Propositional Encoding of Constraints over Tree-Shaped Data

Alexander Bau* Johannes Waldmann

F-IMN, HTWK Leipzig, Germany

July 3, 2013

* supported by an ESF grant

Motivation

using a subset of Haskell for constraint system specifications

- in general: constraint system = formula in predicate logic
- here: $\exists x : f(x)$, with f being quantifier-free
- search for a satisfying assignment for x by generic (i.e. problem independent) techniques
- specification of predicate f as Haskell function

```
constraint :: (Int,Int) -> Bool
constraint (a,b) = a * b == 42
```

search for satisfying assignment through transformation of f to a finite-domain constraint system

- domain: $\{0, 1\}$
- constraint system = formula in propositional logic
- Boolean satisfiablity problem (SAT)

Motivation (II)

$\exists x : f(x)$ `where f :: D -> Bool`

- possible types `D`
  - algebraic data types (`Bool, Maybe a, [a], data T = ...`)
  - restrict depth of recursions
- specification of f
  - pattern-matching, polymorphism, higher-order functions

advantages of using Haskell for constraint system specifications

- application of an established language in another paradigm
- reuse existing code
- simple testing of found solutions against original program
- comparison to similar approaches, e.g. Curry (Hanus et al.)

our contribution: implementation by compilation to SAT

- apply fast SAT solvers like Minisat (Een, Sörensson)

Applications (Work in Progress)

- termination analysis for rewrite systems
  - precedences for path orders
  - coefficients for interpretations
  - models for semantic labelling
  - looping derivations
- computational biology (RNA design)

SAT is assembly language of constraint programming: one wants to use it, but nobody wants to write it

SAT compilation gives

- correctness
- flexibility

Example

```
data Bool       = False | True
data Pair a b = Pair a b
data Nat        = Z | S Nat

add x y = case x of { Z -> y; S x' -> S (add x' y) }

eq x y = case x of
  Z    -> case y of { Z    -> True    ; _ -> False }
  S x' -> case y of { S y' -> eq x' y'; _ -> False }

constraint (Pair x y) = eq (S (S (S Z))) (add x y)

----------------------------------------------------
Start producing CNF
CNF finished (#variables: 71, #clauses: 199)
Starting solver
Solver finished in 0.0 seconds (result: True)
Just (Pair Z (S (S (S Z))))
```

Concept of Implementation

parametric constraint system

```
constraint p x = ...
```

for `p` given at runtime: search for satisfying assingment for `x`

1. compilation-time:
   - transformation of `constraint` to a Haskell function that generates a propositional formula
2. run-time:
   2.1 generate propositional formula
   2.2 solve formula by external SAT solver
   2.3 reconstruct satisfying assingment

main challange: pattern matches on unknown data

Usage

transformation of `constraint` using Template-Haskell during GHC's compilation time

```
$( [d| ...
       constraint (Pair x y) = eq (S (S (S Z)))
                                  (add x y)
   |] >>= runIO . configurable [] . compile
  )

result :: IO (Maybe (Pair Nat Nat))
result = solveBoolean ... encConstraint

main = result >>= putStrLn . show
```

Example (compiled constraint system)

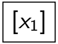
```
encAdd = \encX_6 encY_7 ->
  do bindCase_267 <- return encX_6
     if isInvalid bindCase_267
      then return bindCase_267
      else do bindArgument_274 <- return encY_7
              bindArgument_275 <-
                let encX'_8 = constructorArgument 0 1 bindCase_267
                 in do bindArgument_272 <-
                        do bindArgument_269 <- return encX'_8
                           bindArgument_270 <- return encY_7
                           bindResult_268 <- encAdd bindArgument_269
                                                    bindArgument_270
                           return bindResult_268
                       bindResult_271 <- encSCons bindArgument_272
                       return bindResult_271
              bindResult_273 <- caseOf bindCase_267 [bindArgument_274,
                                                     bindArgument_275]
              return bindResult_273
  ...
```

Data Transformation

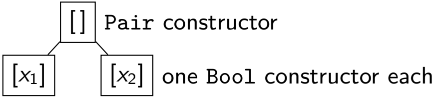
abstract value is a tree that represents a set of concrete values

- each node contains propositional variables $[x_1, x_2, \ldots]$
- they encode the index of a constructor

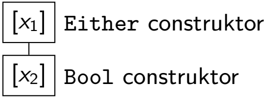
```
data Bool = False | True
```



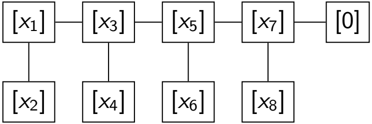

```
data Pair = Pair Bool Bool
```




```
data Either = Left Bool | Right Bool
```




```
data List = Nil | Cons Bool List
```



Program Transformation

pattern matches on unknown data generates clauses of the resulting propositional formula

```
r,e,u,v :: Bool
let r = case e of { False -> u ; True -> v }
```

if

- $\text{abstract-value}(\text{env}, \text{compile}(\mathtt{e})) = [x_e]$
- $\text{abstract-value}(\text{env}, \text{compile}(\mathtt{u})) = [x_u]$
- $\text{abstract-value}(\text{env}, \text{compile}(\mathtt{v})) = [x_v]$

then

- $\text{abstract-value}(\text{env}, \text{compile}(\mathtt{r})) = [(\overline{x_e} \to x_u) \land (x_e \to x_v)]$

program transformation (II)

top-level constraint is applied to two abstract values

- encoded parameter `p`
- abstract value that represents the domain of the unknown `x`
  - depth of abstract value restricts recursion

optimizations

- assumption: smaller formula $\rightarrow$ easier to solve
- direct evaluation of pattern-matches on known data (represented by Boolean constants)
  - do not generate formulas for unreached branches
- memoization of function calls during abstract evaluation
- built-in operations for fixed-width binary numbers

Example - Find Looping Derivations in SRS

```
type Symbol =  [ Bool ]
type Word = [ Symbol ]
type Rule = ( Word , Word )
type SRS = [ Rule ]

-- Step p (l,r) s represents p ++ l ++ s --> p ++ r ++ s
data Step = Step Word Rule Word

data Looping_Derivation = Looping_Derivation Word [Step] Word

constraint :: SRS -> Looping_Derivation -> Bool
constraint srs (Looping_Derivation pre d suf) =
  conformant srs d && eqWord (pre ++ start d ++ suf) (result d)
  ...
```

→ code size: 100 lines

Example - Find Looping Derivations in SRS

```
> ./ttt2 -s 'dp;loop -dp -r 16 -c 16' -pstat \
                    SRS/Gebhardt/03.srs

cnf generated: 23759 vars, 39541 clauses (0.746666)
cnf solved (5.373332)

> CO4/Test/Loop +RTS -K1G -RTS 16 16 SRS/Gebhardt/03.srs

CNF finished (#variables: 132954, #clauses: 450132)
Solver finished in 42.276663 seconds (result: True)
```

Conclusion

- use a subset of Haskell for constraint system specifications
- transformation into satisfiablity problem of propositional formulas
- application 1: terminations analysis of term rewriting systems
  - precedences for path orders
  - coefficients for interpretations
  - models for semantic labelling
  - looping derivations

  corresponding Haskell code is already available (CeTA)
- application 2: RNA design in computational biology
- main challenges:
  smaller formulas, faster compilation, bigger Haskell subset
- try: `https://github.com/apunktbau/co4`
- continue: `http://arxiv.org/abs/1305.4957`