Describing and Measuring the Complexity of SAT encodings for Constraint Programs

Alexander Bau* and Johannes Waldmann

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

Abstract. The CO^4 language is a Haskell-like language for specifying constraint systems over structured finite domains. A CO^4 constraint system is solved by an automatic transformation into a satisfiability problem in propositional logic that is handed to an external SAT solver.
We investigate the problem of predicting the size of formulas produced by the CO^4 compiler. The goal is to help the programmer in understanding the resource consumption of CO^4 on his program. We present a basic cost model, with some experimental data, and discuss ongoing work towards static analysis. It turns out that analysis steps will use constraint systems as well.

1 Introduction

CO^4 is a constraint programming language that allows to write a constraint problem as declarative specification. The CO^4 compiler solves it by transforming the constraint to a propositional satisfiability problem, so that a SAT solver can be applied. Syntactically, 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.

In CO^4, a constraint system over elements of set U is specified by a parametrized predicate `constraint` : $P \times U \to$ `Bool`, where P denotes the parameter domain. Thus, `constraint` does not denote a single constraint, but a family of constraints. For a given `constraint` and parameter $p \in P$, $u \in U$ is a solution if `constraint` (p, u) = `True`.

For the CO^4 compiler to generate a propositional encoding, the input `constraint` is transformed into an *abstract program* `constraint'` that operates on *abstract values*. An abstract value represents an undetermined value of the input program by encoding the constructor's index using propositional formulas. Evaluating the abstract program generates the final formula that is passed to the external SAT solver.

It is desirable to predict the runtime of the SAT solver for a generated propositional encoding. Such a prediction is hard because as it depends on a lot of design and implementation decisions of the SAT solver. Therefore we take the

* This author is supported by ESF grant 100088525

size of the SAT encoding as a reasonable indicator for its hardness. To estimate the size of the encoding, we introduce a cost model for abstract values and abstract programs. This cost model captures two important facts: the size of intermediate abstract values and the costs to evaluate them. Especially the evaluation of case distinctions on abstract values is not obvious, and often they cannot be evaluated in a straightforward manner.

This paper has three parts. The first part illustrates the syntax and semantics of CO^4 (Section 2) and gives an overview on of the propositional encoding (Section 3). This is a summary of material that has already been published[1]. The second part presents current work on cost analysis: in Section 4 we present our cost model, and in Section 5 we analyze the cost of the merge operation, which is a basic operator in our translation scheme. Section 6 illustrates how the current CO^4 implementation measures concrete costs of SAT-compiled function calls. The third part outlines future work in static analysis of CO^4 programs. Section 7 describes moded types and their inference, which will allow a more efficient propositional encoding of case distinctions. Section 8 describes an approach to bound function costs by resource types.

2 Syntax and Semantics of CO^4

Syntactically, CO^4 is a subset of Haskell. Domains are specified by algebraic data types (ADT), where *constructors* enumerate the values of the type.

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

CO^4 supports recursive ADTs as well, but recursions must be restricted while generating a propositional encoding. We do not deal with recursions in the scope of this paper.

A constructor may be parametrized either by types or type variables.

```
1  data Pair   a b = Pair a b
2  data Either a b = Left a | Right b
```

Inspecting the constructor of a value `d` of some ADT is done by a case distinction on `d` (the *discriminant* of the case distinction):

```
1 case color of Blue      -> True
2               otherwise -> False
```

Case distinctions provides conditional branching of the control-flow. Other kinds of expressions in CO^4 are constructor calls, applications, abstractions and local bindings. CO^4 provides restricted support of higher-order, polymorphic functions. Besides type definitions, constraint systems in CO^4 contain global function bindings with `constraint` being the top-level function:

```
1  data Bool       = False | True
2  data Color      = Red   | Green | Blue
3  data Monochrome = Black | White
4  data Pixel      = Colored Color | Background Monochrome
5
6  constraint :: Bool -> Pixel -> Bool
7  constraint p u = case p of
8    False -> case u of Background m -> True
9                       otherwise    -> False
10   True  -> isBlue u
11
12 isBlue :: Pixel -> Bool
13 isBlue u = case u of
14   Colored color -> case color of Blue      -> True
15                                  otherwise -> False
16   Background m  -> False
```

Listing 1.1. A trivial constraint over pixels

Semantically, a constraint system in CO^4 over elements of set U is a binary predicate `constraint` : $P \times U \to$ `Bool` on U and some parameter domain P. In Listing 1.1, $P =$ `Bool` and $U =$ `Pixel`.

