Script generated by TTT

Title: Simon: Compilerbau (13.06.2012)

Date: Wed Jun 13 14:05:05 CEST 2012

Duration: 74:01 min

Pages: 27

Kapitel 3:

Typ-Überprüfung

Ziel der Typ-Überprüfung

In modernen (imperativen / objektorientierten / funktionalen) Programmiersprachen besitzen Variablen und Funktionen einen Typ, z.B. `int`, `void*`, `struct { int x; int y; }`.

Typen sind nützlich für:

- die Speicherverwaltung;
- die Vermeidung von Laufzeit-Fehlern

In imperativen / objektorientierten Programmiersprachen muss der Typ bei der Deklaration spezifiziert und vom Compiler die typ-korrekte Verwendung überprüft werden.

Typ-Ausdrücke

Typen werden durch Typ-*Ausdrücke* beschrieben.
Die Menge T der Typausdrücke enthält:

1. Basis-Typen: `int`, `char`, `float`, `void`, ...
2. Typkonstruktoren, die auf Typen angewendet werden

Beispiele für Typkonstruktoren:

- Verbunde: `struct` $\{\ t_1\ a_1; \ldots t_k\ a_k;\ \}$
- Zeiger: $t\ *$
- Felder: $t\ [\,]$
  - in C kann/muss zusätzlich eine Größe spezifiziert werden
  - die Variable muss zwischen t und $[n]$ stehen
- Funktionen: $t(t_1, \ldots, t_k)$
  - in C muss die Variable zwischen t und $(t_1, \ldots, t_k)$ stehen.
  - in ML würde man diesen Typ anders herum schreiben: $t_1 * \ldots * t_k \to t$
- wir benutzen $(t_1, \ldots, t_k)$ als Tupel-Typen

Typ-Namen

Ein Typ-Name ist ein *Synonym* für einen Typ-Ausdruck.
In C kann man diese mit Hilfe von **`typedef`** einführen.
Typ-Namen sind nützlich

- als Abkürzung:

```
typedef struct { int x; int y; } point_t;
```

- zur Konstruktion *rekursiver* Typen:

Erlaubt in C:

```
struct list {
  int info;
  struct list* next;
}

struct list* head;
```

Lesbarer:

```
typedef struct list list_t;
struct list {
  int info;
  list_t* next;
}
list_t* head;
```


Typ-Prüfung

Aufgabe:

Gegeben: eine Menge von Typ-Deklarationen $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

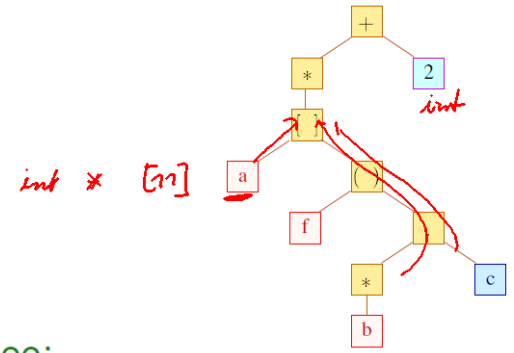
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

Betrachte den Ausdruck:

```
*a[f(b->c)]+2;
```


Typ-Prüfung am Syntax-Baum

Prüfe Ausdruck `*a[f(b->c)]+2`:



Idee:

- traversiere den Syntaxbaum bottom-up
- für Bezeichner schlagen wir in Γ den richtigen Typ nach
- Konstanten wie 2 oder 0.5 sehen wir den Typ direkt an
- die Typen für die inneren Knoten erschießen wir mit Hilfe von *Typ-Regeln*

Typ-Systeme

Formal betrachten wir *Aussagen* der Form:

$$\Gamma \vdash e : t$$

// (In der Typ-Umgebung Γ hat e den Typ t)

Axiome:

Const: $\dfrac{}{\Gamma \vdash c : t_c}$ (t_c Typ der Konstante c)

Var: $\dfrac{}{\Gamma \vdash x : \Gamma(x)}$ (x Variable)

Regeln:

Ref: $\dfrac{\Gamma \vdash e : t}{\Gamma \vdash \&e : t*}$

Deref: $\dfrac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}$

Typ-System für C-ähnliche Sprachen

Weitere Regeln für diverse Typausdrücke:

