SAT compilation for Termination Proofs via Semantic Labelling

Alexander Bau[1], Jörg Endrullis[2], and Johannes Waldmann[1]

1 **HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany**
2 **Vrije Universiteit Amsterdam, The Netherlands**

Abstract

For the termination method of rule removal by semantic labelling, matrix interpretations and then unlabelling, we give a purely functional specification and apply our CO4 constraint compiler to automatically generate a propositional encoding. This extends a "manual" SAT encoding of this method in Jambox (Endrullis, 2008): we allow sequences of interpretations during labelling.

1 Introduction

Finding parameters for termination proof methods is a constraint satisfaction problem: given a rewrite system, we are looking for a precedence for a path order, or for coefficients of interpretations, etc.

For (finite domain) constraint programming, SAT (propositional satisfiably) takes the role of an assembly language: it gives direct access to the machine (i.e., powerful and highly optimized SAT solver), but it is cumbersome and error-prone for actual programming. Instead, we want to use a high-level specification language, to increase expressiveness and safety.

Two of us (Bau, Waldmann) are developing the CO4 language and compiler [1] that translates Haskell specifications to SAT formulas. We have previously applied CO4 for finding loops in string rewriting. With respect to a manual SAT encoding of TTT2 [5], our compiled formula is larger by a factor of < 10, with a similar factor for run-times of the SAT solver, but the source code of the constraint system is just 1/3 in size.

In this note, we report on an application of CO4 in termination of string rewriting, using the method of semantic labelling, interpretations, and unlabelling. This is well-known [6], and it had been implemented in several termination tools already: Teparla and Torpa used semantic labelling in $\approx$ 2006, by a stochastic search for models.

Jambox [2] $\approx$ 2007 used a "manual" SAT encoding for the following constraint: given a rewrite system R, there is a model M for R and an interpretation I for the M-labelled system R_M and a non-empty $S \subseteq R$ such that each rule in S_M is strictly I-decreasing. We then have relative termination $\mathrm{SN}(S/R)$, that is, we can remove S. Correctness of this method had been formalized for CeTA [4].

We now extend Jambox' implementation by allowing for a sequence of interpretations for the labelled system. This is a conceptually simple modification: instead of one order, use a lexicographic product of several orders. Using the CO4 language, this modification can be expressed directly in the source code (see function `lexi`), and all extra encoding is done by the compiler.

2 Semantic Labelling and Unlabelling

An algebra A is a *model* for a rewrite system R over signature Σ if for each $s \to_R t$, we have $A(s) = A(t)$. If we have a model A for R, we construct a labelled version R_A of R where each function symbol is labelled by the value(s) of its argument(s).

▶ **Example 1.** We consider the string rewriting system $R = \{aa \to aba\}$ over the signature $\Sigma = \{a, b\}$. The algebra A with domain $D = \{0, 1\}$, and interpretation $a_A = \{(0, 1), (1, 1)\}, b_A = \{(0, 0), (1, 0)\}$, is a model for R, and R_A is $\{a_1a_0 \to a_0b_1a_0, a_1a_1 \to a_0b_1a_1\}$.

Termination of R and R_A are equivalent (under some conditions that are true for string rewriting). The point of the method is that termination of R_A may be easier to show that termination of R since we increase the signature, and have more room to maneuver.

In the example, we can see that the number of occurrences of a_1 decreases in each rule application. This can be verified by the (linear, additive) interpretation $\{(a_0, 0), (a_1, 1), (b_1, 0)\}$ for R_A. There is no such interpretation for R.

3 SAT compilation with CO4

CO4 [1] is a high-level declarative language for describing constraint systems. The language is a subset of the purely functional programming language Haskell [3] that includes user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order polymorphic types.

Source programs operate on algebraic data (like Booleans, List, Trees) which we call *concrete values*. CO4 compilation creates programs that operate on *abstract values*. An abstract value $\in \mathbb{A}$ represents a set of concrete values of the same type. An abstract value is a tree, where each node contains a sequence of propositional logic formulas. Given a truth assignment σ, the sequence of truth values of these formulas under σ gives a binary number that denotes a constructor. Doing this for each node, a concrete value $\texttt{decode}(a, \sigma)$ is determined.

