A Calculus for Conjecture Synthesis

Moa Johansson[1], Lucas Dixon[2], and Alan Bundy[3]

[1] Dipartimento di Informatica, Università degli Studi di Verona **
[2] Google, New York
[3] School of Informatics, University of Edinburgh
moakristin.johansson@univr.it, ldixon@google.com, 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. To make synthesis tractable, IsaCoSy only considers synthesis of terms that are not 'more complex versions of already known terms'. This lets IsaCoSy avoid synthesis of undesirable terms, such as instances of left-hand sides of rewrite rules. In this paper, we present a formal language for constraining synthesis such that synthesis does not construct terms that can be matched by any of 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: given a set of constraint-terms as input, IsaCoSy produces only terms that are not instances of any of the constraint-terms.

1 Introduction

IsaCoSy is an automated theory formation system for inductive theories [7]. The key idea to make the conjecture synthesis process tractable and the resulting conjectures interesting, is that synthesis is constrained to only construct terms that do not match any term in a set of constraint terms. IsaCoSy 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. In contrast, a naive version, which simply considers all possible terms, will generate a huge number of terms, many of which are instances of the same smaller term. For example, suppose we have a rewrite rule $Suc(x) + y = Suc(x + y)$, and add its left-hand side as a constraint term. IsaCoSy will not generate any terms matching $Suc(x) + y$. A naive generate-and-test style synthesis algorithm would however consider *all* terms, including $Suc(x + x) + y$, $Suc(x + x + x) + y$ and so on. These terms are instances of the constraint term and can be rewritten. Such terms are deemed not interesting by IsaCoSy. In addition to serving as a criteria for interestingness, experimental results suggest that the avoidance of matching terms cuts down the synthesis search space for IsaCoSy by an exponential factor with respect to a naive synthesis algorithm [7].

** This research was funded by EPSRC grant EPE/005713/1 as well as grant 2007-9E5KM8 of the Ministerio dell' Università e Ricerca.

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

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

IsaCoSy builds progressively larger conjectures, starting from a given top-level symbol. The system then passes synthesised conjectures to a counter-example checker, which filters out obviously false statements. The remaining conjectures are given to an automatic inductive prover. Any theorems found are 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 further theorems. The implementation and evaluation of IsaCoSy has been described in [7]. IsaCoSy was able to generate most of the relevant inductive lemmas occurring in Isabelle's libraries for natural number and lists[4], which were hand-created by a human user. We performed a precision/recall analysis, which showed that IsaCoSy achieved high recall, as most library lemmas were found, but somewhat lower precision, as additional theorems also were synthesised. The few library-lemmas missed out could typically be derived easily from ones that were generated. Despite the lower precision, 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[5].

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.

Since IsaCoSy 1.0, which was described in [7], the following improvements have been implemented[6]:

– IsaCoSy can now generate constraints from arbitrary terms, not just left-hand sides of equations. This enables some additional heuristics to be nat-

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

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

[6] Available at `http://dream.inf.ed.ac.uk/projects/isaplanner`

urally encoded. For instance, a sequence of two or more successor symbols can be disallowed by introducing a constraint-term of the form $Suc(Suc(x))$.
- When constraints are generated from left-hand sides of rewrite rules, IsaCoSy first attempts to orient the equation to ensure it is a valid rewrite rule. This avoids over-generation of constraints, which occasionally occurred in version 1.0.
- The choice of automated prover and counter-example checker is now given as a parameter to IsaCoSy, and thus controlled by the user. The default prover is IsaPlanner's rippling-based prover [4], and the default counter-example checker is QuickCheck [1]. However, the user can easily plug in other induction tactics and counter-example finders.
- The top-level function for synthesis is now an input parameter to synthesis. This makes it easier to generate terms other than equations.

The formalisation in this paper highlighted some redundancies in IsaCoSy's constraint language, which led to the development of a more succinct language (see §5). This has however not yet been implemented.

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 mathematical analysis of their properties.

