

Info I - Programmierung in SML

Vorlesungsmitschrift von
Frank Schmidt

1. Einführung in SML

- **Imperative** Programmiersprachen: Pascal, C, Delphi, Java
Intuition: Ausführung von Anweisungen verändern Speicherzustand, Variablen bezeichnen „Behälter“ für Werte
- **Funktionale** Programmiersprachen
Variable sind gebunden an Ausdrücke
Ausdrücke werden ausgewertet

SML ist eine funktionale Programmiersprache mit einigen imperativen Konstrukten.

1.1 Berechnungsmodell von SML

Auswertung von Ausdrücken

Ein Ausdruck hat mehrere Aspekte:

1. hat einen **Typ**, oder auch nicht
 2. hat einen **Wert**, oder auch nicht
 3. erzeugt einen **Effekt**, oder auch nicht
- **Typ eines Ausdrucks:**
gibt an aus welchem Bereich der Wert des Ausdrucks kommt.
Ist der Typ des Ausdrucks *int*, so kann sich der Ausdruck nur zu einer ganzen Zahl auswerten.
Jeder Ausdruck muss mindestens einen Typ haben, sonst wird der Ausdruck zurückgewiesen.
Eigenschaften des Typsystems von SML: Korrektheit
Vom Typsystem berechneter Typ ist Typ des Wertes, wenn es einen gibt.
Nichttypbare Ausdrücke: `1+“Info1“` , `true orelse 2`
 - **Werte:**
Wohlgetypte Ausdrücke werden ausgewertet um den Wert zu bestimmen.
Ein Ausdruck hat keinen Wert,
 - wenn seine Auswertung nicht terminiert
 - wenn seine Auswertung eine Ausnahme auslöst, z.B. Division durch 0
 - **Effekte:**
Die Auswertung eines Ausdrucks kann einen Effekt erzeugen
 - wenn sie eine Ausnahme auslöst
 - wenn sie eine Änderung im Speicher versucht, oder
 - wenn sie Ein-/Ausgabe verursacht

Im ersten Teil der Vorlesung nur effekt-freie Programme, rein funktional
Typen sagen nichts über mögliche Effekte aus.

1.2 Werte und Typen

Programme rechnen über Werte. Jeder Wert kommt aus einem Bereich (domain).

Wir benutzen Typnamen als Bereichsnamen.

Unterschied zwischen Werten (Semantik) und ihren Darstellungen (Syntax):

z.B. 5, 101₂, V, IIII

„Konstanten“ in der Literatur, besser Konstantenbezeichnungen

Typen sind bestimmt durch:

Namen, den Bereich ihrer Werte, die Operationen, die man auf Werte des Typs anwenden kann.

Beispiele:

- **int**
(Darstellung von) Werte(n): 1, 5, 17, ~5
Operationen: +, -, *, div, mod, =, <, > (die durch die Operatoren... dargestellten Operationen)
- **real**
Werte: 1.5, 3.14, 5.2E7
Operationen: +, -, *, /, <, >
- **char**
Werte: #“A“, #“a“, #“\n“ (Ersatzdarstellung)
Operationen: =, < (gemäß Ordnung im ASCII-Code)
- **string**
Werte: „INFO1“, „Abs_10“, ””, ””
Operationen: =, ^, size, <
- **bool**
Werte: true, false
Operationen: not, orelse, andalso, if_then_else

1.2.1 Typüberprüfung

Beispiel: (1 < 2) andalso (5.0 / 2.5 > 2.0)
Typ dieses Ausdrucks?

Notation: e : t „Ausdruck e hat den Typ t“

- 1: 1 : int
- 2: 2 : int
- 3: (1 < 2) : bool (1., 2.)
- 4: 5.0 : real
- 5: 2.5 : real
- 6: 5.0 / 2.5 : real (4., 5.)
- 7: 2.0 : real
- 8: (5.0 / 2.5 > 2.0) : bool (6., 7.)
- 9: (1 < 2) andalso (5.0 / 2.5 > 2.0) : bool
(3., 8.)

wenn Ergebnis false, dann werte e2 aus
 wert von e1 bzw. e2 liefert Wert des gesamten if-Ausdrucks

- If $1 > 0$ then 1 else 1/0
 $(1 > 0) \Downarrow true$
 werte zu 1 aus

1.4 Namen, Deklarationen, Umgebungen

1.4.1 Namen und Deklarationen

Bisher: Ausdrücke über Konstanten (Bezeichnungen)

Jetzt: Namen, die an Werte (und anderes) gebunden werden und dann für diesen stehen können.

