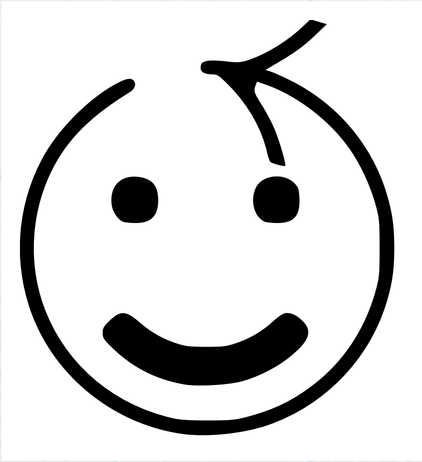
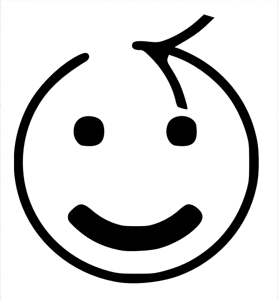
SAT/SMT solving in Haskell



Masahiro Sakai (酒井 政裕)

Haskell Day 2016
2016-09-17



Self Introduction

Masahiro Sakai

* Twitter: @masahiro_sakai
* github: https://github.com/msakai/
* G+: https://plus.google.com/+MasahiroSakai

* Translated “Software Abstractions” and TaPL into Japanese with colleagues
* Interests: Categorical Programming, Theorem Proving / Decision Procedures, …

Agenda

* What are SAT and SMT?
* Haskell libraries for SMT solving
  * sbv
  * toysat/toysmt
* Conclusion

What are SAT and SMT?

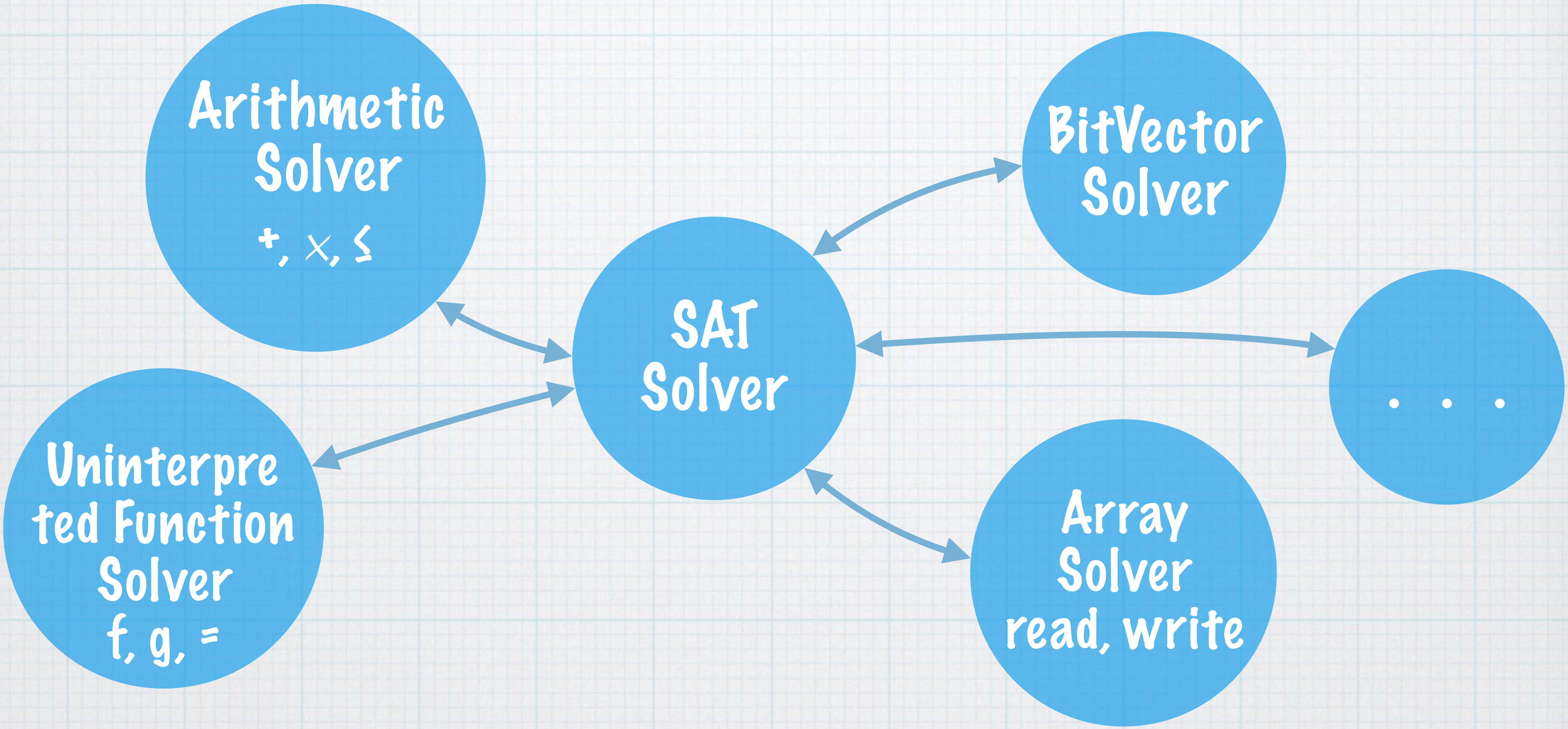
What is SAT?

