Skript zur Vorlesung

Fortgeschrittene Programmierung

SS 2014

Prof. Dr. Michael Hanus
Priv.Doz. Dr. Frank Huch

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Version vom 15. April 2014

Vorwort

In dieser Vorlesung werden forgeschrittene Programmierkonzepte, die über die in den ersten Studiensemestern erlernte Programmierung hinausgehen, vorgestellt. Dabei wird anhand verschiedener Programmiersprachen der Umgang mit den Konzepten der wichtigsten Programmierparadigmen vermittelt. Moderne funktionale Programmierungstechniken werden am Beispiel der Sprache Haskell gezeigt. Logische und Constraint-orientierte Programmierung wird in der Sprache Prolog vermittelt. Konzepte zur nebenläufigen und verteilten Programmierung werden mit der Sprache Java vorgestellt und geübt.

Dieses Skript basiert auf der Vorlesung vom SS 2012 und ist eine überarbeitete Fassung einer Mitschrift, die ursprünglich von Nick Prühs im SS 2009 in LaTeX gesetzt wurde. Ich danke Nick Prühs für die erste LaTeX-Vorlage und Björn Peemöller und Lars Noelle für Korrekturhinweise.

Noch eine wichtige Anmerkung: Dieses Skript soll nur einen Überblick über das geben, was in der Vorlesung gemacht wird. Es ersetzt nicht die Teilnahme an der Vorlesung, die zum Verständnis der Konzepte und Techniken der fortgeschrittenen Programmierung wichtig ist. Ebenso wird für ein vertieftes Selbststudium empfohlen, sich die in der Vorlesung angegebenen Lehrbücher und Verweise anzuschauen.

Kiel, Juni 2013 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per E-Mail mitgeteilt werden.

Inhaltsverzeichnis

1 Java Generics

1.1 Einführung

Seit der Version 5.0 (2004 veröffentlicht) bietet Java die Möglichkeit der *generischen Programmierung*: Klassen und Methoden können mit Typen parametrisiert werden. Hierdurch eröffnen sich ähnliche Möglichkeiten wie mit Templates in C++. Als einfaches Beispiel betrachten wir eine sehr einfache Containerklasse, welche keinen oder einen Wert speichern kann. In Java könnte das z.B. wie folgt definiert werden:

```
public class Optional {
  private Object value;
  private boolean present;

  public Optional() {
    present = false;
  }

  public Optional(Object v) {
    value = v;
    present = true;
  }

  public boolean isPresent() {
    return present;
  }

  public Object get() {
    if (present) {
      return value;
    }
    throw new NoSuchElementException();
  }
}
```

Wir nutzen dabei aus dass `NoSuchElementException` von `RuntimeException` abgeleitet ist, d.h. es ist eine sogenannte *unchecked exception* und muss nicht deklariert werden.
Die Verwendung der Klasse könnte dann wie folgt aussehen:

```
Optional mv = new Optional(new Integer(42));
...
if (mv.isPresent()) {
  Integer n = (Integer) mv.get();
}
```

Der Zugriff auf das gespeicherte Objekt der Containerklasse erfordert also jedes Mal explizite Typumwandlungen (type casts). Es ist *keine Typsicherheit* gegeben: Im Falle eines falschen Casts tritt eine `ClassCastException` zur Laufzeit auf.

Beachte: Die Definition der Klasse `Optional` ist völlig unabhängig vom Typ des gespeicherten Wertes; der Typ von `value` ist `Object`. Wir können also beliebige Typen erlauben und in der Definition der Klasse von diesen abstrahieren: Das Übergeben des Typs als Parameter nennt man *parametrisierter Polymorphismus*.

Syntaktisch markieren wir den Typparameter durch spitze Klammern und benutzen den Parameternamen an Stelle von `Object`. Die Klassendefinition lautet dann wie folgt:

```
public class Optional<T> {
  private T value;
  private boolean present;

  public Optional() {
    present = false;
  }

  public Optional(T v) {
    value = v;
    present = true;
  }

  public boolean isPresent() {
    return present;
  }

  public T get() {
    if (present) {
      return value;
    }
    throw new NoSuchElementException();
  }
}
```

Die Verwendung der Klasse sieht dann so aus:

```
Optional<Integer> mv = new Optional<Integer>(new Integer(42));
```

```
...
if (mv.isPresent()) {
  Integer n = mv.get();
}
```

Es sind also keine expliziten Typcasts mehr erforderlich, und ein Ausdruck wie

```
mv = new Optional<Integer>(mv);
```

liefert nun bereits eine Typfehlermeldung zur Compilezeit.
Natürlich sind auch mehrere Typparameter erlaubt:

```
public class Pair<A, B> {
  private A first;
  private B second;

  public Pair(A first, B second) {
    this.first = first;
    this.second = second;
  }

  public A first()  { return first ; }
  public B second() { return second; }
}
```

1.2 Zusammenspiel mit Vererbung

Es ist auch möglich, Typparameter so einzuschränken, dass dafür nur Klassen eingesetzt werden können, die bestimmte Methoden zur Verfügung stellen. Als Beispiel wollen wir unsere Klasse `Optional` so erweitern, dass sie auch das Interface `Comparable` implementiert:

```
public class Optional<T extends Comparable<T>>
  implements Comparable<Optional<T>> {
  ...
  public int compareTo(Optional<T> o) {
    if (present) {
      return other.isPresent() ? value.compareTo(other.get()) : 1;
    } else {
      return other.isPresent() ? -1 : 0;
    }
  }
}
```

Hierbei wird sowohl für Interfaces als auch für echte Vererbung das Schlüsselwort `extends` verwendet. Wir können auch mehrere Einschränkungen an Typvariablen aufzählen. Für drei Typparameter (`T`, `S` und `U`)sieht das z.B. wie folgt aus:

```
<T extends A<T>, S, U extends B<T,S>>
```

Hier muss `T` die Methoden von `A<T>` und `U` die Methoden von `B<T,S>` zur Verfügung stellen. `A` und `B` müssen hierbei Klassen oder Interfaces sein, also insbesondere keine Typvariablen.

