2.2.6 Datentypen

In der bisherigen Semantik imperativer Programme ist noch die Bedeutung der Basistypen (z. B. `Int`, `Bool`) und ihrer Operationen offen, d.h. es ist noch nicht genau spezifiziert, wie z.B. die semantischen Operationen o in "$\sigma \vdash e_1\ op\ e_2 : v_1\ o\ v_2$" definiert sind. Ähnlich wie die Semantik der Sprache ist es wünschenswert, auch diese grundlegenden Operationen unabhängig von einer konkreten Implementierung zu spezifizieren. Eine mögliche Technik hierfür wollen wir nachfolgend diskutieren.

Eine Methode zur *implementierungsunabhängigen* Spezifikation von Datentypen und ihren Operationen sind *abstrakte Datentypen*. „Abstrakt" bedeutet hierbei, dass die Implementierung nicht festgelegt ist, sondern nur die Bedeutung der Operationen wichtig ist.

Definition 2.6 (Abstrakter Datentyp (ADT))**.** *Ein* ***ADT*** *besteht aus einer* ***Signatur*** Σ*, welche eine Menge von* ***Operatoren*** $f : \tau_1, \ldots, \tau_n \to \tau \in \Sigma$ *ist. Hierbei ist* $n \geq 0$ *und* $\tau_1, \ldots, \tau_n, \tau$ *sind Typen. Der Operator* f *bildet somit Elemente aus* $\tau_1, \ldots, \tau_n$ *auf ein Element aus* τ *ab.*
Ein Spezialfall ergibt sich für $n = 0$*: Dann ist* $c :\to \tau \in \Sigma$ *und wir nennen* c *eine* ***Konstante*** *(in Programmiersprachen oft auch als* ***Literal*** *bezeichnet).*

Beispiel 2.6 (Ein ADT für Wahrheitswerte)**.**

```
datatype Bool
  true, false  :             → Bool
  ¬            : Bool        → Bool
  ∨, ∧         : Bool, Bool  → Bool
  =, ≠ , <, >  : Int, Int    → Bool
end
```

Beispiel 2.7 (Ein ADT für ganze Zahlen)**.**

```
datatype Int
  ..., -2, -1, 0, 1, 2, ... : → Int
  +, -, *, div, mod         : Int, Int → Int
end
```

Nun können wir (wohlgeformte) Ausdrücke über Datentypen definieren. Dazu nehmen wir an, dass X eine *Menge getypter Variablen* (disjunkt zu Operatoren und Konstanten) ist, d.h. $x : \tau \in X$ bedeutet „x ist Variable vom Typ τ".

Definition 2.7 (Ausdrücke/Terme)**.** *Die Menge aller Ausdrücke/Terme bzgl.* Σ *wird mit* $T(\Sigma, X)$ *bezeichnet. Ein Ausdruck/Term vom Typ* τ *kann Folgendes sein:*

Variable $x \in T(\Sigma, X)$*, falls* $x : \tau \in X$

Konstanten $c \in T(\Sigma, X)$, *falls* $c :\to \tau \in \Sigma$

zusammengesetzter Term $f(t_1, \ldots, t_n) \in T(\Sigma, X)$, *falls* $f : \tau_1, \ldots, \tau_n \to \tau \in \Sigma$ *und* $t_i \in T(\Sigma, X)$ *vom Typ* τ_i *(für alle* $i \in \{1, \ldots, n\}$*)*

Weiterhin bezeichnen wir einen Term ohne Variablen auch als ***Grundterm****.*

Beispiel 2.8 (Terme)**.**
Terme vom Typ `Int`:

-2 $\quad 1 + 3 * 4$ (Infixdarstellung für $+(1, *(3, 4))$) $\quad$ $3 \bmod x$, falls $x : Int \in X$

Terme vom Typ `Bool`:

`true` $\quad$ `false` $\vee$ `true` $\quad x \neq 0 \wedge x \neq 1$

Bisher haben wir nur die Syntax von Ausdrücken definiert. Was ist aber die Bedeutung dieser Ausdrücke? Intuitiv sollte $1 + 2$ die gleiche Bedeutung wie 3 haben. Zu diesem Zweck definieren wir die Bedeutung durch Gleichungen, um von einer konkreten Implementierung zu abstrahieren.