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 terms that are not instances of any of its constraint terms, IsaScheme further restricts its search space by only considering those that also *are instances* of its given schemes and which have been instantiated by its input set of closed terms. IsaScheme orients candidate equations to exclude any conjectures that are not valid rewrite rules and then applies Knuth-Bendix completion. Candidate theorems are normalised and thus form a rewrite system together with the initial background lemmas from the theory. IsaScheme employs Isabelle's QuickCheck and Nitpick tools for counter-example checking and either IsaPlanner's rippling-based prover or a custom tactic, which can be given as a parameter by the user, for proofs.

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 reasons forward, and sometimes backwards, to derive logical consequences of these. A range of heuristic measures are built into the generation process to avoid generating theorems 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 explores 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 does not employ an inductive theorem prover, and thus cannot attempt to prove the conjectures it produces.

3 Preliminaries

We introduce some notation which will be used throughout the paper. We use HOL terminology, where predicates are treated simply as functions returning a boolean value. All expressions are thus terms, and no distinction is made between terms and formulas.

For our purposes, it is convenient to define (possibly partially synthesised) terms as n-ary trees, captured by the following datatype[7]:

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 function's arguments. A term that consists of a constant, k, with no arguments is represented by $App(k, [\,])$.

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

IsaCoSy does not currently synthesise terms containing 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. $hd(App(f, args)) = f$. denote substitutions on terms. The symbols $=$ and $\neq$ on terms denote syntactic equality and disequality respectively. We use $?h \equiv s$ to represent the instantiation of a hole $?h$ by a term s. Instantiation of holes can be viewed as traditional substitution of a free variable for a term. A *ground synthesised term* is thus a term which does not contain any holes. A substitution σ is called a *grounding substitution* on a term t if it produces a ground term $t\sigma$ by instantiating all remaining holes.

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.

[7] We have abstracted away type information as it adds no interesting complexity.

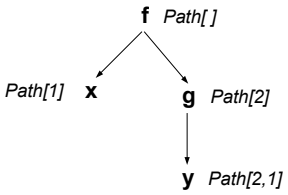

Fig. 1. Term-tree with path-representations of each position.

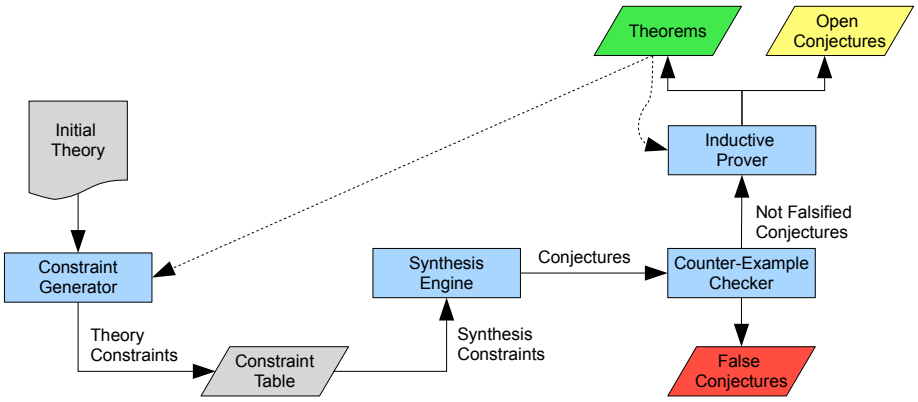

Fig. 2. IsaCoSy's synthesis process.

