SAT Compilation for Constraints over Finite Structured Domains

Alexander Bau*

HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany
abau@imn.htwk-leipzig.de

Abstract. Due to the availability of powerful SAT solvers, propositional encoding is a successful technique of solving constraint systems over finite domains. As these domains are often flat and non-structured, the CO^4 compiler aims to extend this concept by enriching the underlying domain with user-defined algebraic data types. Syntactically, CO^4 is a subset of Haskell and allows to specify constraint systems over such enriched domains using pattern-matching, higher-order functions and polymorphism. This paper illustrates examples and use-cases for CO^4 and provides an conceptual overview over the transformation into propositional logic.

1 Introduction

SAT solvers like Minisat [6] are successfully applied to solve constraints over finite domains. A finite domain U can be expressed by an enumeration type with a distinct constructor for each element of U, e.g., `data` U `= False | True` for propositional logic. This paper illustrates a technique of specifying and solving constraints over more complex domains U, that are written as algebraic data types (ADT):

$$\begin{aligned}
\texttt{data Bool} &= \texttt{False} \mid \texttt{True} \\
\texttt{data Color} &= \texttt{Red} \mid \texttt{Green} \mid \texttt{Blue} \\
\texttt{data Monochrome} &= \texttt{Black} \mid \texttt{White} \\
\texttt{data } U &= \texttt{Colored Color} \mid \texttt{Background Monochrome}
\end{aligned}$$

Such an extension is reasonable because many real-world constraints consists of structured and hierarchical types. Thus, a constraint programming language should reflect these properties by providing an appropriate type system.

Writing constraints over algebraic data types heavily involves pattern matching, i.e., inspecting the constructor a given expression was built with.

$$\texttt{case } x \texttt{ of } \{ \texttt{ Blue -> True; otherwise -> False } \}$$

Pattern matching enables the deconstruction of expressions and often is the main control-flow feature of declarative programming languages that support ADTs.

* This author is supported by ESF grant 100088525

Constraints over finite algebraic data types can be solved using a SAT solver by providing a transformation into propositional logic (*propositional encoding*). Such a transformation tackles two problems:

Data Transformation Firstly, it must represent values of the original domain by sequences of Boolean variables. While this is often straightforward for flat enumeration types, it becomes more complicated for ADTs in general.

Program Transformation Secondly, expressions over the original domains must be mapped to logical connectives that represent the control-flow in the constraint system. The transformation of pattern matches is especially crucial for the propositional encoding of the constraint system on the whole, because pattern matching is the only control-flow feature in CO^4 that enables conditional branching based on the value of a particular expression (the discriminant). As the discriminant may depend on the undetermined solution of the constraint, pattern matches often can't be evaluated directly.

So far, this transformation has been done manually: the programmer has to construct explicitly a formula in propositional logic that encodes the constraint in terms of logical connectives and Boolean variables. In particular, this approach has been successfully used for automatically analyzing (non-)termination of rewriting systems [9,12,5], as can be seen from the results of International Termination Competitions, where most of the participants use propositional encodings. Such a construction is similar to programming in assembly language: the advantage is that it allows for clever optimizations, but the drawbacks are that the process is inflexible and 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 not flat but hierarchical, and one wants to write constraints on such data in a direct way.

This paper illustrates the usage of the CO^4 language and its compiler[1]. CO^4 is a subset of Haskell including user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order and polymorphic types. The CO^4 compiler transforms a high-level constraint system into a satisfiability problem in propositional logic.

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

Outline Section 2 illustrates the CO^4 language to specify constraint systems. Section 3 gives an overview of the transformation process into propositional logic. We omit technical details that already have been published in [2] and [3]. Section 4 presents an encoding of the n-queens problem and reviews the use of recursive algebraic data types in CO^4. Section 5 and Section 6 illustrate real world use-cases for CO^4 and outline the propositional encoding of built-in naturals and partially defined functions.

