3.4 Prozeduren und Funktionen

Im Allgemeinen sind Komponenten von Programmen oder logisch zusammengehörige Programmteile durch folgende Elemente charakterisiert:

- Eingabewerte
- Berechnungsteil
- Ausgabewerte

Prozedurale Abstraktion bedeutet, dass man solchen Programmteilen einen Namen gibt und den Berechnungsteil dann als „black box“ betrachtet.

Aspekte:

- Prozeduren beschreiben eine logisch zusammenhängende Berechnung.
- Prozeduren stellen ein wiederverwendbares Berechnungsmuster dar.
- Manchmal wird unterschieden zwischen Funktionen und Prozeduren:
  - Funktionen: berechnen ein Ergebnis
  - Prozeduren: berechnen kein explizites Ergebnis, sondern sie haben im Wesentlichen nur Seiteneffekte (möglichst nur auf Parameter, s.u.)
- Vorsicht: auch Funktionen können Seiteneffekte haben!
  - schlechter Programmierstil
  - schwer überschaubar
  - Reihenfolge wird bei der Auswertung von Ausdrücken relevant

  besser:
  - verbiete Seiteneffekte in Funktionen
  - übergebe alle zu verändernden Daten als Parameter (z.B. call-by-reference)

Die **Benutzung von Prozeduren** ist abhängig davon, ob diese ein Ergebnis liefern oder nicht:

- Bei (echten) Prozeduren entspricht der Prozeduraufruf einer Anweisung.
- Bei Funktionen kann der Funktionsaufruf in Ausdrücken vorkommen.

Bei einer **Prozedurdeklaration** werden üblicherweise eine Reihe von Dingen festgelegt:

- Typen der Parameter
- evtl. Übergabemechanismus für Parameter (s.u.)
- Ergebnistyp (bei Funktionen)

- Prozedurrumpf (Berechnungsteil, Block)

Ähnlich wie bei anderen Konstrukten gibt es in Programmiersprachen viele unterschiedliche Schreibweisen zur Definition von Prozeduren und Funktionen. Um die Bedeutung von Prozeduren genauer festzulegen, benutzen wir hier die folgende **Notation für Prozeduren und Funktionen**:

$$function\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) :\ \tau\ \texttt{is}\ S$$

mit $\tau_1, \ldots, \tau_n$ Parametertypen, τ Ergebnistyp und S Rumpf der Prozedur.

Bei Funktionen erfolgt im Rumpf üblicherweise die Festlegung des Ergebniswertes auf zwei mögliche Arten:

- durch eine spezielle Anweisung: `return(e)`, wobei `e` der Ergebniswert dieses Funktionsaufrufes ist.
- durch eine Zuweisung an den Prozedurnamen: f `:= e`. Hierbei wird der Prozedurname f wie eine lokale Variable im Rumpf behandelt.

Anwendung einer Prozedur:

- Bei einer echten Prozedur (d. h. der Ergebnistyp τ fehlt bei der Deklaration) ist der Prozeduraufruf $f(e_1, \ldots, e_n)$ syntaktisch eine Anweisung.
- Bei einer Funktion mit Ergebnistyp τ kann der Aufruf $f(e_1, \ldots, e_n)$ in Ausdrücken vorkommen, wo ein Wert vom Typ τ erwartet wird.
- Operationales Vorgehen:
  1. Deklariere $x_1, \ldots, x_n$ als (lokale) Variablen mit Initialwerten $e_1, \ldots, e_n$
  2. Führe den Rumpf S aus
  3. Bei Funktionen: ersetze den Funktionsaufruf durch das berechnete Ergebnis

Im Folgenden werden wir diese Semantik durch entsprechende Erweiterungen unseres Inferenzsystems formalisieren.

3.4.1 Operationale Semantik

Wir wollen nun die operationale Semantik von Prozeduren definieren. Hierzu müssen wir zwischen der Deklaration und Anwendung einer Prozedure trennen.

Prozedurdeklaration:

$$\langle E \mid M \rangle\ \ function\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \texttt{is}\ S\ \ \langle E; f() : func(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, \tau, S, E) \mid M \rangle$$

Anmerkungen:

- Eine Prozedurdeklaration hat keinen Effekt auf den Speicher.

