COS320: Compiling Techniques

Zak Kincaid

March 24, 2020
Logistics

- Midterm due Friday
- HW3 due next week (Tuesday)
Compiler phases (simplified)

1. Source text
   - Lexing
   - Token stream
     - Parsing
     - Abstract syntax tree
       - Translation
       - Intermediate representation
         - Code generation
         - Assembly
         - Optimization
Semantic Analysis
Semantic analysis

- The *semantic analysis phase* is responsible for:
  - Connecting symbol occurrences to their definitions (i.e., implement scoping rules)
  - Checking that the AST is well-typed
  - Various other well-formedness checks not captured by the grammar (e.g., `break` must appear inside a `for`, `while`, or `switch`)
Semantic analysis

- The *semantic analysis phase* is responsible for:
  - Connecting symbol *occurrences* to their definitions (i.e., implement scoping rules)
  - Checking that the AST is well-typed
  - Various other well-formedness checks not captured by the grammar (e.g., `break` must appear inside a `for`, `while`, or `switch`)

- Semantic analysis phase can report *warnings* (potential problems) or *errors* (severe problems that must be resolved in order to compile)
  - ex.c:4:5: warning: assignment makes integer from pointer without a cast
  - ex.c:3:11: error: ‘i’ undeclared (first use in this function)
Semantic analysis

- The *semantic analysis phase* is responsible for:
  - Connecting symbol *occurrences* to their definitions (i.e., implement scoping rules)
  - Checking that the AST is well-typed
  - Various other well-formedness checks not captured by the grammar (e.g., `break` must appear inside a `for`, `while`, or `switch`)

- Semantic analysis phase can report *warnings* (potential problems) or *errors* (severe problems that must be resolved in order to compile)
  - ex.c:4:5: warning: assignment makes integer from pointer without a cast
  - ex.c:3:11: error: ‘i’ undeclared (first use in this function)

- Semantic analysis may not be a separate phase – e.g., may be incorporated into IR translation
Semantic analysis

- The **semantic analysis phase** is responsible for:
  - Connecting symbol occurrences to their definitions (i.e., implement scoping rules)
  - Checking that the AST is well-typed
  - Various other well-formedness checks not captured by the grammar (e.g., break must appear inside a for, while, or switch)
- Semantic analysis phase can report **warnings** (potential problems) or **errors** (severe problems that must be resolved in order to compile)
  - ex.c:4:5: warning: assignment makes integer from pointer without a cast
  - ex.c:3:11: error: ‘i’ undeclared (first use in this function)
- Semantic analysis may not be a separate phase – e.g., may be incorporated into IR translation
- Main data structure manipulated by semantic analysis: **symbol table**
  - Mapping from symbols to information about those symbols (type, location in source text, ...)
  - Symbol table is used to help translation into IR
  - Semantic analysis may also **decorate** AST (e.g., attach type information to expressions, or replace symbols with references to their symbol table entry)
Types

- Type checking catches errors at compile time, eliminating a class of mistakes that would otherwise lead to run-time errors
- Type information is sometimes necessary for code generation
  - Floating-point + is not the same instruction as integer + is not the same as pointer/integer +
    - pointer/integer compiled differently depending on pointer type
  - Assignment $x = y$ compiled differently if $y$ is an int or a struct
What is a type?

- **Intrinsic view** (Church-style): a type is syntactically part of a term.
  - A term that cannot be typed is not a term at all
  - Types do not have inherent meaning – they are just used to define the syntax of a program

- **Extrinsic view** (Curry-style): a type is a property of a term.
  - For any term and every type, either the term has that type or not
  - A term may have multiple types
  - A term may have no types
What is a type system?

- A type system consists of a system of judgements and inference rules
  - (Extrinsic view) A judgement is a claim, which may or may not be valid
    - \( \vdash 3 : \text{int} \) – “3 has type integer”
    - \( \vdash (1 + 2) : \text{bool} \) – “(1+2) has type boolean”
  - Inference rules are used to derive valid judgements from other valid judgements.

\[
\text{ADD} \\
\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\hline
\vdash e_1 + e_2 : \text{int}
\]

Read: “If \( e_1 \) and \( e_2 \) have type \text{int}, so does \( e_1 + e_2 \)”
What is a type system?

