Typäquivalenz, Beispiel

```
TYPE t1 = ARRAY [0..10] OF Integer;
TYPE t2 = ARRAY [0..10] OF Integer;
TYPE t3 = t2;

TYPE t4 = RECORD c: Integer;
                 p: POINTER TO t4;
           END RECORD;

TYPE t5 = RECORD c: Integer;
                 p: POINTER TO t4;
           END RECORD;

TYPE t6 = RECORD c: Integer;
                 p: POINTER TO RECORD c: Integer;
                                      p: POINTER TO t6;
                                 END RECORD;
           END RECORD;
```

Welche Typen sollen gleich (austauschbar) sein?

Typäquivalenz, zwei Arten

- Sprachen mit *struktureller Äquivalenz*:
  - Gleichheit induktiv definiert (erlaubt echte rekursive Typen)

    (a) ein primitiver Typ ist genau sich selbst gleich

    (b) zwei zusammengesetzte Typen sind gleich, wenn beide Typkonstruktoren und die Teilstrukturen elementweise gleich sind
  - verwendet bei funktionalen Programmiersprachen (z.B. Haskell): Typinferenz durch Unifikation (Gleichungslösen in Termalgebra)
- Sprachen mit *Namensäquivalenz*:
  - zwei Typen `t1` und `t2` sind dann und nur dann gleich, wenn sie ausdrücklich als gleich vereinbart wurden (`TYPE t2=t1`)
  - einfacher zu implementieren
  - stärkere Typtrennung wichtig bei Objekt-orientierten Sprachen: Typ(≙Klasse) zur Fallunterscheidung mittels dynamischer Bindung

Typäquivalenz am Beispiel

- Namensäquivalenz: nur `t2` und `t3` sind gleich
- strukturell: zwei Äquivalenzklassen: `{t1,t2,t3}` und `{t4,t5,t6}`

```
TYPE t1 = ARRAY [0..10] OF Integer;
TYPE t2 = ARRAY [0..10] OF Integer;
TYPE t3 = t2;

TYPE t4 = RECORD c: Integer;
                 p: POINTER TO t4;
           END RECORD;
TYPE t5 = RECORD c: Integer;
                 p: POINTER TO t4;
           END RECORD;
TYPE t6 = RECORD c: Integer;
                 p: POINTER TO RECORD c: Integer;
                                   p: POINTER TO t6;
                                END RECORD;
           END RECORD;
```

Typumwandlungen, Arten

- Coercion
  - implizit (automatisch)
  - Bsp. Widening (Java): `int→long→float→double`
  - Gefahr: evtl. unklar, ob Integer oder Fliesskommaoperation
- (explicit) Conversion (vom Typ A zum Typ B)
  - angegeben durch den Programmierer
  - sauberste Form, Anwendung einer Funktion: A→B
    - Semantik klar und Problem-orientiert (z.B. `round`: bei positiven Zahlen bis `.49..` abrunden, ab `.50..` aufrunden)
- Cast
  - Typänderungsangabe explizit, aber Funktionalität implizit
  - Narrowing (Java): abschneiden, z.B. `(short)65535 → -1`
  - Downcast bei Objekten, z.B. aus heterogener `Object`-Menge

Arten impliziter Typumwandlung

(bei coercion oder cast)

- Widening:
  - Übergang auf eine Wertobermenge (`integer → real`)
  - nicht immer sicher (`long integer → real`)
- Narrowing:
  - Übergang auf eine Wertuntermenge (`real → integer`)
  - basiert auf Repräsentation (z.B. abschneiden von Bits)
- Dereferenzierung (implizit):
  - Interpretation einer Adresse als Wert (coercion)
  - Bsp.: Verwendung einer Variable in einem Ausdruck `(x+1)`
    (in imperativen Sprachen stehen Variablen nicht für Werte, sondern für Speicherplätze)

Herausforderungen impliziter Typumwandlung

- mehrere Umwandlungen hintereinander
  (`integer → real → complex`)
- Umwandlung kann vom Kontext abhängen
  - in C: (`float → int`)
    - Coercion wird verwendet für die gesamte rechte Seite einer Zuweisung, falls linke Seite vom Typ `int`
    - (sinnvollerweise!) verboten für Anpassung eines Operanden, die Coercion `int→float` dagegen kann in Ausdrücken vorkommen
- Auflösung von Mehrdeutigkeiten
  - z.B. ist `3+5` eine Addition auf `integer` oder `real`?
  - häufige Regel: widening nur eines Operanden und nur bis zu dem Typ, den der andere Operand ohne widening hat

Typbestimmung in Ausdrücken

- Zwei Algorithmen:
  1. sammelt in jedem Syntaxbaumknoten die Menge aller seiner möglichen Typen (Abb. 6.9)
  2. eliminiert sukzessive Typen, die nicht vorkommen können (Abb. 6.10)
- Beide Algorithmen sind sogenannte Hüllenalgorithmen:
  - Start gegeben durch Menge "Initializations"
  - Hinzufügen von Informationen durch "Inference rules"
  - es wird solange iteriert, bis ein Fixpunkt erreicht ist,
    (eine) Voraussetzung: Monotonie der Inferenzregeln
    deshalb hier zwei Algorithmen hintereinander:
    1. Algorithmus: fügt nur Typen hinzu
    2. Algorithmus: entfernt nur Typen

Typbestimmung / Phase 1

(Mengenerweiterung aufgrund von Coercions)

Data definitions:

Let each node in the expression have a variable type set S attached to it. Also, each node is associated with a non-variable context C.

Initializations:

The type set S of each operator node contains the result types of all identifications of the operator in it; the type set S of each leaf node contains the types of all identifications of the node. The context C of a node derives from the language manual.

Inference rules:

For each node N with context C, if its type set S contains a type T_1 which the context C allows to be coerced to a type T_2, T_2 must also be present in S.

Figure 6.9 The closure algorithm for identification in the presence of overloading and coercions, phase 1.

Typbestimmung / Phase 2

(Mengeneinschränkung basierend auf Kombination von Operanden)

Data definitions:

Let each node in the expression have a type set S attached to it.

Initializations:

Let the type set S of each node be filled by the algorithm of Figure 6.9.

Inference rules:

1. For each operator node N with type set S, if S contains a type T such that there is no operator identified in N that results in T and has operands T_1 and T_2 such that T_1 is in the type set of the left operand of N and T_2 is in the type set of the right operand of N, T is removed from S.
2. For each operand node N with type set S, if S contains a type T that is not compatible with at least one type of the operator that works on N, T is removed from S.

Figure 6.10 The closure algorithm for identification in the presence of overloading and coercions, phase 2.

Typbestimmung / Probleme

1. Algorithmus (sammelt in jedem Syntaxbaumknoten die Menge aller seiner Typen, die aufgrund von Coercions möglich sind)

   Probleme:

   (a) die für einen Knoten ermittelte Menge kann leer sein

   (b) der Algorithmus terminiert nicht, wenn durch Coercing unendlich viele Typen erreicht werden können (vgl. Fol. 3/20)

2. Algorithmus (eliminiert sukzessive Typen, die wegen Operanden-Kombinationen nicht vorkommen können)

   Problem: keine Auflösung von Mehrdeutigkeiten, vgl. Fol. 3/6: widening nur eines Operanden und nur bis zu dem Typ, den der andere Operand ohne widening hat

   Lösung durch Modifikation oder 3. Algorithmus

Kind-Checking: L- und R-Werte

- in imperativen Sprachen stehen Variablen für Speicherstellen
- L-Wert: (kann links von einer Zuweisung stehen), z.B.:
  - (indizierte) Variablen (`x = ..., a[x,y+3] = ...`)
  - keine L-Werte: `x+1`, `f(x)`
- R-Wert: (kann rechts von einer Zuweisung stehen), z.B.:
  - Ausdruck (`2+3`, `f(x)`)
  - keine R-Werte: `x`, `a[3,5]`
- Umwandlung:

  (a) automatisch: L→R mit Dereferenzierung (Speicherzugriff)

  (b) explizit: L→R ohne Dereferenzierung, sowie R→L,
  in manchen Sprachen (in C, aber nicht in Java), z.B. in C:
  `&x` liefert Adresse der Speicherstelle von `x`

Problematik: L- und R-Werte

Lifting (`Name→Wert`) implizit im arithmetischen Ausdruck

Beispiel: `p:=q` ⇒ Coercion “Dereferenzierung” nötig

Semantik: `insertEnv(p,lookupEnv(q))`

Assemblercode:

```
Load_Mem  q,R1
Store_Reg R1,p
```

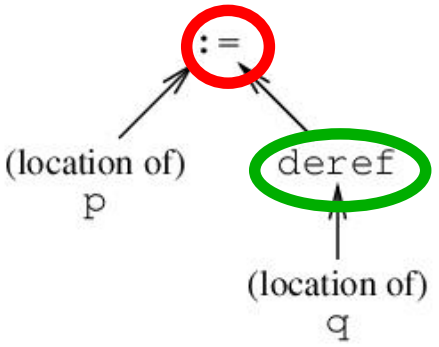

Figure 6.11 AST for `p := q` with explicit deref.