A high level, parametric constraint system `constraint :: P -> U -> Bool` written in CO4 represents a predicate on `U` depending on a parameter $p \in$ `P`. p is not known a-priori. The compilation and evaluation of a CO4 program is a staged process:

1. The original program, operating on *concrete values* $\mathbb{C}$, is compiled into an abstract program that operates on *abstract values* $\mathbb{A}$.
2. Given an parameter p, the abstract program is evaluated, resulting in an abstract value a (representing a Boolean).
3. The formula $a =$ `True` is given to an external SAT solver. It tries to find a satisfying assignment σ.
4. A value $u \in$ `U` is reconstructed by σ, so that `constraint` p u `= True`.

Values that depend on the u parameter of the top-level constraint are not known during abstract evaluation. Abstract evaluation of case-distinctions on those values has to evaluate (abstractly) all branches and then merge the results into a single abstract value.

Natural numbers may be defined as list of Booleans `type Nat = [Bool]` in CO4. For naturals with bit-width k this encoding ends up in abstract values with depth k. A user-defined function on naturals would require k pattern matches to inspect a number: this leads to long runtimes of the compiled program and the resulting formulas would be large. To avoid those issues, CO4 provides built-in naturals, where a number of bit-width k is encoded by an abstract value with k flags and no arguments. CO4 provides arithmetic and relational operations on naturals as well.

4 Implementation

Complete source code of the termination method is available as part of `https://github.com/jwaldmann/matchbox` (file `MB/Label/SLPO.standalone.hs`).

Here, we just indicate the main types and functions. For string rewriting, we have

```
type Symbol = [Bool]; type Word = [Symbol]
data Mode = Strict | Weak ; data Rule = Mode Word Word ; type SRS = [Rule]
```

where lists are built-in (with their usual Haskell definition). A symbol is represented as a binary string. The length of that string is known at run-time (when the implementation sees the signature of the rewrite system) but not at compile-time.

We often access information that belongs to a symbol. We use binary trees where the symbol encodes a path from the root to a leaf

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
get :: Tree a -> Symbol -> a
get t p = case p of
    []   -> case t of Leaf x -> x
    x:p' -> case t of Branch l r ->
         get (case x of False -> l ; True -> r) p'
```

Symbols will be labelled according to an unknown model, but we know the size of the model. We represent a labelled symbol as the concatenation of a list of known booleans (the original symbol) with a list (of known length) of unknown booleans (the label).

Note that the pattern matches on `p` and on `t` can be resolved at run-time (when generating the SAT formula), but the pattern match on `x` has to be encoded (since `x` depends on the model, which is determined by the solver).

We also represent finite algebras (of size 2^n) the same way: the domain is the set of bit strings of length n, and for each symbol, we have a function from domain to domain:

```
type Model = Tree Func; type Func = Tree Value; type Value = [Bool]
```

We compose functions (and give the most general type):

```
timesF :: Tree a -> Func -> Tree a
timesF f g = case g of
    Leaf w -> Leaf (get f w)
    Branch l r -> Branch (timesF f l) (timesF f r)
```

Given an algebra, we compute the labelled version of rewrite system, as a list of systems (for each original rule, one sub-system containing all its labelled versions). Also, each side of each rule is annotated with its value in the model. These values will be constrained to be equal.

```
labelled :: SRS -> Model -> [ [ ((Value,Value), Rule) ] ]
labelled srs mod =
    let ks = keys ( leftmost mod )
        labelRule u k = case u of
            (l,r) -> case labelledW mod l k of
                ( ltop, l' ) -> case labelledW mod r k of
                    ( rtop, r' ) -> ((ltop,rtop),(l',r'))
    in  map ( \ u ->  map ( \ k -> labelRule u k ) ks    ) srs
```

We omit details on interpretations, and come back to the lexicographic comparison mentioned in the introduction:

```
data Comp = Greater | GreaterEquals | None
comp :: Interpretation -> Rule -> Comp
comps :: [ Interpretation ] -> Rule -> Comp
comps ints u = lexi (map ( \ i -> comp i u) ints )
lexi :: [Comp] -> Comp
lexi cs = case cs of
  [] -> GreaterEquals
  c : cs' -> case c of
    Greater -> Greater; GreaterEquals -> lexi cs'; None -> None
```

This shows the expressiveness of the CO4 language. — The main constraint is