- A type system consists of a system of judgements and inference rules
  - (Extrinsic view) A judgement is a *claim*, which may or may not be valid
    - \( \vdash 3 : \text{int} \) – “3 has type integer”
    - \( \vdash (1 + 2) : \text{bool} \) – “(1+2) has type boolean”
  - Inference rules are used to derive *valid* judgements from other valid judgements.

\[
\text{ADD} \quad \frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}}
\]

Read: “If \( e_1 \) and \( e_2 \) have type \text{int}, so does \( e_1 + e_2 \)”

- Type system might involve many different kinds of judgement
  - Well-typed expressions
  - Well-formed types
  - Well-formed statements
  - …
Inference rules, generally

- An *inference rule* consists of a list of *premises* $J_1, \ldots, J_n$ and one *conclusion* $J$ (and optionally a side-condition), typically written as:

$$
\begin{array}{c c c c}
J_1 & J_2 & \cdots & J_n \\
\hline
J
\end{array}
\text{ SIDE-CONDITION}

- Side-condition: additional premise, but not a judgement
- Read *top-down*: If $J_1$ and $J_2$ and ... and $J_n$ are valid, and the side condition holds, then $J$ is valid.
- Read *bottom-up*: To prove $J$ is valid, sufficient to prove $J_1, J_2, \ldots, J_n$ are valid.
A simple expression language

- Syntax of expressions

\[
<\text{Exp}> ::= \text{Var} | \text{Int} \\
| \text{Exp} + \text{Exp} | \text{Exp} \times \text{Exp} \\
| \text{Exp} \land \text{Exp} | \text{Exp} \lor \text{Exp} \\
| \text{Exp} \leq \text{Exp} | \text{Exp} = \text{Exp} \\
| \text{if} \ \text{Exp} \ \text{then} \ \text{Exp} \ \text{else} \ \text{Exp}
\]

- \(3 + (2 \land 0)\) is syntactically well-formed, but not well-typed
- Is \(x + 1\) well-typed?
Type judgements

- A **type environment** is a symbol table mapping symbols to types.
  - E.g., \([x \mapsto \text{int}, y \mapsto \text{bool}, z \mapsto \text{int}]\): \(x\) and \(z\) are ints, \(y\) is a bool
  - Notation: type environment denoted by \(\Gamma\)
  - Notation: \(\Gamma\{x \mapsto t}\) is a functional update

\[
\Gamma\{x \mapsto t\}(y) = \begin{cases} 
t & \text{if } x = y \\
\Gamma(y) & \text{otherwise}
\end{cases}
\]
Type judgements

- A **type environment** is a symbol table mapping symbols to types.
  - E.g., $[x \mapsto \text{int}, y \mapsto \text{bool}, z \mapsto \text{int}]$: $x$ and $z$ are ints, $y$ is a bool
  - Notation: type environment denoted by $\Gamma$
  - Notation: $\Gamma\{x \mapsto t\}$ is a functional update

\[
\Gamma\{x \mapsto t\}(y) = \begin{cases} 
t & \text{if } x = y \\
\Gamma(y) & \text{otherwise}
\end{cases}
\]

- A **type judgement** takes the form $\Gamma \vdash e : t$
  - Read “Under the type environment $\Gamma$, the expression $e$ has type $t$”
Inference rules

**INT**

\[
\Gamma \vdash n : \text{int} \quad n \in \{\ldots, -1, 0, 1, \ldots\}
\]

**VAR**

\[
\Gamma \vdash x : t \\
\Gamma(x) = t
\]

**ADD**

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 + e_2 : \text{int}
\]

**AND**

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
\Gamma \vdash e_1 \land e_2 : \text{bool}
\]

**LEQ**

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 \leq e_2 : \text{bool}
\]

**IF**

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]
A derivation or proof tree is a tree where each node is labelled by a judgement, and edges connect premises to a conclusion according to some inference rule.

Leaves of the tree are axioms (inference rules w/o premises)

Derivation of $x : \text{int} \vdash 2 + x \leq 10 : \text{bool}$:

```
ADD
   \[ x : \text{int} \vdash 2 : \text{int} \]
   \[ x : \text{int} \vdash x : \text{int} \]
   \[ x : \text{int} \vdash 2 + x : \text{int} \]
LEQ
   \[ x : \text{int} \vdash 2 + x : \text{int} \]
   \[ x : \text{int} \vdash 10 : \text{int} \]
   \[ x : \text{int} \vdash 2 + x \leq 10 : \text{bool} \]
```
Derivation for $x : \text{int} \vdash \text{if } x \leq 0 \text{ then } x \text{ else } -1 \ast x : \text{int}$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>$x : \text{int} \vdash x : \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>$x : \text{int} \vdash -1 : \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEQ</td>
<td>$x : \text{int} \vdash x \leq 0 : \text{bool}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAR</td>
<td>$x : \text{int} \vdash x : \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>$x : \text{int} \vdash -1 \ast x : \text{int}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$x : \text{int} \vdash x \leq 0 \text{ then } x \text{ else } -1 \ast x : \text{int}$
Type checking

- Goal of a type checker: given a context $\Gamma$, expression $e$, and type $t$, determine whether a derivation of the judgement $\Gamma \vdash e : t$ exists.
- Method: recurse on the structure of the AST, applying inference rules “bottom-up”
Binders & functions: scope logic

**Let**
\[
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \{x \mapsto t_1\} \vdash e_2 : t}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t}
\]

**Fun**
\[
\frac{\Gamma \{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash \text{fun } (x : t_1) \rightarrow e : t_1 \rightarrow t_2}
\]

**App**
\[
\frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 \ e_2 : t_2}
\]
Type inference

- Goal of type inference: given a context $\Gamma$ and expression $e$, determine a type $t$ for which there is a derivation of the judgement $\Gamma \vdash e : t$.

