MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 2

Problem 1: Operational Semantics: Postfix + {sdup}

Alyssa P. Hacker extended the PostFix language with a new command called `sdup`: `smart dup`. This allows us to compute $square(x) = x^2$ without hurting the termination property of PostFix programs. The informal semantics for `sdup` is as follows: duplicate the top of the stack if it is a number or a command sequence that doesn't contain `sdup`; otherwise, report an error.

Formally, the operational semantics has been extended with the following two transition rules:

$$\langle \texttt{sdup} \,.\, Q_{rest},\ N \,.\, S\rangle \Rightarrow \langle Q_{rest},\ N \,.\, N \,.\, S\rangle \qquad [\textit{sdup-numeral}]$$

$$\langle \texttt{sdup} \,.\, Q_{rest},\ Q \,.\, S\rangle \Rightarrow \langle Q_{rest},\ Q \,.\, Q \,.\, S\rangle \text{ where } \neg(\textit{contains\_sdup}\ Q) \qquad [\textit{sdup-sequence}]$$

$\textit{contains\_sdup} : \textit{Command}^* \rightarrow$ Bool is a helper function that takes a sequence of commands and checks whether it contains `sdup` or not (yes, *contains_sdup* handles even nested sequences of commands)

As a new graduate student in Alyssa's AHRG (Advanced Hacking Research Group), you were assigned to give a proof that all PostFix + {`sdup`} programs terminate. However, you are not alone! Alyssa already took care of most of the mathematical weaponry:

Consider the product domain $P = Nat \times Nat$ (as usual, Nat is the set of natural numbers, starting with 0). On this domain, we define the relation $<_P$ as follows:

Definition 1 (lexicographic order) $\langle a_1, b_1\rangle <_P \langle a_2, b_2\rangle$ *iff:*

a. $a_1 < a_2$ *or*

b. $a_1 = a_2$ *and* $b_1 < b_2$.

E.g. $\langle 3, 10000\rangle <_P \langle 4, 0\rangle$, $\langle 5, 2\rangle <_P \langle 5, 3\rangle$.

Definition 2 *A* strictly decreasing chain *in P is a finite or infinite sequence of elements* $p_1, p_2, \ldots$ *such that* $p_i \in P, \forall i$ *and* $p_{i+1} <_P p_i, \forall i$.

After a long struggle, Alyssa proved the following lemma for you:

Lemma 1 *There is no infinite strictly decreasing chain in P.*

Give a rigorous proof that each PostFix + {`sdup`} program terminates by using a cleverly defined energy function $\mathcal{ES}_{config}$. *Hint:* Each transition of Postfix reduces the energy function $\mathcal{E}_{\text{config}}$ you saw in class. Try to see what is reduced by the two new rules, and how you can combine these two things into a single energy function.

Note: If you need to use some helper functions that are intuitively easy to describe but tedious to define (e.g. *contains_sdup*), just give an informal description of them.

Problem 2: Operational and Denotational Semantics

In this problem we ask you to develop both an operational and denotational semantics for the language EL. The grammar for EL programs is given on page 24 of Course Notes 1.

a. Give a structured operational semantics for EL. Remember, there are five parts to an operational semantics (see section 3.2 of Course Notes 1).

   Your Program domain should be the set Num-Exp $\times$ Numeral, that is, an EL expression paired with a value for the input variable. The Answer domain should be Numeral + Error (division by zero should cause an error in EL). Technically, you will have to define *two* rewriting relations for your semantics, one for Num-Exps and one for Bool-Exps.

   You may take as given the functions

   calculate : Arithmetic-operator → Numeral → Numeral → Numeral
   logical : Logical-operator → Truth-Value → Truth-Value → Truth-Value
   relational : Relational-operator → Numeral → Numeral → Truth-Value

   which evaluate arithmetic, logical, and relational operators in the usual way, e.g.,

   (*calculate* `+` `3.2` `18.4`) $=$ `21.6`,
   (*logical* `or` `true` `false`) $=$ `true`,
   (*relational* `<` `26` `4.1`) $=$ `false`,

   etc.

b. Define a denotational semantics for EL. You should give three meaning functions, $\mathcal{P}$ for Programs, $\mathcal{NE}$ for Num-Exps, and $\mathcal{BE}$ for Bool-Exps. The meaning of a Num-Exp, $\mathcal{NE}[\![N]\!]$, should be a function from the value of the input variable to the final answer, and similarly, the meaning of a Bool-Exp, $\mathcal{BE}[\![B]\!]$, should be a function from the value of the input variable to a boolean result.

   You may take as given the meaning functions $\mathcal{M}$, $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{L}$, for Numerals, Arithmetic-operators, Relational-operators, and Logical-operators respectively, **except** that you must clearly specify any error-related behavior. ($\mathcal{A}$, $\mathcal{R}$, and $\mathcal{L}$ correspond to the functions *calculate*, *relational*, and *logical* from the operational semantics.)

   You must provide the definition of all semantic domains, and the signatures of the meaning functions $\mathcal{P}$, $\mathcal{NE}$, $\mathcal{BE}$, $\mathcal{M}$, $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{L}$.

Programming Exercises

Problem 3: Safe Transformations Lab

Do exercise 2.19 (a) and (b) from the Course Notes, Scheme Supplement, page 73. **In addition**, prove that your rules are terminating. We require that your rules be safe; however you do not have to *prove* that your rules are safe. For this exercise, we will not require that your rules be confluent.

The exercise states that your rules should be "local." A rough guideline is that antecedents should match sequences of no more than three or four commands to be considered local. Any rule matching the entire test program,

```
((swap exec swap exec) (1 sub) swap (2 mul) swap 3 swap exec),
```

will be considered "non-local," and therefore an incorrect answer.

We have provided some useful code in `code/ps2/` in the course directory.