TYPSYSTEME

Inhalt

Typsysteme

Regeln für die Typisierung von Größen

auch Typdisziplin genannt

Bisherige Annahme: Monomorphie

jede Größe (Konstante, Variable, Parameter, Prozedur,...)
hat höchstens einen Typ

Dieser Typ wird bei der Vereinbarung explizit angegeben

Das schränkt zu sehr ein!

für wiederverwendbare Software braucht man eine losere Typdisziplin und ein mächtigeres Typsystem

- Überladen (*ad-hoc*-Polymorphie

$$+ \mid\!\rightarrow \{\tau \times \tau \rightarrow \tau \mid \tau \in \{\text{Integer}, \text{Real}, 2^S\}\}$$

- Polymorphie (parametrische Polymorphie)

$$= \mid\!\rightarrow \{\tau \times \tau \rightarrow \text{Truth-Value} \mid \text{für alle Typen } \tau\}$$

- Untertypen und Vererbung (Einschluß-Polymorphie)

$$= \mid\!\rightarrow \{\tau \times \tau \rightarrow \text{Truth-Value} \mid \text{für alle Untertypen } \tau \text{ von } \sigma\}$$

monomorphe Typsysteme

Imperative Sprachen sind (im wesentlichen) monomorph

Pascal: Alle Benutzer-definierten Typen sind monomorph

```
type Charset = set of Char;
function disjoint (s1, s2 : Charset) : Boolean
begin
   disjoint := s1 *s2 = []
end;
```

Beachte

`disjoint` ist monomorph

$$\texttt{disjoint} \mid\!\rightarrow 2^{\text{Char}} \times 2^{\text{Char}} \rightarrow \text{Truth-Value}$$

`disjoint` kann deshalb nur auf Zeichenmengen angewendet werden

```
var chars : Charset;
...
if disjoint(chars, ['a', 'e', 'i', 'o', 'u'])
then ...
```

Im Rumpf von `disjoint` wird benutzt, daß `s1` und `s2` Mengen sind
Der Typ der Elemente (Char) spielt aber keine Rolle

Trotzdem ist Pascal nicht vollständig monomorph

einige eingebaute Prozeduren und Funktionen sind

- überladen: * $\mid\!\rightarrow \{\tau \times \tau \rightarrow \tau \mid \tau \in \{\text{Integer, Real, } 2^S\}\}$
- polymorph: `eof` $\mid\!\rightarrow$ **file** $(\tau) \rightarrow$ Truth-Value

Überladen

Prinzip (ad-hoc-Polymorphie)

Ein Bezeichner/Operator kann mehrere Vereinbarungen haben, die

- semantisch verschieden sind
- in den Anwendungen eindeutig identifiziert werden können

Kontext-abhängiges Überladen (Ada)

überladene Vereinbarungen müssen
verschiedene Parameter oder verschiedene Resultate haben.

Der Operator "/" steht für die vordefinierten Funktionen

Integer × Integer → Integer
Float × Float → Float

Der Benutzer kann den Operator mit weiteren Funktionen überladen

```
function "/" (m,n : Integer) return Float is
begin
   return Float>(m) / Float(n);
end;
```

Der Operator "/" steht dann gleichzeitig für:

Integer × Integer → Integer
Float × Float → Float
Integer × Integer → Float

Anwendungen

```
n: Integer; x : Float; ...
x:= 7.0/2.0;        ergibt 3.5
x:= 7/2;            ergibt 3.5
n:= 7/2;            ergibt 3
n:= (7/2)/(5/2);    ergibt 1
x:= (7/2)/(5/2);    ergibt 1.5 oder 1.4 (mehrdeutig)
```

Varianten des Überladens

Eingebautes Überladen

(Algol-60, Pascal, Modula, ML)

- Überladen eingebauter Operatoren (und Funktionsnamen) möglich
- meist Kontext-unabhängig (siehe unten)

Kontext-unabhängiges Überladen

(Algol-68, C++, Hope, Haskell)

- Überladen aller Operatoren und Funktionen möglich (auch von Benutzer-definierten)
- die Parametertypen müssen verschieden sein
- Identifizierung der gemeinten Operation einfach und immer möglich

Kontext-abhängiges Überladen

(Ada)

- Überladen von Operatoren, Funktionen, Konstanten (auch Benutzer-definierten)
- die Parametertypen oder die Resultattypen müssen verschieden sein (Konstanten können nur Kontext-abhängig überladen werden)
- Identifizierung der gemeinten Operation schwierig, es gibt mehrdeutige Ausdrücke (siehe oben)

polymorphe Funktionen

der Typ einer Funktion kann Typvariable (Parameter) enthalten

(daher parametrische Polymorphie)

DieTypvariable werden mit griechischen Buchstaben bezeichnet

(in Miranda: *, **, ***, ..., in ML ‘a, ‘b ...)

hier meist mit σ, τ, α, ...

