**Script** **generated by TTT**

Title: Petter: Compilerbau (28.06.2018)

Date: Thu Jun 28 14:16:03 CEST 2018

Duration: 83:16 min

Pages: 25



# Chapter 2:
# Decl-Use Analysis



## Symbol Tables

Consider the following Java code:

```
void foo() {
  int A;
  while(true) {
    double A;
    A = 0.5;
    write(A);
    break;
  }
  A = 2;
  bar();
  write(A);
}
```

- within the body of the loop, the definition of A is shadowed by the *local definition*
- each *declaration* of a variable v requires allocating memory for v
- accessing v requires finding the declaration the access is *bound* to
- a binding is not *visible* when a local declaration of the same name is in scope



## Scope of Identifiers

```
void foo() {
  int A;
  while (true) {
    double A;
    A = 0.5;
    write(A);
    break;
  }
  A = 2;
  bar();
  write(A);
}
```

scope of **int** A

# Rapid Access: Replace Strings with Integers

## Idea for Algorithm:

Input: a sequence of strings
Output:
1. sequence of numbers
2. table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier during *scanning*.

## Implementation approach:

- count the number of new-found identifiers in **int** count
- maintain a *hashtable* $S$ : **String** $\to$ **int** to remember numbers for known identifiers

We thus define the function:

```
int indexForIdentifier(String w) {
    if (S (w) ≡ undefined) {
        S = S ⊕ {w ↦ count};
        return count++;
    } else return S (w);
}
```



# Implementation: Hashtables for Strings

1. allocate an array $M$ of sufficient size $m$
2. choose a *hash function* $H$ : **String** $\to [0, m-1]$ with:
   - $H(w)$ is cheap to compute
   - $H$ distributes the occurring words equally over $[0, m-1]$

Possible generic choices for sequence types ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

$$H_0(\vec{x}) = (x_0 + x_{r-1}) \,\%\, m$$
$$H_1(\vec{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m$$
$$= (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m$$

for some prime number $p$ (e.g. $31$)

✗ The hash value of $w$ *may not be unique*!
- → Append $(w, i)$ to a linked list located at $M[H(w)]$
- Finding the index for $w$, we compare $w$ with all $x$ for which $H(w) = H(x)$

✓ access on average:
insert: $\mathcal{O}(1)$
lookup: $\mathcal{O}(1)$



# Example: Replacing Strings with Integers

Input:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Peter | Piper | picked | a | peck | of | pickled | peppers |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| If | Peter | Piper | picked | a | peck | of | pickled | peppers |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| wheres | the | peck | of | pickled | peppers | Peter | Piper | picked |

Output:



# Refer Uses to Declarations: Symbol Tables

Check for the correct usage of variables:

- Traverse the syntax tree in a suitable sequence, such that
  - each declaration is visited before its use
  - the currently visible declaration is the last one visited

  ⇝ perfect for an L-attributed grammar
  - equation system for basic block must add and remove identifiers
- for each identifier, we manage a *stack* of declarations
  1. if we visit a *declaration*, we push it onto the stack of its identifier
  2. upon leaving the *scope*, we remove it from the stack
- if we visit a *usage* of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an undeclared identifier

# Example: A Table of Stacks

```
// Abstract locations in comments
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;     // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

| | |
|---|---|
| 0 | $a$ |
| 1 | $b$ |
| 2 | $c$ |

| | |
|---|---|
| 0 | $a$ |
| 1 | $b$ |
| 2 | $c$ |

| | |
|---|---|
| 0 | $a$ |
| 1 | $b$ |
| 2 | $c$ |

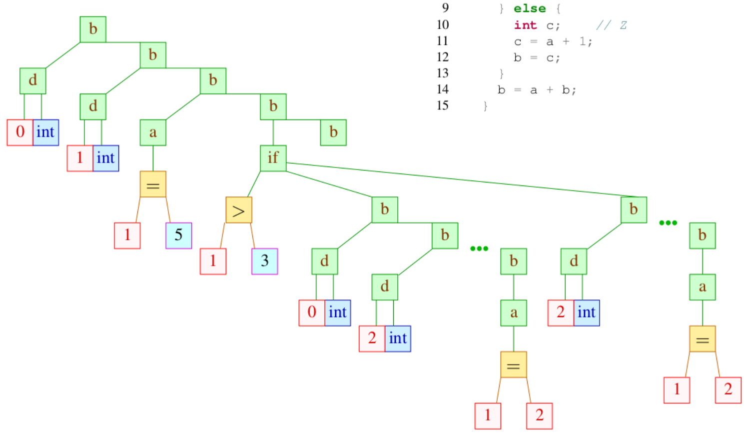
| | |
|---|---|
| 0 | $a$ |
| 1 | $b$ |
| 2 | $c$ |



# Decl-Use Analysis: Annotating the Syntax Tree

- d declaration node
- b basic block
- a assignment

```
{
  int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;     // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```






# Alternative Implementations for Symbol Tables

- when using a list to store the symbol table, storing a marker indicating the old head of the list is sufficient

| |
|---|
| $a$ |
| $b$ |

in front of if-statement



# Type Definitions in C

A type definition is a *synonym* for a type expression.
In C they are introduced using the **`typedef`** keyword.
Type definitions are useful

- as abbreviation:

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

- to construct *recursive* types:

Possible declaration in C:

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

more readable:

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

## Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`.
Consider the following declarations:

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

Relevant C grammar:

| | | |
|---|---|---|
| declaration | → | (declaration-specifier)$^+$ declarator ; |
| declaration-specifier | → | `static` \| `volatile` ... `typedef` |
| | | \| `void` \| `char` \| `char` ... `typename` |
| declarator | → | `identifier` \| ... |