Array:
$$\frac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Array:
$$\frac{\Gamma \vdash e_1 : t[\,] \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Struct:
$$\frac{\Gamma \vdash e : \textbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

App:
$$\frac{\Gamma \vdash e : t(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$$

Op:
$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

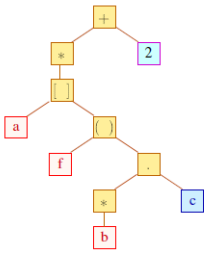
Cast:
$$\frac{\Gamma \vdash e : t_1 \qquad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2)\ e : t_2}$$

Beispiel: Typ-Prüfung

Ausdruck `*a[f(b->c)]+2` und $\Gamma = \{$

```
struct list { int info; struct list* next; };
int f(struct list* l);
struct { struct list* c;}* b;
int* a[11];
```

$\}$:



Typ-System für C-ähnliche Sprachen

Weitere Regeln für diverse Typausdrücke:

Array:
$$\frac{\Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Array:
$$\frac{\Gamma \vdash e_1 : t[\,] \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

Struct:
$$\frac{\Gamma \vdash e : \textbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i : t_i}$$

App:
$$\frac{\Gamma \vdash e : t(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}$$

Op:
$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

Cast:
$$\frac{\Gamma \vdash e : t_1 \qquad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2)\ e : t_2}$$

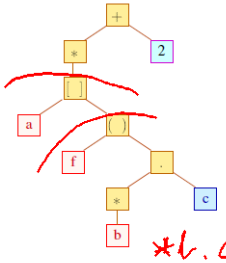
62 / 184

Beispiel: Typ-Prüfung

Ausdruck `*a[f(b->c)]+2` und $\Gamma = \{$

```
struct list { int info; struct list* next; };
int f(struct list* l);
struct { struct list* c;}* b;
int* a[11];
```

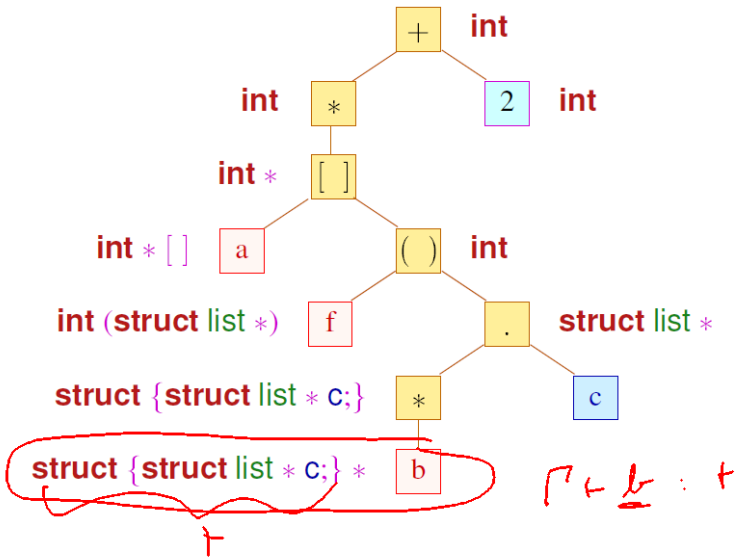
$\}$:



63 / 184

Beispiel: Typ-Prüfung

Ausdruck `*a[f(b->c)]+2`:



Gleichheit von Typen

Zusammenfassung Typprüfung:

- Welche Regel an einem Knoten angewendet werden muss, ergibt sich aus den Typen für die bereits bearbeiteten Kinderknoten
- Dazu muss die *Gleichheit* von Typen festgestellt werden.

Typgleichheit in C:

- **`struct`** `A {}` und **`struct`** `B {}` werden als verschieden betrachtet
  - ⇝ der Compiler kann die Felder von `A` und `B` nach Bedarf umordnen (in *C nicht* erlaubt)
  - um einen Verband `A` um weitere Felder zu erweitern, muss er eingebettet werden:

```
typedef struct B {
          struct A a;
          int field_of_B;
        } extension_of_A;
```

- Nach **`typedef int C;`** haben `C` und **`int`** den gleichen Typ.

