Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

3. November 2015

Semantische Analyse

Ein Variablenname repräsentiert in einer Programmiersprache ein Objekt, das mehrere Eigenschaften besitzt:

1. Eine **Speicheradresse**, ab der dieses Objekt im Speicher abgelegt ist.
2. Eine Codierung des Objektes selbst, d.h. die zugeordneten Speichereinheiten und das darin abgelegte Bitmuster, das den **Wert** des Objektes darstellt.

Um den Wert eines Objektes manipulieren zu können, muss man wissen, wie lang diese Codierung ist und wie man sie zu interpretieren hat.

- Diese Information zu einem Objekt wird durch eine **Typ-Information** festgelegt und dem Objekt als tag oder der Variablen per Deklaration zugeordnet.
- In objektorientierten Sprachen wird der Begriff „Typ“ durch „Klasse“ ersetzt.
- Häufig wird die Typ-Information durch den Deklarationsteil eines Programms einem Variablennamen für einen gewissen Teil des Programms zugeordnet.

Typ-Prüfung

Ein Compiler kann in gewissem Rahmen prüfen, ob die durch die Sprachdefinition der Programmiersprache gegebenen Beschränkungen bzgl. der Typen im vorgelegten Programm eingehalten werden.

- Prüfung während der Übersetzung (**statische Typ-Prüfung**).
- Prüfung zur Laufzeit (**dynamische Typ-Prüfung**)

Es gibt auch Programmiersprachen, in denen keine Typ-Deklaration vorgesehen ist. In diesen Fällen ist jedem Datenobjekt eine Kodierung des Typs (bzw. der Klasse) zugeordnet. Diese Kodierung muss gespeichert werden (**tag-System**) und wird immer überprüft, wenn das Objekt als Operand einer Operation auftritt.

- Eine Programmiersprache mit **starker Typ-Prüfung** hat ein Typ-System, bei dem der Compiler im gewissen Umfang garantieren kann, dass beim übersetzten Programm kein Typ-Fehler zur Laufzeit auftritt.
- Wird der Typ eines Ausdrucks der Programmiersprache aus den vorher bekannten Typen der Teilausdrücke bestimmt, spricht man von **Typ-Synthese**. Diese Strategie erfordert eine Deklaration der Variablen vor dem Gebrauch.
- Den Prozeß des Ableitens von Typ-Information für Konstrukte der Programmiersprache aus dem Gebrauch der Konstrukte nennt man **Typ-Inferenz**.

Typ-Ausdrücke

Jedem Objekt und jedem Ausdruck im Programm wird vom Compiler ein Typ-Ausdruck zugeordnet.

Definition einfacher Typ-Ausdrücke

1. Ein Basistyp ist ein Typ-Ausdruck.
2. Sind Namen als Abkürzungen für Typen erlaubt, dann sind diese Namen ebenfalls Typ-Ausdrücke.
3. Ist T ein Typ-Ausdruck, so ist *array(T)* ebenfalls ein Typ-Ausdruck, der einen Feld-Typ beschreibt.
4. Ist T ein Typ-Ausdruck, so ist *list(T)* ebenfalls ein Typ-Ausdruck, der einen Listen-Typ beschreibt.

5. Sind $T_1, \ldots, T_r$ Typ-Ausdrücke mit $r \geq 1$ und sind $\texttt{name}_1, \ldots, \texttt{name}_r$ unterschiedliche Namen, dann ist
$$\mathit{record}(\mathit{name}_1{:}T_1, \ldots, \mathit{name}_r{:}T_r)$$
ein Typ-Ausdruck, der eine Record-Konstruktion beschreibt.
6. Sind $T_1, \ldots, T_r$ Typ-Ausdrücke mit $r \geq 1$, dann ist
$$T_1 \times \ldots \times T_r$$
ein Typ-Ausdruck, der ein Tupel mit r Komponenten von Typ $T_1, \ldots, T_r$ beschreibt.
Analog würde man einen Typ-Konstruktor für Klassen definieren.

7. Ist T ein Typ-Ausdruck, dann ist *pointer(T)* ein Typ-Ausdruck, der einen Zeigertyp beschreibt.
8. Sind T_1 und T_2 Typ-Ausdrücke, dann ist
$$T_1 \rightarrow T_2$$
ein Typ- Ausdruck, der eine Funktion mit einem Parametern vom Typ T_1 beschreibt, die ein Objekt vom Typ T_2 zurückgibt.

