Thinking Programs: Exercises

Chapter 7: Programming Languages

In the following exercises, when defining a denotational semantics, you may use either functional or relational style (or both to decide which style seems more appropriate).

1. Define the denotational semantics (abstract syntax, semantic algebras, valuation functions) of a language for maintaining the database of a shop. The language shall allow to enter a sequence of commands that perform the following actions:

   - Enter a new article A with price P and quantity N into the store.
   - Sell N copies of article A at its current price.
   - Add N copies of article A to the store.
   - Get the current price of article A.
   - Change the price of article A to a new value P.
   - Determine the quantity of sales of article A and its average sale price P.

   An example of a program in this language is

   ```
   enter Milk 1.1 10; enter Bread 2.7 5; sell Milk 5; sell Bread 2;
   Add Milk 20; price Bread; newPrice Bread 2.4; sales Milk
   ```

   which produces system output

   ```
   okay; okay; okay; okay; okay; 2.7; okay; 5*1.1;
   ```

   Hint: the language must maintain a "database" that maps each article A to its current price, available quantity, number of copies sold, and the total price achieved in these sales. Each command has to read and potentially update the database and to deliver an "answer" (which may be e.g. a boolean value indicating the success of the action or the sales data of an article); the signature of the valuation function for commands is thus of form

   $$[\![\cdot]\!] : \mathit{Command} \to \mathit{Database} \to (\mathit{Database} \times \mathit{Answer})$$

   The result of the overall "command session" is correspondingly (the final state of) the database and the sequence of answers produced.

2. Take the following language of evaluation sessions S, expressions E, numerals N, and identifiers I:

   $$S ::= E$$
   $$E ::= I \mid N \mid E_1 + E_2 \mid I := E \mid E_1; E_2$$

This language contains an expression $I := E$ which returns the value of E and updates as a side-effect the content of variable I with that value. The expression $E_1; E_2$ evaluates first E_1, then E_2 and returns the value of E_2.

An example expression in this language is

```
X:=5; X+(Y:=X); (Z:=Y;Z+1)+Z
```

whose value is 11. Please note that the evaluation of every expression updates the store and returns the value. An evaluation session starts with an arbitrary (internally allocated) store. The result of an evaluation session is a value. In the denotational semantics, we thus want a valuation function $[\![.]\!]$ such that $[\![S]\!]$ is a natural number.

a) Give the language a type system that ensures that a variable is only read *after* it has been assigned a value.

b) Give domains S and E a denotational semantics; please note that the semantics of an expression $[\![E]\!](s)$ is a tuple $\langle n, s' \rangle$ which consists of the value n of E when evaluated in store s and of the new store s' that results from this evaluation.

c) Correspondingly define a big-step operational semantics for S and E with transitions $\langle E, s \rangle \rightarrow \langle n, s' \rangle$ (expression E evaluated in store s yields number n and store s'). Formulate the statement "the operational semantics of E corresponds to its denotational semantics"; do the same for S.

d) Define a small-step operational semantics for S and E. Formulate the statement "the small-step operational semantics of E corresponds to its big-step semantics"; do the same for S.

e) Prove the statements you have formulated above.

3. Modify our command language by introducing a domain of expresssions E:

   $E ::= I \mid N \mid E_1 = E_2 \mid E_1 + E_2 \mid E_1 \wedge E_2 \mid \textbf{exec } C \textbf{ result } E$

   Here the **exec** expression executes C and then returns the result of the evaluation of E. Correspondingly, the evaluation of an expression may alter the store.

   Furthermore, the grammar of a command C is modified by replacing every occurrence of a term T or formula F by an expression E.

   a) Give the language a type system that ensures that wherever the original language had a term T, the new language has an expression E that denotes an integer, and whenever the original language had a formula F, the new langauge has an expression E that denotes a truth value.

   b) Define a denotational semantics for expressions and commands.

   c) Define a big-step operational semantics for expressions and commands. Formulate the statement "the operational semantics of expression E corresponds to its denotational semantics"; do the same for command C.

d) Define a small-step operational semantics for expressions and commands. Formulate the statement "the small-step operational semantics of expression E corresponds to its big-step semantics"; do the same for command C.

e) Prove the statements you have formulated above.

