

14. Imperative Konstrukte

Bisher: rein funktionaler Teil von SML

Semantik benutzt nur Wert-Umgebungen, in denen die (unveränderliche) Bindung von Variablen an Werte verwaltet wird.

Auswertung eines Ausdrucks

- hat als Ergebnis einen Wert (eventuell)
- kreiert eventuell temporär neue Wertumgebungen, modifiziert aber nicht die Ausgangs-Wertumgebung

14.1 Referenzen

Ausführungsmodell wird erweitert um einen **Speicher(zustand)**

Der Speicher besteht aus einer Menge von **Zellen**, in denen wir Werte abspeichern können. Zellen werden eindeutig identifiziert durch **Referenzen**.

Es gibt eine Reihe von Sprachkonstrukten, mit denen man

- Werte in Zellen speichern kann, genannt **Zuweisung** (assignment)
- Werte in Zellen lesen kann, genannt **Dereferenzierung**.

Bei der Zuweisung an eine Referenz geht der vorherige Wert verloren. Deshalb Sprachgebrauch: „aktueller Wert“, „vorheriger Wert“

Reihenfolge der Ausführung dieser Sprachkonstrukte ist deshalb relevant
Operator für die **sequentielle Komposition**: ;

Vergleich mit imperativen Programmiersprachen (Pascal, C):

Pascal:

Konstanten-Bezeichner
Variablen von Basistypen
Variablen vom Zeigertyp

SML:

Variablen von Basistypen
ref t , t ∈ {int, real, char, bool}
ref ref t

Wertzuweisung:

$x := y$

Dereferenzierung: L-Wert R-Wert

L-Wert: Referenzen auf t

R-Wert: Wert vom Typ

R-Wert: automatische Dereferenzierung - explizite Dereferenzierung

14.1.1 Typen

neuer Typkonstruktor, **ref**,

typ ref = Typ aller Referenzen, deren Zellen Werte vom Typ typ aufnehmen können

14.1.2 Ausführungsmodell

Wertumgebung

- endliche Abbildungen von Variablen auf Werte
- Wertbereich erweitert um Referenzen

Speicher

- endliche Abbildungen von Referenzen auf Werte (inklusive Referenzen)

14.1.3 Operatoren

ref : $'a \rightarrow 'a \text{ ref}$

! : $'a \text{ ref} \rightarrow 'a$

:= : $'a \text{ ref} * 'a \rightarrow \text{unit}$

Allokation einer neuen Speicherzelle

Dereferenzierung einer Referenz

Zuweisung eines Wertes an eine Referenz

Operatoren können in Ausdrücken auftreten.

Die Auswertung eines Ausdrucks

- wird in (ρ, σ) , ρ = Wertumgebung, σ = aktueller Speicher, ausgewertet
- verändert eventuell den Speicher, nicht aber die Wertumgebung. Wir sagen, es gibt einen **Seiteneffekt** auf den Speicher.

Allokation: ref e

Typ: Hat e den Typ t, so hat ref e den Typ t ref

Semantik: Auswertung von ref e in (ρ, σ)

1. Auswertung von e in (ρ, σ) , Resultat v, eventuell neuer Speicher σ'
2. Erweiterung des Speichers um eine neue Zelle mit Referenz r, Speicherung von v in dieser Zelle, neuer Speicher σ''
3. Rückgabe von r als Ergebnis von ref e

Beispiel:

Wertumgebung

Speicher

val x = ref 0

$\{x \mapsto r_1\}$

$\{r_1 \mapsto 0\}$

val y = ref 3

$\left\{ \begin{array}{l} x \mapsto r_1 \\ y \mapsto r_2 \end{array} \right\}$

$\left\{ \begin{array}{l} r_1 \mapsto 0 \\ r_2 \mapsto 3 \end{array} \right\}$

val z = ref x

$\left\{ \begin{array}{l} x \mapsto r_1 \\ y \mapsto r_2 \\ z \mapsto r_3 \end{array} \right\}$