Beispiel: polymorphe Funktionen (Haskell)

Eine monomorphe Funktion,

die das zweite Element eines Zahlenpaares liefert:

```
second::(num,num)->num
second (x,y) = y
```

Der Rumpf der Funktion macht keinerlei Annahmen

über den Typ von `x` und `y`

Er “funktioniert” für beliebige Paare.

Daher hat `second` den folgenden allgemeinsten Typ

`second::(σ,τ)->τ` `=(a,b)->b` in “echtem” Haskell

Explizit aufgeschrieben hat `second` die Menge von Typen

$\{\sigma \times \tau\text{->}\tau \mid$ für beliebige Typen $\sigma, \tau\}$

Beispiele für Typen von `second`

Num × Num->Num, Num × Char->Char

Gegenbeispiele für Typen von `second`

Num->Num Num × Char × Num->Num kein Paar

Num × Char->Num Resultat nicht vom Typ des zweiten Elements

weitere Beispiele für polymorphe Funktionen

“polymorphes” Pascal

```
function disjoint (s1, s2 : set of τ) : Boolean;
begin
   disjoint := s1 *s2 = []
end;
```


parameterisierte Typen

Definition

Ein parametrisierter Typ hat (einen oder mehr) Parameter,

für die beliebige Typen eingesetzt werden können.

Die Parameter heißen auch Typvariablen

viele zusammengesetzte Typen sind parametrisiert

zum Beispiel, in Pascal

set of τ	(τ muß aber diskret sein)
array σ **of** τ	(σ muß aber diskret sein)
file of τ	(τ darf aber keine Zeiger enthalten)

Das heißt

für alle (viele bzw. manche) Typen τ ist **set of** τ, **array** σ **of** τ usw. ein zusammengesetzter Typ.

In vielen Sprachen sind nur vordefinierte Typen parametrisiert

Man kann z. B.in Pascal nicht definieren

```
type Pair(τ)  = record fst, snd: τ end;
     RealPair = Pair(Real);
     IntPair  = Pair(Integer);
```

oder genauso wenig

```
type List(τ)  = …;
     RealList = List(Real);
     IntList  = List(Integer);
```

In funktionalen Sprachen ist das allgemein möglich (ML)

```
type     τ pair   = τ * τ
datatype τ list   = nil | cons of (τ * (τ list))
type     intpair  = int pair
type     intlist  = int list
```

poly-Typen

Definition

poly-Typ: Typausdruck mit einer oder mehreren Typvariablen

von einem *poly*-Typen
können *mono*-Typen (ohne Variablen) abgeleitet werden

Ein *poly*-Typ ist aber auch selber ein Typ (eine Wertemenge)

Frage

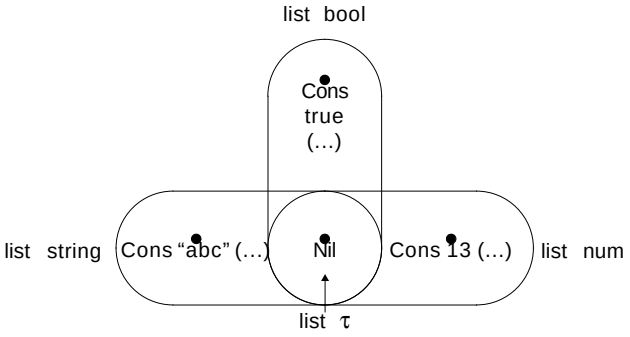
Was ist die Wertemenge eines *poly*-Typen?