For a given parameter $p \in P$, $u \in U$ is a solution if `constraint` (p, u) `= True`. One advantage of specifying constraint systems in a functional language like CO^4 is that a solution can be tested against the constraint simply by evaluating `constraint` (p, u). Note that CO^4 expressions are evaluated strictly, while Haskell features a non-strict evaluation strategy.

3 Propositional Encoding of CO^4 constraints

In the following, we call the source constraint a *concrete program*. Concrete programs operate on *concrete values*, e.g., the concrete program in Listing 1.1 operates on concrete values like `False`, `White` or `Colored Red`.

To find a solution $u \in U$ for a constraint `constraint` : $P \times U \to$ `Bool` and a parameter $p \in P$, CO^4 performs the following steps:

1. The concrete program is transformed into an *abstract program*. An abstract program doesn't operate on concrete values, but on *abstract values*.
2. Evaluating the abstract program for an abstract value that represents parameter p gives a formula $f \in \mathbb{F}$ in propositional logic.
3. An external SAT solver is called to find a satisfying assignment $\sigma \in \Sigma$ 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.

In the following we briefly illustrate the first two steps of this process. Firstly, an abstract program is generated from a given concrete program. This transformation not only modifies the program structure, the domain is changed as well.

Data Transformation An abstract program is an untyped, first-order and imperative program on abstract values.

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 $\overrightarrow{a}$*.*

An abstract value $a \in \mathbb{A}$ represents a (maybe unknown) value of a concrete type T. The flags of an abstract value $a \in \mathbb{A}$ encode the indices of T's constructors in binary code using propositional formulas.

Example 1. For an abstract value $a_1 \in \mathbb{A}$ to represent a value of the ADT `data Color = Red | Green | Blue | Purple` it must contain two flags $f_1, f_2 \in \mathbb{F}$ because `Color` has four constructors. Thus, $a_1 = ((f_1, f_2), ())$. a_1 has no arguments because none of `Color`'s constructors has any arguments.

Consider an ADT `data Maybe a = Nothing | Just a` and an abstract value $a_2 \in \mathbb{A}$ that is supposed to represent a value of type `Maybe Color`. As `Maybe` consists of two constructors, one flag $f_3 \in \mathbb{F}$ is needed to discriminate both. Thus, (f_3, a_1) is a proper value for a_2. Note that a_2 has a single argument a_1 that encodes the constructor argument of type `Color` of `Maybe`'s `Just` constructor.

As the flags of an abstract value $a \in \mathbb{A}$ may contain propositional variables, a can be decoded to different values according to the Boolean values that are assigned to these variables. By $\mathsf{decode}_T : \mathbb{A} \times \Sigma \to T$ we denote a mapping from abstract values and propositional assignments Σ to concrete values.

If the flags of an abstract value $a \in \mathbb{A}$ don't contain propositional variables, then the flags of a index a particular constructor and a can only be decoded to a single concrete value. By $\mathsf{encode}_T : T \to \mathbb{A}$ we denote a mapping from concrete values to abstract values that represent a fixed value.

Example 2. Recall the ADTs defined in Example 1 and assume the flags of an abstract value reference a constructor's index using binary code where the first flag encodes the most significant bit. Then:

$$\mathsf{encode}_{\texttt{Color}}(\texttt{Blue}) = ((\textsc{True}, \textsc{False}), ())$$
$$\mathsf{encode}_{\texttt{Maybe Color}}(\texttt{Just Blue}) = (\textsc{True}, ((\textsc{True}, \textsc{False}), ()))$$

As we've omitted details about abstract values we don't provide definitions for encode and decode.

Program Transformation The program structure of abstract programs resembles the structure of their concrete counterparts. The most important difference concerns case distinctions: while concrete values may be examined by matching on their constructor, this is often not possible for abstract values. That's because an abstract value's flags may contain propositional variables. Therefore, it

is undetermined which constructor is indexed by the flags and there is no way to known which branch to evaluate. Thus, all branches must be evaluated and their result is merged according to the discriminant of the case distinction.

Example 3. The following case distinction matches on a Boolean value x in a concrete program:

$$r \texttt{ = case } x \texttt{ of \{ False -> } g \texttt{ ; True -> } h \texttt{ \}}$$

In the abstract counterpart of this expression, the abstract values g' and h' of both branches are evaluated and merged according to x

$$\begin{aligned} r' \texttt{ = let } & \texttt{\_1 = } g' \\ & \texttt{\_2 = } h' \\ & \texttt{in } \mathsf{merge}_{x'}(\texttt{\_1,\_2}) \end{aligned}$$

where r' (resp. x', g', h') denote the abstract counterpart of r (resp. x, g, h).

The function $\mathsf{merge}_x : \mathbb{A}^* \to \mathbb{A}$ encodes a case distinction on a value $x \in \mathbb{A}$ using the flags of x and the abstract values of all evaluated branches. We don't give a definition for merge, but illustrate its semantics by the following example.

Example 4. Recall the case distinction in Example 3 and assume r' (resp. x', g', h') denotes the abstract counterpart of r (resp. x, g, h). The following two clauses are emitted when evaluating $\mathsf{merge}_{x'}(g', h')$:

$$\begin{aligned} & (x' \equiv \mathsf{encode}_{\texttt{Bool}}(\texttt{False}) \implies r' \equiv g') \\ \wedge\ & (x' \equiv \mathsf{encode}_{\texttt{Bool}}(\texttt{True}) \implies r' \equiv h') \end{aligned}$$

Informally, both clauses encode the semantics of the original case distinction in terms of abstract values: r' equals g' if x' equals $\mathsf{encode}_{\texttt{Bool}}(\texttt{False})$, otherwise r' equals h'.

However, if none of the flags of an abstract value contain any propositional variables, then the constructor that is indexed by these flags can be determined and the associated branch can be evaluated. In this case it is not necessary to evaluate the other branches.

Evaluation of Abstract Programs The `constraint` : $P \times U \to$ `Bool` function in a concrete program has a counterpart `constraint'` : $\mathbb{A} \times \mathbb{A} \to \mathbb{A}$ in the abstract program of the same arity. Evaluating `constraint'` p' u' on

- $p' = \mathsf{encode}_P(p)$ for some parameter $p \in P$, and
- $u' \in \mathbb{A}$, which represents a undetermined value in U,

gives a value $a \in \mathbb{A}$ that represents a Boolean value, i.e., a contains a single flag $f \in \mathbb{F}$. Solving f using an external SAT solver gives a satisfying assignment $\sigma \in \Sigma$ for all variables in f if there is such an assignment. The final solution $u \in U$ can be constructed by $\mathsf{decode}_U(u', \sigma)$.

We refer to [1] for more technical details on the transformation process.

4 Cost Model

We illustrate an approach for formalizing the costs associated with a function in a CO^4 program. For readability we stick to unary functions and omit details about functions of higher arities.

We measure the cost of a function $f : A \to B$ in a concrete program by analyzing its counter part $f' : \mathbb{A} \to \mathbb{A}$ in the corresponding abstract program. The costs of f' depend on the size of its argument. Thus, we introduce a function $\mathsf{size} : \mathbb{A} \to \mathbb{N}$ to measure the size of an abstract value.

Example 5. There are at least two naive definitions for size: one that counts the number of nested abstract values

$$\mathsf{size}_1(\overrightarrow{f}, (a_1, \ldots, a_n)) = 1 + \sum_{i=1}^{n} \mathsf{size}_1(a_i)$$

and one that counts the number of flags in an abstract value

$$\mathsf{size}_2((f_1, \ldots, f_m), (a_1, \ldots, a_n)) = m + \sum_{i=1}^{n} \mathsf{size}_2(a_i)$$

Fixing a particular implementation for size, the cost of the abstract function f' is described by a pair of functions $s_f, c_f : \mathbb{N} \to \mathbb{N}$.

Definition 2. *$s_f(n)$ gives the maximal output size for all arguments of f' with size n or smaller:*

$$s_f(n) = \max\{\mathsf{size}(f(\mathsf{encode}_A(x))) \mid x \in A \wedge \mathsf{size}(\mathsf{encode}_A(x)) \leq n\}$$

Whereas s_f quantifies the size of a function's result, c_f measures the evaluation costs of f'.

Definition 3. *$c_f(n)$ gives the evaluation costs for all arguments of f' with size n or smaller*

$$c_f(n) = \max\{\mathsf{work}(f, \mathsf{encode}_A(x)) \mid x \in A \wedge \mathsf{size}(\mathsf{encode}_A(x)) \leq n\}$$

where $\mathsf{work}(f, x)$ equals the cost of evaluating $f'(\mathsf{encode}_A(x))$ in the abstract program.

