Waitomo

Compilerbaupraktikum Wintersemester 2006/2007

Stefan Wehr

24. Oktober 2006

1 Einleitung

Quellsprache für das Compilerbaupraktikum ist Waitomo, ein Java-ähnliche Sprache mit Unterstützung für Typklassen im Stile von Haskell.[1] Waitomo fügt einige neuen Konzepte zu Java hinzu bzw. generalisiert bestehende Features, welche anhand von Beispielen im Abschnitt 2 vorgestellt und erklärt werden. Ausgangspunkt ist dabei die Version 5 von Java, d.h. Waitomo enthält insbesondere auch generische Klassen und Methoden. In Abschnitt 3 wird dann die abstrakte Syntax von Waitomo vorgestellt.

2 Erweiterungen gegenüber Java

2.1 Externe Methoden

Waitomo erlaubt es, Methoden nachträglich einer Klasse hinzuzufügen. Soll etwa eine Methode `deleteWhitespace` zum Löschen aller Whitespacezeichen der Klasse `String` hinzugefügt werden, so kann dies mittels folgender Deklaration vorgenommen werden:

```
extension String {
  String deleteWhitespace() { ... }
}
```

Nun kann im Programm etwa `‘‘Hello World!’’.deleteWhitespace()` geschrieben werden, obwohl die Klasse `String` selbst gar keine Methode mit Namen `deleteWhitespace` definiert.

2.2 Mächtigere Interfaces

Interfaces in Waitomo sind mächtiger als in Java. Die folgenden Abschnitte beschreiben die Unterschiede.

[1]Diese Einführung setzt keine Kenntnisse von Haskell voraus, allerdings wird von einer gewissen Vertrautheit mit Java ausgegangen.

2.2.1 Expliziter Bezug auf die implementierende Klasse

Eine Interfacedefinition kann explizit Bezug nehmen auf die Klasse, welches das Interface implementiert. Ein Beispiel:

```
interface Comparable [This] {
  class This {
    int compareTo(This other);
  }
}
```

Das Beispiel führt für die implementierende Klasse den Namen `This` ein. Der Rumpf der Interfacedefinition spezifiziert dann, dass die Klasse, welche später einmal für `This` eingesetzt wird, eine `compareTo` Methode bereitstellen muss.

Das sieht zunächst komplizierter aus als in Java. Dagegen lässt sich zum einen einwenden, dass die Java Notation als syntaktischer Zucker für die Waitomo Syntax gesehen werden kann (siehe Übung). Zum anderen kann man dank der expliziten Parametrisierung über die implementierende Klasse auch Interfaces definieren, die von mehreren Klassen im Verbund implementiert werden müssen.

Das folgende Beispiel demonstriert anhand des Subject-Observer Patterns, warum das sinnvoll ist. Am Subject-Observer Pattern sind bekanntlich zwei Klassen beteiligt:

- `Subject`: Repräsentiert Daten die sich ändern können. Die Methode `notify` wird bei eintretender Änderung aufgerufen. Am `Subject` können sich `Observer` mittels der Methode `register` registrieren, um bei Änderungen benachrichtigt zu werden.
- `Observer`: Wartet auf Änderungen des `Subject`s. Bei einer Änderung wird die Methode `update` aufgerufen.

Die folgende Interfacedefinition setzt diese Anforderungen um:

```
interface SubjectObserver [Subject,Observer] {
  class Subject {
    void register(Observer o);
    void notify();
  }
  class Observer {
    void update(Subject s);
  }
}
```

2.2.2 Existenzielle Typen

In Waitomo wird der Typ, über den lediglich bekannt ist, dass er das Interface `Comparable` implementiert, wie folgt aufgeschrieben:

$$\texttt{\textbf{exists} X \textbf{where} X \textbf{implements} Comparable . X}$$

Das sieht kompliziert aus, allerdings kann der in Java übliche Typ (nämlich einfach `Comparable`) als syntaktischer Zucker gewonnen werden (siehe Übung). Die Mächtigkeit von existenziellen Typen wird erst deutlich, wenn wir als Beispiel den Typ eines beliebigen `Subjects` hinschreiben:

$$\texttt{\textbf{exists} S,O \textbf{where} S,O \textbf{implements} SubjectObserver . S}$$

2.2.3 Externe Implementierungsdeklarationen

In Waitomo können Interfaces auch außerhalb von Klassen implementiert werden. Seien als Beispiel folgende Klassen gegeben:

```
class Complex {
  ...
  int compareTo(Complex other) {
    // Ordnung für komplexe Zahlen
  }
}

class MyData {
  ...
  List<DataObserver> os = new ArrayList<DataObserver>();
  void register(DataObserver o) {
    os.add(o);
  }
  void notify() {
    for(DataObserver o : os) {
      o.update(this);
    }
  }
}

class DataObserver {
  void update(MyData data) {
    System.out.println("data updated");
  }
}
```

Für die beiden Interfaces des vorigen Abschnitts kann man nun *im Nachhinein* Implementierungsdeklarationen angeben:

```
implementation Complex of Comparable
implementation MyData, DataObserver of SubjectObserver
```

Kombiniert man nun externe Methoden und externe Implementierungsdeklarationen, so kann man Code schreiben, der das `Comparable` Interface für Listen implementiert, vorausgesetzt die Listenelemente implementieren `Comparable`. (Den nun folgenden Code kann man auch kürzer schreiben, indem man den Code aus der **`extension`** Deklaration in die **`implementation`** Deklaration legt; siehe Übung.)

```
extension AbstractList<X> where X implements! Comparable {
  int compareTo(AbstractList<X> other) {
    // compare element-wise
  }
}

implementation<X> AbstractList<X> of Comparable
  where X implements! Comparable
```

Der Teil nach dem Schlüsselwort **`where`** in den beiden vorangehenden Deklarationen ist ein *Constraint*, welcher für `X` einsetzbare Typen auf solche Typen beschränkt, die `Comparable` implementieren. Das Ausrufezeichen nach dem Schlüsselwort **`implements`** spezifiziert, dass die Implementierung explizit sein muss; die genaue Definition von "explizit" ist an dieser Stelle unwichtig.

2.3 Statische Methoden in Interfaces

In Java ist es nicht möglich, über Konstruktoren von Klassen zu abstrahieren; man braucht dazu das Factory Pattern. Waitomo hingegen unterstützt Abstraktion über Konstruktoren mittels statischer Methoden in Interfaces. Zum Beispiel lassen sich viele Objekte aus einer Stringrepräsentation erzeugen. Das folgende Interface abstrahiert über dieses Pattern:

```
interface Parseable [This] {
  static This parse(String s);
}
```

Das Interface `Parseable` kann man nun etwa für `Integer`s implementieren:

```
implementation Parseable [Integer] {
  static Integer parse(String s) {
    return new Integer(s);
  }
}
```

3 Abstrakte Syntax

Die abstrakte Syntax von Waitomo ist in Figure 1 dargestellt. Die Notation $\overline{\xi}$ für ein beliebiges Syntaxelement ξ steht für $\xi_1, \ldots, \xi_n$. Der Typ $@T$ ist ein sogenannter *exakter*

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{def}}\ e \\
\mathit{def} & ::= & \mathit{cdef} \mid \mathit{idef} \mid \mathit{edef} \mid \mathit{impl} \\
\mathit{cdef} & ::= & \texttt{class}\ C\langle\overline{X}\rangle\ \texttt{extends}\ N\ \texttt{where}\ \overline{Q}\,\{\,\overline{T\,f}\ \overline{\mathit{mdef}}\,\} \\
\mathit{idef} & ::= & \texttt{interface}\ I\langle\overline{X}\rangle\,[\overline{X}]\ \texttt{where}\ \overline{Q}\,\{\,\overline{\mathit{smdec}}\ \overline{\mathit{mdecs}}\,\} \\
\mathit{edef} & ::= & \texttt{extension}\ C\langle\overline{X}\rangle\ \texttt{where}\ \overline{Q}\,\{\,\overline{\mathit{smdef}}\,\} \\
\mathit{impl} & ::= & \texttt{implementation}\langle\overline{X}\rangle\ \overline{N}\ \texttt{of}\ K\ \texttt{where}\ \overline{Q}\ \{\,\overline{\mathit{smdef}}\,\} \\
\mathit{mdecs} & ::= & \texttt{class}\ X\,\{\,\overline{\mathit{mdec}}\,\} \\
\mathit{mdec} & ::= & \langle\overline{X}\rangle T\ m(\overline{T})\ \texttt{where}\,\overline{Q} \\
\mathit{mdef} & ::= & \langle\overline{X}\rangle T\ m(\overline{T\,x})\ \texttt{where}\,\overline{Q}\,\{e\} \\
\mathit{smdec} & ::= & \texttt{static}\,\mathit{mdec} \\
\mathit{smdef} & ::= & \texttt{static}\,\mathit{mdef} \\
e & ::= & \texttt{new}\,N \mid x \mid \texttt{null} \mid e.f \mid e.f = e \mid \texttt{let}\,x = e\,\texttt{in}\,e \mid e.m\langle\overline{T}\rangle(\overline{e}) \\
 & & \mid K[\overline{T}].m\langle\overline{X}\rangle(\overline{e}) \mid (T)\,e \\
T & ::= & X \mid N \mid @T \mid \exists\overline{X}\,\texttt{where}\,\overline{Q}\,.\,T \\
N & ::= & C\langle\overline{T}\rangle \mid \mathit{Object} \\
K & ::= & I\langle\overline{T}\rangle \\
Q & ::= & \overline{T}\,\texttt{implements}^{\varphi}\,K \mid X\,\texttt{extends}\,N \\
\varphi & ::= & \pi \mid \mu
\end{array}
$$

Abbildung 1: Abstrakte Syntax für Waitomo

Typ, d.h. T' ist mit $@T$ nur dann kompatibel falls $T = T'$. Die Annotation φ in einem Constraint $\overline{T}\,\texttt{implements}^{\varphi}\,K$ gibt an, ob $\overline{T}$ das Interface K explizit (μ, in Abschnitt 2 haben wir dafür das Ausrufezeichen verwendet) oder implizit (π) implementieren müssen.

Die Implementierung der abstrakten Syntax ist im Modul `Syntax.Dst` zu finden. Allerdings wird dort eine erweiterte Syntax verwendet, die insbesondere auch eine Ebene für Statements unterstützt. Das Kürzel `Dst` steht dabei für "desugared syntax tree"; die Implementierung verwendet den Begriff "abstrakter Syntaxbaum" für den direkt nach dem Parsen erzeugten Syntaxbaum, welcher noch syntaktischen Zucker und unaufgelöste Bezeichner enthält.