Antwort

Der Durchschnitt der Werte
in den aus dem *poly*-Typen ableitbaren *mono*-Typen

Beispiel (`list τ`)

nicht-leere Listen haben einen *mono*-Typ
Nil hat den *poly*-Typ `list τ`



Typüberprüfung und Typinferenz

Typüberprüfung für imperative Sprachen

Der Typ aller vereinbarten Bezeichner muß explizit spezifiziert werden

dann kann für jeden Ausdruck festgestellt werden, ob er wohlgetypt ist

kleine Ausnahme

Konstantenvereinbarung in Pascal
`const` *v* `=` *E*
Der Typ von *v* wird implizit durch den Typen von *E* bestimmt

Typinferenz (noch monomorph)

der Typ von vereinbarten Bezeichnern wird hergeleitet (inferiert)
aus den Regeln für ihre Benutzung in Ausdrücken

Typinferenz für eine Miranda-Funktion

`plural x = x ++ "s"` *Begründung*

`"s"`: [char]	(vordefinierter Typ)
⇓	
++ : [char] → [char] → [char]	(vordefinierter Bezeichner*)
⇓	
x : [char]	(x ist ein Argument von ++)
⇓	
x ++ "s" : [char]	(Resultat von ++)
⇓	
plural x: [char]	(Seiten von = gleich getypt)
⇓	
plural : [char] → [char]	(plural wird auf x angewendet)

polymorphe Typinferenz (1)

Beispiele aus Miranda

Identität

`id x = x` *Begründung*

x : α	(Annahme)
⇓	
id x : α	(Seiten von = gleich getypt)
⇓	
id : α → α	(id wird auf x angewendet)

Komposition

`compose f g x = f (g x)` *Begründung*

x : α	(Annahme)
⇓	
g : α → β	(g wird auf x angewendet)
⇓	
g (x) : α → β	(Resultat von g)
⇓	
f : β → γ	(f wird auf (g x) angewendet)
⇓	
f (g x) : γ	(Resultat von f)
⇓	
compose f g x : γ	(Seiten von = gleich getypt)
⇓	
compose f g : α → γ	(compose f g wird auf x angewendet)
⇓	
compose f : (α → β) →(α → γ)	(compose f wird auf g und x angewendet)
⇓	
compose : (β → γ) →(α → β) →(α → γ)	(compose wird auf f und g und x angewendet)

polymorphe Typinferenz der Längenfunktion

Länge von Listen

`length [] = 0` *Begründung*

0 : num	(vordefinierte Konstante)
⇓	
length [] : num	(Seiten von = gleich getypt)
⇓	
[] : [α]	(vordefinierte Konstante)
⇓	
length : [α] → num	(length wird auf [] angewendet)

`length (h:t) = 1+(length t)` *Begründung*

: : α → [α] → [α]	(vordefinierte Funktion)
⇓	
h: α	(h ist erstes Argument von :)
⇓	
t : [α]	(t ist zweites Argument von :)
⇓	
h:t : [α]	(Resultat von :)
⇓	
1 : num	(vordefiniertes Literal)
⇓	
+ : num → num → num	(vordefinierte Funktion)
⇓	
length t : num	(zweites Argument von +)
⇓	
1+(length t)	(Resultat von +)
⇓	
length (h:t): num	(Seiten von = gleich getypt)
⇓	
length : [α] → num	(length wird auf (h:t) angewendet)

Regeln für Typinferenz

Typstruktur

τ ::= bool | num | char	vordefinierte Typen
| $\alpha \mid \beta \ldots$	Typvariablen
| $[\sigma]$	Listen
| $(\sigma_1, \ldots, \sigma_n)$	Tupel ($n > 1$)
| $\sigma \to \tau$	Funktionen

Ausdrücke

E ::= C	vordefinierte Namen (Konstanten)
| X	Typvariablen
| E E'	Funktionsanwendung
| E = E'	Gleichungen

Inferenzregeln

vordefiniert $\dfrac{\varnothing}{c \Rightarrow \alpha}$ (für bekannte Typen α)

Variable $\dfrac{\varnothing}{x \Rightarrow \alpha}$ (für neue Typvariablen α)

Funktionsanwendung $\dfrac{F \Rightarrow (\alpha \to \beta) \wedge A \Rightarrow \beta}{FA \Rightarrow \beta}$ $\dfrac{FA \Rightarrow \beta \wedge A \Rightarrow \alpha}{F \Rightarrow (\alpha \to \beta)}$

Gleichung $\dfrac{(L \equiv R) \wedge L \Rightarrow \alpha}{R \Rightarrow \alpha}$ $\dfrac{(L \equiv R) \wedge R \Rightarrow \alpha}{L \Rightarrow \alpha}$

Typanpassungen (*coercions*)

Definition

Eine Typanpassung ist eine implizite Abbildung
von einem Typ in einen anderen

Zweck

In einem Kontext, wo ein Typ *T* erwartet wird,
kann auch ein Typ *S* akzeptiert werden,
wenn es eine Typanpassung von *S* nach *T* gibt.

