Static semantics

- Last time: introduced formal specification of type-checking rules: static semantics
- Concise form of static semantics: inference rules/typing rules
- Expression/statement/program is well-formed/well-typed/legal if
  - a typing derivation (proof tree) can be constructed using available inference rules
  - syntax-directed rules: no search required to construct proof, simple recursive checker works

Sequence

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:

  \[ A \vdash S_1 : T_1 \]
  \[ A \vdash (S_2 ; S_3 ; \ldots ; S_n) : T_n \] (block)

- What about variable declarations?

Declarations

\[ A \vdash \text{id} : T [ = E ] : T_1 \]
\[ A, \text{id} : T \vdash (S_2 ; \ldots ; S_n) : T_n \] (decl block)
\[ A \vdash (\text{id} : T[ = E ]; S_2 ; \ldots ; S_n) : T_n \]

- This formally describes the type-checking code from two lectures ago!

Implementation

```java
class Block { Stmt stmts; Type typeCheck(SymTab s) { Type t;
  for (int i = 0; i < stmts.length; i++) {
    t = stmts[i].typeCheck(s);
    if (stmts[i] instanceof Decl)
      Decl d = (Decl)stmts[i];
      s = s.add(d.id, d.type.interpret());
  }
  return t;
}
}
```

Function application

- If expression E is a function value, it has a type \( T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \)
- \( T_i \) are argument types; \( T_r \) is return type
- How to type-check \( E(E_{a_1}, \ldots, E_{a_n}) \)?

\[ A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \]
\[ A \vdash E_i : T_1 \] (in \( 1 \ldots n \))
\[ A \vdash E(E_{a_1}, \ldots, E_{a_n}) : T_r \] (fn call)
Function-checking rule

- Iota function syntax
  
  \[ f(a_1 : T_1, ..., a_n : T_n) : T_r = E \]

  (f an identifier)

- Type of \( E \) must match declared return type of function \((E : T)\), but in what type context?

Add arguments to environment!

- Let \( A \) be the context surrounding the function declaration. Function decl
  
  \[ f(a_1 : T_1, ..., a_n : T_n) : T_r = E \]

  is well-formed if

  \[ A, a_i : T_i, ..., a_n : T_n \vdash E : T_r \]

  - Almost...what about recursion?

Example

```plaintext
fact(x: int) : int = {
  if (x==0) 1; else x * fact(x - 1);
}
```

```plaintext
let A be the context surrounding the function declaration. Function decl
f(a_1 : T_1, ..., a_n : T_n) : T_r = E is well-formed if
A, a_i : T_i, ..., a_n : T_n \vdash E : T_r
```

Almost...what about recursion?

How to check return?

- A return statement produces no value for its containing context to use
- Also does not return control to containing context
- For now: type \texttt{unit}

- But... how to make sure the return type of the current function is \( T \)?

Put it in the symbol table

- Add entry \{ \texttt{return} : int \} when we start checking the function, look up this entry when we hit a return statement.
- To check \( f(a_1 : T_1, ..., a_n : T_n) : T_r = E \), in environment \( A \), check
  
  \[ A, a_i : T_i, ..., a_n : T_n, \texttt{return} : T_r \vdash E : T_r \]

Completing static semantics

- Rest of static semantics written in this style: read!
- Provides complete recipe for how to show a program type-safe
- Induction on size of expressions
  - have axioms for atoms: \( A \vdash \texttt{true} : \texttt{bool} \)
  - for every AST node in language, have a rule showing how to prove it type-safe in terms of smaller exprs
- Therefore, have rules for checking all syntactically valid programs for type safety & type checker always terminates!
Handling Recursion

- Java, Iota: all global identifiers visible throughout their module (even before defn.)
- Need to create environment (symbol table) containing all of them for checking each function definition
- Global identifiers bound to their types
  \[ x: \text{int} \Rightarrow \ldots, x: \text{int}, \ldots \]
- Functions bound to function types
  \[ \text{gcd}(x: \text{int}, y: \text{int}): \text{int} \Rightarrow \ldots, \text{gcd}: \text{int} \times \text{int} \rightarrow \text{int}, \ldots \]

Auxiliary environment info

- Entries representing functions are not normal environment entries
  \{ \text{gcd}: \text{int} \times \text{int} \rightarrow \text{int} \}
- Functions not first-class values in Iota: can’t use gcd as a variable name
- Need to flag symbol table entries
- Other entries (return, etc.) also must be flagged

Handling Recursion

\[ f(x: \text{int}): \text{int} = g(x) + 1 \quad g(x: \text{int}): \text{int} = f(x) \cdot 1 \]

- Need environment containing at least
  \[ f: \text{int} \rightarrow \text{int}, g: \text{int} \rightarrow \text{int} \]
when checking both f and g
- Two-pass approach:
  - Scan top level of AST picking up all function signatures and creating an environment binding all global identifiers
  - Type-check each function individually using this global environment

Recursive Types

- Type declarations may be recursive too

Java:
  ```java
class List { Object head; List tail; }
```

C:
  ```c
  struct GraphNode { struct Arc *firstArc; }
  struct Arc { struct GraphNode *from, *to; struct Arc *nextArc; }
  ```

Interpreting type expressions

- How to convert recursive type expressions into cyclical graph structure?

  Solution: more semantic analysis passes
  - First pass: pick up all type names, create placeholder type objects and put into symbol table
  - Second pass: fill in type objects using symbol table to look up type names (can build global type context too)
  - Third pass: type-check actual code
- Mantra #2: add another pass

Where we are

```plaintext
Source code (character stream)
  + type objects, symbol tables
  + Intermediate Code
  `- Intermediate Code

Lexical analysis
  regular expressions

Syntactic Analysis
  grammars

Semantic Analysis
  static semantics

Intermediate Code Generation
```
**Intermediate Code**

- Abstract machine code - simpler
- Allows machine-independent code generation, optimization

AST \[\rightarrow\] Java bytecode

- Pentium
- Alpha

**Optimizing compilers**

- Goal: get program closer to machine code without losing information needed to do useful optimizations
- Need multiple IR stages

AST \[\rightarrow\] HIR \[\rightarrow\] LIR \[\rightarrow\] Java bytecode

- Pentium
- Alpha

**High-level IR (HIR)**

- AST + new node types not generated by parser
- Preserves high-level language constructs – structured flow, variables, methods
- Allows high-level optimizations based on properties of source language (e.g. inlining, reuse of constant variables)
- Translation ideal for visitor impl.

**Medium-level IR (MIR)**

- Intermediate between AST and assembly
- Appel’s IR: tree structured IR (triples)
- Unstructured jumps, registers, memory loc’ns
- Convenient for translation to high-quality machine code
- Other MIRs:
  - quadruples: a = b \[OP\] c ("a" is explicit, not arc)
  - UCODE: stack machine based (like Java bytecode)
  - advantage of tree IR: easy to generate, easier to do reasonable instruction selection
  - advantage of quadruples: easier optimization

**Low-level IR (LIR)**

- Assembly code + extra pseudo-instructions
- Translation to assembly code is trivial
- Allows optimization of code for low-level considerations: scheduling, memory layout

- Next time: an MIR