- * SAT = Boolean **SAT**isfiability problem
    - "Is there an assignment that makes given formula true?"
    - Examples:
        - $(P \lor Q) \land (P \lor \neg Q) \land (\neg P \lor \neg Q)$ is satisfiable with $\{P \mapsto \text{True}, Q \mapsto \text{False}\}$
        - $(P \lor Q) \land (P \lor \neg Q) \land (\neg P \lor \neg Q) \land (\neg P \lor Q)$ is unsatisfiable
- SAT is **NP complete**, but state-of-the-art SAT-solver can often solve problems with **millions of variables / constraints**.

What is SMT?

* Weakness of SAT: Really low-level representation
  * Encoding problems into SAT sometimes blows-up
  * SAT solver cannot leverage high-level knowledge
* SMT = **S**atisfiability **M**odulo **T**heories
  * An approach to overcome the weakness of SAT
  * Problem Example:
    Is there array a, function f, integers i, j such that
    "$0 \leq i \wedge i < 10 \wedge (2i+1 = j \vee read(a,i)=0) \wedge f(read(write(a,i,3), j-2)) \neq f(j-i+1)$"?

SMT Solver Impl.
SAT Solver + Theory solvers



* SAT solver is responsible for Boolean reasoning
* Theory solvers are responsible for handling specific functions/relations etc.

Some Applications of SAT/SMT

- Software/Hardware verification
  - Model checking, Test-case generation, ...
- Theorem proving
- Puzzles: Sudoku, Numberlink, Nonogram, etc.
- Type checking in Liquid Haskell
  - eg: doubles :: [{x : Int | x >= 0}]→[{x : Int | x `mod` 2 = 0}]
- Program Synthesis
- and more

Haskell libraries for SMT solving

Some Haskell packages for SMT

* Binding
  * sbv, smtlib2, simple-smt
  * z3, bindings-yices, yices-easy, yices-painless
* SMT solvers written in Haskell:
  * toysolver, Smooth
* SMT-LIB2 file parser/printer
  * smt-lib, SmtLib

SMT-LIB2 is a standard input/output format for SMT solvers

SBV: SMT Based Verification in Haskell

* SMT library developed by Levent Erkok
* It provides:
    * High-Level DSL for specifying problems in Haskell, and
    * Interfaces to multiple SMT solver backends including Z3, CVC4, Yices, Boolector.
* You can install simply using stack/cabal
    * "stack install sbv" or "cabal install sbv"

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
        ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
        let isDigit x = x .>= 0 &&& x .<= 9
            val xs     = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
            send       = val [s,e,n,d]
            more       = val [m,o,r,e]
            money      = val [m,o,n,e,y]
        constrain $ bAll isDigit ds
        constrain $ allDifferent ds
        constrain $ s ./= 0 &&& m ./= 0
        solve [send + more .== money]
```

SEND
+MORE

MONEY

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
        ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
        let isDigit x = x .>= 0 &&& x .<= 9
            val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
            send      = val [s,e,n,d]
            more      = val [m,o,r,e]
            money     = val [m,o,n,e,y]
        constrain $ bAll isDigit ds
        constrain $ allDifferent ds
        constrain $ s ./= 0 &&& m ./= 0
        solve [send + more .== money]
```

SMT problem is defined using Symbolic monad,
and SMT solving is performed by
sat :: Symbolic SBool → IO SatResult

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
        ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
        let isDigit x = x .>= 0 &&& x .<= 9
            val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
            send      = val [s,e,n,d]
            more      = val [m,o,r,e]
            money     = val [m,o,n,e,y]
        constrain $ bAll isDigit ds
        constrain $ allDifferent ds
        constrain $ s ./= 0 &&& m ./= 0
        solve [send + more .== money]
```

sInteger :: String → Symbolic SInteger

creates integer variable

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
    ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
    let isDigit x = x .>= 0 &&& x .<= 9
        val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
        send      = val [s,e,n,d]
        more      = val [m,o,r,e]
        money     = val [m,o,n,e,y]
    constrain $ bAll isDigit ds
    constrain $ allDifferent ds
    constrain $ s ./= 0 &&& m ./= 0
    solve [send + more .== money]
```

Comparison over symbolic values:
we have to use slightly difference operators like (.>=), (&&&).
Because Haskell's (>=), (&&) returns Bool, but we want SBool.

SBV Example: “send more money”

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
      ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
      let isDigit x = x .>= 0 &&& x .<= 9
          val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
          send      = val [s,e,n,d]
          more      = val [m,o,r,e]
          money     = val [m,o,n,e,y]
      constrain $ bAll isDigit ds
      constrain $ allDifferent ds
      constrain $ s ./= 0 &&& m ./= 0
      solve [send + more .== money]
```

val :: [SInteger] → SInteger is defined as in normal Haskell.

Thanks to the Num type class.

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
    ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
    let isDigit x = x .>= 0 &&& x .<= 9
        val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
        send      = val [s,e,n,d]
        more      = val [m,o,r,e]
        money     = val [m,o,n,e,y]
    constrain $ bAll isDigit ds
    constrain $ allDifferent ds
    constrain $ s ./= 0 &&& m ./= 0
    solve [send + more .== money]
```

Actual constraints specification

SBV Example: “send more money”

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO SatResult
sendMoreMoney = sat $ do
        ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
        let isDigit x = x .>= 0 &&& x .<= 9
            val xs     = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
            send       = val [s,e,n,d]
            more       = val [m,o,r,e]
            money      = val [m,o,n,e,y]
        constrain $ bAll isDigit ds
        constrain $ allDifferent ds
        constrain $ s ./= 0 &&& m ./= 0
        solve [send + more .== money]
```

```
Satisfiable. Model:
  s = 9 :: Integer
  e = 5 :: Integer
  n = 6 :: Integer
  d = 7 :: Integer
  m = 1 :: Integer
  o = 0 :: Integer
  r = 8 :: Integer
  y = 2 :: Integer
```

You need SMT solver Z3 to run the code.

SBV Example: "send more money"

Data.SBV.Examples.Puzzles.SendMoreMoney module

