# Propositional Encoding of Constraints over Tree-Shaped Data

Johannes Waldmann and Alexander Bau

HTWK Leipzig, Germany
{waldmann,abau}@imn.htwk-leipzig.de

The paper presents a high-level declarative language CO4 for describing constraint systems. The language includes user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order and polymorphic types. This language comes with a compiler that transforms a high-level constraint system into a satisfiability problem in propositional logic. This is motivated by the following.

Recent years have seen a tremendous development of constraint solvers for propositional satisfiability (SAT solvers). Based on the Davis-Putname-Logemann-Loveland algorithm and extended with conflict-driven clause learning, SAT solvers like Minisat [2] are able to find satisfying assignments for conjunctive normal forms with $10^6$ and more clauses in a lot of cases. SAT solvers are used in industrial-grade verification of hardware and software.

With the availability of powerful SAT solvers, *propositional encoding* is a promising method to solve constraint systems that originate in different domains. In particular, this approach had been used for automatically analysing (non-)termination of rewriting [4, 8, 1] successfully, as can be seen from the results of International Termination Competitions (most of the participants use propositional encodings).

So far, these encodings are written manually: the programmer has to construct explicitly a formula in propositional logic that encodes the desired properties. This has the advantage that the formula can be optimized in clever ways, but also the drawback that correctness of the formula is not evident, so the process is error-prone.

This is especially so if the data domain for the constraint system is remote from the "sequence of bits" domain that naturally fits propositional logic. In typical applications, data is tree-structured (e.g., terms, and lists of terms) and one wants to write constraints on such data in a direct way.

Our language is similar to Haskell [3] in the following sense: CO4 syntactically is a subset of Haskell (including data declarations, case expressions, higher order functions, polymorphism, but no type classes), and semantically CO4 is evaluated strictly.

The advantages of re-using a high level declarative language for expressing constraint systems are: the programmer can rely on established syntax and semantics, does not have to learn a new language, can re-use his experience and intuition, and can re-use actual code.

For instance, the (Haskell) function that describes the application of a rewrite rule at some position in some term can be directly used in a constraint system that describes a rewrite sequence with a certain property. We treat this application in detail in Section 2, but need some preparation first.

A constraint programming language needs some way of parametrizing the constraint system to data that is not available when writing the program. For instance, a constraint program for finding looping derivations for a rewrite system $R$, will not contain a fixed system $R$, but will get $R$ as run-time input.

To accomodate for such applications, CO4 programs are handled and executed in two stages: The input program defines a function of type $f : K \times U \to \{\textsc{False}, \textsc{True}\}$ where $K$ is some parameter domain (e.g., rewrite systems) and $U$ is the domain of the unknown object (e.g., derivations). In the first processing stage (at *compile-time*), the program for $f$ is translated into a program $g : K \to (F, \Sigma \to U)$ with $F$ being the set of formulas of propositional logic, and $\Sigma$ being the set of assignments from variables of $F$ to truth values.

In the second stage (at *run-time*), a parameter value $p \in K$ is given, and $g\ p$ is evaluated to produce a pair $(v, d) \in (F, \Sigma \to U)$. An external SAT solver then tries to determine a satisfying assignment $\sigma \in \Sigma$ of $v$. On success, $d(\sigma)$ is evaluated to a *solution* value $s \in U$. Proper compilation ensures that $f\ p\ s = \textsc{True}$.

A specification of the compilation is given in section 1. In section 2 we discuss an application of CO4 in the area of automated program analysis. Note that there will be an extended version of this paper (submission ID: 720924) available on `http://arxiv.org` that elaborates our implementation of the given specification and proofs its correctness.

## 1 Semantics of Propositional Encodings

In this section we give the specification for the compilation of CO4 expressions, in the form of an invariant. When applied to the full input program, the specification implies that the compiler works as expected: a solution for the constraint system can be found via the external SAT solver. We defer discussion of our implementation of this specification to the extended paper.

**Evaluations on concrete data.** By $\mathbb{P}$ we denote the set of expressions in the input language. For instance, `f (Just x) Nothing` is an expression of $\mathbb{P}$, containing a variable `x`.

