

3. Grundlagen von SML

3.1 Ausdrücke und Aufrufe

Grundkonzepte funktionaler Sprachen:

Funktionen und **Aufrufe**, **Ausdrücke**

keine Variablen mit Zuweisungen, **keine Ablaufstrukturen**,
keine Seiteneffekte (im Prinzip: aber E/A, Interaktion, Abbruch etc.)

Funktionale Sprachen sind **ausdrucksorientiert** (statt anweisungsorientiert):
Programme bestehen aus Definitionen und Ausdrücken (statt Anweisungen).
Typisch: bedingter Ausdruck statt bedingter Anweisung.

```
if a>b then a-b else b-a
```

Die Auswertung jedes Programmkonstruktes liefert einen Wert
(statt einen Effekt zu erzeugen, d.h. den Programmzustand zu ändern).

Vorlesung Funktionale Programmierung SS 2013 / Folie 301

Ziele:

Sprachen ohne Anweisungen verstehen

in der Vorlesung:

- Seiteneffekt-frei und Ausnahmen davon,
- Rolle bedingter Ausdrücke

Aufruf-Semantik Call-by-value (strikt)

Auswertung von Funktionsaufrufen (`mul (2, 4)`) und von Ausdrücken mit Operatoren (`2 * 4`) sind semantisch gleichwertig.

In SML haben alle Funktionen genau einen Parameter, ggf. ein Tupel.

Aufruf: (Funktionsausdruck Parameterausdruck)

Auswertung nach **call-by-value**, **strikte** Auswertung:

1. **Funktionsausdruck auswerten und Closure bestimmen**; Ergebnis ist eine Funktion mit einer Closure, in der die freien Variablen der Funktion gebunden werden.
2. **Parameterausdruck auswerten**; Ergebnis an den formalen Parameter der Funktion binden.
3. **Funktionsrumpf** mit Bindungen des formalen Parameters und der Closure **auswerten**; Ergebnis ist das Ergebnis der Ausdrucksauswertung.

Beispiel:

```
fun sqr x : int = x * x;
fun zero (x : int) = 0;
```

Auswertung modelliert durch **Substitution von innen nach außen**:

```
sqr (sqr (sqr 2)) => sqr (sqr (2 * 2)) => ...
zero (sqr (sqr (sqr 2))) => ...
```

Bedingte Ausdrücke werden nicht strikt ausgewertet!

Vorlesung Funktionale Programmierung SS 2013 / Folie 302

Ziele:

Aufruf-Semantik von SML verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- Aufruf-Semantik,
- Substitution von innen nach außen,
- nicht-strikte bedingter Ausdruck.

Aufruf-Semantik Call-by-need - lazy evaluation

Aufruf: (Funktionsausdruck Parameterausdruck)

Auswertung nach **call-by-name**:

1. und 3. wie oben

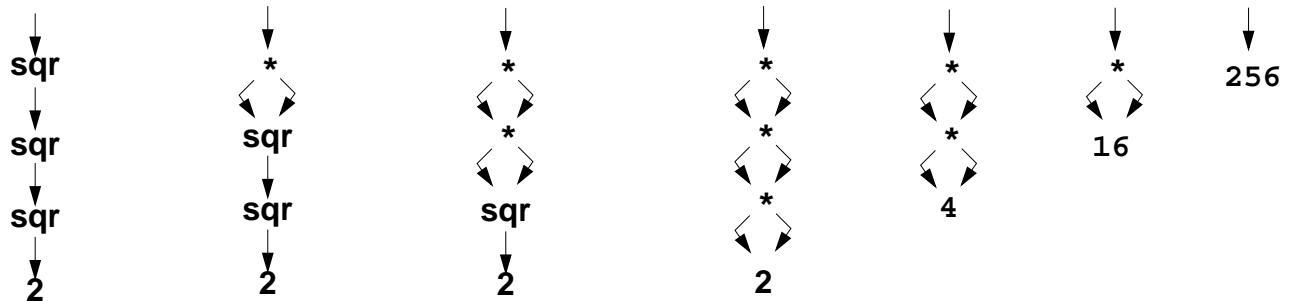
2. **Parameterausdruck** an jedes Auftreten des formalen Parameters im Funktionsrumpf substituieren (nach konsistenter Umbenennung der Namen im Parameterausdruck).

Beispiel: Auswertung modelliert durch **Substitution von außen nach innen**:

`sqr (sqr (sqr 2)) => (sqr (sqr 2)) * (sqr (sqr 2)) => ...`
`zero (sqr (sqr (sqr 2))) => 0`

* wird als Elementaroperator strikt ausgewertet.

Auswertung nach **call-by-need (lazy evaluation)**: wie **call-by-name**, aber der aktuelle Parameter wird **höchstens einmal ausgewertet** und sein Wert ggf. wiederverwendet. modelliert durch **Graph-Substitution von außen nach innen**:



Vorlesung Funktionale Programmierung SS 2013 / Folie 303

Ziele:

Call-by-need verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- alternative Aufruf-Semantik: Grundlage für lazy evaluation,
- Graph-Substitution,
- nicht-strikte bedingter Ausdruck ist formulierbar,
- nicht-endliche Datenstrukturen sind benutzbar,
- aber `zero(E)` liefert 0, sogar wenn die Auswertung von E nicht terminiert! Aufruf von zero wird nicht strikt ausgewertet.

3.2 Deklarationen in SML, Muster

Grundform von Deklarationen:

```
val Muster = Ausdruck
```

Der Ausdruck wird ausgewertet und liefert einen Wert *w*.

Das Muster ist hierarchisch aufgebaut aus

- **Bezeichnern**, die gebunden werden sollen; derselbe Bezeichner darf nicht mehrfach in einem Muster vorkommen;
- **Konstruktoren** für Datentypen, z. B. Tupel (,), Listen :: oder durch **datatype** eingeführte Konstruktoren, Zahlen;
- `_` anstelle eines Bezeichners (es wird nicht gebunden).

Der Wert *w* wird gemäß der Struktur des Musters zerlegt. Die Teilwerte werden an die entsprechenden Bezeichner gebunden.

```
fun foo x = (x, x);
```

```
val x = sqr 3;
```

```
val (c, d) = foo 42;
```

```
val h::_ = [1, 2, 3];
```

```
val (a, b) = (sqr 2, sqr 3);
```

```
val (x,y)::z = [foo 41, (3,4), (5,6)];
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 304

Ziele:

Muster mit mehrfachen Bindungen verstehen

in der Vorlesung:

An Beispielen werden Muster erläutert.

Funktionsdeklarationen

val-Deklaration einer rekursiven Funktion:

```
val rec Fac = fn n => if n <= 1 then 1 else n * Fac (n-1);
```

Kurzform für Funktionsdeklarationen:

```
fun Name Parametermuster = Ausdruck;
```

```
fun Fac n = if n <= 1 then 1 else n * Fac (n-1);
```

Funktionsdeklaration mit Fallunterscheidung über Muster:

```
fun FName Muster1 = Ausdruck1
  | FName Muster2 = Ausdruck2
  ...;
```

Die Muster werden nacheinander auf den Parameter angewandt, bis das erste trifft.

```
fun app (nil, lr) = lr
  | app (ll, nil) = ll
  | app (h::t, r) = h :: (app (t, r));
```

statt mit bedingten Ausdrücken über den Parameter:

```
fun app (ll, lr) = if ll = nil then lr else
                  if lr = nil then ll else
                  (hd ll) :: (app (tl ll, lr));
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 305

Ziele:

Lang- und Kurzformen verstehen

in der Vorlesung:

An den Beispielen werden die Formen erläutert.

Statische Bindung in SML

Auswerten einer `val`-Deklaration erzeugt eine **Menge von Bindungen** *Bezeichner -> Wert*, je eine für jeden Bezeichner im Muster.

In einer **Gruppe von Deklarationen**, die mit `and` verknüpft sind, gelten **alle Bindungen** der Gruppe **in allen Ausdrücken** der Gruppe (Algol-Verdeckungsregel)

```
fun f x = if p x then x else g x and
    g x = if q x then x else f x;
```

In **einzelnen Deklarationen**, die durch `;` getrennt werden, gelten die Definitionen **erst nach dem Ausdruck** der Deklaration.

Ausnahme: `val rec Muster = Ausdruck;` Bindungen gelten schon im Ausdruck.

Jede **einzelne Deklaration** oder Deklarationsgruppe bildet einen einzelnen **Abschnitt** im Sinne der Verdeckungsregeln: **Gleichbenannte Deklarationen verdecken Bindungen** des umfassenden (vorangehenden) Konstruktes:

```
val a = 42;
val b = 2 * a;
val a = 3;
val c = a + 1;
a + b * c;
```

`let`-Konstrukt fasst Deklarationen mit dem Ausdruck zusammen, in dem ihre Bindungen gelten:

```
let D1; D2; ... in Ausdruck end
```

`local`-Konstrukt fasst Deklarationen mit der Deklaration zusammen, in der ihre Bindungen gelten:

```
local D1; D2; ... in Deklaration end
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 306

Ziele:

Gültigkeit von Bindungen

in der Vorlesung:

An Beispielen wird erläutert:

- Verdeckung in Folgen von Deklarationen,
- rekursive Deklarationen,
- lokale Deklarationen.

3.3 Typen, Grundtypen

int und real:

real-Literale: `1.2E3` `7E~5`

binäre Operatoren: `+` `-` `*` `/`

unäres Minus: `~`

sind **überladen** für `int` und `real`.

Deshalb sind Typangaben nötig,
wenn der Typ der Operanden nicht eindeutig ist:

```
fun sqr (x : real) = x * x;
```

Funktionsbibliotheken `Int`, `Real`, `Math`:

```
Int.min (7, Int.abs i);
```

```
Math.sin (r) / r;
```

string:

Literale wie in C: `"Hello World!\n"`

Konkatenationsoperator: `^`

Funktionsbibliothek `String`

char:

Literale: `"a"` `"\n"`

bool:

Literale: `true` `false`

Operatoren: `orelse` `andalso` `not`

nicht strikt, d. h. Kurzauswertung (wie in C)

Vergleichsoperatoren: `=`, `<>`, `<`, `>`, `>=`, `<=`

Vorlesung Funktionale Programmierung SS 2013 / Folie 307

Ziele:

Grundtypen verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- Literale,
- Operatoren,
- Bibliotheken

Tupel, Records

Tupel:

```
val zerovec = (0.0, 0.0); val today = (5, "Mai", 2010);
```

Funktion mit Tupel als Parameter:

```
fun average (x, y) = (x+y)/2.0; average (3.1, 3.3);
```

Typdefinitionen:

```
type Vec2 = real * real;
fun trans ((a,b):Vec2, x):Vec2 = (a+x, b+x);
trans (zerovec, 3.0);
```

Records - Tupel mit Selektornamen:

```
type Date = {day:int, month:string, year:int};
val today = {year=2010, month="Mai", day=5}:Date;
fun add1year {day=d, month=m, year=y} =
  {day=d, month=m, year=(y+1)};
```

Benutzung von Selektorfunktionen:

```
#day today;
```

unvollständiges Record-Pattern:

```
fun thisyear ({year,...}:Date) = year = 1997;
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 308

Ziele:

Grundlegende Verbundtypen verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- Notation, Unterscheidung, Anwendung von Tupel- und Record-Typen.

Parametrisierte Typen (GdP-5.9)

Parametrisierte Typen (Polytypen, polymorphe Typen):

Typangaben mit **formalen Parametern, die für Typen** stehen.

Man erhält aus einem Polytyp einen konkreten Typ durch **konsistentes Einsetzen eines beliebigen Typs** für jeden Typparameter.

Ein Polytyp beschreibt die **Typabstraktion**, die allen daraus erzeugbaren konkreten Typen gemeinsam ist.

Beispiele in SML-Notation mit `'a`, `'b`, ... für Typparameter:

Polytyp	gemeinsame Eigenschaften	konkrete Typen dazu
<code>'a × 'b</code>	Paar mit Komponenten beliebigen Typs	<code>int × real</code> <code>int × int</code>
<code>'a × 'a</code>	Paar mit Komponenten gleichen Typs	<code>int × int</code> <code>(int->real) × (int->real)</code>

rekursiv definierte Polytypen:

<code>'a list = 'a × 'a list {nil}</code>	homogene, lineare Listen	<code>int list</code> <code>real list</code> <code>(int × int) list</code>
---	--------------------------	--

Verwendung z. B. in **Typabstraktionen** und in **polymorphen Funktionen**

Vorlesung Funktionale Programmierung SS 2013 / Folie 309

Ziele:

Polymorphe Typen wiederholen

in der Vorlesung:

Das Konzept wird an Beispielen kurz erläutert.

Polymorphe Funktionen (GdP-5.9a)

(Parametrisch) **polymorphe Funktion**:

eine Funktion, deren **Signatur ein Polytyp** ist, d. h. Typparameter enthält.

Die Funktion ist auf Werte eines jeden konkreten Typs zu der Signatur anwendbar.
D. h. sie muss unabhängig von den einzusetzenden Typen sein;

Beispiele:

Eine Funktion, die die Länge einer beliebigen homogenen Liste bestimmt:

```
fun length l = if null l then 0 else 1 + length (tl l);
```

polymorphe Signatur: 'a list -> int

Aufrufe: length ([1, 2, 3]); length ([(1, true), (2, true)]);

Funktionen mit Paaren:

```
fun pairself x = (x, x);
```

```
fun car (x, _) = x;
```

```
fun cdar (_, (x, _)) = x;
```

```
fun id x = x;
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 310

Ziele:

Polymorphe Funktionen wiederholen

in der Vorlesung:

Das Konzept wird an Beispielen kurz erläutert. Signaturen bestimmen.

Typinferenz

SML ist **statisch typisiert**. **Typangaben** sind meist **optional**.

Typinferenz:

Der **Typ T** eines Programmobjektes (benannt in Deklaration) oder eines Programmkonstruktes (unbenannter Ausdruck) wird aus dem Programmtext statisch ermittelt und geprüft.

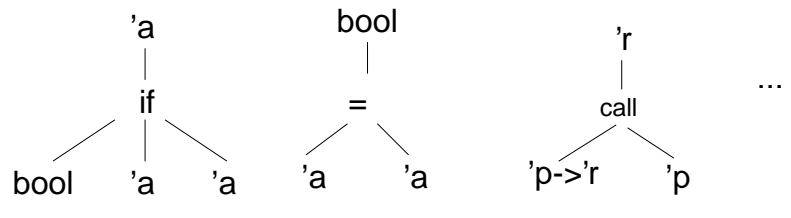
T ist der **allgemeinste Typ** (hinsichtlich der Typparameter), der mit den Operationen in der Deklaration bzw. in dem Ausdruck konsistent ist.

Verfahren:

Gleichungssystem mit Typvariablen vollständig aufstellen:

- Typ von Literalen ist bekannt.
- Typ von gebundenen Namen ist bekannt.
- Für hier definierte Namen n (in Mustern) $\text{Typ}(n)$ einsetzen
- Typregeln für jedes Programmkonstrukt auf Programmbaum systematisch anwenden, liefert **alle** Gleichungen.

einige Typregeln:



Gleichungssystem lösen:

- Widersprüche -> Typfehler
- Alle Typvariablen gebunden -> Typen der definierten Namen gefunden
- Einige Typvariablen bleiben offen -> der Typ ist **polymorph**

Vorlesung Funktionale Programmierung SS 2013 / Folie 311

Ziele:

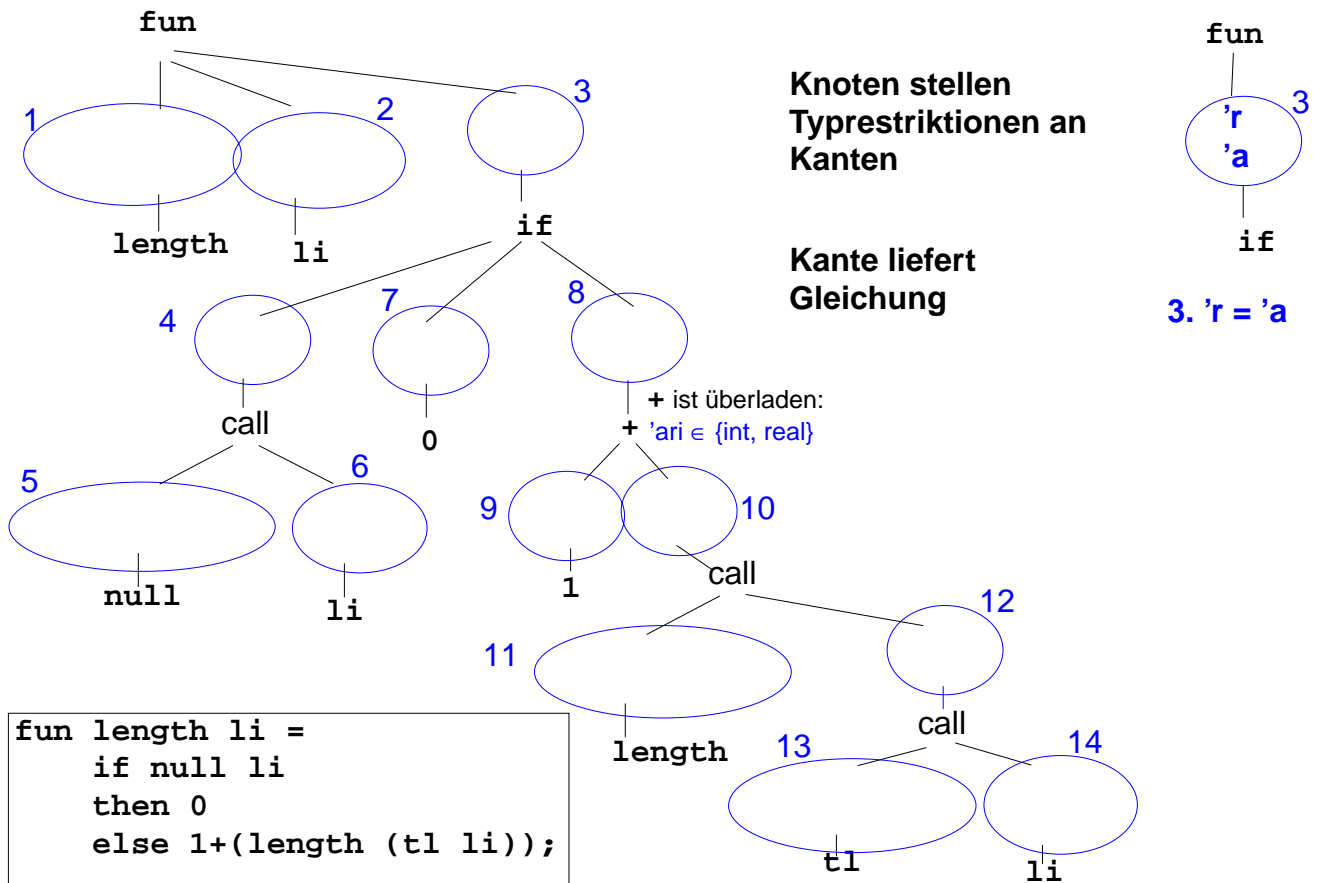
Typinferenz verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- allgemeinsten Typ,
- Gleichungssystem aufstellen,
- Gleichungssystem lösen.

Beispiel zur Typinferenz, Strukturbaum



Vorlesung Funktionale Programmierung SS 2013 / Folie 311a

Ziele:

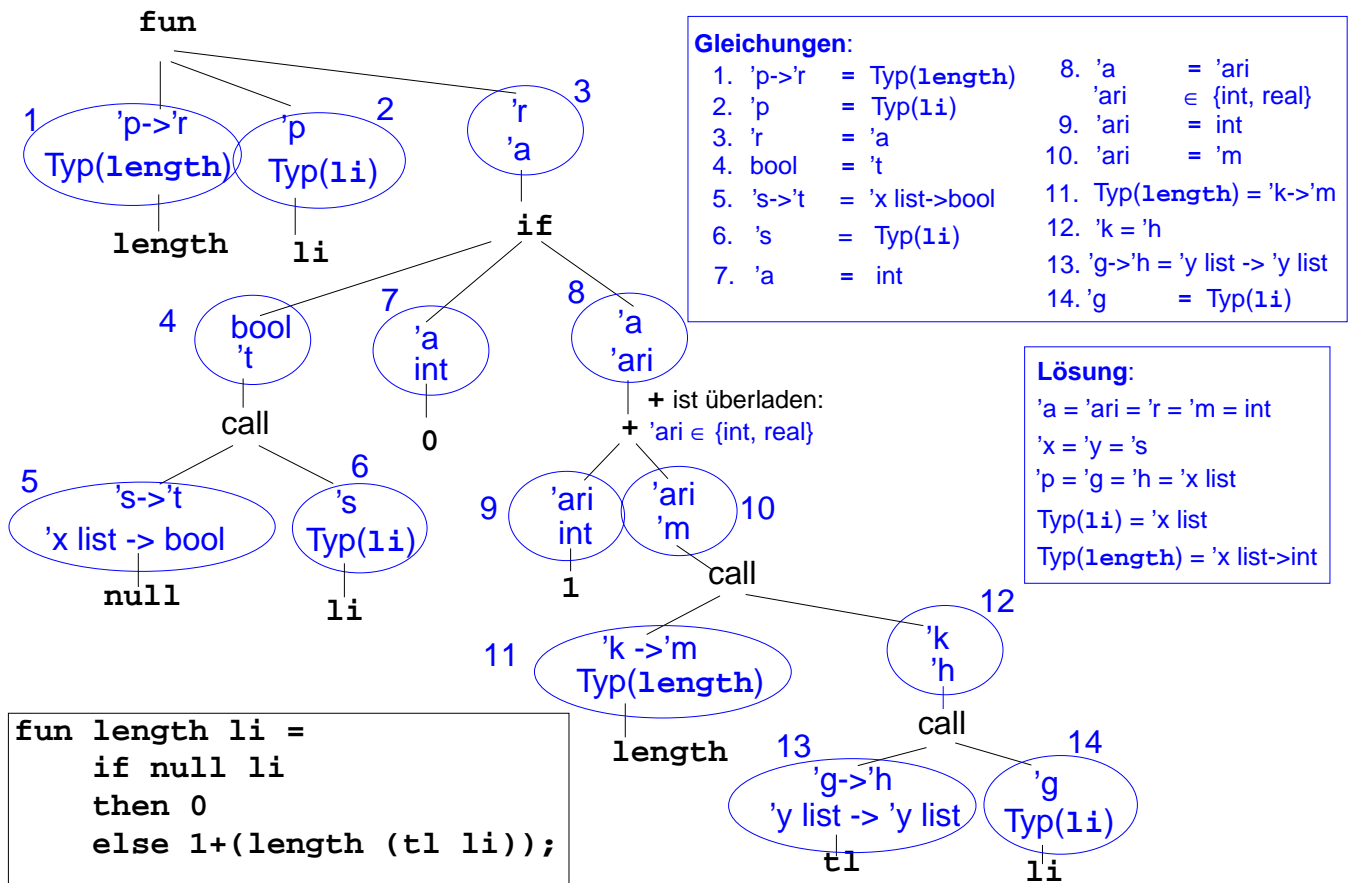
Typinferenz verstehen

in der Vorlesung:

Am Beispiel der length-Funktion wird erläutert:

- Baum erstellen,
- Typen aus Typregeln einsetzen,
- Gleichungssystem aufstellen,
- Gleichungssystem lösen.

Beispiel zur Typinferenz



Vorlesung Funktionale Programmierung SS 2013 / Folie 311b

Ziele:

Typinferenz verstehen

in der Vorlesung:

Am Beispiel der length-Funktion wird erläutert:

- Baum erstellen,
- Typen aus Typregeln einsetzen,
- Gleichungssystem aufstellen,
- Gleichungssystem lösen.