Compilation I: Stack-Based Languages

CAS CS 320: Principles of Programming Languages

December 3, 2024 (Lecture 24)

Practice Problem

$$\cdot \vdash \lambda x.xx : \tau \dashv \mathcal{C}$$

Determine τ and $\mathcal{C}$ so that the above judgment is derivable in Hindley-Milner Light (HM$^-$). Explain why $\mathcal{C}$ doesn't have a unifier

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x:\alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \quad \alpha \text{ is fresh}}{\Gamma \vdash e_1 e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, \mathcal{C}_1, \mathcal{C}_2}$$

Answer

$$\cdot \vdash \lambda x.xx : \tau \dashv \mathcal{C}$$

$\cdot \vdash \lambda x.xx : \alpha \rightarrow \beta \dashv \alpha \doteq \alpha \rightarrow \beta$

$\quad x : \alpha \vdash xx : \beta \dashv \alpha \doteq \alpha \rightarrow \beta$

$\quad\quad x : \alpha \vdash x : \alpha \dashv \emptyset \quad (var)$

$\quad\quad x : \alpha \vdash x : \alpha \dashv \emptyset \quad (var)$

$\tau = \alpha \rightarrow \beta$

$C = \{\alpha \doteq \alpha \rightarrow \beta\}$

var   ty

α appears free

α

$\downarrow$

$\alpha \rightarrow \beta$

$\downarrow$

$(\alpha \rightarrow \beta) \rightarrow \beta$

$\downarrow$

$((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \beta$

$\downarrow \vdots$

Today

- Finish up an implementation of HM$^-$
- Briefly discuss stack-based languages

Learning Objectives

- Write an evaluation sequence for a program in a stack-based language for arithmetic with variables
- Compile an arithmetic expression to a program in this stack-based language

$$
\begin{aligned}
e \;&::=\; \lambda x.e \mid ee \\
&\mid\; \mathsf{let}\ x = e\ \mathsf{in}\ e \\
&\mid\; \mathsf{let\ rec}\ f\ x = e\ \mathsf{in}\ e \\
&\mid\; \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
&\mid\; e + e \mid e = e \\
&\mid\; num \mid x \\
\sigma \;&::=\; \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid \alpha \\
\tau \;&::=\; \sigma \mid \forall \alpha.\tau
\end{aligned}
$$

Recap: Type System (Basics)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int} \dashv \varnothing} \text{ (int)} \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma \dashv \varnothing} \text{ (var)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 + e_2 : \text{int} \dashv \tau_1 \doteq \text{int}, \tau_2 \doteq \text{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (add)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_3 \dashv \tau_1 \doteq \text{bool}, \tau_2 \doteq \tau_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3} \text{ (if)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \text{int} \dashv \tau_1 \doteq \text{int}, \tau_2 \doteq \text{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (eq)}$$

Recap: Type System (Functions and Variables)

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}} \ \text{(fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1 e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, \mathcal{C}_1, \mathcal{C}_2} \ \text{(app)}$$

$$\frac{(x : \forall \alpha_1.\forall \alpha_2 \ldots \forall \alpha_k.\tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1] \ldots [\beta_k/\alpha_k]\tau} \ \text{(var)}$$

polymorphism (handwritten annotation in red)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \text{let } x \doteq e_1 \text{ in } e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2} \ \text{(let)}$$

$$\frac{\alpha, \beta \text{ are free} \qquad \Gamma, f : \alpha \to \beta, x : \alpha \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, f : \alpha \to \beta \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \text{let rec } f \ x = e_1 \text{ in } e_2 : \tau_2 \dashv \beta = \tau_1, \mathcal{C}_1, \mathcal{C}_2} \ \text{(letRec)}$$

Recap:Type Unification

```
type ty =
   | TInt
   | TBool
   | TFun of ty * ty
   | TVar of string
```

Type unification is the particular unification problem over the `ty`
ADT (with type variable acting as variables in the sense from the
previous slide)

$$\alpha \doteq int$$

$$\beta \doteq \alpha \to \alpha$$

$$S = \{ \alpha \mapsto int,$$
$$\beta \mapsto int \to int \}$$

Recap: Principle Types

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

The constraints $\mathcal{C}$ define a unification problem

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ can get the "actual" type of e, its τ *after the substitution $\mathcal{S}$*, i.e., $\mathcal{S}\tau$}

If $\mathcal{S}$ is the most general unifier then $\mathcal{S}\tau$ (after "polymorphization") is called the **principle type** of e. The principle type has the property that every other type is an *instance* of it

Recap: Putting Everything Together

▸ Constraint-based inference:

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \alpha \to \beta \to \eta \dashv \alpha \doteq \delta \to \eta, \gamma \doteq \mathsf{int} \to \delta,$$
$$\mathsf{int} \to \mathsf{int} \to \mathsf{int} \doteq \beta \to \gamma$$

▸ Unification:

$$\mathcal{S} = \{\alpha \mapsto \delta \to \eta, \gamma \mapsto \mathsf{int} \to \delta, \beta \mapsto \mathsf{int}\}$$
$$\mathcal{S}(\alpha \to \beta \to \eta) = (\mathsf{int} \to \eta) \to \mathsf{int} \to \eta$$

▸ Generalization:

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \forall \eta.(\mathsf{int} \to \eta) \to \mathsf{int} \to \eta$$

Demo: Type Inference

We'll finish up an implementation of `type_of` for HM^-

Outline

High-Level

A stack-oriented language is a programming language which directly manipulates a stack of values (or multiple stacks)

There are roughly two categories of stack-oriented languages:

- ▸ "usable" stack-oriented languages, e.g. Forth
- ▸ instruction sets for virtual machines, e.g., JVM, CPython interpreter, Lua (not any more), OCaml bytecode interpreter

A virtual (stack) machine is a computational abstraction, like a Turing machine (but usually **easier to implement**).

Virtual machines are typically implemented as bytecode interpreters, where "programs" are streams of bytes and a command in the language are represented as a byte

Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

Portability: Any OS should be able to handle a stream of bytes, so the machine dependent part of our programming language can be simplified.

Efficiency (sort of): They can be implemented in low-level languages, and so will generally be faster than the interpreters we build in this course (though not as fast as natively compiled code).

Looking Forward: Compilation
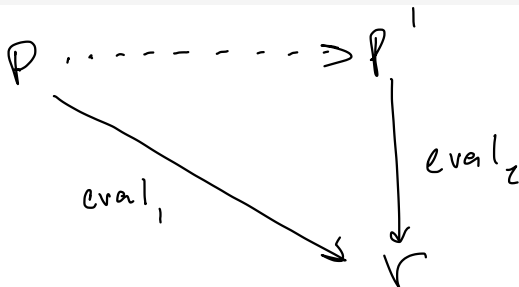
Compilation is the process of *translating* a program in one language to another, maintaining semantic behavior.

Compilation can be a part of interpretation as well, like with bytecode interpretation (this is what OCaml does).

The simple case for today: *every arithmetic expression can be represented as an equivalent expression in reverse polish notation.*

Arithmetic (Syntax)

$$\begin{array}{lll}
\text{<prog>} & ::= & \{\text{<com>}\} \\
\text{<com>} & ::= & \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{PUSH}\ \text{<num>} \\
\text{<num>} & ::= & \mathbb{Z}
\end{array}$$

$$\langle \; \mathcal{S} \; , \; \mathcal{P} \; \rangle$$

We take a value to be an integer ($\mathbb{Z}$)

A **configuration** is made up of a stack ($\mathcal{S}$) of values and a program ($\mathcal{P}$) given by `<prog>`

Arithmetic (Small-step Semantics)

