Deklarative (= fortgeschrittene) Programmierung Vorlesung WS 09, WS 10, SS 12, SS 13

Johannes Waldmann, HTWK Leipzig

27. März 2013

Formen der deklarative Programmierung

- funktionale Programmierung:

```
foldr (+) 0 [1,2,3]
foldr f z l = case l of
    [] -> z ; (x:xs) -> f x (foldr f
```

- logische Programmierung:

```
append(A,B,[1,2,3]).
append([],YS,YS).
append([X|XS],YS,[X|ZS]):-append(XS,Y
```

- Constraint-Programmierung

```
(set-logic QF_LIA) (set-option :produce-model
(declare-fun a () Int) (declare-fun b () Int)
(assert (and (>= a 5) (<= b 30) (= (+ a b) 20
(check-sat) (get-value (a b))
```

Definition

deklarativ: jedes (Teil-)Programm/Ausdruck hat einen *Wert*
(. . . und keine weitere (versteckte) *Wirkung*).
Werte können sein:

- “klassische” Daten (Zahlen, Listen, Bäume. . .)
- Funktionen (Sinus, . . .)
- Aktionen (Datei schreiben, . . .)

Softwaretechnische Vorteile

. . . der deklarativen Programmierung

- Beweisbarkeit: Rechnen mit Programmen wie in der Mathematik mit Termen
- Sicherheit: es gibt keine Nebenwirkungen und Wirkungen sieht man bereits am Typ
- Wiederverwendbarkeit: durch Entwurfsmuster (= Funktionen höherer Ordnung)
- Effizienz: durch Programmtransformationen im Compiler,
- Parallelisierbarkeit: durch Nebenwirkungsfreiheit

Beispiel Spezifikation/Test

```
import Test.SmallCheck

append :: forall t . [t] -> [t] -> [t]
append x y = case x of
    [] -> y
    h : t -> h : append t y

associative f =
    \ x y z -> f x (f y z) == f (f x y) z

test1 = smallCheckI
  (associative (append::[Int]->[Int]->[In
..
```

Beispiel Verifikation

```
app x y = case x of
    []  -> y
    h : t -> h : app t y
```

Beweise

```
app x (app y z) == app (app x y) z
```

Beweismethode: Induktion nach `x`.

- Induktionsanfang: `x == []` ...
- Induktionsschritt: `x == h : t` ...

Beispiel Parallelisierung

Klassische Implementierung von Mergesort

```
sort :: Ord a => [a] -> [a]
sort [] = [] ; sort [x] = [x]
sort xs =  let ( left,right ) = split xs
               sleft  = sort left
               sright = sort right
           in  merge sleft sright
```

wird parallelisiert durch *Annotationen*:

```
  sleft  = sort left
                `using` rpar `dot` spineLis
  sright = sort right `using` spineList
```

vgl. http://thread.gmane.org/gmane.comp.

Deklarative Programmierung in der Lehre

- funktionale Programmierung: diese Vorlesung
- logische Programmierung: in *Angew. Künstl. Intell.*
- Constraint-Programmierung: als Master-Wahlfach

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Alg.+DS, Grundlagen Theor. Inf.)
- Logik (Grundlagen TI, Softwaretechnik)

Anwendungen:

- Softwarepraktikum

Konzepte und Sprachen

Funktionale Programmierung ist ein *Konzept*.

Realisierungen:

- in prozeduralen Sprachen:
  - Unterprogramme als Argumente (in Pascal)
  - Funktionszeiger (in C)
- in OO-Sprachen: Befehlsobjekte
- Multi-Paradigmen-Sprachen:
  - Lambda-Ausdrücke in C#, Scala, Clojure
- funktionale Programmiersprachen (LISP, ML, Haskell)

Die Erkenntnisse sind sprachunabhängig.

- A good programmer can write LISP in any language.

