COS320: Compiling Techniques

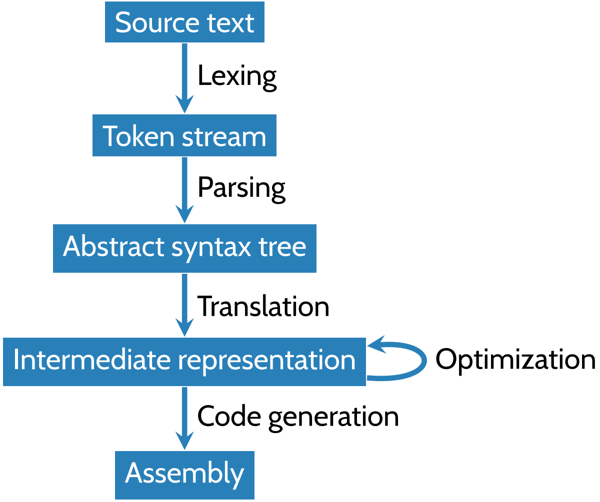
Zak Kincaid

March 24, 2020

Logistics

- Midterm due Friday
- HW3 due next week (Tuesday)

Compiler phases (simplified)



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
    - $\vdash 3 : \texttt{int}$ – "3 has type integer"
    - $\vdash (1+2) : \texttt{bool}$ – "(1+2) has type boolean"
  - **Inference rules** are used to derive *valid* judgements from other valid judgements.

$$\textsc{Add}\quad \frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

Read: "If e_1 and e_2 have type `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
    - $\vdash 3 : \texttt{int}$ – "3 has type integer"
    - $\vdash (1+2) : \texttt{bool}$ – "(1+2) has type boolean"
  - **Inference rules** are used to derive *valid* judgements from other valid judgements.

$$\textsc{Add}\ \frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

  Read: "If e_1 and e_2 have type $\texttt{int}$, 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:

$$\frac{J_1 \qquad J_2 \qquad \cdots \qquad J_n}{J} \textsc{Side-condition}$$

  - 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, … J_n are valid

A simple expression language

- Syntax of expressions

```
<Exp> ::= <Var> | <Int>
        | <Exp>+<Exp> | <Exp>*<Exp>
        | <Exp>∧<Exp> | <Exp>∨<Exp>
        | <Exp>≤<Exp> | <Exp>=<Exp>
        | if <Exp> then <Exp> else <Exp>
```

- `3 + (2 ∧ 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., $[\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{bool}, \mathtt{z} \mapsto \mathtt{int}]$: `x` and `z` are `int`s, `y` is a bool
  - Notation: type environment denoted by Γ
  - 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., $[\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{bool}, \mathtt{z} \mapsto \mathtt{int}]$: x and z are ints, y is a bool
  - Notation: type environment denoted by Γ
  - 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 Γ, the expression e has type t"

Inference rules

Int

$$\frac{}{\Gamma \vdash n : \texttt{int}} \; n \in \{..., -1, 0, 1, ...\}$$

Var

$$\frac{}{\Gamma \vdash x : t} \; \Gamma(x) = t$$

Add

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

And

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \wedge e_2 : \texttt{bool}}$$

Leq

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \leq e_2 : \texttt{bool}}$$

If

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

Derivations

- 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 $x : \texttt{int} \vdash 2 + x \leq 10 : \texttt{bool}$:

$$\textsc{Leq}\;\dfrac{\textsc{Add}\;\dfrac{\textsc{Int}\;\dfrac{}{x : \texttt{int} \vdash 2 : \texttt{int}} \quad \textsc{Var}\;\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}}}{x : \texttt{int} \vdash 2 + x : \texttt{int}} \quad \textsc{Int}\;\dfrac{}{x : \texttt{int} \vdash 10 : \texttt{int}}}{x : \texttt{int} \vdash 2 + x \leq 10 : \texttt{bool}}$$

Derivation for $x : \texttt{int} \vdash \textbf{if } x \leq 0 \textbf{ then } x \textbf{ else } -1 * x : \texttt{int}$:

$$\textsc{If}\ \dfrac{\textsc{Leq}\ \dfrac{\textsc{Var}\ \dfrac{}{x : \texttt{int} \vdash x : \texttt{int}} \qquad \textsc{Int}\ \dfrac{}{x : \texttt{int} \vdash -1 : \texttt{int}}}{x : \texttt{int} \vdash x \leq 0 : \texttt{bool}} \qquad \textsc{Var}\ \dfrac{}{x : \texttt{int} \vdash x : \texttt{int}} \qquad \textsc{Mul}\ \dfrac{\textsc{Int}\ \dfrac{}{x : \texttt{int} \vdash -1 : \texttt{int}} \qquad \textsc{Var}\ \dfrac{}{x : \texttt{int} \vdash x : \texttt{int}}}{x : \texttt{int} \vdash -1 * x : \texttt{int}}}{x : \texttt{int} \vdash \textbf{if } x \leq 0 \textbf{ then } x \textbf{ else } -1 * x : \texttt{int}}$$

Type checking

- Goal of a type checker: given a context Γ, 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

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma\{x \mapsto t_1\} \vdash e_2 : t}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t}$$

Fun

$$\frac{\Gamma\{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash \textbf{fun}\ (x : t_1)\texttt{->}e : t_1 \to t_2}$$

App

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

Type inference

- Goal of type inference: given a context Γ 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

Type soundness

- 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 $H \vdash t$
  - 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"

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"

INT

$$\frac{}{H \vdash \texttt{int}}$$

BOOL

$$\frac{}{H \vdash \texttt{bool}}$$

ARROW

$$\frac{H \vdash t_1 \qquad H \vdash t_2}{H \vdash t_1 \rightarrow t_2}$$

NAMED

$$\frac{}{H \vdash s} \; s \in H$$

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"

INT
$$\frac{}{H \vdash \texttt{int}}$$

BOOL
$$\frac{}{H \vdash \texttt{bool}}$$

ARROW
$$\frac{H \vdash t_1 \qquad H \vdash t_2}{H \vdash t_1 \rightarrow t_2}$$

NAMED
$$\frac{}{H \vdash s} \; s \in H$$

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

FUN
$$\frac{H \vdash t_1 \qquad H, \Gamma\{x \mapsto t_1\} \vdash e : t_2}{H, \Gamma \vdash \textbf{fun } (x : t_1)\texttt{->}e : t_1 \rightarrow t_2}$$

Statements

- 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
  - Γ is a type environment (variables $\rightarrow$ types)
  - rt is a type
  - $D; \Gamma; rt \vdash s$ – "with type definitions D, assuming type environment Γ, s is a valid statement within the context of a function that returns a value of type rt"

Statements

- 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
  - Γ is a type environment (variables $\rightarrow$ types)
  - rt is a type
  - $D; \Gamma; rt \vdash s$ – "with type definitions D, assuming type environment Γ, s is a valid statement within the context of a function that returns a value of type rt"

ASSIGN

$$\frac{\Gamma \vdash e : \Gamma(x)}{D; \Gamma; rt \vdash x := e}$$

RETURN

$$\frac{\Gamma \vdash e : rt}{D; \Gamma; rt \vdash \textbf{return } e}$$

DECL

$$\frac{\Gamma \vdash e : t \qquad D; \Gamma\{x \mapsto t\}; rt \vdash s_2}{D; \Gamma; rt \vdash \textbf{var } x = e; s_2}$$

Additional aspects

- In OCaml, can have a variable and a type with the same name
  - Multiple namespaces $\Rightarrow$ 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, coersion