Type Checking

CS2210
Lecture 8

Review:
Syntax-directed Translation &
Type checking

Background Reading
- Chapter 5: read 5.1 - 5.4
- Rest of chapter is optional -- won't be tested in exam
- Chapter 6: complete
Syntax-directed Definition

- Add attribute(s) to grammar symbols
  - E.g., type, value, memory location
- Semantic rules
  - Compute attributes from rhs of production (synthesized attribute)
  - Pass attribute from lhs to rhs (inherited attribute)
- Intrinsic attributes (assigned to leaves)
  - Sometimes also called synthesized (but from "external info")
- Decorating/annotating the parse tree = computing attributes for all parse tree nodes

Syntax-directed definitions and Grammars

- Attribute Grammar = syntax-directed definition w/o side-effects
- S-attributed definition
  - A syntax-directed definition where all attributes are synthesized

Example Production Semantic Rules
L -> E n      print(E.val)
L->E_1 + T    E.val = E_1.val+T.val
E->T          E.val = T.val
T->T_1*F      T.val = T_1.val * F.val
T->F          T.val = F.val
F->(E)        F.val = E.val
F->digit      F.val = digit.lexval
Inherited Attributes

- Value from a parent or sibling in parse tree
- Used to express dependences of PL constructs
- E.g. types, lvalue vs. rvalue

Example: Inherited Attributes

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D -&gt; T L</td>
<td>L.in := T.type</td>
</tr>
<tr>
<td>T -&gt; int</td>
<td>T.type := integer</td>
</tr>
<tr>
<td>T -&gt; real</td>
<td>T.type := real</td>
</tr>
<tr>
<td>L -&gt; L_i, id</td>
<td>L_i.in := L.in addtype(id.entry, L.in)</td>
</tr>
<tr>
<td>L -&gt; id</td>
<td>addtype(id.entry, L.in)</td>
</tr>
</tbody>
</table>

Dependency Graphs

- Semantic rules produce dependencies, e.g., b := f(c_1, ..., c_n) can compute b only after computing all c_i
- Graphical representation
  - Directed graph with an edge from c_i to b
- Evaluation order
  - Topological sort
  - Rule-based (fixed at compile time)
  - Oblivious (does not work for all syntax-directed definitions)
L-attributed Definitions

- \( A \rightarrow X_1 X_2 \ldots X_n \)
  - Attribute of \( X_i \) depends only on inherited attributes of \( A \) and on \( X_{1\ldots i-1} \)
  - Can be evaluated by a left-to-right (recursive) traversal

Semantic Analysis

- Type checks
  - Statically or dynamically
- Control-flow checks
  - E.g., no goto into the middle of a (different procedure)
- Uniqueness checks
  - E.g., redefinition of variables
- Name (matching) checks
  - Procedure name (e.g., Modula 2)

Type checking

- Check
  - That operands have expected types
  - Expressions are well-typed
- What is a type?
  - "A Collection of computational entities that share some common property" (Mitchell)
  - Determines the range of values and set of operations that are defined for values of the type
Type Systems

- Rules for assigning types to programming language constructs
- "A type system is a syntactic method for enforcing levels of abstraction in programs." (Benjamin C. Pierce)

Why Types?

- Program organization and documentation
- Consistent interpretation of bits in memory (avoid unintended semantics)
- Optimization
  - Compiler can exploit type information for better performance, example:
    - type { = record name: array[1..256] of char;
           year: integer;
        end;
    - Can generate efficient code to access record fields:
      address(year) = base_address(s) + 256 * sizeof(char)

Basic & Composite Types

- Basic types
  - Depend on PL, usually integers, reals, characters, booleans etc.
- Type constructors
  - Used to build new types from existing ones
  - Common constructors:
    - Array types
    - Product types (unnamed tuples)
- Record types (named tuples)
  - E.g.:
    - type row = record
      - address: integer;
      - lexeme: array[1..15] of char;
    - end;
    - record((address X integer)
        X (lexeme X array(1..15, char)))
  - Pointers
**Type Expression Definition**

- A type expression is one of
  - Basic type
  - Type name
  - Type constructor
  - Type variable

