Concepts of Programming Language Design

Abstract Machines Exercises

Liam O'Connor-Davis
Gabriele Keller

November 6, 2024

1. **Decision Machines**: Suppose we have a language of nested brackets N (where ϵ is the empty string):

$$\frac{}{\epsilon\ N} \quad (N\text{-}1)$$

$$\frac{e\ N}{(e)\ N} \quad (N\text{-}2)$$

$$\frac{e\ N}{\langle e\rangle\ N} \quad (N\text{-}3)$$

$$\frac{e\ N}{[e]\ N} \quad (N\text{-}4)$$

Note that `()()` is not a string in this language.

We developed a simple abstract machine to check if strings are in this language. We set the states for the machine to be simply strings. Initial states are all non-empty strings, and the final state is the empty string. Then, our state transition relation is:

$$\frac{}{(e) \mapsto e} \quad (M\text{-}1)$$

$$\frac{}{[e] \mapsto e} \quad (M\text{-}2)$$

$$\frac{}{\langle e\rangle \mapsto e} \quad (M\text{-}3)$$

   (a) A machine recognises a language if any machine in the state corresponding to a string S will eventually reach a final state if and only if the string S is in the language.
      i. Show that the string `([⟨⟩])` is in the language N, and show that our machine reaches a final state given the same string, i.e: `([⟨⟩])` $\overset{!}{\mapsto} \epsilon$.
      ii. Show that the string `[]()[]` is not in the language juN, and show that our machine reaches a stuck state given the same string, i.e, there exists some stuck state s such that `[]()[]` $\overset{\star}{\mapsto} s$
      iii. **Possibly difficult** Prove that the machine recognises the language N, that is:

α) $s \; \boldsymbol{N} \implies s \overset{!}{\mapsto} \epsilon$. Remember that $\overset{!}{\mapsto}$ is just $\overset{\star}{\mapsto}$, where the second state is a final state. $\overset{\star}{\mapsto}$ of course being the reflexive transitive closure of $\mapsto$, that is:

$$\frac{}{s \overset{\star}{\mapsto} s}\textsc{Refl}^* \qquad \frac{s_1 \mapsto s_2 \quad s_2 \overset{\star}{\mapsto} s_3}{s_1 \overset{\star}{\mapsto} s_3}\textsc{Trans}^*$$

β) $s \overset{!}{\mapsto} \epsilon \implies s \; \boldsymbol{N}$

(b) Suppose that we were unable to efficiently read from both the beginning and end of the string simultaneously (For example, if a tape or a linked list is used to represent the string). This makes our original machine highly inefficient, as each state transition must examine the end of a string for a closing bracket.

We develop a new, stack-based machine that attempts to solve this problem. Our stack consists of three symbols, `P`, `A`, and `B`, one for each type of bracket. The states of the machine are of the form $s \mid e$, where s is a stack and e is a string. Our initial states are all states with an empty stack and a non-empty string, i.e: $\circ \mid e$, our final state is $\circ \mid \epsilon$, and our state transitions are as follows:

$$\frac{}{s \mid (e \mapsto \texttt{P} \triangleright s \mid e}S_1 \qquad \frac{}{s \mid \langle e \mapsto \texttt{A} \triangleright s \mid e}S_2 \qquad \frac{}{s \mid [e \mapsto \texttt{B} \triangleright s \mid e}S_3$$

$$\frac{}{\texttt{P} \triangleright s \mid)e \mapsto s \mid e}S_4 \qquad \frac{}{\texttt{A} \triangleright s \mid \rangle e \mapsto s \mid e}S_5 \qquad \frac{}{\texttt{B} \triangleright s \mid]e \mapsto s \mid e}S_6$$

i. Show the execution of the new stack machine given the start state $\circ \mid$ `[(⟨⟩)]`.

ii. Does the new machine recognise N?

