KONTEXTANALYSE

(Dienstag, den 8. Juni 1999)

- die Aufgabe 100
- Gültigkeitsbereiche (*scopes*) 103
- Attributgrammatiken106
- Gültigkeitsanalyse109
- lineare Sichtbarkeit 111
- totale Sichtbarkeit 116

(Dienstag, den 13. Juni 1999)

- Implementierung der Vereinbarungstabelle 122
- Vereinbarungstabelle mit überladenen Bezeichnern129
- Typsysteme 132
- Typäquivalenz 133
- Typen 134
- Typüberprüfung 135
- polymorphe Typinferenz 138

die Aufgabe

Kontextbedingungen

sind *syntaktische* Regeln für Programme,
die *nicht* kontext-frei definiert werden können
aber *vor* seiner Ausführung festgestellt werden können

oft mißverständlich (*statisch-*) *semantische Bedingungen* genannt

Beispiel (Zuweisung)

```
a[i,j] := 5;
```

Kontextbedingungen an `a`

- `a` ist ein zweidimensionales Feld
- `i` und `j` müssen vom Typ seiner Indizes sein
- `a`'s Elemente müssen ganze Zahlen sein

dynamische semantische Bedingung

- die Werte von `i` und `j` müssen im Indexbereich liegen

Grenzfälle

- `i` und `j` sind Konstanten, Indexbereiche sind konstant:
  dann ist auch diese Bedingung statisch

Scopes and Types

Kontextanalyse

Identifizierung (*scope analysis*)

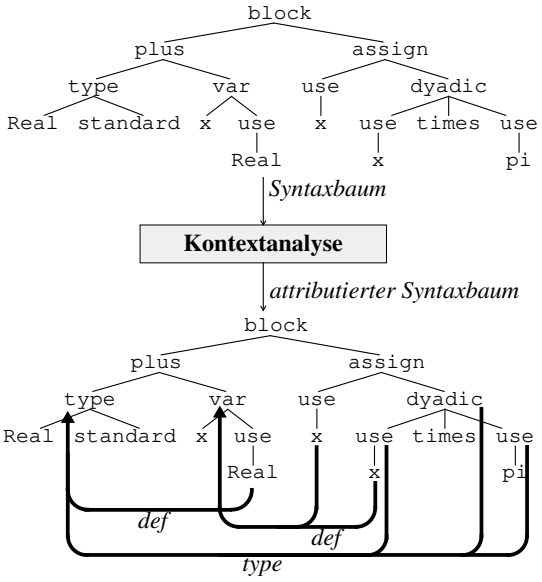
- ordne jedem Bezeichner seine Definition zu

Typ-Prüfung (*type checking*)

- bestimme den Typ jeden Ausdrucks

repräsentiere diese Information durch zusätzliche Verweise im Baum (*Attribute*)

- *definition* verweist auf die Vereinbarung eines benutzen Bezeichners
- *type* verweist auf die Vereinbarungdes Typs, den ein Ausdruck hat



Begriffe für die Kontextanalyse

Vereinbarung

bindet einen Bezeichner an ein Objekt
(und macht evtl. noch mehr, z.B. Speicher reservieren)

Bindung (eines Bezeichners *x*)

Auftreten von *x* (in einer Vereinbarung), daß x bindet

Benutzung (von *x*)

alle anderen Auftreten von *x*

Attribut (von *x*)

Eigenschaften von *x*, die mit seiner Vereinbarung festgelegt werden

Typ

Einteilung der Daten einer Programmiersprache in

Wertebereiche (mit geeigneten Operationen)
(typischerweise ein Teil des Atributs von *x*)

Art

alle weiteren Eigenschaften von *x*

Beispiel

```
type Line = array [1..linelength] of Char;
```

Bindung von `Line`, *Benutzung* von `linelength` und `Char`

`Line` bekommt die *Art* Typbezeichner und den *Typ* ...

Gültigkeitsbereiche

Block

ein Programmteil, der die Gültigkeit von Vereinbarungen begrenzen kann

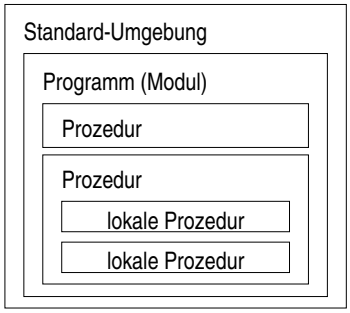
Blockstruktur

wie können Blöcke kombiniert werden?