Strukturelle Typ-Gleichheit

Alternative Interpretation von Gleichheit (*gilt nicht in C*):

Semantisch können wir zwei rekursive Typen t_1, t_2 als *gleich* betrachten, falls sie die gleiche Menge von Pfaden zulassen.

Beispiel:

```
struct list {
  int info;
  struct list* next;
}
```

```
struct list1 {
  int info;
  struct {
    int info;
    struct list1* next;
  }* next;
}
```

Sei **`struct`** `list* l` oder **`struct`** `list1* l`. Beide erlauben

```
l->info   l->next->info
```

jedoch hat `l` jeweils einen anderen Typen in C.

Algorithmus zum Test auf Semantische Gleichheit

Idee:

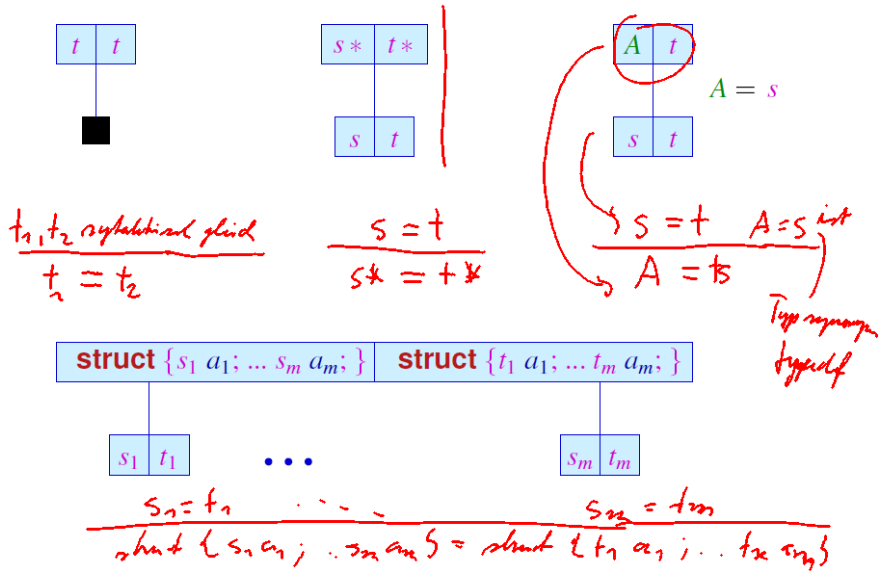
- Verwalte Äquivalenz-Anfragen für je zwei Typausdrücke
- Sind die beiden Ausdrücke syntaktisch gleich, ist alles gut
- Andernfalls reduziere die Äquivalenz-Anfrage zwischen Äquivalenz-Anfragen zwischen (hoffentlich) einfacheren anderen Typausdrücken

Nehmen wir an, rekursive Typen würden mit Hilfe von Typ-Gleichungen der Form:

$$A = t$$

eingeführt (verzichte auf ein Γ). Dann definieren wir folgende Regeln:

Regeln für Wohlgetyptheit



Beispiel:

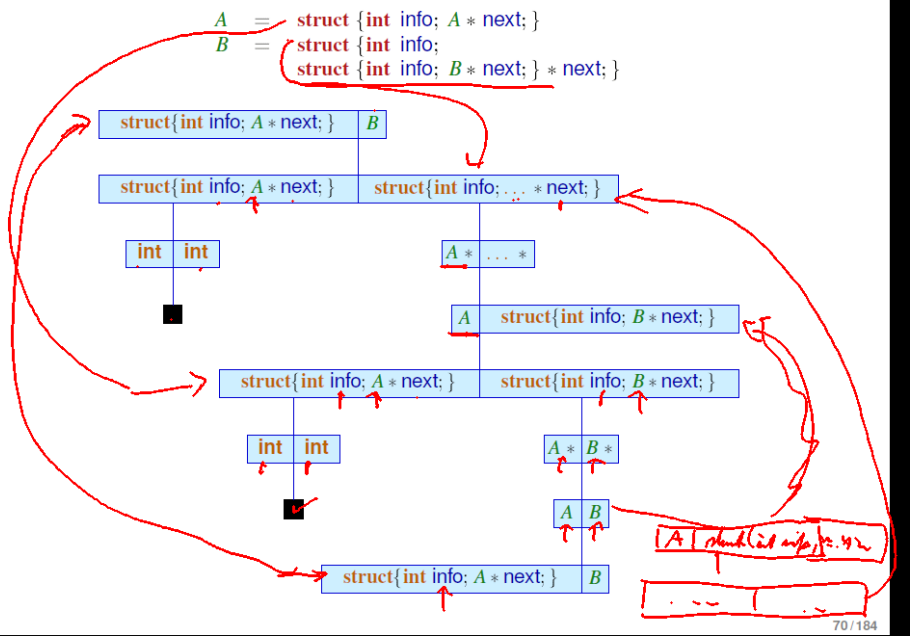
$$
\begin{array}{lcl}
A & = & \textbf{struct}\ \{\textbf{int}\ \text{info};\ A * \text{next};\ \} \\
B & = & \textbf{struct}\ \{\textbf{int}\ \text{info};\ \\
 & & \textbf{struct}\ \{\textbf{int}\ \text{info};\ B * \text{next};\ \} * \text{next};\ \}
\end{array}
$$

