Type Checking

Chapter 6, Section 6.3, 6.5
Inside the Compiler: Front End

• Lexical analyzer (aka scanner)
  – Converts ASCII or Unicode to a stream of tokens

• Syntax analyzer (aka parser)
  – Creates a parse tree from the token stream

• Semantic analyzer
  – Type checking and conversions; other semantic checks

• Generator of intermediate code
  – Creates lower-level intermediate representation (IR): e.g., three-address code
Types in Compilers

• **Type checking**: at compile time, guarantee that the run-time behavior of the program will be correct
  – The type of the **operands** match the type of the **operator** (e.g., in Java && requires boolean operands)
  – The types of actual parameters in a function call match the types of the formal parameters
  – Many other examples based on the **type system** of the language

• **Code generation**
  – **Allocation of memory** based on types (e.g., how many bytes do we need for a struct with an int and a float?)
  – Insert **explicit type conversions**
Outline

- **Analysis of declarations**
  - Representation of types
  - Memory allocation based on types

- **Type checking**
  - What is the type of an expression, given the types of its subexpressions? (synthesized attributes)
  - Is there a type error in the program?

- **Implicit type conversions**: not in the source code, but must be accounted for during type checking and code generation
  - E.g., for `float x, z; int y; z = x+y;` we need to generate `t=(float)y; z = x+t;` three-address instructions
Type Expressions

• What is a type and how do we represent it inside a compiler? We will use type expressions for this

• A basic type is a type expression (e.g., boolean, char, byte, integer, float, void)

• An array type constructor, applied to a number and a type expression, is a type expression
  – E.g., array(10,integer) or array(1024,array(24,char))

• A record type constructor, applied to a list of pairs (field name, type expression), is a type expression
  – E.g., record { x:float, y:float, rgb:array(3,byte) } could be the type expression for a C struct with fields x,y for point coordinates and field rgb for RGB point color
Type Expressions

- An **function** type constructor $\rightarrow$, applied to two type expressions, is a type expression
  - E.g., $\text{array}(10,\text{float}) \rightarrow \text{float}$ for a function that computes the sum of all array elements and returns it

- A **tuple** type constructor $\times$, applied to a list of type expressions
  - E.g., $\text{record} \{ x:\text{float}, y:\text{float}, rgb:\text{array}(3,\text{byte}) \} \times \text{float} \rightarrow \text{record} \{ x:\text{float}, y:\text{float}, rgb:\text{array}(3,\text{byte}) \}$ is a function taking two parameters: a record and a float

- Type expressions can naturally be represented with trees or DAGs (similarly to ASTs/DAGs for expressions appearing in the source code; see Fig. 6.14 in book)
Simplified Grammar for Declarations

\[ D \rightarrow T \; id \; ; \; D_1 \]
\[ addType(id \cdot symtbl\_entry, T \cdot type) \]
\[ \mid \varepsilon \]

\[ T \rightarrow B \; C \]
\[ C \cdot base = B \cdot type \]
\[ T \cdot type = C \cdot type \]

\[ B \rightarrow \text{int} \]
\[ B \cdot type = \text{integer} \]
\[ \mid \text{float} \]
\[ B \cdot type = \text{float} \]
\[ \mid \text{bool} \]
\[ B \cdot type = \text{boolean} \]

\[ C \rightarrow \left[ \text{const} \right] \; C_1 \]
\[ C_1 \cdot base = C \cdot base \]
\[ C \cdot type = \text{array} \left( \text{const} \cdot \text{lexval}, C_1 \cdot type \right) \]
\[ \mid \varepsilon \]
\[ C \cdot type = C \cdot base \]

- Assume that lexical analysis and parsing have populated the symbol table will all names \text{id}
- Helper function \text{addType} should ensure that only one type is declared for a particular \text{id}
- Things get more complicated when we have multiple scopes
How Much Memory?

