Seri

A High-Level User Language for SMT Queries

Richard Uhler

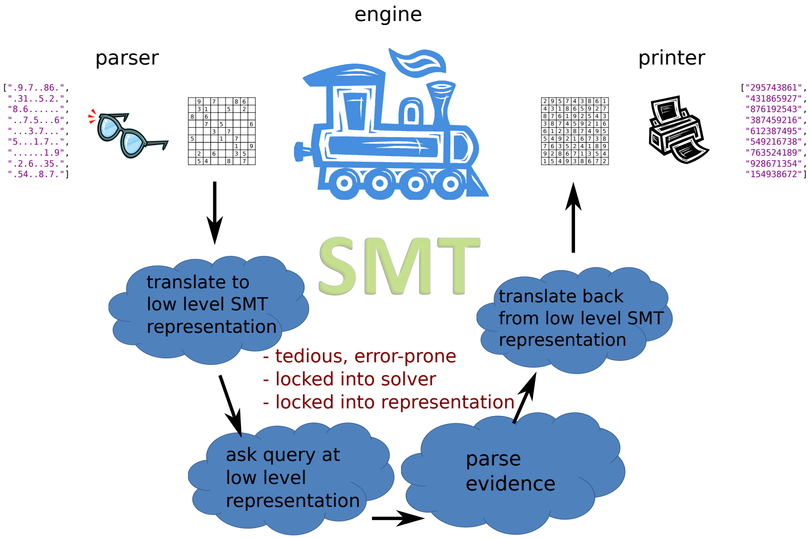
August 30, 2012

How to Implement a Sudoku Solver

```haskell
diabolical :: [[Char]]
diabolical =
     [".9.7..86.",
      ".31..5.2.",
      "8.6......",
      "..7.5...6",
      "...3.7...",
      "5...1.7..",
      "......1.9",
      ".2.6..35.",
      ".54..8.7."]
```

	9		7			8	6	
	3	1			5		2	
8		6						
		7		5				6
			3		7			
5				1		7		
						1		9
	2		6			3	5	
	5	4			8		7	

How to Implement a Sudoku Solver



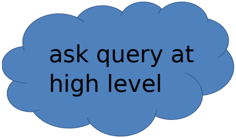
How to Implement a Sudoku Solver

engine

parser

```
[".9.7..86.",
 ".31..5.2.",
 "8.6......",
 "..7.5...6",
 "...3.7...",
 "5...1.7..",
 "......1.9",
 ".2.6..35.",
 ".54..8.7."]
```

	9		7			8	6	
	3	1			5		2	
8		6						
		7		5				6
			3		7			
5				1		7		
						1		9
	2		6			3	5	
	5	4			8		7	



2	9	5	7	4	3	8	6	1
4	3	1	8	6	5	9	2	7
8	7	6	1	9	2	5	4	3
3	8	7	4	5	9	2	1	6
6	1	2	3	8	7	4	9	5
5	4	9	2	1	6	7	3	8
7	6	3	5	2	4	1	8	9
9	2	8	6	7	1	3	5	4
1	5	4	9	3	8	6	7	2

printer

```
["295743861",
 "431865927",
 "876192543",
 "387459216",
 "612387495",
 "549216738",
 "763524189",
 "928671354",
 "154938672"]
```

SMT

- solver agnostic
- easy to change representations
- intuitive queries
- leverage existing expertise

The Parser in Seri

```
data Cell = C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9
data Board = Board [[Cell]]
```

algebraic data types

```
readCell :: Char -> Query Cell
readCell '1' = return C1
readCell '2' = return C2
...
readCell '9' = return C9
readCell '.' = free
readCell  c  = error ("readCell:" ++ [c])
```

pattern matching

introduce a free variable

```
readRow :: [Char] -> Query [Cell]
readRow = mapM readCell

readBoard :: [[Char]] -> Query Board
readBoard rows = do
    brows <- mapM readRow rows
    return (Board brows)
```

The Sudoku Constraints

```
instance Eq Cell where
    (==) C1 C1 = True
    (==) C2 C2 = True
    ...
    (==) C9 C9 = True
    (==) _ _ = False
```

ad-hoc polymoprhism (overloading)

recursive functions

```
unique :: (Eq a) => [a] -> Bool
unique [] = True
unique (x:xs) = notElem x xs && unique xs
```

```
rows :: Board -> [[Cell]]
rows (Board x) = x

cols :: Board -> [[Cell]]
cols (Board x) = transpose x

boxes :: Board -> [[Cell]]
boxes = ...
```

higher order functions

```
isvalid :: Board -> Bool
isvalid b = all unique (concat [rows b, cols b, boxes b])
```

The Sudoku Solver

```
solve :: [[Char]] -> Query [[Char]]
solve input = do
    board <- readBoard input
    assert (isvalid board)
    result <- query board
    case result of
        Unsatisfiable -> return ["no solution"]
        Satisfiable v -> return (print v)

main :: Query [[Char]]
main = solve diabolical
```

- Yices1: 1m15s
- Yices2: 1.6s

No change in source code going from Yices1 to Yices2!

A Different Cell Representation

```
data Cell = Cell (Bit #9)

instance Eq Cell where
    (==) (Cell a) (Cell b) = (a == b)

freeCell :: Query Cell
freeCell = do
    x <- free
    assert (isValidCell x)
    return x

join :: [Bit #n] -> Bit #n
join [] = 0
join (x:xs) = bv_or x (join xs)

value :: Cell -> Bit #9
value (Cell x) = x

unique :: [Cell] -> Bool
unique cells = join (map value cells) == 0x1ff
```

- Yices1: 3m53s (was 1m15s)
- Yices2: 1.0s (was 1.6s)

Interactive and Reusable Queries

```haskell
allQ :: (Eq a) => (a -> Bool) -> Query [a]
allQ p = do
    x <- free
    assert (p x)
    r <- query x
    case r of
        Unsatisfiable -> return []
        Satisfiable v -> do
            vs <- allQ (\a -> (p a) && (a /= v))
            return (v:vs)

predicate :: Integer -> Bool
predicate x = (x > 3) && (x < 6)

main :: Query [Integer]
main = allQ predicate
```

Current Status, Future Plans

Current Status

- Yices1, Yices2 supported
- All queries shown work

Future Work (Ph.D. Thesis)

- Optimize generated queries
- Add support for more solvers
- Integrate tool more seamlessly with Haskell
- Explore implementation of formal tools, such as model checkers, built in Seri with reusable library components