Script **generated by TTT**

Title: Petter: Compilerbau (20.07.2017)

Date: Thu Jul 20 14:15:11 CEST 2017

Duration: 76:58 min

Pages: 26

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

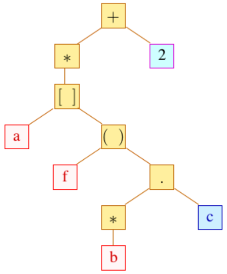
Type Checking

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 Γ
- 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 Γ 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: $\dfrac{\Gamma \vdash \boxed{e \; : \; t*}}{\Gamma \vdash \boxed{*e} : \boxed{t}}$

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

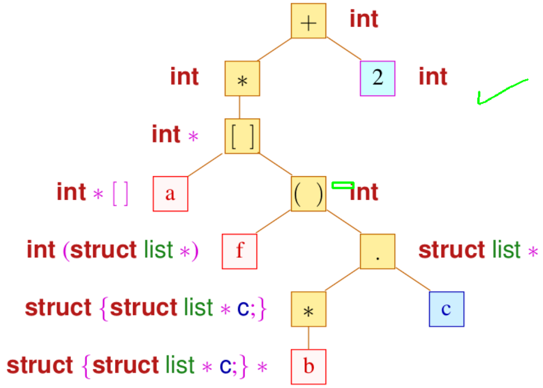
App: $\dfrac{\Gamma \vdash e \; : \; t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 \; : \; t_1 \quad \ldots \quad \Gamma \vdash e_m \; : \; t_m}{\Gamma \vdash e(e_1, \ldots, e_m) \; : \; t}$

Op: $\dfrac{\Gamma \vdash e_1 \; : \; \mathbf{int} \qquad \Gamma \vdash e_2 \; : \; \mathbf{int}}{\Gamma \vdash e_1 + e_2 \; : \; \mathbf{int}}$

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

Example: Type Checking

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



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:

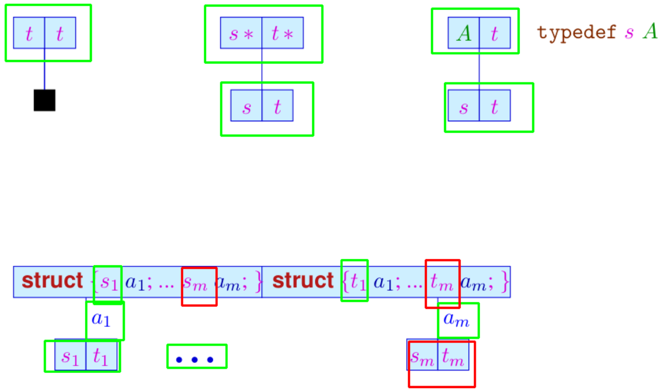
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

`typedef` $A\ t$

(we omit the Γ). Then define the following rules:

Rules for Well-Typedness



Example:

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

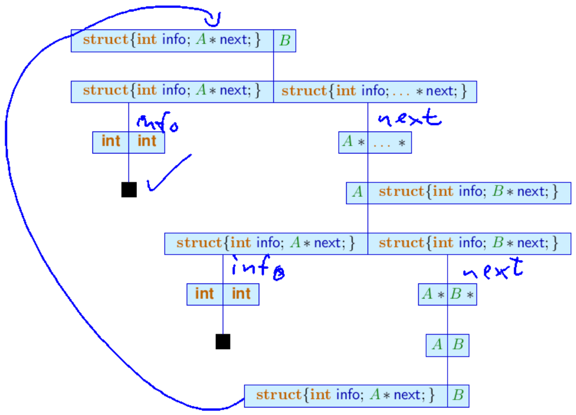
We ask, for instance, if the following equality holds:

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

We construct the following deduction tree:

Proof for the Example:

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



struct{int info; $A *$ next; } | B
struct{int info; $A *$ next; } | struct{int info; … $*$ next; }
int | int
$A *$ | … $*$
A | struct{int info; $B *$ next; }
struct{int info; $A *$ next; } | struct{int info; $B *$ next; }
int | int
$A *$ | $B *$
A | B
struct{int info; $A *$ next; } | B

Implementation

We implement a function that implements the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are *not equal*
- if the deduction rule for expanding a type definition applies, the function is called recursively with a *potentially larger* type
- in case an equivalence query appears a second time, the types are *equal by definition*

Overloading and Coercion

Some operators such as $+$ are *overloaded*:

- $+$ has *several possible* types
  for example: `int` $+$(`int`,`int`), `float` $+$(`float`, `float`)
  but also `float`$*$ $+$(`float`$*$, `int`), `int`$*$ $+$(`int`, `int`$*$)
- depending on the type, the operator $+$ has a different implementation
- determining which implementation should be used is based on the type of the *arguments* only

Subtypes

On the arithmetic basic types **`char`**, **`int`**, **`long`**, etc. there exists a rich *subtype* hierarchy

Subtypes

$t_1 \leq t_2$, means that the values of type t_1

1. form a subset of the values of type t_2;
2. can be converted into a value of type t_2;
3. fulfill the requirements of type t_2;
4. are assignable to variables of type $t2$.

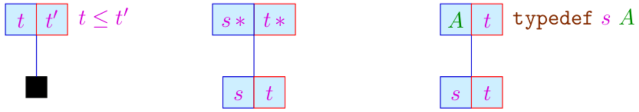
Example: Subtyping

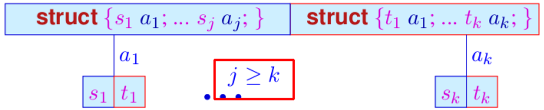
Extending the subtype relationship to more complex types, observe:

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

- we want `extractInfo` to be applicable to all argument structures that return a `string` typed field for accessor `info`
- the idea of subtyping on values is related to subclasses
- we use deduction rules to describe when $t_1 \leq t_2$ should hold…

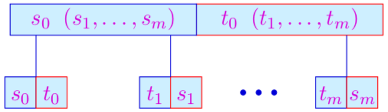
Rules for Well-Typedness of Subtyping





struct $\{int\ u, \boxed{int\ v}\}$ x;
struct $\{int\ u\}$ y;
$y = x$;

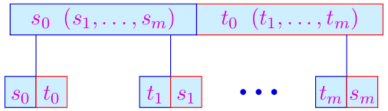
Rules and Examples for Subtyping



Examples:

struct $\{$**int** a; **int** b; $\}$		**struct** $\{$**float** a; $\}$
int (**int**)		**float** (**float**)
int (**float**)	≤	**float** (**int**)

Rules and Examples for Subtyping



Examples:

$\mathbf{struct}\ \{\mathbf{int}\ a;\ \mathbf{int}\ b;\}$	$\mathbf{struct}\ \{\mathbf{float}\ a;\}$
$\mathbf{int}\ (\mathbf{int})$	$\mathbf{float}\ (\mathbf{float})$
$\mathbf{int}\ (\mathbf{float})$	$\mathbf{float}\ (\mathbf{int})$

Definition

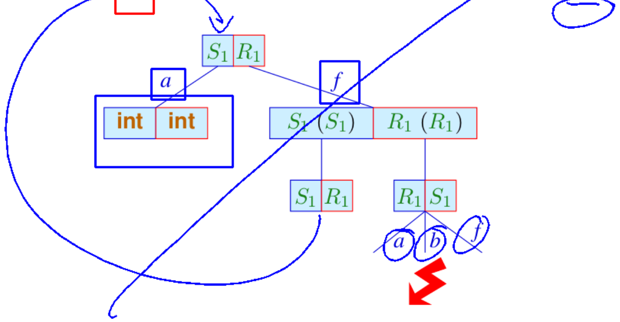
Given two function types in subtype relation $s_0(s_1,\dots s_n) \le t_0(t_1,\dots t_n)$ then we have

- co-variance of the return type $s_0 \le t_0$ and
- contra-variance of the arguments $s_i \ge t_i$ für $1 < i \le n$

Subtypes: Application of Rules (I)

Check if $S_1 \le R_1$:

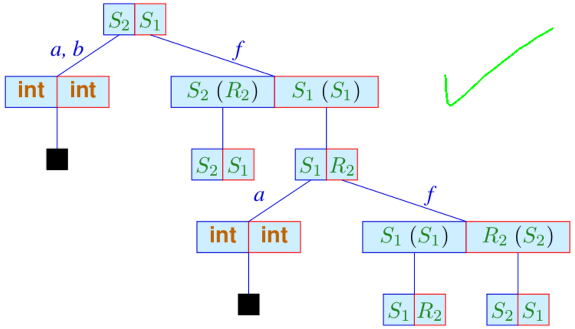
$R_1 = \mathbf{struct}\ \{\mathbf{int}\ a;\ R_1\ (R_1)\ f;\}$
$S_1 = \mathbf{struct}\ \{\mathbf{int}\ a;\ \mathbf{int}\ b;\ S_1\ (S_1)\ f;\}$
$R_2 = \mathbf{struct}\ \{\mathbf{int}\ a;\ R_2\ (S_2)\ f;\}$
$S_2 = \mathbf{struct}\ \{\mathbf{int}\ a;\ \mathbf{int}\ b;\ S_2\ (R_2)\ f;\}$



Subtypes: Application of Rules (II)

Check if $S_2 \le S_1$:

$R_1 = \mathbf{struct}\ \{\mathbf{int}\ a;\ R_1\ (R_1)\ f;\}$
$S_1 = \mathbf{struct}\ \{\mathbf{int}\ a;\ \mathbf{int}\ b;\ S_1\ (S_1)\ f;\}$
$R_2 = \mathbf{struct}\ \{\mathbf{int}\ a;\ R_2\ (S_2)\ f;\}$
$S_2 = \mathbf{struct}\ \{\mathbf{int}\ a;\ \mathbf{int}\ b;\ S_2\ (R_2)\ f;\}$



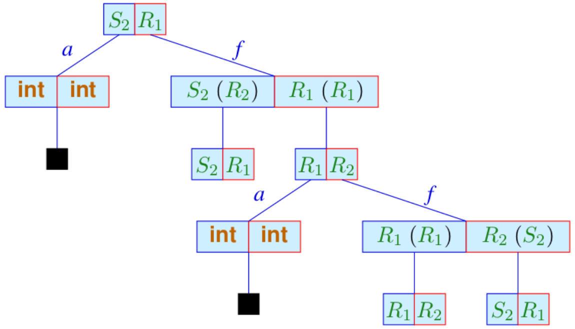
Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- Java generalizes records to objects/classes where a sub-class A inheriting form base class O is a subtype $A \le O$
- subtype relations between classes must be explicitly declared
- inheritance ensures that all sub-classes contain all (visible) components of the super class
- a shadowed (overwritten) component in A must have a subtype of the the component in O
- Java does not allow argument subtyping for methods since it uses different signatures for overloading

Subtypes: Application of Rules (III)

Check if $S_2 \leq R_1$:

$$\begin{array}{lcl} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\,(R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\,(S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\,(S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\,(R_2)\ f;\} \end{array}$$



Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- Java generalizes records to objects/classes where a sub-class A inheriting form base class O is a subtype $A \leq O$
- subtype relations between classes must be explicitly declared
- inheritance ensures that all sub-classes contain all (visible) components of the super class
- a shadowed (overwritten) component in A must have a subtype of the the component in O
- Java does not allow argument subtyping for methods since it uses different signatures for overloading

---

Topic:

Code Synthesis