- Die aktuelle Umgebung E der Deklaration muss wegen der lexikalischen Bindung bei der Prozedur gespeichert werden (vgl. Prozeduraufruf).
- τ kann bei einer echten Prozedur auch fehlen. In diesem Fall schreiben wir *proc* statt *func*.

Anwendung einer echten Prozedur:

$$\frac{\begin{array}{lcl} & \langle E \mid M\rangle \vdash^R e_1 : v_1 & \\ E \vdash^{lookup} p() : P & \vdots & \langle E' \ ; \ p() : P \mid M\rangle \ S' \ \langle E'' \mid M'\rangle \\ & \langle E \mid M\rangle \vdash^R e_n : v_n & \end{array}}{\langle E \mid M\rangle \ \ p(e_1, \ldots, e_n) \ \ \langle E \mid M'\rangle}$$

wobei

$$P = \mathit{proc}(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, S, E')$$

und

$$S' = \tau_1 \ x_1 = v_1; \ldots; \ \tau_n \ x_n = v_n; \ S$$

Das Hinzufügen der Prozedurdeklaration $p(); P$ zu E' vor Ausführung des Rumpfes ist wichtig, um rekursive Aufrufe zu ermöglichen.

Funktionsanwendung:

$$\frac{\begin{array}{lcl} & \langle E \mid M\rangle \vdash^R e_1 : v_1 & \\ E \vdash^{lookup} f() : F & \vdots & \langle E' \ ; \ f() : F \mid M\rangle \ S' \ \langle E'' \mid M'\rangle \quad \langle E'' \mid M'\rangle \vdash^R f : v \\ & \langle E \mid M\rangle \vdash^R e_n : v_n & \end{array}}{\langle E \mid M\rangle \vdash^R f(e_1, .., e_n) : v}$$

wobei

$$F = \mathit{func}(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, \tau, S, E')$$

und

$$S' = \tau_1 \ x_1 = v_1; \ldots; \ \tau_n \ x_n = v_n; \ \tau \ f; \ S$$

Anmerkungen:

- **Parameterübergabe**: zunächst müssen die *aktuellen* Parameterwerte ausgerechnet werden (in der Umgebung des Aufrufs mittels $\langle E \mid M\rangle \vdash^R e_j : v_j$), um dann die *formalen* Parameter x_j als lokale Variablen mit entsprechenden Initialwerten ($\tau_j \ x_j = v_j$) vor Ausführung des Prozedurrumpfes zu deklarieren (die hier spezifizierte Übergabeart ist also "call by value", s.u.).
- Der Rumpf S wird in der Umgebung der Deklaration abgearbeitet (lexikalische Bindung!), wobei die Deklaration der Prozedur für eventuelle rekursive Aufrufe eingeführt wird. Eine Modellierung einer dynamischen Bindung wie in Lisp wäre auch einfach möglich: dazu müsste man im Aufruf die aktuelle Umgebung lassen, d. h. E' durch E ersetzen.

- Bei Funktionen haben wir angenommen, dass diese keine Seiteneffekte haben (d. h. wir ignorieren den veränderten Speicher M'). Wenn wir auch mögliche Seiteneffekte von Funktionen modellieren wollten, müssten wir den veränderten Speicher in den Inferenzregeln für $\vdash^R$ berücksichtigen, wodurch wir allerdings eine feste Auswertungsreihenfolge bei Ausdrücken festlegen müssten.

- **Ergebnisrückgabe bei Funktionen**: Hierzu haben wir den Funktionsnamen f als lokale Variable deklariert ($\tau\ f$). Damit können wir beide Arten der Funktionswertrückgabe modellieren:
  - Falls in S die Zuweisung f `=` e vorkommt, wird hierdurch der Rückgabewert festgelegt.
  - Falls in S die Anweisung `return(`e`)` vorkommt, interpretieren wir diese als „f `=` e`; break`“

  Man sollte beachten, dass es hierzu wichtig ist, in der Umgebung zwischen der Funktionsdeklaration $f()$ und der lokalen Variablen f zu unterscheiden!

- **formale Parameter:** In den obigen Inferenzregeln haben wir diese als lokale Variablen deklariert, so dass Zuweisungen an diese im Rumpf S erlaubt sind, aber keine globalen Auswirkungen haben (im Gegensatz zur Parameterübergabeart “call by reference”, s.u.). Dies ist die Bedeutung von formalen Parametern in Java. Wenn man Zuweisungen an formale Parameter verbieten wollte, könnte man diese einfach als Konstanten deklarieren.