We write $t[s]_p$ for a term t with a subterm s in position defined by the path p. The term s may 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 constant symbols are $+$, 0 and *Suc*. Most of the initial constraints are generated from the left-hand sides of defining equations, as well as the left-hand side of other 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

```
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.

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 focus of this paper.

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 constant 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 remain 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

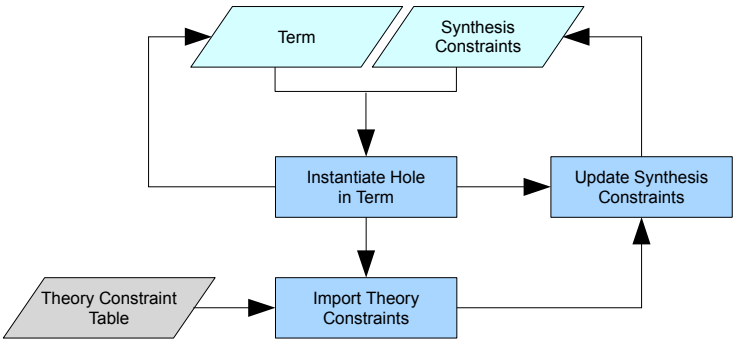

Fig. 4. The synthesis engine. While there are still open holes in the term, IsaCoSy picks a hole and a symbol to instantiate it with, in accordance with the constraints. New constraints for new holes are imported for the relevant symbol, and old constraints are updated to take the instantiation into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

symbol. Here, we import constraints about $+$, which will restrict instantiations of $?h_4$ and $?h_5$.

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. This means that the space of synthesisable terms is incrementally reduced and kept manageable even when larger term-sizes are considered. Proved theorems are also used by the prover in subsequent proofs. This makes the prover more powerful as more theorems are discovered. The synthesis algorithm is described in more detail in §6.

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.

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 being a redex for that rule. Otherwise, we say that t *violates* the constraint. Def. 5 in §5.2 specifies the

Satisfies relation. For a synthesised term, we require that *all its subterms satisfy all constraints*, meaning that no subterm of a synthesised term is an instance of any of the constraint terms (see Theorem 3).

The constraints refer to positions as paths from the top of the term tree, which allows us to simplify and revise the constraint language presented in [7], which 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 2 (Constraint Language)

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

The *NotConst* constraints express that a constant symbol k is not allowed to occur in position p_i. The *UnEqual* constraints specify a list of positions, $p_1, \ldots, p_n$, which are 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[8].

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

- 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 theory 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 constraint term will often give rise to a set of constraints for various positions. They express instantiations not simultaneously allowed if we are to ensure that the synthesised term is not an instance of the constraint term (see Example 2). Synthesis may violate some of these 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:

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

Definition 3 (Theory Constraints for a term t)

$$\begin{aligned} &\mathit{ThyConstrs}(t) := \\ &\qquad \bigvee \left(\{\mathit{NotConst}_T(p,k) \ : \ t|_p = k \wedge \mathit{IsConst}(k)\} \cup \right. \\ &\left. \{\mathit{UnEqual}_T(p_j,\ldots,p_m) \ : \ t|_{p_j} = \ldots = t|_{p_m} \wedge p_j \neq p_m \wedge \mathit{IsVar}(t|_{p_j})\}\right) \end{aligned}$$

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 left-hand side 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 first generates two constraints forbidding the constants + and 0 in their respective positions: $\mathit{NotConst}_T(\mathit{Path}[\,], +)$ and $\mathit{NotConst}_T(\mathit{Path}[1], 0)$. Synthesised terms must satisfy *at least* one of the two, hence IsaCoSy create the disjunction:

$$\mathit{NotConst}_T(\mathit{Path}[\,], +) \vee \mathit{NotConst}_T(\mathit{Path}[1], 0)$$

This specifies that if 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 $\mathit{Path}[\,]$ contains the symbol f, while position $\mathit{Path}[2]$ contains the symbol g and positions $\mathit{Path}[1]$ and $\mathit{Path}[2,1]$ both contain the variable x. This produces the constraint:

$$\mathit{NotConst}_T(\mathit{Path}[\,], f) \vee \mathit{NotConst}_T(\mathit{Path}[2], g) \vee \mathit{UnEqual}_T(\mathit{Path}[1],\ \mathit{Path}[2,1])$$

The theory constraints are stored in a map from a function symbol f to sets of constraints where f occurs in the head position (i.e. the theory constraints containing a disjunct forbidding f in the position $\mathit{Path}[\,]$).

Definition 4 (Theory constraints of a function f) *Let TC denote the set of all theory constraints generated from the given constraint-terms. The theory constraints for a particular function f is thus:*

$$\mathit{ThyConstrs}(f) = \{c \ : c \in \mathit{TC},\ \mathit{NotConst}_T(\mathit{Path}[\,], f) \in c\}$$

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

5.2 Semantics of Constraints

We define a function $\mathit{Satisfies}(t,\ c)$ 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 because t is ground, it does not contain any holes.

Definition 5 (Semantics for Constraints) *The Satisfies function is defined below for the constructs of the constraint language:*

NotConst:

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

UnEqual:

$$\mathit{Satisfies}(t, \mathit{UnEqual}(p_1,\ldots,p_n)) \Longrightarrow \bigvee_{1\leq i,j\leq n} t|_{p_i} \neq t|_{p_j}$$

Or:

$$\mathit{Satisfies}(t,\ c_1 \vee c_2) \Longrightarrow \mathit{Satisfies}(t,\ c_1) \vee \mathit{Satisfies}(t,\ c_2)$$

Top:

$$\mathit{Satisfies}(t,\ \top) \Longrightarrow \mathit{True}$$

Bottom:

$$\mathit{Satisfies}(t,\ \bot) \Longrightarrow \mathit{False}$$

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

The constraint update mechanism (see Def. 7 in §7), is a lazy unfolding of *Satisfies*, operating over the terms being synthesised, where the terms being synthesised 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. 5) 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 from which they were generated. We then show that the constraints only correspond to the terms from which they were generated, the *no over-coverage* property.

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

Lemma 1 (Sufficient coverage). *Given a term s and a constraint-term r, if s is an instance of r, then s violates $\mathit{Constraints}(r)$.*

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

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

$$UnEqual_T(p_j, \ldots, p_m)$$

By assumption $s = r\sigma$ and the substitution σ must map the variable x to the same term everywhere it occurs in r, 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 $UnEqual_T$ in Def. 5:

$$Satisfies(s,\ UnEqual_T(p_j, \ldots,\ p_m))$$

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

Thus $Satisfies(s,\ c_1 \vee \ldots \vee c_n)$ evaluates to false, as s violates $Constraints(r)$.

Lemma 2 (No over-coverage). *Given a constraint-term r, if s is a term that violates $Constraints(r)$, then s is an instance of r.*

Proof. By contradiction, assume s violates $Constraints(r)$ and is not an instance of r. $Constraints(r)$ is a disjunction: $c_1 \vee \ldots \vee c_n$. As s violates the constraints, we know that $Satisfies(s,\ c_1 \vee \ldots \vee c_n) \implies False$. By Def. 5, we hence have $Satisfies(s,\ c_i) \implies False$ 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 r containing a constant symbol k, will have contributed a constraint $NotConst_T(p_i,\ k)$. We know that $Satisfies(s,\ NotConst_T(p_i,\ k)) = False$, so we must have $s|_{p_i} = k$ for each position p_i. Hence s and r 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

$$UnEqual_T(p_j, \ldots, p_m)$$

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

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

Theorem 1 (Exact coverage). *Given a term s and a constraint-term r, the constraint produced by the constraint generation algorithm is satisfied by s iff s does not match r.*

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. 6 to a partially synthesised term, t, that contains some uninstantiated hole $?h$.

In addition to a partially synthesised term, t, the synthesis rules refer to a collection of synthesis constraints, C, associated with t, denoted by $C \parallel t$. We write C_h for the constraints in C which contain a reference to the (position of) hole $?h$. We use $Dom(?h)$ as the set from which synthesis selects candidate instantiations of a compatible type for a hole $?h$. Synthesis tries all instantiations from $Dom(?h)$ that are not forbidden by the presence of a singleton $NotConst_S$ constraint[9]. When a function symbol f is picked to instantiate a hole, the theory constraints associated with that function symbol, denoted by $ThyConstrs(f)$, are imported as new synthesis constraints. As we wish the new synthesis constraints to refer to the subterm rooted at the newly instantiated hole, we prefix all paths in the theory constraints by the path to the hole. Prefixing all paths in a constraint c, denoted by $\forall p_j \in Paths(c)$, by another path p_i is written $c[\forall p_j \in Paths(c).\ p_j \mapsto p_i \mathbin{@} p_j]$.

Existing synthesis constraints must also be updated to capture the instantiation of a hole. The function *Update*, 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. 7).

Definition 6 (Synthesis Algorithm) *The synthesis algorithm instantiates holes by the following two rules:*

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

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

$$if \begin{cases} f \in Dom(?h) \\ NotConst_S(?h, f)\ \notin C_h \end{cases}$$

$$where\ Constrs(f) =$$

$$\{c\ :\ c' \in ThyConstrs(f),\ c = c'[\forall p_j \in Paths(c').\ p_j \mapsto p_i \mathbin{@} p_j]$$

Constant: $?h \equiv k$

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

[9] 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.

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

Example 4. Recall Example 1, where a the hole $?h_1$ in the term t: $?h_1 + ?h_2 = ?h_3$ was instantiated by $?h_4 + ?h_5$. Def. 6. The set of constraints C associated with t is updated: Any existing constraints associated with the hole $?h$ are updated, and new constraints on $?h_4$ and $?h_5$ may be added.

The hole to be instantiated, $?h_1$, occurs in position $Path([1,1])$. The set of synthesis constraints associated with $?h_1$ is:

$$\{NotConst_S(Path([1,1]), 0), NotConst_S(Path([1,1]), Suc)\}$$

which came from the left-hand sides of the equations defining + (see Figure 3). As $?h_1$ is instantiated to +, both of these are satisfied, so they can be dropped.

For the new subterm rooted at the position of $?h_1$, we also import constraints for the new holes $?h_4$ and $?h_5$, from the theory constraints for +:

$$\{NotConst_T(Path([1]), 0), NotConst_T(Path([1]), Suc)\}$$

To produce synthesis constrains referring to the right subterm of t, we must prefix the paths in the theory constraints by the path to $?h_1$ (namely $Path([1,1])$:

$$\{NotConst_S(Path([1,1,1]), 0), NotConst_S(Path([1,1,1]), Suc)\}$$

These constraints state that, like $?h_1$, which also was in the first argument position of a +, $?h_4$ is not allowed to be instantiated to 0 or *Suc*.

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. This is done by the function *Update*, mentioned in the previous section. This is essentially a lazy unfolding of the *Satisfies* relation. We write p_h for the position of the most recently instantiated hole $?h$. The *Update* function is defined structurally on the type of the given constraint. If the constraint is not mentioning the instantiated hole $?h$, then $Update(c) = c$.

Definition 7 (Constraint Update Function)

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

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

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

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

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

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

UnEqual-Const: $?h \equiv k$

$$\begin{aligned} &\mathit{Update}(\mathit{UnEqual}_S(p_h,\ p_1, \ldots, p_n)) \Longrightarrow \\ &\qquad \mathit{NotConst}_S(p_1,\ k) \vee \ldots \vee \mathit{NotConst}_S(p_n,\ k) \end{aligned}$$

Or-Top:

$$\mathit{Update}(\top \vee c) \Longrightarrow \top$$

Or-Bottom:

$$\mathit{Update}(\bot \vee c) \Longrightarrow \mathit{Update}(c)$$

Or:

$$\mathit{Update}(c_1 \vee c_2) \Longrightarrow \mathit{Update}(c_1) \vee \mathit{Update}(c_2)$$

The correctness of the constraint update machinery is crucial to the efficiency and correctness of the entire synthesis process. We will now prove this by showing that the value of the *Satisfies*-function is the same as before after updating a constraint. Recall that the *Satisfies* function works with ground terms. We will thus prove that satisfiability/violation of updated constraints are preserved over all possible instantiations of remaining holes. That is, if the term t satisfied a constraint, it will also satisfy the updated constraint.

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

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

