*

Proof. By contradiction, assume no subterm of t is an instance of l. $\mathit{Constraints}(l)$ is a disjunction: $c_1 \vee \ldots \vee c_n$. As t violates the constraints, we know there must exist a subterm $t[s]$, such that $\mathit{Satisfies}(s,\ c_1 \vee \ldots \vee c_n) \Longrightarrow \mathit{False}$. By definition (7), we hence have $\mathit{Satisfies}(s,\ c_i) \Longrightarrow \mathit{False}$ for each c_i, $1 \leq i \leq n$. We have two cases, depending on the type of each c_i:

NotConst: By construction, each position p_i in l containing a constant symbol k, will have contributed a constraint $\mathit{NotConst}_T(p_i,\ k)$. We know that $\mathit{Satisfies}(s,\ \mathit{NotConst}_T(p_i,\ k)) = \mathit{False}$, so we must have $s|_{p_i} = k$ for each position p_i. Hence s and l contain the same constant symbols in the same positions.

UnEqual: By construction, all position $p_j \ldots p_m$ containing the same variable x in l, will have contributed a constraint

$$\mathit{UnEqual}_T(p_j, \ldots, p_m)$$

As $\mathit{Satisfies}(\mathit{UnEqual}_S(s,\ p_j, \ldots, p_m) \Longrightarrow \mathit{False}$, s must contain identical subterms $s|p_j = \ldots = s|p_m$. Hence there exist a substitution σ such that $s = l\sigma$ where $\sigma\{x \mapsto s|_{p_j}\}$.

As s and l agree on all positions of constant symbols, and we can find a substitution for the variables in l with subterms of s, then s is an instance of l, contradicting our assumption. Hence, t violates $\mathit{Constraints}(l)$ whenever it has a subterm s which is an instance of l.

Theorem 1 (Exact coverage). *Given a term t and a constraint-term l, the constraint produced by the constraint generation algorithm is satisfied by t iff there is no subterm within t that matches l.*

Proof. Follows from lemmas 1 and 2.

6 The Synthesis Algorithm

When synthesising a term, IsaCoSy picks an open hole and explores all instantiations adhering to the constraints. The synthesis algorithm applies the inference rules specified in Def. (8) to a partially synthesised term, t, that contains some uninstantiated hole $?h$.

In addition to a partially synthesised term, the synthesis rules refer to a collection of synthesis constraints, C, associated with t, denoted by $C \parallel t$. As described above, each constraint $c \in C$ may be a disjunction, forbidding the combination of instantiations that would render a particular rewrite rule applicable to a subterm of t. We write C_h for the constraints in C which contain a reference to the (position of) hole $?h$.

Definition 8 (Syntesis Algorithm) *The synthesis algorithm instantiate holes by the following two rules:*

Function: $?h \equiv f(?h_1 \ldots ?h_n)$

$$\frac{C \parallel t[?h]_{p_i}}{(C \mapsto (\forall\ c \in C_h.\ \mathit{Update}(c))) \ \cup\ \mathit{Constrs}(f) \parallel t[f(?h_1 \ldots ?h_n)]_{p_i}}$$

$$\mathit{if} \begin{cases} f \in \mathit{Dom}(?h) \\ \mathit{NotConst}_S(?h, f) \ \notin C_h \\ \mathit{Constrs}(f) = \{c \mid c \in \mathit{ThyConstrs}(f) \wedge \forall p_j \in c.\ p_j \mapsto p_i \ @\ p_j\} \end{cases}$$

Constant: $?h \equiv k$

$$\frac{C \parallel t[?h]_{p_i}}{C \mapsto (\forall\ c \in C_h.\ \mathit{Update}(c)) \parallel t[k]_{p_i}} \qquad \mathit{if}\{\ \mathit{NotConst}_S(?h, k) \notin C_h$$

The function *Update*, used in the rules above, takes a constraint (which might be a conjunction of dependent constraints) on the instantiated hole and updates it according to the constraint update algorithm (see Def. 9). We use $\mathit{Constrs}(f)$ for the new constraints that are introduced on new holes arising from instantiating some hole with a function symbol f. These come from the theory constraints associated with f (see Def. (6)).