Warnung bezüglich Sprachgebrauch:

1. Es gibt ein syntaktisches Konstrukt, **Name** (identifiziert) mit einem vorgeschriebenen Aufbau.
2. Je nachdem, woran der Name gebunden ist, werden verschiedene Bezeichnungen für ihn verwendet
 - **Variable**, wenn er an einen Wert gebunden ist
 - **Typname**, Typkonstruktor (Harper) wenn er an einen Typ gebunden ist

Deklaration `val x : int = 4` (1)

Führt einen Namen, x , ein welcher den Typ *int* hat und bindet ihn an den Wert 4.

x kann jetzt in einem Ausdruck z.B. $x + 5$, auftreten und steht für 4, $x + 5$ wertet sich also zu 9 aus.

Ein Vorkommen eines Namens auf der linken Seite des „`=`“ in einer Deklaration heißt **definierend** (defining).

Ein Vorkommen eines Namens in einem Ausdruck heißt **angewandt** (applied).

Ein angewendetes Vorkommen eines Namens bezieht sich (in einem korrekten Programm) auf genau ein definierendes Vorkommen.

Einschub für imperative Programmierer

Pascal, Definition `var x : integer ; x := 4`

C, Java: `int x = 4`

Deklarationen führen einen Namen, x , ein, binden ihn an eine Speicherstelle und speichern den Wert darin ab.

Anschließende Zuweisungen an x überschreiben diesen Wert durch einen neuen Wert.

In funktionalen Sprachen: Die hergestellte Bindung ist **nicht veränderbar**.

Folgt eine weitere Definition von x

`val x : real = 1.0`

führt ein neues definierendes Vorkommen von x ein und bindet es an den Wert 1.0. Ab jetzt beziehen sich angewandte Vorkommen auf dieses neue definierende Vorkommen.

1.4.2 Umgebungen

Umgebungen führen Buch über Bindungen.

- **Wertumgebungen** über Bindungen an Werte P (rho)
- **Typumgebungen** über Bindung an Typinformation Γ (gamma)

Mathematisch gesehen sind Umgebungen Funktionen, die Namen abbilden auf Werte bzw. Typinformationen.

Deklaration (1) führt zu

$$P = \{x \mapsto 4\} \quad \Gamma = \{x \mapsto (val, int)\}$$

Abarbeitung einer weiteren Deklaration val $y : int = 6$ führt zu:

$$P = \left\{ \begin{array}{l} x \mapsto 4 \\ y \mapsto 6 \end{array} \right\} \quad \Gamma = \left\{ \begin{array}{l} x \mapsto (val, int) \\ y \mapsto (val, int) \end{array} \right\} \quad (2)$$

Bei angewandten Vorkommen von Namen wird in der „aktuellen“ Umgebung „nachgeschlagen“, an welchen Wert die Namen gebunden sind. Mathematisch gesehen wird die aktuelle Umgebung auf den Namen angewendet.

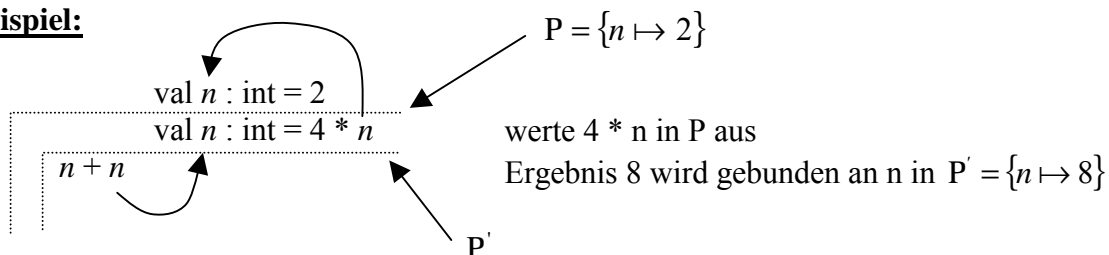
1.4.3 Gültigkeitsbereich

Mehrere Deklationen eines Namens sind zugelassen.

