Type Analysis

Eva Rose Kristoffer H. Rose

NYU Courant Institute

March 10, 2014

Outline

Types in Programming Languages

What is a type?

Types in Programming Languages

Some examples of types:

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

Types in Programming Languages

Purpose of types:

- Error detection (static and dynamic checks),
- Disambiguate operations (error prevention).
- Optimization purposes (storage layout).

Types in Programming Languages

Type implementation/storage requirements according to:

- language spec,
- compiler spec,
- machine architecture spec.

Types in Programming Languages

Some (modern) language implementation details:

Type	*Implementation (bit)*	*language specification*
`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++ unspec.`
`long`	64 (or 128)	`JVM-Spec` : 64, `C/C++ unspec.`
`float`	32 (or 64)	`JVM-Spec` : 32, `C/C++ unspec.`
`double`	64 (or 128)	`JVM-Spec` : 64, `C/C++ unspec.`

C-unspecified: read *limits.h* for min and max specifications.

Types in Programming Languages

Traditional/old system (C-compiler) implementation details :

Type	*Implementation (bit)*
`short`	8
int	**16**
`long`	32
`long long`	64

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.

Types in Programming Languages

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

Types in Programming Languages

Structurally equivalent types:

```
struct A {hello: int};
struct B {hello: int};

struct A x = {hello: 1};
struct B y = {hello: 1};

  x = y?     -->  YES: structural typing (Subscript)
             -->  NO:  name typing (C)
```

Structural typing is sometimes called ***duck typing****.*

Types in Programming Languages

Some types typically checked at compile time (**static**):

- primitive types (boolean, integrals, floating points, chars),
- composite types (arrays, records, strings),
- reference types (pointers),
- abstract data types (class names),
- subtypes (class hierarchies),
- recursive types (linked lists),
- function types (binary functions),
- ...

Types in Programming Languages

Some types typically checked at runtime time (**dynamic**):

- object types (type variables),
- derived types,
- explicit type checks (reflection),
- ...

Types in Programming Languages

Static (compiler checked) type analysis:

- Type synthesis
- Type inference

Types in Programming Languages

Type synthesis (denotational formulation):

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

Types in Programming Languages

Type inference (denotational formulation):

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

Types in Programming Languages

How is the compiler inclined to react?

```
0. position: float;
1. initial: int;
2. rate: float;
3. position = initial + rate * 60.00;
```

Types in Programming Languages

What is most likely to happen here?

```
0. position: int;
1. initial: float;
2. rate: int;
3. position = initial + rate * 60;
```

Types in Programming Languages

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

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

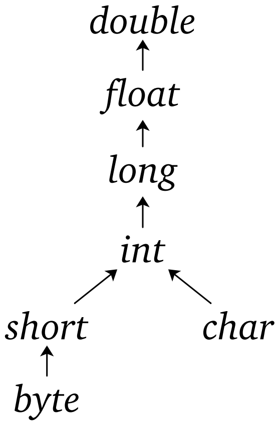
Types in Programming Languages

Type conversions:

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

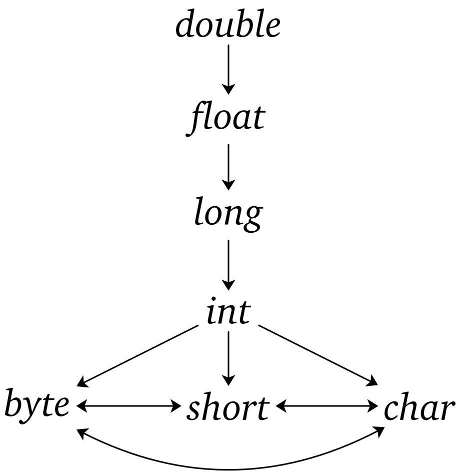
Types in Programming Languages

Widening conversion hierarchy (primitive Java types):



Types in Programming Languages

Narrowing conversion hierarchy (primitive Java types):



Types in Programming Languages

Other type conversions (Java):

```
"U" + 2                          becomes  "U2"
"Types are fun: " + true becomes  "Types are fun: true"
```

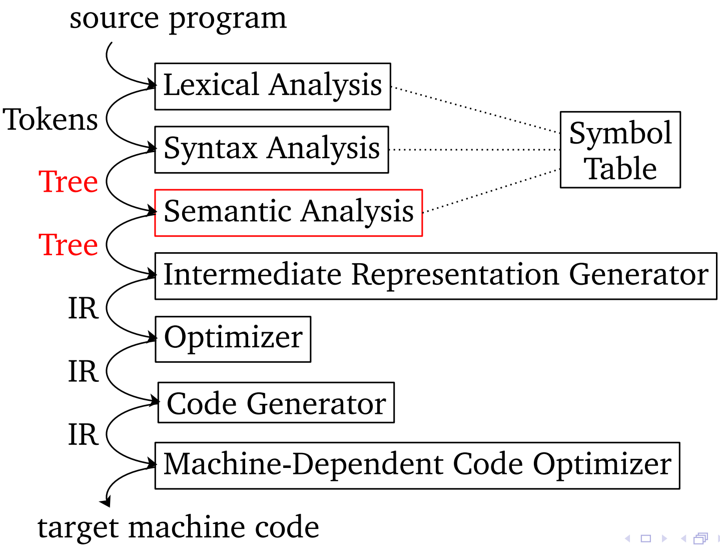
Types in Programming Languages

Operator and function *overloading*:

- '+' meaning addition or string concatenation (Java),
- user defined functions (Java).

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

Third compilation phase



Types in Compilers

SDD of array types with storage attribute (Fig. 6.15):

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

b-inherited; *t*,*width*-synthesized

Third compilation phase

Type expressions (TE): a way so assign structure to types...

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

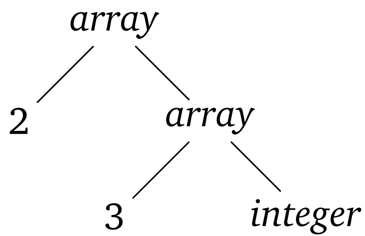
Third compilation phase

Type expression examples:

- short, int, long, float, char, ...
- T[], struct T, ...

Types in Compilers

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



Types in Compilers

SDD translate **T[num]...[num]** into type expressions (Fig.5.16):

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

b-inherited; *t*-synthesized

Types in Compilers

Three-address code gen (SDD) for expressions (Fig.6.20)

Production	Semantic Rules
$S \rightarrow \mathbf{id} = E$	$\{ gen(\text{top}.\text{get}(\mathbf{id}.lexeme)' =' E.addr); \}$
$E \rightarrow E_1 + E_2$	$\{ E.addr' =' E_1.addr' +' E_2.addr); \}$
$\mid - E$	$\{ E.addr = \text{new}\, Temp(); \}$ $gen(E.addr' =' \ '\mathbf{minus}' E_1.addr); \}$
$\mid (E_1)$	$\{ E.addr = E_1.addr; \}$
$\mid \mathbf{id}$	$\{ E.addr = \text{top}.\text{get}(\mathbf{id}.lexeme)$

Types in Compilers

... with type conversion added (Fig.6.27):

PRODUCTION	SEMANTIC RULES
$E \to E_1 + E_2$	$\{\ E.type = \mathbf{max}(E_1.type, E_2.type);$ $a_1 = \mathbf{widen}(E_1.addr, E_1.type, E.type);$ $a_2 = \mathbf{widen}(E_2.addr, E_2.type, E.type);$ $E.addr = \text{new } Temp();$ $gen(E.addr\ \text{'='}\ a_1\ \text{'+'}\ a_2);\ \}$

semantic attribute: $E.type$; semantic action: $E \to E_1 + E_2$

Types in Compilers

Pseudo code for *widen* (*integer* and *float*):

```
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;
```

Types in Subscript

What is the type system in Subscript?

Types in Subscript

A *SubScript* type (*Type*) has one of he following forms:

```
any
boolean
number
string
void
Type [ ]
Identifier
(Identifier:Type,...,Identifier:Type) => Type
```

Types in Subscript

The *SubScript* type system at present:

- basic primitive types (booleans, numbers, strings, ...),
- basic composite types (arrays),
- abstract data type (classes),
- *no* casts, *no* coercions, (almost) *no* overloading,
- no sybtypes (i.e., no 'extends' syntax),
- *yes* structural typing.

Types in HACS

What does type analysis look like in HACS?