Beispiel Pascal

wo kann implizit angepasst werden?

```
var x : Real;
    i : Integer
procedure P (n: Integer);
    ... ;

x := i;
i := x;
P(x);
```

Welche Anpassungen gibt es in Pascal?

Ausweitung (*widening*): Integer → Real
bei Zuweisungen

- Weshalb gibt es keine Einschränkung Real → Integer ?
- Wie ist es bei Parametern?

Typanpassungen in Algol-68

Es gibt extrem viele Anpassungen

Ausweitung (*widening*): Integer → Real → Complex
und Integer → **long** Integer→ **long long** Integer
Dereferenzierung (*dereferencing*): **ref** $\tau \to \tau$
Deprozedurierung (*deproceduring*): **proc**() $\tau \to \tau$
Reihung (*rowing*): $\tau \to$ [1:1] **of** τ ([1:1] **of** $\tau \to [n{:}n]$ **of** τ)
Vereinigung (*uniting*): $\tau \to \tau \cup \sigma$
Wegwerfen (*voiding*): $\tau \to ()$

Es gibt etliche Anpassungskontexte

linke und rechte Seite von Zuweisungen

Parameterübergabe

bedingte Anweisung (Balancieren)

Typanpassungen vertragen sich nicht mit Überladen und Polymorphie
(Weshalb?)

Da Überladen und Polymorphie wichtiger sind,
sind Typanpassungen auf dem Rückmarsch

Die Alternative sind explizite Typanpassungen

```
x := Real(i);
i := Truncate(x);
P(Real(x));
```


Untertypen (*subtypes*) allgemein

Definition

zwischen den Typen $\mathcal{T}$ einer Sprache besteht eine *Untertyprelation* $\leq$

Gilt $u \leq t$, dann ist u ein Untertyp von t, und t der *Basistyp* von u.

Dann erbt u Eigenschaften von t, z .B. Operationen

Diese Operationen sind *polymorph*.

Weil die Polymorphie auf der Untertyprelation basiert,
heißt sie Einschluß (*inclusion*)-Polymorphie

$\leq$ ist eine *partielle Ordnung*:
- $t \leq t$ (Reflexivität)
- Wenn $u \leq t$ und $t \leq u$, dann $u = t$ (Antisymmetrie)
- Wenn $u \leq t$ und $t \leq s$, dann $u \leq s$ (Transitivität)

Arten der Untertyprelation

konkrete Untertypen
- $u \leq t$, weil alle Werte von u Werte von t sind

abstrakte Untertypen
- $u \leq t$, weil die Operationen von t auch Operationen für u sind

Arten der Vererbung
- Wenn $u \leq t$ und $u \leq s$, dann $s = t$ (einfache Vererbung)

Untertypen (*subtypes*) in Pascal

Definition

Ein Untertyp ist eine Untermenge eines Typen
der Eigenschaften seines Basistypen (Obertypen) erbt
(z. B. die Operationen)

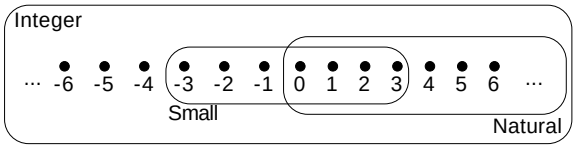
Typen enthalten ihre Untertypen
— daher der Name Einschluß (*inclusion*)-Polymorphie

Untertypen können Programme lesbarer machen... und mehr

Beispiel: Abschnittstypen in Pascal

Abschnittstypen (*subranges*) sind Untertypen diskreter Typen

```
type Natural = 0..maxint;
     Small   = -3..+3;
var  i : Integer;
     n : Natural;
     s : Small;
```




```
i := n; i := s ;     sichere Zuweisung
n := s; n := i ;     unsichere Zuweisung (Laufzeitüberprüfung)
```

analoges gilt für andere Funktionen des Basistypen,
wenn sie auf Untertypen angewandt werden (und umgekehrt)

Untertypen (*subtypes*) in Ada

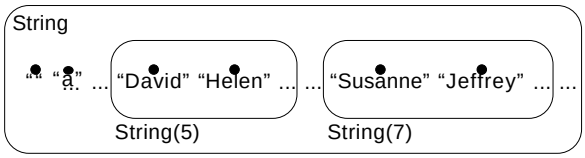
Beispiel: Untertypen in Ada

Abschnittstypen (*subranges*) sind Untertypen diskreter Typen

```
subtype Natural is Integer range 0..Integer'last;
        Small   is Integer range -3..+3;
```

Feldtypen sind Untertypen offener Feldtypen

```
type String is array (Integer range <>)of Character;
subtype String5 is String(1..5);
subtype String7 is String(1..7);
```



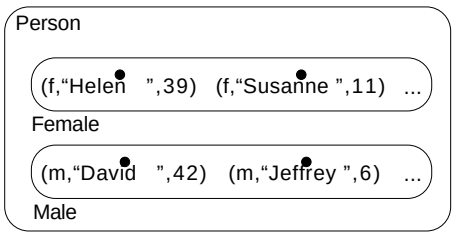
Untertypen von Verbunden (Ada)

Beispiel: Untertypen in Ada (Fortsetzung)

Verbunde mit Diskriminanten sind Untertypen von Verbunden

```
type    Sex     is (f, m);
type    Person (gender : Sex) is
           record
              name: String(8);
              age : 0..120;
           end record;

subtype Female is Person(gender => f);
subtype Male   is Person(gender => m);
```



Bemerkungen

In Ada sind Diskriminanten immer diskrete Typen

Deshalb sind verschiedene Untertypen immer disjunkt

(Weshalb?)

Vererbung (*inheritance*)

Beispiel: Vererbung in (hypothetischem) Ada

präfixierte Verbunde sind Untertypen ihrer Präfixe

```
type    Point  is record x, y: Real end record;
subtype Circle extending Point is
               record r: Real; end record;
subtype Box    extending Point is
               record w, d : Real; end record;
```

präfixierte Verbunde erben die Operationen auf den Präfixen

```
function distance (p,q : Point) return Real is
   begin return sqrt(sqr(p.x-q.x) + sqr(p.y-q.y)) end;
```

die Funktion arbeitet auf Punkten, Kreisen und Rechtecken

Verschieben von Objekten geht nicht so einfach:

```
function move (in p: Point, dx, dy: Real)
   return Point is
   begin return (x=>p.x+dx, y=>p.y+dy) end;
```

Die Funktion `move` liefert einen verschobenen Punkt,
auch dann wenn sie auf einen Kreis / Rechteck angewendet wird
(und nicht das verschobene Objekt!)

hypothetische Erweiterung von Ada

```
function move (in p: Point, in dx, dy: Real)
   return like p is
   begin return p with (x => x+dx, y => y+dy) end;
```

Diese Version von `move` liefert ein verschobenes Objekt,
egal von welchem Untertypen
(***with*** übernimmt alle nicht erwähnten Komponenten von `p`)

Vererbung (*inheritance*)

Beispiel: Überladen von Funktionen für Untertypen
in (hypothetischem) Ada

```
function area (p: Point) return Real is
   begin return 0.0 end;
```

Die Funktion `area` soll nicht vererbt werden!
(sie macht so keinen Sinn auf Kreisen und Rechtecken)

Die Funktion wird verdeckt durch
Neu-Definition anderer Versionen für die Untertypen

```
function area (p: Circle) return Real is
   begin return pi*sqr(p.r) end;

function area (p: Box) return Real is
   begin return p.w * p.d end;
```


Mehrfachvererbung

Ein Typ erbt von mehr als einem Typ

```
subtype U      extending S, T is
               record ... end record;
```

das ist kein Problem, aber ...

wiederholte Vererbung

```
subtype S      extending B is
               record ... end record;
subtype T      extending B is
               record ... end record;
```

Fragen

- erbt *U* den Typ *B* einmal oder zweimal?
- wenn *einmal*: via *S* oder via *T?*
  (diese Typen könnten Eigenschaften von *B* (verschieden) überladen!)

abstrakte Basistypen

ein abstrakter Basistyp definiert nur Signaturen von Operationen
und hat selbst keine Implementierung

alle Untertypen müssen diese Operationen implementieren

Beispiel

```
type    Ord    is record end record;
function "<=" (in a, b: Ord) return Boolean;

subtype Point  extending Ord is
               record x, y : Real; end record;
function "<" (in a, b: Point) return Boolean;
begin ... end
```

Übungen 7: Typsysteme

Grundsätzliches

Was unterscheidet die Typ-Disziplinen?

- monomorph
- *ad-hoc*-polymorph (Überladen)
- parametrisch polymorph (polymorph)
- Teilmengen-polymorph (Untertypen)

Überladen

1. Nimm an, die Operation "&" sei definiert für
   $C \times C \to S, C \times S \to S, S \times C \to S, S \times S \to S$

   Identifiziere die Operationen in
   `c & s, (s&c)&c, s&(c&s)`