2.7 Polymorphie

In der Praxis erscheint die strenge Typisierung manchmal als zu enges Konzept, um Probleme angemessen und elegant zu lösen. Zur Motivation drei Beispiele:

1) Man betrachte einen Datentyp stack mit den üblichen Operationen push, pop, top, is_empty und newstack:

```
type stack =
    uses bool with sorts BOOL
    sorts STACK, ITEM
    functions
        newstack: → STACK
        push: STACK × ITEM → STACK
        pop: STACK → STACK
        top: STACK → ITEM
        is_empty: STACK → BOOL
    laws
        is_empty(newstack)≡true
        is_empty(push(s,x))≡false
        pop(push(s,x))≡s
        top(push(s,x))≡x
end.
```

   Benötigt man innerhalb eines Programms sowohl einen Stack mit int-Elementen als auch einen Stack mit char-Elementen, so muß man den gesamten Datentyp mit allen Zugriffsprozeduren zweimal in der jeweiligen Ausprägung ITEM=int bzw. ITEM=char definieren. Dies erscheint unangemessen.

   Eine Lösung dieses Problems besteht darin, den Datentyp stack zu parametrisieren und den gewünschten Typ als aktuellen Parameter zu übergeben, z.B. in der Form

```
type stack(ITEM) = ...
```

Ein analoges Problem tritt auch im Bereich der Funktionen auf:

2) Man betrachte die Identitätsfunktion

```
function id x:D→D≡x.
```

   Sie ist eine sinnvolle Funktion auf *allen* vorstellbaren Datentypen D. Es scheint daher wenig zweckmäßig, für jeden vorkommenden Datentyp D eine eigene Definition anzugeben, wobei der Funktionsbezeichner womöglich noch jeweils unterschiedlich gewählt werden muß. Vielmehr sollte id auf allen Typen D des Universums definiert sein.

3) Zum Sortieren von Zahlen, Zeichen, Records usw. muß man in streng typisierten Programmiersprachen jeweils eine eigene Sortierfunktion schreiben. Zweckmäßiger

wäre es auch hier, der Sortierfunktion die zu sortierenden Objekte nebst ihrem Datentyp als Parameter zu übergeben.

Diese Lösungsansätze werden von dem Konzept der Polymorphie erfaßt.

Definition A:
Ein Datentyp heißt **polymorph**, wenn er einen Typparameter enthält.
Eine Funktion heißt **polymorph**, wenn der Typ eines ihrer Argumente oder des Ergebnisses polymorph ist.

Bezeichnung: Kommen in einem Datentyp Typparameter vor, so kennzeichnen wir diese durch einen großen griechischen Buchstaben (meist Δ).

Beispiele:
Folgende Datentypen sind polymorph:

$\Delta\times\Delta'$	Menge aller Paare
$\Delta\times$nat	Menge aller Paare mit zweiter Komponente nat
$\Delta\rightarrow\Delta$	Menge aller Abbildungen eines Datentyps in sich.

Folgende Funktionen sind polymorph:

```
function id x:Δ→Δ≡x.
function projx (x,y): Δ×Δ'→Δ≡x.
function projy (x,y): Δ×Δ'→Δ'≡y.
function swap (x,y): Δ×Δ'→Δ'×Δ≡(y,x).
function null x: Δ→nat≡0.
```

Anwendungen:

```
swap(-2,(true,'a'))
```

liefert

```
((true,'a'),-2).
```

Hier wurden die Typparameter von swap wie folgt konkretisiert:

Δ=int
Δ'=bool$\times$char.

Man unterscheidet verschiedenen Formen der Polymorphie (Abb. 1), wobei die ad hoc-Polymorphie nur eine als ob-Polymorphie ist.

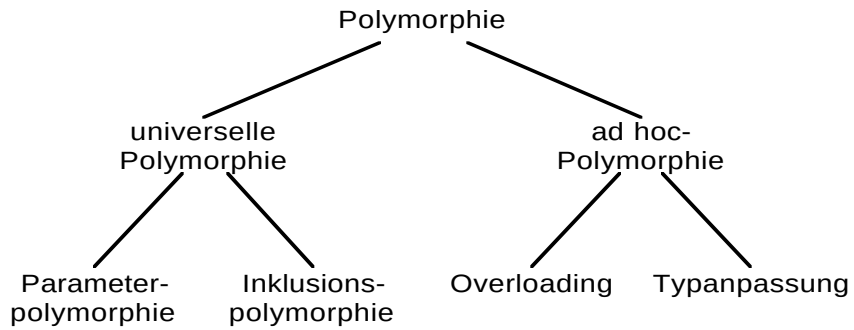

Abb. 1: Verschiedene Formen von Polymorphie

Universelle Polymorphie.

