Automated SAT Encoding for Termination Proofs with Semantic Labelling*

Alexander Bau, René Thiemann, and Johannes Waldmann

HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany
{abau|waldmann}@imn.htwk-leipzig.de

University of Innsbruck, Austria, rene.thiemann@uibk.ac.at

Abstract

We discuss design choices for SAT-encoding constraints for termination orders based on semantic labelling and unlabelling, linear interpretations, recursive path orders with argument filters, within the dependency pairs framework.

We specify constraints in a high-level Haskell-like language, and translate to SAT fully automatically by the CO4 compiler. That way, constraints can be combined easily.

This allows to write a single constraint for *find a model, and a sequence of ordering constraints for the labelled system, such that at least one original rule can be removed completely.* Reliability is achieved via certification of generated proofs.

The size of the resulting propositional logic formulas can be reduced by strengthening the constraints. We discuss an encoding of finite maps via patterns.

1 Introduction

Termination of a (rewrite) relation $\to$ can be shown by embedding $\to$ in a well-founded order $>$. Such an order can be given syntactically, by comparing the shape of terms, and occurences of symbols, as it happens in recursive path orders. Another option is to define the order semantically, where each term is assigned an element of a well-founded algebra, by giving interpretations of symbols. For the domain of such an algebra, we can use, e.g., numbers, vectors, matrices. Then there are methods for constructing a well-founded order by combination of others, for example, using the lexicographic product. We also have methods that transform a termination problem into another. Semantic labelling uses a finite algebra that is a model for the rewrite relation, to assign labels to function symbols, thereby increasing the signature, allowing a more fine-grained analysis of the transformed system. The dependency pairs transformation also transforms a rewrite system, so that sequences of "function calls" can be analyzed.

When we write a program to prove termination automatically, we prescribe a certain termination proof method, and the task of the program (the "prover") is to fill in all the parameters. E.g., if we want to use interpretations by linear functions, then suitable coefficients of such functions must be determined. If we want to use a recursive path order, then a suitable precedence of symbols is needed. In each case, suitability can be described by a formula in predicate logic, which we call a "termination (ordering) constraint". Thus, the termination prover is actually a "solver" of constraints.

The present paper is not about new methods of proving termination, but about the pragmatics of writing down termination constraints. As with all source code, one goal is efficiency of execution, and another goal is efficiency of expression, leading to readability and maintainability. We advocate the use of our high-level declarative constraint programming language

* Supported by ESF grant 100088525 and FWF project P22767.

CO4, that comes with a compiler that targets propositional satisfiabiliy (SAT).

SAT encoding is a successful method of solving finite domain constraints. Some ordering constraints have finite domain by definition (a finite model, a precedence for a finite signature), others do not, e.g., because they involve numbers. SAT encoding can still be used, by restricting to some finite subset, e.g., numbers of certain bit width.

The current state of SAT encoding (for termination) can be described as: it is successful (several successful termination provers use it) but it is also laborious. This is mainly due to explicit manipulation of propositional variables, corresponding to manual assignments from identifiers to memory locations in low-level (assembly) programs.

This increases the work in encoding combined constraints. E.g., argument filtering and path order comparison are handled at the same time, while they are conceptually independent. A filter π denotes a mapping F^π from terms to terms (that removes some nodes and subterms). Then, mapped terms are compared w.r.t. a *p*ath *o*rder: $F^\pi(s) >_{PO} F^\pi(t)$. In the source code of our termination prover, this is literally expressed as the Haskell expression

```
case order of
  FilterAndPrec f p ->
    lpo p (filterArgumentsDPTerm f lhs) (filterArgumentsDPTerm f rhs)
```

In contrast, the encoding described in [5] combines these steps, so that the encoding of F^π is "fused" into the encoding of the path order, realizing a relation $>^\pi_{PO}$ on terms.

We implement the following constraint for termination proofs that use semantic labelling [11], path orders and linear interpretations in the context of the dependency pairs framework [1]. The known input is a DP problem (set of (dependency) pairs and set of rewrite rules). The unknown is a pair of an interpretation into a finite domain, and a termination order (on the labelled signature), such that the interpretation is a model for the rewrite rules; all labelled pairs and rules are weakly compatible with the order, and for at least one pair, all its labelled versions are strictly compatible with the order.

The order is a lexicographic combination of basic orders, where a basic order is either a recursive path order with argument filter, or built from a linear interpretation.

That means our constraint can describe a proof step where we first apply semantic labelling, then remove a number of labelled pairs by using several orders in succession (e.g., first, a linear interpretation, then a path order, or path orders with different argument filters), where at each step, we recompute usable rules, and finally unlabel. Proofs are certified by CeTA [9].

The source code of our termination prover (and the SAT compiler CO4 [4]) is available at `https://github.com/apunktbau/co4`

The present paper builds on [3]. New contributions are: extension from string rewriting to term rewriting, applying the dependency pairs framework, and certifiable proofs.

2 Structured Finite Domain Constraints and their SAT Encoding

We briefly review the concepts of constraint programming with CO4. A parametric constraint is given as a function `c :: P -> U -> Bool`, written in a subset of Haskell, where `P` is a domain of parameters, and `U` is the domain of unknowns. The constraint `c` is compiled to a function `cc :: P -> CNF` such that `cc p` gives a propositional logic formula `f` in conjunctive normal form such that from a satisfying assignment of `f`, an object `u :: U` can be reconstructed with `c p u == True`.

CO4 handles algebraic data types (`data`), and case distinctions by pattern matching. A set of unknown objects of an algebraic data type is represented as a tree, where each node

contains propositional variables. Each assignment determines a (binary) number, which in turn determines the constructor in this node. Pattern match on the constructor is realized by compiling all branches, and adding selector functions. When merging results from different branches of a case distinction, the corresponding trees are overlapped. This allows to handle finite domain constraints in Haskell notation. For infinite types (lists, trees), we can specifiy finite subsets by restricted recursion. While the core language of CO4 is first-order, we allow higher order functions and remove them by specialization.

In our application, the main constraint is `c :: DPProblem -> Proof -> Bool`, where `c d p == True` if `p` proves that at least one pair can be removed from the DP problem `d`. We use these types:

```
data Proof = Proof (Model Symbol) [ UsableOrder (Symbol,Label) ]
type UsableOrder key = (UsableSymbol key, TerminationOrder key)
type UsableSymbol key = Map key Bool
```

In particular, a `Proof` object consists of a model m, and a list $[(u_1, o_1), \ldots, (u_k, o_k)]$. Here, o_i is an order, and u_i describes an over-approximation of the symbols that are usable w.r.t. the dependency pairs that remain after removing those that are decreasing w.r.t. the lexicographic product of o_1 to o_{i-1}. For orders, we use

```
data TerminationOrder key = FilterAndPrec (ArgFilter key) (Precedence key)
                          | LinearInt (LinearInterpretation key)
data Index = This | Next Index
data Filter = Selection [ Index ] | Projection Index
type ArgFilter key = Map key Filter
data Precedence key = EmptyPrecedence | Precedence (Map key Nat)
```

3 Encoding of Finite Maps with Patterns

We discuss in more detail the cost of SAT-encoding of maps (lookup tables). These are used for finite algebras (as models of rewrite systems). The basic functionality is

```
lookup :: Map k v -> k -> Maybe v
```

Let us estimate the cost of the encoding. Consider the (common) case that the key to be looked up is unknown (i.e., encoded). Then the naive algorithm is to compare the key with each key in the map. This requires a linear (in the size of the map) number of (encoded) key comparisons. This is exactly what CO4 generates from the following program.

```
type Map k v = [(k,v)]
lookup x m = case m of
    [] -> Nothing
    (k,v): m' -> if x == k then Just v else lookup x m'
```

We might think of a balanced tree instead of a list, as in

```
data Map k v = Leaf | Branch k v (Map k v) (Map k v)
lookup x m = case m of
    Leaf -> Nothing
    Branch k v l r -> if x == k then Just v else
        if x < k then lookup x l else lookup x r
```

Note that the result of `x < k` is unknown when generating the formula, since `x` is encoded. This means that both branches of `if` have to be encoded. Again, this is what CO4 does

when translating the program. This results, again, in linear formula size. We conclude that using balanced trees for lookups with encoded keys is not helpful.

We can still achieve smaller formulas by giving up on completeness: We encode only a subset of all maps. We restrict to encoding maps where keys are lists of domain elements (that is, `Map [d] v`) as it happens in semantic labelling.

We represent such a map by a list of patterns that are matched from left to right. If we are lucky, the model is representable in the subset.

```
data Match d = Any | Exactly d
type MapList d v = [ ( [Match d] , v) ] --  represents Map [d] v
```

For instance, the pattern `[(([Any,Exactly 0],0), ([Any,Any],1)]` represents the function `[([0,0],0), ([0,1],1), ([1,0],0), ([1,1],1)]`.

As discussed earlier, we do have linear cost for the lookup of an encoded key. This cost increases for pattern matching, but the plan is to reduce it by making the number of patterns (much) smaller than the domain of the function.

In the extreme case, we use just one pattern `[Any,Any,... Any]`. This will give a model where all symbols are interpreted by constant functions.

4 Certification

The approach of iteratively applying "1) labelling, 2) applying several orders with usable rules, 3) unlabelling" is not easy to certify, since if one would allow arbitrary sound termination techniques in 2), the whole approach would be unsound: the problem is that unlabelling on its own is unsound, cf. [9, Example 4.3]. To solve this problem, [9] utilizes a dedicated semantics for DP problems w.r.t. semantic labelling.

But since this semantics was hard to extend, CeTA is now based on a more general semantics of DP problems which borrows ideas from relative rewriting [10], where there are relative DP problems with strict and weak pairs and rules. Then all unlabelling steps (UL) can be eliminated as follows: CeTA automatically transforms every proof of the form $(\mathcal{P}_0, \mathcal{R}_0) \overset{\text{SL}}{\hookrightarrow} (\mathcal{P}_1, \mathcal{R}_1) \overset{\succ_1}{\hookrightarrow} \cdots \overset{\succ_{n-1}}{\hookrightarrow} (\mathcal{P}_n, \mathcal{R}_n) \overset{\text{UL}}{\hookrightarrow} (\mathcal{P}', \mathcal{R}')$ into the following proof: first, a *split* processor is applied on $(\mathcal{P}_0, \mathcal{R}_0)$ which returns two new DP problems: the resulting DP problem $(\mathcal{P}', \mathcal{R}')$ and a relative DP problem $(\mathcal{P}_0 - \mathcal{P}', \mathcal{P}', \mathcal{R}_0 - \mathcal{R}', \mathcal{R}')$ where the first and third components are strict rules which have to be deleted. Then semantic labelling (SL) is applied on this relative DP problem and afterwards all orders $\succ_1, \ldots, \succ_{n-1}$ are used to finally get the DP problem $(\varnothing, \mathcal{P}_n, \varnothing, \mathcal{R}_n)$. Termination of this relative DP problem is then trivially proven as it does neither contain strict pairs nor strict rules.

All generated proofs have been certified via this approach, though initially problems occurred: the termination tool had bugs in its CPF-export; and CeTA rejected some valid proofs due to a buggy implementation of the transformation to eliminate unlabelling steps.

5 Related Work and Discussion

Semantic labelling had been used in "early" (2006) termination competitions [8] in termination provers Torpa (Zantema), Jambox (Endrullis), in TPA (Koprowski), Teparla (van der Wulp). Here, Jambox used SAT encoding, TPA used predictive labelling [7], and Teparla used recursive labelling with Boolean models.

Except for TPA, we assume that these early implementations used some kind of generate-and-test approach, where a model is found in one step, and in an independent step, a

termination proof is attempted for the labelled system. This means that often, several models need to be tried, and trivial models are to be excluded somehow.

Currently, semantic labelling with finite models is implemented in AProVE (Giesl et. al) which still uses a generate-and-test approach, but benefits from using several termination techniques between labelling and unlabelling. Also the complexity tool TcT (Avanzini et. al) uses semantic labelling, and both TcT and TPA are similar to our approach: they perform a combined SAT search for suitable models and orderings [2, 7]. However, both TcT and TPA use a manual encoding to SAT and do not support certifiable output for labelling.

Our current implementation uses "cheap" termination methods first (SCC decomposition, arctic matrices with small bit width and small dimension), and semantic labelling, as described here, only as a "last resort". We remark that proof search is nicely controlled in the `LogicT IO` monad [6]. We are currently evaluating experimentally the influence of different choices and shortcuts in the formulation of the termination constraint. Results will be made available from `http://www.imn.htwk-leipzig.de/~abau/wst2014.html`. We believe that Matchbox is the first to produce a certified proof of termination of TRS/AProVE/JFP_Ex31. This termination problem had been solved by AProVE and Jambox in 2008, but not later.

Our approach allows to SAT-encode large constraints automatically. This is also the drawback: resulting formulas can get huge, and pose a challenge to SAT solvers. This motivates to work further on more efficient compilation in CO4 on the one hand, but also on tools (type systems) for statically analyzing the "circuit complexity" (the size of the generated formula) of constraint programs.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

2 Martin Avanzini. POP* and semantic labeling using SAT. In *ESSLLI Student Sessions*, volume 6211 of *LNCS*, pages 155–166. Springer, 2009.

3 Alexander Bau, Jörg Endrullis, and Johannes Waldmann. SAT compilation for termination proofs via semantic labelling. In *WST 2013*, 2013.

4 Alexander Bau and Johannes Waldmann. Propositional encoding of constraints over tree-shaped data. *CoRR*, abs/1305.4957, 2013.

5 Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reasoning*, 49(1):53–93, 2012.

6 Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*, pages 192–203. ACM, 2005.

7 Adam Koprowski and Aart Middeldorp. Predictive labeling with dependency pairs using SAT. In *CADE*, volume 4603 of *LNAI*, pages 410–425. Springer-Verlag, 2007.

8 Claude Marché and Hans Zantema. The termination competition. In *RTA*, volume 4533 of *LNCS*, pages 303–313. Springer, 2007.

9 Christian Sternagel and René Thiemann. Modular and certified semantic labeling and unlabeling. In *RTA*, volume 10 of *LIPIcs*, pages 329–344. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

10 Christian Sternagel and René Thiemann. A relative dependency pair framework. In *Proc. WST'12*, pages 79–83, 2012.

11 Hans Zantema. Termination of term rewriting by semantic labelling. *Fundam. Inform.*, 24(1/2):89–105, 1995.