02141 Computer Science Modelling
Mandatory Assignment 3

Version 3

Operational Semantics and Interpreters

Henrik Pilegaard — hepi@imm.dtu.dk

April 22, 2010

Abstract

The sequence of mandatory assignments in the 2010 edition of 02141 is intended to demonstrate how difficult problems in advanced systems may be addressed using the language-based techniques presented in the course. The fundamental object of study will be a basic imperative programming language called WHILE and you will be asked to develop the following series of tools for this language: A *static program analyser* to decide if variables are used before initialised, a *scanner and parser* framework to recognise and read WHILE programs, and an *interpreter* to execute WHILE programs.

This third mandatory assignment has two parts, which may be completed in any order.

In the first part we shall prove some fundamental properties of a simple while language where the while loop is replaced by a repeat construct. Your task is to specify a structural operational semantics and a natural semantics for the repeat construct and conduct important (and mostly inductive) proofs based on these specifications.

In the second part we shall develop an interpreter for a While variant that extends the While language from assignment 2 with a simple notion of methods that are not allowed to produce side-effects. Your task is to specify a natural semactics for this language, and to implement an interpreter based on your specification.

Technically, the assignment concerns operational semantics and interpreters for programming languages. The theoretical aspects of these

topics are covered in chapters 2-3 of "Semantics with Applications - An Appetizer" by Nielson and Nielson. The practical parts can be completed using the tools from the previous assignments. No new files are provided.

You are expected to submit a working **implementation** as well as a standard **technical report** describing your work on **CampusNet** no later than **11:59pm on Monday the 10th of May**.

Part 1

In this part of the exercise we consider a version of the while language of the textbook where the while-construct itself is replaced by a repeat-construct. Thus the syntax of statements, S, in the language is given by:

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{repeat } S \texttt{ until } b$$

Here the syntax of arithmetic expressions, a, and Boolean expressions, b, are as in the textbook.

Question 1

Specify a natural semantics for the repeat-construct - you are not allowed to rely on the existence of the while-construct in the language. Prove that

$$\texttt{repeat } S \texttt{ until } b$$

is semantically equivalent to

$$S; \texttt{ if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b)$$

(This exercise appears in the textbook as Exercise 2.7.)

Question 2

Specify a structural operational semantics for the repeat-construct; you are not allowed to rely on the existence of the while-construct. (This exercise appears in the textbook as Exercise 2.17.)

Question 3

Using the specifications above prove that the repeat-construct

$$\texttt{repeat}\ S\ \texttt{until}\ b$$

is semantically equivalent to

$$S;\ \texttt{while}\ \neg b\ \texttt{do}\ S$$

You may conduct the proof based on the natural semantics (as in Exercise 2.10 of the textbook) or based on the structural operational semantics (as in Exercise 2.24 of the textbook) - the choice is yours but make clear in your report which semantics you use.

Question 4

Extend the proof of Theorem 2.26 in the textbook to hold for the language with the repeat-construct; you should use the specifications of the semantics developed in the exercises above.

Part 2

Here we shall construct an interpreter for the While language. The aim of the interpreter is to evaluate well-formed While programs.

First, we shall develop a natural semantics for the subset of the While language defined in Appendix A (Basically this is the While language of assignment 2 extended wth a notion of methods). Subsequently we shall use the semantics to implement an interpreter for the language.

Question 1

Assume the following domain for values:

$$v \quad \in \quad \mathbf{Val} = \mathbf{Num}$$

i.e., values are simply numbers.

Then take the domain of variable environments to be:

$$env_V \quad \in \quad \mathbf{Env}_V = \mathbf{Var} \hookrightarrow \mathbf{Loc}$$

i.e. simple mappings from variable names to store locations.

In this context a suitable domain of stores is:

$$sto \quad \in \quad \mathbf{Store} = \mathbf{Loc} \cup \{\mathsf{next}\} \hookrightarrow \mathbf{Val}$$

i.e. simple mappings from store locations to values. Here next is a special token used to hold the next *free* location.

Finally, method environments can be defined as:

$$env_M \quad \in \quad \mathbf{Env}_M = \mathbf{MetID} \to \mathbf{Var} \times \mathbf{Var} \times \mathbf{Stm}$$

to reflect that methods can take only one parameter. Note that this is different from the textbook as our methods do take parameters, but do not allow nested method declarations.

Arithmetic expressions are evaluated according to the judgment

$$env_V, env_M \vdash_{\mathsf{A}} \langle a, sto \rangle \to v$$

and boolean expressions are according to the judgment

$$env_V, env_M \vdash_{\mathsf{B}} \langle a, sto \rangle \to v$$

where we insist that methods are evaluated **without** side effects.

a. Define these judgments in detail, i.e. the operational details of expression evaluation. Example:

$$\frac{\begin{array}{c} env_V, env_M \vdash_{\mathsf{A}} \langle A, sto\rangle \to v \\ \vdash_{\mathsf{D}} \langle S, env_v, sto\rangle \to \langle env'_v, sto'\rangle \\ env'_V[V_1 \mapsto \ell], env_M \vdash_{\mathsf{S}} \langle S, sto'[next \mapsto \ell + 1][\ell \mapsto v]\rangle \to sto'' \\ (sto'' \circ env'_V)(V_2) = v' \end{array}}{env_V, env_M \vdash_{\mathsf{A}} \langle M(A), sto\rangle \to v'}$$

where $sto(next) = \ell$ and $env_M(M) = (V_1, V_2, S)$

Statements, on the other hand, are evaluated according to the judgment

$$env_V, env_M \vdash_{\mathsf{S}} \langle S, sto\rangle \to sto'$$

where we see that statements produce side-effects but not values.

b. Again, define this judgment in detail, i.e. the operational details of statement evaluation. Example:

$$\frac{\begin{array}{c} env_V, env_M \vdash_{\mathsf{S}} \langle S_1, sto\rangle \to sto' \\ env_V, env_M \vdash_{\mathsf{S}} \langle S_2, sto'\rangle \to sto'' \end{array}}{env_V, env_M \vdash_{\mathsf{S}} \langle S_1; S_2, sto\rangle \to sto''}$$

Declarations do not cause any evaluation. Rather they simply cause the appropriate environments (and stores) to be updated/extended with new values. Although we do not have explicit variable declarations we still need variable environments (and stores) to account for the assigned variables. We update these in accordance with a judgment

$$\vdash_{\mathsf{D}} \langle S, env_V, sto\rangle \to \langle env'_V, sto'\rangle$$

c. Define this judgment in detail, i.e. exactly how a variable assignment causes the variable environment and store to be updated.

For method declarations the update is performed by the function

$$\mathsf{upd}_M : \mathbf{MDec} \to \mathbf{Env}_M$$

d. Now, also define this function in detail.

Finally, for programs we introduce judgments of the form

$$\vdash_{\mathsf{P}} P \to sto$$

defined by the single rule

$$\frac{\vdash_{\mathsf{D}} \langle S, [\,], [\,] \rangle \to \langle env_V, sto \rangle \quad env_V, env_M \vdash_{\mathsf{S}} \langle S, sto \rangle \to sto'}{\vdash_{\mathsf{P}} D_M \; S \to sto'}$$

where $env_M = \mathsf{upd}_M(D_M)(env_V)$ and occurences of [ ] denote empty environments/stores.

The format of the method environments and their use in the evaluation judgments of the semantics should enable the definition of recursive and mutually recursive methods.

f. Argue that this is the case.

Question 2

Assume the input of the interpreter to be internal representations of While programs created by (a minimal extension of) your parser.

The data structures required for the interpretation process are implementations of the domains treated in the previous question.

a. As we do not provide these structure you should implement them now.

Once the appropriate data structures are defined they may be used to implement the four semantic functions defined previously.

b. Now, in a sub-directory **Interpreter** of the working directory implement a Java class **Interpreter**. The class should encapsulate the semantic functions and expose a main method that, given the file name of a valid While program, evaluates the program and dumps the final state to a file (or standard out).

In the case of a While program not being well-formed and/or giving rise to run-time errors (like those specified in the last question) it is appropriate for the parser/interpreter to give meaningful error messages.

For the purpose of the report it is a good idea to test and document the central features of your interpreter.

d. Implement a number of test programs and use those to test and document the features of your interpreter.

Abstract Syntax

Syntactic categories:

N	$\in$	**Num**	numerals
V	$\in$	**Var**	variables
A	$\in$	**AExp**	arithmetic expressions
B	$\in$	**BExp**	boolean expressions
S	$\in$	**Stm**	statements
M	$\in$	**MetID**	method identifiers
D_M	$\in$	**MDec**	method declarations
P	$\in$	**Prog**	programs

Abstract syntax:

$$
\begin{array}{rcl}
A & ::= & V \mid N \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 * A_2 \mid M(A) \\
B & ::= & \texttt{true} \mid \texttt{false} \mid A_1 = A_2 \mid A_1 > A_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B_1 \\
S & ::= & \texttt{skip} \mid V{:=}A \mid S_1; S_2 \mid \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \mid \texttt{while } B \texttt{ do } S_1 \texttt{ od} \\
D_M & ::= & M = \{\texttt{input}(V_1); S; \texttt{output}(V_2)\}; D_M \mid \epsilon \\
P & ::= & D_M \; S
\end{array}
$$

Changes and History

Version 2

- Changed wording of Part 1 Question 3 into

  > Using the specifications above prove that the repeat-construct
  >
  > $$\texttt{repeat } S \texttt{ until } b$$
  >
  > is semantically equivalent to
  >
  > $$S; \texttt{while } \neg b \texttt{ do } S$$

Version 3

- Updated the details of functions in Part 2 to more naturally reflect the requirements regarding side-effects. In particular:
  - the type of method environments was changed to

    $$env_M \quad \in \quad \mathbf{Env}_M = \mathbf{MetID} \to \mathbf{Var} \times \mathbf{Var} \times \mathbf{Stm}$$

– the example in Question 1a was changed into

$$\frac{\begin{array}{c} env_V, env_M \vdash_{\mathsf{A}} \langle A, sto\rangle \to v \\ \vdash_{\mathsf{D}} \langle S, env_v, sto\rangle \to \langle env'_v, sto'\rangle \\ env'_V[V_1 \mapsto \ell], env_M \vdash_{\mathsf{S}} \langle S, sto'[next \mapsto \ell + 1][\ell \mapsto v]\rangle \to sto'' \\ (sto'' \circ env'_V)(V_2) = v' \end{array}}{env_V, env_M \vdash_{\mathsf{A}} \langle M(A), sto\rangle \to v'}$$

where $sto(next) = \ell$ and $env_M(M) = (V_1, V_2, S)$

– The wording about statement was changed into:

> Statements, on the other hand, are evaluated according to the judgment
>
> $$env_V, env_M \vdash_{\mathsf{S}} \langle S, sto\rangle \to sto'$$
>
> where we see that statements produce side-effects but not values.

– The type of the method environment update function was changed into

$$\mathsf{upd}_M : \mathbf{MDec} \to \mathbf{Env}_M$$