

Compilation II: In Practice

CAS CS 320: Principles of Programming Languages

December 5, 2024 (Lecture 25)

Practice Problem

$$\cdot \vdash \lambda x. \lambda y. \lambda z. xz(yz) : \tau \multimap \mathcal{C}$$

Determine τ and \mathcal{C} so that the above judgment is derivable in Hindley-Milner Light (HM^-). Solve the constraints \mathcal{C} and determine the principle type of this expression

Answer

$$\cdot \vdash \lambda x. \lambda y. \lambda z. xz(yz) : \tau \rightarrow C$$

$$\cdot \vdash \lambda x. \lambda y. \lambda z. xz(yz) : \alpha \rightarrow \beta \rightarrow \delta \rightarrow \varepsilon \vdash C$$

$$\vdash \{x : \alpha\} \vdash \lambda y. \lambda z. xz(yz) : \beta \rightarrow \delta \rightarrow \varepsilon \vdash C$$

$$\vdash \{x : \alpha, y : \beta, z : \delta\} \vdash (xz)(yz) : \varepsilon \vdash$$

$$\vdash \{ \dots \} \vdash xz : \delta \vdash \alpha \doteq \delta \rightarrow \delta$$

$$\vdash \{ \dots \} \vdash x : \alpha \vdash \emptyset$$

$$\vdash \{ \dots \} \vdash z : \delta \vdash \emptyset$$

$$\vdash \{ \dots \} \vdash yz : \eta \vdash \beta \doteq \delta \rightarrow \eta$$

$$\vdash \{ \dots \} \vdash y : \beta \vdash \emptyset$$

$$\vdash \{ \dots \} \vdash z : \delta \vdash \emptyset$$

$$C =$$

$$\delta \doteq \eta \rightarrow \varepsilon$$

$$\alpha \doteq \delta \rightarrow \delta$$

$$\beta \doteq \delta \rightarrow \eta$$

$$S = \{ \begin{array}{l} \delta \mapsto \eta \rightarrow \varepsilon \\ \alpha \mapsto \gamma \rightarrow \eta \rightarrow \varepsilon \\ \beta \mapsto \gamma \rightarrow \eta \end{array} \}$$

S_T

$$T = \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \varepsilon$$

$$C = \begin{array}{l} r \doteq + \quad \delta \doteq \eta \rightarrow \varepsilon \\ \alpha \doteq \gamma \rightarrow \eta \rightarrow \varepsilon \\ \beta \doteq \gamma \rightarrow \eta \end{array}$$

$$\begin{array}{l} \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \varepsilon \\ \downarrow \delta \mapsto \eta \rightarrow \varepsilon \\ \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \varepsilon \\ \downarrow \alpha \mapsto \gamma \rightarrow \eta \rightarrow \varepsilon \\ (\gamma \rightarrow \eta \rightarrow \varepsilon) \rightarrow \beta \rightarrow \gamma \rightarrow \varepsilon \\ \downarrow \beta \mapsto \gamma \rightarrow \eta \end{array}$$

$$\varepsilon \leftarrow \gamma \leftarrow (\eta \leftarrow \delta) \leftarrow (\gamma \leftarrow \eta \rightarrow \varepsilon) \leftarrow \beta \leftarrow \gamma \leftarrow \eta$$

Today

- ▶ Discuss stack-based languages
- ▶ Look a bit more deeply at the inner-workings of OCaml's compiler
- ▶ Talk briefly about what you can do if you're still interested in PL
- ▶ Fill out course evals(!)

Recap: Stack-Based Languages

Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

Recall: 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

Arithmetic (Syntax)

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

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

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

PUSH 2 PUSH 3 ADD
 2 3 +

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>**

$$\langle 2 :: 3 :: \emptyset, \text{ADD} \rangle$$

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

$\langle \emptyset, P2 P3 A P4 M \rangle \rightarrow$

$\langle 2 :: \emptyset, P3 A P4 M \rangle \rightarrow$

$\langle 3 :: 2 :: \emptyset, A P4 M \rangle \rightarrow$

$\langle 5 :: \emptyset, P4 M \rangle \rightarrow$

$\langle 4 :: 5 :: \emptyset, M \rangle \rightarrow$

$\langle 20 :: \emptyset, \epsilon \rangle \checkmark$

20

Recap: Stack-Based Languages

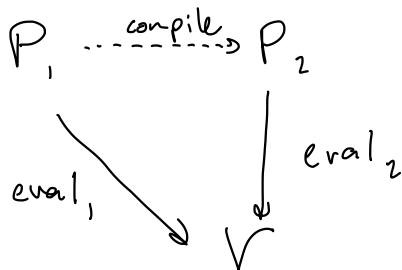
Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

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).

