y

Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 10.12.08: Signaturen und Eigenschaften

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
  - Abstrakte Datentypen
  - Signaturen & Axiome
  - Korrektheit von Programmen
  - Bäume und Mengen
  - Zustand
- Teil III: Beispiele, Anwendungen, Ausblicke

Abstrakte Datentypen

- Letzte Vorlesung: Abstrakte Datentypen
- Typ plus Operationen
  - In Haskell: Module
- Heute: Signaturen und Eigenschaften

Signaturen

Definition: Die Signatur eines abstrakten Datentyps besteht aus den Typen, und der Signatur der Funktionen darauf.

- Obs: Keine direkte Repräsentation in Haskell
- Signatur: Typ eines Moduls

Der Speicher: Signatur

- Ein Speicher (Store, FiniteMap, State)
- Typen: der eigentliche Speicher S, Adressen a, Werte b
- Operationen:
  - leerer Speicher: S
  - in Speicher an eine Stelle einen Wert schreiben: $S \to a \to b \to S$
  - aus Speicher an einer Stelle einen Wert lesen: $S \to a \rightharpoonup b$ (partiell)

Der Speicher: Signatur

- Adressen und Werte sind Parameter

```
type Store a b
```

- Leerer Speicher:

```
empty :: Store a b
```

- In Speicher an eine Stelle einen Wert schreiben:

```
upd :: Store a b-> a -> b-> Store a b
```

- Aus Speicher an einer Stelle einen Wert lesen:

```
get  :: Store a b-> a -> Maybe b
```


Signatur und Eigenschaften

- Signatur genug, um ADT typkorrekt zu benutzen
  - Insbesondere Anwendbarkeit und Reihenfolge
- Signatur nicht genug, um Bedeutung (Semantik) zu beschreiben
  - Beispiel Speicher: Was wird gelesen? Wie verhält sich der Speicher?

Beschreibung von Eigenschaften: Axiome

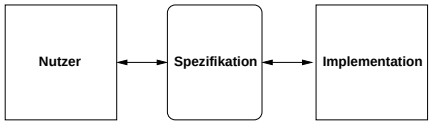
- Axiome sind Prädikate über den Operationen der Signatur
  - Elementare Prädikate P
  - Gleichheit $s == t$
  - Bedingte Prädikate: $A \Longrightarrow B$
- Beobachtbare Typen: interne Struktur bekannt
  - Vordefinierte Typen (Zahlen, Zeichen, Listen), algebraische Datentypen
- Abstrakte Typen: interne Struktur unbekannt
  - Gleichheit (wenn definiert)

Beispiel: Speicher

- Beobachtbar: Adressen und Werte, abstrakt: Speicher
- Axiome für das Lesen:
  - Lesen aus dem leeren Speicher undefiniert:
    `get empty == Nothing`
  - Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:
    `get (upd s a v) a == Just v`
  - Lesen an anderer Stelle als vorher geschrieben liefert ursprünglichen Wert:
    `a1 /= a2` $\Longrightarrow$ `get (upd s a1 v) a2 == get s a2`
- Axiome für das Schreiben:
  - Schreiben an dieselbe Stelle überschreibt alten Wert:
    `upd (upd s a v) a w == upd s a w`
  - Schreiben über verschiedene Stellen kommutiert (Reihenfolge irrelevant):
    `a1 /= a2` $\Longrightarrow$ `upd (upd s a1 v) a2 w == upd (upd s a2 w) a1 v`

Axiome als Interface

- Axiome müssen gelten
  - Für alle Werte der freien Variablen zu `True` auswerten
- Axiome spezifizieren:
  - nach außen das Verhalten
  - nach innen die Implementation
- Signatur + Axiome = Spezifikation



- Implementation kann mit quickCheck getestet werden
- Axiome können (sollten?) bewiesen werden

Implementation des Speicher: erster Versuch

- Speicher als Funktion

```
type Store a b = a-> Maybe b
```

- Leerer Speicher: konstant undefiniert

```
empty :: Store a b
empty = \x-> Nothing
```

- Lesen: Funktion anwenden

```
get :: Eq a=> Store a b-> a-> Maybe b
get s a = s a
```

- Schreiben: punktweise Funktionsdefinition
  - Auf Adresstyp `a` muss Gleichheit existieren

```
upd :: Eq a=> Store a b-> a-> b-> Store a b
upd s a b = \x-> if x== a then Just b else s x
```

Nachteil dieser Implementation

- Typsynomyme immer sichtbar
- Deshalb Verkapselung des Typen:

```
data Store a b = Store (a-> Maybe b)
```

Implementation des Speicher: zweiter Versuch

- Speicher als Funktion

```
data Store a b = Store (a-> Maybe b)
```

- Leerer Speicher: konstant undefiniert

```
empty :: Store a b
empty = Store (\x-> Nothing)
```

- Lesen: Funktion anwenden

```
get :: Eq a=> Store a b-> a-> Maybe b
get (Store s) a = s a
```

- Schreiben: punktweise Funktionsdefinition
  - Auf Adresstyp `a` muss Gleichheit existieren

```
upd :: Eq a=> Store a b-> a-> b-> Store a b
upd (Store s) a b =
  Store (\x-> if x== a then Just b else s x)
```

Beweis der Axiome

- Lesen aus leerem Speicher:

```
  get empty a
= get (Store (\x-> Nothing)) a
= (\x-> Nothing) a
= Nothing
```