4. Extend the command language by a C/Java-style do-while loop:

   $C ::= \ldots \mid \textbf{do}\ C\ \textbf{while}\ F$

   Here F is a formula of our formula language. The loop first executes C and then evaluates F. If F is true, this process is repeated; otherwise the loop terminates.

   a) Define denotational semantics for this command.

   b) Define a big-step operational semantics for this command. Prove that the big-stop operational semantics of the command corresponds to its denotational semantics.

   c) Define a small-step operational semantics for this language. Prove that the small-step operational semantics of the command corresponds to its big-step semantics.

   Repeat the exercise for a version of the loop where the formula F is replaced by an expression E as described in Exercise 3; the evaluation of this expression does not only return a truth value but may also alter the store.

5. Extend our command language by a high-level loop construct

   $C ::= \ldots \mid \textbf{for}\ I\ \textbf{from}\ T_1\ \textbf{to}\ T_2\ \textbf{by}\ T_3\ \textbf{do}\ C$

   where the evaluation of term T yields an integer. The loop iteratively executes the loop body C with the value of variable I set subsequently to $i_1, i_1 + i_3, i_1 + 2 \cdot i_3, \ldots, i_1 + k \cdot i_3$ where i_1, i_2, i_3 are the values of T_1, T_2, T_3, respectively, and $i_1 + k \cdot i_3$ is the largest value less than or equal i_2 (if $i_1 > i_2$, the loop is not executed at all). After the execution of the loop, I has the same value that it had before the (i.e., I is only temporarily assigned).

   Extend the type system of the language to consider the (re)declaration of I as a variable of sort `int` within the loop. Perform the same tasks as in Exercise 4, once using the grammar above, once replacing every occurrence of a term T by an expression E whose evaluation may alter the store.

6. Extend our command language by a C/Java-style for-while loop

   $C ::= \ldots \mid \textbf{for}\ (C_1; F; C_2)\ C_3$

   where first C_1 is executed, then B is evaluated. If the result is "true", C_3 and C_2 are executed and then B is evaluated again.

   Perform the same tasks as in Exercise 4, once using F, once replacing F by a Boolean expression E whose evaluation may alter the store.

7. Extend our command language as follows:

   $C ::= \ldots \mid \textbf{print}\ T \mid \textbf{read}\ I$
   $F ::= \ldots \mid \textbf{okay}$

Command **print** T prints the value of term T to the output stream; command **read** I reads the next value from the input stream and writes it into variable I. If the value read from the stream was not of the type expected for variable I, the value of I remains unchanged. The value returned by the formula **okay** indicates whether the last input operation was successful or not.

Give this language a semantics by extending the program state (which currently consists of a mapping of variables to values) by an infinite stream of input values (type $Value^\omega$), a finite stream of output values (type $Value^*$), and a Boolean value that indicates the success of the last input operation. Most commands leave these streams unchanged; the component **print** T, however, extends the output stream by one value, while the command **read** I removes one value from the input stream. Define the semantics in denotational style as well as in big-step operational style and show their correspondence. To test the validity of inputs, you may use the formula $v \in A(S)$ to test whether value v is in the given interpretation $A(S)$ of sort S.

8. Extend our command language by the following commands:

   $C ::= \ldots \mid \textbf{proc}\ I = C \mid \textbf{call}\ I$

   The command **proc** $I = C$ defines a (parameter-less) procedure I by command C; the command **call** I invokes this procedure which causes the execution of C. Thus the effect of a command is not any more only to alter the store but also to alter a *procedure environment*, i.e., a mapping of identifiers (procedure names) to the semantics of commands (the semantics of the procedure bodies).

   a) Give the language a type system that ensures that only defined procedures are called.

   b) Define a denotational operational semantics.

   c) Define a big-step operational semantics. Prove that the big-stop operational semantics corresponds to its denotational semantics.

   d) Define a small-step operational semantics. Prove that the small-step operational semantics corresponds to its big-step semantics.

9. Extend our command language by the following commands:

   $C ::= \ldots \mid \textbf{proc}\ I_1(I_2) = C \mid \textbf{call}\ I(T)$

   The command **proc** $I_1(I_2) = C$ defines a procedure I_1 with a single value parameter I_2 and body C; the command **call** I invokes this procedure which causes the execution of C where I_2 is assigned the value of term T. Thus the effect of a command is not any more only to alter the store but also to alter a *procedure environment*, i.e., a mapping of identifiers (procedure names) to functions that map values (procedure arguments) to the semantics of commands (the semantics of the procedure bodies).

   Perform the same tasks as in Exercise 8.

