TYPSYSTEME

Inhalt

Typäquivalenz (Wiederholung)

dynamische / statische Typisierung

Polymorphie

- ad-hoc: Überladen
- parametrisch: "echt" polymorphe Funktionen
- Inklusion
  - imperativ: Teilmenge
  - objektorientiert: Typerweiterung

Typumwandlungen

Typanpassungen

Was ist ein Typsystem?

eine Menge von Typausdrücken $T = \{t_1, ..., t_n\}$

eine Aufteilungung der Wertemenge W in Typen $(W_1, O_1), ..., (W_n, O_n)$

Regeln zum Umgang von "Operationen" mit Werten

"Operationen" sind dabei:

- Funktionen
- Operatoren
- Befehle
- Prozeduren

Typäquivalenz (Wiederholung)

Wann gilt S ? T?

strukturell

S und T haben die gleiche Wertemenge

namentlich

S und T haben den gleichen Typnamen

Beispiel

type Name = Cardinal ? Char
type String = Cardinal ? Char

String ? Name ?

dynamische und statische Typisierung

dynamisch

jeder Wert hat einen Typ

statisch

Programmstücke haben Typen ? ?Werte haben *vorhersehbare* Typen

Beispiel für dynamische Typisierung

```
var Monat;
read(Monat);
if 1 <= Monat and Monat <= 12 then ...
else if Monat = "jan" then ...
else if Monat = "feb" then ...
...
```


Monomorphie

Monotypen

jeder Wert hat einen konkreten Typ

jedes Programmstück hat einen konkreten Typ

Typen haben wechselseitig disjunkte Wertemengen

"Mono-Bindungen"

Jeder Name ist an höchstens eine Größe gebunden

- jeder Funktionsname bezeichnet genau eine Funktion

rein monomorphe Programmiersprachen gibt es praktisch nicht!

Überladen

Prinzip (ad-hoc-Polymorphie)

Ein Name kann gleichzeitig an mehrere verschiedene Größen gebunden werden

aus jeder Benutzung des Namens muss hervorgehen, welche Größe gemeint ist

Beispiele

Bezeichner-"Töpfe" in C und C++

Überladen von Funktionen und Prozeduren in Ada, ML und Java

Überladen von Funktionen und Prozeduren

kontextfrei

Anzahl und/oder Typ der Parameter müssen verschieden sein

"/": Integer ? Integer ? Integer
"/": Float ? ? Float? ? Float

kontext-sensitiv (Ada)

Parameter und/oder Ergebnistyp müssen verschieden sein

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

Integer? Integer?? Float

- dann sind nicht alle Ausdrücke sind eindeutig getypt

parametrische Polymorphie (die *echte*)

Polytypen

Polymorphe Funktionen

Typinferenz

Polytypen

Monotypen

$t = T$ T ist ein Typausdruck

Polytypen

$t(?_1, ..., ?_n) = T[?_1, ..., ?_n]$ T ist ein Typausdruck mit *Typvariablen*

Typinstanzen

$t(T_1, ..., T_n) = T[?_1/T_1, ..., ?_n/T_n]$ $T_1, ..., T_n$ sind Typausdrücke

Beispiel (ML)

```
datatype 'a list = Nil | Cons 'a * ('a list)
```

Polytypen haben unendlich viele Typinstanzen, z.B. die Monotypen

`int list` … `('a list) list` … `(int list * real) list`

polymorphe Funktionen

eine Funktion, deren Typ ein Polytyp ist

Beispiel: polymorphe Funktionen (ML)

```
fun second (x,y) = y
> val second : fn ('a * 'b) -> 'b

second(42,[1,2,3,4]);
> [1,2,3,4]: int list
second(42,[1,2,3,4],0);
> error ...
second(42,[1,2,3,4])+13;
> error ...
```


Typüberprüfung und Typinferenz

Typüberprüfung

Der Typ aller vereinbarten Bezeichner wird explizit spezifiziert

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

Typinferenz

der Typ von Bezeichnern wird aus ihrer Vereinbarung hergeleitet (*inferiert*)

Typinferenz

$T ::= \text{bool} \mid \text{int} \mid \text{string} \mid ? \mid ? ... \mid [T] \mid (T, ..., T) \mid T??$ T Typen

$D ::= E = E'$ Gleichungen

$E ::= c_1 \mid c_2 \mid ... \mid x_1 \mid x_2 \mid ... \mid E\ E' \mid$ Ausdrücke