- monolithisch (Cobol) ein einziger Block
- flach (Fortran) ein globaler plus unabhängige lokale
- geschachtelt (Algol usw.) beliebig rekursiv

typische Blöcke

- Standard-Umgebung
- Programm
- Modul
- Prozedurrumpf
- Anweisungsfolge (Algol, C, Ada)
- **`with`**-Anweisung



Namensüberdeckung (*hiding*)

lokale Vereinbarung

im Block getroffen

globale Vereinbarung

in einem textuell umgebenden Block getroffen

Namensüberdeckung

lokale Vereinbarugen überdecken
globale Vereinbarungen desselben Bezeichners

Annahmen

- statische Bindung
- statische Typisierung
- Monomorphie (jeder Bezeichner hat *genau eine* Vereinbarung)

Anfang der Sichtbarkeit

lineare Sichtbarkeit (Pascal, Ada)

textuell *nach* der Vereinbarung im Block

rekursive Sichtbarkeit (Algol, PL3)

gleich ab Anfang des Blockes

Beispiel

```
procedure P;
   const b = 3;
   ...
   procedure Q;
      const c = b;   <-- c=3 oder c=7?
            b = 7;
            ...
      begin ...
      end;
   begin ... end
```

Antworten

Ada: `b=3`

Pascal: **`const`** `c = b` *ist illegal*
(wird ein Bezeichner in einem Block vereinbart,
darf er darin nicht textuell vorher benutzt werden)

Algol, PL3: `b=7`

Spezifikation von Kontextbedingungen

Prinzip

Syntax-orientiert (an der *abstrakten* Syntax, also an *Syntaxbäumen*)

Knoten-orientiert (bzw. Regel-orientiert)

- *Attribut-Grammatiken* [Knuth 68]

Attribute

Werte, mit denen die Kontextbedingungen überprüft werden können

werden an die Knoten (Nichtterminale) geknüpft

Beispiele Vereinbarungstabelle, Typ

Auswertungsregeln

definieren die Werte der Attribute

- durch *Attributgleichungen*,
  die Attribute von Knoten mit denen ihrer Kinder in Beziehung setzen

Richtungsangabe für Attribute

wie bestimmt sich der Wert eines Attributs an einem Knoten?

- aus den Unterbäumen: aufsteigend, *abgeleitet* (*derived*)
- aus dem Kontext: absteigend, *geerbt* (*inherited*)

Auswertung

wie wird der Wert der Attribute berechnet?

- durch Anwendung der Auswertungsregeln beim Traversieren des Baumes

Attributgrammatiken

erfunden von D.E. Knuth (1968)

zur "Übersetzung kontextfreier Sprachen"

Syntax: $AG = (G, B, A, SR)$

- unterliegende kf Grammatik $G = (V, N, R, s)$
  mit Regeln der Form $n \to t n_1 \ldots n_k$ mit *Konstruktor t*
- *semantische Basis*:
  abstrakte Datentypen über einer Signatur $B = (S, \Sigma)$
- Eine Menge $A = \bigcup_{n \in N} A(n)$ von *Attributnamen*,
  die den Nichtterminalen zugeordnet sind
- Eine Menge $SR = \bigcup_{r \in R} SR(r)$ von *semantischen Regeln*,
  die den unterliegenden Regeln zugeordnet sind

  semantische Regeln haben die Form
  $$n_0 \cdot a_0 = F(n_1 \cdot a_1, \ldots, n_k \cdot a_k)$$
  wobei F eine semantische Funktion ist
  und die $n_i \cdot a_i$ *Attributvorkommen* in der Regel sind

Bedeutung (Semantik)

Die Bäume der unterliegenden Grammatik
werden so mit Attributinstanzen *dekoriert*,
so daß ihre Werte die semantischen Regeln erfüllen

Auswertung von Attributgrammatiken

Ergänzung der Definition

zur leichteren Implementierung

Atribute werden unterteilt in
geerbete (*inherited*) und
synthetisierte Attribute

(Analogie zu Prozedurparametern:
geerbt = Wert-P., synthetisiert = Ergebnis-P.)

die semantische Regeln werden so abgefaßt, daß

- geerbte Attribute im Kontext eines Knotens und
- synthetisierte Attribute im Unterbaumdes Knotens

bestimmt werden

Attributauswertung

Traversieren des Syntaxbaumes

mit Ausführung der semantischen Regeln als Zuweisungen

☞ ABER: jedes Attribut wird nur einmal zugewiesen
(deklarative Semantik)

Gültigkeitsanalyse: semantische Basis

Vereinbarungstabelle

als "abstrakter Datentyp" in Haskell

Prototyp für die Implementierung

(die Implementierung muß *effizienter* sein!)

```
type TAB = [[DCL]]
type ID = String
type DCL = (ID,KND)
data KND = Unbound | Typ | Const | Var | Proc
         deriving (Eq, Text)

initial :: TAB
initial = [[("read",Proc), ("write",Proc)]]

nest :: TAB -> TAB
nest t = []:t

unnest :: TAB -> TAB
unnest (l:t) = t

enter :: TAB -> DCL -> TAB
enter (l:t) (x,k) = if (is_local l x)
                    then error "double definition"
                    else ((x,k):l):t

definition :: TAB -> ID -> KND
definition t x
   = if dcls /= []
     then (snd (head x_d))
     else Unbound
     where
       x_d = (filter (\y->x==(fst y)) (concat t))

is_local :: [DCL] -> ID -> Bool
is_local l x = any (\y->x == (fst y)) l
```


Gültigkeitsanalyse: abstrakte Syntax

unterliegende kontextfreie Grammatik

Regeln haben die Form $n \to t n_1 \ldots n_k$

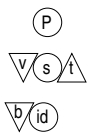
wobei t Konstruktor ist und die n_i Baumtypen

hier beschrieben als *algebraischer Typ* in Miranda

```
p ::=prog s
s ::=use ident | decl ident kind
   | block s   | list first: s rest: s
```

entspricht einem Datentyp (**pointer to record** ...) in Oberon etc.

Attribute für die Definition der linearen Sichtbarkeit

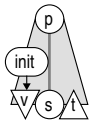



```
vis: inh tab of s;
total: syn tab of s;
binding: exp syn kind of ident;
```

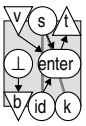
lineare Sichtbarkeit

Auswertungsregeln Abhängigkeitsgraphen

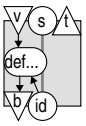
```
p ::= prog s
      vis(s) = initial
```



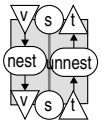

```
s ::= decl ident kind
      total(s) = enter(ident, d, vis(s))
      binding(ident) = unbound
```



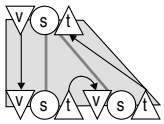

```
    | use ident
      binding(ident)
          = definition(ident,vis(s))
      total(s) = vis(s)
      check binding(ident) # unbound
```




```
    | block s
      vis(s) = nest(vis(block))
      total(block) = unnest(total(s))
```



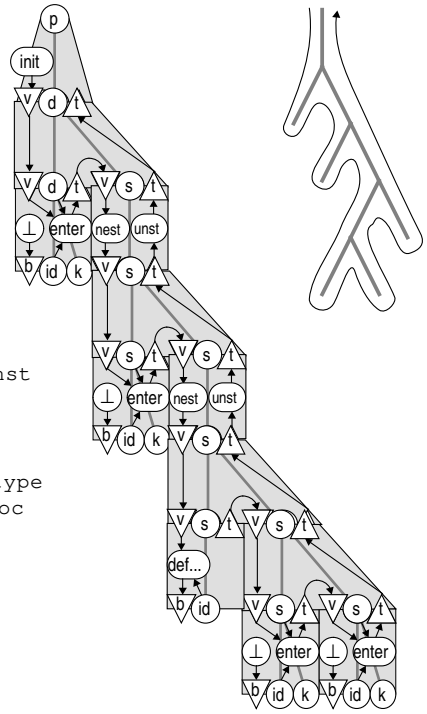

```
    | list first: s rest: s
      vis(first) = vis(list)
      vis(rest) = total(first)
      total(list) = total(rest)
```



lineare Sichtbarkeit: Attributierung

Beispielprogramm (-baum)

```
prog
 block
  decl x: var
  block
    decl y : const
    block
     list
      use x
      decl t: type
     decl x: proc
```



Welche Vereinbarung wird in `use x` benutzt?

```
decl x : var
```


Attributauswerter für lineare Sichtbarkeit in Haskell

underlying (abstract) program syntax

```
data PR = Prog SS
data SS = Use ID KND  | Block SS
        | Decl ID KND | List SS SS
```

linear scope analysis by an "attribute grammar"

```
eval_PR :: PR -> PR
eval_PR (Prog s)
   = Prog s'
     where (s',g) = eval_S initial s

eval_S:: TAB -> SS -> (SS,TAB)
eval_S v (Use x k )
   = (Use x k',v)
     where k' = definition v x
eval_S v (Block s )
   = (Block s',g')
     where (s',g)  = eval_S (nest v) s
           g'      = unnest g
eval_S v (Decl x k )
   = (Decl x k, enter v (x,k))
eval_S v (List s1 s2)
   = (List s1' s2',g')
     where (s1',g) = eval_S v s1
           (s2',g')= eval_S g s2
```


Attributauswerter für lineare Sichtbarkeit in Pascal

rekursive Prozeduren

```
procedure check_P(t: P_tree);
  var tot: tab;
begin
  tot:= check_S(t^.body, initial)
end

procedure check_S(t: S_tree; vis: tab): tab;
  var tot: tab;
begin
  case t^.kind of
    decl:
      return enter(t^.ident, t, vis)
  | use:
      t^.ident^.binding:= definition(t^.ident, vis);
      if t^.ident^.binding = unbound
      then error end;
      return vis
  | block:
      return unnest(check_S(t^.stmt, nest(vis)))
  | list:
      tot:= check_S(t^.first, vis);
      return check_S(t^.rest, tot)
  else compiler_error
  end
end
```

optimierter Attributauswerter (lineare Sichtbarkeit)

Die Tabelle kann als globale Variable realisiert werden

```
var env: tab;

procedure check_P(t: P_tree);
   var g: tab;
begin
   env:= initial; check_S(t^.body)
end

procedure check_S(t: S_tree);
begin
   case t^.kind of
      decl:
         env:= enter(t^.ident, t, env)
      use:
         t^.ident^.binding:= definition(t^.ident, env);
         if t^.ident^.binding = unbound
         then error end;
   | block:
         env:= nest(env);
         check_S(t^.stmt);
         env:= unnest(env)
   | list:
         check_S(t^.first);
         check_S(t^.rest)
   else compiler_error
   end
end
```


totale Sichtbarkeit

Problem

in einer Vereinbarung dürfen *alle* lokalen Bindungen benutzt werden

auch wenn sie im selben Block *textuell* danach stehen

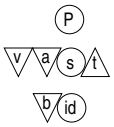
sichtbar sind alle Vereinbarungen, die im Block aufgesammelt werden

Deshalb

1. Phase: lokale Vereinbarungen sammeln
2. Phase: lokale Vereinbarungen prüfen

Attribute

```
vis: inh tab of s;
glob: der tab of s;
acc: inh tab of s;
binding: der kind of ident;
```

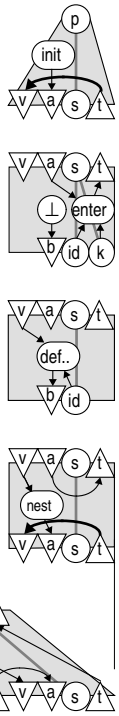


totale Sichtbarkeit: Auswertungsregeln

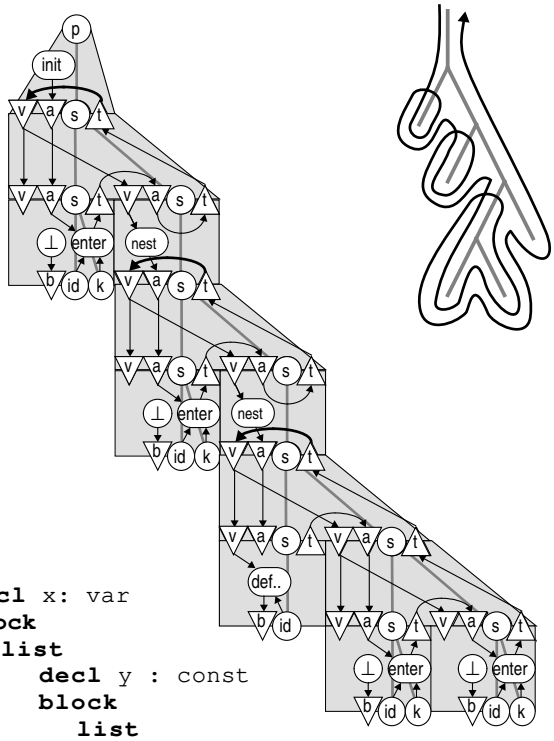
Auswertungsregeln

```
p ::= prog s
      acc(s) = initial
      vis(s) = total(s)

s ::= decl ident kind
      total(s)
          = enter(ident, d, acc(s))
      binding(ident) = unbound

    | use ident
      binding(ident)
          = definition(ident,vis(s))
      total(s) = acc(s)
      check binding(ident) # unbound

    | block s
      acc(s) = nest(vis(block))
      vis(s) = total(s)
      total(block) = unnest(total(s))

    | list first: s rest: s
      acc(first) = acc(list)
      acc(rest) = total(first)
      total(list) = total(rest)
      vis(first) = vis(list)
      vis(rest)  = vis(list)
```

Abhängigkeitsgraphen



Beispielprogramm (-baum)




```
prog
   list
      decl x: var
      block
         list
            decl y : const
            block
               list
                  use x
                  list
                     decl t: type
                     decl x: proc
```

Attributauswerter für totale Sichtbarkeit in Pascal

abstrakte Syntaxbäume

```
data PR = Prog SS
data SS = Use ID KND | Block SS
| Decl ID KND | List SS SS
```

Attributierungsregeln

```
eval_PR :: PR -> PR
eval_PR (Prog s)
  = Prog s'
    where (s',g) = eval_S (g,initial) s
eval_S:: (TAB,TAB) -> SS -> (SS,TAB)
eval_S (v,a) (Use x k )
   =  (Use x k',a)
      where k' = definition v x
eval_S (v,a) (Block s )
   =  (Block s',g')
      where (s',g)   = eval_S (g,nest a) s
            g'       = unnest g
eval_S (v,a) (Decl x k )
   =  (Decl x k, enter a (x,k))
eval_S (v,a) (List s1 s2)
   =  (List s1' s2',g')
      where (s1',g)  = eval_S (v,a) s1
            (s2',g') = eval_S (v,g) s2
```


Attributauswerter für totale Sichtbarkeit in Pascal

zwei Besuche: Sammeln und Prüfen

```
procedure check_P(t: P_tree);
  var vis: tab;
begin
  vis:= collect_S(t^.body, initial);
  check_S(t^.body, vis)
end

procedure collect_S(t: S_tree; acc: tab): tab;
  var tot: tab;
begin
  case t^.kind of
    use, block: return acc
  | decl:       return enter(t^.ident, t, acc)
  | list:       tot:= collect_S(t^.first, acc);
                return collect_S(t^.rest, tot)
  else compiler_error
  end
end

procedure check_S(t: S_tree; vis: tab);
  var tot: tab;
begin
  case t^.kind of
    decl:
    use: t^.ident^.binding:=definition(t^.ident, vis);
         if t^.ident^.binding = unbound
         then error end
  | block: tot:= collect_S(t^.stmt, nest(vis));
         check_S(t^.stmt, tot)
  | list:  check_S(t^.first, vis);
         check_S(t^.rest, vis)
  else compiler_error
  end
end
```


optimierter Attributauswerter (totale Sichtbarkeit)

```
var env: tab;

procedure check_P(t: P_tree);
begin
  env:= initial; collect_S(t^.body); check_S(t^.body)
end

procedure collect_S(t: S_tree);
begin
  case t^.kind of
    use,
    block:
    decl:       env:= enter(t^.ident, t, env)
  | list:       collect_S(t^.first);
                collect_S(t^.rest)
  else compiler_error
  end
end

procedure check_S(t: S_tree);
begin
  case t^.kind of
    decl:
    use:
      t^.ident^.binding:= definition(t^.ident, env);
      if t^.ident^.binding = unbound
      then error end;
  | block:
      env:= nest(env);
      collect_S(t^.stmt);
      check_S(t^.stmt)
  | list:
      check_S(t^.first);
      check_S(t^.rest)
  else compiler_error
  end
end
```


Vereinbarungstabelle als Liste

Prototyp (Spezifikation in Haskell)

lineare Liste von Bezeichnern und Arten

(spezieller Eintrag für Blockanfänge)

```
tab  ==  [decl]
decl == (ident, kind)
kind ::= unbound | block | …

initial:: tab % Standard-Umgebung
initial
=  [("",block),  ("read",  proc…),  ("write",
proc…), …]

nest:: tab->tab & Blockeintritt
nest t = ("",block):t

unnest:: tab->tab % Blockaustritt
unnest t = tail(forgetuntil "" t)

enter:: ident kind tab->tab % eintragen
enter x k t = (x,k):t, if notlocal x t
   = error "double declaration", otherwise

definition::ident tab->kind % abfragen
definition x t = snd (head rest), if rest <> []
   unbound, otherwise
   where rest = forgetuntil x t

not_local::   ident   tab->bool   %   doppelt
vereinbart?
not_local x (d:t) = true, if d = ("", block)
   = false, if x = fst d
   = not_local x t, otherwise

forgetuntil x []= []

forgetuntil x (y,d): t)
   = (y,d): t), if x=y
   = forgetuntil x t, otherwise
```

Komplexitätsüberlegungen

Aufwand für einige wichtige Funktionen

(*d* = # Vereinbarungen, *i* = # Bezeichner,
l = # lokale Vereinb., *n* = maximale Schachtelungstiefe)

`nest`: konstant

`unnest`: linear in *l*

`enter`: linear in *l*

`definition`: linear in *d*

`not_local`: linear in *l*

Das ist zu langsam

`definition` und `enter` (`not_local`) werden oft aufgerufen

Gültigkeitsanalyse hätte *quadratischen* Aufwand!

Ursache

Listen haben *sequenziellen* Zugriff

Abhilfe

Strukturen mit direktem Zugriff: endliche Abbildungen (Felder)

Problem

`Id → Decl` ist zu einfach wegen der Blockschachtelung

Varianten:

- `Id → stack(Decl)`
- `stack(Id → Decl)`

Vereinbarungstabelle als `Id → stack(Decl)`

```
type Tab = row Id of stack(num, decl)
var env: Tab; curr_lev: Integer;
procedure initial is
   for all x in Id
   do env(x).lev := unbound;
   enter ("read", proc…);
   enter("write", proc…);
   … ;
procedure nest is
   incr curr_lev
procedure unnest is
   for all x in Id
   do if env(x).lev = curr_lev
      then env(x) := pop(env(x) end;
   decr curr_lev
procedure enter (x: Id; k: Kind) is
   if not_local(x)
   then env[x] := push(x, k, env[x]):t
   else error "double declaration"
procedure definition(x: Id) return Kind is
   return top(env[x])
procedure not_local (x: Id) return Bool is
   return env(x).lev # curr_lev
```


Vereinbarungstabelle als `stack(Id → Decl)`

```
type Tab = stack(row Id of Decl);
var env: Tab;
procedure initial is
     var loc: row Id of Decl;
   for all x in Id
   do loc(x).lev := unbound;
   enter ("read", proc…);
   enter("write", proc…);
   … ;
   empty(env);
   push(loc, env)
procedure nest is
     var loc: row Id of Decl;
   for all x in Id
   do loc(x).lev := unbound;
   push(loc, env)
procedure unnest is
   pop(env)
procedure enter (x: Id; k: Kind) is
   if not_local(x)
   then top(env)[x] := k
   else error "double declaration"
procedure definition(x: Id) return Kind is
   var loc:  row Id of Decl;
       glob: Tab;
   repeat
      loc := top(env);
      glob := rest(env);
   until rest = nil or loc[x] # unbound;
   return loc(x)
procedure not_local (x: Id) return Bool is
   return top(env)(x) = unbound
```

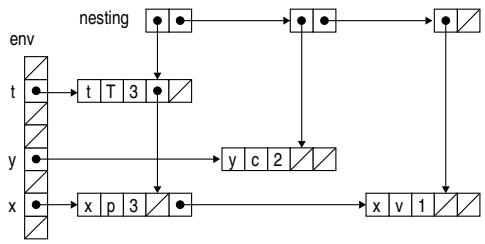

gitterartige Vereinbarungstabelle

```
type Entry = pointer to record
               decl: Decl;
               level: Integer;
               nextlocal,
               name: Id;
               global: Entry;
             end;
     Tab =   array Id of Entry;
     Blocks =pointer to record
               top: Entry;
               rest: Stack
             end;
```

Organisation

doppelte Verkettung der Einträge nach Bezeichnern und Schachtelung

```
prog
 list
  decl x: var
  block
   list
    decl y : const
    block
     list
      decl x: proc
      list
       decl t: type
       use x
```



gitterartige Tabelle: Operationen

```
var env: Tab;
    nesting: Blocks:= nil;
    level: Integer:= 0;

procedure initial is
   for all x in Id do env(x) := nil;
   enter ("read", proc…);
   enter("write", proc…); … ;
   nest;

procedure nest is
   nesting := new (nil, nesting); incr level

procedure unnest is
     var loc: Entry;
   loc:= nesting^.top;
   while loc # nil
   do env[loc^.name] := loc^.name.global;
      loc := loc^.nextlocal;
   end; { dispose(nesting^.top); }
   nexting:= nesting^.global;
   decr level

procedure enter (x: Id; k: Kind) is
   if not_local(x)
   then env[x]:=
          new(x, k, level, nesting^.top, env[x])
      nesting^.top:= env[x]
   else error "double declaration"

procedure definition(x: Id) return Kind is
     return env(x)^.decl

procedure not_local (x: Id) return Bool is
   return env(x)^.lev # level
```


Aufwandsvergleich

Parameter

d = # Vereinbarungen

i = # Bezeichner

l = # lokale Vereinb

n = maximale Schachtelungstiefe

k = # vordefinierte Vereinbarungen

Prozedur	list	row(stack)	stack(row)	Gitter
`initial`	k	i	i	k
`nest`	c	c	i	c
`unnest`	l	i	c	l
`enter`	l	c	c	c
`not _local`	l	c	c	c
`definition`	d	c	n	c

Aufwand der Gültigkeitsanalyse

linear (bei gitterartiger Implementierung)

praktische Realisierung

In der Bezeichnertabelle wird für jeden Bezeichner
ein Verweis auf `Entry` reserviert (und auf **`nil`** gesetzt)

Zugriff: `x.entry` statt `env(x)`

überladene Vereinbarungstabelle (1)

Überladen (ad-hoc-Polymorphie)

Bezeichner können *gleichzeitig*
an *mehrere* Vereinbarungen gebunden werden!

Beispiele: Operatoren, Prozeduren, Konstanten

Forderung
diese Vereinbarungen müssen unterscheidbar sein

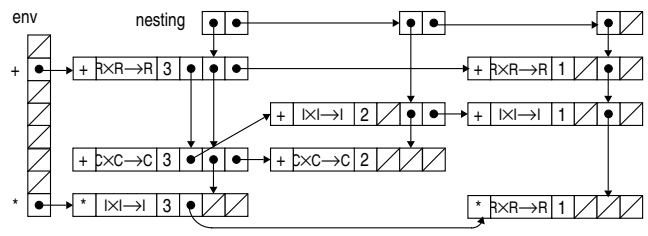
Organisation

zusätzliche Verkettung der *anderen* Vereinbarungen

unterscheidbare neue Vereinbarungen werden hinzugefügt

nicht unterscheidbare Vereinbarungen werden überdeckt

`definition` liefert eine Kette von Einträgen
(nur die Felder `other` und `decl` sollen benutzt werden)



überladene Vereinbarungstabelle (2)

```
type Entry = pointer to record
                name: Id;
                decl: Decl;
                level: Integer;
                nextlocal, other, global: Entry;
             end;
     Tab =   array Id of Entry;
     Blocks =pointer to record
                top: Entry;
                rest: Stack
             end;

var env: Tab;
    nesting: Blocks:= nil;
    level: Integer:= 0;

procedure  initial; …
procedure nest; …

procedure unnest is
   var loc, p : Entry;
begin
loc:= nesting^.top;
while loc # nil
do if loc^.glob = nil
   then env[x]:= loc^.other
   else p:= loc^.glob^.other;
        loc^.glob^.other:=loc^.other;
        if    p = nil
        then  env[x]:= loc^.glob
        else  p^.other:= loc^.glob;
        end;
   loc := loc^.nextlocal
   end
end;
```

überladene Vereinbarungstabelle (3)

```
nexting:= nesting^.global;
decr level
end;

procedure enter (x: Id; d: Decl) is
   var g, e, p: Entry; sim: Boolean;
   begin
   sim:= false;
   g:= env[x];

   p:= nil;
   while s # nil
   do sim:= confusable(d, g^.decl);
      p:= g;
      g:= g^.other
   end;
   if g = nil
   thenenv[x]:=
      new(x, d, curr_lev, nesting^.top, env[x],
nil);
      nesting^.top:= env[x]
   else if g^.lev # curr_lev
   then p^.other :=
      new(x,    d,    curr_lev,    nesting^.top,
g^.other, g);
      g^.other:= p;
      nesting^.top:= env[x]
   else error "double declaration"

   end;

procedure definition(x: Id): Entry;
   ...
```


Typsysteme

Wertemenge

Menge aller Daten,
die in der Programmiersprache berechnet werden kann

Typ

(Teil-) Menge von Werten, mit Operationen darauf

Typsystem

Regeln für Operationen auf Werten

Typisierung

statisch: vom Übersetzung vor der Ausführung überprüft

dynamisch: Bei der Ausführung überprüft

statisch ist sicherer (und heute üblich)

explizit: Alle Typen müssen spezifiziert werden
(Pascal, Ada usw.)

implizit: die Typen werden hergeleitet (Inferenz)
(ML, Miranda, Gofer usw.)

Typäquivalenz

textuell

Die Typen haben gleichlautende Vereinbarungen

strukturell

die Typen haben die gleiche Darstellung

namentlich

Die Typen sind in derselbven Vereinbarung definiert

Beispiel

```
type α = array [1..10] of β;
     β = array [0..99] of Integer;
     γ = array [1..10] of [0..99] of Integer;
     δ = array [0..99] of Integer;
var A, A': α; B: β; c: γ; D: δ;
```

textuell gleich: `B` und `D`

strukturell gleich: `A` und `C` (sowie `B` und `D`)

namens-gleich: `A` und **A**'

Vor- und Nachteile

strukturelle Gleichheit
- hart zu überprüfen (z. B. bei rekursiven Typen)
- unverträglich mit *nformation hiding*

Namengleichheit
- manchmal zu streng (Zeichenketten)

Typen

Modell		Operationen	Gesetze	
T ::=	Error	$\perp$	Error + T = Error, ...	*Basistypen*
|	Bool	true, false, not, and, or		
|	Int	0, 1, +, -, *, div, mod		
|	Real	0, 1, +, -, *, /		
|	Void	—		
|	$S \times T$	select$_i$: $\alpha_1 \times .. \times \alpha_n \to \alpha_i$	$T \times$ Void = T,	*Konstruktoren*
|	$S + T$			
|	$S \to T$	apply: $\alpha \to \beta \times \alpha \to \beta$		
|	Row T	subscribe$_i$: Row $\alpha \times$ Int $\to \alpha$		
|	Ref T	deref: Ref $\alpha \to \alpha$		
		assign: Ref $\alpha \times \alpha \to \alpha$		
		if: Bool $\times \alpha \times \alpha \to \alpha$		

Beispiele

Row (Row Int))

$\alpha \to \beta$ (= $\forall\alpha, \beta$: $\alpha \to \beta$)

(Int $\times$ Real) + (Real $\times$ Void) + Error = (Int $\times$ Real) + Real + Error = Error

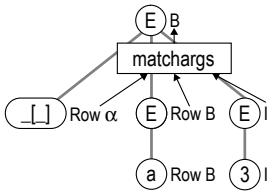
Typüberprüfung ohne Überladen

Attribute

```
type: der Type of E; imp Type of X;

E ::= | use X
        type(E) = type(X)
E ::= | apply X E1 ... En
        type(E) =
        matchargs (type(X), type(E1) × ... × type(En))

matchargs(T' → S, T) = σ(S) where ∃mgu σ: T=σ(T)
```

Beispiel

```
apply "_[_]" (use "a") (use "3")
```




```
matchargs(Row α× I → α, Row B × I) = σ(α) = B
where ∃mgu σ = {α→B}: Row B× I =σ(Row α× I)
```

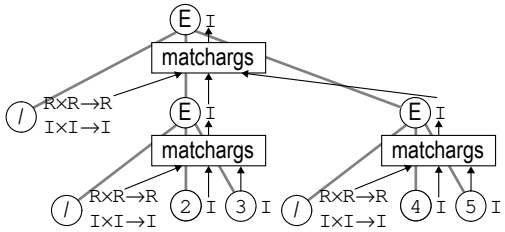

Typüberprüfung mit kontextfreiem Überladen

Attribute

```
type: der Type of E; imp Type of X;

E ::= | use X
        type(E) = type(X)
E ::= | apply X E1 ... En
        type(E) =
        matchargs (type(X), type(E1) × ... × type(En))

matchargs(TS, T) = σ(S)
      where ∃T' → S ∈ TS:∃ mgu σ: T=σ(T)
```

Beispiel

```
apply "_[_]" (use "a") (use "3")
```



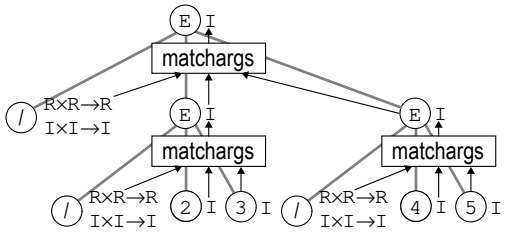
Typüberprüfung mit kontextsensitivem Überladen

Attribute

```
type: der Type of E; imp Type of X
res:  inh Type of E; inh Type of X

E ::= | use X
        type(E) = type(X)
        res(X)    = res(E)
E ::= | apply X E1 ... En
        type(E) = resulttype(F)
        res(X) = filter(res(E))
        res(E1) = arg(1, (res(X))
        ...
        res(En) = arg(n, (res(X))
        where
        F = matchfuns (type(X), type(E1) × ... × type(En))
        such that unique res(X)
matchfuns(TS, T)
        = { σ(T' →S) | ∃T' → S ∈ TS:∃ mgu σ: T=σ(T) }
```

Beispiel

```
apply "_[_]" (use "a") (use "3")
```



polymorphe Typinferenz

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)