Inferenz der Art des Namens

- einzige Komplikation:
  - der i.d.R. nicht denotierte Unterschied zwischen L- und R-Werten
  - soll dereferenziert werden oder nicht?
- Bsp.: `a[1]` dereferenzieren?
  - `a[1]` ist Variable, also ein L-Wert:
    - `a[1]=x` L-Wert erwartet (nicht dereferenzieren)
    - `a[a[1]]` als Index R-Wert erwartet (dereferenzieren)
- L/R-Wertregeln: (Abb. 6.12, 6.13)

Coercion-Regeln für L/R-Werte

		expected	
		`lvalue`	`rvalue`
found	`lvalue`	-	`deref`
	`rvalue`	`error`	-

Figure 6.12 Basic checking rules for lvalues and rvalues.

Bedingungen an L/R-Werte

expression construct	*resulting kind*
`constant`	`rvalue`
`identifier` (variable)	`lvalue`
`identifier` (otherwise)	`rvalue`
`&lvalue`	`rvalue`
`*rvalue`	`lvalue`
V`[rvalue]`	*V*
V`.selector`	*V*
`rvalue+rvalue`	`rvalue`
`lvalue:=rvalue`	`rvalue`

Figure 6.13 lvalue/rvalue requirements and results of some expression constructs.

Datenstrukturen in imperativen Programmen

- Basistypen: direkt durch Maschinenkonzepte realisiert
  - Zahlentypen (`int`, `long`, `float` (32 bit), `double` (64 bit))
  - Zeichen: `char`
  - leerer Typ (`void`) (taucht im Zielcode nicht auf)
  - in Java, nicht in C: `boolean`
- Aufzählungstypen: direkt durch Maschinenkonzepte realisiert
  - total geordnet (im Zielcode durch Integer repräsentiert)
  - Operationen:
    - kopieren
    - Gleichheitstest, Größenvergleich
    - Vorgänger (`pred`) und Nachfolger (`succ`)
    - in einigen Sprachen auch Ein-/Ausgabe
  - Bsp. (in C, nicht in Java(s.o.): `bool` (`false`≙`0`,`true`≙`1`)

Zeiger-/Referenztypen

- Operationen
  - erzeugen
    - direkt in Java (`new`)
    - indirekt in C (`malloc(sizeof(...))`)
  - kopieren, zuweisen
  - Gleichheitstest, Test auf Null
  - dereferenzieren
  - Quotieren (Adressoperator `&` in C)
- getypte Zeiger, z.B. Referenzen in Java:

  `MyObj p = new MyObj();` p zeigt nur auf `MyObj`-Objekte
- generische Zeiger:
  - C: `*void`
  - Modula-2: `pointer to byte`

Adressoperator `&` in C, `ref`-Notation

- Operand: L-Wert (Variable)
- Ergebnis: R-Wert (der Zeiger auf die Variable)
- Beispiel:

```
int x    = 42;
int adr  = &x;   Warnung: Zeiger ist vom Typ int*, nicht int
int* ptr = &x;   richtig  (int* : Typ von Zeigern auf int)
adr = (int)ptr;  Cast: von Zeiger nach int
```

- `ref`-Notation
  - `ref` Bezeichnung für Referenz in Algol68
  - verwendet zum Zählen von Indirektionen (Zeiger von Zeiger)
  - Int-Variable (L-Wert) hat Algol68-Typ `ref int`
  - wenn `x` vom Typ `T`, dann `&x` vom Typ `ref T`

Indirektionsoperator in C

- Operand: R-Wert (Zeiger *p*)
- Ergebnis: L-Wert (des Datenobjekts, auf das *p* verweist)
- Bsp. 1:

```
int x        = 42;     x: ref int
int* ptr     = &x;     ptr Zeiger auf x
                       ptr: ref ref int
int** ptr2   = &ptr;   ptr2 Zeiger auf ptr
                       ptr2: ref ref ref int
int* ptr3    = *ptr2;  ptr3 gleich ptr (* Indirektion)
                       ptr3: ref ref int
```

- Bsp. 2 (benutze Speicherstellen `[a+3,a+2,a+1,a]` (32bit))

```
*((int *)adr) = 43;          schreibe 43 (code für '+')
char c = *((char *)adr);     danach: c enthält '+'
```

`ref`-Notation und Coercing

C-Quelltext	Aus-druck	(Bsp.) L-Wert	R-Wert	Kind und Typ
`int x = 42;`	`x`	`1000`	`42`	`ref int`
`int* p;`	`p`	`1004`	`--`	`ref ref int`
`p = &x;`	`&x`	`--`	`1000`	`int` (Adresse)
	`p`	`1004`	`1000`	`ref ref int`
`*p = 123;`	`*p`	`1000`	`123`	`ref int`
	`x`	`1000`	`123`	`ref int`

Wie passen `&x : int` und `p : ref ref int` zusammen?

- die linke Seite von = muss mindestens einen `ref` mehr haben als die rechte ✓
- die rechte Seite von = wird so oft coerced (hinzufügen von `ref`), bis sie genau einen `ref` weniger hat als die linke (hier: einmal)

Dereferenzieren in C

- Implementierung: betrachte Zeiger `p` auf ein Datum vom Typ `T`
  - T-Werte von Wortlänge: Dereferenzieren ein Maschinenbefehl
  - T-Werte länger als ein Maschinenwort:
    - R-Wert wird auf dem Laufzeitstack abgelegt
    - Iteration oder Folge von Maschinenbefehlen:
      `q = *p;` → `byte_copy(&q, p, sizeof(T));`
  - Zugriff auf eine Record-Komponente: `ptr->field`

    (a) gesamter Record auf Stack, dann Selektion: `(*ptr).field`

    (b) Zeiger auf die Komponente, dann Komponente auf Stack:

    ```
    *(&(ptr->field))
    ```

    Compiler kann Code für die effizientere Version (b) erzeugen

Korrupte Zeiger, Gegenmassnahmen

- Uninitialisierte Zeiger:
  - automatisch initialisieren (z.B. auf `null`) oder Fehlermeldung
- Dereferenzierung von Nullzeigern
  - Speicherzugriff auf Hardware-Ebene abfangen (trap)
  - zwei verschiedene Zeigerkonstruktoren
    - Pointer: darf `null` sein und darauf getestet werden (#)
    - Referenz (ist nie `null`)
      - darf mit einem Nicht-null-Pointer belegt werden (siehe #)
      - darf dereferenziert werden
- vorzeitig freigegebene Zeiger in den Heap:
  - Verbot expliziter Freigabe (kein `free`)
- Stackframe, auf das Zeiger verweisen, wird frei (neu belegt)
  - Verbot von Zeigern in den Stack (auf lokale Variablen)

Problem: Zeiger auf den Stack

```
void do_buffer(void) {
    char *buffer;

    obtain_buffer(&buffer);
    use_buffer(buffer);
}

void obtain_buffer(char **buffer_pointer) {
    char actual_buffer[256];

    *buffer_pointer = &actual_buffer;
        /* this is a scope-violating assignment: */
        /* scope of *buffer_pointer > scope of &actual_buffer */
}
```

Fehler im Buch
(Arrayname ist Adresse)

Figure 6.14 Example of a scope violation in C.

"Scope" eines Zeigers

- Definition: Scope eines Zeigers
  - ➔ der Existenzbereich der Stelle, auf die er zeigt
    (andere Bedeutung von Scope eines Datenobjekts: sein eigener Definitionsbereich)
- partielle Ordnung auf Zeiger-Scopes
  - Scope von *P* kleiner als Scope von *Q* (*P*<*Q*) ⇔ *Q* umschließt *P*
- Forderung:
  - Existenzbereich eines Zeigers kleinergleich seinem Scope (dem Existenzbereich des referenzierten Wertes), d.h.

`Existenzbereich(p) ≤ Existenzbereich(*p)`

Problem: Beendung eines Unterprogramms

- Ziel: verhindern, dass Zeiger auf Stackframes existieren können, nachdem diese durch Verlassen des Unterprogramms frei werden
- Scope-Definition:
  - Scope einer Stelle auf dem Heap und des statischen Programmdatenbereichs: unendlich
  - Scope der Stackframes: endlich
  - Scope eines Wertes: kleinster Scope der Zeiger, die er enthält (unendlich, falls er keinen Zeiger enthält)
- Einschränkung von Zeigerzuweisungen

  `p:=q` nur erlaubt wenn `Scope(p)≤Scope(q)`

  `(Existenzbereich(*p) ⊆ Existenzbereich(*q))`
- Konsequenzen: keine Zeiger
  - vom Heap in den Stack
  - von einem Bereich in ein von dort aufgerufenes Unterprogramm (vgl. Abb. 6.14)