[1] available at `https://github.com/apunktbau/co4`

2 Constraint Systems in CO^4

Syntactically CO^4 is a subset of Haskell[10]. Thus, domains are specified using algebraic data types (ADT):

$$\texttt{data}\ T\ v_1\ v_2\ \ldots\ =\ C_1\ a_{11}\ a_{12}\ \ldots\ |\ C_2\ a_{21}\ a_{22}\ \ldots\ |\ \ldots$$

An ADT T may be parametrized by n type variables $v_1, \ldots, v_n$. If $n > 0$, then T is a type operator, e.g., lists are usually defined as unary type operator ($n = 1$) whereas pairs are defined as binary ($n = 2$) type operator:

```
data List a   = Nil | Cons a (List a)
data Pair a b = Pair a b
```

The constructors C_i of an ADT T enumerate all values of T. A constructor C_i may be parametrized by arguments a_{ij}. A constructor argument either refers to a type or to one of T's type variables. Each constructor C_i denotes a function $C_i : a_{i1} \to a_{i2} \to \cdots \to T v_1 v_2 \ldots$ from its argument types to the ADT.

Example 1. `data Maybe a = Nothing | Just a` defines a type operator with one type variable and two constructors. Because constructor `Nothing` doesn't mention variable `a`, it denotes a polymorphic constant of type $\forall \texttt{a} : \texttt{Maybe a}$.

As CO^4 features a strict and static type system, each expression inhabits at least one type. An expression is either

- a variable, e.g., `x`, or
- a constructor call, e.g., `Just`, or
- an application, e.g., `Just x`, or
- an abstraction, e.g., `\x -> Just x`, or
- a local binding, e.g., `let f = ... in ...`,

or a pattern match. Pattern matching is the only expression that allows to diverge the control-flow based on the value of a particular expression (discriminant). The discriminant is compared against a sequence of patterns, where each pattern is associated with an expression (branch). The value of the pattern match equals the value of the first branch whose pattern matches the discriminant.

Example 2. `case x of { Just y -> f y; Nothing -> g }` matches the discriminant `x` against the patterns `Just y` and `Nothing`. If `x` matches the first (resp. second) pattern, expression `f y` (resp. `g`) is evaluated.

Constraint systems consist of a set of top-level declarations, where a declaration either defines an ADT or binds an expression to an identifier. Listing 1.1 in the next section shows an example of a constraint system written in CO^4.

3 Propositional Encoding of Constraint Systems

A CO^4 program always contains a parametrized top-level constraint `constraint` :: $P \times U \rightarrow$ `Bool` over a finite domain U where P is a (possibly singleton) parameter domain. Listing 1.1 gives an unrealistic simple example.

```
data Bool       = False | True
data Color      = Red   | Green | Blue
data Monochrome = Black | White
data Pixel      = Colored    Color
                | Background Monochrome

constraint :: Bool -> Pixel -> Bool
constraint p u = case p of
  False -> case u of Background m -> True
                     otherwise    -> False
  True  -> isBlue u

isBlue :: Pixel -> Bool
isBlue u = case u of
  Colored color -> case color of Blue      -> True
                                 otherwise -> False
  Background m  -> False
```

Listing 1.1. A trivial constraint over pixels

A solution for a constraint and a given parameter $p \in P$ is an element $u \in U$ of the problem domain U, so that `constraint` $p\ u =$ `True`. Given the parameter `True`, `Colored Blue` is the only solution in Listing 1.1. In the following, we call the input constraint a *concrete program*. A concrete program operates on *concrete values*, like `False`, `White` or `Colored Red`.

The CO^4 compiler uses an external SAT solver to find a solution for the top-level constraint. To do so, the following steps are performed:

1. The concrete program is transformed into an *abstract program*. An abstract program doesn't operate on the domains of the original program , but on *abstract values*.
2. Evaluating the abstract program for a given parameter $p \in P$ gives a formula $f \in \mathbb{F}$ in propositional logic.
3. An external SAT solver is called to find a satisfying assignment σ for f.
4. If there is a satisfying assignment, the solution $u \in U$ is constructed from σ. Optionally, testing whether `constraint` $p\ u =$ `True` ensures that there are no implementation errors. This check must always succeed if there is a solution.

Note that the transformation into an abstract program is done independently from an actual parameter. Thus, the same abstract program may be called for different parameters without the necessity to recompile the original program.

We do not prescribe a concrete representation for propositional formulas in $\mathbb{F}$. For efficiency reasons, our implementation[2] allows some form of sharing. Names are assigned to subformulas by doing the Tseitin transformation [11] on-the-fly, creating a fresh propositional literal for each subformula.

Propositional Encoding of Concrete Values The abstract program operates on abstract values. An abstract value represents a set of concrete values by encoding constructor indices using a sequence of propositional formulas.

Definition 1. *Assume $\mathbb{F}$ being the set of propositional formulas. Then, the set of abstract values $\mathbb{A}$ is the smallest set with $\mathbb{A} = \mathbb{F}^* \times \mathbb{A}^*$ where $\mathbb{F}^*$ denotes the set of sequences with elements from $\mathbb{F}$. An abstract value $a \in \mathbb{A}$ is a tuple $(\overrightarrow{f}, \overrightarrow{a})$ of* flags $\overrightarrow{f}$ *and* arguments $\overrightarrow{a}$.

The flags encode a constructor index using binary code.

Example 3. Consider the type `data Color = Red | Green | Blue` from Listing 1.1. For an abstract value $a \in \mathbb{A}$ to represent all the elements of `Color` it must consist of at least two flags, where each of them is a propositional formula. Depending on the satisfying assignment given by the SAT solver, a can be decoded to any value of type `Color`. As no constructor of `Color` has any arguments, the abstract value contains no arguments as well.

The arguments of an abstract value $a \in \mathbb{A}$ encode the constructor arguments of the concrete values that a is representing. To reduce the size of the generated propositional formula, the arguments of all constructors are overlapped in a.

Example 4. Consider the type

```
data Pixel = Colored Color | Background Monochrome
```

from Listing 1.1 and an abstract value $a_1 \in \mathbb{A}$ that represents all concrete values of type `Pixel`. As `Pixel` has two constructors, one flag f_1 is enough to encode its constructor index. Each constructor of `Pixel` has at most one argument, thus, the abstract value a_1 has one argument $a_2 \in \mathbb{A}$ as well. a_2 represents all concrete values of type `Color` and `Monochrome`. To do so, a_2 needs at least two flags f_{21} and f_{22}, because `Color` has three constructors:

$$a_1 = (f_1, a_2) \qquad a_2 = ((f_{21}, f_{22}), ())$$

Propositional Encoding of Concrete Programs As mentioned in Section 1, the propositional encoding of pattern matches is crucial for the encoding on the whole, because they are the only control-flow feature in CO^4.

In general, a pattern match on a discriminant v in the concrete program cannot be evaluated in the abstract program, because v might be an element of the problem domain. For example, the function `isBlue` in Listing 1.1 is a

[2] `https://github.com/apunktbau/satchmo-core`

predicate on the problem domain `Pixel` and its inner pattern match can't be evaluated for that reason. That's because values of the problem domain are yet to be determined by the SAT solver and are undefined during the evaluation of the abstract program. A way to resolve this situation is to evaluate each branch of a pattern match and to *merge* all the resulting abstract values.

Example 5. `isBlue` in Listing 1.1 contains the pattern match

$$\texttt{case}\ u\ \texttt{of}\begin{cases}\texttt{Colored \_} & \rightarrow b_1\\ \texttt{Background \_} & \rightarrow b_2\end{cases}$$

where u (of type `Pixel`) is the discriminant and b_1 and b_2 are concrete expressions of type `Bool`. The abstract program for this pattern match is

$$\begin{aligned}\texttt{let}\ &(f_u, \_) = u'\\ &(f_1, ()) = b'_1\\ &(f_2, ()) = b'_2\\ &f_r \qquad = \mathsf{merge}_{f_u}(f_1, f_2)\\ \texttt{in}&\\ &(f_r, ())\end{aligned}$$

where

- u' (resp. b'_1, b'_2) denotes the abstract program of discriminant u (resp. branch b_1, b_2)
- f_u (resp. f_1, f_2) denotes the single flag in the abstract value that results of evaluating u' (resp. b'_1, b'_2)
- f_r denotes the single flag in the resulting abstract value. Note that the result of the pattern match is of the same type as the branches are.

merge encodes a discrimination on the constructor indices of u on a binary level.

$$\mathsf{merge}_{f_u}(f_1, f_2) = (\neg f_u \implies f_1) \wedge (f_u \implies f_2)$$

Informally, f_r equals flag f_1 if the discriminant's flag f_u does not hold, and otherwise f_r equals flag f_2.