Gliederung der Vorlesung

- Terme, Termersetzungssysteme algebraische Datentypen, Pattern Matching, Persistenz
- Funktionen (polymorph, höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
- Typklassen zur Steuerung der Polymorphie
- Bedarfsauswertung, unendl. Datenstrukturen (Iterator-Muster)
- Code-Qualität, Code-Smells, Refactoring

Softwaretechnische Aspekte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
  Entwurfsmuster Kompositum, immutable objects,
  das Datenmodell von Git
- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
  Lambda-Ausdrücke in C#, Entwurfsmuster Besucher
  Codequalität, code smells, Refaktorisierung
- Typklassen zur Steuerung der Polymorphie
  Interfaces in Java/C# , automatische Testfallgenerierung

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben (teilw. autotool)
- Prüfung: Klausur (ohne Hilfsmittel)

Literatur

- Skripte:
  - Deklarative Programmierung WS10
    `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws10/dp/folien/main/`
  - Softwaretechnik II SS11
    `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss11/st2/folien/main/`
- Entwurfsmuster:
  `http://www.imn.htwk-leipzig.de/~waldmann/draft/pub/hal4/emu/`
- Maurice Naftalin und Phil Wadler: *Java Generics and Collections*, O'Reilly 2006
- `http://haskell.org/` (Sprache, Werkzeuge, Tutorials),

Übungen KW11

- im Pool Z423
  - `export PATH=/usr/local/waldmann/bin:$PATH`
- Beispiele f. deklarative Programmierung
  - funktional: Haskell mit ghci,
  - logisch: Prolog mit swipl,
  - constraint: mit mathsat, z3
- Haskell-Entwicklungswerkzeuge
  - (eclipsefp, leksah, . . . ,
  - aber *real programmers . . .* `http://xkcd.org/378/`)
  - API-Suchmaschine `http://www.haskell.org/hoogle/`

Wiederholung: Terme

- (Prädikatenlogik) *Signatur* Σ ist Menge von Funktionssymbolen mit Stelligkeiten
  ein Term t in Signatur Σ ist
  - Funktionssymbol $f \in \Sigma$ der Stelligkeit k
    mit Argumenten $(t_1, \ldots, t_k)$, die selbst Terme sind.

  Term(Σ) = Menge der Terme über Signatur Σ
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
  - Funktionssymbol = Konstruktor, Term = Baum

Beispiele: Signatur, Terme

- Signatur: $\Sigma_1 = \{Z/0, S/1, f/2\}$
- Elemente von Term(Σ_1):
  $Z(), \quad S(S(Z())), \quad f(S(S(Z())), Z())$
- Signatur: $\Sigma_2 = \{E/0, A/1, B/1\}$
- Elemente von Term(Σ_2): ...

Algebraische Datentypen

```
data Foo = Foo { bar :: Int, baz :: Strin
    deriving Show
```

Bezeichnungen (benannte Notation)

- `data Foo` ist Typname
- `Foo { .. }` ist Konstruktor
- `bar, baz` sind Komponenten

```
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
y = Foo 3 "bar"
```

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

```
data T = A { foo :: Int }
       | B { bar :: String, baz :: Bool }
    deriving Show
```

Bespiele (in Prelude vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

Rekursive Datentypen

```
data Tree = Leaf {}
    | Branch { left :: Tree
             , right :: Tree }
```

Übung: Objekte dieses Typs erzeugen (benannte und positionelle Notation der Konstruktoren)

Bezeichnungen für Teilterme

- *Position*: Folge von natürlichen Zahlen

  (bezeichnet einen Pfad von der Wurzel zu einem Knoten)

  Beispiel: für $t = S(f(S(S(Z())), Z()))$

  ist $[0, 1]$ eine Position in t.

- $\mathrm{Pos}(t) =$ die Menge der Positionen eines Terms t

  Definition: wenn $t = f(t_1, \ldots, t_k)$,

  dann $\mathrm{Pos}(t) = \{[]\} \cup \{[i-1] +\!\!+ p \mid 1 \leq i \leq k \wedge p \in \mathrm{Pos}(t_i)\}$.

dabei bezeichnen:

[] die leere Folge

Operationen mit (Teil)Termen

- $t[p] =$ der Teilterm von t an Position p
  Beispiel: $S(f(S(S(Z())), Z()))[0, 1] = \ldots$
  Definition (durch Induktion über die Länge von p): ...
- $t[p := s]$: wie t, aber mit Term s an Position p
  Beispiel:
  $S(f(S(S(Z())), Z()))[[0, 1] := S(Z)]x = \ldots$
  Definition (durch Induktion über die Länge von p): ...

