Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 07.11.2006: Funktionen und Basistypen

Christoph Lüth

Organisatorisches

- Tutorien: Ungleichverteilung.
  - Di nachmittag leer
- Übungsblätter:
  - Lösungen in LaTeX (siehe Webseite)

Inhalt

- Wie definiere ich eine Funktion?
  - Syntaktische Feinheiten
  - Von der Spezifikation zum Programm
- Basisdatentypen:
  - Wahrheitswerte
  - numerische Typen
  - alphanumerische Typen

Wie definiere ich eine Funktion?

Generelle Form:

- Signatur:

```
max :: Int-> Int-> Int
```

- Definition

```
max x y = if x < y then y else x
```

  - Kopf, mit Parametern
  - Rumpf (evtl. länger, mehrere Zeilen)
  - Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf
  - Was gehört zum Rumpf (Gültigkeitsbereich)?

Die Abseitsregel

Funktionsdefinition:

$\texttt{f}\ \texttt{x}_1\ \texttt{x}_2 \ldots \texttt{x}_n = \texttt{E}$

- Gültigkeitsbereich der Definition von `f`:

  alles, was gegenüber `f` eingerückt ist.

- Beispiel:

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

- Gilt auch verschachtelt.

Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen . . .

```
f x y = if B1 then P else
        if B2 then Q else ...
```

. . . bedingte Gleichungen:

```
f x y
  | B1 = ...
  | B2 = ...
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: Laufzeitfehler! Deshalb:

```
  | otherwise = ...
```

Kommentare

- Pro Zeile: Ab `--` bis Ende der Zeile

```
f x y = irgendwas  -- und hier der Kommentar!
```

- Über mehrere Zeilen: Anfang `{-`, Ende `-}`

```
{-
   Hier fängt der Kommentar an
   erstreckt sich über mehrere Zeilen
   bis hier                                -}
f x y = irgendwas
```

  - Kann geschachtelt werden.

Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y                          f x y =
  | g = P y                     let y = M
  | otherwise = Q where             f x = N x
    y = M                       in  if g then P y
    f x = N x                           else Q
```

- `f`, `y`, ... werden gleichzeitig definiert (Rekursion!)
- Namen `y` und Parameter (`x`) überlagern andere
- Es gilt die Abseitsregel
  - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

Die Alternative: Literate Programming

- Literate Haskell (`.lhs`): Quellcode besteht hauptsächlich aus Kommentar, Programmcode ausgezeichnet.
- In Haskell zwei Stile:
  - Alle Programmzeilen mit `>` kennzeichnen.
  - Programmzeilen in `\begin{code}` ... `\end{code}` einschließen
- Umgebung `code` in LaTeX definieren (oder `pi3.cls` nutzen)

```
\def\code{\verbatim}
\def\endcode{\endverbatim}
```

- Mit LaTeX setzen, mit Haskell ausführen: (Quelle, ausführen).

```
test x = if x == 0 then 1 else 0
```

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur
- Programmentwurf:
  - Gibt es ein ähnliches (gelöstes) Problem?
  - Wie kann das Problem in Teilprobleme zerlegt werden?
  - Wie können Teillösungen zusammengesetzt werden?

  $\rightsquigarrow$ Erster Entwurf

Funktionaler Entwurf und Entwicklung

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Mögliche Verallgemeinerungen?
  - Gibt es hilfreiche Büchereifunktionen?
  - Wie würde man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung

Funktionaler Entwurf und Entwicklung

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Mögliche Verallgemeinerungen?
  - Gibt es hilfreiche Büchereifunktionen?
  - Wie würde man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung
- Test:
  - Black-box Test: Testdaten aus der Spezifikation
  - White-box Test: Testdaten aus der Implementierung
  - Testdaten: hohe Abdeckung, Randfälle beachten.

Einfaches Beispiel: größter gemeinsame Teiler.

- Eingabebereich: `Int Int` Ausgabebereich: `Int`

Einfaches Beispiel: größter gemeinsame Teiler.

- Eingabebereich: `Int Int` Ausgabebereich: `Int`
- Analyse des Problems
  - Reduktion auf kleineres Teilproblem:
    $a > b \rightsquigarrow \gcd(a, b) = \gcd(a - b, b)$

Einfaches Beispiel: größter gemeinsame Teiler.

