

# 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  $\tau$  and  $\mathcal{C}$  so that the above judgment is derivable in Hindley-Milner Light (HM<sup>-</sup>). Explain why  $\mathcal{C}$  doesn't have a unifier

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

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

# Answer

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

$$\cdot \vdash \lambda x. xx : \boxed{\alpha} \rightarrow \boxed{\beta} \vdash \alpha \doteq \alpha \rightarrow \beta \quad (\text{fun})$$

$$\vdash x : \boxed{\alpha} \vdash xx : \boxed{\beta} \vdash \alpha \doteq \alpha \rightarrow \beta \quad (\text{app})$$

$$\left\{ \begin{array}{l} x : \alpha \vdash x : \alpha \vdash \emptyset \quad (\text{var}) \\ x : \alpha \vdash x : \alpha \vdash \emptyset \quad (\text{var}) \end{array} \right.$$

$$\tau = \alpha \rightarrow \beta$$

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

*var*
*ty*

*α appears in ty*

$$\begin{array}{c}
 \alpha \\
 \downarrow \\
 \alpha \rightarrow \beta \\
 \downarrow \\
 (\alpha \rightarrow \beta) \rightarrow \beta \\
 \downarrow \\
 (\lambda \alpha \rightarrow \beta) \rightarrow \beta \rightarrow \beta
 \end{array}$$

# 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

Recap: Type Inference

Stack-Based Languages

## Recap: Syntax (Mathematical)

$$\begin{aligned} e &::= \lambda x.e \mid ee \\ &\mid \text{let } x = e \text{ in } e \\ &\mid \text{let rec } f \text{ } x = e \text{ in } e \\ &\mid \text{if } e \text{ then } e \text{ else } e \\ &\mid e + e \mid e = e \\ &\mid \text{num} \mid x \\ \sigma &::= \text{int} \mid \text{bool} \mid \sigma \rightarrow \sigma \mid \alpha \quad \text{monotype} \\ \tau &::= \sigma \mid \forall \alpha. \tau \quad \text{polytype} \end{aligned}$$

## Recap: Type System (Basics)

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

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

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

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

bool



## Recap: Type System (Functions and Variables)

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

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

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

poly morphism

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

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

## 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)

## 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  $\tau$  *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:

$$\begin{aligned} \cdot \vdash \lambda f. \lambda x. f(x+1) : \alpha \rightarrow \beta \rightarrow \eta \dashv \vdash \alpha \doteq \delta \rightarrow \eta, \gamma \doteq \text{int} \rightarrow \delta, \\ \text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq \beta \rightarrow \gamma \end{aligned}$$

- ▶ Unification:

$$\begin{aligned} \mathcal{S} &= \{\alpha \mapsto \delta \rightarrow \eta, \gamma \mapsto \text{int} \rightarrow \delta, \beta \mapsto \text{int}\} \\ \mathcal{S}(\alpha \rightarrow \beta \rightarrow \eta) &= (\text{int} \rightarrow \eta) \rightarrow \text{int} \rightarrow \eta \end{aligned}$$

- ▶ Generalization:

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

## Demo: Type Inference

We'll finish up an implementation of `type_of` for  $\text{HM}^-$

Recap: Type Inference

Stack-Based Languages

# 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

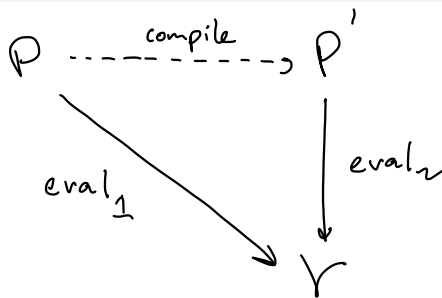
**Simplicity:** Stacks aren't too complicated

**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)

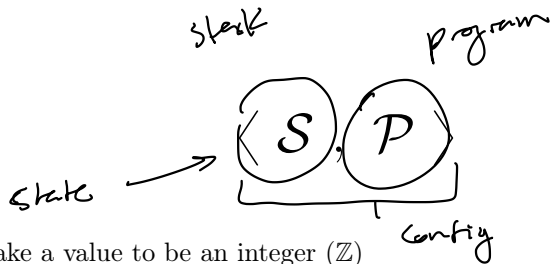
$$\langle \text{prog} \rangle ::= \{ \langle \text{com} \rangle \}$$

```
<com> ::= ADD | SUB | MUL | DIV | PUSH <num>
```

$$\langle \text{num} \rangle ::= \mathbb{Z}$$

PUSH 2      PUSH 3      ADD  
2                      3              +

# Arithmetic (Values and Configurations)



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

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

# Arithmetic (Small-step Semantics)

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

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

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

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

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

## Example (Evaluation)

PUSH 2 PUSH 3 ADD PUSH 4 MUL

$$\begin{aligned} \langle \emptyset, P \rangle &\rightarrow \langle 2 :: \emptyset, \text{PUSH 3 ADD PUSH 4 MUL} \rangle \\ &\rightarrow \langle 3 :: 2 :: \emptyset, \text{ADD PUSH 4 MUL} \rangle \\ &\rightarrow \langle 5 :: \emptyset, \text{PUSH 4 MUL} \rangle \\ &\rightarrow \langle 4 :: 5 :: \emptyset, \text{MUL} \rangle \\ &\rightarrow \langle 20 :: \emptyset, \epsilon \rangle \quad \checkmark \end{aligned}$$

## Example (Compilation)

4 \* (2 + 3)

4 \* (2 + 3)  $\longrightarrow$  P3 P2 ADD PUSH 4 MUL

↓

4 \* 5

↓

20

⋮

↓

$\langle 20 :: \cancel{0}, 2 \rangle$

## Demo: Compiling Arithmetic Expressions

We'll walk through 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)

$\langle \text{prog} \rangle ::= \{ \langle \text{com} \rangle \}$

$\langle \text{com} \rangle ::= \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV} \mid \text{PUSH } \langle \text{num} \rangle$   
 $\quad \mid \text{ASSIGN } \langle \text{var} \rangle \mid \text{LOOKUP } \langle \text{var} \rangle$

$\langle \text{num} \rangle ::= \mathbb{Z}$

$\langle \text{var} \rangle ::= \mathbb{I}$



## (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  
<prog>

## (Immutable) Variables (Small-step Semantics)

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

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

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

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

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

## (Immutable) Variables (Small-step Semantics)

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

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

## 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.