All paths defining positions in a theory constraint are prefixed by the path to the position of the newly instantiated hole to construct the new synthesis constraints. This captures that the new constraints apply to a subterm of the whole synthesised term, rooted at the position of the newly instantiated hole.

We use $\mathit{Dom}(?h)$ as the set from which synthesis selects candidate instantiations of a compatible type for a hole $?h$. Synthesis tries all instantiations that are not forbidden by the presence of a singleton $\mathit{NotConst}_S$ constraint[8].

The correctness of the synthesis algorithm is that it maintains the invariant that, after each step where a hole is instantiated, no active constraint in C is violated (see lemma 3).

7 Constraint Update Algorithm

After each instantiation during synthesis, the constraints associated with the term must be updated to reflect any new holes created, and propagate existing constraints onto these. The function *Update* is a lazy unfolding of the *Satisfies* relation. We write p_h for the position of the instantiated hole $?h$. The *Update* function is defined as follows:

Definition 9 (Constraint Update Function)

NotConst-violation: $?h \equiv s,\ hd(s) = k.$

$$\mathit{Update}(\mathit{NotConst}_S(p_h,\ k)) \Longrightarrow \bot$$

[8] In the implementation, constant symbols occurring in singleton constraints are in fact removed from the domain of the relevant hole, but for the purpose of clarity the constraints have been made explicit in here.

NotConst-satisfied: $?h \equiv s,\ hd(s) \neq k.$

$$Update(NotConst_S(p_h,\ k)) \Longrightarrow \top$$

UnEqual-Fun: $?h \equiv f(?h_1 \ldots ?h_m)$

$$\begin{aligned} &Update(UnEqual_S(p_h,\ p_1, \ldots, p_n)) \Longrightarrow \\ &\qquad NotConst_S(p_1,\ f) \vee \ldots \vee NotConst_S(p_n,\ f) \vee \\ &UnEqual_S(p_{h_1},\ p_{[1,\ 1]}, \ldots p_{[n,\ 1]}) \vee \ldots \vee UnEqual_S(p_{h_m},\ p_{[1,\ m]}, \ldots p_{[n,\ m]}) \end{aligned}$$

UnEqual-Const: $?h \equiv k$

$$\begin{aligned} &Update(UnEqual_S(p_h,\ p_1, \ldots, p_n)) \Longrightarrow \\ &\qquad NotConst_S(p_1,\ k) \vee \ldots \vee NotConst_S(p_n,\ k) \end{aligned}$$

Or:

$$Update(c_1 \vee c_2) = Update(c_1) \vee Update(c_2)$$

The correctness of the constraint update machinery is crucial to the efficiency and correctness of the entire synthesis process. We will now prove this.

Theorem 2 (Correctness of Constraint Update). *Suppose we have a term $t[?h]_{p_h}$ and instantiate the hole $?h \equiv s$. For each constraint $c \in C_h$, satisfiability is preserved over Update:*

$$Satisfies((t[s]_{p_h})\sigma,\ c) = Satisfies((t[s]_{p_h})\sigma,\ Update(c))$$

where σ is an arbitrary grounding substitution.

Proof. Let $t' = (t[s]_{p_h})\sigma$ and $c' = Update(c)$. There are three cases, depending on the type of c:

1. c is of the form $NotConst_S(p_h,\ k)$:

   (a) Assume $s \neq k$. The rule **NotConst-satisfied** applies, which returns the updated constraint $\top$. Applying the *Satisfies* function to both the new and old constraints gives:

   $$c' : \quad Satisfies(t',\ \top) \Rightarrow True$$

   $$c : \quad Satisfies(t',\ c) \Rightarrow s \neq k \Rightarrow True$$

   Hence both the old and new constraints evaluate to true.

   (b) Assume $s = k$. Then the rule **NotConst-violated** applies, which detects that $c' = \bot$, By the semantic for $NotConst_S$, and the instantiation $?h \equiv k$, *Satisfies* produce the following:

   $$c' : \quad Satisfies(t',\ \bot) \Rightarrow False$$

   $$c : \quad Satisfies(t',\ c) \Rightarrow k \neq k \Rightarrow False$$

   Hence both the old and new constraints evaluate to false.

2. c is of the form $UnEqual(p_h, q_1, \ldots, q_n)$:
   (a) Assume $?h$ is instantiated to constant k. Then the rule **UnEqual-Const** applies. The updated constraint c' returned is:

$$c': \quad NotConst_S(q_1, k) \vee \ldots \vee NotConst_S(q_n, k)$$

   Evaluating the updated and old constraints respectively gives:

$$c': \quad Satisfies(t',\ c') \Rightarrow hd(t'|_{q_1}) \neq k \vee \ldots \vee hd(t'|_{q_n}) \neq k$$

$$c: \quad Satisfies(t',\ c) \Rightarrow (t'|_{q_1}) \neq k \vee \ldots \vee (t'|_{q_n}) \neq k$$

   Clearly $Satisfies(t', c')$ only evaluates to true when at least one of $hd(t'|_{q_i}) \neq k$ holds. Note that in this situation $Satisfies(t',\ c)$ will also be true, as it contains the corresponding conjuncts $t'|_{q_i} \neq k$.
   If $Satisfies(t', c')$ evaluates to false, then all its conjuncts are false, which means that $hd(t'|_{q_1}) = k \vee \ldots \vee hd(t'|_{q_n}) = k$ Hence all $(t'|_{q_i})\sigma$ must be equal. In this situation, all inequalities between $t'|_{q_i}$'s in $Satisfies(t',\ c)$, will also evaluate to false.
   (b) Now assume $?h$ is instantiated to a function, introducing new holes for its arguments: $f(?h_1, \ldots ?h_m)$. Here t' abbreviate $(t[f(?h_1, \ldots, ?h_m)]_{p_h})\sigma$. The rule **UnEqual-Fun** applies and returns the updated constraint c':

$$NotConst_S(q_1,\ f) \vee \ldots \vee NotConst_S(q_n,\ f) \bigvee_{i=1}^{i=m} UnEqual(p_{[h,\ i]},\ q_{[1,\ i]}, \ldots q_{[n,\ i]})$$

   As before, we can apply a grounding substitution to the term resulting from the hole's instantiation, and evaluate the updated constraint to:

$$Satisfies(t',\ c') \Rightarrow hd(t'|_{q_1}) \neq f \vee \ldots \vee hd(t'|_{q_n}) \neq f \vee$$
$$\forall i \in \{1 \ldots n\}.\ t'|_{p_{[h,1]}} \neq t'|_{q_{[i,1]}} \vee t'|_{q_{[1,1]}} \neq t'|_{q_{[i,1]}} \vee \ldots \vee t'|_{q[n,1]} \neq t'|_{q[i,1]}$$
$$\vee \ldots \vee$$
$$\forall i \in \{1 \ldots n\}.\ t'|_{p_{[h,m]}} \neq t'|_{q_{[i,m]}} \vee t'|_{q_{[1,m]}} \neq t'_{q_{[1,m]}} \vee \ldots \vee t'|_{q_{[n,m]}} \neq t'|_{q_{[1,m]}}$$

   The original constraint c, given the instantiation of $?h$ evaluates to:

$$Satisfies(t',\ c) \Rightarrow t'|_{q_1} \neq f(?h_1, \ldots, ?h_m)\sigma \vee \ldots \vee t'|_{q_n} \neq f(?h_1, \ldots,\ ?h_m)\sigma$$

   Assume $Satisfies(t',\ c')$ evaluates to true. Either, one of the disjuncts $hd(t'|_{q_i}) \neq f$ does indeed have a symbol other than f in the head-position, in which case the corresponding disjunct in $sat(t',\ c)$, namely $t'|_{q_i} \neq f(?h_1, \ldots, ?h_m)\sigma$, will also be true, so both constraints evaluate to true. Otherwise, the difference is further down the term-tree. For c', at least one of the disjuncts $t'|_{q_{[x,z]}} \neq t'|_{q_{[y,z]}}$ must hold[9]. In this case,

[9] Here x and y range over the positions required to be unequal in the constraint: $1 \leq x, y \leq n$, while z ranges over the arguments of the function f: $1 \leq z \leq m$

for c, we must compare the term trees further down, as the top level symbols were all the same. This means inspecting exactly the sub-trees rooted at $t'|_{q_{[i,j]}} \neq t'|_{q_{[k,j]}}$, the same as for c'. Hence *Satisfies* produces the same result in both cases.
Assume $\mathit{Satisfies}(t',\ c')$ evaluates to false. Then all disjuncts $hd(t'|_{q_i}) \neq f$ are false, as well as all disjuncts for terms further down the tree. In other words, we must have all $t'|_{q_i}$ equal. In this case, $\mathit{Satisfies}(t',\ c)$ also evaluates to false.

Hence, the constraint update function is correct, it always return a constraint which preserves satisfiability of the original constraint after the instantiation of a hole.

8 Correctness of the Synthesis Algorithm

Having established the correctness of the constraint update algorithm, we can now prove the correctness of the synthesis algorithm.

Lemma 3 (No instances of constraint-terms). *After each instantiation of some hole by the synthesis algorithm, the partially synthesised term t does not contain any subterm that is an instance of any constraint-term.*

Proof. By contradiction. Assume there is a subterm s in t which matches some constraint-term s_g: $g(x_1, \ldots, x_n)$. We thus have $g(x_1, \ldots, x_n)\sigma \equiv s$. Then s must have the same top-level constant symbol as the constraint-term, namely g, which must have been introduced by the rule **Function** from Def. (8).

This instantiation would have added the set of constraints associated with g, $\mathit{Constrs}(g)$, to the set of constraints C associated with the term t that we are synthesising. We know that there is a constraint associated with the constraint-term s_g in $\mathit{Constrs}(g)$.

Furthermore, there must have been one last hole $?h$ that was instantiated in s to make it match s_g. By Theorem 2, the original constraint c_g may have been updated to a constraint c'_g, which evaluates to the same boolean value on the s. Either the position p_h of the last hole occurs in one of the conjuncts of the initial constraint, or it has been introduced by updates to some *UnEqual* constraint. As we shall see, c'_g must at the point of instantiation of $?h$ consist of a single $\mathit{NotConst}_S$ constraint, which would have prevented synthesis from ever instantiating $?h$ in such a way as to produce the reducible subterm s.

1. p_h is mentioned in c_g:
   We assume c_g was initially a disjunction $c_{g1} \vee \ldots c_{gh} \vee \ldots \vee c_{gn}$. Because s is assumed to be an instance of s_g, all disjuncts except c_{gh} must have been violated, and evaluated to false.
   $c_{gh} = \mathit{NotConst}_S(?h,\ k)$**:** This constraint was generated as s_g contains the symbol k in position p_h. Hence, we must instantiate $?h$ to k for to build an instance of s_g. Synthesis can do this by applying the **Constant** rule from Def. (8). However, the side-condition of the rule forbids such an

instantiation. Hence, s cannot be synthesised, and we have a contradiction.

$c_{gh} = \mathit{UnEqual}_S(p_1, \ldots p_h, \ldots p_n)$**:** The constraint-term s_g must have contained the same variables in the positions mentioned. As we assume $?h$ is the last hole to be instantiated, all other positions mentioned in the constraint, must have been instantiated to the same constant as in s_g. This would have updated the constraint by the rule **UnEqual-const**, to n $\mathit{NotConst}_S$ constraints. All of these except the one mentioning $?h$ must have been violated. As above, this prevents synthesis from instantiating $?h$ to the symbol that would make it and instance of s_g.

2. p_h has been introduced through constraint updates:
   Some position above $?h$ will have been involved in a constraint:

$$\mathit{UnEqual}_S(p_1, \ldots p_n)$$

   All positions up to the level on which $?h$ occurs must have been instantiated to equal terms, which in turn recursively introduce new $\mathit{UnEqual}_S$ constraints for each level. For the level of $?h$, the other holes must have been instantiated to the same constant, as $?h$ is the last hole. The proof is then analogous to the second part of Case 1.

Theorem 3 (Correctness of synthesis). *The synthesis algorithm only produces terms that are not instances of any constraint-term.*

Proof. By lemma 3, the synthesis algorithm maintains the invariant that no instantiation produces a subterm that is an instance of any term in the constraint-term set. This obviously also holds for the final iteration, so each term produced is irreducible.

From the above theorem we get the following corollary for the special case when constraints are generated from rewrite rules:

Corollary 1 (Synthesis of irreducible terms). *When the constraint-term set is derived from the left-hand sides of a set of rewrite rules, the synthesis algorithm only produce terms that are irreducible.*

9 Conclusions and Further Work

We have presented a formal account of term synthesis in IsaCoSy. This introduces a much simpler constraint language than that previously presented in [7]. Using this language, we described the constraint generation and synthesis machinery in a more general fashion than previously, abstracting away implementation details. This clarifies what the techniques does (and does not do), and facilitates future re-implementation. Moreover, the simplicity of the concept behind IsaCoSy, along with the mathematical language, admit a mathematical analysis of the properties of theory exploration.

The mathematical account has allowed us to prove important properties about IsaCoSy. We proved the correctness of the machinery for generating constraints from rewrite rules, as well as the correctness of constraint updates during synthesis. Finally, we also proved the correctness of the synthesis algorithm itself.

We believe that formal accounts of theory exploration will be helpful in enabling comparison between different approaches to theory formation by clearly highlighting the fundamental properties of different systems. This is the first mathematical account of a property of theory exploration that we are aware of. As further work, we plan to include the simplified constraint language in the implementation of IsaCoSy. An interesting direction for further theoretical work on theory formation, based on IsaCoSy's approach, is consider when the need for generalisation can be avoided by synthesising the needed background lemmas. Some results showing the potential of this idea, compared to other lemma-speculation techniques, can be found in [6].

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. IEEE Computer Society, 2004.
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. K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *TAP'10 Proceedings of the 4th international conference on Tests and proofs*, volume 6143 of *LNCS*, pages 6–21. Springer, 2010.
4. L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *TPHOLs-17*, LNCS, pages 83–98. Springer, 2004.
5. M. Hodorog and A. Craciun. Scheme-based systematic exploration of natural numbers. In *Synasc-8*, pages 26–34, 2006.
6. M. Johansson, L. Dixon, and A. Bundy. Dynamic rippling, middle-out reasoning and lemma discovery. In *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2010. 10.1007/978-3-642-17172-7_6.
7. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, pages 1–39, 2011. 10.1007/s10817-010-9193-y (To appear in print, published online at: `http://www.springerlink.com/content/bk711q2u247mr967`).
8. 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 CS, 2006.
9. R. McCasland, A. Bundy, and S. Autexier. Automated discovery of inductive theorems. *Special Issue of Studies in Logic, Grammar and Rhetoric: Festschrift in Honor of A. Trybulec*, 10(23):135–149, 2007.
10. O. Montano-Rivas, R McCasland, L. Dixon, and A Bundy. Scheme-based synthesis of inductive theories. In *Proceedings of the 9th Mexican International Conference on Artificial Intelligence*, volume 6437 of *LNCS*, pages 348–361. Springer, 2010.