Definition 2.8 (Gleichung)**.** *Eine* ***Gleichung*** $t_1 = t_2$ *ist ein Paar von Termen gleichen Typs. Falls in einer Gleichung Variablen vorkommen, bezeichnet diese Gleichung ein Schema, das für alle Gleichungen steht, bei denen Variablen durch passende Werte ersetzt werden können (dies bezeichnet man auch als* ***Instanz*** *des Schemas).*

Beispiel 2.9 (Gleichungen)**.**
Gleichungen für `Bool` (mit b : `Bool` $\in X$)

$$\begin{aligned} \neg true &= false \\ \neg false &= true \\ true \wedge b &= b \\ false \wedge b &= false \\ true \vee b &= true \\ false \vee b &= b \end{aligned}$$

Gleichungen für `Int` (mit x : `Int` $\in X$)

$$\begin{aligned} 0 + x &= x \\ x + 0 &= x \\ 1 + 1 &= 2 \\ 1 + 2 &= 3 \\ &\vdots \end{aligned}$$

Ausrechnen von Gleichheiten Wir stellen die üblichen Gleichungsaxiome als Inferenzregeln dar:

- Reflexivität:
$$x = x$$
- Symmetrie:
$$\frac{x = y}{y = x}$$
- Transitivität:
$$\frac{x = y \quad y = z}{x = z}$$
- Kongruenz:
$$\frac{x_1 = y_1 \quad \ldots \quad x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}$$
(für alle Operatoren f)
- Axiom:
$$t_1 = t_2$$
(falls $t_1 = t_2$ Gleichung aus der Spezifikation ist, d.h. Gleichungen des ADTs sind Axiomenschemata)

Beispiel 2.10. Wir wollen zeigen, dass $\neg(true \wedge false) = true$ ableitbar ist:

$$\frac{\dfrac{true \wedge false = false \text{ (Axiom)}}{\neg(true \wedge false) = \neg(false) \text{ (Kongruenz)}} \qquad \neg false = true \text{ (Axiom)}}{\neg(true \wedge false) = true \text{ (Transitivität)}}$$

Somit können wir das **Ausrechnen** von Ausdrücken mit Gleichheitsinferenzen beschreiben. Ein Problem dabei ist, dass es nicht klar ist, was das Berechnungsergebnis sein soll. Zum Beispiel ist auch
$$\neg(true \wedge false) = \neg(false \wedge true)$$
ableitbar, wobei intuitiv der Ausdruck nicht ausgerechnet wurde, weil die rechte Seite vereinfacht werden kann. Um „vereinfachte“ Terme zu charakterisieren, definieren wir den Begriff der Konstruktoren.

Definition 2.9 (Konstruktoren)**.** *Eine Menge C von Konstruktoren eines ADTs τ ist eine kleinste Menge C von Operationen mit dem Ergebnistyp τ, sodass für alle Grundterme t vom Typ τ gilt: es existiert ein Grundterm s mit:*

- *$t = s$ ist ableitbar, und*
- *s enthält nur Operationen aus C.*

Beispiel 2.11. Der ADT `Bool` hat die Konstruktoren *true* und *false*.
Der ADT `Int` hat die Konstruktoren ..., -2, -1, 0, 1, 2, ...

Für ADTen mit Konstruktoren besitzen wir folgende Intuition:

- Konstruktoren erzeugen Daten
- andere Operationen sind **Selektoren** oder **Verknüpfungen**
- Ausrechnen: Finden eines äquivalenten („gleichen“) Konstruktorterms

Diese Definition ist zwar unabhängig von einer konkreten Implementierung, aber dies ist noch keine konstruktive Rechenvorschrift, denn auf Grund der Definition mit einem Inferenzsystem ist das „Ausrechnen“ nur aufzählbar, aber nicht entscheidbar.