$\left\{ \begin{array}{l} r_1 \mapsto 0 \\ r_2 \mapsto 3 \\ r_3 \mapsto r_1 \end{array} \right\}$

val u = ref [1,4,7]

$$\left\{ \begin{array}{l} x \mapsto r_1 \\ y \mapsto r_2 \\ z \mapsto r_3 \\ u \mapsto r_4 \end{array} \right\}$$

$$\left\{ \begin{array}{l} r_1 \mapsto 0 \\ r_2 \mapsto 3 \\ r_3 \mapsto r_1 \\ r_4 \mapsto [1,4,7] \end{array} \right\}$$

val w = x

$$\left\{ \begin{array}{l} x \mapsto r_1 \\ y \mapsto r_2 \\ z \mapsto r_3 \\ u \mapsto r_4 \\ w \mapsto r_1 \end{array} \right\}$$

$$\left\{ \begin{array}{l} r_1 \mapsto 0 \\ r_2 \mapsto 3 \\ r_3 \mapsto r_1 \\ r_4 \mapsto [1,4,7] \end{array} \right\}$$

Zwei Namen für die Referenz r_1 : w und x, alias-Namen

ref-Muster:

val ref p = z

z gebunden an Referenz

p wird durch Musterabgleich an die Referenz gebunden

Dereferenzierung: !e

Typ: Hat e den Typ t ref, so hat !e den Typ t

Semantik: Auswertung von !e in (ρ, σ)

1. Auswertung von e in (ρ, σ) , Ergebnis Referenz r, eventuell neuer Speicher σ'
2. Ergebnis von !e ist der Inhalt der entsprechenden Zelle

fun !(ref x) = x definiert ! über ref Muster.

Zuweisung: $e_1 := e_2$

Typ: Gilt $e_1 : t \text{ ref}$ und $e_2 : t$, dann ist $e_1 := e_2 : \text{unit}$ (zur Erinnerung: der Typ unit hat nur einen Wert, nämlich ())

Semantik: Auswertung von $e_1 := e_2$ in (ρ, σ)

1. Auswertung von e_1 in (ρ, σ) , Ergebnis Referenz r, neuer Speicher σ'
2. Auswertung von e_2 in (ρ, σ') , Ergebnis v, neuer Speicher σ''
3. Änderung σ'' beim Argument r zu v, neuer Speicher σ'''
4. Gib () als Ergebnis von $e_1 := e_2$ zurück

Beispiel:

$$\begin{array}{l} x := !x + 1 \\ z := y \end{array} \Rightarrow \left\{ \begin{array}{l} r_1 \mapsto 1 \\ r_3 \mapsto \underline{r_2} \end{array} \right\} \text{ Änderung der Werte von x und w!}$$

14.1.4 Prozeduren

imperative Sprachkonstrukte in Funktionaldeklarationen

Die Anwendung einer solchen Funktion kann Seiteneffekte auf den Speicher haben und ein Ergebnis erbringen, welches abhängig vom aktuellen Speicher ist.

Semantik einer solchen Funktion ist im Allgemeinen keine mathematische Funktion von Argumenten auf Ergebnisse.

Deswegen sprechen wir von **Prozeduren**.

Beispiel:

`fun incr (t) = t := !t + 1`

14.2 while-Schleife

while p do e mit p:bool und e:t
ist ein Ausdruck vom Typ unit.

Semantik: Auswertung von while p do e in (ρ, σ)

1. Auswertung von p in (ρ, σ) , neuer Speicher σ'
2. Ist Ergebnis true,
Auswertung von e in (ρ, σ') , neuer Speicher σ'' ;
anschließend Auswertung von while p do e in (ρ, σ'')
3. Ist Ergebnis false,
so beende die Auswertung und gib () zurück

Der **Rumpf**, e, der Schleife while p do e sollte den Speicher so modifizieren, dass die Bedingung irgendwann zu false auswertet, ansonsten erhält man eine nicht terminierende Schleife!

Beziehung zur Endrekursion:

endrekursive Funktion fun f x = if p (x) then h (x) else f (g(x))

Die **Auswertung** einer while-Schleife - while p do e – verläuft wie die Auswertung wh () der Prozedur

fun wh () = if p then (wh (e;())) else ()

Die **Auswertung** von wh () und von while p do e

- terminiert entweder in beiden Fällen nicht oder
- sie erbringt in beiden Fällen das Ergebnis () und hat den gleichen Effekt auf den Speicher.

endrekursive Funktion \rightarrow while-Schleife

Gegeben: fun w (z) = if p (z) then w (f z) else h (z) mit Aufruf w (e)
(allgemeines Schema einer endrekursiven Funktion)

Transformation in eine while-Schleife:

z zu einer Referenz machen
val z = ref e;
while p (!z) do (z = f (!z); ())

Literatur zu diesem Thema: Bauer / Wössner: Algorithmische Sprachen und
 Programmentwicklung, Springer Verlag

14.3 Semantische Gleichheit

Prinzip:

Zwei Ausdrücke e_1, e_2 (gleicher Typ) sind **semantisch gleich**, wenn ihre Bedeutung mit Mitteln der Programmiersprache nicht unterschieden werden kann.

Präzisieren:

Gegeben sei ein „vollständiges“ Programm, in dem e_1 vorkommt.

Frage: Ändert sich das „beobachtbare“ Verhalten des Programms, wenn wir e_1 durch e_2 ersetzen?

Vollständiges Programm: Ausdruck vom Basistyp

Beobachtbares Verhalten:

1. Erbringt der Ausdruck einen Wert und wenn ja, dann welchen, oder erbringt er keinen Wert wegen Nichtterminierung oder nicht abgefangener Ausnahme?
2. Welches sind die sichtbaren Modifikationen des Speichers und/oder Ein-/Ausgabe?

Was ist irrelevant für dieses beobachtbare Verhalten?

1. Laufzeit und Platzverbrauch
2. interne Allokation von Referenzzellen
3. Die Namen von nicht abgefangenen Ausnahmen

Beispiele:

1. Im Kontext

val r = ref 0

val s = ref 0

sind die beiden Ausdrücke s und r nicht semantisch gleich

Begründung: Vollständiges Programm (s := 1; r!) \Downarrow 0
 Ersetzen von r durch s (s := 1; s!) \Downarrow 1
 Speicherveränderung ist in beiden Fällen identisch

2. Im Kontext

val r = ref 0

val s = r

sind die beiden Ausdrücke r und s semantisch gleich
 r und s sind an die gleiche Referenz gebunden

3. Im Kontext

val r = ref ()

val s = ref ()

sind die Ausdrücke $r!$ und $s!$ semantisch gleich, weil es nur einen Wert vom Typ unit gibt.
 r und s sind nicht semantisch gleich, weil sie an verschiedene Referenzen gebunden sind
 und weil man in SML Referenzen vergleichen kann.

if (r = r) then 1 else 2 \Downarrow 1

if (s = r) then 1 else 2 \Downarrow 2

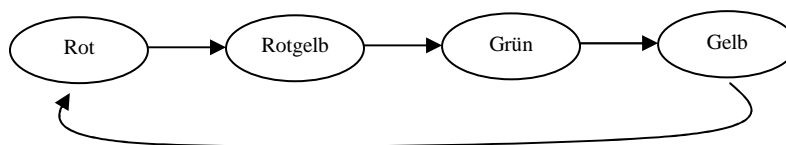
14.4 Zustandsbehaftete Programme

Wozu braucht man Referenzen?

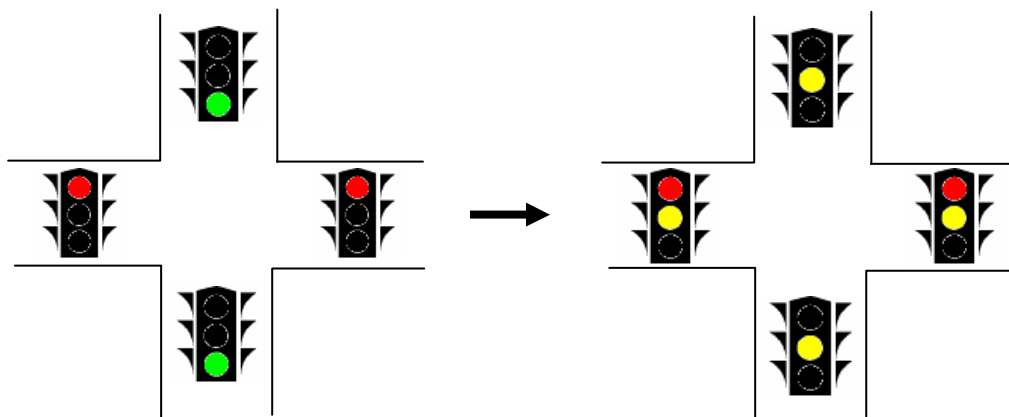
Manche Programme schreibt man auf natürliche Weise unter Benutzung von Speicher zur Darstellung eines Zustandes, vgl. Datenbanken.

Prinzipiell ist Zustand (Speicher) überflüssig. Man kann den Speicher als zusätzliches Argument und als zusätzliches Ergebnis durch die Funktionen „mitschleppen“. Das führt allerdings zu unnatürlichen Programmen und ineffizienter Ausführung.

Beispiel: Modellierung einer Ampel



Kreuzung mit 4 Ampeln: (<http://rw4.cs.uni-sb.de/~joba/Info1/Material/ampel.sml>)



Invarianten:

- sich kreuzende Straßen haben nicht gleichzeitig grün
- eine Kombination aus zwei Farben aus {Rotgelb, Gelb, Grün}

Erlaubt: alles andere

Programmiertechnik

fun neu_ampel ist vergleichbar mit einem **Konstruktor** einer Klasse in einem **objekt-orientierten** Programm.

Objekte (in unserem Fall Ampeln) als Instanzen einer Klasse:
bestehen aus: Zustand (Wert von ampel)

Funktionen zur Veränderung und zur Abfrage des Zustandes

Nur mit diesen Funktionen kann man auf den Zustand zugreifen. Dies heißt **Datenkapselung** (data encapsulation).

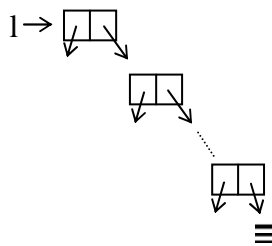
14.5 Imperative Datenstrukturen

Rein funktionale Datenstrukturen (Listen, Bäume, selbstdefinierte Datenstrukturen) sind „**unveränderlich**“. Wenn wir Funktionen auf sie anwenden, um ihre Struktur oder ihren Inhalt zu verändern, dann bleibt die Originaldatenstruktur erhalten, nur Teile werden **kopiert** und neu zusammengesetzt..

Problem: Platz- und Zeiteffizienz

Imperative Datenstrukturen können dies teilweise vermeiden.

Listen als eingebauter Datentyp

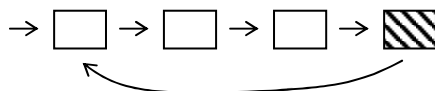


einseitig verkettete Liste,
Listenelemente nur in
Vorwärtsrichtung verkettet

sind immer „**azyklisch**“, enthalten keinen Rückverweis

Jetzt: imperativ einseitig verkettet, eventuell mit Rückverweis

Wie setzt man den **Rückverweis**?



```
datatype 'a sll = Sll of 'a sllcell ref (* zeigt auf ein Listenelem. *)
and 'a sllcell = Nil | Cons of 'a * 'a sll (* Typ der Listenelem. *)
```

```
fun cons (x, xs) = Sll (ref (Cons( x, xs)));
```

```
fun nill() = Sll ( ref Nil);
```

```
fun shd ( Sll( ref( Cons(x, xs))) ) = x;
```

```
fun stl ( Sll( ref( Cons(_, xs))) ) = xs;
```

```
fun settl( Sll( r as ref(Cons(x,_))), u) = (r:= Cons(x, u));
```

```
val acyclic= cons("a", cons("b", cons("c", nill() )));
```

```
val tail   = cons("d", nill());
```

```
val cyclic = cons("a", cons("b", cons("c", tail)));
```

```
val _ = settl( tail, cyclic);
```

14.6 Felder, Reihungen (arrays)

Imperative Datenstrukturen

- Endliche Folge von Werten **gleichen Typs**
- Komponenten des Feldes sind zugreifbar über **Index**, eine Zahl zwischen 0 und $n - 1$ (n = Zahl der Komponenten)
- Zugriff auf Komponenten kostet konstante Zeit (vgl. Zugriff auf Listenkomponenten (lineare Zeit))
- Komponenten können ersetzt werden durch neue Werte

Neuer Typkonstruktor: **array**

t **array** Typ der Felder mit Komponenten vom Typ t

Die wichtigsten Operationen:

- **array**: $\text{int} * 'a \rightarrow 'a \text{ array}$
array (n,v) kreiert ein Feld der Länge n, Indizes laufen von 0, ... , $n - 1$
Komponenten haben alle den Wert V.
- **length**: $'a \text{ array} \rightarrow \text{int}$
bestimmt die Größe (Größe ist nach der Kreation fest)
- **sub**: $'a \text{ array} * \text{int} \rightarrow 'a$
sub (arr,i) liefert die i-te Komponente, falls $0 \leq i \leq n - 1$, für n Größe von arr, sonst Ausnahme Subscript
- **update**: $'a \text{ array} * \text{int} * 'a \rightarrow \text{unit}$
update (arr,i,v) überschreibt die i-te Komponente von arr mit v, falls $0 \leq i \leq n - 1$, sonst Ausnahme Subscript

SML	Pascal, C, ???
sub (arr,i)	arr [i]
update (arr,i,v)	arr [i] := v

Beispiel: Prüfen, ob in einem gegebenen String alle 26 Buchstaben des Alphabets vorkommen

1.Lösung: benutzt Listen

chr: $\text{int} \rightarrow \text{char}$ liefert Zeichen zu ASCII-Code
ord: $\text{char} \rightarrow \text{int}$ liefert ASCII-Code zu Zeichen

Laufzeit:

- allMember: Rekursion über Kleinbuchstabenalphabet ~ 26 Schritte
- jeder Schritt ruft member auf
- member ist rekursiv in Listenargument

Ist die Listenlänge von der gleichen Größenordnung wie das Alphabet, dann hat allMember quadratische Laufzeit.

2.Lösung: mit Feldern

Feld a : $a_i = \text{true}$, wenn das Zeichen ch mit ASCII-Code $\text{ord}(ch) - \text{ord}(\#“a“) = i$ im String vorkommt.

Laufzeit: 2 Phasen:

1.Phase: FillandCheck: Rekursion durch Zeichenliste

2.Phase: Checkall: Rekursion durch Feld a

Beide sind linear, also ist die Laufzeit linear

14.7 Ein-/Ausgabe von Texten

Standard ML Basis Library enthält plattformunabhängige I/O-Funktionen

14.7.1 Ströme

Programmsicht von I/O ist:

Programm verfügt über ein oder mehrere

- Eingabeströme
 - Ausgabeströme
- verbunden mit Eingabegeräten / Ausgabegeräten

Strom: im allgemeinen unbegrenzte Folge von Zeichen

Eingabegeräte: Dateien, Tastatur,...

Ausgabegeräte: Dateien, Bildschirm,...

Strom wird kreiert und mit einem Gerät verbunden durch ein open:

openIn (S)

openOut (S) S String, Bezeichnung eines Geräts

Eingabeströme haben Typ instream

Ausgabeströme haben Typ ostream

Immer vorhanden sind die folgenden Ströme:

- Eingabestrom stdIn (i.a. verbunden mit Tastatur)
- Ausgabestrom stdOut (i.a. verbunden mit Bildschirm)
- Ausgabestrom stderr (i.a. verbunden mit Bildschirm) für Fehlermeldungen

Öffnen mit TextIO.openIn

.openOut

Schließen mit TextIO.closeIn

.closeOut

Anschließend sind die geschlossenen Ströme nicht mehr für I/O verfügbar.

Gepufferte Ein-/Ausgabe:

Ströme sind unbegrenzt lang. Zwischen Programm und Gerät ist ein Puffer

- **Eingabestrom von der Tastatur:** was der Tipper schon getippt hat, was aber noch nicht vom Programm gelesen wurde, steht im Puffer.
- **Ausgabestrom:** was das Programm ausgegeben hat, und was noch nicht zum Gerät geschickt wurde, steht im Puffer.

Operationen auf Eingabeströmen:

inputN (is,n)	liest die nächsten n-Zeichen von Eingabestrom is
inputLine (is)	liest die nächste Zeile von Eingabestrom is