Evaluation of expressions is defined in the standard way: The domain of *concrete values* $\mathbb{C}$ is the set of data terms. For instance, `Just False` $\in \mathbb{C}$. A *concrete environment* is a mapping from program variables to $\mathbb{C}$. A *concrete evaluation function* $\mathsf{concrete\text{-}value} : E_{\mathbb{C}} \times \mathbb{P} \to \mathbb{C}$ computes the value of a concrete expression $p \in \mathbb{P}$ in a concrete environment $e_{\mathbb{C}}$. Evaluation of function and constructor arguments is strict. This is where we deviate from Haskell's lazy evaluation.

**Evaluations on abstract data.** The CO4 compiler transforms an input program that operates on concrete values, to an *abstract program* that operates on *abstract values*. An abstract value contains propositional logic formulas that may contain free propositional variables. An abstract value represents a set of concrete values. Each assignment of the propositional values produces a concrete value.

We formalize this in the following way: the domain of abstract values is called $\mathbb{A}$. The set of assignments (mappings from propositional variables to truth values $\mathbb{B} = \{0, 1\}$) is called $\Sigma$, and there is a function $\mathsf{decode} : \mathbb{A} \times \Sigma \to \mathbb{C}$.

We now specify abstract evaluation. We use *abstract environments* $E_{\mathbb{A}}$ that map program variables to abstract values, and an *abstract evaluation function* $\mathsf{abstract\text{-}value} : E_{\mathbb{A}} \times \mathbb{P} \to \mathbb{A}$.

**Allocators.** As explained in the introduction, the constraint program receives known and unknown arguments. The compiled program operates on abstract values.

The abstract value that represents a (finite) set of concrete values of an unknown argument is obtained from an *allocator*. For a property $q : \mathbb{C} \to \mathbb{B}$ of concrete values, a $q$-allocator constructs an object $a \in \mathbb{A}$ that represents all concrete objects that satisfy $q$:

$$\forall c \in \mathbb{C} : q(c) \iff \exists \sigma \in \Sigma : c = \mathsf{decode}(a, \sigma).$$

We use allocators for properties $q$ that specify $c$ uses constructors that belong to a specific type. In the extended paper we also specify a size bound for $c$. An example is an allocator for lists of booleans of length $\leq 4$.

As a special case, an allocator for a singleton set is used for encoding a known concrete value. This *constant allocator* is given by a function $\mathsf{encode} : \mathbb{C} \to \mathbb{A}$ with the property that $\forall c \in \mathbb{C}, \sigma \in \Sigma : \mathsf{decode}(\mathsf{encode}(c), \sigma) = c$.

**Correctness of constraint compilation.** The semantical relation between an expression $p$ (a concrete program) and its compiled version $\mathsf{compile}(p)$ (an abstract program) is given by the following relation between concrete and abstract evaluation:

We say that $p \in \mathbb{P}$ is compiled *correctly* if

$$\begin{aligned}\forall e \in E_{\mathbb{A}}\ \forall \sigma \in \Sigma : &\mathsf{decode}(\mathsf{abstract\text{-}value}(e, \mathsf{compile}(p)), \sigma) \\ &= \mathsf{concrete\text{-}value}(\mathsf{decode}(e, \sigma), p)\end{aligned}$$

Here we used $\mathsf{decode}(e, \sigma)$ as notation for lifting the decoding function to environments, defined element-wise by

$$\forall e \in E_{\mathbb{A}}\ \forall v \in \mathrm{dom}(e)\ \forall \sigma \in \Sigma : \mathsf{decode}(e, \sigma)(v) = \mathsf{decode}(e(v), \sigma).$$

**Application of the Correctness Property.** We are now in a position to show how the stages of CO4 compilation and execution fit together.

The top-level parametric constraint is given by a function declaration `main k u = b` where `b` (the *body*, a concrete program) is of type `Bool`. It will be processed in the following stages:

1. *compilation* produces an abstract program $\mathsf{compile}(b)$,
2. *abstract computation* takes a concrete parameter value $p \in \mathbb{C}$ and a $q$-allocator $a \in \mathbb{A}$, and computes the formula

$$F = \mathsf{abstract\text{-}value}(\{k \mapsto \mathsf{encode}(p), u \mapsto a\}, \mathsf{compile}(b))$$

3. *solving* calls the backend SAT solver to determine $\sigma \in \Sigma$ with $\mathsf{decode}(F, \sigma) = \textsc{True}$. If this was successful,

4. *decoding* produces a concrete value $s = \mathsf{decode}(a, \sigma)$,
5. and optionally, *testing* checks that $\mathsf{concrete\text{-}value}(\{k \mapsto p, u \mapsto s\}, b) = \textsc{True}$.