Beispiel 2.12. Listen von ganzen Zahlen

```
datatype IntList
  nil  : → IntList
  cons : Int,IntList → IntList
  head : IntList → Int
  tail : IntList → IntList
  empty: IntList → Bool
equations a:Int, l:IntList
  head(cons(a,l))  =  a
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Die Konstruktoren des ADT `IntList` sind `nil` und `cons`. Nach Definition 2.9 wäre zusätzlich auch `tail` ein Konstruktor, denn `tail(nil)` ist nicht reduzierbar zu einem Term, der nur aus `nil` und `cons` aufgebaut ist. Allerdings wird üblicherweise `tail(nil)` als Fehlerwert angesehen und ist daher kein Wert des ADT `IntList`.

Beispiel 2.13. Listen von Wahrheitswerten: diese könnte man wie `IntList` definieren, wobei überall `Int` durch `Bool` ersetzt wird:

```
datatype BoolList
  nil  : → BoolList
  cons : Bool,BoolList → BoolList
  head : BoolList → Bool
  tail : BoolList → BoolList
  empty: BoolList → Bool
equations a:Bool, l:BoolList
  head(cons(a,l))  =  a
```

```
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Wie man hierbei sieht, ist die Definition von Listen unabhängig von der Art der Elemente. Daher ist es sinnvoll, vom Typ der Elemente zu abstrahieren und diesen als Parameter des ADT zuzulassen. In diesem Fall spricht man auch von einem **parametrisierten ADT** $\tau(\alpha_1, \ldots, \alpha_n)$, wobei die Variablen α_i **Typparameter** sind, die in der ADT-Definition anstelle von Typen vorkommen können. Eine **Instanz eines parametrisierten ADTs** wird mit $\tau(\tau_1, \ldots, \tau_n)$ bezeichnet, wobei $\tau_1, \ldots, \tau_n$ Typen sind. Diese Instanz steht für die textuelle Ersetzung jedes α_i durch τ_i in der ADT-Definition.

Beispiel 2.14. Parametrisierte Listen `List(`α`)`

```
datatype List(α)
  nil  : → List(α)
  cons : α,List(α) → List(α)
  head : List(α) → α
  tail : List(α) → List(α)
  empty: List(α) → Bool
equations a:α, l:List(α)
  head(cons(a,l))  =  a
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Damit ist `List(Int)` äquivalent zu obiger Definition von `IntList`.

Datentypen in Programmiersprachen

Allgemein gilt, dass ein Typ in einer Programmiersprache einer Implementierung eines ADTs entspricht. Sind alle Konstruktoren Konstanten, so heißt dieser Typ **Grundtyp**, ansonsten **zusammengesetzter Typ**.
Viele funktionale Programmiersprachen erlauben eine direkte Implementierung vieler ADTen:

- Die Typdefinition erfolgt durch Angabe der Konstruktoren.
- Die Gleichungen können direkt angegeben werden. Dies ist allerdings nur mit folgender Einschränkung möglich: in der linken Seite $f(t_1, \ldots, t_n)$ darf f kein Konstruktor sein und die Terme $t_1, \ldots, t_n$ dürfen nur Konstruktoren oder Variablen enthalten.

Beispiel 2.15. Parametrisierte Listen in Haskell

```
data List a = Nil | Cons a (List a)

empty Nil         = True
empty (Cons x xs) = False

head (Cons x xs) = x

tail (Cons x xs) = xs
```

2.3 Bindungen und Blockstruktur

Programme bestehen aus bzw. manipulieren verschiedene Einheiten (Entitäten), z. B. Variablen, Prozeduren, Klassen, oder Ähnliches. Die konkreten Eigenschaften dieser Programmeinheiten werden durch **Attribute** beschrieben.

Beispiel 2.16. Attribute einzelner Einheiten:

- Variablen: Name, Adresse, Typ, Wert

  Die Adresse ist beispielsweise wichtig für sharing, unter anderem bei `var`-Parametern (Pascal) oder Zeigern. Zusätzlich könnte z. B. der Gültigkeitsbereich o. ä. auch als Attribut relevant sein.

- Prozeduren: Name, Parametertypen, Übergabemechanismen, ...
- Klassen: Name, Instanzvariablen, Klassenvariablen, Methoden, ...

Die konkrete Zuordnung von Programmeinheiten zu Attributen ist abhängig von der jeweiligen Programmiersprache. Unter einer **Bindung** verstehen wir die Festlegung bestimmter Attribute. Hier gibt es Unterschiede bei konkreten Programmiersprachen:

- Welche Einheiten gibt es?
- Welche Attribute haben diese?
- Wann werden die Attributwerte definiert? (**Bindungszeit**)

Bei der Bindung unterscheiden wir die

Statische Bindung Attributwert liegt zur Übersetzungszeit fest (z. B. Typ bei getypter Syntax, Wert bei Konstanten)

Dynamische Bindung Attributwert ist erst zur Laufzeit festgelegt (z. B. Werte von Variablen).

Hierbei sollte beachtet werden, dass die Art der Bindung, d.h. statisch oder dynamisch, für jedes Attribut unterschiedlich sein kann.

Beispiel 2.17 (Typisierung von Programmiersprachen)**.**

Statisch getypte Programmiersprache: Typen von Bezeichnern liegen zur Übersetzungszeit fest (z. B. Pascal, Modula, Java, Go).

Dynamisch getypte Programmiersprache: Typen der Bezeichner werden erst zur Laufzeit bekannt (z. B. Lisp, Prolog, SmallTalk, Skriptsprachen wie Tcl, PHP, Ruby, Python,...)

Ungetypte Programmiersprache: Variablen haben kein Attribut „Typ“ (z. B. Assembler, z. T. auch C)

Beispiel 2.18 (Typisierung von Programmiersprachen)**.**

- in Modula-3:

  ```
  FOR i:=1 TO 10 DO ...
  ```

  Hier hat `i` den Typ `INTEGER`.

- in Go:

  ```
  x := 42
  ```

  Hier hat `x` den Typ `int`.

- in Scheme:

  ```
  (define (p x) (display x))
  ```

  Hier ist der Typ von `x` erst zur Laufzeit bekannt, z. B. hat `x` im Aufruf `(p 1)` den Typ `number`, während `x` im Aufruf `(p (list 1 2))` den Typ `list` hat.

- Eine ungetypte Sprache:

  ```
  x := 65; printchar(x)
  ```

  Hier hat `x` offensichtlich keinen Typ

Dynamische Bindungen sind eventuell **veränderbar**:

Definition 2.10 (Seiteneffekt)**.** *Ein **Seiteneffekt** ist die Veränderung einer existierenden Bindung. In einer imperativen Sprache ist dies z. B. die Veränderung von Wertbindungen von Variablen. In deklarativen Sprachen gibt es keine Wertänderung, sie sind* seiteneffektfrei.

Definition 2.11 (Gültigkeitsbereich)**.** *Der **Gültigkeitsbereich** (engl. scope) ist eine Folge von Anweisungen/Ausdrücken, in denen Bindungen einer bestimmten Einheit im Programm gültig oder sichtbar sind.*

Beispiel 2.19. In der Programmiersprache C gibt es nur zwei Gültigkeitsbereiche:

- Prozeduren: diese sind im gesamten Programm sichtbar
- Variablen: hier wird unterschieden zwischen

– globalen Variablen (diese sind im gesamten Programm sichtbar), und
– lokalen Variablen (diese sind definiert in Prozedur, z.B. auch als Parameter, und nur dort sichtbar)

Im Gegensatz dazu bieten Sprachen wie Algol, Pascal oder auch Haskell beliebig verschachtelte Gültigkeitsbereiche.

Definition 2.12 (Block)**.** *Ein **Block** ist eine Folge von Anweisungen oder Ausdrücken, in denen lokal deklarierte Einheiten sichtbar sind.*

In Programmiersprachen gibt es unterschiedliche Einheiten, die einen Block bilden, wie z.B. das Gesamtprogramm, Prozedurrümpfe, Methodendefinitionen in Klassen, `for`-Schleifen, u.ä.

Die **Sichtbarkeitsregeln** einer Programmiersprache beschreiben, welche Bindungen wo sichtbar sind. Die übliche Sichtbarkeitsregel ist: Ein Bezeichner x ist sichtbar an einer Stelle, falls x in diesem (lokaler Bezeichner) oder einem umfassenden Block (globaler Bezeichner) vereinbart wurde.

Definition 2.13 (Umgebung)**.** *Die **Umgebung** (an einer Stelle) ist die Menge aller sichtbaren Bezeichner und deren Bindungen.*

Definition 2.14. *Eine **blockstrukturierte Sprache** ist eine Programmiersprache mit einem expliziten oder impliziten Konzept von Blöcken.*

Beispiele für Blöcke in Programmiersprachen:

- Assembler: keine Blöcke
- C: nur 2 Hierarchien
- Ada, Haskell: beliebige Blockschachtelungen

Beispiel 2.20. In der Programmiersprache Ada werden Blöcke durch das Schlüsselwort `declare` eingeleitet und können auch einen Namen haben:

```
B: declare a: Real;
           b: Boolean;
   begin
     C: declare b,c: Float;  -- b verdeckt b aus B!
        begin
          c := a+b;  -- a aus B, b,c aus C
        end C;
     b := (a=3);  -- b aus B
   end B;
```

Wenn es erlaubt ist, in unterschiedlichen Blöcken Bezeichner mit gleichem Namen zu deklarieren, stellt sich die Frage, welche Deklaration bei Auftreten eines Bezeichner gemeint ist. Hier gilt üblicherweise das einfache Prinzip, dass die innerste umgebende Deklaration eines Bezeichners verwendet wird.
Dieses Prinzip erscheint klar bei statischen Blöcken, aber bei Prozeduraufrufen ist dies nicht so offensichtlich. Betrachten wir dazu folgendes Beispiel:

```
declare
  a: Integer;

  procedure p1 is
  begin
    print(a);  -- (*)
  end;

  procedure p2 is
    a: Integer;  -- lokal in p2 deklariert
  begin
    a := 0;
    p1;
  end;

begin
  a := 42;
  p2;
end;
```

Die Frage ist nun, was an der Stelle `(*)` ausgegeben wird, was gleichbedeutend ist mit der Frage, welche Deklaration von `a` an der Stelle `(*)` im Aufruf von `p1` gemeint ist. Eine eindeutige Antwort ist nicht möglich, denn dies ist abhängig von der Bindungsregel der Programmiersprache. Hier gibt es zwei Möglichkeiten:

Lexikalische (statische) Bindung von Bezeichnern zu den Deklarationen (**lexical/static scoping**): Die Bindung stammt aus der Umgebung, in der diese Prozedur (z. B. `p1`) deklariert ist, d.h. die genaue Bindung ist aus dem Programmtext, also statisch, festgelegt.

⇒ in `(*)` ist mit `a` die äußere Deklaration gemeint: Ausgabe: 42

Dynamische Bindung von Bezeichnern zu den Deklarationen (**dynamic scoping**): Die Bindung stammt aus der Umgebung, in der diese Prozedur aufgerufen wurde, d.h. die genaue Bindung wird erst dynamisch zur Laufzeit festgelegt.

⇒ in `(*)` ist mit `a` die Deklaration in `p2` gemeint: Ausgabe: 0

Die lexikalische Bindung hat die folgenden Eigenschaften:

- Sie ist einfacher zu überschauen.
- Sie ist weniger fehleranfällig.
- Mögliche Fehler können durch Prüfung eines Compilers erkannt werden.
- Sie wird in den meisten Programmiersprachen verwendet (Ausnahmen sind z.B. Sprachen wie APL, Snobol-4, Lisp, Emacs-Lisp)