```
sendMoreMoney :: IO AllSatResult
sendMoreMoney = allSat $ do
        ds@[s,e,n,d,m,o,r,y] <- mapM sInteger ["s", "e", "n", "d", "m", "o", "r", "y"]
        let isDigit x = x .>= 0 &&& x .<= 9
            val xs    = sum $ zipWith (*) (reverse xs) (iterate (*10) 1)
            send      = val [s,e,n,d]
            more      = val [m,o,r,e]
            money     = val [m,o,n,e,y]
        constrain $ bAll isDigit ds
        constrain $ allDifferent ds
        constrain $ s ./= 0 &&& m ./= 0
        solve [send + more .== money]
```

By changing sat :: Symbolic SBool → IO SatResult with

allSat :: Symbolic SBool → IO AllSatResult

SBV Summary

* This is only one example and sbv includes variety of examples. You should try!

Examples
- *BitPrecise*
  - Data.SBV.Examples.BitPrecise.BitTricks
  - Data.SBV.Examples.BitPrecise.Legato
  - Data.SBV.Examples.BitPrecise.MergeSort
  - Data.SBV.Examples.BitPrecise.MultMask
  - Data.SBV.Examples.BitPrecise.PrefixSum
- *CodeGeneration*
  - Data.SBV.Examples.CodeGeneration.AddSub
  - Data.SBV.Examples.CodeGeneration.CRC_USB5
  - Data.SBV.Examples.CodeGeneration.Fibonacci
  - Data.SBV.Examples.CodeGeneration.GCD
  - Data.SBV.Examples.CodeGeneration.PopulationCount
  - Data.SBV.Examples.CodeGeneration.Uninterpreted
- *Crypto*
  - Data.SBV.Examples.Crypto.AES
  - Data.SBV.Examples.Crypto.RC4
- *Existentials*
  - Data.SBV.Examples.Existentials.CRCPolynomial
  - Data.SBV.Examples.Existentials.Diophantine
- *Misc*
  - Data.SBV.Examples.Misc.Auxiliary
  - Data.SBV.Examples.Misc.Enumerate
  - Data.SBV.Examples.Misc.Floating
  - Data.SBV.Examples.Misc.ModelExtract
  - Data.SBV.Examples.Misc.NoDiv0
  - Data.SBV.Examples.Misc.Word4
- *Polynomials*
  - Data.SBV.Examples.Polynomials.Polynomials
- *Puzzles*
  - Data.SBV.Examples.Puzzles.Birthday
  - Data.SBV.Examples.Puzzles.Coins
  - Data.SBV.Examples.Puzzles.Counts
  - Data.SBV.Examples.Puzzles.DogCatMouse
  - Data.SBV.Examples.Puzzles.Euler185
  - Data.SBV.Examples.Puzzles.Fish
  - Data.SBV.Examples.Puzzles.MagicSquare
  - Data.SBV.Examples.Puzzles.NQueens
  - Data.SBV.Examples.Puzzles.SendMoreMoney
  - Data.SBV.Examples.Puzzles.Sudoku
  - Data.SBV.Examples.Puzzles.U2Bridge
- *Uninterpreted*
  - Data.SBV.Examples.Uninterpreted.AUF
  - Data.SBV.Examples.Uninterpreted.Deduce
  - Data.SBV.Examples.Uninterpreted.Function
  - Data.SBV.Examples.Uninterpreted.Shannon
  - Data.SBV.Examples.Uninterpreted.Sort
  - Data.SBV.Examples.Uninterpreted.UISortAllSat

toysolver package

* I'm implementing some decision procedure in Haskell to leaning the algorithms
  * https://github.com/msakai/toysolver
  * http://hackage.haskell.org/package/toysolver
* It contains some algorithms/solvers.
* In particular, it contains a SAT solver 'toysat' and SMT solver 'toysmt'

Recalling Last Year …

http://www.slideshare.net/sakai/satsmt

* At Proof Summit 2015, I talked about how SAT/SMT solver works.
* At that time, I already had implemented SAT solver 'toysat', but not implemented SMT solver yet.
* It triggered my motivation to implement a SMT solver, I worked hard, and finally I did it!

toysat / toysmt

* Written in pure Haskell
  * but implemented in very imperative way
* toysat is modestly fast.
  * It was once the fastest among SAT solvers written in Haskell. But now mios by Shoji Narazaki is faster.
* toysmt is slow, and has very limited features.

toysmt

* toysat based SMT solver
    * implementation is really native and not-efficient at all
* Theories
    * Equality and Uninterpreted functions ✓
    * Linear Real Arithmetic ✓
    * Bit-vector (currently implementing)
    * Linear Integer Arithmetic, Array, etc. (not yet)

toysmt: demonstration

QF_UFLRA.smt2

```
(set-option :produce-models true)
(set-logic QF_UFLRA)
(declare-sort U 0)
(declare-fun x () Real)
(declare-fun f (U) Real)
(declare-fun P (U) Bool)
(declare-fun g (U) U)
(declare-fun c () U)
(declare-fun d () U)
(assert (= (P c) (= (g c) c)))
(assert (ite (P c) (> x (f d)) (< x (f d))))
(check-sat)
(get-model)
(exit)
```

toysmt: demonstration

```
$ toysmt  QF_UFLRA.smt2
success
...
sat
((define-fun P ((x!1 U)) Bool
   (ite (= x!1 (as @3 U)) true false))
 (define-fun c () U (as @3 U))
 (define-fun d () U (as @4 U))
 (define-fun f ((x!1 U)) Real
   (ite (= x!1 (as @4 U)) 0 (/ 555555 1)))
 (define-fun g ((x!1 U)) U
   (ite (= x!1 (as @3 U)) (as @3 U) (as @-1 U)))
 (define-fun x () Real (/ 1 10)))
```

For those who do not read SEXP