Operationen mit Variablen in Termen

- Term(Σ, V) = Menge der Terme über Signatur Σ mit Variablen aus V
  Beispiel: $\Sigma = \{Z/0, S/1, f/2\}$, $V = \{y\}$,
  $f(Z(), y) \in \text{Term}(\Sigma, V)$.
- Substitution σ: partielle Abbildung $V \to \text{Term}(\Sigma)$
  Beispiel: $\sigma_1 = \{(y, S(Z()))\}$
- eine Substitution auf einen Term anwenden: $t\sigma$:
  Intuition: wie t, aber statt v immer $\sigma(v)$
  Beispiel: $f(Z(), y)\sigma_1 = f(Z(), S(Z()))$
  Definition durch Induktion über t

Termersetzungssysteme

- Daten $=$ Terme (ohne Variablen)
- Programm $R =$ Menge von Regeln
  Bsp: $R = \{(f(Z(), y), y),\ (f(S(x), y), S(f(x, y)))\}$
- Regel $=$ Paar (l, r) von Termen mit Variablen
- Relation $\rightarrow_R$ ist Menge aller Paare (t, t') mit
  - es existiert $(l, r) \in R$
  - es existiert Position p in t
  - es existiert Substitution
    $\sigma : (\mathrm{Var}(l) \cup \mathrm{Var}(r)) \rightarrow \mathrm{Term}(\Sigma)$
  - so daß $t[p] = l\sigma$ und $t' = t[p := r\sigma]$.

Termersetzungssysteme als Programme

- to_R beschreibt *einen* Schritt der Rechnung von R,
- transitive Hülle $\rightarrow_R^*$ beschreibt *Folge* von Schritten.
- *Resultat* einer Rechnung ist Term in R-Normalform ($=$ ohne $\rightarrow_R$-Nachfolger)

dieses Berechnungsmodell ist im allgemeinen

- *nichtdeterministisch*
  $R_1 = \{C(x, y) \rightarrow x, C(x, y) \rightarrow y\}$
  (ein Term kann mehrere $\rightarrow_R$-Nachfolger haben, ein Term kann mehrere Normalformen erreichen)

Übung Terme, TRS

- Geben Sie die Signatur des Terms $\sqrt{a \cdot a + b \cdot b}$ an.
- Geben Sie ein Element $t \in \mathrm{Term}(\{f/1, g/3, c/0\})$ an mit $t[1] = c()$.

mit `ghci`:

- ```
  data T = F T | G T T T | C deriving Sh
  ```
  erzeugen Sie o.g. Terme (durch Konstruktoraufrufe)

Die *Größe* eines Terms t ist definiert durch $|f(t_1, \ldots, t_k)| = 1 + \sum_{i=1}^{k} |t_i|$.

- Bestimmen Sie $|\sqrt{a \cdot a + b \cdot b}|$.
  ```

Funktionale Programme

. . . sind spezielle Term-Ersetzungssysteme. Beispiel:
Signatur: S einstellig, Z nullstellig, f zweistellig.
Ersetzungssystem
$\{f(Z, y) \to y, f(S(x), y) \to S(f(x, y))\}$.
Startterm $f(S(S(Z)), S(Z))$.
entsprechendes funktionales Programm:

```
data N = S N | Z
f :: N -> N -> N
f a y = case a of
    Z   -> y
    S x -> S (f x y)
```

Aufruf: `f (S (S Z)) (S Z)`
Auswertung = Folge von Ersetzungsschritten $\to^*$

`data` und `case`

typisches Vorgehen beim Programmieren einer Funktion

```
f :: T -> ...
```

- Für jeden Konstruktor des Datentyps

```
data T = C1 ...
       | C2 ...
```

- schreibe einen Zweig in der Fallunterscheidung