Bemerkung

Der Operator $\times$ ist linksassoziativ und hat eine größere Priorität als der rechtsassoziative Operator $\rightarrow$.

Ein einfaches Übersetzungsschema zur Typ-Synthese

$$
\begin{array}{rl}
P \to D; E & \\
D \to D; D & \\
D \to \texttt{id} : T & \{\ \texttt{addtype}(\texttt{id}.\mathit{entry}, T.\mathit{type})\ \} \\
T \to \texttt{char} & \{\ T.\mathit{type} := \mathit{char}\ \} \\
T \to \texttt{integer} & \{\ T.\mathit{type} := \mathit{integer}\ \} \\
T \to \uparrow T_1 & \{\ T.\mathit{type} := \mathit{pointer}(T_1.\mathit{type})\ \} \\
T \to \texttt{array[num] of } T_1 & \{\ T.\mathit{type} := \mathit{array}(\texttt{num}.\mathit{val}, T_1.\mathit{type})\ \}
\end{array}
$$

$$
\begin{aligned}
E \to \texttt{literal} \quad & \{\ E.type := char\ \} \\
E \to \texttt{num} \quad & \{\ E.type := integer\ \} \\
E \to \texttt{id} \quad & \{\ E.type := \texttt{lookup}(\texttt{id}.entry)\ \} \\
E \to E_1\ \texttt{op}\ E_2 \quad & \{\ E.type := \texttt{if}\ \ E_1.type = integer \\
& \qquad \texttt{and}\ \ E_2.type = integer \\
& \quad \texttt{then}\ integer\ \texttt{else}\ type\text{-}error\ \} \\
E \to E_1[E_2] \quad & \{\ E.type := \texttt{if}\ \ E_2.type = integer \\
& \qquad \texttt{and}\ \ E_1.type = array(s,\ t) \\
& \quad \texttt{then}\ t\ \texttt{else}\ type\text{-}error\ \} \\
E \to E_1 \uparrow \quad & \{\ E.type := \texttt{if}\ \ E_1.type = pointer(t) \\
& \quad \texttt{then}\ t\ \texttt{else}\ type\text{-}error\ \}
\end{aligned}
$$

Erweiterung des SDTS auf Anweisungen

$$
\begin{array}{rl}
P \to D\,;\,S & \\
S \to \texttt{id} := E & \{\ S.type := \texttt{if}\ \ \texttt{lookup}(\texttt{id}.entry) = E.type \\
 & \quad \texttt{then}\ \ void\ \ \texttt{else}\ \ type\text{-}error\ \} \\
S \to \texttt{if}\ \ E\ \ \texttt{then}\ \ S_1 & \{\ S.type := \texttt{if}\ \ E.type = boolean \\
 & \quad \texttt{then}\ \ S_1.type\ \ \texttt{else}\ \ type\text{-}error\ \} \\
S \to \texttt{while}\ \ E\ \ \texttt{do}\ \ S_1 & \{\ S.type := \texttt{if}\ \ E.type = boolean \\
 & \quad \texttt{then}\ \ S_1.type\ \ \texttt{else}\ \ type\text{-}error\ \} \\
S \to \texttt{begin}\ \ L\ \ \texttt{end} & \{\ S.type := L.type\ \} \\
L \to S; L_1 & \{\ L.type := \texttt{if}\ \ S.type = void \\
 & \quad \texttt{then}\ \ L_1.type\ \ \texttt{else}\ \ type\text{-}error\ \} \\
L \to \varepsilon & \{\ L.type := void\ \}
\end{array}
$$

Kodierung der Typ-Ausdrücke in Java

Kodierung der Basistypen:

Typ	**Kodierung**	**Bemerkung**
byte	`B`	signed byte
char	`C`	Unicode Zeichen
double	`D`	Floating-Point-double
float	`F`	Floating-Point single
int	`I`	integer
long	`J`	long integer
reference	`L<classname>`	Objekt der Klasse <classname>
short	`S`	short
boolean	`Z`	boolean
reference	`[`	array

Der `<classname>` repräsentiert dabei einen vollen qualifizierten Klassen- oder Interfacenamen.