- Lesen und Schreiben an gleicher Stelle:

```
  get (upd (Store s) a v) a
= get (\x-> if x == a then Just v else s x) a
= (\x-> if x == a then Just v else s x) a
= if a == a then Just v else s a
= if True then Just v else s a
= Just v
```

Beweis der Axiome

- Lesen an anderer Stelle:

```
  get (upd (Store s) a v) b
= get (\x-> if x == a then Just v else s x) b
= (\x-> if x == a then Just v else s x) b
= if a == b then Just v else s b
= if False then Just v else s b
= s b
= get (Store s) b
```

Bewertung der Implementation

- Vorteil: effizient (keine Rekursion!)
- Nachteile:
  - Keine Gleichheit auf `Store a b` — Axiome nicht erfüllbar
  - Speicherleck — überschriebene Werte bleiben im Zugriff

Der Speicher als Graph

- Typ `Store a b`: Liste von Paaren `(a, b)`
- Graph $G(f)$ der partiellen Abbildung $f : A \rightharpoonup B$

$$G(f) = \{(a, f(a)) \mid a \in A, f(a) \neq \bot\}$$

- Invariante der Liste: für jedes a höchsten ein Paar (a, v)
- Leerer Speicher: leere Menge
- Lesen von a:
  - Wenn Paar (a, v) in Menge, `Just v`, ansonsten $\bot$
- Schreiben von a an der Stelle v:
  - Existierende (a, w) für alle w entfernen, dann (a, v) hinzufügen

Der Speicher als Funktionsgraph

- Datentyp (verkapselt):

```
data Store a b = Store [(a, b)]
```

- Leerer Speicher:

```
empty :: Store a b
empty = Store []
```

Operationen (rekursiv)

- Lesen: rekursive Formulierung

```
get :: Eq a=> Store a b-> a-> Maybe b
get (Store s) a = get' s where
  get' []          = Nothing
  get' ((b, v):s) =
    if a == b then Just v else get' s
```

- Schreiben

```
upd :: Eq a=> Store a b-> a-> b-> Store a b
upd (Store s) a v = Store (upd' s) where
  upd' []          = [(a, v)]
  upd' ((b, w):s) =
    if a == b then (a, v):s
    else (b,w): upd' s
```

Operationen (kürzer)

- Lesen: kürzere Alternative

```
get (Store s) a =
  case filter ((a ==). fst) s of
    ((b, v):_) -> Just v
    []         -> Nothing
```

- Schreiben:

```
upd (Store s) a v =
  Store ((a, v): filter ((a ==) . fst) s)
```

Axiome als Testbare Eigenschaften

- Test: zufällige Werte einsetzen, Auswertung auf `True` prüfen
- Polymorphe Variablen nicht testbar
- Deshalb Typvariablen instantiieren
  - Typ muss genug Element haben (`Int`)
  - Durch Signatur Typinstanz erzwingen
- Freie Variablen der Eigenschaft werden Parameter der Testfunktion

Axiome als Testbare Eigenschaften

- Für das Lesen:

```
prop_read_empty :: Int-> Bool
prop_read_empty a =
  get (empty :: Store Int Int) a == Nothing

prop_read_write :: Store Int Int-> Int-> Int-> Bool
prop_read_write s a v=
  get (upd s a v) a == Just v
```

Axiome als Testbare Eigenschaften

- Bedingte Eigenschaft in `quickCheck`:
  - A `==>` B mit A, B Eigenschaften
  - Typ ist `Property`

```
prop_read_write_other ::
  Store Int Int-> Int-> Int-> Int-> Property

prop_read_write_other s a v b=
  a /= b ==> get (upd s a v) b == get s b
```

Axiome als Testbare Eigenschaften

- Schreiben:

```
prop_write_write :: Store Int Int-> Int-> Int-> Int-> Bool
prop_write_write s a v w =
  upd (upd s a v) a w == upd s a w
```

- Schreiben an anderer Stelle:

```
prop_write_other ::
  Store Int Int-> Int-> Int-> Int-> Int-> Property
prop_write_other s a v b w =
  a /= b ==> upd (upd s a v) b w == upd (upd s b w) a v
```

- Test benötigt Gleichheit auf `Store a b`
  - Mehr als Gleichheit der Listen — Reihenfolge irrelevant

Beweis der Eigenschaften

- Problem: Rekursion

```
  read (upd (Store s) a v) a
= read (Store (upd' s a v)) a
= read (Store ( ... ?)) a
```

- Lösung: nächste Vorlesung

ADTs vs. Objekte

- ADTs (z.B. Haskell): Typ plus Operationen
- Objekte (z.B. Java): Interface, Methoden.
- Gemeinsamkeiten: Verkapselung (information hiding) der Implementation
- Unterschiede:
  - Objekte haben internen Zustand, ADTs sind referentiell transparent;
  - Objekte haben Konstruktoren, ADTs nicht (Konstruktoren nicht unterscheidbar)
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt, Haskell: Signatur eines Moduls nicht (aber z.B. SML).

Zusammenfassung

- Signatur: Typ und Operationen eines ADT
- Axiome: über Typen formulierte Eigenschaften
- Spezifikation = Signatur + Axiome
  - Interface zwischen Implementierung und Nutzung
  - Testen zur Erhöhung der Konfidenz
  - Beweisen der Korrektheit
- `quickCheck`:
  - Freie Variablen der Eigenschaften werden Parameter der Testfunktion
  - `==>` für bedingte Eigenschaften