Moa Johansson, Lucas Dixon

April 17, 2008

Structured Inductive Formation of Theories

Abstract

1 Introduction

We are trying to build a program for synthesising inductive lemmas for theory development as an alternative to lemma speculation. We found that lemma speculation with middle-out rewriting is rarely applicable and in many of the cases where it is applicable, it fails to fully instantiate the lemma (BBN 1640, 1634). This failure leads to the intractable problem of trying to synthesise a term for the uninstantiated variables.

In this note, we suggest synthesising lemmas from the available constants in a 'bottom-up' fashion. The general idea is to build incrementally larger terms using from a set of constants in a given theory. They key idea which we believe will make this process tractable is to turn rewriting and term normalisation upside-down: only terms that do not match a rewrite rule or normalisation procedure will be generated. We will combine this with counter-example finding as well as memoisation techniques to prune out obviously false theorems. In terms of an implementation, these restrictions become constraints on the term-synthesis process and thus avoid a naive and inefficient generate and test style process.

We hope to be able to automatically generate inductive lemmas which will be needed by later proofs within an arbitrary given theory. An advantage of this approach is that user's are likely to be less concerned with the lemma-generation process taking a matter minutes, whereas when in the middle of an exploratory interactive proof waiting for several minutes is not acceptable. Lemma generation can also be performed in the background while the user performs interactive proof in the foreground. Alternatively, we might consider using the technique as an alternative to lemma speculation with middle-out reasoning for blocked proofs. In this scenario, lemmas would be build from the constants in the goal.

2 Notation

Term size counts how many constants and variables occur in the term. This is used to restrict synthesis. For instance, when synthesising equations, because equality is commutative operation, we only need to consider terms where the size of the LHS is greater-than or equal to the RHS.

Free variables are those that are universally quantified at the outermost level. A free variable x of type nat, written with Isabelle notation as $x :: nat$[1].

During synthesis, *holes* are the positions in the term tree which have not yet finished being synthesised. Holes will be represented by meta-variables, and are thus given names which are prefixed by a "?" character, following Isabelle's notation. For example, a partially synthesised term $?n + x = x$ has a single hole called $?n$. Holes will have various restrictions associated with them, such as a specified size, and restrictions on which constants and variables are allowed to occur inside them.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the Mathematical Reasoning Group.

[1]There are some open questions about needing a total ordering on term sizes or not, in which case variables can be treated as larger than constants of the same type.

3 Constraints on Synthesis

There are two main types of constraints we impose on holes during synthesis:

Symbol-Occurrence constraints specify the symbols that are allowed to occur in each argument position of each function-symbol. For example, below we restrict + to not allowing 0 or *Suc* to occur as the top symbol of the first argument.

Inequality constraints state that certain holes may not be instantiated to the same term. For example, when synthesising an equational statement, we do not allow the left and right hand sides to be the same.

Term-Size constraints come from structural rules like commutativity as well as from synthesising smaller lemmas first. Size constraints define the term size for a particular hole.

Constraints are dynamic in the following sense: when a hole is filled in by a new function symbol with further holes, the further holes may be given constraints that are derived from the constraints on the original hole.

The purpose of these cons taints is to ensure that no rewrite rule applies to a newly synthesised term, and thus only terms in normal form are generated.

In addition to these constraints, we can use discrimination nets to quickly check that we do not generate inductive conjectures which are instances of known theorems or which are known to subsume known false conjectures. This is the same idea used in IsaPlanner for avoiding repeated proof attempts during lemmas calculation and speculation.

We will use the domain of natural numbers as an example; in Isabelle these can be defined with the following data-type:

```
datatype nat = 0 | Suc nat
```

We will now give some examples of such constraints and note where some of the theorems that induce them come from.

3.1 Examples of Symbol-Occurrence Constraints

These constraints arise only from the LHS of the available rewrite rules and thus do not change throughout the synthesis of a term. They are updated when new rewrite rules are proved.

3.1.1 Function definitions

A number of rewrite rules will come from recursive function definitions. Here, we consider at the two equations defining +:

```
0 + n = n
(Suc n) + m = Suc(n + m)
```

Looking at these equations, we can see that it is not be productive to synthesise a term with a `0` or a `Suc` in the position of the first argument of +, as it would be possible to rewrite such a term using the existing definitions. To avoid search space explosion, we only want to generate terms in normal form: terms which cannot themselves be rewritten. Given the above equations, this means that the first argument of + is allowed to be instantiated to something in the set $\{Var, +\}$, while the second argument can be instantiate to one of $\{0, Suc, Var, +\}$, where Var is an arbitrary free variable.