---

**Type System revisited**

- Set of rules to assign types
  - Implemented by a type checker
- Type checking can be done
  - Statically (= at compile time)
    - Modula 2, C, Java -> statically typed language
    - In practice have to do some checks at run-time as well, e.g. array bounds checking a[i] is i within bounds?
  - Dynamically (when the program is running)
    - Lisp -> dynamically typed language

---

**Strong vs. Weak**

- Strongly typed language
  - A program that passes the type checker will not have a run-time (type) error
  - Example: Java
- Weakly typed language
  - Can have type errors
    - C, C++
Declared vs. Inferred Types

- Most programming languages require the programmer to declare types
  - E.g., int x, y, z; int foo(int x)...
- Some languages infer all types based on inference rules
  - E.g., ML (statically typed but don’t have to declare types)

Example: TC Expressions

E -> literal {E.type := char}
E -> num {E.type := integer}
E -> id {E.type := lookup(id.entry)}
E -> E_1 [E_2] {E.type := if E_2.type = integer and E_1.type = array(s, t) then t else type_error}
E -> E_1 ^ {if E_1.type = pointer(t) then t else type_error}

Type Compatibility

- What are the types that can legally be used in a particular context?
  - E.g. 2.5 + x, what type should x be?
- Type conversion ("cast")
  - 2.5 + ((float) x)
  - Widening (int to float) versus narrowing (float to int)
- Type coercion
  - Implicit conversion dictated by language rules
Structural Equivalence

- Two types are compatible if either of the same basic type or built by same constructor applied to structurally equivalent types

Name Equivalence

- Named types
  - Types equivalent only when they have the same name
  - Unnamed types get fresh names associated with their declaration

Comparison

- Structural equivalence
  - More flexible
  - More expensive to check
- Name equivalence
  - Efficiently checkable
  - More restrictive
Polymorphism
- Parametric polymorphism
  - Function can be applied to any arguments whose type match a type expression with type variables
- Ad hoc polymorphism
  - (aka overloading) two or more implementations with different types referred to by same name (e.g., `+` can be integer and floating point addition)
- Subtype polymorphism
  - Subtype relation between types allows an expression to have many possible types
  - Common in oo-languages, e.g., COOL

Parametric Polymorphism
- Characteristic: type variables
  - Often found in functional languages, e.g. ML: sort : (`a * `a -> bool) * `a list -` `a list
  - Function may have infinitely many types
- Implicit or explicit
  - ML implicit: programmer does not have to declare types
    - Type inference computes necessary types
  - Explicit: program text contains type variables, e.g., C++ templates

C++ Templates
Template `<T>
void swap (T& x, T& y) {
  T tmp = x; x = y; y = tmp;
}
Implementing Parametric Polymorphism

- C++ (explicit)
  - Templates are instantiated at link time (why?)
- Different copies of polymorphic function (e.g., swap) are created
  - Why?
- One of the major reasons for the complexity of C++ linkers

Implementing Parametric Polymorphism (2)

- ML (implicit)
- No need for multiple function copies
  - Uniform data representation for all types: “boxed”
  - Advantage:
    - No code duplication
  - Disadvantage
    - Performance degradation - can be mitigated by “unboxing optimization”

Typechecking for COOL & Type System Operations
Static vs. Static Typing

- Static typing
  - More efficient (fewer run-time checks)
  - Detect errors earlier
  - May disallow some programs that in fact are fine
- Dynamic typing
  - More flexible
  - Run-time cost
  - Errors detected only at run time

COOL Types

- Class types
  - Every class name is also a type
  - Includes Int, String, Bool
- SELF_TYPE
- User declares types for identifiers
  - Compiler (type checker) infers types for all expressions

Dynamic versus Static Type

- The dynamic type of an object is the class C that is used in the “new C” expression that created it
  - A run-time notion
- The static type of an expression captures all possible dynamic types the expression could take
  - A compile-time notion
Type Systems where Static Type = Dynamic Type

- Soundness theorem: for all expressions $E$
  $$\text{dynamic_type}(E) = \text{static_type}(E)$$
  (in all executions, $E$ evaluates to values of the type inferred by the compiler)

- No additional checking required at run-time
  - Other than dynamic properties, e.g., array bounds

---

Dynamic and Static Types in COOL

class A {
...
};
class B inherits A {
...
};
class Main {
    A x ← new A;
    ...
    x ← new B;
    ...
}

$X$ has static type $A$

- $X$'s value has dynamic type $A$

$X$ has static type $B$

- $X$'s value has dynamic type $B$

- A variable of static type $A$ can hold values of static type $B$, if $B \leq A$

---

Dynamic and Static Types

Soundness theorem for the Cool type system:

$$\forall E. \text{dynamic_type}(E) \leq \text{static_type}(E)$$

- In COOL, $\leq$ is defined on classes i.e., subtype $\Rightarrow$ subclass
  - $X \leq X$
  - $X \leq Y$ if $X$ inherits from $Y$
  - $X \leq Z$ if $Y \leq Z$

  Works because all operations that can be used on an object of type $C$ can also be used on an object of type $C \leq C$
  - Such as fetching the value of an attribute
  - Or invoking a method on the object

- Subclasses can only add attributes or methods
- Methods can be redefined but with same type!
**An Example**

class Count {
  i : int ← 0;
  inc () : Count {
    i ← i + 1;
    self;
  }
};

- Class `Count` incorporates a counter
- The `inc` method works for any subclass
- But there is disaster lurking in the type system

**An Example (Cont.)**

- Consider a subclass `Stock` of `Count`
  class Stock inherits Count {
    name : String; -- name of item
  }
  And the following use of `Stock`:

  class Main {
    Stock a ← (new Stock).inc ();  // Type checking error!
    ... a.name ...
  };

**What Went Wrong?**

- `(new Stock).inc()` has dynamic type `Stock`
- So it is legitimate to write
  
  ```java
  Stock a ← (new Stock).inc ()
  ```

- But this is not well-typed
  - `(new Stock).inc()` has static type `Count`
  - The type checker "looses" type information
  - This makes inheriting `inc` useless
  
  So, we must redefine `inc` for each of the subclasses, with a specialized return type
SELF_TYPE to the Rescue

- We will extend the type system
- Insight:
  - Inc returns "self"
  - Therefore the return value has same type as "self"
  - Which could be Count or any subtype of Count!
- Introduce the keyword SELF_TYPE to use for the return value of such functions
- We will also need to modify the typing rules to handle SELF_TYPE

(Cont.)

- SELF_TYPE allows the return type of inc to change when inc is inherited
- Modify the declaration of inc to read
  \[
  \text{inc()} : \text{SELF_TYPE} \{ \ldots \}
  \]
- The type checker can now prove:
  \[
  (\text{new Count}).\text{inc()} : \text{Count}
  \]
  \[
  (\text{new Stock}).\text{inc()} : \text{Stock}
  \]
- The program is now well typed

Notes About SELF_TYPE

- SELF_TYPE is not a dynamic type but it is a static type
- It helps the type checker to keep better track of types
- It enables the type checker to accept more correct programs
- In short, having SELF_TYPE increases the expressive power of the type system
SELF_TYPE and Dynamic Types (Example)

- What can be the dynamic type of the object returned by inc?
  - Answer: whatever could be the type of "self"

```java
class A inherits Count { };
class B inherits Count { };
class C inherits Count { };
```

(inc could be invoked through any of these classes)

- Answer: Count or any subtype of Count

SELF_TYPE and Dynamic Types (Example)

- In general, if SELF_TYPE appears textually in the class C as the declared type of E then
dynamic_type(E) ≤ C
- Note: The meaning of SELF_TYPE depends on where it appears
  - We write SELF_TYPE to refer to an occurrence of SELF_TYPE in the body of C
  - This suggests a typing rule:
    SELF_TYPE ≤ C

(*)

Type Checking

- Rule (*) has an important consequence:
  - In type checking it is always safe to replace SELF_TYPE by C
  - This suggests one way to handle SELF_TYPE:
    - Replace all occurrences of SELF_TYPE by C
  - This would be correct but it is like not having SELF_TYPE at all
Least Upper Bound on Types

- lub(X, Y), the least upper bound of X and Y, is Z if
  - X ≤ Z ∧ Y ≤ Z
    - Z is an upper bound
  - X ≤ Z' ∧ Y ≤ Z' → Z ≤ Z'
    - Z is least among upper bounds
- In COOL, the least upper bound of two types is their least common ancestor in the inheritance tree

