Compiler Construction

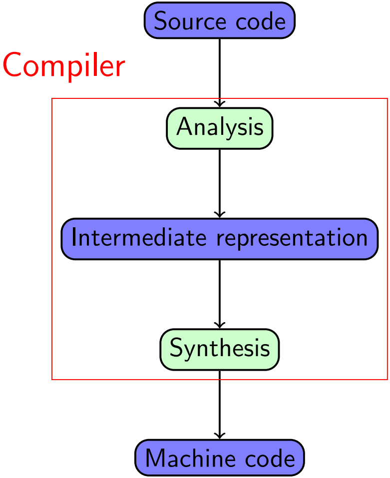
Chapter 4 – Type Checking

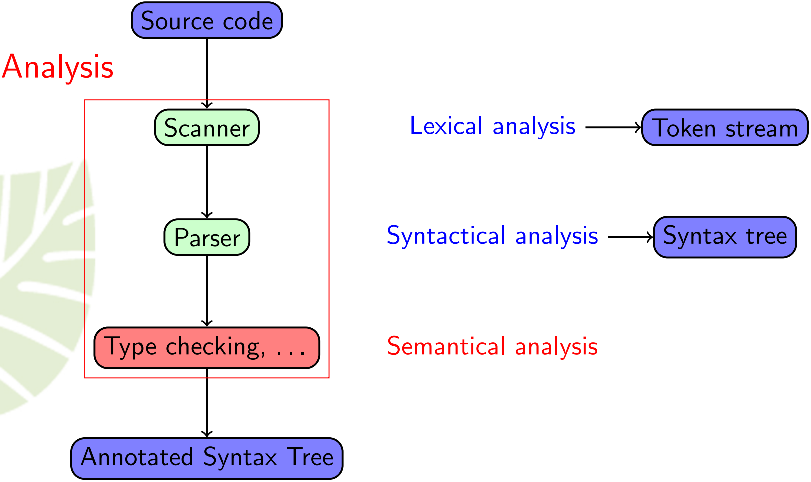
Prof. Dr. Jörg Kreiker
joerg.kreiker@informatik.hs-fulda.de

Fachbereich Angewandte Informatik
Hochschule Fulda – University of Applied Sciences

May 18, 2017

Compiler Overview




Source code
Analysis
Scanner
Parser
Type checking, ...
Annotated Syntax Tree
Lexical analysis
Token stream
Syntactical analysis
Syntax tree
Semantical analysis

Type Checking

- In most modern programming languages variables and functions are typed
- Examples
  - `int`
  - `void*`
  - `struct { int x; int y; }`
- Useful
  - Memory management
  - avoiding run-time errors
  - well-typed programs don't go wrong
- Types can be
  - declared and then checked
  - inferred

Type Expressions

- Types are defined by type expressions
- Type expressions
  1. base types like `int`, `float`
  2. type constructors applied to types
- Examples of type constructor expressions
  - record:
  - pointer: `t*`
  - arrays: `t[]`
  - functions: `t(t1,...,tk)`

Type Names

- Synonym for type expression
- Omitted from project
- Used as a short-hand notion

  ⇒ `typedef struct { int x; int y; } point_t`
- Used to define recursive types

Two definitions of singly-linked lists

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

struct list* head;
```

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

Type Checking

Given: set of type declarations $\Gamma = [t_1\ x_1, \ldots, t_k\ x_k]$

Check: Can expression `e` have type `t`?

Random type expressions

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

Is the following expression type correct? `*a[f(b->c)]*2`

Type Checking the AST

Traverse the syntax tree bottom-up!

Variables	look up type in type environment Γ
Constants	determine type directly
Inner nodes	apply typing rules

Typing Rules

Formally we consider statements of the form

$$\Gamma \vdash e : t$$

In type environment Γ expression e has type t.

Axioms

$$\begin{array}{lll} \textsc{Const}: & \Gamma \vdash c : t_c & (t_c \text{ type of constant } c) \\ \textsc{Var}: & \Gamma \vdash x : \Gamma(x) & (x \text{ variable}) \end{array}$$

Rules

$$\textsc{Ref}: \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \&e : t*} \qquad\qquad \textsc{Deref}: \quad \frac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}$$

More C Rules

ARRAY1: $$\frac{\Gamma \vdash e_1 : t* \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1[e_2] : t}$$

ARRAY2: $$\frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1[e_2] : t}$$

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

APP: $$\frac{\Gamma \vdash e : t(t_1 \ldots t_k) \quad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_k : t_k}{\Gamma \vdash e(e_1 \ldots e_k) : t}$$

OP: $$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

CAST: $$\frac{\Gamma \vdash e : t_1 \quad t_1 \text{ converts to } t_2}{\Gamma \vdash (t_2)e : t_2}$$

Type Equality

- In order to apply typing rules, type equality needs to be checked
- In C: `struct A { }` and `struct B { }` are considered different types
- Extending a record works only by embedding it into a larger one
- Type synonmys are considered equal

⇒ `typedef int C;` means that `int` and `C` become equal types

Overloading

- Some operators like `+` are overloaded
- Possible types for $7+$ (non-exhaustive)
  - `int +(int,int)`
  - `float +(float,float)`
  - `float* +(float*, int)`
- Depending on its type, `+` may have different implementations
- Arguments determine which implementation

```
1 + 2.4
```

- Instead of defining all possible combinations of argument types, argument types are converted

⇒ Coercion

- Such a conversion may generate extra code
- Usually one converts to supertypes
  - → expression above will have type `float`

C features particular coercion rules for integer types: promotion

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

Integer promotion may lead to subtle mistakes

What's the output?

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < ui);
```

Good for you: Project requires only ints.

Summary

- Why types?
- What are types (expressions, names)?
- Checking algorithm and typing rules
- C specific stuff