Haskell zur Constraint-Programmierung

HaL8

Alexander Bau*

2. Mai 2013

Wir benutzen eine Teilmenge von Haskell zur Spezifikation von Constraint-Systemen über Haskell-Datentypen. Ein Constraint-Compiler erzeugt zur Übersetzungszeit aus einem parametrisierten Constraint-System ein Haskell-Programm, welches zur Laufzeit für einen gegebenen Parameter eine aussagenlogische Formel generiert. Dabei werden rekursive Datentypen geeignet in ihrer Größe beschränkt, um eine endlich große Formel zu garantieren. Zu dieser Formel bestimmt ein SAT-Solver eine erfüllende Belegung, aus der schließlich eine erfüllende Belegung des ursprünglichen Constraint-Systems rekonstruiert wird. Es wird die Transformation von algebraischen Datentypen und Pattern-Matches auf diesen Typen diskutiert.

1 Einleitung

In der Constraint-Programmierung sind für Constraint-Systeme f der Form $f : \mathrm{U} \to \mathcal{B}$ solche Belegungen $u \in \mathrm{U}$ von Interesse, für die $f(u) = \textsc{True}$ gilt; wobei $\mathcal{B} = \{\textsc{False}, \textsc{True}\}$. Es existieren effiziente Suchverfahren für verschiedene Mengen U, um solche erfüllenden Belegungen zu finden [3, 1]. Der wesentliche Vorteil der Constraint-Programmierung besteht in der Trennung der Problemspezifikation, in Form eines Constraint-Systems, von jenen problemunabhängigen Suchverfahren.

Dieser Beitrag beschäftigt sich mit der Möglichkeit, eine Teilmenge von Haskell (Haskell–) als Spezifikationssprache für Constraint-Systeme über Haskell–-Daten zu nutzen. Dadurch kann Haskell als etablierte Programmiersprache in einem anderen Programmierparadigma genutzt werden. Spracheigenschaften wie algebraische Datentypen, rekursive Funktionen, Pattern-Matching und polymorphe Typen höherer Ordnung erlauben eine wesentlich ausdrucksstärkere Spezifikation von Constraint-Systemen gegenüber bisherigen Ansätzen, z.B. SAT und SMT.

Ein Constraint-Compiler übersetzt ein über P parametrisiertes Constraint-System in Haskell– vom Typ $f : \mathrm{P} \times \mathrm{U} \to \mathcal{B}$ in eine aussagenlogische Formel F, die von einem

*abau@imn.htwk-leipzig.de

SAT-Solver gelöst werden kann. Da die Erfüllbarkeit aussagenlogischer Formeln ein erfolgreich erforschtes Gebiet ist, existieren laufzeiteffiziente SAT-Solver wie Minisat[1]. Damit erlaubt unserer Ansatz sowohl eine natürliche Spezifikation des Constraint-Systems in einer etablierten Sprache, als auch dessen Lösung durch schnelle Suchverfahren.

Schließlich kann eine erfüllende Belegung $u \in \mathrm{U}$ für f aus einer erfüllenden Belegung von F rekonstruiert werden, so dass $f(p, u) = \textsc{True}$ gilt.

Beispiel 1 Das Beispiel zeigt ein Constraint-System $f : \textsc{Nat} \times \textsc{Nat} \to \mathcal{B}$ über Peano-Zahlen:

```
data Bool = False | True
data Nat  = Z | S Nat

eqNat x y = case x of
  Z    -> case y of Z    -> True
                    S y' -> False
  S x' -> case y of Z    -> False
                    S y' -> eqNat x' y'

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

f p u = eqNat p (add u (S (S Z)))
```

Der Übersetzungsprozess ist zweistufig:

1. zunächst wird aus einem Constraint-System $f : \mathrm{P} \times \mathrm{U} \to \mathcal{B}$ in Haskell– ohne Kenntnis des Parameter zur Übersetzungszeit eine Haskell-Funktion $f' : \mathrm{P} \to (\mathcal{B}^n \to \mathcal{B}), n \in \mathbb{N}$ erzeugt.
2. f' wird zur Laufzeit auf einen Parameter $p \in \mathrm{P}$ angewendet. Die resultierende aussagenlogische Formel F kann schließlich mit einem externen SAT-Solver gelöst werden. Eine erfüllende Belegung für f wird aus der erfüllenden Belegung für F rekonstruiert.

