Assignment #3:
Language Design and Implementation

CSCI 5535 / ECEN 5533: Fundamentals of Programming Languages

Spring 2018: Due Friday, March 9, 2018

This homework has two parts. The first asks you to consider the relationship between a denotational formalization and an operational one. The second asks you to extend your language implementation in OCaml to further gain experience translating formalization to implementation.

1 Denotational Semantics: IMP

Recall the syntax chart for IMP:

Typ	τ	::=	`num`	`num`	numbers
			`bool`	`bool`	booleans
Exp	e	::=	$\texttt{addr}[a]$	a	addresses (or "assignables")
			$\texttt{num}[n]$	n	numeral
			$\texttt{bool}[b]$	b	boolean
			$\texttt{plus}(e_1;e_2)$	$e_1 + e_2$	addition
			$\texttt{times}(e_1;e_2)$	$e_1 * e_2$	multiplication
			$\texttt{eq}(e_1;e_2)$	$e_1 == e_2$	equal
			$\texttt{le}(e_1;e_2)$	$e_1 <= e_2$	less-than-or-equal
			$\texttt{not}(e_1)$	$!e_1$	negation
			$\texttt{and}(e_1;e_2)$	$e_1 \,\&\&\, e_2$	conjunction
			$\texttt{or}(e_1;e_2)$	$e_1 \,\|\|\, e_2$	disjunction
Cmd	c	::=	$\texttt{set}[a](e)$	$a := e$	assignment
			`skip`	`skip`	skip
			$\texttt{seq}(c_1;c_2)$	$c_1;\, c_2$	sequencing
			$\texttt{if}(e;c_1;c_2)$	$\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditional
			$\texttt{while}(e;c_1)$	$\texttt{while}\ e\ \texttt{do}\ c_1$	looping
Addr	a				

As before, addresses a represent static memory store locations and are drawn from some unbounded set Addr and all memory locations only store numbers. A store σ is thus a mapping from addresses to numbers, written as follows:

$$\text{Store} \quad \sigma \quad ::= \quad \cdot \mid \sigma, a \hookrightarrow n$$

The semantics of IMP is as a formalized in the previous assignment operationally. In this section, we will consider a denotational formalization.

The set of values Val are the disjoint union of numbers and booleans:

$$\mathsf{Val} \quad v ::= \texttt{num}[n] \mid \texttt{bool}[b] \,.$$

1.1. (a) Formalize the dynamics of IMP as two denotational functions.

$$\begin{array}{lcl} [\![\cdot]\!] & : & \mathsf{Exp} \to (\mathsf{Store} \rightharpoonup \mathsf{Val}) \\ [\![\cdot]\!] & : & \mathsf{Cmd} \to (\mathsf{Store} \rightharpoonup \mathsf{Store}) \end{array}$$

(b) Prove that your denotational definitions coincide with your operational ones.

i. State the lemma that your definitions for expressions coincide.

ii. Prove the equivalence of your definitions for commands, that is,
$(\sigma, \sigma') \in [\![c]\!]$ if and only if $\langle \sigma, c \rangle \Downarrow \sigma'$.
Begin by copying your definition of $\langle \sigma, c \rangle \Downarrow \sigma'$ from your previous homework submission.

1.2. **Manual Program Verification**. Prove the following statement about the denotational semantics of IMP.

If $[\![\texttt{while}\; e\, \texttt{do}\; a := a + 2]\!]\, \sigma \;=\; \sigma'$ such that $\text{even}(\sigma(a))$, then $\text{even}(\sigma'(a))$

Unlike in the previous assignment, this time you should use your denotational semantics for the proof. *Hint*: your proof should proceed by mathematical induction.

2 Comparing Operational and Denotational Semantics

Regular expressions are commonly used as abstractions for string matching. Here is an abstract syntax for regular expressions:

r	$::=$	‘c’	singleton – matches the character c
	$\mid$	empty	skip – matches the empty string
	$\mid$	$r_1\; r_2$	concatenation – matches r_1 followed by r_2
	$\mid$	$r_1 \mid r_2$	or – matches r_1 or r_2
	$\mid$	$r*$	Kleene star – matches 0 or more occurrences of r
	$\mid$	.	matches any single character
	$\mid$	[‘c_1’–‘c_2’]	matches any character between c_1 and c_2 inclusive
	$\mid$	$r+$	matches 1 or more occurrences of r
	$\mid$	$r?$	matches 0 or 1 occurrence of r

We will call the first five cases the *primary* forms of regular expressions. The last four cases can be defined in terms of the first five. We also give an abstract grammar for strings (modeled as lists of characters):

s	$::=$	$\cdot$	empty string
	$\mid$	cs	string with first character c and other characters s

We write “`bye`" as shorthand for `bye`$\cdot$.

