Type Analysis

Eva Rose  Kristoffer H. Rose

NYU Courant Institute

March 10, 2014
Outline

1. Types in Programming Languages
2. Types in Compilers
3. Types in Subscript
4. Types in HACS
What is a type?
Some examples of types:

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

- Error detection (static and dynamic checks),
- Disambiguate operations (error prevention).
- Optimization purposes (storage layout).
Type implementation/storage requirements according to:

- language spec,
- compiler spec,
- machine architecture spec.
Some (modern) language implementation details:

<table>
<thead>
<tr>
<th>Type</th>
<th>Implementation (bit)</th>
<th>language specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>JVM-Spec : 8</td>
</tr>
<tr>
<td>short</td>
<td>16 (or 32)</td>
<td>JVM-Spec : 16, C/C++ unspec.</td>
</tr>
<tr>
<td>char</td>
<td>8 or 16</td>
<td>JVM-Spec : 16, C/C++:8</td>
</tr>
<tr>
<td>int</td>
<td>32 or 64</td>
<td>JVM-Spec : 32, C/C++ unspec.</td>
</tr>
<tr>
<td>long</td>
<td>64 (or 128)</td>
<td>JVM-Spec : 64, C/C++ unspec.</td>
</tr>
<tr>
<td>float</td>
<td>32 (or 64)</td>
<td>JVM-Spec : 32, C/C++ unspec.</td>
</tr>
<tr>
<td>double</td>
<td>64 (or 128)</td>
<td>JVM-Spec : 64, C/C++ unspec.</td>
</tr>
</tbody>
</table>

C-unspecified: read limits.h for min and max specifications.
Traditional/old system (C-compiler) implementation details:

<table>
<thead>
<tr>
<th>Type</th>
<th>Implementation (bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>16</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Approach</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denotational</td>
<td>set of values (<em>Type Theory</em>)</td>
</tr>
<tr>
<td>Abstract</td>
<td>set of operations (<em>OO-systems</em>)</td>
</tr>
<tr>
<td>Named typing</td>
<td>set of names</td>
</tr>
<tr>
<td>Structural typing</td>
<td>set of structures</td>
</tr>
</tbody>
</table>
**Types in Programming Languages**

*Structurally equivalent types:*

```plaintext
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*. 
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),
- ...

CSCI-GA.2130-001/Spring 2014

Eva Rose & Kris Rose
Some types typically checked at runtime time (dynamic):

- object types (type variables),
- derived types,
- explicit type checks (reflection),
- ...
Static (compiler checked) type analysis:

- Type synthesis
- Type inference
Type synthesis (denotational formulation):

\[
f : S \rightarrow T \quad x : S
\]

\[
f(x) : T
\]
Type inference (denotational formulation):

\[
\exists S \quad f : S \rightarrow T \quad x : S
\]

\[
f(x) : T
\]
How is the compiler inclined to react?

0. `position: float;`
1. `initial: int;`
2. `rate: float;`
3. `position = initial + rate * 60.00;`
What is most likely to happen here?

0. position: int;
1. initial: float;
2. rate: int;
3. position = initial + rate * 60;
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).
Widening conversion hierarchy (primitive Java types):

```
  double
     ↑
    float
     ↑
   long
     ↑
  int
     ↘
  short
     ↗
  byte
```

(char)
Narrowing conversion hierarchy (primitive Java types):

\[
\begin{align*}
  & \text{double} \\
  & \downarrow \\
  & \text{float} \\
  & \downarrow \\
  & \text{long} \\
  & \downarrow \\
  & \text{int} \\
  & \downarrow \\
  & \text{byte} \quad \text{short} \quad \text{char}
\end{align*}
\]
Other type conversions (Java):

"U" + 2 becomes "U2"
"Types are fun: " + true becomes "Types are fun: true"
Operator and function *overloading*:

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

```java
void err() {...}
void err(String s) {...}
void err(Integer x) {...}
```
Third compilation phase

source program

Tokens

Lexical Analysis

Syntax Analysis

Symbol Table

Tree

Semantic Analysis

Intermediate Representation Generator

Tree

Optimizer

IR

Code Generator

IR

Machine-Dependent Code Optimizer

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

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T \rightarrow BC )</td>
<td>( { T.t = C.t; T.width = C.width; } ) ( { C.b = B.t; w = B.width; } )</td>
</tr>
<tr>
<td>( B \rightarrow \text{int} )</td>
<td>( { B.t = \text{integer}; B.width = 4; } )</td>
</tr>
<tr>
<td>( B \rightarrow \text{float} )</td>
<td>( { B.t = \text{float}; B.width = 8; } )</td>
</tr>
<tr>
<td>( C \rightarrow [\text{num}]C_1 )</td>
<td>( { C.t = \text{array}(\text{num}.val, C_1.t); ) ( C.width = \text{num}.val \times C_1.width; } )</td>
</tr>
<tr>
<td>( C \rightarrow \epsilon )</td>
<td>( { C.t = C.b; C.width = w; } )</td>
</tr>
</tbody>
</table>

\( b \)-inherited; \( t, width \)-synthesized
Type expressions (TE): a way to 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, ...
Type structure: type expression for \( \text{int}[2][3] \)

```
array
  /
2
```

array
```
  /
3
integer
```
SDD translate $T[n\text{um}]...[n\text{um}]$ into type expressions (Fig.5.16):

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \rightarrow BC$</td>
<td>$T.t = C.t$</td>
</tr>
<tr>
<td></td>
<td>$C.b = B.t$</td>
</tr>
<tr>
<td>$B \rightarrow \text{int}$</td>
<td>$B.t = \text{integer}$</td>
</tr>
<tr>
<td>$B \rightarrow \text{float}$</td>
<td>$B.t = \text{float}$</td>
</tr>
<tr>
<td>$C \rightarrow [n\text{um}]C_1$</td>
<td>$C.t = \text{array}(\text{n\text{um}.val}, C_1.t)$</td>
</tr>
<tr>
<td></td>
<td>$C_1.b = C.b$</td>
</tr>
<tr>
<td>$C \rightarrow \epsilon$</td>
<td>$C.t = C.b$</td>
</tr>
</tbody>
</table>

$b$-inherited; $t$-synthesized
Three-address code gen (SDD) for expressions (Fig.6.20)

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E$</td>
<td>${ \text{gen}(\text{top} . \text{get} (\text{id}.\text{lexeme})' = E.\text{addr}); }$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>${ E.\text{addr}' = E_1.\text{addr}' + E_2.\text{addr}); }$</td>
</tr>
<tr>
<td>$</td>
<td>- E$</td>
</tr>
<tr>
<td>$</td>
<td>(E_1)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{id}$</td>
</tr>
</tbody>
</table>
... with type conversion added (Fig.6.27):

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| $E \rightarrow E_1 + E_2$ | { $E.type = \text{max}(E_1.type, E_2.type)$;  
+ $a_1 = \text{widen}(E_1.addr, E_1.type, E.type)$;  
+ $a_2 = \text{widen}(E_2.addr, E_2.type, E.type)$;  
+ $E.addr = \text{new Temp}()$;  
+ $\text{gen}(E.addr \ ‘=’ \ a_1 ‘+’ a_2) ;$ } |

semantic attribute: $E.type$; semantic action: $E \rightarrow E_1 + E_2$
## 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;
```
What is the type system in Subscript?
Types in Subscript

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

- `any`
- `boolean`
- `number`
- `string`
- `void`
- `Type [ ] Identifier (Identifier:Type,...,Identifier:Type) => Type`
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 subtypes (i.e., no 'extends' syntax),
- yes structural typing.
What does type analysis look like in HACS?