Using this transformation scheme and a parameter from the parameter domain, the evaluation of an abstract program results in an abstract value that represents a concrete Boolean value $a \in \mathbb{A}$, because `constraint`'s resulting type is `Bool`. a contains a single flag f which is the result of all merge operations that occurred while evaluating the abstract program. Thus, f represents the propositional formula that has to be solved by a SAT solver. If there is a satisfying assignment σ for f, a solution of the problem domain can be constructed from σ.

We refer to [2] for more technical details on the transformation process.

4 Example: The N-Queens Problem

Listing 1.2 illustrates an excerpt[3] of a specification for the n-queens problem in CO^4. The board is represented by a list of naturals, where each natural denotes the ordinate of a queen. The constraint holds if there are no two queens on each row, column and diagonal.

```
data Bool   = False | True            deriving Show
data Nat    = Z     | S Nat           deriving Show
data List a = Nil   | Cons a (List a) deriving Show
type Board  = List Nat

constraint :: Board -> Bool
constraint board = let n = length board
                   in
                     and2 (all (\q -> less q n) board)
                          (allSafe board)
allSafe :: Board -> Bool
allSafe board = case board of Nil       -> True
                              Cons q qs -> and2 (safe q qs (S Z))
                                                (allSafe qs)
safe :: Nat -> Board -> Nat -> Bool
safe q board delta = case board of
  Nil       -> True
  Cons x xs -> and2 (noAttack q x delta)
                    (safe q xs (S delta))

noAttack :: Nat -> Nat -> Nat -> Bool
noAttack x y delta = and2 (noStraightAttack x y)
                          (noDiagonalAttack x y delta)
```

Listing 1.2. The n-queens problem in CO^4 (excerpt)

In contrast to the introductory example in Section 1, the constraint in Listing 1.2 is not parametrized and makes use of recursive algebraic data types, e.g., `Nat` and `List`. A type that is defined as a recursive ADT is inhabited by infinitely many values, i.e., it's not finite domain. Thus, those types can not be represented using a finite propositional encoding.

To use recursive ADTs anyway, CO^4 uses *allocators* to restrict the set of concrete values that is represented by an abstract value.

Definition 2. *Let $\mathbb{C}$ be the set of concrete values. Then, an allocator $q_a : \mathbb{C} \to \{0, 1\}$ is a predicate on concrete values. For a given value $c \in \mathbb{C}$ $q_a(c)$ holds, if the abstract value $a \in \mathbb{A}$ represents c.*

When evaluating an abstract program in CO^4, the user must provide an allocator for the abstract value that represents the undetermined solution of

[3] full version available at `https://github.com/apunktbau/co4/blob/master/test/CO4/Example/QueensSelfContained.hs`

`constraint`, e.g., the `board` parameter in Listing 1.2. The propositional formula generated by evaluating the abstract program not only specifies a solution for `constraint`, CO^4 also enforces that a potential solution satisfies the allocator provided by the user.

Example 6. For the n-queens problem, allocators are utilized to restrict the recursion depth of the `board` parameter in `constraint`. Assume $a \in \mathbb{A}$ being the abstract value that represents the concrete `board` parameter. A possible allocator q_a may be informally described by

$$q_a(b) = \begin{cases} 1 & \text{if } b \text{ is an } 8 \times 8 \text{ board and each queen's ordinate} \\ & \text{is from the interval } [0, 7] \\ 0 & \text{otherwise} \end{cases}$$

Allocators that restrict recursion depths effect the evaluation of the abstract program and therefore the size of the resulting propositional formula. Table 1 lists some experimental results for different board sizes. All experiments were run on a 3.2 GHz CPU with 8 GB RAM.

Table 1. Formula sizes for different instances of the n-queen problem

n	#variables	#clauses	#literals	solver runtime
4	363	958	2441	0.1s
8	3621	10146	26353	0.1s
16	41033	118690	311649	0.16s
32	523921	1541826	4075713	3s

For all instances of the n-queens problem CO^4 generates a propositional encoding that is solved in less time compared to an equivalent encoding in Curry[8]. [2] gives a more detailed comparison of CO^4 and Curry.

5 Use-case: RNA Design

This section illustrates the application of CO^4 for RNA design[4] in bioinformatics. A strand of RNA (ribonucleic acid) is a molecule that is described as chain of the organic bases adenine, cytosine, guanine, and uracil, typically abbreviated as A, C, G and U. Thus, a string over $\{A, C, G, U\}$ is denoted as the RNA's *primary structure*. Many aspects of RNA are studied by inspecting its *secondary structure*, i.e., strings over the canonical base pairs $\{AU, CG, GC, GU, UA, UG\}$. Each RNA's primary structure and one of its corresponding secondary structure is associated with a certain amount of *free energy* based on a given energy model.

