Waitomo, eine interface-orientierte Programmiersprache

Peter Thiemann Stefan Wehr
Universität Freiburg
{thiemann,wehr}@informatik.uni-freiburg.de

Zusammenfassung

Waitomo ist eine von Java abgeleitete experimentelle Programmiersprache. Das Typsystem von Waitomo garantiert größere Typsicherheit als Java, da Casts kein Bestandteil der Sprache sind. Stattdessen wird Flexibilität durch parametrische Polymorphie, durch Selbst- und Vereinigungstypen, sowie durch eine neue Sichtweise auf Interfaces erzielt.

Schlüsselwörter: Generizität, Interfaces, Programmiersprachendesign, Typsysteme.

1 Motivation

Objekt-orientierte Programmiersprachen wie Java und C♯ stellen eine Cast–Operation zur Verfügung, die zur Laufzeit den Typ eines Objekts anpassen kann. Damit sind Casts eine ständige Quelle für Laufzeitfehler. Die Programmiersprache Waitomo kommt ohne Casts aus, verfügt aber über viele dem objektorientierten Programmierer vertrauten Konzepte. Die Benutzung von Java oder C♯ Bibliotheken aus Waitomo ist möglich.

Parametrische Typen [6], oft auch generische Typen genannt, sind ein wichtiger Bestandteil von Waitomo, da sie viele Casts überflüssig machen, wie die Einführung von parametrischen Typen in Java 1.5 [3] zeigt. Funktionale Programmiersprachen wie Standard ML [4] oder Haskell [5] zeigen darüberhinaus, dass ohne Casts aber mit parametrischen Typen die Implementierung großer System durchaus möglich ist.

Interfaces sind auch eine Ursache für Casts in Java:

- Es ist kompliziert, in einem Interface zu formulieren, dass Empfänger und Argument einer Methode vom selben Typ sind. Daher wird die Typidentität oft durch Casts zur Laufzeit sichergestellt.
- Casts sind der einzige Weg, um von einem Interfacetypen an den konkreten Typ des Objekts zu kommen, welches das Interface implementiert.
- Es ist umständlich, Interfaces zu kombinieren. Daher wird oft `Object` als der Typ der Kombination mehrerer Interfaces deklariert und Casts auf das entsprechende Interface werden eingefügt.
- Interfaces können keine Beziehungen zwischen mehreren Typen ausdrücken (z.B. zwischen Kanten und Knoten eines Graphens). Stehen nun die Argumente einer Methode in einer solchen Beziehung, muss sie entweder konkrete Typen oder `Object` und Casts benutzen.

2 Sprachfeatures

Die im vorangehenden Abschnitt erwähnten Gründen führten zu einem Sprachdesign, das sich von Java 1.5 im wesentlichen in drei Punkten unterscheidet:

- Wir haben Java Interfaces durch eine verallgemeinerte Form von Interfaces ersetzt, die sich am Design von Haskells Typklassen [7] orientiert.
- Wir haben Casts vollständig aus der Sprache entfernt und dafür Selbst- und Vereinigungstypen [2, 1] eingeführt.
- Wir haben die Kopplung zwischen Vererbung und Subtypbeziehung aufgehoben: Eine abgeleitete Klasse ist nicht mehr automatisch ein Subtyp der Oberklasse.

Der letzte Punkt mag als eine starke Einschränkung erscheinen, allerdings lässt sich zum einen mittels Interfaces eine an der Klassenhierarchie orientierten Subtyphierarchie aufbauen, zum anderen ersetzen Vereinigungstypen die durch Vererbung implizierten Subtypen in vielen Fällen.

3 Beispiele

3.1 Binäre Methoden

Vor Java 1.5 war es nicht möglich, ein typsicheres Interface zum Vergleich zweier Objekte zu schreiben. Seit der Einführung von parametrischen Typen in Java 1.5 ist dies zwar möglich, allerdings wird die Verwendung eines solchen Interfaces durch rekursive Schranken unnötig kompliziert. Das folgende Beispiel zeigt das entsprechende Waitomo Interface:

```
interface Comparable {
  int compareTo(This that);
}
```

Die Typvariable `This` steht dabei für die implementierende Klasse. Damit ist sichergestellt, dass der Typ des Empfängers und des Arguments eines Aufrufs von `compareTo` übereinstimmen. Die Implementierung eines Interfaces kann in Waitomo auch außerhalb von Klassen erfolgen, wie dieses Beispiel zeigt:

```
Integer implements Comparable {
  int compareTo(Integer that) {
    return this - that;
  }
}
```

Waitomo unterscheidet zwischen explizit und implizit implementierten Interfaces; eine explizite Implementierung liegt nur für Klassen und für speziell gekennzeichnete Typvariablen vor. Methoden mit `This` in der Typsignatur können nur auf Objekten aufgerufen werden, deren statischer Typ das entsprechende Interface explizit implementiert. Dies ist nötig, um statische Typsicherheit trotz kovariantem Subtyping auf Methodenargumenten zu garantieren.

3.2 Statische Interfacemethoden

Waitomo unterstützt neben herkömmlichen Methoden auch statische Methoden in Interfaces. Damit kann man Konstruktormethoden in Interfaces spezifizieren, was das Factory Patterns häufig unnötig macht.

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

Über die Notation `Parseable@T.parse` wird die Methode `parse` des Interfaces `Parseable` aus der Implementierung für den Typ `T` ausgewählt:

```
<T implements Parseable> T foo(String s) {
  return Parseable@T.parse(s);
}
```

3.3 Heterogene Sammlungen

Interfaces in Waitomo sind keine Typen sondern Constraints auf Typparametern. Demnach ist etwa der Typ `Collection<I>` für ein Interface `I` nicht gültig.

Um in Waitomo auch heterogone Datenstrukturen zu unterstützen, gibt es existenzielle Typen der Form $\exists$`X implements I.X`, geschrieben `{I}`. Damit ergibt sich die Möglichkeit, beliebige Typen durch mehrer Interfaces zu beschränken, ein Feature das in Java nur für Typparameter zur Verfügung steht. Eine Sammlung vom Typ `Collection<{Parseable,Comparable}>` enthälte demnach Objekte, die sowohl `Parseable` als auch `Comparable` implementieren.

3.4 Vereinigungstypen und das Visitor Pattern

Im folgenden Beispiel werden Vereinigungstypen benutzt, um eine Baumstruktur zu definieren. Die kanonische Java–Implementierung würde dazu das Composite Pattern benutzen, welches die Kopplung von Vererbung und Subtyping voraussetzt.

```
type Tree = Leaf|Node
class Leaf { ... }
class Node { Tree left, right; ... }
```

Da die Implementierung von Interfaces in Waitomo außerhalb von Klassen erfolgen darf, können nachträglich Operationen zu einer Datenstruktur hinzugefügt werden, ohne das Visitor Pattern zu benutzen.

```
interface Count {
  static int count(This x)
}
Leaf implements Count {
  static int count(Leaf l) { return 0; }
}
Node implements Count {
  static int count(Node n) {
    return Count@Tree.count(n.left) +
           Count@Tree.count(n.right) + 1;
  }
}
```

Der Aufruf `Count@Tree.count` ist gültig, da mit `Leaf implements Count` und `Node implements Count` auch `(Leaf|Node) implements Count` gilt. Zu beachten ist allerdings, dass diese Implikation nur gültig ist, da `This` im Ergebnistyp von `count` überhaupt nicht und im Argumenttyp nur einmal auftritt.

4 Fazit

Waitomo ist eine an Java angelehnte Programmiersprache, die ohne Casts auskommt und Subtyping von Vererbung abkoppelt. Flexibilität wird durch parametrische Polymorphie, eine Erweiterung des Interfacekonzepts sowie durch Selbst- und Vereinigungstypen erreicht. Ein korrektes Typsystem für eine Kernsprache von Waitomo sowie eine prototypenhafte Implementierung ist vorhanden.

Literatur

[1] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[2] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. pages 27–51, 1995.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[4] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[5] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[6] C. Strachey. Fundamental concepts of programming languages. In *NATO Summer School in Programming*, Copenhagen, 1967.

[7] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.