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \text{ADD } \mathcal{P} \,\rangle \longrightarrow \langle\, (m + n) :: \mathcal{S} \,,\, \mathcal{P} \,\rangle} \ (\text{add})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \text{SUB } \mathcal{P} \,\rangle \longrightarrow \langle\, (m - n) :: \mathcal{S} \,,\, \mathcal{P} \,\rangle} \ (\text{sub})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \text{MUL } \mathcal{P} \,\rangle \longrightarrow \langle\, (m \times n) :: \mathcal{S} \,,\, \mathcal{P} \,\rangle} \ (\text{mul})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z} \qquad n \neq 0}{\langle\, m :: n :: \mathcal{S} \,,\, \text{DIV } \mathcal{P} \,\rangle \longrightarrow \langle\, (m/n) :: \mathcal{S} \,,\, \mathcal{P} \,\rangle} \ (\text{div})$$

$$\frac{}{\langle\, \mathcal{S} \,,\, \text{PUSH } n \ \mathcal{P} \,\rangle \longrightarrow \langle\, n :: \mathcal{S} \,,\, \mathcal{P} \,\rangle} \ (\text{push})$$

Example (Evaluation)

```
PUSH 2 PUSH 3 ADD PUSH 4 MUL
```

Example (Compilation)

```
4 * (2 + 3)
```

Demo: Compiling Arithmetic Expressions

We'll walk though a small bit of code for compiling arithmetic expressions, both into a program and into a stream of bytes which piped to a bytecode interpreter.

We'll talk more about bytecode on Thursday. The main idea: *represent commands in our language as bytes for better portability.*

(Immutable) Variables (Syntax)

$$
\begin{array}{rcl}
\texttt{<prog>} & ::= & \{\texttt{<com>}\} \\
\texttt{<com>} & ::= & \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{PUSH <num>} \\
& & \mid \quad \texttt{ASSIGN <var>} \mid \texttt{LOOKUP <var>} \\
\texttt{<num>} & ::= & \mathbb{Z} \\
\texttt{<var>} & ::= & \mathbb{I}
\end{array}
$$

(Immutable) Variables (Values and Configurations)

$$\langle \; \mathcal{S} \; , \; \mathcal{E} \; , \; \mathcal{P} \; \rangle$$

We take a value to be an integer ($\mathbb{Z}$)

A **configuration** is made up of a stack ($\mathcal{S}$) of values, *an environment* ($\mathcal{E}$) *mapping identifiers* ($\mathbb{I}$) *to values,* and a program ($\mathcal{P}$) given by

(Immutable) Variables (Small-step Semantics)

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{ADD } \mathcal{P} \,\rangle \longrightarrow \langle\, (m+n) :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \; (\text{add})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{SUB } \mathcal{P} \,\rangle \longrightarrow \langle\, (m-n) :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \; (\text{sub})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\, m :: n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{MUL } \mathcal{P} \,\rangle \longrightarrow \langle\, (m \times n) :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \; (\text{mul})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z} \qquad n \neq 0}{\langle\, m :: n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{DIV } \mathcal{P} \,\rangle \longrightarrow \langle\, (m/n) :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \; (\text{div})$$

$$\frac{}{\langle\, \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{PUSH } n \; \mathcal{P} \,\rangle \longrightarrow \langle\, n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \; (\text{push})$$

(Immutable) Variables (Small-step Semantics)

$$\frac{}{\langle\, n :: \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{ASSIGN } x \; \mathcal{P} \,\rangle \longrightarrow \langle\, \mathcal{S} \,,\, \mathcal{E}[x \mapsto n] \,,\, \mathcal{P} \,\rangle} \;\; \text{(assign)}$$

$$\frac{\mathcal{E}(x) \neq \bot}{\langle\, \mathcal{S} \,,\, \mathcal{E} \,,\, \texttt{LOOKUP } x \; \mathcal{P} \,\rangle \longrightarrow \langle\, \mathcal{E}(x) :: \mathcal{S} \,,\, \mathcal{E} \,,\, \mathcal{P} \,\rangle} \;\; \text{(assign)}$$

Example

```
PUSH 2 ASSIGN x PUSH 3 ASSIGN y
LOOKUP x LOOKUP y ADD
```

Scoping

The language we've just described is only good for compiling from languages with dynamic scoping.

How would we compile the following program?

```
let y = 1 in
let x =
  let y = 2 in
  y + y
in x + y
```

Next time. We will introduce closures into our stack-based language so that it a better target for a function language like OCaml.