Script generated by TTT

Title: Petter: Compilerbau (05.07.2018)

Date: Thu Jul 05 14:15:10 CEST 2018

Duration: 84:59 min

Pages: 24

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

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

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

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:

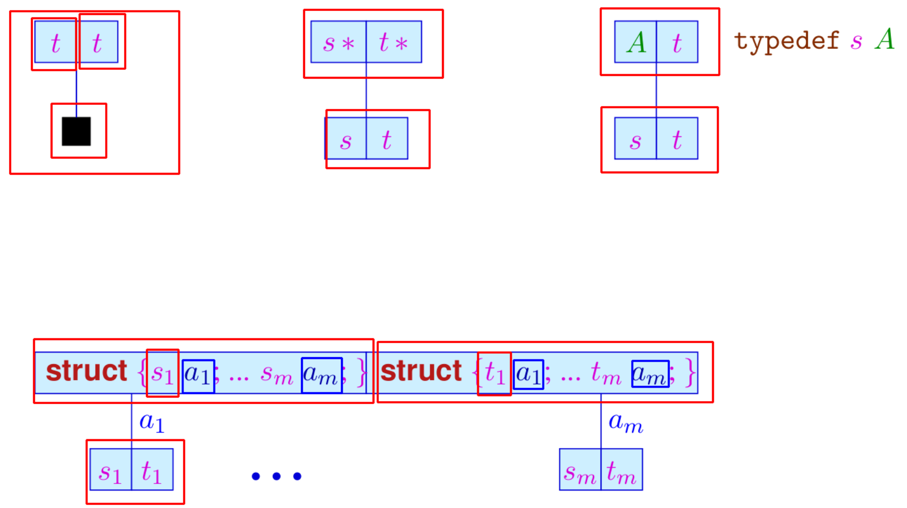
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

$$\texttt{typedef}\ A\ t$$

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

Rules for Well-Typedness



Example:

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

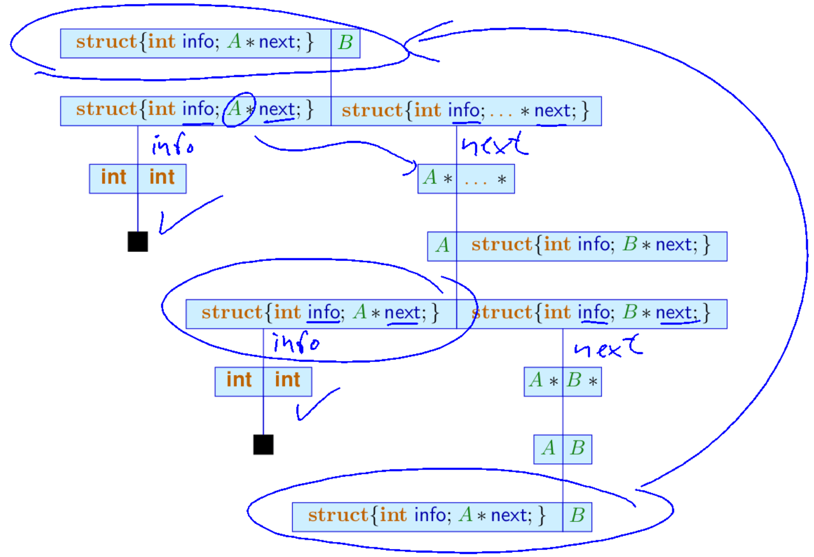
We ask, for instance, if the following equality holds:

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

We construct the following deduction tree:

Proof for the Example:

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



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*

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 $t2$.

$t_1 \leq t_2$

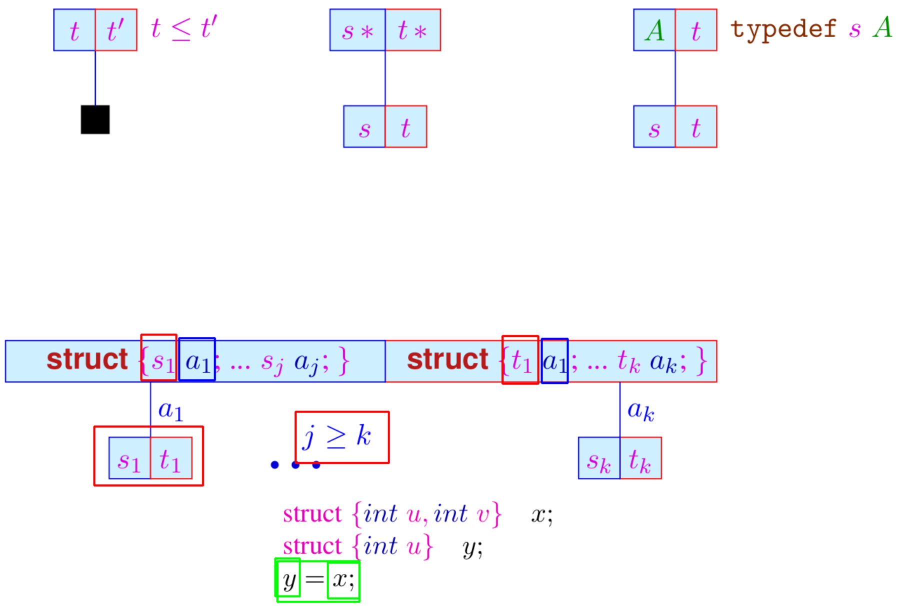
Example: Subtyping

Extending the subtype relationship to more complex types, observe:

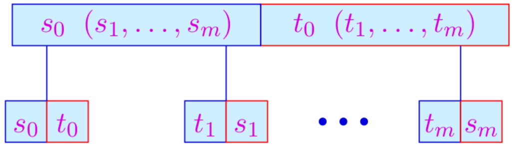
```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

- 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



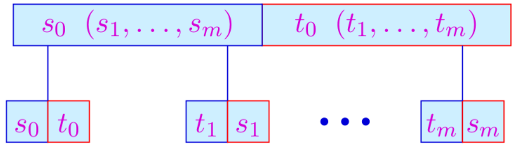
Rules and Examples for Subtyping



Examples:

struct {int a; int b; }	struct {float a; }
int (int)	float (float)
int (float)	float (int)

Rules and Examples for Subtyping



Examples:

struct {int a; int b; }	struct {float a; }
int (int)	float (float)
int (float)	float (int)

Definition

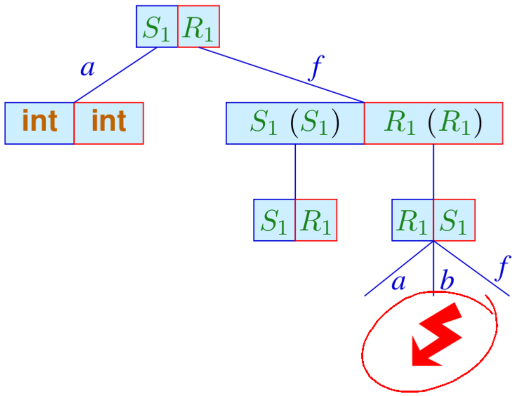
Given two function types in subtype relation $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ then we have

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

Subtypes: Application of Rules (I)

Check if $S_1 \leq R_1$:

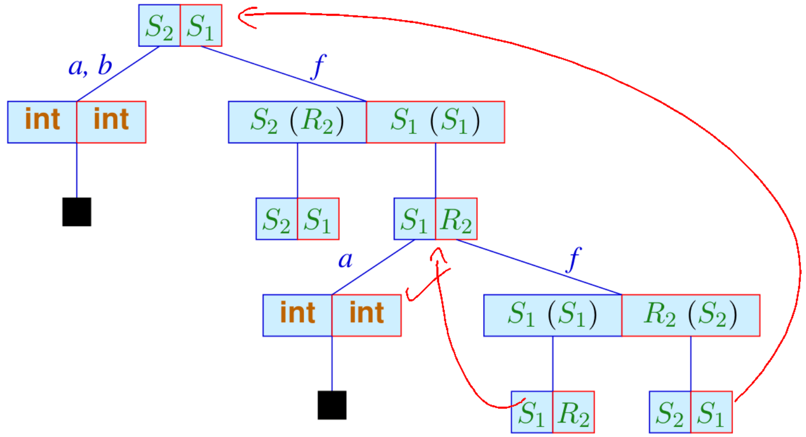
$$\begin{array}{lcl} R_1 & = & \text{struct } \{\text{int } a; \ R_1 \, (R_1) \ f; \} \\ S_1 & = & \text{struct } \{\text{int } a; \ \text{int } b; \ S_1 \, (S_1) \ f; \} \\ R_2 & = & \text{struct } \{\text{int } a; \ R_2 \, (S_2) \ f; \} \\ S_2 & = & \text{struct } \{\text{int } a; \ \text{int } b;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



Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

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



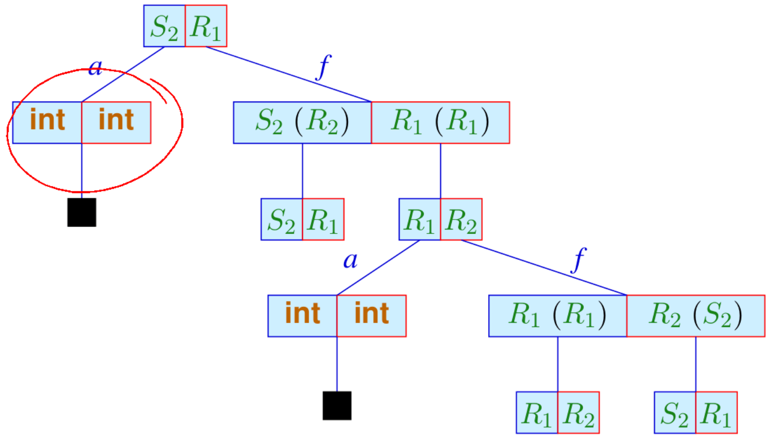
Discussion

- 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 structs 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

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 structs 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

Chapter 1: The Register C-Machine

The Register C-Machine (R-CMa)

We generate Code for the Register C-Machine.
The Register C-Machine is a virtual machine (VM).

- there exists no processor that can execute its instructions
- . . . but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no **`double`**, **`float`**, **`char`**, **`short`** or **`long`** types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

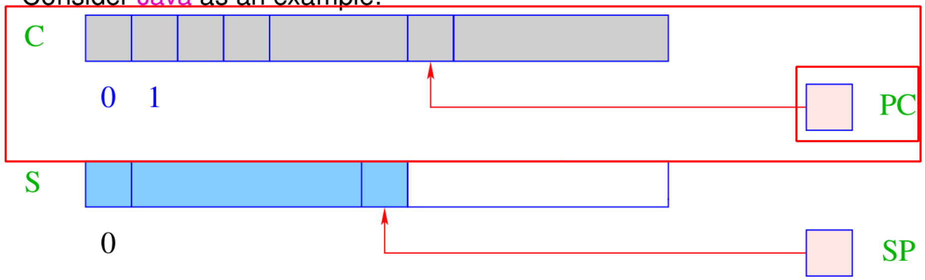
Virtual Machines

A virtual machine has the following ingredients:

- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system, namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Components of a Virtual Machine

Consider Java as an example:



A virtual machine such as the Dalvik VM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order ⇝ stack.
- SP: (≙ stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap

Executing a Program

- the machine loads an instruction from C[PC] into the instruction register IR in order to execute it
- before evaluating the instruction, the PC is incremented by one

```
while (true) {
  IR = C[PC]; PC++;
  execute (IR);
}
```

- node: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system

Chapter 2:

Generating Code for the Register C-Machine