The last step is just for reassurance against implementation errors, since the invariant implies that the test returns True. This highlights another advantage of re-using Haskell for constraint programming: one can easily check the correctness of a solution candidate.

## 2 Case study: Loops in Term Rewriting

As an application, we use CO4 for compiling constraint systems that describe looping derivations. This is motivated by automated analysis of programs. A loop is an infinite computation, which may be unwanted behaviour, indicating an error in the program's design. In general, it is undecidable whether a rewriting system admits a loop. By enumerating finite derivations, one can hope to find loops.

Our approach is to write the predicate "the derivation $d$ conforms to a rewrite system $R$ and $d$ is looping" as a Haskell function, and solve the resulting constraint system, after putting bounds on the sizes of the terms that are involved.

Previous work uses several heuristics for enumerations resp. hand-written propositional encodings for finding loops in string rewriting systems [8].

We extend to (1) systematic compilation and (2) term rewriting.

In the following, we show the data declarations we use, and give code examples.

- we fix a signature, and a set of variables, and define the set of terms

```
data Term = V Name | F Term Term Term | A | B | C
data Name = X | Y
```

- a rule is pair of terms, a rewrite system is list of rules

```
data Rule = Rule Term Term
data List a = Nil | Cons a (List a)
type TRS = List Rule
```

- a rewrite step is a tuple $(t_0, (l, r), P, \sigma, t_1)$ where $t_0, t_1$ are terms, $(l, r)$ is a rule, $p$ is a position, $\sigma$ is a substitition with $l\sigma = t_0[p]$ and $t_0[p := r\sigma] = t_1$

```
data Pair a b = Pair a b
type Substitution = List (Pair Name Term)
data Step = Step Term Rule (List Pos) Substitution Term
```

- a derivation w.r.t. a TRS `trs` is a list of steps

```
type Derivation = List Step
```

  where

  – the result `term` of one step is the input term of the next step

```
derive_ok :: TRS -> Term -> Derivation -> Maybe Term
derive_ok trs term deriv = case deriv of
  Nil           -> Just term
  Cons s deriv' -> case s of
```

```
Step t0 rule pos sub t1 -> case equalTerm term t0 of
  False -> Nothing
  True  -> case step_ok trs s of
             False -> Nothing
             True  -> derive_ok trs t1 deriv'
```

  - each step's rule is from the `trs`

- a looping derivation's output of the last step has a subterm that is a substitution instance of the input of the first step

Overall, the complete CO4 code (available at `https://github.com/apunktbau/co4/blob/master/CO4/Test/TRS_Loop_Toyama.standalone.hs`) consists of roughly 300 lines of code including the definition of all involved data types and auxiliary functions. The code snippets above shows that the constraint system literally follows the textbook definitions.

Our test case is the following term rewriting system, where $X, Y$ are variables,

$$\{f(a,b,X) \to f(X,X,X), f(X,Y,c) \to X, f(X,Y,c) \to Y\},$$

(corresponding to the classical example from [6]). We use allocators that restrict to derivations of length 3, and terms of depth 2. Abstract evaluation of the compiled program results in a propositional formula with 774663 variables and 2301608 clauses. On a standard Intel Core 2 Duo CPU with 2.20 GHz, Minisat SAT solver finds the following loop in around 10 seconds:

$$\begin{aligned} &f(a,b,f(a,b,c)) \Rightarrow f(f(a,b,c), f(a,b,c), f(a,b,c)) \\ \Rightarrow &f(a,f(a,b,c),f(a,b,c)) \Rightarrow f(a,b,f(a,b,c)) \end{aligned}$$

## References

[1] 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.

[2] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

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

[4] Masahito Kurihara and Hisashi Kondo. Efficient bdd encodings for partial order constraints with application to expert systems in software verification. In Orchard et al. [5], pages 827–837.

[5] Robert Orchard, Chunsheng Yang, and Moonis Ali, editors. *Innovations in Applied Artificial Intelligence, 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2004, Ottawa, Canada, May 17-20, 2004. Proceedings*, volume 3029 of *Lecture Notes in Computer Science.* Springer, 2004.

[6] Yoshihito Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25(3):141–143, 1987.

[7] Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors. *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science.* Springer, 2010.

[8] Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In van Leeuwen et al. [7], pages 755–766.