Wir fragen uns etwa, ob gilt:

$$\textbf{struct}\ \{\textbf{int}\ \text{info};\ A * \text{next};\ \} = B$$

Dazu konstruieren wir:

Beweis Beispiel:



Implementierung

- Stoßen wir bei der Konstruktion des Beweisbaums auf eine Äquivalenz-Anfrage, auf die keine Regel anwendbar ist, sind die Typen *ungleich*
- Die Konstruktion des Beweisbaums kann dazu führen, dass die gleiche Äquivalenz-Anfrage ein weiteres Mal auftritt
- Taucht eine Äquivalenz-Anfrage ein weiteres Mal auf, sind die Typen der Anfrage per Definition gleich

Terminierung?

Implementierung

- Stoßen wir bei der Konstruktion des Beweisbaums auf eine Äquivalenz-Anfrage, auf die keine Regel anwendbar ist, sind die Typen *ungleich*
- Die Konstruktion des Beweisbaums kann dazu führen, dass die gleiche Äquivalenz-Anfrage ein weiteres Mal auftritt
- Taucht eine Äquivalenz-Anfrage ein weiteres Mal auf, sind die Typen der Anfrage per Definition gleich

Terminierung?

- die Menge D aller deklarierten Typen ist endlich
- es gibt höchstens $|D|^2$ viele Äquivalenzanfragen
- wiederholte Anfragen sind automatisch erfüllt

$\rightsquigarrow$ Terminierung ist gesichert

Überladung und Koersion

Manche Operatoren wie z.B. $+$ sind *überladen*:

- $+$ besitzt *mehrere mögliche* Typen
  Zum Beispiel: `int +(int,int)`, `float +(float, float)`
  aber auch `float* +(float*, int)`, `int* +(int, int*)`
- je nach Typ hat der Operator $+$ ein unterschiedliche Implementation
- welche Implementierung ausgewählt wird, entscheiden die Argument-Typen

Überladung und Koersion

Manche Operatoren wie z.B. $+$ sind *überladen*:

- $+$ besitzt *mehrere mögliche* Typen
  Zum Beispiel: `int +(int,int)`, `float +(float, float)`
  aber auch `float* +(float*, int)`, `int* +(int, int*)`
- je nach Typ hat der Operator $+$ ein unterschiedliche Implementation
- welche Implementierung ausgewählt wird, entscheiden die Argument-Typen

Koersion: Erlaube auch die Anwendung von $+$ auf `int` und `float`.

- anstatt $+$ für alle Argument-Kombinationen zu definieren, werden die Typen der Argumente konvertiert
- Konvertierung kann Code erzeugen (z.B. Umwandlung von `int` nach `float`)
- Konvertiert wird in der Regel auf die Supertypen, d.h. `5+0.5` hat Typ `float` (da `float` $\geq$ `int`)

$$\frac{\Gamma : e : \texttt{int}}{\Gamma : e : \texttt{float}}$$

Koersion von Integer-Typen in C: Promotion

C enthält spezielle Koersionsregeln für Integer: *Promotion*

$$\begin{matrix}\texttt{unsigned char}\\ \texttt{signed char}\end{matrix} \leq \begin{matrix}\texttt{unsigned short}\\ \texttt{signed short}\end{matrix} \leq \texttt{int} \leq \texttt{unsigned int}$$

. . . wobei eine Konvertierung über alle Zwischentypen gehen muss.

