Recent Developments in the Matchbox Termination Prover

Alexander Bau, Tobias Kalbitz, Maria Voigtländer, Johannes Waldmann

Hochschule für Technik, Wirtschaft und Kultur Leipzig

27. Februar 2012

Overview

Problem

- Use of constraint programming in termination provers
  - Finding precedences for path orders
  - Finding matrix interpretations over various domains
  - ...
- Constaints encoded using
  - Boolean formulas, linear equations
  - Embedded domain specific languages (EDSL) (e.g. logic programming)
  - Specification languages (e.g. SMT-lib)
- Proposal: use a more expressive language you're already familiar with
- Solving using
  - Domain-specific methods
  - Generic search
  - Transformation to target constraint domain (here: SAT)
    - Complete: finite domain constraints
    - Incomplete: numbers ($\rightarrow$ restrict bit width)

Satchmo

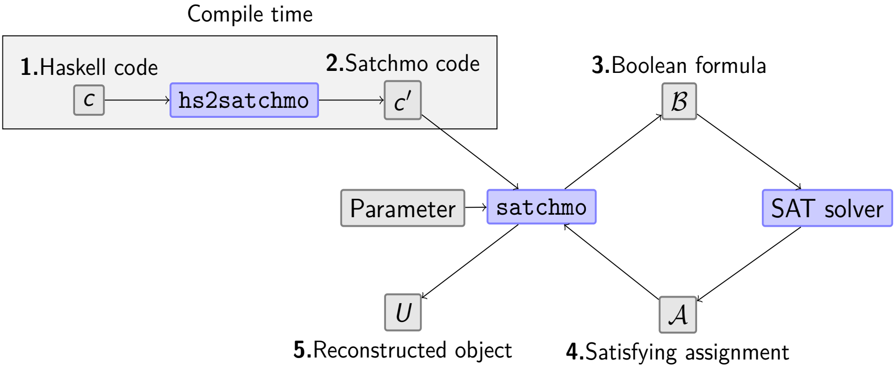
Current situation in Matchbox: encoding of $\exists a\, b\, c\, .\, \neg a \wedge (b \vee c)$

```
constraintSystem :: SAT (Decoder [Boolean])
constraintSystem = do
   [a,b,c] <- sequence [boolean,boolean,boolean]

   tmp1 <- return (not a)
   tmp2 <- or  [b, c]
   tmp3 <- and [tmp1, tmp2]

   assert [ tmp3 ]
   return ( decode [ a, b, c ] )
```

- `SAT` monad as global constraint storage
- state changes through commands (`and, assert, ...`)
  - imperative programming
- Reconstruction of found solution (`decode`)
- Supports integers, polynomials, relations

Transformation overview

- Compile time: Haskell $\to$ Satchmo
- Run time: Satchmo $\to$ Boolean formula



Parameter: e.g. rewriting system

hs2satchmo

- Use Haskell as specification language
- pure, lazy, functional

```
constraintSystem = \a b c -> not a && (b || c)
```

- Type-directed transformation of Haskell abstract syntax tree to monadic code
  - Type determines transformation
- Required extensions to type system:
  - Computations using unknown values (e.g. `Bool?`, `Integer?`)
  - Mixing of known and unknown values (e.g. list of known length of `Bool?`)
- Two ways:
  1. Infer types by yourself
  2. Use extensions of Haskells type system (`MultiParamTypeClasses`, `FunctionalDependencies`)

Matrix interpretations over fuzzy semi ring

(domain: $\{-\infty\} \cup \mathbb{Z} \cup \{+\infty\}$, addition: min, multiplication: max)
fuzzy matrix interpretation = match-bounded automaton

```
fuzzy_interpretation srs int =
  and ( for int mvalid ) && ( and ( for srs ( \ [ lhs, rhs ] ->
     let eval w = foldr1 mtimes ( for w ( \ i -> int !! i ))
     in
       mgreater ( eval lhs ) ( eval rhs )
  )))

fvalid xs = and ( zipWith boolean_geq ( tail xs ) xs)

fplus xs ys = zipWith ( && ) xs ys

ftimes xs ys = zipWith ( || ) xs ys

...
```

