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

Answer

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

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

Outline

Recap: Type Inference

Stack-Based Languages

Recap: Syntax (Mathematical)

$$
\begin{array}{rcl}
e & ::= & \lambda x.e \mid ee \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid & \color{red}{\mathsf{let\ rec}\ f\ x = e\ \mathsf{in}\ e} \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & e + e \mid e = e \\
& \mid & num \mid x \\
\sigma & ::= & \mathsf{int} \mid \mathsf{bool} \mid \sigma \rightarrow \sigma \mid \alpha \\
\tau & ::= & \sigma \mid \forall \alpha.\tau
\end{array}
$$

Recap: Type System (Basics)

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

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

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

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

Recap: Type System (Functions and Variables)

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

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

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

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

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

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 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

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)

<prog> ::= $\{$<com>$\}$

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

<num> ::= $\mathbb{Z}$

Arithmetic (Values and Configurations)

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

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)

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

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

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

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

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

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)

<prog> ::= {<com>}

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

| `ASSIGN` <var> | `LOOKUP` <var>

<num> ::= $\mathbb{Z}$

<var> ::= $\mathbb{I}$

(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)

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

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

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

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

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

(Immutable) Variables (Small-step Semantics)

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

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

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.