Kodierung der Typ-Ausdrücke von Methoden

Die Kodierung von Typ-Ausdrücken von Methoden haben die Form:

(<Typ-Ausdrücke der Parameter>) <Typ des Rückgabewerts>

Der Rückgabetyp `void` wird dabei mit `V` kodiert.

Beispiel

Der Typ-Ausdruck für eine Instanzenvariable vom Typ `int` ist `I`.

Der Typ-Ausdruck für eine Instanzenvariable vom Typ `Object` ist `Ljava/lang/Object;` .

Ein 2-dimensionales Feld `float d[][]` hat den zugehörigen Typ-Ausdruck `[[F`.

Einer Deklaration

```
Object myMethod (int i, double d, Thread t)
```

würde also die Kodierung

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

zugeordnet.

Äquivalenz von Typ-Ausdrücken

Bei der Typ-Prüfung muss immer wieder getestet werden, ob zum Beispiel der Typ-Ausdruck *typ1* eines Operanden mit dem Typ-Ausdruck *typ2* eines Operanden einer Operation zusammenpasst.
Man sagt in diesem Fall, dass die Typen *typ1* und *typ2* **äquivalent** sind.
Solange keine Namen als Abkürzungen für Typ-Ausdrücke auftreten, ist die Sache relativ einfach:

Zwei Typ-Ausdrücke sind äquivalent, wenn sie identisch sind.

Der Typ-Prüfer muss also nur beide Typ-Ausdrücke bzgl. ihres Aufbaus miteinander vergleichen. Dies geschieht am besten rekursiv.

Es gibt leider viele Konstruktionen in höheren Programmiersprachen, die implementationsabhängig sind.

In Pascal betrachte man etwa

```
var   x: integer;
      y: 1..20;
      z: 10..50;
      a: array[1..10] of integer;
      b: array[0..9] of integer;
```

Läßt man zu, dass Namen für Typ-Ausdrücke benutzt werden können, dann gibt es zusätzliche Probleme.
Man betrachte etwa das folgende Pascal-Programmfragment:

```
type link = ↑cell;
      slk = link;
var  next : link;
     last : link;
        s : slk;
     p    : ↑cell;
     q,r  : ↑cell;
```

Interpretation von Namen in Typ-Ausdrücken

Namensäquivalenz:
Jeder Typ-Name legt einen neuen Typ fest.

Strukturäquivalenz:
Jeder Typ-Name ist nur eine Abkürzung für den definierten Typ-Ausdruck.

Weitere Probleme der Typ-Äquivalenz

```
t1 = array[-1..9] of integer;
t2 = array[0..10] of integer;
rec1 = record
    x: boolean;
    y: integer;
    end;
rec2 = record
    a: boolean;
    b: integer;
    end;
```

Ein Probleme der Struktur-Äquivalenz

```
type p=record
        info:integer;
        next:↑p;
     end;
     q=record
        x:integer;
        z:↑r;
     end;
     r=record
        x:integer;
        z:↑q;
     end;
```

Typ-Umwandlungen

Speziell für die Basistypen gibt es in vielen Programmiersprachen eine Reihe von Umwandlungsregeln, um eine automatische Anpassung der Typen von Operanden an zulässigen Typen eines Operators zu ermöglichen.

Beispiel

Man betrachte folgendes Programmfragment:

`j := x + i;` mit `x` *real* und `i`, `j` *integer*

Da es i.A. keinen Operator für eine derartige Additionsoperation gibt, muss der Compiler oder aber auch der Programmierer etwas tun:

- Der Compiler führt eine automatische Typ-Anpassung durch Einfügen einer Typ-Umwandlung durch. Dies ist üblicherweise nur dann erlaubt, wenn damit keine Genauigkeitsverluste verbunden sind.
- Der Compiler signalisiert eine Fehlersituation. Dann
  - kann der Programmierer eine explizite Typ-Umwandlung durch Aufruf einer Bibliotheksfunktion programmieren, etwa `x + float(i)` oder
  - der Programmierer kann eine explizite Typ-Umwandlung durch sogenanntes „casting“ vornehmen. Dies bewirkt eine Änderung des zugehörigen Typ-Ausdrucks, aber nicht notwendig eine Änderung der internen Darstellung des Wertes, etwa in `(char)20`.

Operator Identifikation (Überladene Funktionen)

Verschiedene Operatoren können eine identische lexikale Darstellung, z.B. kann + eine Integer-Addition, eine Float-Addition oder eine String-Konkatenation repräsentieren.
Man bezeichnet + daher als **überladenen Operator**.
Aufgabe des Compiler ist es, aus dem Kontext den „richtigen“ Operator zu identifizieren.

