6. Type Analysis

Eva Rose Kristoffer Rose

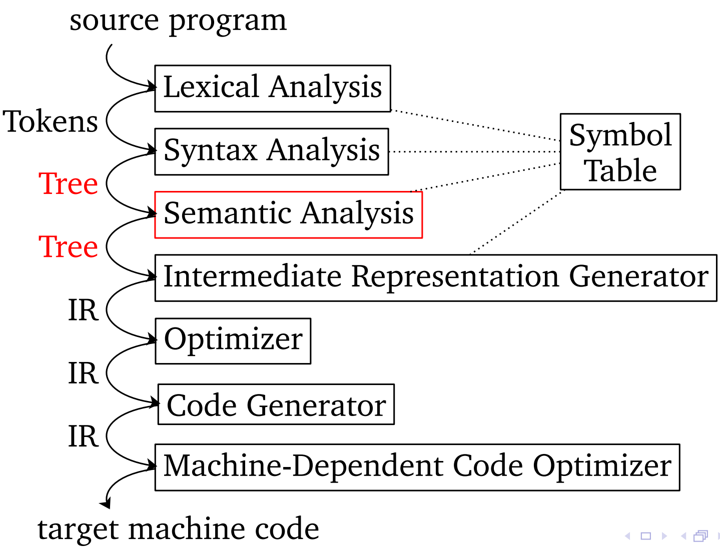
NYU Courant Institute

Compiler Construction (CSCI-GA.2130-001)

http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-6.pdf

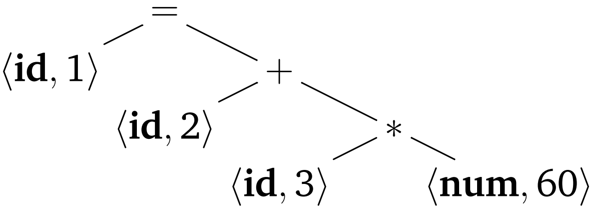
October 9, 2014

Context


source program
Lexical Analysis
Tokens
Syntax Analysis
Tree
Semantic Analysis
Tree
Intermediate Representation Generator
IR
Optimizer
IR
Code Generator
IR
Machine-Dependent Code Optimizer
Symbol Table
target machine code

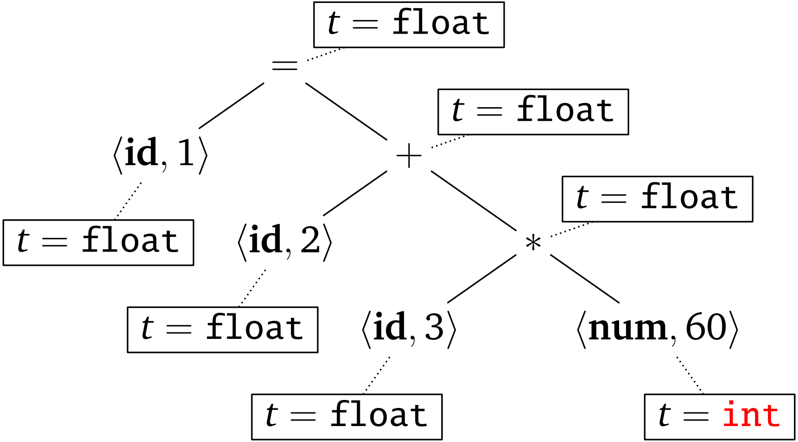
From Abstract Syntax Tree (AST)

```
position = initial + rate * 60
```



id	*lexeme*
1	`position`
2	`initial`
3	`rate`

Type Checking Gives Annotated AST. . .



id	*lexeme*	*t*
1	position	float
2	initial	float
3	rate	float

Type Checking

- Languages come with type system:
  - Set of rules
  - What are the basic value types?
  - How can existing values be composed and decomposed to form new types?
- Compiler's job:
  - Assign type expressions to each component
  - Determine that these type expressions conform to type systems

Type Checking

- Languages come with type system:
  - Set of rules
  - What are the basic value types?
  - How can existing values be composed and decomposed to form new types?
- Compiler's job:
  - Assign type expressions to each component
  - Determine that these type expressions conform to type systems

Purpose of Types

- Development Help (completion, documentation)
- Error detection (static and dynamic checks)
- Error Prevention (disambiguate operations)
- Optimization purposes (storage layout)

Dynamic *vs* Static Type Checking

- Type checking can be done dynamically for any language (*i.e.*, at *run-time*)
  - *compiler generates code to do runtime checks*
- Usually preferred to do type checking statically (*i.e.*, at *compile-time*)

Properties of Type Checking

- A sound type system eliminates the need for dynamic checking
- A language is strongly typed if compiler guarantees that type errors cannot happen at runtime
  - Examples: Java, C#, Ruby, Python, OCaml, Haskell

Rules for Type Checking

- Type Synthesis
  - Builds the type system of an expression from the types of subexpressions
  - Requires names to be declared before use
- Type Inference
  - Determines the type of a construct from the way it is used
  - Complex

Type Synthesis (Denotational Formulation)

$$\text{if } f : S \to T \text{ and } x : S \text{ then } f(x) : T$$

$$\frac{f : S \to T \qquad x : S}{f(x) : T}$$

Type Synthesis (Denotational Formulation)

if $f : S \to T$ and $x : S$ then $f(x) : T$

$$\frac{f : S \to T \qquad x : S}{f(x) : T}$$

Type Inference (Denotational Formulation)

$$\frac{f(x) : T}{\exists S[\, f : S \rightarrow T \qquad x : S \,]}$$

Values in Programming Languages

Some examples of types:

- `short`, **`int`**, `long`, `char`, `bool`, **`float`**, …
- **`int`**`[2][3]`, `struct Link`, `String`, …
- *class names*, …

Type Concepts

- primitive types (boolean, integral, floating point, chars, strings*)
- composite types (arrays, records, strings*)
- reference types (pointers, object references)
- abstract data types *aka.* ADT (class)
- subtype (class hierarchies)
- recursive types (linked lists)
- function types (ordering)
- ...

Type Concepts

- primitive types (boolean, integral, floating point, chars, strings*)
- composite types (arrays, records, strings*)
- reference types (pointers, object references)
- abstract data types *aka.* ADT (class)
- subtype (class hierarchies)
- recursive types (linked lists)
- function types (ordering)
- ...

Type Specification

- Language manual
- Compiler manual
- Machine architecture

Type Specification

- Language manual
- Compiler manual
- Machine architecture

Type Specification

- Language manual
- Compiler manual
- Machine architecture

Some (Modern) Language Type Implementation Details

Type	*Implementation (bit)*	*Language Specifications*
`byte`	`8`	`JVM-Spec: 8`
`short`	`16 (or 32)`	`JVM-Spec: 16, C/C++ unspec.`
`char`	`8 or 16`	`JVM-Spec: 16, C/C++: 8`
`int`	`32 or 64`	`JVM-Spec: 32, C/C++` $\geq$ `16`
`long`	`64 (or 128)`	`JVM-Spec: 64, C/C++` $\geq$ `32`
`float`	`32 (or 64)`	`JVM-Spec: 32, C/C++ unspec.`
`double`	`64 (or 128)`	`JVM-Spec: 64, C/C++ unspec.`

C: part of system – read *limits.h* for min and max specifications.

Traditional/Old System (C-compiler) Type Details

Type	*Implementation (bit)*
`short`	8
int	**16**
`long`	32

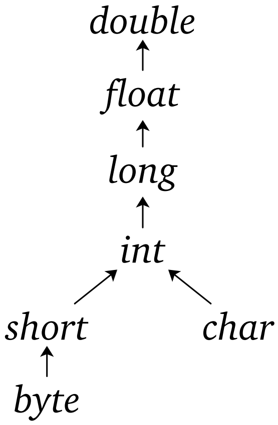
Today:

- unix systems : specify **integers** as **64** bit
- window systems : specify **integers** as **32** bit
- computers (AMD64): ALU datapath, registers are 64 bit,
- mobile phones (ARM): ditto, but 32 bit (moving to 64)

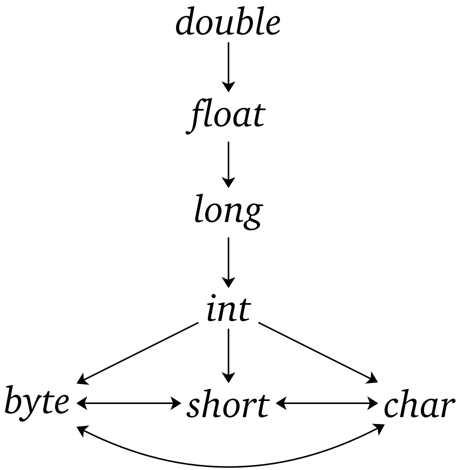
Type Conversions

- Widening (preservation) – often implicit (*coercions*).
- Narrowing (lossy) – often explicit (*casts*).

Widening Conversion Hierarchy (primitive Java types)



Narrowing Conversion Hierarchy (primitive Java types)



Example Type Conversion I

```
float position;
int initial;
float rate;
...
position = initial + rate * 60.00;
```

Example Type Conversion II

```
float position;
int initial;
float rate;
...
position = initial + rate * 60;
```

Example Type Conversion III

```
int position;
int initial;
float rate;
...
position = initial + rate * 60;
```

Explicit Conversions (cast)

Tell the compiler that you MEAN this type conversion (type cast).

```
...
...
...
position = initial + ((int) rate) * 60;
```

Example Java Conversions. . .

```
"U" + 2
2 + "U"
"friends are " + true
true + " types"
```

Example Java Conversions. . .

```
"U" + 2     OK
2 + "U"
"friends are " + true
true + " types"
```

Example Java Conversions. . .

```
"U" + 2        OK
2 + "U"        Fail
"friends are " + true
true + " types"
```

Example Java Conversions. . .

```
"U" + 2        OK
2 + "U"        Fail
"friends are " + true     OK
true + " types"
```

Example Java Conversions. . .

```
"U" + 2        OK
2 + "U"        Fail
"friends are " + true     OK
true + " types"     Fail
```

Operator and Function Overloading

- $+$ meaning addition or string concatenation (Java)
- user defined functions (Java)

```
void err() {...}
void err(String s) {...}
```

Operator and Function Overloading

- $+$ meaning addition or string concatenation (Java)
- user defined functions (Java)

```
void err() {...}
void err(String s) {...}
```

Type Equivalence Approaches

What are the values "in" a type?

Approach	**Definition**
Denotational	set of values (*Type Theory*)
Abstract Data Types (ADT)	set of operations (*OO-systems*)

When are two types considered the same?

Approach	**Definition**
Denotational	set of values (*Type Theory*)
Abstract Data Types (ADT)	set of operations (*OO-systems*)
Named typing	set of names
Structural typing	set of structures (*duck* typing)

Type Expressions (TE)

A way so assign structure to types:

- primitive types are TEs,
- ***type constructor*** operator assigned to a TE.

TE Examples

- short, int, long, float, char, ...
- T[]
- Map$\langle$S,T$\rangle$
- S$\times$T
- $S \to T$

Type structure: AST for *int*[2][3]

$$T \to B\ C \tag{1}$$

$$B \to \textbf{int} \mid \textbf{float} \tag{2}$$

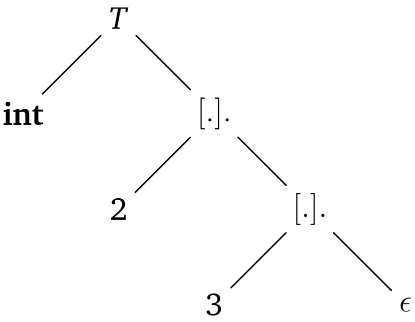
$$C \to [\textbf{num}]C \mid \epsilon \tag{3}$$

Type structure: AST for *int*[2][3]

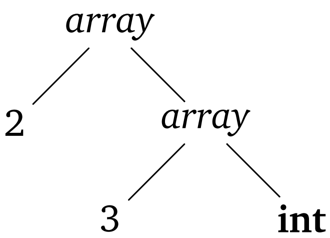
$$T \to B\ C \qquad (1)$$

$$B \to \mathbf{int} \mid \mathbf{float} \qquad (2)$$

$$C \to [\mathbf{num}]C \mid \epsilon \qquad (3)$$



Type structure: type expression for *int*[2][3]



Example: SDD for array TEs

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t;\ \ C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = \mathit{integer}$
$B \rightarrow \textbf{float}$	$B.t = \mathit{float}$
$C \rightarrow [\textbf{num}]C_1$	$C.t = \mathit{array}(\textbf{num}.\mathit{val}, C_1.t);\ \ C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

b-inherited; t-synthesized

Example: SDD with Type Storage Attribute

Production	Semantic Rules
$T \to BC$	$T.t = C.t; T.width = C.width;$ $C.b = B.t; w = B.width;$
$B \to \textbf{int}$	$B.t = integer; B.width = 4;$
$B \to \textbf{float}$	$B.t = float; B.width = 8;$
$C \to [\textbf{num}]C_1$	$C.t = array(\textbf{num}.val, C_1.t);$ $C.width = \textbf{num}.val \times C_1.width;$
$C \to \epsilon$	$C.t = C.b; C.width = w;$

b inherited; $t, width$ synthesized

Example: SDT for Translation with Types

PRODUCTION	SEMANTIC ACTIONS
$S \rightarrow \textbf{id} = E$	$\{\, gen(\text{top . get}(\textbf{id}.lexeme)' =' E.addr); \}$
$E \rightarrow E_1 + E_2$	$\{\, E.addr' =' \; E_1.addr' +' \; E_2.addr); \}$
$\mid - E$	$\{\, E.addr = \text{new}\; Temp();$ $gen(E.addr' =' \; '\textbf{minus}' E_1.addr); \}$
$\mid (E_1)$	$\{\, E.addr = E_1.addr; \}$
$\mid \textbf{id}$	$\{\, E.addr = \text{top . get}(\textbf{id}.lexeme)$

Example: SDT for Translation with Types and Type Conversion

Production	Semantic Actions
$E \rightarrow E_1 + E_2$	$\{\ E.type = \mathbf{max}(E_1.type, E_2.type);$ $a_1 = \mathbf{widen}(E_1.addr, E_1.type, E.type);$ $a_2 = \mathbf{widen}(E_2.addr, E_2.type, E.type);$ $E.addr = \text{new } Temp();$ $gen(E.addr\ \text{'='}\ a_1\ \text{'+'}\ a_2);\ \}$

Pseudo-code for Widening

```
Addr widen(Addr a, Type t, Type W)
    if (t=w) return a;
    else if (t=integer and w=float){
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
```

Example: Type Synthesis SDD

Production	Semantic Rules	
$E \to E_1 + E_2$	$E.t = \text{Unif}(E_1.t, E_2.t)$	(1)
$\mid E_1 * E_2$	$E.t = \text{Unif}(E_1.t, E_2.t)$	(2)
$\mid$ **int**	$E.t = \text{Int}$	(3)
$\mid$ **float**	$E.t = \text{Float}$	(4)

Example: Type Synthesis HACS Sorts

```
sort Type | Int | Float;

sort Type | scheme Unif(Type,Type);
Unif(Int, Int) →Int;
Unif(Float, #) →Float;
Unif(#, Float) →Float;
```

Questions?

evarose@cs.nyu.edu krisrose@cs.nyu.edu