Universell polymorphe Funktionen operieren auf gleiche Weise i.a. auf einer unendlichen Zahl verschiedener Typen, deren Struktur jedoch im wesentlichen übereinstimmt. Beispiel ist die Sortierfunktion, die auf allen Typen operiert, auf denen eine lineare Ordnung gegeben ist.

Weiter unterscheidet man die

- *Parameterpolymorphie*,
  bei der eine Funktion einen impliziten oder expliziten Typparameter besitzt, der bei jeder Funktionsanwendung mit einem anderen konkreten Typ belegt wird (*Beispiel:* die Funktion id von oben oder das Objekt nil in PASCAL), und die
- *Inklusionspolymorphie*.
  Sie beschreibt die Verhältnisse, bei denen ein Objekt zu mehreren (ggf. disjunkten) Typen (Oberklassen) gehören kann, so daß die auf diesen Typen definierten Funktionen auch für das Objekt definiert sind (Prinzip der *Vererbung*).
  Beispiel: Bildung von Unterbereichstypen durch Restriktion und Vererbung der Funktionen des Obertyps auf den Untertyp.

Ad hoc-Polymorphie.

Hier liegt den Typen keine gemeinsame Struktur zugrunde. Vielmehr kann eine Funktion (zufällig) auf mehrere Typen angewendet werden. Sie verhält sich jedoch auf jedem Typ anders.

Aus Sicht der Implementierung: Bei universeller Polymorphie wird für jeden konkreten Typ der gleiche Code durchlaufen, bei ad hoc-Polymorphie ist es stets ein anderer. Ferner beschränkt sich ad hoc-Polymorphie meist nur auf eine begrenzte mehr oder weniger bekannte Anzahl von verschiedenen Typen. Beispiel ist hier die Funktion + in PASCAL: Auf dem Typ integer realisiert sie die Addition, auf Mengen die Vereinigung.

Man unterscheidet zwei ad hoc-Polymorphien:

- Polymorphie durch *Overloading* (*Überladen*): Hier verwendet man denselben Namen (z.B. das Operationszeichen +) für völlig verschiedene Objekte, und nur aus dem

Zusammenhang kann der Compiler entscheiden, um welches Objekt es sich handelt und welcher Code zu erzeugen ist.

- Polymorphie durch *Typanpassung:* Wieder verwendet man (scheinbar) denselben Namen für unterschiedliche Objekte (z.B. 2 für das real- und für das int-Objekt). Faktisch wird jedoch das Objekt durch automatische Typanpassungen an den erforderlichen Argumenttyp einer Funktion angeglichen.

Aufgabe:
Welchen Typ besitzt der Ausdruck swap(sin,(3,true))?

2.7.1 Typinferenz

In gängigen Programmiersprachen muß man für alle Objekte bei ihrer Definition den Typ festlegen. Bei der Übersetzung eines Programms kann der Compiler dann aus den Typen der Variablen und Konstanten den Typ der Ausdrücke und Anweisungen ableiten und Typprüfungen durchführen. Häufig erfolgt solch eine Typinferenz bottom-up anhand der Baumdarstellung des Ausdrucks.

Beispiel:
Man betrachte den Baum aus Abb. 1.

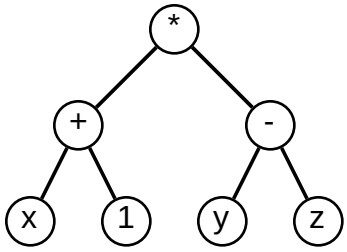

Abb. 1: Arithmetischer Ausdruck in Baumdarstellung

Durch Bottom-up Analyse gewinnt man hier folgendermaßen den Typ des Ergebnisses: Man ergänzt den Baum an den Blättern um konkrete Typen, wenn sie für die jeweiligen Objekte bekannt sind, oder um Typvariablen, wenn sie nicht bekannt sind. Sukzessive schiebt man dann die Typen nach oben und eliminiert die Typvariablen. Ggf. sind hierzu mehrere Baumdurchläufe erforderlich (Abb. 2).