We can instantiate this scheme in several ways: for example, $\mathsf{work}(f, x)$ could give the number of propositional variables or clauses that are allocated while computing the abstract value $f'(\mathsf{encode}_A(x))$. Other techniques may include additional characteristics about the propositional encoding, like the number of literals or the depth of the formula.

5 Cost of merge

Example 4 illustrated the semantics of the merge operation on abstract values. Now we quantify the cost of merge in terms of the cost model in Section 4.

Assume the following case distinction with n branches $b_1, \ldots, b_n$ in a function f:

$$\begin{aligned} f(x) \texttt{ = case } x \texttt{ of } & C_1 \texttt{ -> } b_1 \\ & \ldots \\ & C_n \texttt{ -> } b_n \end{aligned}$$

Listing 1.2. A case distinction over n branches

where $f' : \mathbb{A} \to \mathbb{A}$ denotes the abstract counterpart of f. In order to evaluate $f'(x')$ for some abstract argument $x' \in \mathbb{A}$ we need to evaluate all abstract branches $b'_i \in \mathbb{A}$ for $i \in [1, n]$ and merge the results by $\mathsf{merge}_{x'}(b'_1, \ldots, b'_n)$. We denote the result of this merge by $r' \in \mathbb{A}$.

A first cost measure determines the numbers of variables that are needed to represent the result of an application of merge (variable-cost).

Definition 4. $\mathsf{work}_{\mathbb{V}}(f, x)$ *denotes the variable-cost of function* f *in Listing 1.2 and equals the maximum number of flags in the branches, i.e., if* m_i *denotes the number of flags in branch* b'_i *for* $i \in [1, n]$*, then*

$$\mathsf{work}_{\mathbb{V}}(f, x) = \max\{m_i \mid 1 \leq i \leq n\}$$

As the result of $f'(x)$ must equal one of the branches b'_i (c.f. Example 4) it is reasonable for $\mathsf{work}_{\mathbb{V}}$ to assume that r' must consist of the maximum number of flags that are present in the abstract branches $b'_i \in \mathbb{A}$ for $1 \leq i \leq n$.

Furthermore, we define the clause-cost of an application of merge. Example 4 illustrated that the flags in a result of merge encode the case distinction in terms of abstract values. The clause-costs represent the number of clauses in a propositional formula that are needed to encode a case distinction.

Definition 5. $\mathsf{work}_{\mathbb{C}}(f, x)$ *denotes the clause-cost of function* f *in Listing 1.2 where* n *denotes the number of branches:*

$$\mathsf{work}_{\mathbb{C}}(f, x) = 2 * \mathsf{work}_{\mathbb{V}}(f, x) * n$$

$\mathsf{work}_{\mathbb{C}}$ is reasonable because two clauses are emitted for each of the $\mathsf{work}_{\mathbb{V}}(f, x)$ flags in r' and each of the n branches.

6 Profiling CO4

In the following we compare the profiling output of CO^4 for some examples and show the relation to the previously defined cost-model.

The first example illustrates the difference between the cost of evaluating a concrete program and an abstract program.

```
1  data Bool = False | True deriving Show
2  data T    = T1 | T2 | T3 deriving Show
3
4  g :: T -> Bool
5  g t = case t of T1 -> True
6                  T2 -> False
7                  T3 -> False
8
9  f1 :: Bool -> Bool
10 f1 b = case b of False -> g T1
11                  True  -> g T2
12
13 f2 :: Bool -> Bool
14 f2 b = g (case b of False -> T1
15                     True  -> T2)
```

Listing 1.3. Profiling two semantically equivalent functions

Listing 1.3 defines two functions `f1,f2` with the same concrete semantics. Assume `f1'` (resp. `f2',g'`) being the abstract counterpart of `f1` (resp. `f2,g`). Further assume that $b \in \mathbb{A}$ is an abstract value that represents an undetermined value of type `Bool`. Then, evaluating `f1'` b gives

```
("f1'", {numCalls = 1, numVariables = 1, numClauses = 4})
("g'", {numCalls = 2, numVariables = 0, numClauses = 0})
```

`g'` does not allocate any variables nor clauses as its argument is constant in both calls `g T1` and `g T2` in the concrete program. Thus, the case distinction in `g'` can be evaluated straightforwardly without applying merge.