Matrix interpretations over fuzzy semi ring (II)

```
fuzzy_interpretation srs[a3C3] int[a3C4]
  = do { bind[a3Dg] <- for int[a3C4] mvalid;
         bind[a3Dh] <- and bind[a3Dg];
         let lambda[a3Dp] [lhs[a3C5], rhs[a3C6]]
              = do { let eval[a3C7] w[a3C8]
                           = do { let lambda[a3Dj] i[a3C9]
                                        = do { bind[a3Di] <- (int[a3C4
                                               return bind[a3Di] };
                                  bind[a3Dk] <- for w[a3C8] lambda[a3D
                                  bind[a3Dl] <- foldr1 mtimes bind[a3D
                                  return bind[a3Dl] };
                     bind[a3Dm] <- eval[a3C7] lhs[a3C5];
                     bind[a3Dn] <- eval[a3C7] rhs[a3C6];
                     bind[a3Do] <- mgreater bind[a3Dm] bind[a3Dn];
                     return bind[a3Do] };
         bind[a3Dq] <- for srs[a3C3] lambda[a3Dp];
         bind[a3Dr] <- and bind[a3Dq];
         bind[a3Ds] <- (bind[a3Dh] && bind[a3Dr]);
         return bind[a3Ds] }
```

Matrix interpretations over fuzzy semi ring (III)

```
SRS/Zantema/z002.srs
(RULES b c a  -> a b a b , b  -> c c , a a  -> a c b a )

mresult <- Satchmo.SAT.Mini.run ( do
   int <- unknowns 3 4 4
   r   <- fuzzy_interpretation z002 int
   assert [r] ; return ( decode int ) )

start producing CNF ; CNF finished vars 5522 clauses 15862
starting solver ; solver finished

-- interpretation as list of matrices of unary numbers:
mresult = Just [[[[False,True,True,True],[True,True,True,True],[Fals

-- Interpretation  (4 = +infty, 0 = -infty)
[[[3,4,1,4],[4,4,4,4],[3,2,4,4],[2,1,0,4]], [[2,4,2,4],[4,4,4,1],[1

( runIdentity . fuzzy_constraint z002 ) <$> mresult
Just True
```

Future plans: Complexity

- Complexity measure:
  - # of clauses/variables of resulting boolean formula as function of size of parameter
  - Run time for solving
- $\rightarrow$ instance of estimation of resource usage of functional programs
- Refined types contain complexity information
- Modular approach: if f depends on g, `type-of`(f) depends on `type-of`(g), but not on g's implementation
- Programmer may provides complexity type declarations and compiler checks them

Conclusion

- Constraint programming is a useful tool in termination provers
- Encoding of problem into target domain can be hard, therefore:
- Two step transformation of more expressive language into existing domain specific language:
  - Haskell → Satchmo → Boolean formula
- Challenges:
  - Handling mixings of known/unknown data
  - Type-directed transformation
- Current state: working prototype compiler
- Next steps
  - Correctness proof
  - Support for more structured data types
  - Standalone solver for SMT-lib problems (QF_BV, NIA)
- Future plans: complexity analysis

Overview

Constraint Compiler

Massively Parallel Constraint Solving

Summary

Basic Ideas

- constraint satisfaction problem $\Rightarrow$ optimization problem
  domain: assignments, objective function: penalty value ("degree of satisfaction"), zero penalty $=$ solution
- evolutionary optimization: population (finite subset of the domain), modify by mutation, recombination, prefer "fitter" individuals
- used by Dieter Hofbauer's termination prover *MultumNonMulta* (Competitions 2006, 2007)

New Ideas

- parallelization, on graphics hardware: (e.g., GTX 580, with 512 compute cores, 1.4 GHz, for $\leq$ 500 EUR)
  - parallel matrix multiplication (in computation of penalty)
  - treat several individuals in parallel
- apply Multum-Non-Multa path mutation idea for rational and arctic domain.
- embed within Matchbox
  can re-use problem parser, strategy combinators, other solvers (simplex, SAT), (certified) proof output

Fitness (Penalty) computation

for relative termination problem R/S over alphabet Σ,
penalty of a d-dimensional matrix interpretation $[\cdot] : \Sigma \to D^{d\times d}$ is

- (weak compatibility for $R \cup S$)

  $$\sum\{f\cdot\Delta^2 \mid (l,r) \in R\cup S, 1 \le \{p,q\} \le d, \text{let } \Delta = [l]_{p,q} - [r]_{p,q}, \Delta \not\geq 0\}$$

  where
  $f = (\text{if } (p,q) = (1,d) \text{ then } 10^3 \text{ else } 1) \cdot (\text{if } x = 0 \text{ then } 10^3 \text{ else } 1)$
- plus (strict compatibility for R) $\sum\{10^9 \mid (l,r) \in R, [l]_{1,d} \not> [r]_{1,d}\}$.