for all grounding substitutions σ instantiating remaining holes.

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

1. c is of the form $\mathit{NotConst}_S(p_h,\ k)$:

   (a) Assume $?h \equiv t_h$ and $t_h \neq k$. The rule **NotConst-satisfied** applies, which returns the updated constraint $\top$. Applying the *Satisfies* function to both c and c' gives:

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

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

   Hence both the old and updated constraint evaluates to true.

(b) Assume $?h \equiv k$. Then the rule **NotConst-violated** applies, which returns the updated constraint $c' = \bot$, The constraints c and c' are evaluated to:

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

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

Hence both the old and updated constraint evaluates to false.

2. c is of the form $\mathit{UnEqual}(p_h, q_1, \ldots, q_n)$:

(a) Assume $?h \equiv k$ for some constant k. Then the rule **UnEqual-Const** applies. The updated constraint c' returned is:

$$c': \quad \mathit{NotConst}_S(q_1, k) \vee \ldots \vee \mathit{NotConst}_S(q_n, k)$$

Evaluating the updated and old constraints respectively gives:

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

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

- If $\mathit{Satisfies}(t', c) \Longrightarrow \mathit{True}$ then at least one of $t'|_{q_i} \neq k$ holds. Then also $\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{True}$, as c' contains the corresponding disjunct $\mathit{hd}(t'|_{q_i}) \neq k$, which also holds.
- If $\mathit{Satisfies}(t', c) \Longrightarrow \mathit{False}$ then all its conjuncts are false, which means that $t'|_{q_1} = k \wedge \ldots \wedge t'|_{q_n} = k$ holds. Then also $t'|_{q_i} = \mathit{hd}(t'|_{q_i}) = k$ for all q_i, so $\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{False}$.

(b) Assume $?h \equiv f(?h_1, \ldots ?h_m)$ for a function f, introducing new holes $?h_1, \ldots ?h_m$ for its arguments. Let t' abbreviate $(t[f(?h_1, \ldots, ?h_m)]_{p_h})\sigma$ for any grounding substitution σ. The rule **UnEqual-Fun** applies and returns the updated constraint c':

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

We evaluate c' to:

$$\begin{aligned} &\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{hd}(t'|_{q_1}) \neq f \vee \ldots \vee \mathit{hd}(t'|_{q_n}) \neq f \vee \\ &\forall i \in \{1 \ldots n\}.\ t'|_{p_{[h,1]}} \neq t'|_{q_{[i,1]}} \vee t'|_{q_{[1,1]}} \neq t'|_{q_{[i,1]}} \vee \ldots \vee t'|_{q[n,1]} \neq t'|_{q[i,1]} \\ &\qquad \vee \ldots \vee \\ &\forall i \in \{1 \ldots n\}.\ t'|_{p_{[h,m]}} \neq t'|_{q_{[i,m]}} \vee t'|_{q_{[1,m]}} \neq t'_{q_{[1,m]}} \vee \ldots \vee t'|_{q_{[n,m]}} \neq t'|_{q_{[1,m]}} \end{aligned} \tag{7.1}$$

The original constraint c, given the instantiation $?h \equiv f(?h_1, \ldots ?h_m)$ evaluates to:

$$\begin{aligned} &\mathit{Satisfies}(t',\ c) \Longrightarrow \\ &\qquad t'|_{q_1} \neq f(?h_1, \ldots, ?h_m)\sigma \ \vee \ldots \vee t'|_{q_n} \neq f(?h_1, \ldots, ?h_m)\sigma \end{aligned} \tag{7.2}$$

– Assume $\mathit{Satisfies}(t',\ c) \Longrightarrow \mathit{True}$.
  Then for some i, we have that $t'|_{q_i} \neq f(?h_1, \ldots, ?h_m)\sigma$. This is true when either
  - $hd(t|_{q_i}) \neq f$ holds, which is also a disjunct in (7.1), hence also $\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{True}$.
  - $hd(t|_{q_i}) = f$, for all positions q_i, $1 \leq i \leq n$, and the difference is further down the term tree in t': there must exist positions where $t'|_{p[h,j]} \neq t'|_{q[i,k]}$ holds, with j, k ranging over the positions of the arguments of f: $1 \leq j, k \leq m$. All such positions occur as conjuncts in (7.1), and as at least one of them must hold, we also have that $\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{True}$.

– Assume $\mathit{Satisfies}(t',\ c) \Longrightarrow \mathit{False}$.
  Then all disjuncts $(t'|_{q_i}) \neq f(?h_1, \ldots, ?h_m)\sigma$ in (7.2) are false. In other words, we must have all $t'|_{q_i}$ equal. In this case, all disjuncts in (7.1) also evaluates to false and $\mathit{Satisfies}(t',\ c') \Longrightarrow \mathit{False}$.

3. c is of the form $c_1 \vee c_2$:
   By induction on the structure of c_1 and c_2. Cases (1) and (2) above are the base cases. Assuming $\mathit{Satisfies}(t', c_i) = \mathit{Satisfies}(t', \mathit{Update}(c_i))$ for $i \in \{1, 2\}$, then clearly also $\mathit{Satisfies}(t', c_1 \vee c_2) = \mathit{Satisfies}(t', \mathit{Update}(c_1) \vee \mathit{Update}(c_1))$.