```
data Label = Label Model [ Interpretation ] [ Bool ]
constraint :: SRS -> Label -> Bool
```

where `constraint srs (Label mod ints flags)` is True iff `mod` is a model for `srs` and the lexicographic combination of `ints` is compatible with the labelled system, and removes the flagged rules (of the original system) completely, and at least one rule is flagged.

5 Performance

We describe work-in-progress, so we don't have complete performance data on a larger set of problems. We give an example (that shows the power of the method) and a comparison (that shows the quality of the SAT compilation).

▶ **Example 2.** Our implementation produces this termination proof for $a^2b^2 \to b^3a^3$:

```
matchbox ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l2,2 --dim 1 --bits 4 --nat
   # 2 bits for model values, 2 interpretations for labelled system
CNF finished (#variables: 21483, #clauses: 82956)
Solver finished in 4.152 seconds (result: True)

model:  3b0 2b1 3b2 1b3
        2a0 0a1 0a2 0a3

labelled system: a0 a1 b3 b0 -> b1 b3 b2 a0 a2 a0
                 a0 a1 b3 b2 -> b1 b3 b0 a2 a0 a2
                 a0 a3 b2 b1 -> b1 b3 b0 a2 a0 a1
                 a0 a2 b1 b3 -> b1 b3 b0 a2 a0 a3

natural matrix interpretation 1    natural matrix interpretation 2
    [(b0,x -> [[1]] * x +  [[0]])      [(b0,x -> [[1]] * x +  [[1]])
    ,(b2,x -> [[3]] * x +  [[1]])      ,(b2,x -> [[5]] * x +  [[9]])
    ,(b1,x -> [[1]] * x +  [[4]])      ,(b1,x -> [[1]] * x +  [[0]])
    ,(b3,x -> [[1]] * x +  [[1]])      ,(b3,x -> [[1]] * x +  [[4]])
    ,(a0,x -> [[1]] * x +  [[0]])      ,(a0,x -> [[1]] * x +  [[0]])
    ,(a2,x -> [[1]] * x +  [[1]])      ,(a2,x -> [[3]] * x +  [[0]])
    ,(a1,x -> [[3]] * x +  [[7]])      ,(a1,x -> [[1]] * x +  [[0]])
    ,(a3,x -> [[1]] * x +  [[0]])]     ,(a3,x -> [[1]] * x +  [[1]])]
```

▶ **Example 3.** We can mimick Jambox' behaviour by restricting the number of interpretations for the labelled system to 1. With a model of size 2^3, we get a termination proof (details omitted)

```
matchbox  ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l3,1 -d1 -b3 --nat
   CNF finished (#variables: 23876, #clauses: 91447)
   Solver finished in 2.282 seconds (result: True)
```

Jambox' formula (for the same parameters) has 33492 variables and 232683 clauses. That means that our CO4 compiler produced a SAT encoding that is comparable to a manual encoding.

6 Extensions

Our implementation additionally allows for each removal step for the labelled system to use arctic or natural matrix interpretation, or lexicographic path order with argument filtering, and also to reverse rules or not—where all these options are encoded in the program, and thus chosen by the solver.

▶ **Example 4.** This gives one of the shorter termination proofs for Zantema's system:

```
matchbox  ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l2,1 -d1 -b1 --lpo
    CNF finished (#variables: 4752, #clauses: 17075)
    Solver finished in 1.811 seconds (result: True)
model:    2b0 3b1 3b2 0b3
          1a0 2a1 1a2 1a3
labelled system: a1 a3 b2 b0 -> b0 b3 b1 a2 a1 a0
                 a1 a0 b3 b2 -> b0 b3 b1 a2 a1 a2
                 a1 a0 b3 b1 -> b0 b3 b2 a1 a2 a1
                 a1 a2 b0 b3 -> b0 b3 b1 a2 a1 a3
LPO:  delete symbols: a1 a3
      precedence: a0 = b2 > a2 = b3 > b0 > b1
```

Here we compare (with respect to the path order) after applying the morphism (argument filter) that deletes some symbols.

References

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

2 Jörg Endrullis. Jambox, 2009. Available at `http://joerg.endrullis.de`.

3 Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

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

5 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.

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