Kapitel 5

Typisierung

Motivation

Überprüfung einer kontextsensitiven Eigenschaft nach dem Parsen.

- Ist das Programm korrekt?
- Z.B. : *int a = true;*

Relevante Information für Codegenerierung

- Welche Instruktionen müssen erzeugt werden?
- Z.B. : *x = y + f(x);* (Codegenerierung i.d.R. ohne Typisierung unmöglich)

Typen I

- Typüberprüfung
  - Typsynthese:

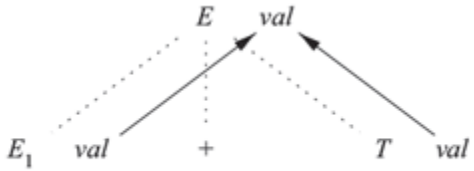


Abbildung 5.6: *E.val* wird aus E_1.*val* und E_2.*val* synthetisiert

    - Typ eines Ausdrucks anhand der Teilausdrücke abgeleitet
    - Falls f den Typ s → t hat und x den Typ s dann hat f(x) den Typ t
  - Typinferenz
    - Type wird anhand der Verwendungsweise festgestellt
    - f(x) kann vom Typ β sein falls f vom Typ α → β sein kann und x vom Typ α sein kann
    - basierend auf Unifikation und Typvariablen (α,β,...)

Typen II

- Typkonvertierung
  - Implizit (coercion) oder Explizit
  - **Erweiterend** (ohne Verlust) oder **Einengend** (möglicherweise mit Verlust)
    - int i = 2;
    - double d = i; erweiternd, implizit
    - i = (int) d; einengend, explizit (Cast)

In den meisten Sprachen: Implizite Konvertierung auf erweiternde eingeschränkt

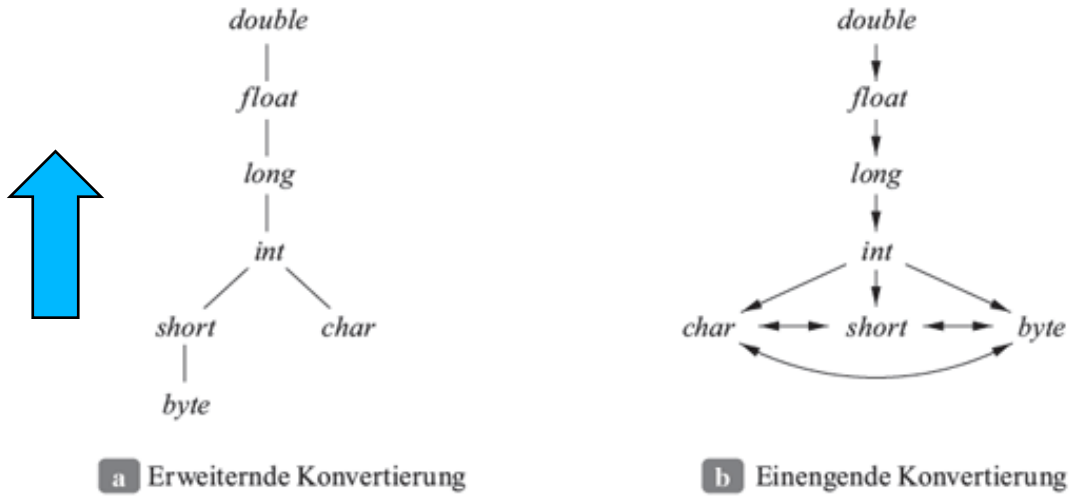


Abbildung 6.25: Konvertierungen zwischen primitiven Typen in Java

Überraschungen bei impliziten Konvertierungen JavaScript

2) JavaScript's loose typing and aggressive coercions exhibit odd behaviour.

[] + [] → "" // Empty string? These are arrays!
[] + {} → [object object]
{} + [] → 0 // Why isn't the operation commutative???
{} + {} → NaN // ???