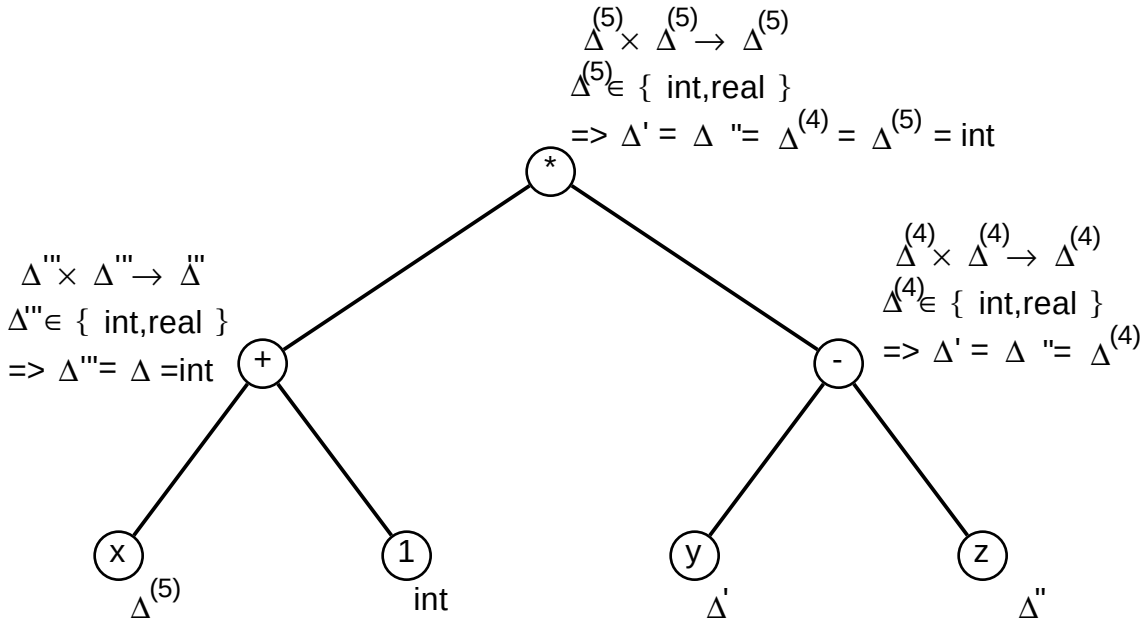

Abb. 2: Inferenz des Typs eines Ausdrucks

Ähnlich geht man auch vor, wenn polymorphe Funktionen beteiligt sind. Wir wollen hierfür zunächst einige einfache Typinferenzregeln definieren und das Verfahren sodann an einigen Beispielen erläutern.

Typinferenzregeln:

1) Regel für *Funktionsanwendung:* Gilt (f x):Δ, dann setze x:Δ' und f: Δ'→Δ für einen neuen Typ Δ' (E:Δ bedeutet hier und im folgenden also: E besitzt den Typ Δ).
2) Regel für *Gleichsetzung:* Hat man x:Δ und x:Δ' abgeleitet, so setze Δ=Δ'. (Hier kommt die strenge Typisierung zum Zuge: Verschiedene Datentypen Δ und Δ' sind disjunkt: Gehört ein Objekt x zu Δ und Δ', so muß zwangsläufig Δ=Δ' gelten!)
3) Regel für *Funktionalität:* Falls Δ→Δ'=Δ"→Δ"' abgeleitet wurde, so setze Δ=Δ" und Δ'=Δ"'.

Beispiele:

1) Gegeben sei die polymorphe Funktion

   function comp f:Δ g:Δ' x:Δ"→Δ"'≡f(g(x)).

   Wie lautet der Typ von comp? Aus Regel 1 folgt:

   g(x): $\Delta^{(4)}$

   f: $\Delta^{(4)}\rightarrow\Delta'''$ für einen neuen Typ $\Delta^{(4)}$.

   Analog folgt:

   x: $\Delta^{(5)}$

   g: $\Delta^{(5)}\rightarrow\Delta^{(4)}$ für einen neuen Typ $\Delta^{(5)}$.

   Mit Regel 2 folgt dann aus dem Funktionskopf von comp:

f: $\Delta=\Delta^{(4)}\to\Delta'''$

g: $\Delta'=\Delta^{(5)}\to\Delta^{(4)}$

x: $\Delta''=\Delta^{(5)}$.

Folglich besitzt comp den Typ:

$\Delta\to\Delta'\to\Delta''\to\Delta'''=(\Delta^{(4)}\to\Delta''')\to(\Delta^{(5)}\to\Delta^{(4)})\to\Delta^{(5)}\to\Delta'''$.

2) Man betrachte die Funktion:

function g f:$\Delta\to\Delta'\equiv$f(g(f)).

Wie lautet der Typ von g? Aus Regel 1 folgt:

g(f): Δ''

f: $\Delta''\to\Delta'$ für einen neuen Typ Δ''.

Analog:

f: Δ'''

g: $\Delta'''\to\Delta^{(4)}$ für einen neuen Typ $\Delta^{(4)}$.

