

Properties of IsaCoSy's Constraint Generation Algorithm

Moa Johansson, Lucas Dixon, Alan Bundy

February 16, 2010

Abstract

This report states and sketches the proofs for some important properties of IsaCoSy's constraint generation algorithm. We claim that for any rewrite rule, the constraint generation algorithm produces constraints that will prohibit the synthesis of exactly the terms containing a redex which can be rewritten by that rule (and no others).

1 Introduction

IsaCoSy is a theory formation system for inductive theories. A full account of the system can be found in the journal paper by these authors [1]. This report is a complement to the journal paper, and contains some additional proofs about properties of IsaCoSy's constraint generation algorithm which prevents the synthesis of any reducible terms.

2 Properties of the Constraint Generation

This is brief exposition of the main theoretical results regarding IsaCoSy's constraint generation algorithm. It also helps understand the high level goals of the constraint language. We first introduce some notation and terminology:

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

- Positions can be uniquely named using the datatype described in §4.2 of [1]. We shall assume the convention of referring to unique positions by names of the form pos_i . In particular, we refer to all argument positions in this manner.

The constraint language assumes a synthesis algorithm that attaches the constraints associated with a function symbol to every occurrence of that function symbol in the synthesised term. The synthesis algorithm then only produces terms that satisfy the constraints. We give such an algorithm in §8 of [1].

Using these definitions and assumptions, a *sufficient coverage* theorem is proved. This states that the constraint generation algorithm generates enough constraints from a rule such that every term that can be rewritten by that rule violates its constraints. This is then extended to show an *exact coverage* theorem which ensures that the constraints are violated exactly when a term contains a redex matching the rule used in the formation of the constraints.

Before proceeding with the proof of these theorems, we make two observations about the constraint language:

- Since terms are built in a top-down structural way, they monotonically add constraints. Hence it is enough to show that a given term t will not be synthesised in order to prove that no term c , containing t as a subterm, will be synthesised.
- The *IfThen* constraint is logically equivalent to a *NotSimult* constraint with exactly two dependent constraints on a particular subterm. A constraint

$$C_x : \text{IfThen}(pos_i, f, C_j)$$

can be ‘normalised’ to the equivalent constraints

$$C_{x'} : \text{NotSimult}((pos_i, C_i), (pos_j, C_j))$$

and

$$C_i : \text{NotAllowed}(pos_i, f)$$

where pos_i is the position of the function symbol f , and pos_j is some argument position of f . The distinction between *IfThen* and *NotSimult* is for pragmatic reasons, but for the purpose of a theoretical analysis we can ignore *IfThen*.

With these observations in mind, we can state the coverage theorem more formally and then sketch its proof:

Theorem 1 (Sufficient Coverage) *Given a rule $l \rightarrow r$, if a term t contains a redex of l , then t violates $\text{Constraints}(l \rightarrow r)$.*

Proof The proof is by case analysis on the constraints generated by the rule’s left-hand side. Without loss of generality, we may assume the left-hand side l is of the form $f(s_1, \dots, s_k)$. There are then three possible cases for the constraint attached to the symbol f :

1. No explicit constraint is generated, but the symbol f is removed from the domain of all other function symbols. In this case, f is a constant or all of its arguments s_1, \dots, s_k are distinct variables. A term with a redex of l must contain f . Such a term will violate the constraints as there is no domain containing f .
2. A constraint of the form $NotAllowed(pos_i, c)$ is attached to f , where pos_i corresponds to the position of one f 's arguments s_1, \dots, s_k . A term with a redex of l will contain a subterm of f with c as the argument at pos_i . Such a term will violate the generated constraint.
3. A constraint of the form $NotSimult(\dots, (pos_i, C_i), \dots)$ is attached to f , where pos_i may refer to any position in l . Furthermore, each C_i is either a constraint of form $UnEqual$ or $NotAllowed$. Recall that child- $NotSimult$ constraints get merged with their parent and $VarNotAllowed$ constraints only occur during synthesis, not during initial constraint generation from a rewrite rule. Observe that a $NotSimult$ constraint is violated iff every sub-constraint is violated.

An arbitrary $UnEqual$ sub-constraint refers to a set of positions in a term for a given variable. A redex will have to have the same term at each location that corresponds to an occurrence of the variable. Such a term will violate this $UnEqual$ sub-constraint. For an arbitrary sub-constraint of the form $NotAllowed$, the constant it refers to will have to occur at the corresponding location in the redex and will hence violate this sub-constraint. Thus, a redex of a rewrite rule which produces a $NotSimult$ constraint will violate every sub-constraint of the $NotSimult$, and hence such a redex will violate the $NotSimult$ constraint.

For each case, we have observed the shape of the term generating the constraints implies that a redex in a synthesised term will violate the constraints. This completes the proof: if a rewrite rule matches a subterm of a conjecture, then the constraints IsaCoSy generates from this rewrite rule will prevent the synthesis of the conjecture.

This result ensures that reducible terms are not synthesised. However, it does not show that all irreducible terms are synthesised. For example, the algorithm might generate constraints that stops the synthesis of every term. The *no over-coverage* theorem states that the constraints generated from a rule are only violated by a term that contains a redex of that rule.

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

Proof The proof is again by case analysis on the constraints generated by the rule's left-hand side. The cases are identical to the sufficient coverage theorem: the constraints imply a structure on a term that is violated by them. The proof is concluded by observing that each case in the coverage proof can be inverted in the sense that a term violating the constraints also implies a redex within the term.

The combination of the no over-coverage theorem and the sufficient coverage theorem proves exact coverage, which is the desired specification for such a constraint generation algorithm:

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

3 Conclusion

We have shown that IsaCoSy's constraint language captures constraints from a rewrite rule that always disallows the synthesis of any term reducible by that rule. A more formal account of IsaCoSy's constraint language and the above properties is left as further work. We also plan to revise and simplify the constraint language for the next version of IsaCoSy. This will facilitate the formal analysis of the system.

References

- [1] M. Johansson, L. Dixon, A. Bundy. *Conjecture Synthesis for Inductive Theories*. Submitted to the Journal of Automated Reasoning, 2010.