Whenever we instantiate a hole to an application of +, the two new holes, representing the arguments of +, inherit restrictions on instantiations from these global restriction on +, in addition to any other restrictions they might have.

3.1.2 Injectivity

Isabelle will automatically derive an injectivity theorem for the datatype of natural numbers:

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

This induces a restriction on Suc under $=$: if we have Suc as the top symbol on one side of $=$, then Suc is not allowed at the top level on the other side. If this is not the case, the injectivity theorem can be applied as a rewrite rule.

This constraint is slightly different from the constraint above. Here Suc is only disallowed on the RHS of an $=$ sign in some cases, depending on the top-level symbol on the LHS. The constraint is dropped if the LHS is instantiated to something other than Suc.

3.2 Examples of Term-Size Constraints

There are two kinds of term-size constraints: restriction to a given size, written as $?x = n$, where n is the size; and ordering constraints written $?x \leq ?y$ which says that $?x$ must be of smaller term size than $?y$.

3.2.1 Basic constraints on term size

We envisage our algorithm attempting to generate progressively larger terms, starting from some minimal term size. Each hole will therefore always have a specified term size that its instantiation must satisfy. Instantiating a hole with some function symbol will propagate new size constraints to the new holes arising in the argument positions of the function.

3.2.2 Commutativity of Addition

If we know that a function if commutative, such as addition for natural numbers, $n + m = m + n$, then we can impose an ordering on its arguments during synthesise. For example, we can require that the first argument is $\leq$ the second. This avoids considering symmetries modulo commutativity during synthesis. More generally all structural rules give rise to similar restriction.

3.2.3 Symmetry of Equality

Like the commutativity of addition, but for equations, a sensible constraint is to require the size of the LHS to always be greater or equal to that of the RHS. This comes from the symmetry of equality $((x = y) = (y = x))$ and has the same form as commutativity.

An interesting result of this restriction is that we way be able to quite easily identify which synthesised lemmas can automatically be added as new rewrite rules.

3.3 Example Inequality Constraint: Reflexivity

Reflexivity is expressed in Isabelle as the rewrite rule:

$$(x = x) = True$$

To avoid this rule being applicable, we do not want to synthesise an equation with identical left and right-hand sides. This creates a constraint between the two holes occurring as arguments of =, ensuring they are instantiated to different terms. Of course, this might not be possible to establish until the term has been fully synthesised. Should we detect that the instantiations will be different (eg. different top level symbols), this constraint is dropped. Unlike the injectivity of *Suc*, we do not know in advance how many levels down the term we will have to look before the reflexivity constraint can be checked.

4 Example: Theory of Natural Numbers

Return to our minimal theory about natural numbers. We have three function symbols: *Suc*, + and =. We know that *Suc* is injective as the data-type package derives this. We also know that equality is reflexive as it's part of the meta-theory of HOL. We also assume that the basic defining equations have been considered.
Initially, addition has the following associated constraints:

Name:	$?x+?y$
Min size:	3
Allowed symbols:	$?x \in \{Var, +\}$, $?y \in \{0, Suc, Var, +\}$
Term-Size:	-
Inequality:	-

Recall the omission 0 and *Suc* from the symbols for the first argument come from the defining equations being treated as rewrite rules.
For the function = the initial constraints are:

Name:	$?l =?r$
Min size:	3
Allowed symbols:	$?l \in \{0, Suc, Var, +\}, ?r \in \{0, Suc, Var, +\}$
	$(?l \equiv (Suc\ ...)) \Rightarrow (?r \in \{0, Var, +\})$ (*suc_inject*)
	$(?r \equiv (Suc\ ...)) \Rightarrow (?l \in \{0, Var, +\})$ (*suc_inject*)
Term-Size:	$?l \leq ?r$ (*symmetry*)
Inequality:	$?l \neq ?r$ (*reflexivity*)

Finally the initial constraints for *Suc*:

Name:	*Suc* $?m$
Min size:	2
Allowed symbols:	$?m \in \{0, Suc, Var, +\}$
Term-Size:	-
Inequality:	-

For this example, we assume that we want to synthesise equations. This means we have to start synthesising terms of size 3, with = as the top level symbol and two holes, each of size 1. The initial term is thus $\underbrace{?x_1}_{size\ 1} = \underbrace{?x_2}_{size\ 1}$. The holes, represented by the meta-variables $?x_1$ and $?x_2$ will, in addition to their specified size, inherit the restrictions specified for the arguments of = above.

We can generate two terms of size 1, the constant 0 and a single variable, a. Below are the terms that could be built to fit the template above:

- $0 = 0$ and $a = a$ are discarded due to the inequality constraint from reflexivity.
- $a = 0$, $0 = a$ and $a = b$ are accepted, but we can find a counterexamples.

The size of the terms synthesised is incremented, while keeping the restriction that the LHS is larger than the RHS. When the term size is increased to 4, the LHS of the equation is allowed to be of size 2, while the RHS remain of size 1. We generate five conjectures (we use / to separate alternative right hand sides): $Suc\ 0 = 0/a$ and $Suc\ a = 0/a/b$. None of these three pass counter-example checking.

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

$$\underbrace{?x_1}_{size\ 2} = \underbrace{?x_2}_{size\ 2}$$

or

$$\underbrace{?y_1}_{size\ 3} = \underbrace{?y_2}_{size\ 1}$$

.

From the size restriction, the former can only attempt to generate terms of the form $Suc\ ?x = Suc\ ?y$, but this is disallowed due to the injectivity of Suc, so no terms will be generated for this case.

The second template give rise to one theorem: $a + 0 = a$. If this can be proved, we add it to the set of rewrite rules, and can conclude that we no longer should generate terms where 0 is the second argument to $+$.

We already avoid generating any terms where 0 is the first argument of $+$, as this is subsumed by the definition. We will however generate some other non-theorems that need to be rejected by counter-example checking:

$$a + 0 = 0/a$$

$$a + b = 0/a/b$$

$$Suc(Suc\ 0) = 0/a$$

$$Suc(Suc\ a) = 0/a/b$$

In total, for terms of size 5, we generate 11 conjectures, of which one is a theorem and the rest dismissed by counterexamples. A naive version would generate 20 conjectures.

For terms of size 6 we really start to notice a difference between our approach and a naive variant. From the size restriction, we can now generate terms of the form:

$$\underbrace{?y_1}_{size\ 3} = \underbrace{?y_2}_{size\ 2}$$

or

$$\underbrace{?y_1}_{size\ 4} = \underbrace{?y_2}_{size\ 1}$$

For the first template we can only build three terms: $a + b = Suc\ 0/Suc\ a/Suc\ b$. This is thanks to injectivity of Suc, and the fact that we no longer try to generate terms with 0 as an argument to $+$ (due to the theorem $a + 0 = a$, proved in the last iteration).

For the second template we generate 11 terms, none of which is a theorem:

$$a + (Suc\ 0) = 0/a/b$$

$$Suc(a + b) = 0/a/b$$

$$a + (Suc\ b) = 0/a/b$$

$$Suc(Suc(Suc\ 0)) = 0/a$$

$$Suc(Suc(Suc\ a)) = 0/a/b$$

In total 16 conjectures are generated, all of which fail counter-example checking, compared to 29 conjectures for the naive approach.

This process continues either up to some specified term size, or until no more terms are allowed to occur as arguments to any of the functions. As theorems are discovered, we can reduce the number of allowed symbols for the arguments of the functions involved. When no more arguments are allowed for a function, we can furthermore remove it from occurring in an argument for another function.

5 More Related Ideas

Interesting areas for further investigation include:

- Generalise the idea of using commutativity rule to give a size restriction to any rules involve argument position changes.
- Make use of other representations for synthesis. For example, if an operator is AC, then its arguments can be placed into a multi-set and thus all structural properties can be absorbed by the representation.
- Consider other restrictions for synthesis. For example, the rewriting restriction of requiring the variables in the RHS to be a subset of those in the LHS could be used to try and synthesise only rewriting rules. In the example above, this would reduce the number of terms generated by two or three for each iteration.
- Consider the question of completeness. So far we have been considering machinery that generates all terms and thus has a relative completeness w.r.t. the rewrite rules. When the rewrite rules provide a normal form, then this form of synthesis is complete. However, it is interesting to consider other cases also, and even to wonder about the importance of completeness.
- Consider the question of termination: under what cases does synthesis terminate? Can these theories be classified? What constraints on synthesis should be imposed to ensure termination - an obvious candidate being a maximal size.

6 Summary

We have outlined what we hope will be a tractable way to synthesis lemmas for an inductive theory. It is based on generating only "new" terms that cannot themselves be written and which are not trivially false. We've run through a hand example in Peano arithmetic with positive results: we only get useful lemmas and we get all the lemmas we need. There are many of directions the work can go in, although we are first aiming for a basic implementation, which is currently under development.