RNA design is a fundamental problem in bioinformatics that asks for a primary structure that folds optimally into a given RNA's secondary structure, so that the amount of free energy is minimized. Listing 1.3 shows an excerpt[4] of a specification for RNA design constraints. For technical reasons the constraint is formalized to maximize the bound energy instead of minimizing the free energy.

```
data Base    = A | C | G | U
type Primary = List Base

data Energy  = MinusInfinity | Finite Nat

constraint :: Secondary -> (Primary, Matrix Energy) -> Bool
constraint secondary (primary, energy) = ...

cost :: Base -> Base -> Energy
cost b1 b2 = case (b1,b2) of
  (C,G) -> Finite (nat 8 2)
  (G,U) -> Finite (nat 8 1)
  ...
  _     -> MinusInfinity

max :: Energy -> Energy -> Energy
max e f = case e of
  Finite x -> case f of
    Finite y      -> Finite (maxNat x y)
    MinusInfinity -> e
  MinusInfinity -> f

plus :: Energy -> Energy -> Energy
plus e f = case e of
  Finite x -> case f of
    Finite y      -> Finite (plusNat x y)
    MinusInfinity -> f
  MinusInfinity -> e
```

Listing 1.3. RNA design using CO^4 (excerpt)

The constraint is parametrized by the RNA's secondary structure, where the solution is a pair of a primary structure and a matrix of bound energies. This matrix contains the energy values for the unknown primary structure and is computed using the ADP framework[7].

The energy model in function `cost` associates a certain amount of bound energy to each each of the canonical base pairs. Other base pairs are associated with $-\infty$ to exclude them as pairs in the potential solution. The dominant operations while evaluating the abstract program are applications of `max` and `plus` on elements of energy matrices.

[4] full version available at `https://github.com/apunktbau/co4/blob/master/test/CO4/Example/WCB_MatrixStandalone.hs`

In contrast to the n-queens problem in Listing 1.2, this example makes use of CO^4's built-in naturals in order to reduce the size of the propositional encoding. These naturals are binary encoded and CO^4 provides built-in arithmetic and comparison functions.

Example 7. A call to `nat` $w\ n$ in an abstract program gives an abstract value $a = ((f_1, \dots, f_w), ()) \in \mathbb{A}$ representing n in binary code using w flags $f_1, \dots, f_w$, where $w \geq \lceil \log_2 n \rceil$. a does not contain any arguments. Using this representation it is straightforward to implement arithmetic for naturals using binary arithmetic. CO^4 provides common arithmetic functions on naturals, e.g., `plusNat` and `maxNat` in Listing 1.3.

Table 2 cites some experimental results[4] performed on different instances of the RNA design problem.

Table 2. Formula sizes for different instances of the RNA design problem

length of primary structure	#variables	#clauses	#literals	solver runtime
20	77951	368036	1166086	2s
30	235714	1164984	3712214	4s
40	526111	2666878	8525686	7m
50	989133	5096071	16324618	36s

6 Use-case: Termination Analysis of Term Rewriting Systems

The application of CO^4 to termination analysis of term rewriting systems is motivated by the automated analysis of programs. A non-terminating program may be an unwanted behavior that indicates an error in the program's design. Unfortunately, termination is an undecidable property of programs, but there are techniques that may prove termination in some cases.

A term rewriting system (TRS) is a computational model for terms, where a term is either a variable or a n-ary function symbol applied to n terms. Terms can be modeled using the following ADT:

```
data Term = Var Symbol | Node Symbol (List Term)
```

Term rewriting is based on the repeated application of rewriting rules, where each rule $l \to r$ replaces a (sub-)term l by a term r. A TRS is a set of rewriting rules $\{l_1 \to r_1, r_2 \to l_2, \dots\}$. One common technique to prove termination of a TRS is to find a *reduction order* $>$ on terms, so that $l > r$ holds for all rules $l \to r$ in the TRS[1].