- Eingabebereich: `Int Int` Ausgabebereich: `Int`
- Analyse des Problems
  - Reduktion auf kleineres Teilproblem:
    $a > b \rightsquigarrow \gcd(a, b) = \gcd(a - b, b)$
  - Abbruchbedingung: beide Argumente gleich

Einfaches Beispiel: größter gemeinsame Teiler.

- Eingabebereich: `Int Int` Ausgabebereich: `Int`
- Analyse des Problems
  - Reduktion auf kleineres Teilproblem:
    $a > b \rightsquigarrow \gcd(a, b) = \gcd(a - b, b)$
  - Abbruchbedingung: beide Argumente gleich
- Implementierung:

```
gcd :: Int-> Int-> Int
gcd a b
   | a == b     = a
   | a < b      = gcd b a
   | otherwise = gcd (a- b) b
```

- Testfälle: `gcd 2 2, gcd 5 1, gcd 50 45, gcd 123 167` Testen.

Kritik der Lösung

- Terminiert nicht bei negativen Zahlen oder 0.

```
gcd2 :: Int-> Int-> Int
gcd2 a b = gcd' (abs a) (abs b) where
  gcd' a b | a == 0 && b == 0 = error "gcd 0 0 undefined"
           | a == b || b == 0 = a
           | a < b            = gcd' b a
           | otherwise        = gcd' (a- b) b
```

Kritik der Lösung

- Terminiert nicht bei negativen Zahlen oder 0.

```
gcd2 :: Int-> Int-> Int
gcd2 a b = gcd' (abs a) (abs b) where
  gcd' a b | a == 0 && b == 0 = error "gcd 0 0 undefined"
           | a == b || b == 0 = a
           | a < b            = gcd' b a
           | otherwise        = gcd' (a- b) b
```

- Ineffizient — es gilt auch $\gcd(a, b) = \gcd(b, a \mod b)$ (Euklid'scher Algorithmus)

Kritik der Lösung

- Terminiert nicht bei negativen Zahlen oder 0.

```
gcd2 :: Int-> Int-> Int
gcd2 a b = gcd' (abs a) (abs b) where
  gcd' a b | a == 0 && b == 0 = error "gcd 0 0 undefined"
           | a == b || b == 0 = a
           | a < b            = gcd' b a
           | otherwise        = gcd' (a- b) b
```

- Ineffizient — es gilt auch $\gcd(a, b) = \gcd(b, a \mod b)$ (Euklid'scher Algorithmus)
- Es gibt eine Büchereifunktion.

Ein längeres Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.

Ein längeres Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.
- Eingabe: Anzahl Hölzchen gesamt, Zug
- Zug = Anzahl genommener Hölzchen
- Ausgabe: Gewonnen, ja oder nein.

```
type Move= Int
winningMove :: Int-> Move-> Bool
```

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

```
winningMove total move =
  legalMove total move &&
  mustLose (total-move)
```

- Überprüfung, ob Zug legal:

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

```
winningMove total move =
  legalMove total move &&
  mustLose (total-move)
```

- Überprüfung, ob Zug legal:

```
legalMove :: Int-> Int-> Bool
legalMove total m =
   (m<= total) && (1<= m) && (m<= 3)
```

- Gegner kann nicht gewinnen, wenn
  - nur noch ein Hölzchen über, oder
  - kann nur Züge machen, bei denen es Antwort gibt, wo wir gewinnen

- Gegner kann nicht gewinnen, wenn
  - nur noch ein Hölzchen über, oder
  - kann nur Züge machen, bei denen es Antwort gibt, wo wir gewinnen

```
mustLose :: Int-> Bool
mustLose n
  | n == 1     = True
  | otherwise = canWin n 1 && canWin n 2 && canWin n 3
```

- Wir gewinnen, wenn es legalen, gewinnenden Zug gibt:

```
canWin :: Int-> Int-> Bool
canWin total move =
  winningMove (total- move) 1 ||
  winningMove (total- move) 2 ||
  winningMove (total- move) 3
```

- Analyse:
  - Effizienz: unnötige Überprüfung bei `canWin`
  - Testfälle: Gewinn, Verlust, Randfälle. Testen.

```
canWin :: Int-> Int-> Bool
canWin total move =
  winningMove (total- move) 1 ||
  winningMove (total- move) 2 ||
  winningMove (total- move) 3
```

- Analyse:
  - Effizienz: unnötige Überprüfung bei `canWin`
  - Testfälle: Gewinn, Verlust, Randfälle. Testen.
- Korrektheit:
  - Vermutung: Mit $4n + 1$ Hölzchen verloren, ansonsten gewonnen.
  - Beweis durch Induktion $\rightsquigarrow$ später.