\[
D \rightarrow T \text{ id } ; D_1 \\
| \varepsilon \\
T \rightarrow B \ C \\
B \rightarrow \text{ int } \\
| \text{ float } \\
| \text{ bool } \\
C \rightarrow [ \text{ const } ] \ C_1 \\
| \varepsilon
\]

\[
addWidth(\text{id.symtbl_entry}, T.width) \\
C.basew = B.width \\
T.width = C.width \\
B.width = 4 \\
B.width = 8 \\
B.width = 1 \\
C_1.basew = C.basew \\
C.width = \text{ const.lexval } * C_1.width \\
C.width = C.basew
\]

Note: there may be \textbf{hardware alignment} constraints – e.g., each integer must start at an address divisible by 4; for \textit{record \{ integer, boolean, integer \}} \textbf{padding} may be needed; we ignore such issues here
Type Checking

• Now, look at expressions/statements to see if declared types are consistent with variable usage

• Many checks of the form `if (type expression 1 == type expression 2) OK otherwise report type error`
  – When are two type expressions equivalent?
  – Structural equivalence: the underlying type expression DAGs are the same
  – Other options: e.g., name equivalence

• Checking: (1) types of subexpressions OK? (2) decide the type of the whole expression
S → E ;
E → id | int_const | float_const | true | false
E → E₁ + E₂ | E₁ < E₂ | E₁ || E₂ | ...
E → - E₁ | ! E₁ | ...
E → id [E₁]
E → (E₁)
E → E₁ = E₂ | ...
SDD for Type Checking

- \( S \rightarrow E ; \)
  - \( S.type = \text{void} \)

- \( E \rightarrow E_1 + E_2 \)
  - If \( E_1.type \) is not integer or float) type error
  - If \( E_2.type \) is not integer or float) type error
  - If \( E_1.type \) is not equivalent to \( E_2.type \) type error
  - \( E.type = E_1.type \)

- \( E \rightarrow E_1 < E_2 \)
  - If \( E_1.type \) is not integer or float) type error
  - If \( E_2.type \) is not integer or float) type error
  - If \( E_1.type \) is not equivalent to \( E_2.type \) type error
  - \( E.type = \text{boolean} \)

- \( E \rightarrow E_1 \mid E_2 \)
  - If \( E_1.type \) is not boolean) type error
  - If \( E_2.type \) is not boolean) type error
  - \( E.type = \text{boolean} \)
SDD for Type Checking

- $E \rightarrow - E_1$
  - If ($E_1.type$ is not integer or float) type error
  - $E.type = E_1.type$

- $E \rightarrow ! E_2$
  - If ($E_1.type$ is not boolean) type error
  - $E.type = boolean$

- $E \rightarrow id$
  - $E.type = \text{getType}(\text{id.symtbl\_entry})$

- $E \rightarrow \text{int\_const}$
  - $E.type = \text{integer}$

- $E \rightarrow \text{float\_const}$
  - $E.type = \text{float}$

- $E \rightarrow true$
  - $E.type = boolean$
SDD for Type Checking

• $E \rightarrow \text{id} \ [ \ E_1 \ ]$
  – If $(\text{getType}(\text{id}.\text{symtbl\_entry})$ is not $\text{array}(X,Y))$ type error
  – If $(E_1.\text{type}$ is not $\text{integer})$ type error
  – $E.\text{type} = Y$

• $E \rightarrow (E_1)$
  – $E.\text{type} = E_1.\text{type}$

• $E \rightarrow E_1 = E_2$
  – If $(E_1.\text{type}$ is not equivalent to $E_2.\text{type})$ type error
  – $E.\text{type} = E_1.\text{type}$

Also need to check that the left-hand-side of an assignment operator has an l-value. Add a synthesized attribute \textit{lvalue} which is true for \textit{id} and \textit{id} \ [ \ E_1 \ ] and is false for all other expressions. In the rule for $E_1 = E_2$ check that $E_1.\text{lvalue}$ is true
Implicit Type Conversions

• Values of one type are silently converted to another type
  – e.g. addition: 3.0 + 4 : converts 4 to 4.0
  • Our earlier typechecking rules imply that operator + has types \texttt{int \times int \rightarrow int} and \texttt{float \times float \rightarrow float}

• In a context where the type of an expression is not appropriate
  – either an automatic conversion to another type is performed automatically
  – or if not possible: compile-time error

• Implications for code generation:
  – E.g., for \texttt{float x, z; int y; z = x+y;} need to generate \texttt{t=(float)y; z = x+t;} three-address instructions
Example: Conversions in Java

- Widening: converting a value into a “larger” type

- Widening primitive conversions in Java
  - byte to short, int, long, float, or double
  - short to int, long, float, or double
  - char to int, long, float, or double
  - int to long, float, or double
  - long to float or double
  - float to double

- “integral type to integral type” and “float to double” do not lose any information
Example: Conversions in Java

• Language Spec says
  – Conversion of an *int* or *long* value to *float*, or of a *long* value to *double*, may result in loss of precision
  – The result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode
Example: Conversions in Java

• **Assignment conversion**: when the value of an expression is assigned to a variable, convert the value to the type of the variable

• **Method invocation conversion**: applied to each argument value in a method or constructor invocation
  – The type of the argument expression must be converted to the type of the corresponding formal parameter

• **Typecasting conversion**: applied to the operand of a *typecast operator*: (float) 5
  – Note: this is *explicit* rather than *implicit*: the programmer writes code to request the conversion
Example: Conversions in Java

• **Numeric promotion**: converts operands of a numeric operator to a common type

• Example: binary numeric promotion
  – e.g. +, -, *, etc.
  – If either operand is *double*, the other is converted to *double*
  – Otherwise, if either operand is of type *float*, the other is converted to *float*
  – Otherwise, if either operand is of type *long*, the other is converted to *long*
  – Otherwise, both are converted to type *int*

Note: there is a type lattice; *double* is the top element; binary numeric promotion is the least upper bound of the operand types (Fig. 6.25)
Example: Conversions in Java

• Narrowing: converting a value into a “smaller” type

• Narrowing primitive conversions in Java
  – e.g. long to byte, short, char, or int
  – float to byte, short, char, int, or long
  – double to byte, short, char, int, long, or float
  – More cases in the Language Spec

• Examples of loss of information
  – int to short loses high bits
  – int not fitting in byte changes sign and magnitude
  – double too small for float underflows to zero
Back to Our Simplified Language

• Let’s allow implicit widening conversions from integer to float. What will get affected?

• Old rule for $E \rightarrow E_1 + E_2$
  – If $(E_1.type$ is not integer or float) type error
  – If $(E_2.type$ is not integer or float) type error
  – If $(E_1.type$ is not equivalent to $E_2.type$) type error
  – $E.type = E_1.type$

• New rule
  – First two checks are the same
  – $E.type = E_1.type$, if $E_2.type$ is integer
  – $E.type = float$, otherwise

• Same for $E \rightarrow E_1 < E_2$
Modified Code Generation

• Old rule for $E \rightarrow E_1 + E_2$
  
  $E$.addr = newTemp() and $E$.code = $E_1$.code || $E_2$.code ||
  
  $E$.addr = "$E_1$.addr + "$ $E_2$.addr

• New rule
  
  $E$.addr = newTemp() and $E$.code = $E_1$.code || $E_2$.code ||
  
  $E$.addr = "$widen(E_1.addr, E_1.type, E.type) + "$
  
  widen($E_2.addr, E_2.type, E.type$)

• Helper function widen(addr, $t$, $w$)
  
  – If $t$=$w$ return addr
  
  – Create floating-point temp, create and append new
    typecast instruction temp = (float) addr, return temp

float x, z; int y; x = y+z; becomes $t1=$(float)$y; $t2=t1+z; x=t2;
How About Assignments?

- Old rule for $E \rightarrow E_1 = E_2$
  - If ($E_1.type$ is not equivalent to $E_2.type$) type error
  - $E.type = E_1.type$

- New rule (assignment conversion, as in Java: right-hand-side value will be converted to the type of the left-hand side expression)
  - $E.type = E_1.type$

- Code generation for $E \rightarrow E_1 = E_2$
  - Old: $E.code = \ldots "$ = " $E_2.addr$ and $E.addr = E_2.addr$
  - New: $E.code = \ldots "$ = " widen($E_2.addr$, $E_2.type$, $E_1.type$) and $E.addr = \text{address returned by widen}$

```c
float x; int y, z; x = y+z; becomes t1=y+z; t2=(float)t1; x=t2;
```