10. A *relational database* maps names to tables where a table consists of a sequence of *column names* and a set of *rows* where every row is a tuple of values, one for each column name. The database language SQL defines various commands on relational databases, in particular:

- Command `CREATE TABLE` creates a new table in the database.
- Command `DROP TABLE` drops a table from the database.
- Command `UPDATE` updates some rows in a table.
- Command `INSERT INTO` inserts new rows into a table.
- Command `DELETE FROM` deletes rows from a table.
- Command `SELECT` queries the database for information.

Lookup the definition of SQL and define an abstract syntax of simplified versions of above commands (assume that there is only a single type of entries and reduce the commands to some minimal form). From the commands above, the most complex one is the `SELECT` command. Define here a version of the command that allows queries of the form

```
SELECT table1.column1, table2.column2, ...
FROM table1
INNER JOIN table2, ON table1.column1 = table2.column2
...
WHERE table.column >= 0 AND ... >= ...
```

This query first constructs an inner join of the table listed in the `FROM` clause with each table subsequently listed in an `INNER JOIN` clause (i.e., every row of the first table is concatenated with every row in the second table provided that the two columns listed in the `ON` clause are identical); from the result only those combined rows are considered that satisfy the conditions in the `WHERE` clause (which can be conjunctions of atomic formulas referring to table columns and to constant values).

Let a "session" be a sequence of the first four kinds commands (that starting with an empty database update the database) followed by a second sequence of `SELECT` commands (that query the database). Define the abstract syntax of a session and give it a type system that ensures that all references to table names and column names in all commands are valid, i.e., refer to existing tables and existing columns in these tables only.

Then give the langauge a denotational semantics where the first four commands denote mappings from databases to databases and the `SELECT` command denotes a a mapping from a database to a set of rows. The semantics of a session consists of the database resulting from the first kind of commands and the sequence of results of the `SELECT` commands.

11. Consider the programming language with procedures introduced in Definitions 7.1 and 7.11 of the manuscript "Thinking Programs". However, rather than having separate syntactic domains for formulas and terms, have a *single* domain of "expressions" that can denote integers (with literals, operations $+$, $*$, unary and binary $-$, $/$, $=$ and $\leq$) or booleans (with truth literals and the logical connectives "and", "or", and "not"). *Do not separate this domain into two domains.*

    Then perform the following tasks:

    a) Adapt/extend the grammar for the abstract syntax of the language and its type system (presented in Figures 7.1 and 7.18) to accommodate your domain of expressions.

b) Implement datatypes for the abstract syntax of this language including a function to print out a linear representation of programs.

c) Implement a type checker for this language, i.e., a function that accepts as input the abstract syntax for a program and indicates (by its return value or by throwing an exception) whether the program is well-typed.

See Sections "Abstract Syntax Trees in OCaml" and "Denotational Semantics in OCaml" for inspiration on how to develop such implementations.

12. Consider your programming language with expressions and procedures introduced in Exercise 11 and perform the following tasks:

    - Adapt/extend the denotational semantics for the language presented in Figures 7.2/7.3, 7.6, and 7.20 to accommodate your domain of expressions.
    - Implement your denotational semantics as a function that takes as its argument a sequence of values (the initial values of the program parameters) and returns as its result such a sequence (the final values of the program parameters). The function may abort if the evaluation of an expression is not well-defined (e.g., division by zero).
    - (Optional) Adapt/extend your type checker to include the translation of variables to addresses depicted in Figure 7.22 by annotating every variable in the syntax tree with an address. Adapt/extend your implementation of the denotational semantics to make use of this annotation as indicated in Figure 7.23.

13. Consider the programming language with expressions introduced in Exercise 11 (you may omit procedures) and perform the following tasks:

    - Adapt/extend the abstract syntax of machine instructions (introduced in Definition 7.9) to include all machine instructions required for evaluating the expressions of your language. Implement a datatype for the abstract syntax of this extended machine language including a function to print out a linear representation.
    - Adapt/extend the small-step operational semantics of the machine language introduced in Figure 7.12 to accommodate the additional instructions. Implement this operational semantics as a function that takes as argument the current configuration of the machine and returns as result the next configuration. The function may abort if the evaluation of an expression is not well-defined (e.g., division by zero). Also implement a function that repeatedly applies the first function as long as the machine configuration indicates that the machine program has not terminated (i.e., the program counter still refers to an instruction in the sequence).
    - (Optional) Adapt/extend the translation of expressions and commands introduced in Figures 7.14 and 7.15 to accommodate your extensions. Implement this translation as a function that takes as argument a command and returns as a result a sequence of machine instructions. Test the translation by executing the generated program.