Koersion von Integer-Typen in C: Promotion

C enthält spezielle Koersionsregeln für Integer: *Promotion*

`unsigned char` `signed char` $\leq$ `unsigned short` `signed short` $\leq$ `int` $\leq$ `unsigned int`

... wobei eine Konvertierung über alle Zwischentypen gehen muss.

Subtile Fehler möglich! Berechne Zeichenverteilung in `char* str`:

```
char* str = "...";
int dist[256];
memset(dist, 0, sizeof(dist));
while (*str) {
  dist[(unsigned) *str]++;
  str++;
};
```

Beachte: `unsigned` bedeutet `unsigned int`.

Ausblick

Registerverteilung hat weitere Aufgaben:

- unnötige move Instruktionen müssen vermieden werden
- Variablen müssen auf den Stack ausgelagert werden
  - $\rightsquigarrow$ evtl. benötigt dies wiederum Register
- übersetze Funktionen in eine single static assignment Form
- optimale Färbung möglich (allerdings müssen evtl. Register getauscht werden)

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

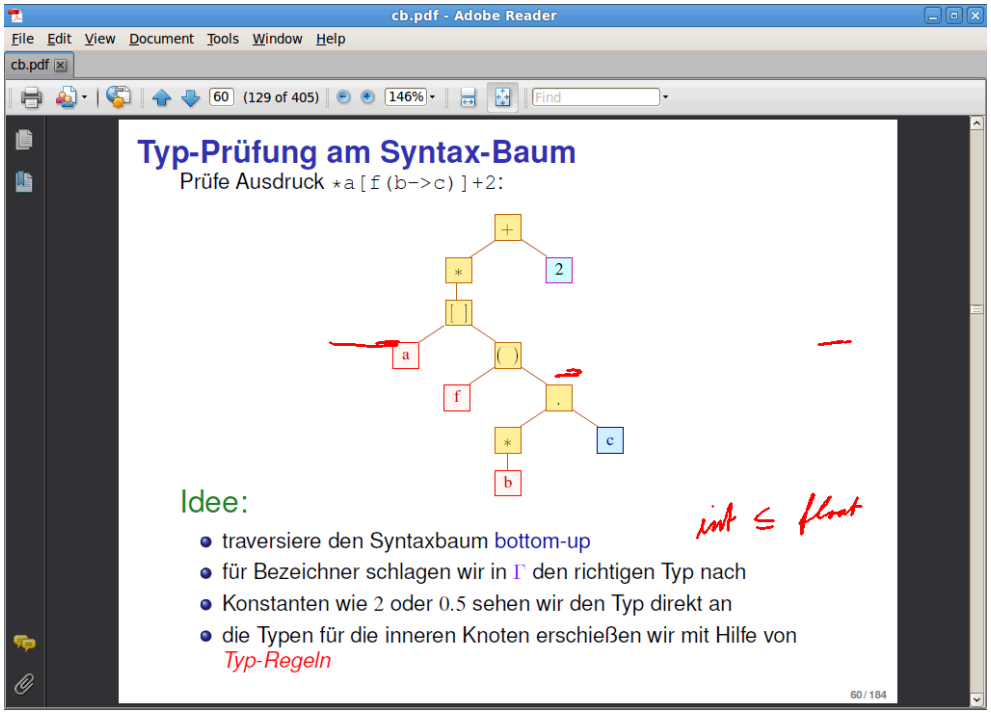
Schematisch präsentierte liveness-Analyse verbesserungsfähig:

- nach $x \leftarrow y + 1$ ist x nur lebendig wenn y lebendig ist
- saveloc hält Register unnötig am Leben $\rightsquigarrow$ Zwischensprache
- gibt es *optimale* Regeln für die liveness-Analyse?

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

Wie berechnet man die liveness Mengen, Registerverteilung zügig?

$\rightsquigarrow$ Vorlesung *Programmoptimierung*



Beispiel: Teiltypen

Betrachte:

```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

- Wir möchten dass `extractInfo` für alle Argument-Strukturen funktioniert, die eine Komponente `string info` besitzen
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner)