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
1. Introduction
2. Type Systems
3. Types in Programming Languages
4. Types in the Compiler
5. Type Checking
From Abstract Syntax Tree (AST)

\[ \text{position} = \text{initial} + \text{rate} \times 60 \]
Type Checking Gives Annotated AST…

\[
\begin{align*}
t &= \text{float} \\
\langle \text{id}, 1 \rangle &+ t = \text{float} \\
\langle \text{id}, 2 \rangle &\times t = \text{float} \\
\langle \text{id}, 3 \rangle &\times \langle \text{num}, 60 \rangle \\
\text{position} &\rightarrow \text{float} \\
\text{initial} &\rightarrow \text{float} \\
\text{rate} &\rightarrow \text{float}
\end{align*}
\]
Introduction

Type Systems

Types in Programming Languages

Types in the Compiler

Type Checking
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)

\[
\begin{align*}
\text{if } f : S \to T \text{ and } x : S \text{ then } f(x) : T
\end{align*}
\]
**Type Synthesis (Denotational Formulation)**

\[
\text{if } f : S \rightarrow T \text{ and } x : S \text{ then } f(x) : T
\]

\[
\begin{array}{c}
\frac{f : S \rightarrow T \quad x : S}{f(x) : T}
\end{array}
\]
Type Inference (Denotational Formulation)

\[ f(x) : T \]

\[ \exists S [ f : S \rightarrow T \quad x : S ] \]
1. Introduction

2. Type Systems

3. Types in Programming Languages

4. Types in the Compiler

5. Type Checking
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Implementation (bit)</th>
<th>Language Specifications</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++ ≥ 16</td>
</tr>
<tr>
<td>long</td>
<td>64 (or 128)</td>
<td>JVM-Spec: 64, C/C++ ≥ 32</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: part of system – read *limits.h* for min and max specifications.
Traditional/Old System (C-compiler) Type 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>
</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 (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

```c
1 float position;
2 int initial;
3 float rate;
4 ...
5 position = initial + rate * 60.00;
```
Example Type Conversion II

```java
float position;
int initial;
float rate;

position = initial + rate * 60;
```
Example Type Conversion III

```c
int position;
int initial;
float rate;
...
position = initial + rate * 60;
```
Explicit Conversions (cast)

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

```
1 ... 
2 ... 
3 ... 
4 position = initial + ((int) rate) * 60;
```
Example Java Conversions...

```
1 "U" + 2  
2 2 + "U"  
3 "friends are " + true 
4 true + " types"
```
Example Java Conversions…

1. "U" + 2  \textit{OK}
2. 2 + "U"
3. "friends are " + true
4. true + " types"
Example Java Conversions…

1. "U" + 2  OK
2. 2 + "U"  Fail
3. "friends are " + true
4. true + " types"
Example Java Conversions...
Example Java Conversions…

1. "U" + 2   \textit{OK}
2. 2 + "U"   \textit{Fail}
3. "friends are " + true   \textit{OK}
4. true + " types"   \textit{Fail}
Operator and Function Overloading

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

```java
1 void err() {...}
2 void err(String s) {...}
```
Operator and Function Overloading

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

1. `void err() {...}
2. `void err(String s) {...}`
Type Equivalence Approaches

What are the values “in” a type?

<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 Data Types (ADT)</td>
<td>set of operations (<em>OO-systems</em>)</td>
</tr>
</tbody>
</table>

When are two types considered the same?

<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 Data Types (ADT)</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 (<em>duck typing</em>)</td>
</tr>
</tbody>
</table>
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⟨S,T⟩
- S × T
- S → T
Type structure: AST for $\text{int}[2][3]$

\[
T \rightarrow B \ C \\
B \rightarrow \text{int} \mid \text{float} \\
C \rightarrow [\text{num}]C \mid \epsilon
\]
Type structure: AST for \textit{int}\{2\}\{3\}

\[
T \rightarrow B \ C \\
B \rightarrow \text{int} \mid \text{float} \\
C \rightarrow \text{[num]}C \mid \epsilon
\]

(1)  
(2)  
(3)
Type structure: type expression for \texttt{int}[2][3]
# Example: SDD for array TEs

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

$b$-inherited; $t$-synthesized
**Example: SDD with Type Storage Attribute**

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

$b$ inherited; $t, width$ synthesized
### Example: SDT for Translation with Types

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E$</td>
<td>${ \text{gen(top.get(id.lexeme)'} = 'E.addr'); }$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>${ E.addr' = 'E_1.addr' + 'E_2.addr'; }$</td>
</tr>
<tr>
<td>$\mid - E$</td>
<td>${ E.addr = \text{new Temp}(); }$</td>
</tr>
<tr>
<td>$\mid (E_1)$</td>
<td>${ E.addr = E_1.addr; }$</td>
</tr>
<tr>
<td>$\mid \text{id}$</td>
<td>${ E.addr = \text{top.get(id.lexeme)} }$</td>
</tr>
</tbody>
</table>
Example: SDT for Translation with Types and Type Conversion

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

<table>
<thead>
<tr>
<th></th>
<th>Introduction</th>
<th>Type Systems</th>
<th>Types in Programming Languages</th>
<th>Types in the Compiler</th>
<th>Type Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Type Systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Types in Programming Languages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Types in the Compiler</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Type Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Type Synthesis SDD

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.t = \text{Unif}(E_1.t, E_2.t)$ (1)</td>
</tr>
<tr>
<td>$E_1 * E_2$</td>
<td>$E.t = \text{Unif}(E_1.t, E_2.t)$ (2)</td>
</tr>
<tr>
<td>int</td>
<td>$E.t = \text{Int}$ (3)</td>
</tr>
<tr>
<td>float</td>
<td>$E.t = \text{Float}$ (4)</td>
</tr>
</tbody>
</table>
Example: Type Synthesis HACS Sorts

\begin{verbatim}
sort Type | Int | Float;

sort Type | scheme Unif(Type,Type);
Unif(Int, Int) \rightarrow Int;
Unif(Float, #) \rightarrow Float;
Unif(#, Float) \rightarrow Float;
\end{verbatim}
Questions?

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