Definition 3. *Assume $>_{prec}$ being a strict order on function symbols. $>_{prec}$ is denoted as* precedence. *Then, the lexicographic path order (LPO) $>_{lpo}$ is a reduction order on terms s and t induced by precedence $>_{prec}$, where $s >_{lpo} t$, if*

- ***LPO-1**: t is a variable and $s \neq t$, or*
- ***LPO-2**: $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$, and*
  - ***LPO-2a**: there exists $i \in [1, m]$ with $s_i \geq_{lpo} t$, or*
  - ***LPO-2b**: $f >_{prec} g$ and $s >_{lpo} t_j$ for all $j \in [1, n]$, or*
  - ***LPO-2c**: $f = g$, $s >_{lpo} t_j$ for all $j \in [1, n]$ and ther exists $i \in [1, m]$ so that $s_1 = t_1, \dots, s_{i-1} = t_{i-1}$ and $s_i >_{lpo} t_i$*

In listing 1.4 `Symbols` are represented by naturals and the precedence $>_{\text{prec}}$ is modeled using a mapping `prec` from function symbols to naturals. The semantics are: $f >_{\text{prec}} g$, if $\texttt{prec}(f) > \texttt{prec}(g)$.

```
type Map k v    = List (Pair (k,v))
type Symbol     = Nat
type Precedence = Map Symbol Nat
data Order      = Gr | Eq | NGe

lpo :: Precedence -> Term -> Term -> Order
lpo prec s t = case t of
  Var x -> case eqTerm s t of
    False -> case varOccurs x s of
                False -> NGe
                True  -> Gr     -- LPO-1
    True  -> Eq

  Node g ts -> case s of
    Var _     -> NGe
    Node f ss ->                -- LPO-2
      case all (\si -> eqOrder (lpo prec si t) NGe) ss of
        False -> Gr             -- LPO-2a
        True  -> case ord prec f g of
          Gr  -> case all (\ti -> eqOrder (lpo prec s ti) Gr) ts of
                   False -> NGe
                   True  -> Gr  -- LPO-2b
          Eq  -> case all (\ti -> eqOrder (lpo prec s ti) Gr) ts of
                   False -> NGe
                   True  -> lex (lpo prec) ss ts -- LPO-2c
          NGe -> NGe
```

Listing 1.4. Lexicographic path orders in CO^4 (excerpt)

`lpo` computes the lexicographic path order between two terms using a given precedence. It is almost an direct encoding of the textbook definition 3.

The top-level constraint[5] over the set of precedences is parametrized by a TRS and simply checks whether `lpo` is a reduction order for all rules in the TRS.

[5] full version available at `https://github.com/apunktbau/co4/blob/master/test/CO4/Example/LPOStandalone.hs`

```
type Rule       = Pair Term Term
type Trs        = List Rule

constraint :: Trs -> Precedence -> Bool
constraint rules prec =
  all (\(lhs,rhs) -> eqOrder (lpo prec lhs rhs) Gr) rules
```

Listing 1.5. Top-level constraint for lexicographic path orders

For the following term rewriting system with function symbols $\{a/2, s/1, n/0\}$

$$\begin{aligned} a(n, y) &\to s(y) \\ a(s(x), n) &\to a(x, s(n)) \\ a(s(x), s(y)) &\to a(x, a(s(x), y)) \end{aligned}$$

CO^4 finds a precedence $a >_{\text{prec}} s =_{\text{prec}} n$ using a propositional encoding with 167 variables, 517 clauses and 1365 literals almost immediately.

The Tyrolean Termination Tool 2 (TTT2)[6] provides a hand-crafted propositional encoding for lexicographic path orders. For solving the aforementioned rewriting system, TTT2 generates a formula with 7 variables and 9 clauses. This result emphasizes the main drawback of CO^4: most manually crafted propositional encodings that exploit low-level optimizations outperform the encodings derived by CO^4.

Propositional Encoding for Partial Functions Recall that the precedence is represented by a mapping from symbols to naturals. This mapping is realized as a list of pairs. Accessing such a mapping using `lookup` is done by traversing the whole list until the provided key is found:

```
lookup :: Symbol -> Precedence -> Nat
lookup symbol prec = case prec of
  []   -> undefined
  p:ps -> case p of
    (key,value) -> case eqSymbol symbol key of
      False -> lookup symbol ps
      True  -> value
```

Note that `lookup` is a partially defined function, because it is `undefined` if `prec` is empty. To support partially defined functions in the abstract program, the propositional encoding of abstract values illustrated in section 3 is extended by an *definedness flag*.