Wahrheitswerte: `Bool`

- Werte `True` und `False`
- Funktionen:

```
not :: Bool-> Bool              Negation
&&  :: Bool-> Bool-> Bool       Konjunktion
||  :: Bool-> Bool-> Bool       Disjunktion
```

- `&&`, `||` sind rechts nicht strikt
  - `False && 1 'div' 0 == 0` $\rightsquigarrow$ `False`
- Beispiel: ausschließende Disjunktion:

```
exOr :: Bool-> Bool-> Bool
exOr x y = (x || y) && (not (x && y))
```

- Alternative: explizite Fallunterscheidung

```
exOr x y
  | x == True  = if y == False then True else False
  | x == False = if y == True  then True else False
```

- Alternative: explizite Fallunterscheidung

```
exOr x y
  | x == True  = if y == False then True else False
  | x == False = if y == True  then True else False
```

- Igitt! Besser: Definition mit pattern matching

```
exOr True  y = if y == False then True else False
exOr False y = y
```

- Alternative: explizite Fallunterscheidung

```
exOr x y
  | x == True  = if y == False then True else False
  | x == False = if y == True  then True else False
```

- Igitt! Besser: Definition mit pattern matching

```
exOr True  y = if y == False then True else False
exOr False y = y
```

- Noch besser: (`y == False` = `not y`)

```
exOr True  y = not y
exOr False y = y
```

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

$\longleftrightarrow$

beliebige Genauigkeit,
wachsender Aufwand

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand $\longleftrightarrow$ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte ($\geq$ 31 Bit)
- `Integer` - beliebig große ganze Zahlen
- `Rational` - beliebig genaue rationale Zahlen
- `Float` - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: `Int` und `Integer`

- Nützliche Funktionen (überladen, auch für `Integer`):

```
+, *, ^, - :: Int-> Int-> Int
abs        :: Int-> Int -- Betrag
div, quot  :: Int-> Int-> Int
mod, rem   :: Int-> Int-> Int
```

  Es gilt `(x `div` y)*y + x `mod` y == x`

- Vergleich durch `==`, `/=`, `<=`, `<`, ...
- Achtung: Unäres Minus
  - Unterschied zum Infix-Operator `-`
  - Im Zweifelsfall klammern: `abs (-34)`
- Fallweise Definitionen möglich

Fließkommazahlen: `Double`

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen
  - siehe Thompson S. 44
- Konversion in ganze Zahlen:
  - `fromIntegral :: Int, Integer-> Double`
  - `fromInteger :: Integer-> Double`
  - `round, truncate :: Double-> Int, Integer`
  - Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

Exkurs: Operatoren in Haskell

- Operatoren: Namen aus Sonderzeichen `!$%&/?+^ ...`
- Werden infix geschrieben: `x && y`
- Ansonsten normale Funktion.
- Andere Funktion infix benutzen:

  ```
  x `exOr` y
  ```

  - In Apostrophen einschließen.
- Operatoren in Nicht-Infixschreibweise (präfix):

  ```
  (&&) :: Bool-> Bool-> Bool
  ```

  - In Klammern einschließen.

Alphanumerische Basisdatentypen: `Char`

- Notation für einzelne Zeichen: `'a'`,...
  - NB. Kein Unicode.
- Nützliche Funktionen:

```
ord :: Char -> Int
chr :: Int -> Char

toLower :: Char-> Char
toUpper :: Char-> Char
isDigit :: Char-> Bool
isAlpha :: Char-> Bool
```

- Zeichenketten: Listen von Zeichen $\rightsquigarrow$ nächste Vorlesung

Zusammenfassung

- Funktionsdefinitionen:
  - Abseitsregel, bedingte Definition, pattern matching
  - Lokale Definitionen
- Numerische Basisdatentypen:
  - `Int`, `Integer`, `Rational` und `Double`
- Funktionaler Entwurf und Entwicklung
  - Spezifikation der Ein- und Ausgabe $\rightsquigarrow$ Signatur
  - Problem rekursiv formulieren $\rightsquigarrow$ Implementation
  - Test und Korrektheit
- Alphanumerische Basisdatentypen: `Char`
- Nächste Vorlesung: Strukturierte Typen und Funktionen darüber.