Das Problem verschärft sich bei Programmiersprachen, die dem Programmierer mehrere Definitionen einer Funktion mit unterschiedlichen Parameter- oder Ergebnistypen erlaubt.

In Ada sind z.B folgende Definitionen gleichzeitig erlaubt:

```
function '*' (i, j: integer) return complex;
function '*' (x, y: complex) return complex;
```

Damit hat eine Funktion (genauer ein Funktionsname) eine Vielzahl von Typ-Ausdrücken und auch arithmetische Ausdrücke haben nicht notwendigerweise nur einen Typ!

Beispiel

Nehmen wir an, dass $*$ unter anderen die folgenden Typ-Ausdrücke hat:

- $\mathit{integer} \times \mathit{integer} \to \mathit{integer}$
- $\mathit{integer} \times \mathit{integer} \to \mathit{complex}$
- $\mathit{complex} \times \mathit{complex} \to \mathit{complex}$

Haben dann die Literale 2, 3 und 5 den einzig möglichen Typ `integer` und ist `z` eine Variable vom Typ `complex`, dann kann $3 * 5$ den Typ `integer` oder `complex` haben.
Im Ausdruck $(3 * 5) * 2$ **muss** $3 * 5$ den Typ `integer` haben, im Ausdruck $(3 * 5) * z$ **muss** $3 * 5$ den Typ `complex` haben.

Ein SDTS für die Operatoridentifikation

Beispielhaft soll die Identifikation eines Operators oder einer einer Funktion an dem Beispiel einer Übersetzung in Postfix-Notation erläutert werden.
Es werden die folgenden Schritte durchgeführt:

1. Bottom-Up werden die möglichen Typ-Mengen eines Teilausdrucks berechnen (synthetisches Attribut *types*)
2. Top-Down werden die identifizierten Typen für jeden Teilausdruck festgelegt inherites Attribut *utype*)
3. Bottom-Up wird der Postfix-Code erzeugt (mit dem synthetischen Attribut *code*)

$E' \to E$ $\quad$ { $E'.types := E.types$;

$E.utype :=$ `if` $E.types = \{t\}$ `then` t `else` *type-error*;

$E'.code := E.code$; }

$E \to$ `id` $\quad$ { $E.types :=$ `lookup( id.entry)`;

$E.code :=$ `concat(id.lexstring, ":",` $E.utype$ `);` }

$E \to E_1$ op E_2 $\quad$ { $E.types := \{t \mid$ es gibt Typ $r \in E_1.types$

und Typ $s \in E_2.types$ und Typ $r \times s \to t \in$ op$.types$;

Sei $\sigma = \{(r,s) \mid r \times s \to t \in$ op$.types$,

$r \in E_1.types, s \in E_2.types$ und $t = E.utype\}$;

$E_1.utype :=$ `if` $\sigma = \{(r,s)\}$ `then` r `else` *type-error*;

$E_2.utype :=$ `if` $\sigma = \{(r,s)\}$ `then` s `else` *type-error*;

$E.code :=$ `concat(`$E_1.code, E_2.code,$ op $: r \times s \to t$,

r, s, t wie oben ; }

Probleme bei der Methodenausahl

Viel komplexer ist das Problem bei objekt-orientierten Sprachen mit Klassenhierarchie und statischer Typprüfung. Hier geht es um die Auswahl der korrekten Methode, die zum Teil auch erst zur Laufzeit getroffen werden kann.
Da Java das dynamische Laden von Klassen erlaubt und viele größere Programme auch noch mit *reflection* arbeiten, ist eine genaue Auswahl der „passenden“ Methode vom Compiler nicht immer zu erreichen und man muss die Auswahl auf die Laufzeit des Programms verschieben (dynamische Methodenauswahl).

Beispiel

```
class T {
     T n() {return new R(); }
}
class S extends T {
     T n() {return new S();}
}
 class R extends S {
     T n() {return new R();}
}
main () {
    T a;
    if (....)
        a = new T();
    else
        a = new S();
     a = a.n();
}
```

Untertypen

Untertypen erzeugen eine Relation auf den Typen, die es erlaubt, Werte eines Typs anstelle von Werten eines anderen Typs zu benutzen.
Ist X Untertyp von Y, geschrieben $X <: Y$, dann kann jeder Ausdruck vom Typ X ohne Hervorrufen eines Typ-Fehlers in jedem Kontext benutzt werden, in dem ein Ausdruck vom Typ Y benötigt wird.