α) **Difficult**: Prove or disprove that $s \; \boldsymbol{N} \implies \circ \mid s \overset{!}{\mapsto} \circ \mid \epsilon$ for all strings s. Hint: You may find it useful to prove the following lemma:

$$\frac{s_1 \overset{\star}{\mapsto} s_2 \quad s_2 \mapsto s_3}{s_1 \overset{\star}{\mapsto} s_3}\textsc{Lemma}$$

Also, you may need to generalise your proof goal to a broader claim.

β) Prove or disprove that $\circ \mid s \overset{!}{\mapsto} \circ \mid \epsilon \implies s \; \boldsymbol{N}$

iii. If your answer to the previous question was no, amend the structure of the stack machine so that it does recognise N (efficiently). Explain your answer.

2. **Computing Machines**: Abstract machines are not just used for decision problems (yes/no answers), they can also be used to compute results. Can you think of a machine to compute binary addition?

(a) Formalise such a machine.

Hint: Think about the algorithm you would use when adding up large binary numbers on paper.

(b) Compute the result `110 + 1010` with your machine. Show each execution step.

3. **Evaluation Machines**: We can also use abstract machines to express the operational semantics of the λ-calculus. As abstract higher-order syntax representation of λ-terms we chose the following:

$$\frac{x \; \textit{is a var}}{x \;\; \lambda\textit{-term}} \qquad \frac{t \;\; \lambda\textit{-term}}{(\texttt{Function}\; x.t) \;\; \lambda\textit{-term}} \qquad \frac{t_1 \;\; \lambda\textit{-term} \quad t_2 \;\; \lambda\textit{-term}}{(\texttt{Apply}\; t_1\; t_2) \;\; \lambda\textit{-term}}$$

$$\frac{}{(\texttt{Function}\; x.t) \Downarrow \langle\!\langle x.t \rangle\!\rangle}\textsc{Lambda} \qquad \frac{e_1 \Downarrow \langle\!\langle x.t \rangle\!\rangle \quad e_2 \Downarrow e_2' \quad t[x := e_2'] \Downarrow r}{(\texttt{Apply}\; e1\; e2) \Downarrow r}\textsc{Apply}$$

Note: You may notice that this language only has functions! Turns out, you can encode any kind of data as a function, and that this is sufficient for a turing-complete programming language. This language was invented by Alonzo Church as an alternative solution to the Entscheidungsproblem, and is called the lambda calculus.

(a) Develop a structural operational ("small step") semantics for this language.
   i. Include a rule for function literals, if necessary.
   ii. Include three rules for function application. Assume the function expression is evaluated before the argument expression. Note that this language does not include recursion.

(b) Now define an abstract machine which eliminates recursion from the meta-level of the semantics to include an explicit stack, a la the C Machine.
   i. Define a suitable stack formalism.
   ii. Define the set of states Q, the set of initial states $I \subseteq Q$, and the set of final states $F \subseteq Q$.
   iii. Define a state transition rule for function literals
   iv. Include three rules for function application, using capture-avoiding substitution as a built-in machine operation.

(c) Now suppose that we want to eliminate substitution from our machine. Extend the semantics to include environments, a la the E Machine. Recall than an environment is commonly defined as:

$$\frac{}{\bullet \ \textit{Env}} \qquad \frac{x \ \textit{Ident} \quad t \ \textit{Expr} \quad \Gamma \ \textit{Env}}{x \leftarrow y; \Gamma \ \textit{Env}}$$

   i. Revise your definition of the state sets Q, I and F, and of the stack.
   ii. Revise your transition rule for function literals. Note that these function literals should produce closures which capture the environment at their definition.
   iii. Revise your rules for function application
   iv. Include any additional rules necessary to complete the definition, such as variable lookup.
   v. Give an example of an expression in this language which requires closures in order to evaluate correctly. Explain your answer.