```
U = {@-1, @1, ..., @4, ...}
x = 1/10 : Real
c = @3  : U
d = @4  : U
P(x) = if x = @3 then true else false
f(x) = if x = @4 then 0 else 55555
g(x) = if x = @3 then @3 else @-1
```

toysmt in SMT-COMP 2016

QF_LRA (Main Track)

http://smtcomp.sourceforge.net/2016/results-QF_LRA.shtml?v=1467876482

Solver	Sequential performance			Parallel performance				Other information
	Error Score	Correctly Solved Score	avg. CPU time	Errors	Corrects	avg. CPU time	avg. WALL time	Unsolved benchmarks
CVC4	0.000	1601.997	61.989	0.000	1601.997	62.004	62.088	20
MathSat5[n]	0.000	1574.475	109.915	0.000	1574.475	109.957	109.870	64
OpenSMT2	0.000	1510.710	214.228	0.000	1510.710	214.323	214.198	263
SMT-RAT	0.000	1415.279	344.351	0.000	1415.279	344.538	344.329	433
SMTInterpol	0.000	1571.240	118.790	0.000	1572.061	127.002	112.669	54
Yices2	0.000	1593.356	65.700	0.000	1593.356	65.730	65.654	34
toysmt	0.000	1172.885	582.313	0.000	1172.885	582.500	582.280	811
veriT-dev	0.000	1577.045	95.717	0.000	1577.045	95.759	95.701	55
z3[n]	0.000	1547.832	157.261	0.000	1547.832	157.327	157.232	145

'toysmt' ended up dead last.
But without wrong results! (Thanks to QuickCheck!)

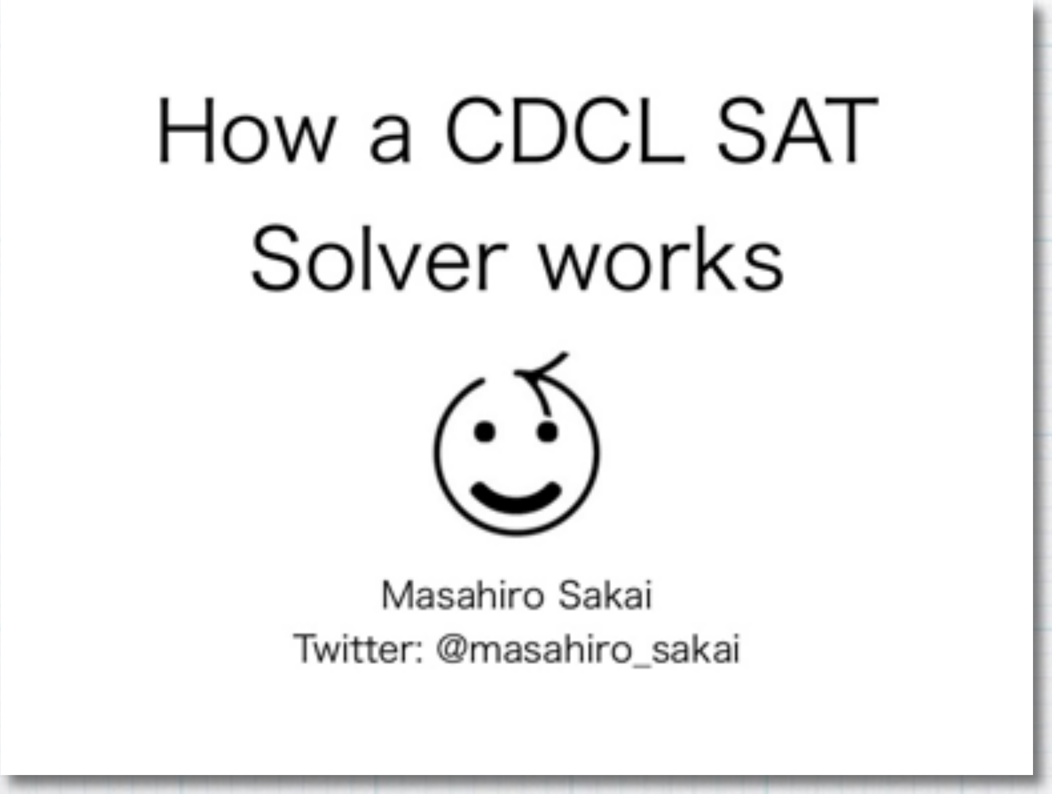
toysmt: Future work

* Fill the gap with state-of-the-art solvers (even a little)
  * There're lots of rooms for performance improvement.
  * More theories: Bit-vectors, Integer arithmetic, Array, ...
  * More features: e.g. Proof-generation
* Using 'toysmt' as a backend of 'sbv'.
* Re-challenge in next year's SMT-COMP competition.

Conclusion

* SAT solvers are amazingly fast for solving many combinatorial problems
* SMT is an extension of SAT to handle high-level constraints using specialized solvers.
* sbv is a neat Haskell library for using SMT solvers
* toysmt is a SMT solver written in Haskell

Further readings



http://www.slideshare.net/sakai/how-a-cdcl-sat-solver-works

http://www.slideshare.net/sakai/satsmt

Further readings

* Handbook of Satisfiability
    * A. Biere, M. Heule, H. Van Maaren, and T. Walsh, Eds.
    * IOS Press, Feb. 2009.
* It is a very good book covering variety of topics related to SAT/SMT.