Hence, the constraint update function is correct, it always returns 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 r: $g(x_1, \ldots, x_n)$, hence $g(x_1, \ldots, x_n)\sigma \equiv s$, for some substitution σ. 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. 6.

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. $\mathit{Constrs}(g)$ include the particular constraint c_r associated with the constraint-term r.

Furthermore, there must have been one last hole $?h$, in a position mentioned in c_r[10], that was instantiated in $s[?h]_{p_h}$ to allow it to match r. Assume c_r is a disjunction $c_{r_1} \vee c_{r_n}$. There are three cases, depending on the position p_h of the last hole in s to allow it to match r:

[10] Thus, $?h$ is assumed to be in some position *not* corresponding to a singleton variable in r, as such positions have no associated constraints.

1. p_h correspond to a position of a constant k in r:
   The constraint c_r will contain a disjunct $NotConst_S(p_h, k)$. As $?h$ is the last hole, we may assume that all other constraints in the disjunction have been violated, as s already match r in all other positions.
   To make s match r, we must apply the the **Constant** rule from Def. 6, instantiating $?h$ with k. However, there is a single constraint $NotConst_S(p_h, k)$, so the side-condition of the **Constant** rule forbids k as an instantiation. Hence, s cannot be synthesised and we have a contradiction.
2. p_h correspond to a position of a variable (occurring more than once) in r:
   The constraint c_r will contain a disjunct $UnEqual_S(p_1, \ldots, p_h, \ldots, p_n)$. As we assume $?h$ is the last hole to be instantiated of those mentioned in the constraint, all other positions $p_1, \ldots, p_n$ must have been instantiated to the same constant k. This would have updated the constraint by the rule **UnEqual-Const**, to a disjunction of n $NotConst_S$ constraints. All of these except the one mentioning p_h must have been violated. As above, this prevents synthesis from instantiating $?h$ to k, and hence we cannot synthesise s.
3. p_h correspond to a position *below* a variable (occurring more than once) in r:
   The position p_h must have been introduced in the synthesis constraint by constraint updates. Some position above $?h$ will have been involved in a constraint: $UnEqual_S(p_1, \ldots p_n)$.
   WLOG suppose p_h occurs below p_n. All subterms of s at positions $s|_{p_1}, \ldots,$ $s|p_{n-1}$ must be equal for s to potentially match r. The constraint update function would thus recursively have introduced new $UnEqual_S$ constraints for each level below these, finishing at the level of p_h, where the other holes must have been instantiated to the same constant k, introducing a disjunction of n $NotConst_S$ constraints. All except the one mentioning p_h have been violated, as $?h$ is the last hole. As before this prevents instantiating $?h$ to k, so s cannot be synthesised.

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

Proof. By induction. The base case trivially holds. For the step-case, 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.

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 produces 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 technique does (and does not do), and facilitates future re-implementation and extension. Moreover, the simplicity of the concept behind IsaCoSy, along with the mathematical language, admits 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 terms, showing that the constraints generated excludes exactly the terms that are instances of the constraint term. We also proved the correctness of constraint updates during synthesis, satisfiability of the constraint is preserved also when constraints are updated. Finally, we also proved the correctness of the synthesis algorithm itself: IsaCoSy only produces terms that do not contain instances of any constraint term. For constraint terms coming from left-hand sides of rewrite rules, this means that only irreducible terms are synthesised.

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 of which we are aware.

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 to 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.
7. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

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.