Definition 4. *The set of (possibly undefined) abstract values $\mathbb{A}$ is the smallest set with $\mathbb{A} = \mathbb{F}^* \times \mathbb{A}^* \times \mathbb{F}$. An abstract value $a \in \mathbb{A}$ is a tuple $(\overrightarrow{f}, \overrightarrow{a}, d)$ of flags $\overrightarrow{f}$, arguments $\overrightarrow{a}$ and an definedness flag d.*

[6] `http://cl-informatik.uibk.ac.at/software/ttt2/`

For abstract values that represent ordinary concrete values, the definedness flag is constant 1, i.e., the abstract value is defined. Only the `undefined` symbol results in an abstract value whose definedness flag is constant 0.

The definedness flags are merged as well as all the other flags during the evaluation of pattern matches in the abstract program. Evaluating the abstract top-level constraint then results in an abstract value $a \in \mathbb{A}$ with a single flag f and a definedness flag d: as discussed in section 3, f discriminates the two constructors of `constraint`'s resulting type `Bool`. d indicates whether a is defined or not. As we aim to exclude undefined values from the set of potential solutions, we search a satisfying assignment for $f \wedge d$ using the external SAT solver. If there are no undefined values in a constraint system, d is constant 1. Otherwise, d is a propositional formula that represents the result of merging the definedness flags of all abstract values, that have been evaluated.

Example 8. Consider a pattern match on a value u of type `Bool` where one branch is undefined and the other branch b is of type `Bool` as well:

$$\texttt{case}\ u\ \texttt{of} \begin{cases} \texttt{False} & \rightarrow \texttt{undefined} \\ \texttt{True} & \rightarrow b \end{cases}$$

Assume the following flags

- f_u denotes the single flag in the abstract value that represents the result of evaluating discriminant u.
- d_b denotes the definedness flag of the abstract value that represents the result of evaluating branch b. The definedness of `undefined` is constant 0.

The resulting definedness flag d_r is merged equally as the other flags (c.f. Example 5):

$$\begin{aligned} d_r = \mathsf{merge}_{f_u}(0, d_b) &= (\neg f_u \implies 0) \wedge (f_u \implies d_b) \\ &= f_u \wedge (f_u \implies d_b) \\ &= f_u \wedge d_b \end{aligned}$$

7 Conclusion

In this paper we presented examples of using CO^4 to write constraints on finite structured domains using a subset of Haskell. We illustrated two real world use cases where CO^4 enables a natural way of specifying properties of application specific data:

1. specifying a primary RNA structure that folds optimally into a given secondary structure
2. specifying a reduction order on terms that prove termination of a term rewriting system

We outlined the propositional encoding provided by the CO^4 compiler including the encoding for restricted recursive ADTs, built-in naturals and partially defined functions. For a more technical description of CO^4 we refer the reader to [2].

The work on CO^4 is ongoing. We strive to reduce the size of the generated propositional encoding. While ideally CO^4 should be competitive against other encodings, carefully crafted manual encodings outperform CO^4 in most cases because of potential low-level optimizations. Thus, our goal is to minimize the gap between manual propositional encodings and CO^4.

Further work includes the support for a greater subset of Haskell's syntax. Supporting more features in the input language, e.g. type-classes, allows an even more natural way of specifying constraint systems.

References

1. Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.
2. Alexander Bau and Johannes Waldmann. Propositional Encoding of Constraints over Tree-Shaped Data. In *22nd International Workshop on Functional and (Constraint) Logic Programming*, 2013.
3. Alexander Bau and Johannes Waldmann. Propositional encoding of constraints over tree-shaped data. *CoRR*, abs/1305.4957, 2013.
4. Alexander Bau, Johannes Waldmann, and Sebastian Will. RNA Design by Program Inversion via SAT Solving. In *Workshop on Constraint-Based-Methods for Bioinformatics*, 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. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.
7. Robert Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16(8):665–677, 2000.
8. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
9. Masahito Kurihara and Hisashi Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In Robert Orchard, Chunsheng Yang, and Moonis Ali, editors, *IEA/AIE*, volume 3029 of *Lecture Notes in Computer Science*, pages 827–837. Springer, 2004.
10. Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.
11. G.S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.
12. 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.