Simple case: *every arithmetic expression can be represented as an equivalent expression in reverse polish notation.*

Bytecode Interpreters

A **bytecode interpreter** is an implementation of a virtual machine with a simple-command based language where each operation is mapped to a *byte*. This is called the **code** of the operation, or the **opcode**

The primary benefits are simplicity and portability. There's no parser, there's no fussing with string

Let's take a quick look at some code

Example (Compilation, by Intuition)

$$4 * (2 + 3)$$

compile

$$4 * (2 + 3) \dashrightarrow \langle \phi, p_3 \ p_2 \ A \ p_4 \ M \rangle$$

$$\downarrow$$
$$\langle 3 :: \phi, p_2 \ A \ p_4 \ M \rangle$$

$$\downarrow$$
$$\langle 2 :: 3 :: \phi, A \ p_4 \ M \rangle$$

$$\langle 5 :: \phi, p_4 \ M \rangle$$

$$\langle 4 :: 5 :: \phi, M \rangle$$

$$\downarrow$$
$$\langle 20 :: \phi, \epsilon \rangle$$

$$4 * 5$$

20

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.

Outline

Recap: Stack-Based Languages

Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

(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)

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

env

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)

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

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

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

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

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

(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

$\langle \emptyset, \emptyset, P2 \text{ As } x \ P3 \text{ As } y \ Lx \ Ly \ A \rangle \rightarrow$

$\langle 2 :: \emptyset, \emptyset, \dots \rangle \rightarrow$

$\langle \emptyset, \{x \mapsto 2\}, \dots \rangle \rightarrow$

$\langle 3 :: \emptyset, \{x \mapsto 2\}, \dots \rangle \rightarrow$

$\langle \emptyset, \{x \mapsto 2, y \mapsto 3\}, \dots \rangle \rightarrow$

$\langle 2 :: \emptyset, \{\dots 3\}, \dots \rangle \rightarrow$

$\langle 3 :: 2 :: \emptyset, \{\dots 3\}, \dots \rangle \rightarrow \langle 5 :: \emptyset, \{\dots 3\}, \epsilon \rangle$

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
```

Answer: *closures*. This is also how we deal with functions. (feel free to chat with me after if you're interested)

Outline

Recap: Stack-Based Languages

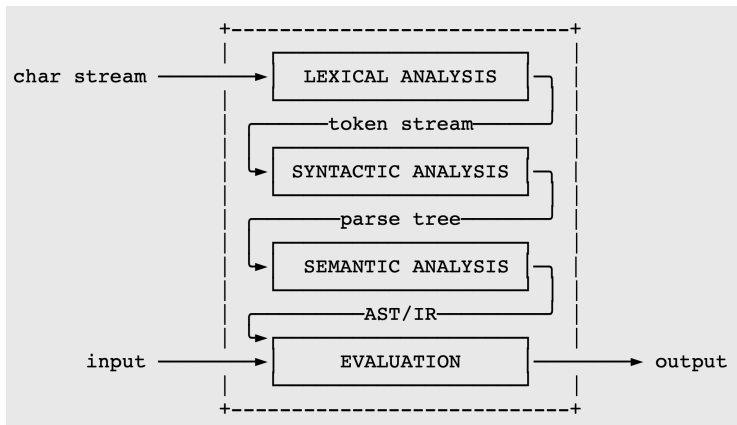
Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

The Pipeline



OCaml's interpretation/compilation pipeline is *roughly* what we've implemented (with a couple more steps). Let's take a **look**

The OCaml Bytecode Compiler

OCaml has an underlying `bytecode interpreter` for a specially designed `Caml virtual machine` (designed by Xavier Leroy in 1990 in his master's thesis).

It's like the one we just looked at but *much* more complicated. Let's take a **look**

OCaml code is transformed into bytecode and then run with `ocamlrun`

Let's do a quick demo

An Aside: The OCaml Native Compiler

If you want *fast* OCaml code, OCaml has a native compiler. This means it will generate assembly language

It works *completely differently* from the bytecode compiler

But it means it's possible to generate some pretty competitive code

If you're interested I recommend taking a look at [the chapter](#) in Real-World OCaml on the compiler backend

Outline

Recap: Stack-Based Languages

Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

Directions for Exploration

- ▶ More OCaml
 - ▶ More advanced module usage
 - ▶ Functional Data Structures
 - ▶ Real-World OCaml
 - ▶ Jane Street LINK
- ▶ More PL
 - ▶ Come learn Rust (and linear types) with me next semester!
 - ▶ Learn Haskell, Elm, Scala?
- ▶ More Math/Type Theory
 - ▶ Go learn more about grammars with Professor Stoughton next semester!
 - ▶ Go learn about session types with Professor Das next semester!
 - ▶ Category theory (functors, monads, comonads)
 - ▶ Logic
 - ▶ Type theory
- ▶ More Computers
 - ▶ Compilers (e.g., LLVM)
 - ▶ Formal methods

Fill out a Course Eval(!)