`f1'` is called once and allocates one variable (resp. four clauses). That matches the $\mathsf{work}_{\mathbb{V}}$ (resp. $\mathsf{work}_{\mathbb{C}}$) cost function, because

- $\mathsf{work}_{\mathbb{V}}(\mathtt{f1}, \mathtt{b}) = \max\{1, 1\} = 1$ as each branch in `f1'` is represented by an abstract value with one flag (because `Bool` has two constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{f1}, \mathtt{b}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{f1}, \mathtt{b}) * 2 = 4$ as there are $n = 2$ branches in `f1'`

On the other hand, evaluating `f2'` b gives

```
("f2'", {numCalls = 1, numVariables = 2, numClauses = 8})
("g'", {numCalls = 1, numVariables = 1, numClauses = 6})
```

Again, the profiling information matches with the cost functions $\mathsf{work}_{\mathbb{V}}$ and $\mathsf{work}_{\mathbb{C}}$ defined in Section 5, because

- $\mathsf{work}_{\mathbb{V}}(\mathtt{f2}, \mathtt{b}) = \max\{2, 2\} = 2$ as each branch in `f2'` is represented by an abstract value with two flags (because `T` has three constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{f2}, \mathtt{b}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{f2}, \mathtt{b}) * 2 = 8$ as there are $n = 2$ branches in `f2'`
- $\mathsf{work}_{\mathbb{V}}(\mathtt{g}, \mathtt{t}) = \max\{1, 1, 1\} = 1$ as each branch in `g'` is represented by an abstract value with one flag (because `Bool` has two constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{g}, \mathtt{t}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{g}, \mathtt{t}) * 3 = 6$ as there are $n = 3$ branches in `g'`

Note that `f2'` allocates more variables than `f1'` because it merges branches of type `T`, which has more constructors than `Bool`. In the second case, `g'` is only called once, but this time with an unknown argument: its argument indirectly depends on the unknown b. Thus, `g'` allocates variables and emits clauses.

We give a more complex example: CO^4 has been applied to problems of termination analysis of term rewriting systems. One exemplary problem is the specification of a lexicographic path order (LPO) that proves the termination of a given term rewriting system[1].

```
Profiling (inner-under):
("constraint'", {numCalls = 1, numVariables = 160, numClauses = 514})
("allHOInst'", {numCalls = 1, numVariables = 160, numClauses = 514})
("mapHOInst'", {numCalls = 4, numVariables = 157, numClauses = 506})
("globalLam'", {numCalls = 3, numVariables = 157, numClauses = 506})
("globalLamSat'", {numCalls = 3, numVariables = 157, numClauses = 506})
("lpo'", {numCalls = 41, numVariables = 154, numClauses = 500})
...
```

Listing 1.4. Exemplary inner-under-profiling

Listing 1.4 shows the *inner-under-profiling* for a LPO constraint. For each function f in the abstract program, inner-under-profiling associates the number of variables and clauses to f that has been allocated by f and by all functions transitively called in f. Unsurprisingly, `constraint'` allocates the most resources according to inner-under-profiling as it is the top-level function in every abstract program.

```
Profiling (inner):
("gtNat'", {numCalls = 9, numVariables = 36, numClauses = 171})
("lpo'", {numCalls = 41, numVariables = 30, numClauses = 92})
("ordNat'", {numCalls = 9, numVariables = 27, numClauses = 63})
("eqNat'", {numCalls = 19, numVariables = 27, numClauses = 99})
("and2'", {numCalls = 26, numVariables = 20, numClauses = 44})
("eqOrder'", {numCalls = 31, numVariables = 18, numClauses = 40})
...
```

Listing 1.5. Exemplary under-profiling

Listing 1.5 shows the *under-profiling* for a LPO constraint. For each function f in the abstract program, under-profiling only associates the number of variables and clauses to f that has been allocated by f. Listing 1.5 shows that for the LPO constraint the abstract function `gtNat'` allocates the most propositional variables.

CO^4 also provides information about the number of variables and clauses allocated in the abstract program as a whole:

```
#variables: 167, #clauses: 517, #literals: 1365
```

[1] available at `https://github.com/apunktbau/co4/blob/master/test/CO4/Example/LPO.hs`

We give one more example: Listing 1.6 shows the profiling data for a CO^4 specification of the n-queens problem (with $n = 8$)[2].

```
Profiling (inner-under):
("constraint'", {numCalls = 1, numVariables = 2324, numClauses = 6447})
("allSafe'", {numCalls = 9, numVariables = 2251, numClauses = 6237})
("safe'", {numCalls = 36, numVariables = 2244, numClauses = 6217})
("noAttack'", {numCalls = 28, numVariables = 2216, numClauses = 6140})
("equal'", {numCalls = 717, numVariables = 1724, numClauses = 5100})
("noDiagon'", {numCalls = 28, numVariables = 1488, numClauses = 4012})
("noStraight'", {numCalls = 28, numVariables = 700, numClauses = 2044})
...

Profiling (inner):
("equal'", {numCalls = 717, numVariables = 1724, numClauses = 5100})
("add'", {numCalls = 359, numVariables = 352, numClauses = 704})
("and2'", {numCalls = 101, numVariables = 100, numClauses = 291})
("not'", {numCalls = 84, numVariables = 84, numClauses = 168})
("less'", {numCalls = 65, numVariables = 64, numClauses = 184})

#variables: 2397, #clauses: 6522, #literals: 16697
```

Listing 1.6. Exemplary profiling for the n-queens problem (with $n = 8$)

Here, inner-profiling reveals that the `equal'` function allocates the most resources. This is reasonable because the n-queens constraint pair-wisely compares the position of all queens in order to exclude all possibilities for two queens to attack each other.

7 Moded Types and Mode Inference

For the future work on CO^4, we plan to develop a mode inference system that allows the generation of propositional encodings with fewer variables and clauses. That is desirable as smaller formulas are often solved in less time by a SAT solver.

Moded types allow the differentiation between expressions that are constant during abstract evaluation and expressions that are not. This information would allow the CO^4 compiler to determine case distinctions that have a constant discriminant, i.e., that can be evaluated during abstract evaluation without allocating any propositional variables.

In this context, a mode is either ! or ?. Mode ! states that the constructor of a value is known during abstract evaluation, while mode ? states that the constructor of a value is not known during abstract evaluation. A moded type is a type that has been annotated by modes. For example, `List`$^!$ `Bool`$^?$ denotes a list type, where each of list constructor is known, but each element of type `Bool` has an unknown constructor. Thus, such a type encodes a list of known length with unknown Boolean elements.

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

We consider a moded program to be a typed program where each type is annotated by modes. For a moded program to be dynamically well-moded, it is required that the constructor of all case distinctions' discriminants must be constant that have mode !, i.e., their flags are constant.

We plan to develop a static mode analysis as a safe approximation for dynamically well-moded programs. One possible approach for a mode inference algorithm is the construction of a Boolean constraint (because there are two different modes) that can be solved by a SAT solver.

A similar approach has been successfully applied to infer modes in for the Mercury language[4].

8 Resource Types and Resource Inference

Mode analysis allows a more strict analysis on the estimated cost for a CO^4 constraint system.

A possible approach to predict the resource cost is to annotate each function in a CO^4 constraint with a resource type, where a resource type for function f according to the cost model introduced in Section 4 is a pair of functions $s_f, c_f : \mathbb{N} \to \mathbb{N}$.

A dynamically well-resource-typed program is a program where each function f has a resource type annotation, so that for each call of f with argument x the actual cost $\mathsf{work}(f, \mathsf{encode}(x))$ is less or equal to $c_f(\mathsf{size}(\mathsf{encode}(x)))$ for some cost function work (see Section 4).

A resource-typed program is considered statically well-typed, if all resource annotations are consistent with some sound set of rules for cost of case distinctions, merge operations and function compositions.

These rules should guarantee that the static resource type is a safe approximation for actual costs. We are especially interested in polynomial upper bounds.

Related work consists of amortized resource analysis in Resource Aware ML (RAML)[2], where polynomial potential functions are used as costs functions. The coefficients for these polynomial are determined by a constraint system.

We want to emphasize again that this approach is ongoing work and there are currently no results nor experimental data to verify it. We plan to extend this approach into a reasonable formalism to capture the resource constraints of CO^4 programs in order to estimate the size of the propositional encodings.

References

1. Alexander Bau and Johannes Waldmann. Propositional Encoding of Constraints over Tree-Shaped Data. In *22nd International Workshop on Functional and (Constraint) Logic Programming*, 2013.
2. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, November 2012.

3. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
4. David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP*, pages 109–120, 2002.