Ein wesentliches Problem bei der Übersetzung ist die Beschränkung auf einen endlichen Bereich von U. Da rekursive Haskell–-Datentypen unendlich große Mengen repräsentieren, müssen Rekursionen geeignet in ihrer Tiefe beschränkt werden. Wir bearbeiten dieses Problem unter Angabe eines Allokators für die Elemente U als Teil der Parametermenge P.

[1] `http://minisat.se`

2 Vergleich zu vorhandenen Arbeiten

Bibliotheken wie Gecode[2] oder Satchmo[3] ermöglichen die explizite Konstruktion von Finite-Domain-Constraint-Systemen. Dadurch ist die Beschränkung des Problems auf einen endlichen Bereich kein Problem des Werkzeugs, sondern der Spezifikation des Constraint-Systems selbst. Der Nachteil gegenüber dem hier vorgestellten Ansatz besteht in der Notwendigkeit, ein neues Werkzeug bzw. eine neue Sprache zu erlernen, um das Constraint-System zu spezifizieren.

Haskell wird in anderen Bereichen bereits erfolgreich als Spezifikationssprache eingesetzt, so z.B. für digitale Schaltungen [2, 4].

3 Datenübersetzung

Zur Übersetzung eines Haskell– -Constraint-Systems f in eine aussagenlogische Formel ist es notwendig, die Daten, auf denen f operiert, als Menge von Booleschen Variablen auszudrücken.

Definition 1 Eine Haskell–-Datendeklaration hat die folgende Form:

$$\begin{aligned} & & c_1\ t_{11}\ t_{12}\ \ldots\ t_{1n_1} \\ c\ v_1\ \ldots v_m = & & \ldots \\ & & c_k\ t_{k1}\ t_{k2}\ \ldots\ t_{kn_k} \end{aligned}$$

wobei

- c einen m-stelligen Typkonstruktor benennt
- $v_i, 0 \leq i \leq m$ die Typvariablen von c benennt
- $c_i, 1 \leq i \leq k$ die Konstruktoren von c benennt
- $n_i \geq 0, 1 \leq i \leq k$ gleich der Anzahl der Konstruktorargumente von c_i ist
- $t_{ij}, 1 \leq i \leq k, 0 \leq j \leq n_i$ der Typ des j-ten Arguments des i-ten Konstruktors ist

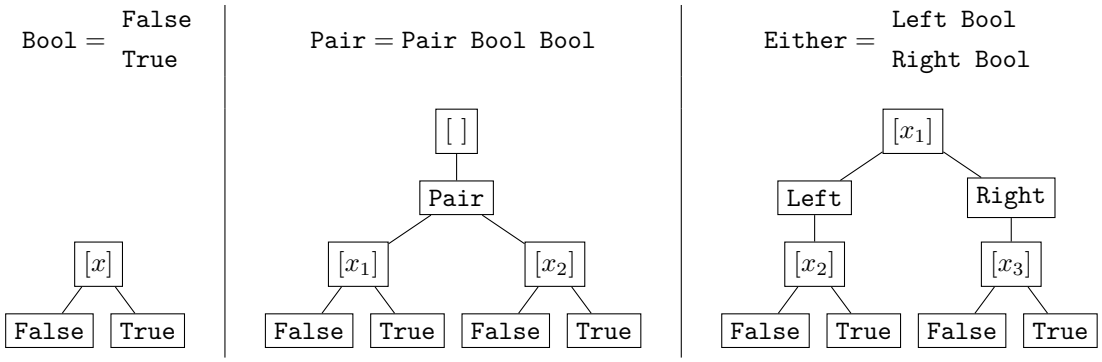
Dabei gilt, dass die in den Konstruktorargumenten vorkommenden Variablen durch die Typvariablen $v_1\ \ldots v_m$ gebunden sein müssen.

Bei der Übersetzung von Haskell–-Daten eines Types, dessen Typdeklaration aus k Konstruktoren besteht, werden die Konstruktoren mit $\left\lceil \log_2^k \right\rceil$ Booleschen Variablen x_i entsprechend ihrer Reihenfolge in der Typdeklaration durch eine Binärkodierung dieser Variablen nummeriert. Die Booleschen Variablen bezeichnen wir als *Flags*. Jedes Konstruktorargument wird rekursiv mit diesem Schema übersetzt.

Abbildung 1 illustriert einige beispielhafte Übersetzungen üblicher Datentypen.

[2] `http://www.gecode.org`
[3] `https://github.com/jwaldmann/satchmo`

Abbildung 1: Einige Datenübersetzungen



4 Programmübersetzung

In Haskell– können durch Pattern-Matching die Konstruktorargumente eines Datums extrahiert werden. Da zur Übersetzungszeit der äußerste Konstruktor des untersuchten Ausdrucks möglicherweise nicht bekannt ist, muss die Programmübersetzung für alle Zweige des Pattern-Matches stattfinden.

Definition 2 Wir nehmen einem Pattern-Match der Form

$$r = \texttt{case}\ \ e\ \ \texttt{of}\ \begin{cases} c_1\ v_{11}\ v_{12}\ \ldots\ v_{1n_1} & \to b_1 \\ \ldots & \\ c_k\ v_{k1}\ v_{k2}\ \ldots\ v_{kn_k} & \to b_k \end{cases}$$

an, wobei

- die Konstruktoren $c_1, \ldots, c_k$ vollständig in der Reihenfolge ihrer Deklaration vorkommen
- die Konstruktorargumente durch Typvariablen t_{ij} gebunden werden
- $b_1, \ldots, b_k$ die Zweige des Pattern-Matches bezeichnen und vom selben Typ sind

Abhängig von den Flags f der Übersetzung von e wird die i-te, im Resultat r vorkommende, Boolesche Variable r_i durch eine Konjunktion vom k Implikationen kodiert:

$$r_i \leftrightarrow (f_{10} \stackrel{?}{=} 0 \to b_{1i}) \wedge (f_{10} \stackrel{?}{=} 1 \to b_{2i}) \wedge \cdots \wedge (f_{10} \stackrel{?}{=} k-1 \to b_{ki})$$

, wobei

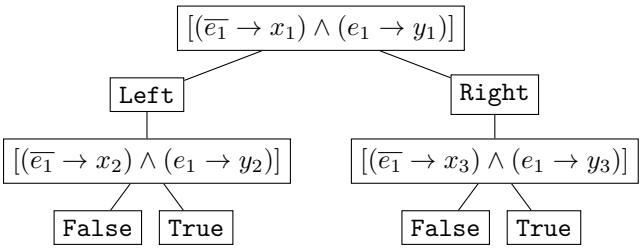
- f_{10} die Interpretation der Flags f der Übersetzung von e im Dezimalsystem darstellt. $f_{10} \stackrel{?}{=} j$ gilt also genau dann, wenn die Flags f den $j+1$-ten Konstruktoren indexieren.

- b_{ji} die i-te Variable im j-ten Zweig des Pattern Matches bezeichnet (s. Nummerierung der Variablen in Abb. 1)

Beispiel 2 Wir betrachten folgenden Pattern-Match:

$$r = \texttt{case}\ \ e\ \ \texttt{of}\ \begin{cases}\texttt{False} & \to x \\ \texttt{True} & \to y\end{cases}$$

, wobei e vom Typ `Bool` und x, y vom Typ `Either` sind. Entsprechend der Nummerierung der Flags in Abb. 1 sei e_i (bzw. x_i und y_i) das i-te Flag der Übersetzung von e (bzw. x und y). Die Übersetzung des Resultates r hat nach Definition 2 schließlich folgende Struktur:



5 Ausblick

Zur effizienten Übersetzung von Haskell– -Programmen sind Optimierungen nötig. Diese basieren auf der Annahme, dass Formeln mit weniger Klauseln und Variablen in kürzerer Zeit vom Solver gelöst werden können. Beispielsweise können Haskell– -Daten kompakter kodiert werden, in dem Konstruktorargumente überlappt werden. Ebenso ist die Übersetzung von Ausdrücken, die ausschließlich von bekannten Daten abhängen, nicht sinnvoll, da diese ohne Hilfe des SAT-Solvers ausgewertet werden können. In einem Constraint-Compiler ist also die Unterscheidung zwischen bekannten Daten und unbekannten Daten wesentlich. Weiterhin ist die Speicherung und Wiederverwendung von bereits generierten Teilformeln sinnvoll.

Literatur

[1] Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ICFP*, pages 174–184, 1998.

[3] Petra Hofstedt and Armin Wolf. Einführung in die constraint-programmierung, 2007.

[4] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in hawk. In *ICCL*, pages 90–101, 1998.