We introduce the following big-step operational semantics judgment for regular expression matching:

$$\vdash r \text{ matches } s \text{ leaving } s'$$

The interpretation of the judgment is that the regular expression r matches some prefix of the string s, leaving the suffix s' unmatched. If $s' = \cdot$, then r matched s exactly. For example,

$$\vdash \texttt{`h'}(\texttt{`e'}+) \text{ matches “}\texttt{hello}\text{" leaving “}\texttt{llo}\text{"}$$

Note that this operational semantics may be considered *non-deterministic* because we expect to be able to derive all three of the following:

$$\vdash (\texttt{`h'} \mid \texttt{`e'})* \text{ matches “}\texttt{hello}\text{" leaving “}\texttt{ello}\text{"}$$
$$\vdash (\texttt{`h'} \mid \texttt{`e'})* \text{ matches “}\texttt{hello}\text{" leaving “}\texttt{hello}\text{"}$$
$$\vdash (\texttt{`h'} \mid \texttt{`e'})* \text{ matches “}\texttt{hello}\text{" leaving “}\texttt{llo}\text{"}$$

We leave the rules of inference defining this judgment unspecified. You may consider giving this set of inference rules an optional exercise.

Instead, we will use *denotational semantics* to model the fact that a regular expression can match a string leaving many possible suffixes. Let Str be the set of all strings, let $\wp(\mathsf{Str})$ be the powerset of Str, and let RE range over regular expressions. We introduce a semantic function:

$$[\![\cdot]\!] : \mathsf{RE} \to (S \to \wp(S))$$

The interpretation is that $[\![r]\!]$ is a function that takes in a string-to-be-matched and returns a set of suffixes. We might intuitively define $[\![\cdot]\!]$ as follows:

$$[\![r]\!](s) = \{s' \mid \vdash r \text{ matches } s \text{ leaving } s'\}$$

In general, however, one should not define the denotational semantics in terms of the operational semantics. Here are two correct semantic functions:

$$\begin{aligned} [\![\texttt{`}c\texttt{'}]\!](s) &= \{s' \mid s = \texttt{`}c\texttt{'} :: s'\} \\ [\![\mathsf{empty}]\!](s) &= \{s\} \end{aligned}$$

2.1. Give the denotational semantics functions for the other three primal regular expressions. Your semantics functions *may not* reference the operational semantics.

2.2. We want to update our operational semantics for regular expressions to capture multiple suffixes. We want our new operational semantics to be deterministic—it should give the same answer as the denotational semantics above. We introduce a new judgment as follows:

$$\vdash r \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \texttt{`}c\texttt{'} \text{ matches } s \text{ leaving } \{s' \mid s = \texttt{`}c\texttt{'} :: s'\}} \qquad \frac{}{\vdash \mathsf{empty} \text{ matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash r_1 \text{ matches } s \text{ leaving } S_1 \qquad \vdash r_2 \text{ matches } s \text{ leaving } S_2}{\vdash r_1 \mid r_2 \text{ matches } s \text{ leaving } S_1 \cup S_2}$$

Do one of the following:

- *Either* give operational semantics rules of inference for $r*$ and $r_1\ r_2$. Your operational semantics rules may *not* reference the denotational semantics. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash r\ \mathsf{matches}\ x\ \mathsf{leaving}\ y\}$. Each inference rule must have a finite and fixed set of hypotheses.
- *Or* argue in one or two sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but "wrong" rules of inference and show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research in any area is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible."

3 Implementation: General Recursion and Polymorphism

In this section, we will reformulate language ETPS so that it admits general recursion (and thus non-terminating programs) and parametric polymorphism.

3.1. Adapt your language ETPS with general recursion. That is, replace the language T portion (primitive recursion with natural numbers) with language PCF from Chapter 19 of *PFPL* (general recursion with natural numbers).

3.2. Add recursive types (i.e., language FPC from Chapter 20 of *PFPL*). While type `nat` of natural numbers is definable in FPC, leave the primitive `nat` in for convenience in testing.

3.3. Add parametric polymorphism (i.e., System F from Chapter 16 of *PFPL*).

Explain your testing strategy and justify that your test cases attempt to cover your code as thoroughly as possible (e.g., they attempt to cover different execution paths of your implementation with each test). Write this explanation as comments alongside your test code.

Follow the "Translating a Language to OCaml" guidance from the previous homework assignment.