1.3 Wildcards

Die Klasse `Integer` ist Unterklasse der Klasse `Number`. Somit sollte doch auch der folgende Code möglich sein:

```
Optional<Integer> mi = new Optional<Integer>(new Integer(42));
Optional<Number> mn = mi;
```

Das Typsystem erlaubt dies aber nicht, weil dies festlegt, dass `Optional<Integer>` **kein** Untertyp von `Optional<Number>` ist! Das mögliche Problem wird deutlich, wenn wir zur Klasse `Optional` noch eine Setter-Methode hinzufügen:

```
public void set(T v) {
  present = true;
  value = v;
}
```

Dann wäre auch Folgendes möglich:

```
mn.set(new Float(42));
Integer i = mi.get();
```

Da `mi` und `mn` die gleichen Objekte bezeichnen, würde dieser Code bei der Ausführung zu einem Typfehler führen. Aus diesem Grund erlaubt das Typsystem den obigen Code nicht.
Dennoch benötigt man manchmal einen Obertyp für polymorphe Klassen, also einen Typ, der alle anderen Typen umfasst: So beschreibt

```
Optional<?> mx = mi;
```

einen `Optional` von unbekanntem Typ.
Hierbei wird "`?`" auch als *Wildcard*-Typ bezeichnet. `Optional<?>` repräsentiert alle anderen `Optional`-Instantiierungen, z.B. `Optional<Integer>` oder `Optional<Optional<Object>>`.
Damit können aber nur Methoden verwendet werden, die für jeden Typ passen, also z.B.

```
Object o = mx.get();
```

Können wir auch die Methode `set` für `mx` aufrufen? Hierzu benötigen wir einen Wert, der zu allen Typen gehört: `null`.

```
mx.set(null);
```

Andere `set`-Aufrufe sind nicht möglich.

Der Wildcard-Typ "`?`" ist aber leider oft nicht ausreichend, z.B. wenn man in einer Collection auch verschiedene Untertypen speichert, wie GUI-Elemente. Dann kann man sog. *beschränkte Wildcards* (*bounded wildcards*) verwenden:

- `<? extends A>` steht für *alle* Untertypen des Typs `A` *(Kovarianz)*
- `<? super A>` steht für *alle* Obertypen von `A` *(Kontravarianz)*

Es folgen einige Beispiele.

Ausdruck	erlaubt?	Begründung
`Optional<? extends Number> mn = mi;`		
`Integer i = mn.get();`	nein!	`?` könnte auch `Float` sein
`Number n = mn.get();`	ja	
`mn.set(n);`	nein!	`?` könnte spezieller als `Number` sein
`mn.set(new Integer(42));`	nein!	`?` könnte auch `Float` sein
`mn.set(null);`	ja	
`Optional<? super Integer> mx = mi;`		
`Integer i = mx.get();`	nein!	`?` könnte auch `Number` oder `Object` sein
`Number n = mx.get();`	nein!	`?` könnte auch `Object` sein
`Object o = mx.get();`	ja	
`mx.set(o);`	nein!	`?` könnte auch `Number` sein
`mx.set(n);`	nein!	`n` könnte auch vom Typ `Float` sein
`mx.set(i);`	ja	

Abbildung 1.1: Ausdrücke mit Wildcards

Wir können also aus einem `Optional<? extends A>` Objekte vom Typ `A` herausholen, und in einen `Optional<? super A>` Objekte vom Typ `A` hereinstecken.

1.4 Typinferenz

Bei der Initialisierung einer Variablen mit einem generischen Typparameter fällt auf, dass man den Typparameter zwei Mal angeben muss:

```
Optional<Integer>       o = new Optional<Integer>(new Integer(42));
```

```
List<Optional<Integer>> l = new ArrayList<Optional<Integer>>(o);
```

Seit Version 7 ist der Java-Compiler jedoch in der Lage, bei einer Objektinitialisierung mittels `new` in den meisten Fällen den generischen Typ zu berechnen. Daher muss der Typ dann nur noch bei der Variablendeklaration angegeben werden, beim Aufruf von `new` wird der Typ durch den "Diamant-Operator"`<>` berechnet:

```
Optional<Integer> mv = new Optional<>(new Integer(42));
List<Optional<Integer>> l = new ArrayList<>(mv);
```

Dabei kann nur die gesamte Typangabe in den spitzen Klammern weggelassen werden, Angaben wie `<Optional<>>` sind nicht zulässig. Sofern der Compiler den Typ jedoch nicht inferieren kann, muss dieser nach wie vor manuell angegeben werden.

Neben der Schreiberleichterung erlaubt der Diamant-Operator es nun auch, generische Singleton-Objekte zu erstellen, was vorher nicht möglich war:

```
Optional<?> empty = new Optional<>();
```

Dies ist insbesondere sinnvoll um von unveränderlichen Objekten nur eine Instanz im Speicher vorzuhalten.

Literaturverzeichnis

[1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[3] J. Corbin and M. Bidoit. A rehabilitation of robinson's unification algorithm. In *Proc. IFIP '83*, pages 909–914. North-Holland, 1983.

[4] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[5] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[6] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[8] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Abbildungsverzeichnis

Index