To prove termination of R by "removal of rules", the driver program creates a set of sub-problems

$$\{u\}/(R \setminus \{u\}) \text{ for } u \in R,$$

and evolution handles these by (simulated) parallelism.

Mutation

- *large mutation*: randomly pick some error position $(l, r) \in R, 1 \leq \{p, q\} \leq d$ such that $[l]_{p,q} \not\geq [r]_{p,q}$.
  Let $l = a_1 \dots a_n$. Randomly choose a path $p = p_0 \overset{a_1}{\to} p_1 \dots p_{n-1} \overset{a_n}{\to} p_n = q$ and increase each $[a_i]_{p_{i-1},p_i}$ by 1.
  - this ensures that $[l]_{p,q}$ increases.
  - idea is derived from automata completion for proving match-bounds, and was used in Multum-Non-Multa
  - $[r]_{p,q}$, and "unrelated entries", might increase as well, therefore ...
- *small mutation*: pick $a \in \Sigma, 1 \leq \{p, q\} \leq d$, modify $[a]_{p,q}$ by ± 1.

control flow: do one large mutation, then try several (e.g., 1000) small mutations, accepting only if they are decreasing penalty.
(actually, "not increasing penalty", since this helps biodiversity)

Results

- Sequential implementation of evolution works,
  is used for testing fitness functions and mutation operators.
- some test run on Termcomp Platform,
  results on TPDB are similar to those of Multum-Non-Multa (with simplex solver).
- "killer examples" (not solved in "full run" 2011, in competition 2007):
  - SRS/Trafo/un03 (relative termination)
  - SRS/Trafo/un15

  uses matrix dimensions ≥ 6
- Parallel (CUDA) version is still being improved (as student projects)

SRS/Trafo/un03

```
YES
Claim: system (RULES b a b a b -> b a a b a a a b,
                     b a a b a a b -> b a a a b a a a b a a a b,
                     b a a a b a a a b a a a b -> b b a a b,
                     b b b ->= b a b a a b,
                     b a b a a b ->= b b b)
       is terminating
is true because
remove rules by interpretation
Natural a |-> 1 0 1 2 0 0 0    b |-> 1 2 0 1 1 1 0
              0 0 0 2 0 1 0          0 0 0 0 0 0 1
              0 1 0 4 0 1 0          0 0 0 0 0 0 0
              0 0 0 2 0 1 0          0 0 0 0 0 0 0
              0 2 1 0 0 3 1          0 0 0 0 0 0 1
              0 0 0 0 0 3 0          0 0 0 0 0 0 0
              0 0 0 0 0 0 1          0 0 0 0 0 0 1
```

http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=350227&cid=340

Conclusion

massively parallel matrix constraint solving for termination:
simple idea, *but* only works due to

- application-specific knowledge (fitness function, “large” mutation)
- hardware-specific knowledge (CUDA execution model)

Main design dilemma for massively parallel matrix solvers:

- matrix dimension d should be large:
  - cost for matrix multiplication is
    sequential: d^3 time, parallel: d time ($\times$ d^2 processors)
  - there is some overhead for thread synchronisation
  - for small d, solutions can already be found by SAT-based solvers
- matrix dimension should be small:
  - each full sequence of small mutations (including evaluations) should run in fast memory — but this is a scarce resource (16 kByte),
  - larger $d \Rightarrow$ larger search space (evolution needs more steps)

our impression: matrix dimensions $8 \ldots 16$ are feasible

Overview

Constraint Compiler

Massively Parallel Constraint Solving

Summary

Summary

1. Constraint compiler
   - Transformation of more expressive language into existing domain specific language:
     - Haskell $\rightarrow$ Satchmo $\rightarrow$ Boolean formula
   - Handling mixings of known/unknown data
   - Type-directed transformation
   - Future: complexity analysis
2. Massively Parallel Constraint Solving
   ...it works (sort of), but only due to
   - application-specific knowledge (fitness function, “large” mutation)
   - hardware-specific knowledge (CUDA execution model)