Script generated by TTT

Title: Petter: Compilerbau (29.06.2015)

Date: Mon Jun 29 14:17:41 CEST 2015

Duration: 88:34 min

Pages: 43

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

---

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Consider the declaration of an alternating linked list in C:

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  double info;
  struct list0* next;
}
```

Forward Declarations

Most programming language admit the definition of recursive data types and/or recursive functions.

- a recursive definition needs to mention a name that is currently being defined or that will be defined later on
- old-fashion programming languages require that these cycles are broken by a *forward* declaration

Consider the declaration of an alternating linked list in C:

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  double info;
  struct list0* next;
}
```

⇝ the first declaration `struct list1;` is a forward declaration.

Declarations of Function Names

An analogous mechanism is need for (recursive) functions:

- in case a *recursive function* merely calls itself, it is sufficient to add the name of a function to the symbol table before visiting its body; example:

```
int fac(int i) {
  return i*fac(i-1);
}
```

Declarations of Function Names

An analogous mechanism is need for (recursive) functions:

- in case a *recursive function* merely calls itself, it is sufficient to add the name of a function to the symbol table before visiting its body; example:

```
int fac(int i) {
  return i*fac(i-1);
}
```

- for *mutually recursive functions* all function names at that level have to be entered (or declared as forward declaration). Example: ML and C:

```
fun odd 0 = false
  | odd 1 = true
  | odd x = even (x-1)
and even 0 = true
  | even 1 = false
  | even x = odd (x-1)
```

```
int even(int x);
int odd(int x) {
  return (x==0 ? 0 :
    (x==1 ? 1 : even(x-1)));
}
int even(int x) {
  return (x==0 ? 1 :
    (x==1 ? 0 : odd(x-1)));
}
```

Overloading of Names

The problem of using names before their declarations are visited is also common in object-oriented languages:

- for OO-languages with inheritance, a method's signature contributes to determining its binding
  - qualifies a function symbol with its parameters types
  - also requires resolution of parameter and return types
- the base class must be visited before the derived class in order to determine if declarations in the derived class are correct

---

Overloading of Names

The problem of using names before their declarations are visited is also common in object-oriented languages:

- for OO-languages with inheritance, a method's signature contributes to determining its binding
  - qualifies a function symbol with its parameters types
  - also requires resolution of parameter and return types
- the base class must be visited before the derived class in order to determine if declarations in the derived class are correct

Once the types are resolved, other semantic analyses can be applied such as *type checking* or *type inference*.

Multiple Classes of Identifiers

int i;

typedef int i;

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

---

Multiple Classes of Identifiers

typedef int i;

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

In some cases a declaration may *change* the class of an identifier; for example, a **`typedef`** in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier

---

Multiple Classes of Identifiers

Some programming languages distinguish between several classes of identifiers:

- C: variable names and type names
- Java: classes, methods and fields
- Haskell: type names, constructors, variables, infix variables and -constructors

In some cases a declaration may *change* the class of an identifier; for example, a **`typedef`** in C:

- the scanner generates a different token, based on the class into which an identifier falls
- the parser informs the scanner as soon as it sees a declaration that changes the class of an identifier

the interaction between scanner and parser is *problematic*!

Type Synonyms and Variables 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 Synonyms and Variables 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` | ...

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced

---

Type Synonyms and Variables 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` | ...

Problem:

- parser adds `point_t` to the table of types when the declaration is reduced
- parser state has at least one look-ahead token

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

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

Solution is difficult:

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

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

Solution is difficult:

- try to fix the look-ahead inside the parser

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

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

id (id) id

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: S/R- & R/R- Conflicts!!
  `typename` → `identifier`

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

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

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: S/R- & R/R- Conflicts!!
  `typename` → `identifier`
- register type name earlier

Type Synonyms and Variables in C: Solutions

Relevant C grammar:

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

Solution is difficult:

- try to fix the look-ahead inside the parser
- add a rule to the grammar: S/R- & R/R- Conflicts!!
  `typename` → `identifier`
- register type name earlier
  - separate rule for `typedef` production
  - call alternative declarator production that registers `identifier` as type name

---

Chapter 3:
Type Checking

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. for example: `int`, `void*`, `struct { int x; int y; }`.

Types are useful to

- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.

Type Expressions

typedefed id

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 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

example for type constructors in C:

- records: `struct { ` $t_1\ a_1; \ldots t_k\ a_k;$ ` }`
- pointer: $\boxed{t *}$
- arrays: $\boxed{t\ [\,]}$
  - the size of an array can be specified
  - the variable to be declared is written between t and $[n]$
- functions $\boxed{t}\boxed{(t_1, \ldots, t_k)}$
  - the variable to be declared is written between t and $(t_1, \ldots, t_k)$
  - in ML function types are written as: $t_1 * \ldots * t_k \rightarrow t$

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_tp;
struct list {
  int info;
  list_tp next;
}
list_tp head;
```

Type Checking

Problem:

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

?

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?

Example:

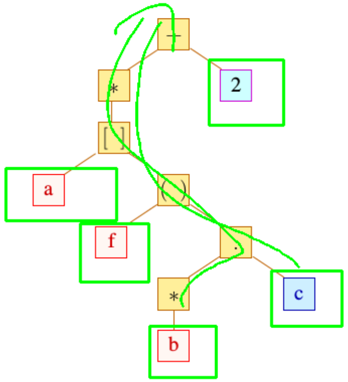
```
struct list { int info; struct list* next; };
int f(struct list* l) { return 1; };
struct { struct list* c;}* b;
int* a[11];
```

Consider the expression:

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

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

Type Systems for C-like Languages

More rules for typing an expression:

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

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

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

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

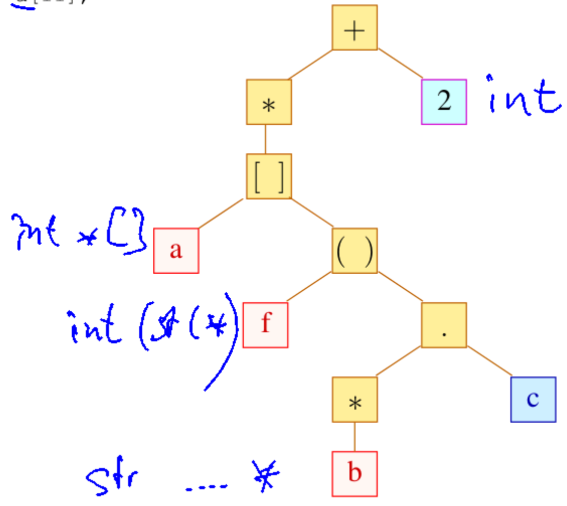
Cast: $\dfrac{\boxed{\Gamma \vdash e \; : \; t_1} \qquad t_1 \text{ can be converted to } \boxed{t_2}}{\Gamma \vdash \boxed{(t_2)\; e} : \boxed{t_2}}$

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];
```

$\}$:



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

Type Systems for C-like Languages

More rules for typing an expression:

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

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

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

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

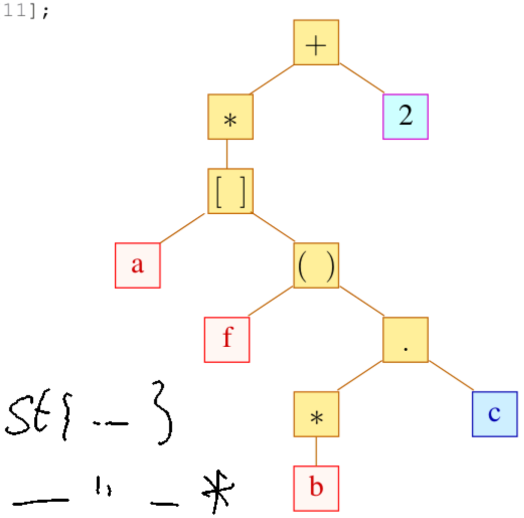
Op: $\dfrac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$

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 $\Gamma = \{$

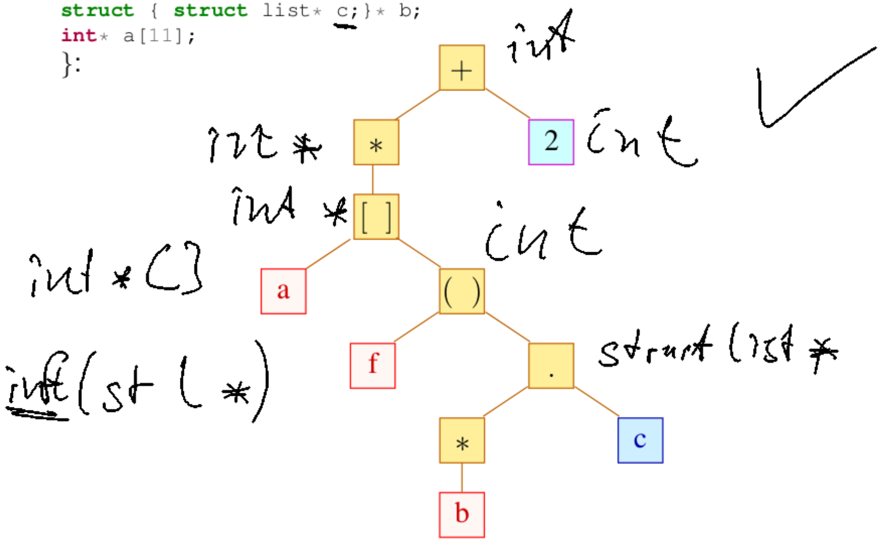
```
struct list { int info; struct list* next; };
int f(struct list* l);
struct { struct list* c;}* b;
int* a[11];
```
}:



Example: Type Checking

Given expression `*a[f(b->c)]+2` and $\Gamma = \{$

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



Equality of Types

Summary 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:

```
typedef struct B {
          struct A 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 type 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:

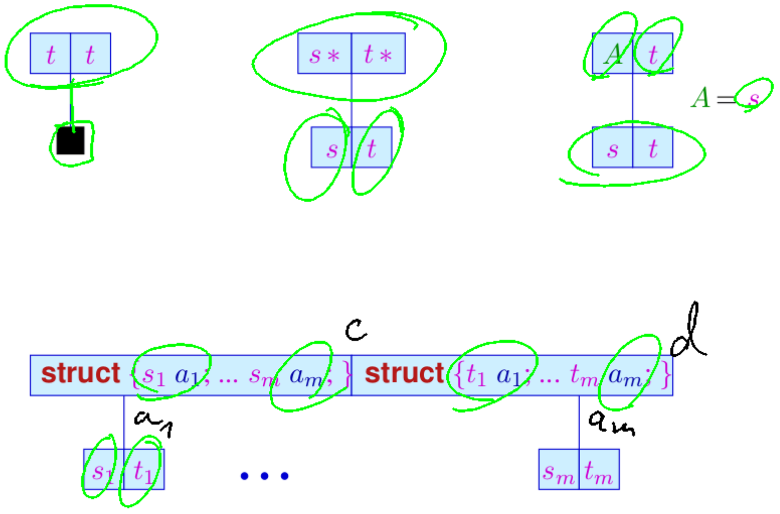
- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type equalities of the form:

$$\boxed{A = t}$$

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

Rules for Well-Typedness



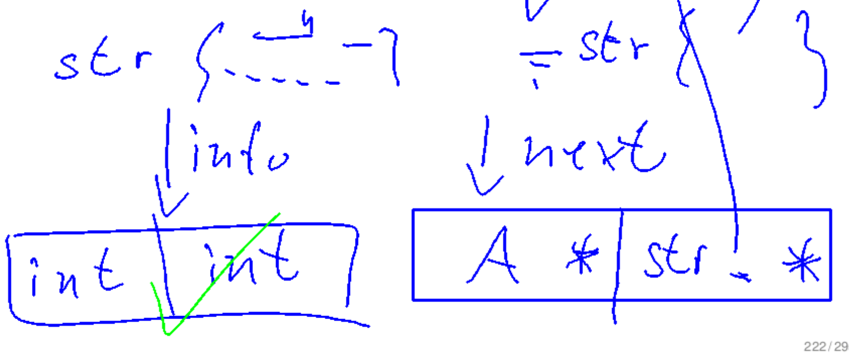
Example:

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

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

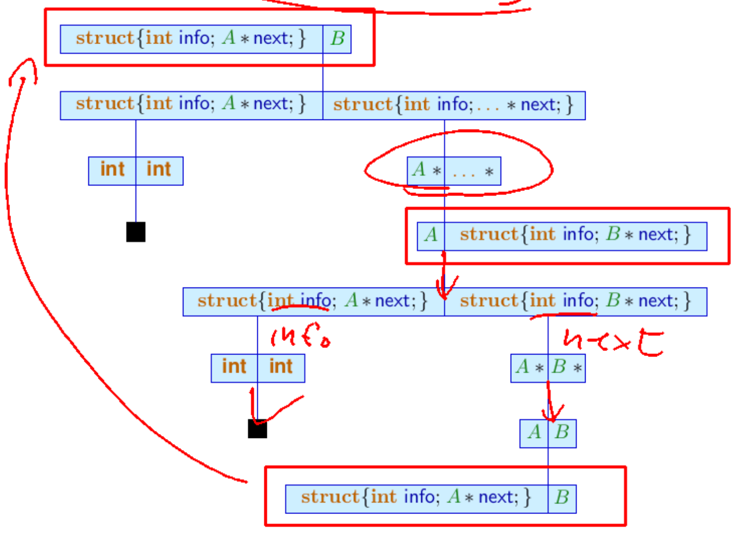
$$\boxed{\text{struct}\ \{\text{int info};\ A * \text{next};\ \}} = \boxed{B}$$

We construct the following deduction tree:



Proof for the Example:

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



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

struct{int info; $A *$ next; } | struct{int info; $\ldots *$ next; }

int | int

■

$A *$ | $\ldots *$

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
- during the construction of the proof tree, an equivalence query might occur several times
- in case an equivalence query appears a second time, the types are by definition equal

Termination?

- the set D of all declared types is finite
- there are no more than $|D|^2$ different equivalence queries
- repeated queries for the same inputs are are automatically satisfied

$\leadsto$ termination is ensured

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 *arguments* only