Frage: Auf welches dieser definierenden Vorkommen bezieht sich ein angewandtes Vorkommen?

Die Definition der Programmiersprache legt fest, in welchem Teil eines Programms sich alle angewandten Vorkommen auf ein bestimmtes Vorkommen beziehen. Dieser Teil heißt der **Gültigkeitsbereich** des definierenden Vorkommens.

Beispiel:



Der Gültigkeitsbereich des zweiten definierenden Vorkommens von n beginnt am Ende seiner Deklaration.

Bindungen können **ganz** oder **temporär überschrieben** werden.

Im SML-System gibt es einen Namen, *it*, an den die Werte von Ausdrücken gebunden werden, die ohne Bindung an einen Namen eingegeben werden.

temporär: let-Ausdrücke
 local-Deklarationen

let-Ausdruck:

$e = \text{let } \text{dec} \text{ in } \text{exp} \text{ end}$

Der Wert von e ist der Wert von exp in der Wertumgebung von dec .

Die definierenden Vorkommen in dec sind nur gültig in exp . Außerhalb von exp gelten die Bindungen aus der Umgebung die vorher gültig waren.

Beispiel:

$P = \{i \mapsto 7\}$

val $i : \text{int} = 7$ ←

let

$P' = \left\{ \begin{array}{l} i \mapsto 7 \\ m \mapsto 5 \end{array} \right\}$

val $m : \text{int} = 5$ ←

in $m + i$

end ← $P = \{i \mapsto 7\}$

Wert 12

1.4.4 Typdeklarationen

führen Namen für Typen ein.

Beispiel: Internationale Telefonnummer (49, 681, 3021000)

type land = int type intTelNr = land x amt x telnr
type amt = int
type telnr = int

val $p : \text{intTelNr} (\dots)$

Syntax:

type $\text{tycon}_1 = \text{typ}_1$
and
 \vdots
and $\text{tycon}_n = \text{typ}_n$

→ führt n Typnamen $\text{tycon}_1, \dots, \text{tycon}_n$ ein und bindet sie an die Typen $\text{typ}_1, \dots, \text{typ}_n$.

Erst nach Ende der Typdeklaration sind die Typnamen $\text{typ}_1, \dots, \text{typ}_n$ gebunden.

Eventuell angewandte Vorkommen von $\text{tycon}_1, \dots, \text{tycon}_n$ in $\text{typ}_1, \dots, \text{typ}_n$ beziehen sich nicht darauf.

$$\text{Typumgebung} : \left\{ \begin{array}{l} \text{tycon}_1 \mapsto (\text{type}, \text{typ}_1) \\ \text{tycon}_n \mapsto (\text{type}, \text{typ}_n) \end{array} \right\}$$

1.4.5 Typberechnung und Ausdrucksauswertung

jetzt Ausdrücke mit Namen

$$\begin{array}{ll} \Gamma \vdash e : t & \text{„in der Typumgebung } \Gamma \text{ hat } e \text{ den Typ } t\text{“} \\ P \vdash e \Downarrow v & \text{„in der Wertumgebung } P \text{ hat } e \text{ den Wert } v\text{“} \end{array}$$

Beispiel:

val $m : \text{int} = 3$
val $n : \text{int} = 5$
val $r : \text{real} = 2.0$

führt zur Typumgebung

$$\left\{ \begin{array}{l} m \mapsto (\text{val}, \text{int}) \\ n \mapsto (\text{val}, \text{int}) \\ r \mapsto (\text{val}, \text{real}) \end{array} \right\}$$

Wertumgebung

$$\left\{ \begin{array}{l} m \mapsto 3 \\ n \mapsto 5 \\ r \mapsto 2.0 \end{array} \right\}$$

Beispiel für eine Typberechnung:

