Sudoku mit SMT

Formale Methoden und Werkzeuge WS24/25 - Oscar Friske

Dieses Projekt implementiert einen Sudoku Solver, sowie einen Generator für neue Sudoku Instanzen. Zur Modellierung des SMT Problems wurde die Bibliothek "hasmtlib" und Haskell als Hostsprache genutzt.

SMT Spezifikationen

- Solver: Z3
- Logik: QF_LIA - linear integer arithmetic

Implementierung

Folgende elementare Datentypen wurden für die Implementierungen des Solvers sowie den Instanzengenerator verwendet:

- Sudoku Instanz: [[Integer]]
  Eine Sudokuinstanz ist eine Liste von Zeilen, wobei jede Zeile eine Liste von Integern ist. Die Zahlen repräsentieren die Werte der Felder.
- Eintrag im Sudokubrett: (Int, (Int, Int))
  Ein Feld eines Sudokubretts besteht aus dem Wert des Feldes und den Koordinaten (Spalte, Zeile)
- vorgegebene Werte (als Eingabe): [Cell]
  Diese Liste gibt die Felder für den Solver an, welche bereits vorgegeben sind.
- vorgegebene Werte (als Ausgabe): [Cell]
  Diese Liste ist die Ausgabe einer neuen (unvollstädnigen) Sudoku Instanze, welche vom Generator ausgegeben wird.
- vorgegebene Werte (als Ausgabe): [Cell]
  Diese Liste ist die Ausgabe einer neuen (unvollstädnigen) Sudoku Instanze, welche vom Generator ausgegeben wird.

Das Projekt ist grundsätzlich in 2 Teile unterteilt:

(1) Solve.hs
Dieser Teil implementiert den eigentlichen Solver für gegebene (unvollständige) Sudokuinstanzen. Dabei wird das Problem in SMT modelliert und anschließend von Z3 gelöst. Für die Modellierung bzw. für das Spiel an sich gibt es 3 wesentliche Constraints:

(a) Row Constraint
Jede Zeile muss genau die Zahlen 1-9 enthalten (keine Doppelungen)
Dies ist bei der gegebenen Datenstruktur leicht umsetzbar, da das Feld bereits als Liste von Zeilen vorliegt. Durch das SMT Keyword "distinct" kann die Eigenschaft leicht überprüft werden.

```
forM_ board $ \row -> assert $ distinct row
```

(b) Column Constraint
Jede Spalte muss genau die Zahlen 1-9 enthalten (keine Doppelungen)
Auch dieses Constraint folgt grundsätzlich dem gleichen Aufbau des Row Constraints. Allerdings muss hierfür die Matrix zunächst transponiert werden. Somit erhalten wir eine Liste von Spalten.

```
forM_ (transpose board) $ \column -> assert $ distinct column
```

(c) Subgrid Constraint
Jedes 3x3 Feld muss genau die Zahlen 1-9 enthalten (keine Doppelungen)
Hierfür müssen die 3x3 Subgrids vorher in eigene Listen gefiltert werden. Dafür werden beginnend bei einer Koordiante (column, row) die nächsten 3 Zeilen in den nächsten 3 Spalten genommen. Siehe *getSubgridAtPosition* und *getAllSubgrids* in Generate.hs

```
let subgrids = getAllSubgrids board
forM_ subgrids $ \subgrid -> do
    assert $ distinct subgrid
```

Des weiteren gibt es zwei Constraints, welche den Zahlenraum einschränken (1-9), ein Constraint, welches die vordefinierten Felder voraussetzt und ein optionales Constraint, welches eine Lösung ausschließt. Dies wird benötigt um später eine Lösung auf ihre Einzigartigkeit zu überprüfen und kommt beim Generieren eines neuen Rätsels zum Einsatz.

```
forM_ predefinedValues $ \(entry, (column, row)) ->
    assert $ ((board !! column) !! row) == fromIntegral entry

forM_ (concat board) $ assert . (>? 0)
forM_ (concat board) $ assert . (<? 10)

case excludedBoard of
  Just boardToExclude -> assert $ not $ foldl1 (&&) $ zipWith (===)
    (concat boardToExclude) (concat board)
  Nothing -> return ()
return board
```

Es gibt einen pattern match auf "excludedBoard", da das Constraint nur beim Generieren eines neuen Rätsels benötigt wird und nicht beim eigentlichen Lösen eines Rätsels.

(2) Generate.hs
Auf Basis des Solvers können neue Sudokuinstanzen generiert werden. Die grundsätzliche Idee ist dabei folgende:

(1) Generiere n zufällige Felder und Löse basierend auf diesen ein Sudokubrett.
Sollten die zufälligen Felder eine Lösung des Sudokus verhindern, werden neue Zufallsfelder generiert. Hier muss abgewogen werden zwischen der Lösungsgeschwindigkeit des Solvers (mehr vorgegebene Werte beschleunigen das Lösen) und der möglichen Anzahl der Neugenerierungen, da die Felder keine gültige Lösung zulassen. 10 Zufällige Anfangswerte haben sich als gutes Mittelmaß herausgestellt.
Siehe *createCompleteBoard* in Generate.hs

(2) Entferne ein zufälliges Feld
Die Lösbarkeit bleibt dadurch erhalten.
Siehe *deleteRandomElement* in Generate.hs

(3) Überprüfe, ob das Sudoku weiterhin nur eine mögliche Lösung hat [1].

i. Falls ja, gehe zu Schritt (2)

ii. Falls nein, mache den letzten Schritt rückgängig und gehe zu (2).
Das ist allerdings nur n mal möglich. Sollte nach n Wiederholungen keine bessere Instanz (mit weniger Feldern) gefunden wurden sein, beende die Generierung.

Schritt (2) und (3) sind Teil der *reduceBoard* Funktion in Generate.hs

[1] Um zu überprüfen, ob das aktuelle Brett nur eine mögliche Lösung hat, durchläuft das aktuelle (unvollständige) Brett den Solver zwei mal. Im ersten Durchlauf wird das Brett auf seine generelle Lösbarkeit untersucht. Das Ergebnis wird anschließend im zweiten Durchlauf dem Solver übergeben mit der Bedingung, dass dies keine Lösung sein darf. Wird dennoch eine Lösung gefunden, war die Lösung des ersten Durchlaufs nicht die einzige. Wird hingegen keine weitere Lösung gefunden weiß man, dass das Sudoku im aktuellen Zustand nur eine Lösung hat.
Siehe *checkIfUniqueSolution* in Generate.hs

Anwendung

Informationen zum Bauen und Starten des Projekts sowie den erwarteten Ausgaben sind in der README.md zu finden.