Compilation II: In Practice

CAS CS 320: Principles of Programming Languages

December 5, 2024 (Lecture 25)

Practice Problem

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

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

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

$$\cdot \vdash \lambda x.\,\lambda y.\,\lambda z.\; xz(yz) : \alpha \to \beta \to \gamma \to \varepsilon \dashv \mathcal{C}$$

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

$$\{\ldots\} \vdash xz : \delta \dashv \alpha \doteq \gamma \to \delta$$

$$\{\ldots\} \vdash x : \alpha \dashv \emptyset$$

$$\{\ldots\} \vdash z : \gamma \dashv \emptyset$$

$$\{\ldots\} \vdash yz : \eta \dashv \beta \doteq \gamma \to \eta$$

$$\{\ldots\} \vdash y : \beta \dashv \emptyset$$

$$\{\ldots\} \vdash z : \gamma \dashv \emptyset$$

$$\mathcal{C} = \begin{array}{l} \delta \doteq \eta \to \varepsilon \\ \alpha \doteq \gamma \to \delta \\ \beta \doteq \gamma \to \eta \end{array}$$

$$T = \alpha \to \beta \to \gamma \to \varepsilon \qquad C =$$

$$\nu \doteq t \qquad \cancel{\delta \doteq \eta \to \varepsilon}$$

$$S = \{\ \delta \mapsto \eta \to \varepsilon, \qquad \nu \doteq t \qquad \cancel{\alpha \doteq \gamma \to \eta \to \varepsilon}$$

$$\alpha \mapsto \gamma \to \eta \to \varepsilon \qquad \nu \doteq t \qquad \cancel{\beta \doteq \gamma \to \eta}$$

$$\beta \mapsto \gamma \to \eta\ \}$$

$$\alpha \to \beta \to \gamma \to \varepsilon \xrightarrow{\delta \mapsto \eta \to \varepsilon} \alpha \to \beta \to \gamma \to \varepsilon$$

S-combinator:

$$\xrightarrow{\alpha \to \gamma \to \eta \to \varepsilon}$$

$$\lambda x. \lambda y. \lambda z.\ x z (y z) \qquad (\gamma \to \eta \to \varepsilon) \to \beta \to \gamma \to \varepsilon$$

$$\xrightarrow{\beta \mapsto \gamma \to \eta} (\gamma \to \eta \to \varepsilon) \to (\gamma \to \eta) \to \gamma \to \varepsilon$$

$$\boxed{\forall \gamma. \forall \eta. \forall \varepsilon. (\gamma \to \eta \to \varepsilon) \to (\gamma \to \eta) \to \gamma \to \varepsilon}$$

$$\cdot \vdash \lambda x. \lambda y. x : \tau \to ($$

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(!)

Outline

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)

<prog> ::= {<com>}

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

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

PUSH 2 PUSH 3 ADD

2 3 +

Arithmetic (Values and Configurations)

state: $\langle \underbrace{\boxed{\mathcal{S}}}_{\text{stack}}, \underbrace{\boxed{\mathcal{P}}}_{\text{program}} \rangle$

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>

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

Arithmetic (Small-step Semantics)

$\langle \emptyset, \text{PUSH } 2 \ \text{ADD} \rangle \rightarrow \bot$

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

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

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

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

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

Example (Evaluation)

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

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

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

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

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

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

$$\langle 20 :: \emptyset, \varepsilon \rangle \checkmark \qquad \boxed{20}$$

Outline

Recap: Stack-Based Languages

Compilation/Bytecode Interpretation

Variables

OCaml Backend

What's next?

Compilation

$$
\begin{tikzcd}
P \arrow[r, dotted] \arrow[dr, "eval_1"'] & P' \arrow[d, "eval_2"] \\
 & v
\end{tikzcd}
$$

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)
```

$4 * (2 + 3) \longrightarrow \langle \emptyset, \ P\,3\ P\,2\ A\ P\,4\ M \rangle$

$\langle 3 :: \emptyset, \ldots \rangle$

$\langle 2 :: 3 :: \emptyset, \ldots \rangle$

$\langle 5 :: \emptyset, \ldots \rangle$

$\langle 4 :: 5 :: \emptyset, \ldots \rangle$

$\langle 20 :: \emptyset, \varepsilon \rangle$ ✓

$4 * 5$

20 ✓

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.

Outline

(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)

$$\langle\ \mathcal{S}\ ,\ \underbrace{\boxed{\mathcal{E}}}_{\text{env.}}\ ,\ \mathcal{P}\ \rangle$$

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)

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

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

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

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

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

(Immutable) Variables (Small-step Semantics)

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

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

lookup

Example

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

$\langle \emptyset, \emptyset, P\ 2\ As\ x\ P\ 3\ As\ y\ L\ x\ L\ y\ A \rangle$

$\langle 2 :: \emptyset, \emptyset, As\ x\ P\ 3\ As\ y\ L\ x\ L\ y\ A \rangle \to$

$\langle \emptyset, \{x \mapsto 2\}, P\ 3\ As\ y\ L\ x\ L\ y\ A \rangle \to$

$\langle 3 :: \emptyset, \{x \mapsto 2\}, \ldots \rangle \to$

$\langle \emptyset, \{x \mapsto 2, y \mapsto 3\}, L\ x\ L\ y\ A \rangle \to$

$\langle 2 :: \emptyset, \{\ldots\}, L\ y\ A \rangle \to \langle 3 :: 2 :: \emptyset, \{\ldots\}, A \rangle$

$\to \langle 5 :: \emptyset, \{\ldots\}, \varepsilon \rangle$

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

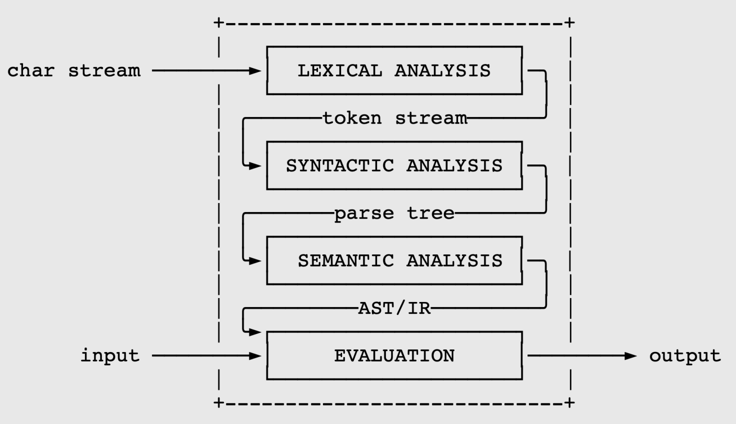
Compilation/Bytecode Interpretation

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 tranformed into bytecode and the 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

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(!)