- **Rekursion** ist problemlos möglich, da die Prozedurdeklaration zur Umgebung des Rumpfes gehört (in älteren Sprachen wie Cobol oder Fortran war dies verboten)

- **indirekte Rekursion**: Dies ist der Fall, falls die Prozedur p die Prozedur q aufruft aber auch p aus q aufgerufen wird.

  Dies haben wir aus Vereinfachungsgründen weggelassen. Man könnte es jedoch auf zwei Arten ermöglichen:

  1. `forward`-Deklarationen: deklariere zunächst nur den Kopf der Prozedur, später dann den Rumpf (dies wurde z. B. in der Programmiersprache Pascal eingeführt).
  2. Sammle alle Deklarationen eines Blocks in einer Umgebung und speichere diese Umgebung in den Prozedurdeklarationen (dies ist üblich in modernen Programmiersprachen).

- **Schlüsselwortparameter** (z. B. Ada, Python, Ruby): Hierbei muss die Parameterübergabe nicht in der Reihenfolge der angegebenen formalen Parameter erfolgen, sondern Parameter können in beliebiger Reihenfolge durch die Angabe des Namens der formalen Parameter übergeben werden. Zum Beispiel werden durch

$$f(e_1, \ldots, e_k, x_{i_1} = e_{i_1}, \ldots, x_{i_l} = e_{i_l})$$

die ersten $e_1, \ldots, e_k$ an die ersten k formalen Parameter übergeben und die restlichen Parameter direkt mittels der Namen der formalen Parameter übergeben.

Vorteil: nicht alle Parameter müssen übergeben werden, sondern man kann default-Werte für einige Parameter verwenden.

Syntax von Prozeduren in Java (hier spricht man von „Methoden“ statt Prozeduren)

`public static` τ f`(`$\tau_1\, x_1, \ldots, \tau_n\, x_n$`) { `$S$` }`

Hierbei bedeutet `public`, dass der Prozedurname nach außen bekannt ist und `static`, dass die Prozedur auch ohne Objekte existiert (vgl. 4). Im Rumpf einer Funktion wird der Rückgabewert mit „`return` e`;`“ festgelegt.

Beispiel 3.13 (Aufsummieren aller Elemente eines Feldes)**.**

```
class XY {
  ...
  public static int sum(int[] a, int n) {
    int i, s = 0;
    for (i = 0; i <= n; i++) s += a[i];
    return s;
  }
  ...
    int[] pow2 = {1,2,4,8};
    System.out.println("Summe: " + sum(pow2, 3));
  ...
}
```

3.4.2 Parameterübergabemechanismen

In unserer bisherigen Modellierung sind die formalen Parameter lokale Objekte. Damit können die Parameter nicht verwendet werden, um globale Änderungen zu bewirken. Dies bedeutet, dass Prozeduren nur globale Effekte haben, indem diese globale bekannte Objekte verändern. Dies führt aber leicht zu undurchschaubaren Seiteneffekten von Prozeduren, die man nur durch Ansehen des Prozedurrumpfes erkennen könnte, was aber der Idee der prozeduralen Abstraktion widerspricht. Wünschenswert ist es, dass alle Effekte einer Prozedur durch den Prozedurkopf, d. h. die Parameter kontrolliert werden. Dies ist aber nur möglich, wenn man mittels der Parameter auch globale Effekte bewirken kann. Zu diesem Zweck gibt es in verschiedenen Programmiersprachen eine Reihe von Parameterübergabemechanismen, die wir im Folgenden vorstellen.

Wertaufruf (call-by-value): Dies ist der zuvor beschriebene Übergabemechanismus:

- berechne R-Wert des aktuellen Parameters

- initialisiere formalen Parameter mit diesem Wert
- Typeinschränkung: Typ des R-Wertes muss an den Typ des formalen Parameters anpassbar sein
- Nachteil: da Wertparameter bei der Übergabe kopiert werden, kann dies bei großen Datenstrukturen (z. B. Felder) aufwändig sein

Ergebnisaufruf (call-by-result):

- aktueller Parameter muss L-Wert haben
- formaler Parameter ist zunächst eine lokale, nicht initialisierte Variable in dieser Prozedur
- bei Prozedurende: kopiere den Wert des formalen Parameters in den aktuellen Parameter

Beispiel 3.14 (Ergebnisaufruf in Ada)**.**

```
procedure squareRoot(x: in real; y: out real) is ...
```

Hierbei wird `x` per Wertaufruf („`in`“) und `y` per Ergebnisaufruf („`out`“) übergeben. Aktuelle Aufrufe dieser Prozedur wären:

```
squareRoot (2.0 + z, r) -- ok;
squareRoot (z, 2.0 + r) -- nicht ok
```

Wert-Ergebnisaufruf (call-by-value-result): Dies bezeichnet die Kombination der beiden vorhergehenden Übergabearten:

- aktueller Parameter muss L-Wert haben
- initialisiere formalen Parameter mit R-Wert des aktuellen Parameters
- bei Prozedurende: rückkopieren wie bei Ergebnisaufruf

Referenzaufruf (call-by-reference):

- aktueller Parameter muss L-Wert haben
- formaler Parameter entspricht Zeiger auf aktuellen Parameter
- Initialisierung bei Prozeduraufruf: die bisherige Regel zum Prozeduraufruf muss leicht abgewandelt werden:
  falls der Parameter x_j per Referenzaufruf übergeben wird und

$$\langle E \mid M \rangle \vdash^L e_j : l_j$$

  dann wird x_j vor dem Rumpf als *Referenzvariable* eingeführt:

$$S' = \ldots ref(\tau_j)\ x_j = l_j; \ldots$$

Die Deklaration solcher Referenzvariablen wird durch folgende neue Regel behandelt:

$$\langle E \mid M \rangle \;\; ref(\tau)\ x = l \;\; \langle E\ ;\ x : ref(l, \tau) \mid M \rangle$$

- Referenzvariablen werden wie Zeiger behandelt, aber mit impliziter Dereferenzierung bei ihrer Benutzung:

$$\frac{E \vdash^{lookup} x : ref(l, \tau)}{\langle E \mid M \rangle \vdash^{L} x : l}$$

$$\frac{E \vdash^{lookup} x : ref(l, \tau)}{\langle E \mid M \rangle \vdash^{R} x : M(l)}$$

- bei typisierten Sprachen muss der Typ des aktuellen Parameters **äquivalent** zum Typ des formalen Parameters sein

  Zum Begriff der **Typäquivalenz**:
  Hierfür gibt es unterschiedliche Definitionen in verschiedenen Programmiersprachen:

  Namensgleichheit: Typen sind äquivalent $\Leftrightarrow$ Typnamen sind identisch.

  Strukturgleichheit: Typen sind äquivalent, falls diese strukturell gleich sind. Zum Beispiel können Verbundtypen mit unterschiedliche Verbundnamen, aber gleichen Komponententypen äquivalent sein.

  In den meisten Programmiersprachen ist die Namensgleichheit üblich. In diesem Fall sind im Beispiel

```
type ziffer = [0..9];
type note   = [0..9];
```

  die Typen `ziffer` und `note` zwar strukturell gleich, aber bezüglich der Namensgleichheit nicht typäquivalent.

- Potenzielles Problem beim Referenzaufruf:
  Aliasing: Dies bedeutet, dass es unterschiedliche Namen für gleiche Objekte (L-Werte) geben kann.

```
procedure p(x, y: in out integer) is B
```

  Dann steht beim Aufruf „`p(i, i)`“ sowohl `x` als auch `y` für die gleiche Speicherzelle, was die Gefahr von unüberschaubaren Seiteneffekten hat. Wenn z. B. in B die Anweisungen

```
x := 0;
y := 1;
while y > x do ...
```

vorkommen, dann ist bei dem Aufruf „`p(i, i)`“ die Bedingung `y > x` nie erfüllt, was vielleicht vom Implementierer dieser Prozedur nicht intendiert ist.

Der Referenzaufruf kann benutzt werden, um z. B. eine universelle Prozedur `swap` zum Vertauschen des Inhaltes zweier Variablen zu definieren (hier: Modula-2, Kennzeichnung des Referenzaufrufs durch „`var`“):

```
procedure swap(var x: integer; var y: integer);
  var z: integer;
begin
  z := x; x := y; y := z
end swap;
⋮
i := 2; a[2] := 42;
swap(i, a[i]);
```

Nach dem Aufruf von `swap` hat `i` den Wert 42 und `a[2]` den Wert 2.

Namensaufruf (call-by-name): Algol-60, Simula-67

- Formale Parameter werden im Rumpf *textuell* durch aktuelle Parameter ersetzt. Hierbei werden jedoch Namenskonflikte gelöst (im Gegensatz zur einfachen Makro-Expansion).

  Beispiel für einen Namenskonflikt:

  ```
  int i; int[] a;
  procedure p(int x) is int i; i = x; end;
  ⋮
  p(a[i]);
  ```

  Das „`int i`“ in der Prozedur und das `i` in der Parameterübergabe verursachen einen Namenskonflikt, wenn man im Rumpf der Prozedur `x` einfach durch `a[i]` ersetzen würde.

  Lösung: Benenne in `p` **vor dem Aufruf** lokale Variablennamen so um, dass diese nicht in Konflikt mit den übergebenen aktuellen Parametern stehen.

- Implementierung des Namensaufrufs:
  - Erzeuge für jeden aktuellen Parameter Prozeduren, die die L- und R-Werte dieses Parameters berechnen.
  - Ersetze jedes Vorkommen eines formalen Namenparameters im Rumpf durch den entsprechende Parameterprozeduraufruf.
- Somit werden die aktuellen Parameter bei *jedem* Zugriff auf die entsprechenden formalen Parameter ausgewertet! Dies ermöglicht die folgende Berechnung, die auch als **Jensen-Trick** aus Algol-60 bekannt ist (hierbei werden

Wertparameter durch „`value`“ gekennzeichnet; alle anderen Parameter sind immer Namenparameter):

```
real procedure sum (i, n, x, y);
  value n;
  integer i, n;
  real x, y;
begin
  real s; s := 0;
  for i := 1 step 1 until n do s := s + x * y;
  sum := s;
end;
```

Aufruf: `sum(i, n, a[i], b[i])` $\rightsquigarrow$ `a[1] * b[1] + ··· + a[n] * b[n]` (wegen wiederholter Auswertung von `a[i]` und `b[i]`!)

Prozeduren als Parameter:

- in manchen Sprachen zulässig:

```
procedure p(x: int; procedure r(y: int)) is ... r(...) ... end;
```

- Parameterübergabe: call-by-value/-reference, d. h. übergeben wird die (Code-)Adresse der aktuellen Prozedur (und die Umgebung bei lexikalischer Bindung!), die nicht veränderbar ist.
- Pascal: typunsicher: Parametertypen von Prozeduren als Parametern sind unbekannt und werden daher nicht überprüft.
- Modula-3: typsicher und Zuweisung an Prozedurvariablen (z. B. Prozeduren in Records) sind zulässig.
- Eine umfassende Behandlung erfolgt in den funktionalen Programmiersprachen (vgl. Kapitel 5).

3.4.3 Einordnung der Parameterübergabemechanismen

- viele Programmiersprachen bieten mehrere Übergabemechanismen für Prozedurparameter an, z. B.
  - Pascal/Modula: Wert- und Referenzaufruf
  - Ada: Wert- (`in`), Ergebnis- (`out`), Wert/Ergebnis- (`in out`) und Referenzaufruf für Felder
  - Fortran: Referenzaufruf für Variablen, sonst Wertaufruf
  - C: nur Wertaufruf
  - Java: nur Wertaufruf, aber beachte: Objektvariablen (Felder, Verbunde) sind Referenzen, daher wird bei Aufruf nur die Referenz übergeben, das Objekt selbst wird nicht kopiert.

```
public static void first99(List f) {
    f.elem = 99; // globale Aenderung
    f = null;    // lokale Aenderung
}
⋮
List head = new List();
head.elem = 3;
first99(head);  // danach gilt head.elem = 99
```

- Namensaufruf (Algol-60, Simula-67): eher historisch wichtig, aber der Namensaufruf ist in abgewandelter Form in der funktionalen Programmierung relevant und wird auch in der Sprache Scala angeboten (dies wird später noch erläutert)
- heute: überwiegend Wert- und Referenzaufruf:
  Wertaufruf $\approx$ Eingabeparameter
  Referenzaufruf $\approx$ Ausgabeparameter bzw. große Eingabeobjekte
- Beachte: Wertaufruf für Zeiger oder Referenzen entspricht (fast) einem Referenzaufruf
- Funktionale Programmiersprachen bieten auch den „faulen Aufruf“ (call-by-need) als Übergabeart (vgl. Kapitel 5).