

Programmierung und Modellierung

Andreas Abel und Martin Hofmann

Sommersemester 2011

Das TCS-Team

TCS = Theoretical Computer Science

- Personen: Martin Hofmann, Andreas Abel, Steffen Jost, Uli Schöpp, Jan Hoffmann, Robert Grabowski, Dulma Rodriguez, Markus Latte, Max Jakob, Sigrid Roden, Stefan Hüller, Thomas Rau, Vivek Nigam.
- Projekte:
 - Graduiertenkolleg PUMA (mit TU-München). Programm- und Modellanalyse
 - InfoZert (MH Grabowski):
 - RAML+RAJA (MH, Hoffmann, Rodriguez, Jost): Laufzeit- und Ressourcenanalyse für funktionale und objektorientierte Sprachen.
 - PURPLE (MH, Schöpp): platzbeschränkte Berechnung mit Zeigern
 - RELICS (MH, Nigam): “relational logics”, ein Werkzeug für die rigorose Rechtfertigung von Programmäquivalenzen und Optimierungen
 - Informatik und Schule (MH, Hüller, Rau)

Inhalt der Vorlesung

- Grundkonzepte funktionaler Programmierung (Mengen, Funktionen, Terme, Typen, Rekursion, Polymorphie)
- Programmierung mit Standard ML
- Datenstrukturen: Listen, Tupel, Bäume, benutzerdefiniert
- Typsysteme und Modularisierung
- Überblick über andere funktionale Programmiersprachen
- Formale Syntax und Semantik

Termine und Organisatorisches

Vorlesungstermine

02.05., 06.05., 09.05., 13.05.,
16.05., 20.05., 23.05., 27.05., 06.06., 10.06.,
17.06., 20.06., 04.07., 08.07., 11.07., 15.07.,
18.07., 22.07., 25.07.

Übungen

Mo 14-16 (C113, The), 16-18 (B015), 18-20 (DZ001)

Di 12-14 (B015), 14-16 (B185), 16-18 (B015)

Fr 10-12 (B015) 12-14 (B015)

Steffen Jost und Team

Klausur

1. Termin:

2. Termin:

Begleitliteratur

Die VL richtet sich nach den Skripten

F. Bry: Informatik I, LMU, 2002

F. Kröger: Kurzsriptum Informatik I, WS 05/06.

M. Wirsing: Programmierung und Modellierung SS 10 (Folien)

Weiterführende Literatur

- ROBERT HARPER, Programming in Standard ML.
<http://www.cs.cmu.edu/~rwh/introsml/>
- L. PAULSON, Standard ML for the working programmer,
Cambridge University Press.
- GUMM-SOMMER: Einf. in die Informatik, Oldenbourg.
- GERT SMOLKA: "Programmierung -eine Einfhrgung in die
Informatik mit Standard ML", Oldenbourg Verlag, 2008
<http://www.ps.uni-saarland.de/prog-buch/>

Inhalt Kapitel 1: Einführung und Grundlagen

- ② Mengen und Funktionen
- ③ Funktionale Programmiersprachen
- ④ Einführung in SML
- ⑤ Konstanten und Funktionen
- ⑥ Auswertung im Substitutionsmodell

Funktionsbegriff

A und B seien Mengen. Eine **Funktion** f von A nach B ist eine Zuordnung von **genau einem** Element $y = f(x)$ zu jedem Element x einer Teilmenge A' von A .

A' ist der **Definitionsbereich** von f .

Ist $A' \neq A$, so ist f eine partielle Funktion.

Man schreibt $A' = D(f)$.

Beispiele

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

$$f(x) = 1/x$$

$$D(f) = \mathbb{R} \setminus \{0\}$$

$A = B =$ Endliche Folgen von 0en und 1en.

$$f(x) = \begin{cases} 0w, & \text{falls } x = 1w \text{ für ein } w \in A \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$$D(f) = \{1w \mid w \in A\}$$

$$g : \mathbb{N} \rightarrow \mathbb{N}$$

$$g(x) = \begin{cases} x/2, & \text{falls } x \text{ gerade} \\ 3x + 1, & \text{sonst} \end{cases}$$

$$D(g) = \mathbb{N}$$

Beispiele

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = \begin{cases} \text{das kleinste } n \in \mathbb{N} \text{ so dass } \underbrace{g(g(g(\dots g(x)\dots))}_{n \text{ Mal}} = 1, \text{ falls es es} \\ \text{undefiniert sonst} \end{cases}$$

Es ist ein **offenes Problem**, ob $D(f) = \mathbb{N}$

Z.B.: $f(27) = 111$.

Terminologie

$f : A \rightarrow B, D(f) = A'$.

A heißt **Quelle**, B heißt **Ziel** von f .

Wenn $a \in D(f)$, so ist $f(a)$ der **Wert** der **Anwendung** von f auf das **Argument** a .

Man schreibt statt $f(a)$ manchmal auch

fa	Präfixnotation
af	Postfixnotation

Funktionen mit zwei Argumenten

Sind A_1 und A_2 Mengen, so bildet man das **kartesische Produkt**

$$A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ und } a_2 \in A_2\}$$

Ist $f : A_1 \times A_2 \rightarrow B$, so kann man f auf Paare anwenden.

Z.B.:

$$\begin{aligned} f : \mathbb{N} \times \mathbb{N} &\rightarrow \mathbb{N} \\ f(x, y) &= x + 2y^2 \end{aligned}$$

Man schreibt *nicht* $f((x, y))$.

Für solche Funktionen gibt es auch die **Infixnotation** xfy für $f(x, y)$. Etwa, wenn $f = '+'$.

Eine Funktion von $A_1 \times A_2$ nach B heißt **zweistellig**.

Funktionen mit mehreren Argumenten

Sind A_1, \dots, A_n Mengen, so bildet man das kartesische Produkt

$$A_1 \times \cdots \times A_n = \{(a_1, \dots, a_n) \mid a_i \in A_i \text{ für } i = 1 \dots n\}$$

Die Elemente von $A_1 \times \dots \times A_n$ heißen **n -Tupel** (Verallg. von Tripel, Quadrupel, Quintupel, Sextupel, ...).

Solch eine Funktion heißt **n -stellig** (Vokabeln: **Stelligkeit**, **n -ary**, **arity**)

Nur äußerst selten ist $n > 6$.

Kartesische Produkte als Ziel

Es gibt auch Funktionen, die Paare oder gar n -Tupel zurückliefern.

$$\text{divmod} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$$

$$D(\text{divmod}) = \mathbb{N} \times (\mathbb{N} \setminus \{0\})$$

$$\text{divmod}(a, b) = (q, r), \text{ wobei } a = qb + r \text{ und } q, r \in \mathbf{N} \text{ und } r < b$$

Funktionale Programmiersprachen

- Programm definiert mathematische Funktionen.
 - (meist) keine Seiteneffekte: Ergebnis hängt nur von den Argumenten ab.
- Funktionale Programmierung erlaubt es, Programme auf viele verschiedene Weisen zu komponieren, insbesondere:
 - Funktionen sind Daten und können als
 - Argumente von Funktionen übergeben und als
 - Resultate zurückgegeben werden.

Theoretische Basis: λ -Kalkül (Alonzo Church 1903–1995)

- Bekannte funktionale Sprachen
 - LISP: die “Urmutter” der fkt. Sprachen, dynamische Bindung
 - Scheme: eine einfache LISP-Variante
 - SML: eine typisierte fkt. Sprache, stat. Bindung
 - OCaml: objekt-orientierte Erweiterung eines SML-Dialekts, mittlerweile verbreiteter als SML.
 - Haskell: “reine” fkt. Sprache mit nichtstriktter Auswertung.
 - F#: ML-Variante für .Net (in VisualStudio2010)
 - Scala: funktionale Erweiterung von Java.
 - Domainspezifische Sprachen: XSLT, Erlang, MapReduce ▶

Arbeitsweise von SML

SML hat zwei Modi:

- Interaktiver Modus: Man gibt Definitionen ein; SML wertet sie aus und zeigt den Wert an.
- Compilierender Modus: Man schreibt ein SML Programm in eine oder mehrere Dateien. Der SML Compiler übersetzt sie und liefert ein ausführbares Programm ("EXE-Datei").

Wir befassen uns hauptsächlich und zunächst mit dem interaktiven Modus.

Die erste SML-Sitzung

Eröffnen einer SML Sitzung durch Eingabe von `sml`.

Es erscheint die Ausgabe: Das Zeichen `-` ist ein **Prompt**. Es fordert uns auf, eine Eingabe zu tätigen.

```
Standard ML of New Jersey v110.69 [built: Mon Jun  8 14:15:08 20
```

```
-  
  
- 17;  
val it = 17 : int  
- 007;  
val it = 7 : int  
- ~5;  
val it = ~5 : int  
- ~(~5);  
val it = 5 : int  
  
- 5.1;  
val it = 5.1 : real  
- val a = 18.35;  
val a = 18.35 : real  
- val aquadrat = a * a;  
val aquadrat = 336.7225 : real  
- val b = ~0.31 / a;  
val b = ~0.01689373297 : real  
-
```

Die Eingaben nach dem Prompt wurden vom Benutzer getätigt, die mit anderen Zeilen sind Ausgaben von SML. `it` bezeichnet den Wert der letzten Auswertung. `int` und `real` sind Datentypen.

Ausdrücke, Werte, Typen

- SML (genauer: das SML-System) wertet Ausdrücke aus.
- Ein Ausdruck kann
 - atomar sein, wie z.B. 17 , ~ 5 , oder
 - zusammengesetzt sein, wie z.B. $12+4$, $12 \text{ div } (3+2)$.
- Jeder korrekt gebildete Ausdruck besitzt einen **Typ**:
- ein Typ ist eine Menge von Werten; z.B.
 - der Typ `int` bezeichnet die Menge der ganzen Zahlen (von -2^{30} bis $2^{30} - 1$)
 - `int` ist der Typ von $12+4$
- Ein Ausdruck hat (meistens) auch einen **Wert**;
- Dieser Wert ist ein Element des Typs des Ausdrucks,
- Z.B.: 16 ist der Wert von $12+4$
- Manche Ausdrücke haben keinen Wert; z.B.
- $1 \text{ div } 0$ (da die partielle Funktion `div` für $1 \text{ div } 0$ undefiniert ist)

Ausdrücke, Werte, Typen

- Auch Operationen (und allgemein Funktionen) haben Typen, z.B.
 - die Funktion `+` erhält als Argumente zwei (atomare oder zusammengesetzte) Ausdrücke vom Typ `int` und liefert einen Wert ebenfalls vom Typ `int`.
 - die Gleichheit für ganze Zahlen ist eine Funktion, die als Argumente zwei Ausdrücke vom Typ `int` erhält und einen Wert vom Typ `bool` liefert.
- Man schreibt:
`+ : int * int -> int`
`= : int * int -> bool`
- Bei der Bildung zusammengesetzter Ausdrücke muss immer auf die Typen der verwendeten Operationen und der eingesetzten Teilausdrücke geachtet werden.
- Das SML-System prüft die Typ-Korrektheit eines Ausdrucks und inferiert dessen Typ.

Namen, Bindungen und Definitionen

- Mit einer Definition kann ein Wert an einen Namen gebunden werden.
- Mögliche Werte, die an Namen gebunden werden können, sind u.a.
 - Konstanten (Konstantendefinition) und
 - Funktionen(Funktionsdefinition).

Konstantendefinition

- Beispiel

```
- val zwei = 2;  
val zwei = 2 : int
```

- Damit wird die Konstante `zwei` deklariert und der Name `zwei` kann genauso wie die Konstante `2` verwendet werden:

```
- zwei + zwei;  
val it = 4 : int  
-zwei * 8;  
val it = 16 : int
```

Funktionsdefinition und Funktionsaufruf

- Beispiel

```
- fun malzwei(x) = x * 2;  
val malzwei = fn : int -> int
```

Damit wird die Funktion `malzwei` definiert.

- Der Wert des Namens `malzwei` ist die Funktion, die als Eingabe eine ganze Zahl erhält und das Doppelte dieser Zahl als Ausgabe liefert.
- Anstelle des Wertes der Funktion, die an den Namen `malzwei` gebunden wird, gibt SML die Abkürzung `fn` (fr Funktion) aus. Der Typ wird wie gewohnt ebenfalls bestimmt und ausgegeben.
- Nachdem eine Funktion definiert wurde, kann sie aufgerufen werden:

```
- malzwei(8);  
val it = 16 : int
```

Funktion als Wert — Anonyme Funktion

- Für SML ist eine Funktion ein Wert wie jeder andere auch
- Insbesondere kann das Definitionskonstrukt `val` verwendet werden:

```
val malzwei = fn x => x * 2;
```

- Die rechte Seite `fn x => x * 2` definiert eine **anonyme Funktion**. In Anlehnung an den λ -Kalkül wird `fn` oft “lambda” ausgesprochen.
- Diese anonyme Funktion wird an den Namen `malzwei` gebunden.
- Verwechseln Sie die SML-Konstrukte `fn` und `fun` nicht!
- Man kann anonyme Funktionen auch ohne Bindung verwenden:

```
- (fn x => x*x + 5*x - 4) 10;  
val it = 146 : int
```

Formale, aktuelle Parameter und Rumpf einer Funktion

- In der Funktionsdefinition

```
fun malzwei(x) = x * 2;
```

bzw. in der anonymen Funktion `fn x => x*2` ist

- `x` ein **formaler Parameter** und
 - `x * 2` der **Rumpf** der Funktion.
- Im Funktionsaufruf
`malzwei(10);`
ist `10` der **aktuelle Parameter** des Funktionsaufrufes.
 - Statt Parameter sagt man auch **Argument**.

Ausdrücke und Variablen

- Der Rumpf einer Funktion ist ein Beispiel eines Ausdrucks. Er enthält die formalen Parameter als Variablen.
- Ein Ausdruck, welcher keine Variablen enthält, kann ausgewertet werden. Am Ende der Auswertung, welche eine Folge von elementaren Auswertungsschritten ist, steht ein Wert.
- Es kann passieren, dass die Auswertung nicht terminiert oder zu einem Fehler führt (`div`). Dann hat der Ausdruck keinen Wert.

Weitere Beispiele von Funktionen

```
- fun loesung(a,b,c) = (~b + Math.sqrt(b*b-4.0*a*c))/(2.0*a);
val loesung = fn : real * real * real -> real
- loesung(1.0,~2.0,1.0);
val it = 1.0 : real
- loesung(2.0,~5.0,2.0);
val it = 2.0 : real
- fun verzinsen(kapital,zinssatz,jahre) =
      kapital * Math.pow(1.0+zinssatz,jahre);
val verzinsen = fn : real * real * real -> real
- verzinsen(0.01,0.03,200.0);
val it = 3.69355815216 : real
- verzinsen(0.01,0.03,400.0);
val it = 1364.23718234 : real
- verzinsen(0.01,0.03,1000.0);
val it = 68742402311.7 : real
-
```

Substitutionsmodell (1)

Zur Erklärung der Auswertung mit dem sog. **Substitutionsmodell** verwenden wir folgendes Beispiel:

```
- fun square(x) = x * x * 1.0;  
val square = fn : real -> real  
- fun dist(x,y) = Math.sqrt(square(x) + square(y));  
val dist = fn : real * real -> real
```

Wir wollen `dist(4.0-1.0, 6.0-2.0)` auswerten.

Substitutionsmodell (2)

- Zunächst werden die aktuellen Parameter ausgewertet:
 $4.0 - 1.0 \rightsquigarrow 3.0$ und $6.0 - 2.0 \rightsquigarrow 4.0$.
- Dann wird der Funktionsaufruf durch Einsetzen in den Rumpf ausgewertet:

$\text{dist}(4.0 - 1.0, 6.0 - 2.0) \rightsquigarrow \text{Math.sqrt}(\text{square}(3.0) + \text{square}(4.0))$

- Ebenso gilt:

$\text{square}(3.0) \rightsquigarrow 3.0 * 3.0 \rightsquigarrow 9.0$

$\text{square}(4.0) \rightsquigarrow 4.0 * 4.0 \rightsquigarrow 16.0$

$9.0 + 16.0 \rightsquigarrow 25.0$

$\text{Math.sqrt}(25.0) \rightsquigarrow 5.0$

Also $\text{Math.sqrt}(\text{square}(3.0) + \text{square}(4.0)) \rightsquigarrow 5.0$.

Und schließlich $\text{dist}(3.0, 4.0) \rightsquigarrow 5.0$.

Substitutionsmodell (3)

- Bei Funktionsaufrufen werden zunächst alle aktuellen Parameter ausgewertet
- und die so erhaltenen Werte werden für die formalen Parameter im Rumpf eingesetzt (“substituiert”).
- Dann wird die Auswertung mit dem substituierten Funktionsrumpf fortgesetzt.
- Man darf statt \rightsquigarrow informell auch das Gleichheitszeichen verwenden, obwohl strenggenommen z.B. $3.0 + 4.0$ und 7.0 zwei verschiedene Ausdrücke mit demselben Wert sind.
- Das Substitutionsmodell erklärt das Verhalten von funktionalen Sprachen, solange keine Seiteneffekte und Fehler auftreten.