4. **Stack Machines**: In this question, we will examine a machine that is quite similar to a type of machine used in virtual machines, such as the JVM, called a stack machine. Imagine an arithmetic expression language with the following big step semantics:

$$\frac{x \in \mathbb{Z}}{\texttt{(Num } x\texttt{)} \Downarrow x}\textsc{Num} \qquad \frac{x \Downarrow x' \quad y \Downarrow y'}{\texttt{(Plus } x\ y\texttt{)} \Downarrow x' + y'}\textsc{Plus} \qquad \frac{x \Downarrow x' \quad y \Downarrow y'}{\texttt{(Times } x\ y\texttt{)} \Downarrow x' \times y'}\textsc{Times}$$

We have a machine, called the J Machine, that's capable of performing these operations, however it works by using a stack to store operands and accumulate results. For example, 4 * (2 + 3) would be the following program in the J Machine's bytecode: `(Val 4); (Val 2); (Val 3); add; Times`. Each `Val` instruction pushes a value to the stack, and each operation instruction pops two values off, and pushes the result of the operation.

Formally, the J Machine is specified as follows: The machine consists of three instructions:

$$\frac{x \in \mathbb{Z}}{\texttt{(Val } x\texttt{)} \ \textit{Inst}} \qquad \frac{}{\texttt{Plus} \ \textit{Inst}} \qquad \frac{}{\texttt{Times} \ \textit{Inst}}$$

The state of the machine consists of a list of instructions, called a Program, and a stack of integers:

$$\frac{}{\texttt{Halt} \ \textit{Program}} \qquad \frac{i \ \textit{Inst} \quad p \ \textit{Program}}{i; p \ \textit{Program}}$$

$$\frac{}{\circ \ \textit{Stack}} \qquad \frac{x \in \mathbb{Z} \quad s \ \textit{Stack}}{x \triangleright s \ \textit{Stack}}$$

They are presented in the form $s \mid p$ where s is a stack and p is program. The initial state consists of the empty stack and any nonempty program p i.e, $\circ \mid p$. The final state consists of a stack with merely one element r (the result of the computation), and the empty program, i.e, $r \triangleright \circ \mid \texttt{Halt}$.

The state transition rules are as follows:

$$\frac{}{s \mid \texttt{(Val } x\texttt{)}; p \mapsto x \triangleright s \mid p} J_1 \qquad \frac{}{y \triangleright x \triangleright s \mid \texttt{Plus}; p \mapsto x + y \triangleright s \mid p} J_2 \qquad \frac{}{y \triangleright x \triangleright s \mid \texttt{Times}; p \mapsto x \times y \triangleright s \mid p} J_3$$

(a) Translate the expression `(Plus (Times (Num -1) (Num 7)) (Num 7))` into a J Machine program, and write down each step the J Machine would take to execute this program.

(b) **Possibly difficult** Formalise (using inference rules) a "compilation" relation $\curlyvee : \mathit{Expr} \times \mathit{Program}$ which translates expressions in the arithmetic language to the semantically equivalent J Machine bytecode.

(c) **Definitely difficult**. Suppose we wanted to add a `Let` construct to add variables to our arithmetic language, using environments as shown:

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash \texttt{(Num } i\texttt{)} \Downarrow i} \textsc{Num} \qquad \frac{\Gamma \vdash x \Downarrow x' \quad \Gamma \vdash y \Downarrow y'}{\Gamma \vdash \texttt{(Plus } x\ y\texttt{)} \Downarrow x' + y'} \textsc{Plus} \qquad \frac{\Gamma \vdash x \Downarrow x' \quad \Gamma \vdash y \Downarrow y'}{\Gamma \vdash \texttt{(Times } x\ y\texttt{)} \Downarrow x' \times y'} \textsc{Times}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \cup \{x \leftarrow v_1\} \vdash e_2 \Downarrow v_2}{\Gamma \vdash \texttt{(Let } x\ e_1\ e_2\texttt{)} \Downarrow v_2} \textsc{Let} \qquad \frac{x \leftarrow v \in \Gamma}{\Gamma \vdash \texttt{(Var } x\texttt{)} \Downarrow v} \textsc{Var}$$

Extend the J Machine to support this construct, and expand your $\curlyvee$ relation to include the correct translation. Don't forget to deal with name shadowing by exploiting stacks.