Beispiel

In C gilt `char` $<:$ `integer`.

Beispiel

Typ-Prüfung bei der Anwendung einer Funktion `f` auf ein Argument `x`:

- ohne Untertypen-
  der Typ-Prüfer bestimmt den Typ $A \to B$ der Funktion `f` und den Typ C des Arguments `x` und prüft, ob $C = A$ gilt.
- mit Untertypen-
  der Typ-Prüfer bestimmt den Typ $A \to B$ der Funktion `f` und den Typ C für `x` und prüft, ob $C <: A$ gilt.

Untertypen und Vererbung in OO-Sprachen

- Untertypen bilden eine Relation auf den Typen
- Vererbung bildet eine Relation auf den Implementationen

Meist werden diese beiden Konzepte in dem Klassenkonzept einer OO-Sprache miteinander vermengt und in vielen Situationen stimmen die Relationen überein. Man beachte aber:

Beispiel

Die Klasse Dequeue implementiere eine Dequeue. Durch Überschreiben von Methoden kann man als Unterklassen eine Klasse Stack und eine Klasse Queue definieren.
Allerdings bilden die durch die Unterklassen definierten Typen stack und queue keinen Untertyp von dequeue!

Polymorphe Funktionen

Weitere Probleme treten auf, wenn in der Programmiersprache sogenannte „polymorphe Funktionen“ oder „polymorphe Operatoren“ zugelassen sind. Dies sind Funktionen, bei denen der Typ der Parameter nicht eindeutig festgelegt werden muss.

Beispiel

Funktion zum Bestimmen der Länge einer Liste (in Scheme):

```
(define (length liste)
        (if (null? liste)
            0
            (+ 1 (length (cdr liste)))))
```

Typ-Variable

Um auch Typ-Ausdrücke für polymorphe Funktionen angeben zu können, muss man das Konzept einer **Typ-Variablen** einführen. Eine Typ-Variable steht dabei für einen beliebigen Typ.

Für das vorige Beispiel könnte man der Funktion `length` den Typ-Ausdruck $\forall\alpha$ *list*(α) $\rightarrow$ *integer* zuordnen, wobei α die Typ-Variable ist.

Bemerkung

Der Quantor wird im folgenden weggelassen, sofern keine Missverständnisse zu befürchten sind.

Die Typ-Prüfung von Programmen, in denen polymorhe Funktionen auftreten können, ist signifikant schwieriger als die Typ-Prüfung in den vorangegangenen Fällen.

- Verschiedene Auftreten einer polymorphen Funktion in einem Ausdruck können Argumente unterschiedlichen Typs haben.
- Da in Typ-Ausdrücken Variablen auftreten können, ist die Typ-Äquivalenz neu zu definieren. Um zwei Typ-Ausdrücke mit Variablen „anzupassen“, müssen die auftretenden Variablen durch Typ-Ausdrücke (eventuell wieder mit neuen Variablen) ersetzt werden.Gelingt es, beide Ausdrücke auf diese Weise identisch zu machen (zu unifizieren), sind die beiden Typen äquivalent.
- Das Problem ist unter dem Namen Unifikation bekannt.

Andere Möglichkeiten der Typ-Prüfung in Programmiersprachen, die polymorphe Funktionen erlauben:

1. Man verzichtet völlig auf die Typisierung von Variablen usw. in der Programmiersprache und auf eine Überprüfung zur Übersetzungszeit und benutzt eine dynamische Typ-Überprüfung zur Laufzeit (etwa in LISP, Scheme oder Smalltalk)
2. Man verzichtet auf die Typisierung von Variablen usw. in der Programmiersprache, prüft aber in der Übersetzungsphase jeden Gebrauch eines Namens oder eines anderen Sprachkonstrukts auf einen konsistenten Gebrauch (etwa in Swift, ML oder Haskell)
3. Man erweitert die Syntax der Programmiersprache um Typ-Variable oder Typ-Parameter. Oder man „durchlöchert“ das übliche Typ-System durch die Verwendung von Zeigern oder durch beliebiges „casting“.