- Method: (again) recurse on the structure of the AST, applying inference rules “bottom-up”

- This only works because we have a very simple type system
  - OCaml type inference: recurse on the structure of the AST to produce a constraint system, then solve the constraints
Robin Milner: “Well typed programs do not go wrong”

More formally: if $\vdash e : t$ is derivable, then evaluating $e$ either fails to terminate or yields a value of type $t$

Note: for our language (extension of simply-typed lambda calculus with integers and booleans), we have something stronger: evaluating $e$ always yields a value of type $t$
Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H ⊢ t$
  - $H$ is set of type names
  - $t$ is a type
  - $H ⊢ t$ - “Assuming $H$ names well-formed types, $t$ is a well-formed type”
Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
  - $H$ is set of type names
  - $t$ is a type
  - $H \vdash t$ – “Assuming $H$ names well-formed types, $t$ is a well-formed type”

\[
\begin{align*}
\text{INT} & \quad \text{BOOL} \\
\hline
H \vdash \text{int} & \quad H \vdash \text{bool} \\
\end{align*}
\]

\[
\begin{align*}
\text{ARROW} & \quad \text{NAMED} \\
\hline
H \vdash t_1 & \quad H \vdash t_2 \\
\hline
H \vdash t_1 \rightarrow t_2 \\
\hline
H \vdash s & \quad s \in H
\end{align*}
\]
Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
  - $H$ is set of type names
  - $t$ is a type
  - $H \vdash t$ – “Assuming $H$ names well-formed types, $t$ is a well-formed type”

<table>
<thead>
<tr>
<th>INT</th>
<th>BOOL</th>
<th>ARROW</th>
<th>NAMED</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ \vdash \ \text{int}$</td>
<td>$\ \vdash \ \text{bool}$</td>
<td>$H \vdash t_1$ \quad $H \vdash t_2$ \quad $H \vdash t_1 \rightarrow t_2$</td>
<td>$H \vdash s$ \quad $s \in H$</td>
</tr>
</tbody>
</table>

- Note: also need to modify the typing rules & judgements. E.g.,

  **Fun**
  $$H \vdash t_1 \quad H, \Gamma \{x \rightarrow t_1\} \vdash e : t_2$$
  $$H, \Gamma \vdash \mathbf{fun} (x : t_1) \rightarrow e : t_1 \rightarrow t_2$$
In languages with statements, need additional rules to defined well-formed statements

E.g., judgements may take the form $D; \Gamma; rt \vdash s$

- $D$ maps type names to their definitions
- $\Gamma$ is a type environment (variables $\rightarrow$ types)
- $rt$ is a type
- $D; \Gamma; rt \vdash s$ - “with type definitions $D$, assuming type environment $\Gamma$, $s$ is a valid statement within the context of a function that returns a value of type $rt$”
• In languages with statements, need additional rules to defined well-formed statements
• E.g., judgements may take the form $D; \Gamma; rt \vdash s$
  • $D$ maps type names to their definitions
  • $\Gamma$ is a type environment (variables $\rightarrow$ types)
  • $rt$ is a type
  • $D; \Gamma; rt \vdash s$ - “with type definitions $D$, assuming type environment $\Gamma$, $s$ is a valid statement within the context of a function that returns a value of type $rt$”

<table>
<thead>
<tr>
<th>ASSIGN</th>
<th>RETURN</th>
<th>DECL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e : \Gamma(x)$</td>
<td>$\Gamma \vdash e : rt$</td>
<td>$\Gamma \vdash e : t$</td>
</tr>
<tr>
<td>$D; \Gamma; rt \vdash x := e$</td>
<td>$D; \Gamma; rt \vdash \text{return } e$</td>
<td>$D; \Gamma; rt \vdash \text{var } x = e; s_2$</td>
</tr>
</tbody>
</table>
Additional aspects

- In OCaml, can have a variable and a type with the same name
  - Multiple namespaces ⇒ multiple environments / symbol tables
- Parametric polymorphism
  - E.g., `fun x -> x` in ocaml has type `'a -> 'a`
  - Finite representation of infinitely many typings
- Subtyping (e.g., object-oriented languages) – next time
  - Related: casting, coercion