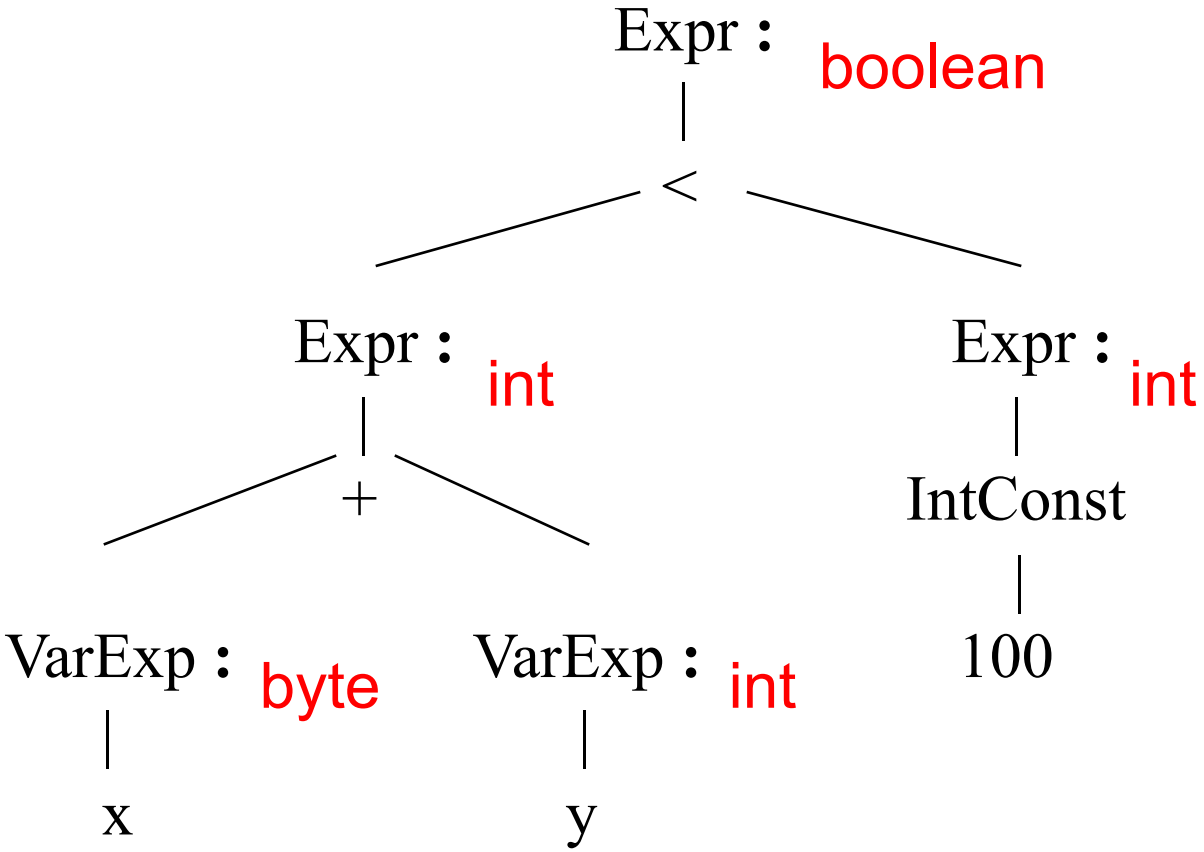
var i = 1;
i = i + ""; // Oops!
i + 1 → "11"
i - 1 → 0
var j = "1";
++j → 2 // Okay, but...

var k = "1";
k += 1 → "11" // What???

https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8

Typsynthese

- Bottom-up Analyse der Ausdrücke
- Anwendung von semantischen Regeln



Symboltabellen

- Datenstruktur die Typinformationen speichert (z.B. Hashmap).
- Löst das Problem der lokalen Sichtbarkeit von Symbolen (Variablen, Funktionen etc.)
- Implementierung ist Problemabhängig (statische/dynamische Typisierung, Blockstrukturen, innere Klassen etc.).
- Werden nicht nur zur Typisierung verwendet.

Symboltabellen 2

```
public class Scope
{
    static boolean a = true;
    static int b = 7;

    public static void f(boolean b) {
        System.out.println(b);
    }

    public static void main(String[] args){
        int a = 42;

        System.out.println(a);
        System.out.println(b);
        f(true);
    }
}
```

Scope:

Symbol	Typ
a	boolean
b	int

f:

Symbol	Typ
b	boolean

main:

Symbol	Typ
args	String[]
a	int

Scope/Methoden:

Symbol	Typ
f	[boolean] $\rightarrow$ void
main	[String[]] $\rightarrow$ void

Symboltabellen 3

```
program AFunction;
var
   a, b, result : integer;


function max(a, b: integer): integer;
var
   result: integer;
begin
   if (a > b) then
      result := a
   else
      result := b;
   max := result;
end;

function f(a, b: boolean): boolean;
begin
   f := a and b;
end;


begin
   a := 100;
   b := 200;
   result := max(a, b);

   writeln( 'Max value is : ', result );
   writeln( 'f : ',  f(true, false) );
end.
```

AFunction:

Symbol	Typ
a	integer
b	integer
result	integer

max:

Symbol	Typ
a	integer
b	integer
result	integer

f:

Symbol	Typ
a	boolean
b	boolean

AFunction/Funktionen:

Symbol	Typ
max	[integer, integer] $\rightarrow$ integer
f	[boolean, boolean] $\rightarrow$ boolean

Typ Inferenz

- ProB Demo

```
>>> :t {x,y| x:y & y(1)=2}
POW((INTEGER*INTEGER)*POW(INTEGER*INTEGER))
>>> {x|x \/ {1} = 2}
 → Type Error
```

- Haskell Demo

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

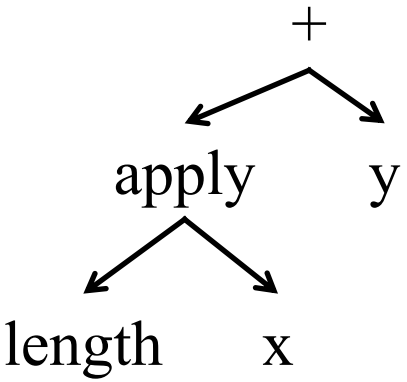
- C++
  - auto

```
int a = 2; int b=40;
auto c = a+b;
auto d = foo();
```

Typ Inferenz

- Basierend auf Unifikation (siehe Prolog)
  - Typen → logische Variablen
  - Unifikation wird verwendet um Typvariablen mit anderen Typvariablen oder konkreten Typen zu instanziieren
  - kann mit polymorphen Funktionen umgehen:
    - append : list(X) : list(X) → list(X)
    - map: (X→Y) : list(X) → list(Y)
- Hindley-Milner Algorithmus

Beispiel



- `foo x y = (length x) + y`
  - logische Variablen für x, y, (length x), (length x)+y, foo
  - length:: [a] -> Int
    - x of type [a] (list with type parameter a)
    - (length x) of type Int
  - + :: Int -> Int -> Int
    - y:: Int
    - return value of foo of type Int
  - Type of foo:
    - foo :: [a] -> Int -> Int

Funktionsaufrufe

```
boolean greater (int x, int y){
   return (x > y);
}
.
.
        if (greater(a, b)) { ... }
```

- Typ von `greater` ist int × int → boolean
- Um Anwendung zu überprüfen:
  - Typ von `greater` in Symboltabelle nachschauen
  - Typen der Parameter überprüfen
  - Ergebnistyp der Funktion (boolean) wird Ergebnistyp des Aufrufs

Dynamische Typisierung

```
def f(a,b):
    return a+b

print f(1,2)
print f("Hallo", "Welt")
print f(1, 3.14)
```

- Ausgabe:

3

Hallo Welt

4.14

- Nicht immer ist eine Typisierung zur Compilezeit möglich (statische Typisierung), sondern erst zu Laufzeit (dynamische Typisierung)

Zusammenfassung

- Typsynthese mit semantischen Aktionen
- Typumwandlung: verengend/erweiternd implizit/explizit
- Symboltabellen erstellen und nutzen
- Typinferenz mit Typvariablen
- Statische / dynamische Typisierung