```
f x = case x of
   C1 ... -> ...
   C2 ... -> ...
```

Rekursive Datentypen und Funktionen

Wenn der Datentyp rekursiv ist,
dann auch die Funktion, die ihn verarbeitet:

```
data Tree a = Leaf {}
    | Branch { left :: Tree a, key :: a
             , right :: Tree a}
leaves :: Tree a -> Int
leaves t = case t of
    Leaf   {} -> 1
    Branch {} -> ...
```

Aufgabe: erzeuge und bestimme Blatt-Anzahl für:

- vollständige Binärbäume

Peano-Zahlen

```
data N = Z | S N

plus :: N -> N -> N
plus x y = case x of
    Z -> y
    S x' -> S (plus x' y)
```

Aufgaben:

- implementiere Multiplikation, Potenz
- beweise die üblichen Eigenschaften (Addition, Multiplikation sind assoziativ, kommutativ, besitzen neutrale Element)

Wiederholung Bäume

```
data Tree a = Leaf {}
    | Branch { left :: Tree a
             , key :: a
             , right :: Tree a }
branches :: Tree a -> Int
branches t = case t of
    Leaf {} -> 0
    Branch {} ->
        branches (left t) + 1 + branches
```

Zusammenhang:

Datentyp	Funktion
zwei Konstruktoren	zwei Zweige

Listen

eigentlich:

```
data List a = Nil {}
    | Cons { head :: a, tail :: List a }
```

wegen früher häufiger Benutztung verkürzte Schreibweise:

```
List a = [a], Nil = [], Cons = (:)
```

```
data [a] = []
         | (:) { head :: a, tail :: [a] }
```

Pattern matching für Listen:

```
append :: [a] -> [a] -> [a]
append a b = case a of
    []     -> b
```

Operationen auf Listen

- append:
  - Definition
  - Beweis Assoziativität, neutrales Element
- reverse:
  - Definition
  - Beweis: `reverse (reverse xs) == xs`
  - benutze Funktion mit dieser Spezifikation

    ```
    arev :: [a] -> [a] -> [a]
    arev xs ys == append (reverse xs) ys
    ```

    zur effizienteren Implementierung von `reverse`

Überblick

- alle Attribute aller Objekte sind unveränderlich (`final`)
- anstatt Objekt zu ändern, konstruiert man ein neues

Eigenschaften des Programmierstils:

- vereinfacht Formulierung und Beweis von Objekteigenschaften
- parallelisierbar (keine updates, keine *data races*)
  `http://fpcomplete.com/the-downfall-of-imperative-programmin`
- Persistenz (Verfügbarkeit früherer Versionen)
- Belastung des Garbage Collectors (. . . dafür ist

Beispiel: Einfügen in Baum

- destruktiv:

```
interface Tree<K> { void insert (K ke
Tree<String> t = ... ;
t.insert ("foo");
```

- persistent (Java):

```
interface Tree<K> { Tree<K> insert (K
Tree<String> t = ... ;
Tree<String> u = t.insert ("foo");
```

- persistent (Haskell):

```
insert :: Tree k -> k -> Tree k
```

Beispiel: (unbalancierter) Suchbaum

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)
insert :: Ord k => k -> Tree k -> Tree k
insert k t = case t of ...
```

Diskussion:

- `Ord k` entspricht
  `K implements Comparable<K>`,
  genaueres später (Haskell-Typklassen)
- wie teuer ist die Persistenz?
  (wieviel Müll entsteht bei einem `insert`?)

Beispiel: Sortieren mit Suchbäumen

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)

insert :: Ord k => k -> Tree k -> Tree k

build :: Ord k => [k] -> Tree k
build = foldr ... ...

sort :: Ord k => [k] -> [k]
sort xs = ... ( ... xs )
```

Persistente Objekte in Git

`http://git-scm.com/`

- *Distributed* development.
- Strong support for *non-linear* development.
  (Branching and merging are fast and easy.)
- Efficient handling of *large* projects.
  (z. B. Linux-Kernel, `http://kernel.org/`)
- Toolkit design.
- Cryptographic authentication of history.

Objekt-Versionierung in Git

- Objekt-Typen:
  - Datei (blob),
  - Verzeichnis (tree), mit Verweisen auf blobs und trees
  - Commit, mit Verweisen auf tree und commits (Vorgänger)