Inferenzregeln

vordefinierte Konstanten	$c_i: T_i$	
Variablen	$x: ?$	für *frische* Typvariablen ?
Definition	$L = R \Rightarrow L: ?, R: ?$	
Funktionsanwendung	$F\ A \Rightarrow F: ? \to ??, A: ?, F\ A: ?$	

Beispiel

```
fun length [] = 0
   | length (h:t) = 1+(length t)
```


Typumwandlung

explizite Umwandlung (*casting*)

eine Abbildung von einem Typ auf einen anderen

Beispiele (Ada)

```
x : Float; i : Integer;
x := Float(i;
i := Integer(x);
```

Typanpassung (*coercion*)

eine automatische, implizite Typumwandlung

Beispiele (Pascal)

```
var x : real; i : integer;
...
x := i;
```


Untertypen (*subtypes*)

aus einem Typen*t* kann ein *Untertyp* *u* abgeleitet werden

$u \leq t$ t *vererbt* Eigenschaften (Operationen) an u

konkret, imperativ (basierend auf Werteteilmengen)

$u \leq t$ wenn $W_u \subseteq W_t$. Es gilt $O_u = O_t|_{W_u}$

abstrakt, objektorientiert (basierend auf Eigenschaftserweiterungen)

$u \leq t$, wenn $O_u \supseteq O_t$

Einschluß-Polymorphie (*inclusion polymorphism*)

Die Operationen in O_u und O_t sind *polymorph*

Werteteilmengen in Ada

Abschnittstypen (*subranges*)

```
subtype Natural is Integer range 0..Integer'last;
        Letter  is Character range 'A'..'Z';
```

Teilfelder

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

Teilverbunde

```
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);
```


dynamische versus statische Typisierung

typsichere und -unsichere Operationen

```
i : Integer := ... ; n: Natural := ...;
s: String := ... ; s5: String5 := ... ;
p: Person := ... ; w : Female := ... ;
n := i;
i := n;

s5 := s;
s := s5;

w := p;
p := w;
```

jeder Wert hat einen *statischen* Basistyp und einen *dynamischen* aktuellen Typ

Eigenschaftserweiterung

Erweiterung von Verbunden in *hypothetischem* Ada

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

Operationen werden an erweiterte Typen vererbt

```
function distance (p,q : ? Point) return Real is
  begin return sqrt(sqr(p.x-q.x) + sqr(p.y-q.y)) end;
function move (p: ? Point, dx, dy: Real)
                                  return like p is
  begin return p with (x=>p.x+dx, y=>p.y+dy) end;
```


Überschreiben (*overriding*)

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

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

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

subtype Cpoint ? Point is
  record c: Color; end record;
```


Mehrfachvererbung

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

wiederholte Vererbung

```
subtype S ? B is record ... end record;
subtype T ? B is record ... end record;
subtype U ? S, T 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) überschreiben!)

abstrakte Obertypen (*interfaces*)

ein abstrakter Basistyp definiert nur Signaturen von Operationen

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

alle Untertypen müssen diese Operationen implementieren

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

Erinnert Euch das an eine andere Art von Polymorphie?

Binden

```
p: Point := (x => 0.0,y => 0.0);
c: Circle := (x => -1.0,y => -1.0, r => 3.0);
b: Box := (x => 1.0,y => 1.0, b => 2.0, w => 4.0);
total_area : float := area(p) + area(c) + area(c)

p := c;
c := b;
total_area := area(p) + area(c) + area(c)
```

statisches Binden (*early binding*, *static dispatch*)

der statische Basistyp des aktuellen Parameters entscheidet

dynamisches Binden (*late binding*, *dynamic dispatch*)

der dynamische aktuelle Typ des aktuellen Parameters entscheidet

Gegenüberstellung

	ad-hoc Überladen	parametrische Polymorphie	Vererbung Überschreiben
Vereinbarung	f : $s = B_s$ f : $t = B_t$	f : t[?] = *B*	f : $s = B_s$ f : $t = B_t$ f : $u = B_u$
Anz. d. Monotypen	einige	unendlich viele	beliebig viele
Typverwandschaft	keine	—	*u* ? *t* ? *s*
Anwendung	... f(*a*) ...	... f(*a*) ...	... f(*a*) ...
Typkorrektheit	bt(*a*) ? ?*s*,*t*?	bt(*a*) ?????*s*?	bt(*a*) ? *s*
Bindung: statisch dynamisch	$B_{bt(a)}$	*B*	$B_{sst(bt(a))}$ $B_{sst(dt(a))}$