Mit Regel 2 folgt dann aus dem Funktionskopf von g:

f:$\Delta=\Delta'''=\Delta''\to\Delta'$ und

g: $\Delta\to\Delta'=\Delta\to\Delta''=\Delta'''\to\Delta^{(4)}$,

also nach Regel 3:

$\Delta=\Delta'''$, $\Delta'=\Delta''=\Delta^{(4)}$.

Dies liefert als Typ von g:

$(\Delta'\to\Delta')\to\Delta'$.

Aufgaben:

Man bestimme die Typen von

function c x:Δ y:$\Delta'\to\Delta''\equiv$x.

function app f:$\Delta\to\Delta'\equiv$f(f).

function s f:Δ g:Δ' x:$\Delta''\to\Delta'''\equiv$fx(gx).

function q f:Δ x:Δ' g:$\Delta''\to\Delta'''\equiv$gf(fxg).

2.7.2 Gleichheit auf Datentypen

Unter den polymorphen Funktionen, die in einer Programmiersprache zur Verfügung stehen, wirft die Überprüfung auf Gleichheit und Ungleichheit zweier Objekte

=: $\Delta\times\Delta\to$bool

$\neq$: $\Delta\times\Delta\to$bool

besondere Probleme auf. Während Gleichheit z.B. auf den elementaren Datentypen oder kartesischen Produkten dieser Typen wohldefiniert und effizient implementierbar ist, bekommt man Schwierigkeiten, wenn Δ ein Funktionstyp $\Delta'\to\Delta''$ ist. Hier ist zunächst nicht klar, wann für zwei Funktionen f,g: Δ gilt:

f=g.

Definition A:
Zwei Funktionen f,g: $\Delta \to \Delta'$ heißen

- **intensional gleich**, wenn ihre Beschreibungen identisch sind;
- **extensional gleich**, wenn für alle x:Δ gilt:

  f(x)=g(x).

Bemerkung: Offenbar ist die intensionale Gleichheit effizient zu implementieren, da dies praktisch auf einen Vergleich von Programmtexten hinausläuft. Andererseits gibt es keinen Algorithmus, der für zwei beliebige Funktionen entscheidet, ob sie extensional gleich sind.

Beispiel:
Wir definieren drei Funktionen

function double1 n:int$\to$int$\equiv$2n.
function double2 n:int$\to$int$\equiv$n+n.
function double3 n:int$\to$int$\equiv$3n-n.

Alle drei Funktionen sind extensional gleich und intensional paarweise verschieden.

Entlang der Philosophie der referenziellen Transparenz (Extensionalitätsprinzip) entscheidet man sich in funktionalen Sprachen in der Regel für die Interpretation der Gleichheit im extensionalen Sinne, da es nur auf die Werte von Ausdrücken ankommen soll, nicht jedoch darauf, wie diese berechnet werden. Folglich ist man gezwungen, das Problem der Unentscheidbarkeit in den Griff zu bekommen. Zwei Lösungen bieten sich hierfür an:

1. Lösung: Man erlaubt den Gleichheitstest in voller Polymorphie

=: $\Delta \times \Delta \to$bool

und nimmt dann in Kauf, daß = auf Typen Δ undefiniert ist, in denen Funktionen vorkommen.

2. Lösung: (Diesen Ansatz verfolgt auch die später zu behandelnde Programmiersprache ML) Man schränkt die Polymorphie des Gleichheitstests ein und erlaubt den Test nur auf Typen, auf denen keine Unentscheidbarkeitsprobleme auftreten. Hierzu isoliert man aus dem Universum aller Typen die sog. *Gleichheitstypen* (*equality types*) und führt hierfür spezielle Typvariablen ein, die man durch ein hochgestelltes Gleichheitszeichen markiert (z.B. $\Delta^=$). Damit ist = dann eine polymorphe Funktion auf Gleichheitstypen:

=: $\Delta^= \times \Delta^= \to$bool.

Gleichheitstypen kann man induktiv definieren durch:

(1) Die elementaren Datentypen int, bool, nat, real, char sind Gleichheitstypen.

(2) Jede Gleichheitstypvariable $\Delta^{=}$ ist ein Gleichheitstyp.

(3) Sind $\Delta_1,...,\Delta_n$ Gleichheitstypen, so auch

$$\Delta_1 \times ... \times \Delta_n.$$

In dieser Definition kommt als einziger Typkonstruktor die Aggregation vor. Verfügt die zugrundeliegende Programmiersprache über weitere Typkonstruktoren, muß man diese ggf. mit in die Definition aufnehmen.

Beispiele:

Gleichheitstypen sind z.B.

$\text{bool} \times \text{char}$,

$\text{int} \times \text{char}$,

$\text{int} \times (\Delta^{=} \times \text{bool})$,

aber nicht

$\text{bool} \rightarrow \text{bool}$,

$\Delta \times \text{int}$.