Extending ≤ for SELF_TYPE

Let T and T’ be any types but SELF_TYPE

There are four cases in the definition of ≤

1. SELF_TYPE_c ≤ SELF_TYPE_c
   - In Cool we never need to compare SELF_TYPES coming from different classes
2. SELF_TYPE_c ≤ T if C ≤ T
   - SELF_TYPE_C can be any subtype of C
   - This includes C itself
   - Thus this is the most flexible rule we can allow
3. T ≤ SELF_TYPE_C always false
   - Note: SELF_TYPE can denote any subtype of C
4. T ≤ T’ (according to the rules from before)

Based on these rules we can extend lub ...
Extending lub(T, T')

Let T and T' be any types but SELF_TYPE

Again there are four cases:
1. lub(SELF_TYPE C, SELF_TYPE C) = SELF_TYPE C

2. lub(SELF_TYPE C, T) = lub(C, T)
   This is the best we can do because SELF_TYPE C ≤ C

3. lub(T, T') defined as before

Type Inference Rules

- Formal specifications in compilers
- Regular expressions for scanners
- Context-free grammars for parsers
- Type (inference) rules for type checkers

Inference Rules

- Form:
  - Hypothesis => conclusion ("modus ponens")
- Notation:
  - Symbol ∧ means "and"
  - Symbol ⇒ means "implies"
  - x:T means x has type T
- Simple notation and rules
  - Just requires some familiarity
English to Inference Rule

- If e₁ has type Int and e₂ has type Int, then e₁ + e₂ has type Int
- (e₁: Int ∧ e₂: Int) ⇒ e₁ + e₂: Int
- Written as:

\[
e₁: \text{Int} \\
e₂: \text{Int} \\
e₁ + e₂: \text{Int}
\]

Type Rules for COOL

- Constants:
  - false: bool
- Strings:
  - s is a string constant
  - s: String
- new:
  - new T: T (ignoring self_type)

More COOL Type Rules

- e₁: bool
  - e₂: T
  - while e₁ loop e₂ pool: Object
- x is an identifier
  - x: ?
Type Environment

- Have to add information about free variables
  - Function O from ObjectIdentifiers to Types
  - A variable is free in an expression if it is not defined in it
  - Adding a type environment:
    - O e = T
    - Assuming that variables have the types given by O, we can prove that expression e has type T
  - Rules look now as follows:
    - O e_1 : Int
    - O e_2 : Int
    - O e_1 + e_2 : Int

More Type Rules

- Variables
  - \( O(x) = T \)
  - O x : T
- Let
  - O[T_0/x] e_1 : T_1
  - O let x : T_0 in e_1 : T_1
  - O[T_0/x] = O modified to return T_0 for x

If-then-else Type Rule

- O e_1 : T_1
- O e_2 : T_2
- O e_3 : T_3
- O if e_1 then e_2 else e_3 fi : lub(T_2, T_3)
- case expression similar with lub over all cases
### Type Checking Method Dispatch

- Method and object identifiers live in different name spaces
  - Can have method named foo and object foo in same scope
  - Need a separate mapping for method name space \( M(C,f) \) = \((T_0, \ldots, T_n, T_{n+1})\) for a method \( f(x_1:T_1, \ldots, x_n:T_n):T_{n+1} \)
- Dispatch Rule:
  
  \[
  \begin{align*}
  O&M e_0: T_0 \\
  M(T_0,f) &= (T_1, \ldots, T_n, T_{n+1}) \\
  T_{n+1} &= T_i \text{ for } i \in [1..n] \\
  O&M e_0(f(e_1, \ldots, e_n)):T_{n+1}
  \end{align*}
  \]

### Additional Environment

- \( C \) = the current class in which checking takes place
- Necessary to type check expressions involving the self_type:
  
  \[
  \begin{align*}
  O&M,C e_0: T_0 \\
  M(T_0,f) &= (T_1, \ldots, T_n, T_{n+1}) \\
  T_{n+1} &= T_0 \text{ if } T_{n+1} \neq \text{SELF_TYPE} \text{ or otherwise } \\
  T_0 &= C \text{ if } T_0 = \text{SELF_TYPE}\text{ or otherwise} \\
  O&M,C e_0(f(e_1, \ldots, e_n)):T_{n+1}
  \end{align*}
  \]