```
git cat-file [-t|-p] <hash>
git ls-tree [-t|-p] <hash>
```

- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit $=$ 40 Hex-Zeichen) identifiziert
- statt Überschreiben: neue Objekte anlegen
- jeder Zustand ist durch Commit-Hash (weltweit) eindeutig beschrieben und kann

Anwendung, Ziele

- aktuelle Quelltexte eines Projektes sichern
- auch frühere Versionen sichern
- gleichzeitiges Arbeiten mehrere Entwickler
- . . . an unterschiedlichen Versionen (Zweigen)

Das Management bezieht sich auf *Quellen* (.c, .java, .tex, Makefile)
abgeleitete Dateien (.obj, .exe, .pdf, .class) werden daraus erzeugt, stehen aber *nicht* im Archiv

Welche Formate?

- Quellen sollen Text-Dateien sein, human-readable, mit Zeilenstruktur: ermöglicht Feststellen und Zusammenfügen von unabhängigen Änderungen
- ergibt Konflikt mit Werkzeugen (Editoren, IDEs), die Dokumente nur in Binärformat abspeichern. — Das ist sowieso *evil*, siehe Robert Brown: Readable and Open File Formats, `http://www.few.vu.nl/~feenstra/read_and_open.html`
- Programme mit grafischer Ein- und Ausgabe sollen Informationen *vollständig* von und nach Text konvertieren können

Daten und Operationen

Daten:

- Archiv (repository)
- Arbeitsbereich (sandbox)

Operationen:

- check-out: repo $\longrightarrow$ sandbox
- check-in: sandbox $\longrightarrow$ repo

Projekt-Organisation:

- ein zentrales Archiv (CVS, Subversion)
- mehrere dezentrale Archive (Git)

Zentrale und dezentrale Verwaltung

zentral (CVS, SVN)

- ein zentrales Repository
- pull: `svn up`, push `svn ci`
- erfordert Verwaltung der Schreibberechtigungen für Repository

dezentral (Git)

- jeder Entwickler hat sein Repository
- pull: von anderen Repos, push: nur zu eigenem

Versionierung (intern)

. . . automatische Numerierung/Benennung

- CVS: jede Datei einzeln
- SVN: gesamtes Repository
- darcs: Mengen von Patches
- git: Snapshot eines (Verzeichnis-)Objektes

Versionierung (extern)

. . . mittels Tags (manuell erzeugt)

empfohlenes Schema:

- Version $=$ Liste von drei Zahlen $[x, y, z]$
- Ordung: lexikographisch. (Spezifikation?)

Änderungen bedeuten:

- x (major): inkompatible Version
- y (minor): kompatible Erweiterung
- z (patch): nur Fehlerkorrektur

Sonderformen:

- y gerade: stabil, y ungerade: Entwicklung
- z Datum

Arbeit mit Zweigen (Branches)

- Repo anlegen: `git init`
- im Haupt-Zweig (master) arbeiten:
  `git add <file>; git commit -a`
- abbiegen:
  `git branch <name>; git checkout <name>`
- dort arbeiten: `... ; git commit -a`
- zum Haupt-Zweig zurück:
  `git checkout master`
- dort weiterarbeiten :`... ; git commit -a`
- zum Neben-Zweig: `git checkout <name>`
- Änderung aus Haupt-Zweig übernehmen:
  `git merge master`

Übernehmen von Änderungen (Merge)

durch divergente Änderungen entsteht Zustand mit 3 Versionen einer Datei:

- gemeinsamer Start *G*
- Versionen *I*, *D* (ich, du)

Merge:

- Änderung $G \rightarrow D$ *bestimmen*
- und auf *I anwenden*,
- falls das *konfliktfrei* möglich ist.

Änderung $=$ Folge von Editor-Befehlen (Kopieren, Einfügen, Löschen)
betrachten dabei immer ganze Zeilen