## Type Definitions in C: Solutions

Relevant C grammar:

| | | |
|---|---|---|
| declaration | → | (declaration-specifier)$^+$ declarator ; |
| declaration-specifier | → | `static` \| `volatile` ··· `typedef` |
| | | \| `void` \| `char` \| `char` ··· `typename` |
| declarator | → | `identifier` \| ··· |

Solution is difficult:





# Chapter 3:

# Type Checking



## Type Expressions

Types are given using type-*expressions*.
The set of type expressions $T$ contains:

1. base types: **`int`**, **`char`**, **`float`**, **`void`**, ...
2. type constructors that can be applied to other types

# Type Checking

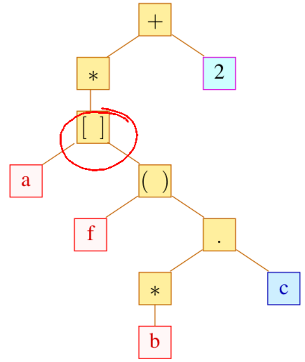
## Problem:

**Given:** A set of type declarations $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$
**Check:** Can an expression $e$ be given the type $t$?



# Type Checking using the Syntax Tree

Check the expression `*a[f(b->c)]+2`:




## Idea:

- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in $\Gamma$
- constants such as $2$ or $0.5$ have a fixed type
- the types of the inner nodes of the tree are deduced using *typing rules*



# Type Systems

**Formally:** consider *judgements* of the form:

$$\Gamma \vdash e : t$$

// (in the type environment $\Gamma$ the expression $e$ has type $t$)

## Axioms:

Const: $\Gamma \vdash c : t_c$ ($t_c$ type of constant $c$)
Var: $\Gamma \vdash x : \Gamma(x)$ ($x$ Variable)

## Rules:

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



# Type Systems for C-like Languages

More rules for typing an expression:

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

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

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

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

Op $\Box$: $\dfrac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \Box e_2 : t}$

Explicit Cast: $\dfrac{\Gamma \vdash e : t_1 \qquad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_2)\ e : t_2}$

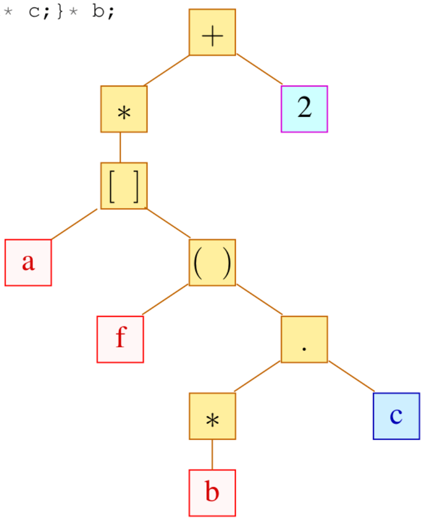
## Example: Type Checking

Given expression `*a[f(b->c)]+2` and

$\Gamma = \{$

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

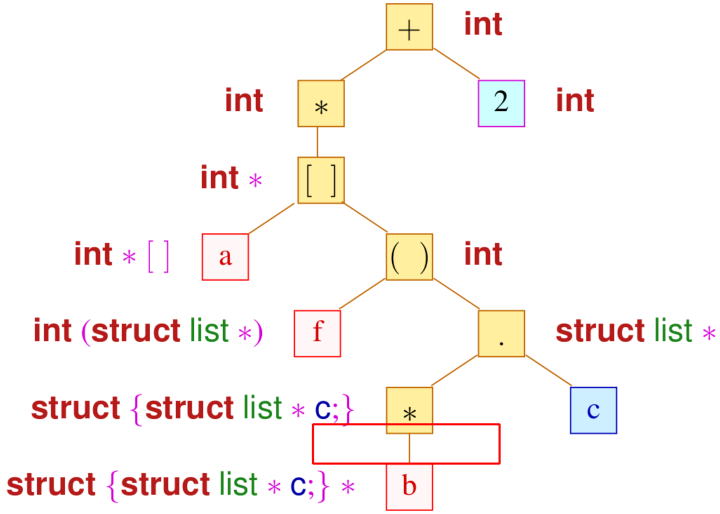
$\}$






## Example: Type Checking

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






## Example: Type Checking – More formally:

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

$$\text{Op}\;\frac{\text{Deref}\;\dfrac{\Gamma \vdash a[f(b \to c)] :}{\Gamma \vdash *a[f(b \to c)] : t} \qquad \text{Const}\;\dfrac{}{\Gamma \vdash 2 : t}}{\Gamma \vdash *a[f(b \to c)] + 2 : t}$$



## Equality of Types

### Summary of Type Checking

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- determining the rule requires a check for ⇝ *equality* of types

*type equality* in C:

- **`struct`** `A {}` and **`struct`** `B {}` are considered to be different
  - ⇝ the compiler could re-order the fields of `A` and `B` independently (*not* allowed in C)
  - to extend an record `A` with more fields, it has to be embedded into another record:

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

- after issuing **`typedef int`** `C;` the types `C` and **`int`** are the same

## Structural Type Equality

Alternative interpretation of type equality (*does not hold in C*):

*semantically*, two types $t_1, t_2$ can be considered as *equal* if they accept the same set of access paths.

Example:

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

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

Consider declarations `struct list* l` and `struct list1* l`. Both allow

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

but the two declarations of `l` have unequal types in C.



## Algorithm for Testing Structural Equality

### Idea:

- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

typedef $A$ $t$

(we omit the $\Gamma$). Then define the following rules: