Formalising Term Synthesis in IsaCoSy

Moa Johansson
Università degli Studi di Verona
moakristin.johansson@univr.it

Lucas Dixon
University of Edinburgh
ldixon@inf.ed.ac.uk

Alan Bundy
University of Edinburgh
a.bundy@ed.ac.uk

Abstract

IsaCoSy is a theory formation system for inductive theories. It synthesises conjectures and uses the ones that can be proved to produce a background theory for a new formalisation within a proof assistant. We present a formal account of the algorithms implemented in the system, and prove their correctness. In particular, we show that IsaCoSy only produces irreducible terms, using constraints generated from the left-hand sides of a set of rewrite rules.

1 Introduction

IsaCoSy is an automated theory formation system for inductive theories. Given a set of datatype and function definitions, it builds progressively larger conjectures, starting from a given top-level symbol and the smallest term size possible. The key to making the synthesis process tractable, and the resulting conjectures interesting, is that only irreducible terms are generated. IsaCoSy generates constraints from all rewrite rules in its input theory, which forbids instantiations that would render a new term reducible. IsaCoSy 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 [3]. The proved results are intended to serve as intermediate lemmas within a user's, or a proof tool's, subsequent attempts to prove more involved theorems. IsaCosy also use theorems found to generate additional constraints on subsequent synthesis attempts.

The implementation and successful evaluation of IsaCoSy has been described in [5]. IsaCoSy was shown able to generate most of the relevant inductive lemmas occurring in Isabelle's libraries for natural number and lists[1], which have been created by a human user. Any library-lemmas missed out by IsaCoSy could typically easily be derived 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[2].

IsaCoSy has so far only been applied to generation of equational terms. However, the desired top-level symbol of the synthesised terms is given as a parameter to IsaCoSy. It can therefore easily be extended to generate terms other than equations.

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

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

[1] http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/index.html
[2] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/synth_results.php

To complement the algorithmic description given in [5], we here present a higher-level, more succinct, formal description of IsaCoSy's constraint generation and synthesis machinery. Using this formalisation, we prove the fundamental correctness property for our system: it generates exactly the irreducible terms of the language. The 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.

Other theory-formation systems, such as MATHsAiD [6] and Theorema [2, 4] have been applied to inductive theories. However, none of the algorithms used in those systems have been proved correct.

2 Background

2.1 Terms

For our purposes, it is convenient to define terms as n-ary trees, captured by the following minimal data-structure[3]:

$$\begin{aligned} \mathit{Atom} &:= \mathit{Const\ of}\ c \mid \mathit{Var\ of}\ v \\ \mathit{Term} &:= \mathit{App\ of}\ (\mathit{Atom} * \mathit{Term\ list}) \end{aligned}$$

Atom is either a variable or a constant symbol, named v or c respectively. Within an application, represented by the *App* constructor, the first argument (an *Atom*) is an atomic variable or constant symbol. The *Term list* represents the arguments when the *Atom* is of function type. A term that consists of a variable or constant with no arguments is represented by $App(x,[])$. We write $hd(t)$ to denote the symbol in the head position of a term, e.g. $hd(App(f,args)) = f$. 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 prefix by ? variables that are allowed to be instantiated by unification, e.g. $?x$. Such variables are referred to as meta-variables. During synthesis, holes are represented by meta-variables.

We use σ to denote substitutions on terms, mapping meta-variables to their instantiations.

The symbols $=$ and $\neq$ on terms denote syntactic (dis)equality. We use $\equiv$ to represent instantiations of meta-variables.

2.2 Positions in Terms

Positions in terms are expressed as *paths*. These are lists of argument positions within the application constructors of 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.

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 written as $t|_p$. 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 a position immediately below p_i in the term tree. To append two paths to each other we write $p_i :: p_j$.

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

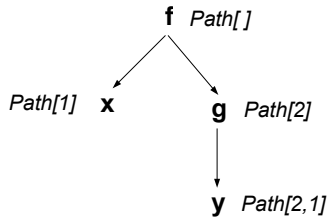

Figure 1: Term-tree with path-representations of each position highlighted.

3 Overview of IsaCoSy

Figure 3 illustrates the synthesis procedure of IsaCoSy. The initial input is a theory which contains a set of datatypes, a set of function definitions, and possibly additional known lemmas. An example toy-theory about natural numbers is given in Figure 2.

```
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)
```

Figure 2: An example input theory for IsaCoSy. 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`. The lemma is derived automatically by Isabelle's definitional machinery for datatypes when the `Nat` type is declared.

The equations from the function definitions and lemmas 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 §4.1. The purpose of the constraints is to ensure the synthesis machinery will not generate terms that are more complex versions of already known rewrite rules. For example, given the definition of addition in Figure 2, we want to avoid generate terms of the form $0+Suc(\ldots(Suc\ y))=Suc(\ldots(Suc\ y))$, which are subsumed by the existing equation $0+y=y$. This is important in order to make the search tractable and to ensure we generate interesting theorems, in the sense that they are not simply specialisations of existing theorems. Furthermore, more general theorems are more widely applicable by the automated prover, which may use any theorems found during synthesis in subsequent proof attempts.

During the synthesis process, IsaCoSy imports theory constraints for the constants (which are typically function symbols) used in the term. 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. The input to the synthesis engine is a term containing *holes*, standing for the

parts yet to be synthesised and represented by meta-variables. For example, to synthesise equations the starting point is a term with two holes: $?h_1 = ?h_2$. 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. As an example, consider a partially synthesised term $?h_3 + ?h_4 = ?h_2$, and suppose we pick $?h_3$ to be instantiated next. The synthesis algorithm now has to pick a symbol to instantiate $?h_3$. Note the constraints on the term will forbid picking 0 or *Suc* as either of these would produce a term which matches the two rewrite rules from the definition of addition in Figure 2. The algorithm may however choose to instantiate $?h_3$ with $+$, resulting in an updated term with two new holes: $(?h_5 + ?h_6) + ?h_4 = ?h_2$. After instantiation, any new applicable constraints are imported from the theory constraints of the newly introduced symbol. In our example, we import constraints about $+$, which will restrict instantiations of $?h_5$ and $?h_6$. The synthesis algorithm is described in more detail in §5.

IsaCoSy will thus synthesise a set of terms adhering to the relevant constraints, given a top-level symbol and starting from some minimal term size (for equations this is 3). After each iteration the set of synthesised terms of the given size 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 for 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.

4 Constraint Language

The purpose of the constraint language is to express restrictions on synthesis in order to avoid generating any reducible term, containing a redex matching a known rewrite rule. This keeps the search space size manageable and ensures avoids the generation of more complex versions of already known theorems. The constraints express 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 a particular set of 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. We give a formal definition of the *Satisfies* relation in §4.2.

4.1 Constructing Constraints

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 [5]. In previous work, the constraint language was unnecessarily complicated as the constraints were built as a tree-like structure reflecting the underlying term. We also clarify the difference between theory constraints and synthesis constraints. The latter have a slightly richer language and are updated during synthesis, where the former characterise the generic constraints associated with constants, which arise from known rewrites rules. We therefore define them separately:

$$ThyConstr := NotConst_T(p,k) \mid UnEqual_T(p_1,\ldots,p_n)$$

$$SynthConstr := NotConst_S(p,k) \mid UnEqual_S(p_1,\ldots,p_n) \mid NotVar_S(p,v)$$

The $NotConst_T$ and $NotConst_S$ constraints express that a constant symbol is not allowed to occur in position p. The $NotVar_S$ is the analogue for variables. During synthesis, $NotVar_S$-constraints may be introduced after the update of an $UnEqual_S$ constraint (see §6). The $UnEqual_S$ and $UnEqual_T$ constraints specify a list of positions not allowed to be instantiated to equal terms. In addition, the language allows

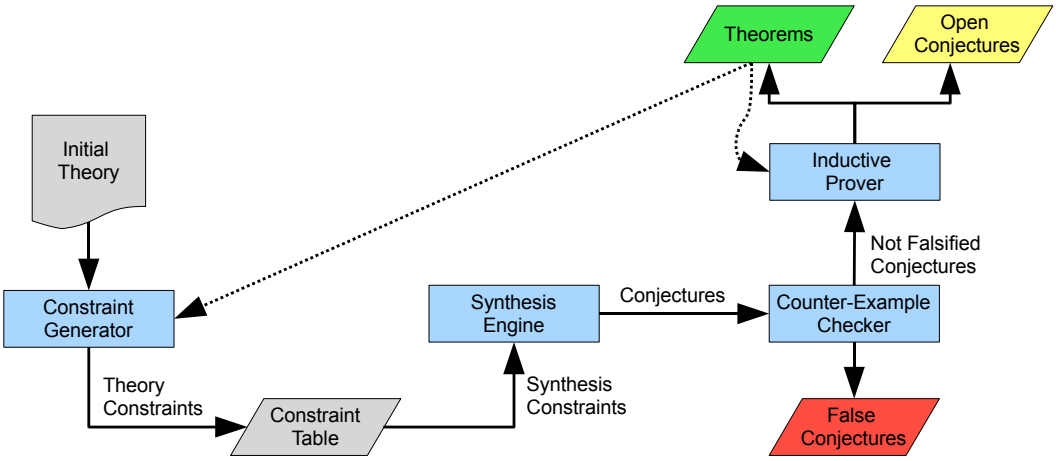

Figure 3: IsaCoSy's synthesis process.

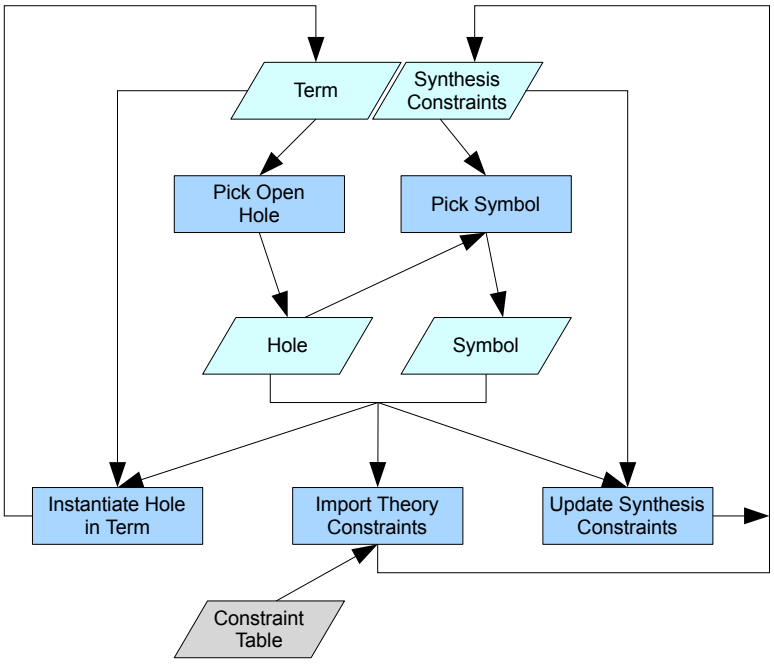

Figure 4: The synthesis engine. While there are still open holes in the term, IsaCoSy picks a hole and a symbol to instantiate it with, in accordance with the constraints. New constraints are imported for the relevant symbol, and old constraints are updated to take the instantiation into account.

disjunctions of constraints, $c_1 \vee c_2$[4], and contains the two constant constraints $\top$ and $\bot$, which are trivially satisfied/violated.

Theory constraints are generated from terms, which in our case typically are the left-hand sides of the available rewrite rules. Suppose we want to generate the constraints from a term l. For each position p in l that contains a constant symbol k we produce a constraint $NotConst_T(p,\ k)$. For all sets of positions $p_j, \ldots\ p_m$ in l, that all contain the same meta-variables (if any), we produce a constraint $UnEqual_T(p_j, \ldots, p_m)$. A term will typically generate 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 summarised below:

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

Here we assume that $\bigvee$ produces a disjunction of the constraints in the set. The predicate *IsConst*, checks if a term is a constant, while *IsVar* checks if the term is a meta-variable.

Example. For the rewrite rule $0 + y = y$, from the definition of addition in Figure 2, IsaCoSy generates a constraint from its left-hand side: $NotConst_T(Path[1], 0)$, forbidding the instantiation of the first argument of $+$ to be 0. A similar constraint is generated for the *Suc*-case. The constraints are stored in the database entry associated with $+$, as this is the head symbol of the left-hand side of the rule.

As a slightly more complex example, consider a term $f(?x,\ g(?x))$. The positions $[1]$ and $[1,2]$ contains symbols f and g respectively, while $[1,1]$ and $[1,2,1]$ contain the same (meta) variable. This produce the constraint:

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

4.2 Semantics of Constraints

We define a function $Satisfies(t,\ c)$ below, which takes a term t and a constraint c and returns returns *True* iff the term *satisfies* the constraint. Otherwise t *violates* the constraint. *Satisfies* is defined as follows for the constructs of the constraint language:

NotConst

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

NotVar

$$Satisfies(t,\ NotVar_S(p,v)) \Longrightarrow t|_p \neq v$$

UnEqual

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

Or

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

[4]The disjunction subsumes the *IfThen* and *NotSimult* constructors from [5]. Also note that *NotConst* and *NotVar* correspond to what was called *NotAllowed* and *VarNotAllowed* in [5]

Top

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

Bottom

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

Satisfies is here defined on synthesis constraint. The definition on theory constraints is by the obvious lifting from the corresponding synthesis constraints.

The constraint update mechanism, described in §6, is a lazy unfolding of *Satisfies*, operating over the terms in the process of being synthesised, which may be non-ground and contain holes (meta-variables). If the constraint refers to paths longer than is possible in t, the constraint is trivially satisfied.

4.3 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 reducible by the given rewrite rule. The properties below were stated in [5], but not proved. Introducing the *Satisfies* function in §4.2 allows us to do so.

We use the notation *Constraints(l)* for the disjunction of constraints the algorithm generates for the rule $l \rightarrow r$ (from its left-hand side).

Lemma 1 (Sufficent Coverage)**.** *Given a term t and a rule $l \rightarrow r$, if t contains a subterm s, which is a redex of $l \rightarrow r$, then t violates Constraints(l).*

Proof: *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 of constraints, each position p_i in l containing a constant symbol k, will have contributed a constraint $\mathit{NotConst}_T(p_i,\ k)$. However, the position p_i in s must contain k, or else $s \neq \sigma l$. 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 of the constraints, each set of positions $p_j, \ldots p_m$ in l containing the same variable $?x$, will contribute a constraint $\mathit{UnEqual}_T(p_j, \ldots, p_m)$. As $s = \sigma l$, the instantiation 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 §4.2, $\mathit{Satisfies}(s,\ \mathit{UnEqual}_T(p_j, \ldots,\ p_m))$ 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 any constraint from a rewrite rule for which it is a redex. Hence also $t[s]$ violates the constraint.

Lemma 2 (No over-coverage)**.** *Given a rule $l \rightarrow r$, if t is a term that violates Constraints(l), then there is a redex in t that matches l.*

Proof: By contradiction, assume no subterm of t is a redex of $l \rightarrow r$.

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 we have $\mathit{Satisfies}(s,\ c_1 \vee \ldots \vee c_n) \Longrightarrow \mathit{False}$. By the rules in §4.2 we hence have $\mathit{Satisfies}(s,\ c_i) \Longrightarrow \mathit{False}$ for each c_i, $1 \leq i \leq n$. We have two cases again, depending on the type of each c_i:

NotConst: By construction of constraints, each position p_i in l 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 l contains the same constant symbols in the same positions.

UnEqual: By the construction of the constraints, all position $p_j \ldots p_m$ containing the same variable $?x$ in l, will have contributed a constraint $UnEqual(p_j, \ldots, p_m)$. As $Satisfies(UnEqual_S(s,\ p_j, \ldots, p_m) \Longrightarrow False$, s must contain identical subterms $s|p_j = \ldots = s|p_m$. Hence there exist a unifier σ between s and l such that $\sigma\{?x \mapsto s|_{p_j}\}$.

As s and l agree on all positions of constant symbols, and we can find a unifier between the variables in l and s, then s is a redex for $l \to r$, contradicting our assumption.

Theorem 1 (Exact coverage)**.** *Given a term t and a rule $l \to r$, the constraint produced by the constraint generation algorithm is satisfied by t iff there is no redex within t that matches l.*

Proof: Follows from lemmas 1 and 2.

5 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 given in Figure 5 to a partially synthesised term t, containing some uninstantiated hole $?h$. In addition to the partially synthesised term, the rules take a collection of constraints, C, associated with 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 $?h$.

Constant

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

$$\text{if} \begin{cases} k \in Dom(?h) \\ NotConst_T(?h,k) \ \notin C_h \\ Constrs(k) = \{c \mid c \in ThyConstrs(k) \wedge \forall p_j \in c.\ p_j \mapsto p_i :: p_j\} \end{cases}$$

Variable

$$\frac{C \parallel t[?h]_{p_i}}{C \mapsto (\forall\ c \in C_h.\ Update(c)) \parallel t[v]_{p_i}} \qquad \text{if} \begin{cases} ?h \equiv v \\ NotVar_S(?h,v) \notin C_h \end{cases}$$

Figure 5: Synthesis inference rules

The function *Update*, used in the rules, takes a constraint (which might be a conjunction of dependent constraints) on the instantiated hole and updates it according to the constraint update algorithm described in §6. We use $Constrs(k)$ for the new constraints that are introduced on instantiating some hole with the symbol k. These come from the theory constraints associated with k. All paths defining positions in the 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 $Dom(?h)$ 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 $NotConst_S$/$NotVar_S$ constraint[5].

As we shall see, the synthesis algorithm maintains the invariant that, at each iteration, no constraint in C is violated (see lemma 3).

6 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. Below we define the function *Update* which is a lazy unfolding of the *Satisfies* relation.

We let *Update* on a disjunction of constraint be defined as: $Update(c_1 \vee c_2) = Update(c_1) \vee Update(c_2)$. We write p_h for the position of the instantiated hole $?h$ as before. The *Update* function has the following cases:

NotConst-violation

$$Update(NotConst_S(p_h,\ f)) \Longrightarrow \bot \qquad \text{if} \begin{cases} ?h \equiv s \\ hd(s) = f \end{cases}$$

NotConst-satisfied

$$Update(NotConst_S(p_h,\ f)) \Longrightarrow \top \qquad \text{if} \begin{cases} ?h \equiv s \\ hd(s) \neq f \end{cases}$$

NotVar-violation

$$Update(NotVar_S(p_h,\ v)) \Longrightarrow \bot \qquad \text{if} \begin{cases} ?h \equiv v \end{cases}$$

NotVar-satisfied

$$Update(NotVar_S(p_h,\ v)) \Longrightarrow \top \qquad \text{if} \begin{cases} ?h \equiv s \\ s \neq v \end{cases}$$

UnEqual-Fun

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

$$\text{if} \begin{cases} ?h \equiv f(?h_1 \ldots ?h_m) \end{cases}$$

UnEqual-Const

$$\begin{gathered} Update(UnEqual_S(p_h,\ p_1, \ldots, p_n)) \Longrightarrow \\ NotConst_S(p_1,\ k) \vee \ldots \vee NotConst_S(p_n,\ k) \end{gathered} \qquad \text{if} \begin{cases} ?h \equiv k \end{cases}$$

[5] 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.

UnEqual-Var

$$\begin{array}{l} \mathit{Update}(\mathit{UnEqual}_S(p_h,\ p_1,\ldots,p_n)) \Longrightarrow \\ \mathit{NotVar}_S(p_1,\ v) \vee \ldots \vee \mathit{NotVar}_S(p_n,\ v) \end{array} \qquad \text{if} \left\{ \begin{array}{l} ?h \equiv v \end{array} \right.$$

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)**.** *Assume the hole ?h in the partially synthesised term $t[?h]_{p_h}$ is instantiated by the symbol s. Then each constraint $c \in C_h$, is updated: $\mathit{Update}(c) = c'$. The updated constraint c' preserves satisfiability over grounding substitutions σ: $\mathit{Satisfies}((t[s]_{p_h})\sigma,\ c) = \mathit{Satisfies}((t[s]_{p_h})\sigma,\ c')$.*

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

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

   (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:

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

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

   Hence both the old and new constraints evaluate to True.

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

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

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

   Hence both the old and new constraints evaluate to False for an arbitrary substitution σ.

2. c is of the form $\mathit{NotVar}_S(p_h,\ v)$:
   Analogous to $\mathit{NotConst}_S$ case. Note that as v is not a meta-variable, it is not allowed to be instantiated by the grounding substitution σ. We can think of v as a constant.

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

   (a) Assume $?h$ is instantiated to constant k. The variable case is analogous. 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:

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

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

   Clearly $\mathit{Satisfies}(t', c')$ only evaluates to True when at least one of $\mathit{hd}(t'|_{q_i}) \neq k$ holds. Note that in this situation $\mathit{Satisfies}(t',\ c)$ will also be True, as it contains the corresponding conjuncts $t'|_{q_i} \neq k$.
   If $\mathit{Satisfies}(t', c')$ evaluates to False, then all its conjuncts are false, which means that $\mathit{hd}(t'|_{q_1}) = k \vee \ldots \vee \mathit{hd}(t'|_{q_n}) = k$ Hence all $(t'|_{q_i})\sigma$ must be equal. In this situation, all inequalities between $t'|_{q_i}$'s in $\mathit{Satisfies}(t',\ c)$, 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:

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

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

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

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[6]. In this case, 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 $Satisfies(t',\ c')$ 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, $Satisfies(t',\ c)$ 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.

7 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 (Irreducible after each instanitation)**.** *After each instantiation of a hole by the synthesis algorithm, the partially synthesised term t does not contain any redex that can be reduced by any rule.*

Proof: By contradiction. Assume there is a subterm, s, in t, that is reducible by some rewrite rule R_g: $g(x_1,\ldots,x_n) \rightarrow r$. We thus have $g(x_1,\ldots,x_n)\sigma \equiv s$. Then s must have the same top-level constant symbol as the rule, g, which must have been introduced by the rule **Constant** in Figure 5. This instantiation would have added the set of constraints associated with g, $Constrs(g)$, to the set of constraints C

[6]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: $1 \leq z \leq m$

associated with the term t that we are synthesising. We know that there is a constraint associated with the rewrite rule R_g in $Constrs(g)$.

Furthermore, there must have been one last hole $?h$ that was instantiated in s to make R_g applicable. By Theorem 2, the original constraint c_g may have been updated to a constraint c'_g, which evaluates to the same value on the instantiated term. 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 $NotConst_S$/$NotVar_S$ 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 becomes reducible by R_g, all disjuncts except c_{gh} must have been violated, and evaluated to false.

   $c_{gh} = NotConst_S(?h,\ k)$**:** This constraint was generated as the lhs of the rule contains the symbol k in position p_h. Hence, we must instantiate $?h$ to k for the rule to apply. Synthesis can do this by applying the **Constant** rule. However, the side-condition of the rule forbids such an instantiation. Hence, s cannot be synthesised.

   $c_{gh} = UnEqual_S(p_1, \ldots p_h, \ldots p_n)$: The lhs of the rule 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 variable or constant (otherwise $?h$ cannot be the last hole). This would have updated the constraint by the rules **UnEqual-const** or **UnEqual-var**, to n $NotConst_S$/$NotVar_S$ 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 R_g applicable.

2. p_h has been introduced through constraint updates:
   Some position above $?h$ will have been involved in a constraint $UnEqual_S(p1, \ldots p_n)$. All position up to the level on which $?h$ occurs must have been instantiated to equal terms, which in turn recursively introduce new $UnEqual_S$ constraints for each level. For the level of $?h$, the other holes must have been instantiated to variables or constants, 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 irreducible terms.*

Proof: By lemma 3, the synthesis algorithm maintains the invariant that no instantiation produces a subterm that is reducible by some rewrite rule. This obviously also holds for the final iteration, so each term produced is irreducible.

8 Conclusions and Further Work

We have presented a formal account of term synthesis in IsaCoSy, with a simpler constraint language that in the current implementation. We describe the constraint generation and synthesis machinery in a less algorithmic fashion than previously, abstracting away implementational details. This clarifies what the techniques does (and does not) do, and facilitates future re-implementation. Furthermore, the formalisation is helpful in comparisons between different approaches to theory formation, highlighting the essential features.

The formalisation 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.

As further work, we plan to include the simplified constraint language in the implementation of IsaCoSy. We also plan to extend the term-synthesis machinery of IsaCoSy to allow generation of novel recursive function definitions.

References

[1] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 230–239, Washington, DC, USA, 2004. IEEE Computer Society.

[2] B. Buchberger, A. Craciun, T Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkrantz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[3] L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *TPHOLs-17*, LNCS, pages 83–98. Springer, 2004.

[4] M. Hodorog and A. Craciun. Scheme-based systematic exploration of natural numbers. In *Synasc-8*, pages 26–34, 2006.

[5] M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 2010. (To appear).

[6] 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.