Programming Languages and Computation

Problem Sheet 7: Semantics of the While Language

We consider the semantics of the While language and reason about it.

We don't reason about programs written in it in this sheet because the last two videos for the week have broken sound, and I'll be shifting content to next week instead. This might turn out to be much more exciting.

Unrolling Execution Traces

In this section, justify each of your answers with a derivation based on the formal definitions of semantics given in the reference material.

Evaluating Arithmetic Expressions

* 1. Evaluate the following arithmetic expressions in state $\sigma = \begin{bmatrix} \texttt{a} \mapsto 42 \\ \texttt{b} \mapsto 154 \\ \texttt{x} \mapsto 11 \end{bmatrix}$

(a) `a + x * 14 - b`

(b) `x * (a - x) * b`

* 2. Evaluate the following arithmetic expressions in state $\sigma = \begin{bmatrix} \texttt{x} \mapsto 11 \\ \texttt{b} \mapsto 12 \\ \texttt{h} \mapsto 84 \end{bmatrix}$

(a) `a + x * 14 - b`

(b) `b - g * x + h`

* 3. Evaluate the following boolean expressions in state $\sigma = \begin{bmatrix} \texttt{a} \mapsto 42 \\ \texttt{b} \mapsto 154 \\ \texttt{x} \mapsto 11 \end{bmatrix}$

(a) `a + x * 14 - b` $\leq$ `b + x` $\wedge$ `true`

(b) `!x * (a - x) * b` $=$ `b + x` $\wedge$ `x` $=$ `11`

(c) `b - g * x + h` $\leq$ `0`

Execution Traces

** 4. Consider the While programs you wrote in week 5. (You may use those from the solution.) Give their execution trace in the following state σ when the trace is finite. (Show enough of the execution trace to find a cycle if the trace is infinite.)

$$\sigma = \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 5 \end{bmatrix}$$

*** 5. Find an initial configuration (program and state) that do not give rise to any finite complete traces, but such that all infinite traces have no repeating configurations.
(This shows that deciding termination is more complex than simply detecting cycles.)

Structural Induction

Pretty much everything we've defined since we started talking about the While language (and some of the things we defined before) were defined inductively: they are defined as the smallest set that contains some base elements (the base cases), and is closed under certain operations (the inductive cases).

A very simple and familiar example of such an inductively defined object is the set $\mathbb{N}$ of natural integers. $\mathbb{N}$ is indeed defined as the smallest set such that:

- $0 \in \mathbb{N}$, and
- if $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

* 6. Let's see why it's important to say "the smallest set such that" in inductive definitions, and try to give a definition for *even* integers.
Let $\mathbb{E} \subseteq \mathbb{N}$ be a set such that

- $0 \in \mathbb{E}$, and
- if $n \in \mathbb{E}$, then $n + 2 \in \mathbb{E}$.

(a) Find a set that meets the two conditions we placed on $\mathbb{E}$ but contains odd integers.

By stating that a set is "the smallest such that", we say that all its elements *must* be constructed using the closure operations applied to the base elements, and that no other elements belong in that set![1]

[1] We kind of gloss over the fact that it is not obvious there exists a smallest set such that the conditions hold, or that it is unique if it exists; that's a topic for another unit. In this unit, we only ever give you inductive definitions for sets that actually exist.

Mathematical Induction. Inductively-defined sets are very nicely behaved. Because we know that all the objects they contain are constructed through the given operations from the given base cases, we can prove properties for all objects by *structural induction*.

In the case of $\mathbb{N}$, you already know this as *mathematical induction*. To prove that some property $P : \mathbb{N} \to \{\top, \bot\}$ holds for all $n \in \mathbb{N}$ (we'll write $P(n)$ for "P holds on n"), we prove, that:

- $P(0)$; and
- if $P(n)$, then $P(n+1)$.

These two combined give us a proof for any $n \in \mathbb{N}$: because all elements in $\mathbb{N}$ are either 0 or the result of applying the successor operation to 0 a finite number of times n, we can obtain a proof that P holds by applying the second lemma n times until we have to prove $P(0)$.

Structural Induction. Structural induction does the same for any inductively-defined set (and also functions and relations, since they are sets, too). For example, let's consider a simple set $\mathbb{T}$ of trees that are only a structure—they won't contain any data. The set $\mathbb{T}$ is the smallest set such that:

- $\bullet \in \mathbb{T}$, and
- if $\ell \in \mathbb{T}$ and $r \in \mathbb{T}$, then $N(\ell, r) \in \mathbb{T}$.

As examples, $\bullet$ is a tree in $\mathbb{T}$, but also $N(\bullet, \bullet)$, and $N(N(\bullet, \bullet), \bullet)$, ...

On this set, we can define some neat functions inductively. For example, the $\mathsf{size} : \mathbb{T} \to \mathbb{N}$ function counts the number of $\bullet$ in the tree, and the $\mathsf{depth} : \mathbb{T} \to \mathbb{N}$ function counts the number of "levels" in the tree.

$$\mathsf{size}(\bullet) = 1 \qquad \mathsf{depth}(\bullet) = 1$$
$$\mathsf{size}(N(\ell, r)) = \mathsf{size}(\ell) + \mathsf{size}(r) \qquad \mathsf{depth}(N(\ell, r)) = \max(\mathsf{depth}(\ell), \mathsf{depth}(r))$$

** 7. Adapting the reasoning principles of mathematical induction to the inductive structure of $\mathbb{T}$, prove the following two facts:

(a) Given any $t \in \mathbb{T}$, we have $\mathsf{depth}(t) \leq \mathsf{size}(t)$.

(b) Given any $t \in \mathbb{T}$, if $\mathsf{depth}(t) = \mathsf{size}(t)$, then $t = \bullet$.

Properties of While

The While language (and its expressions and boolean expressions) are also defined inductively. We formally defined them only as Haskell datatypes, but the reference material also includes a mathematical definition (with unfortunate Haskell notations). This means we can reason about all those objects inductively. Lecture 4 (even without the audio explanations) sketches a proof of the fact that the semantics we gave to the While language is deterministic: each one of its configurations has at most one successor.

We're now going to prove some more properties of the language, starting with some we kind of glossed over so far.

**** 8. In the lectures on syntax, I mentioned that the associativity of sequential composition did not in fact matter, and chose to make it right associative. We'll first explore the choice, then the claim.

This only leverages mathematical induction.
In the following, you can assume an additional rule $\langle \texttt{skip}; S, \sigma \rangle \Rightarrow \langle S, \sigma \rangle$.

(a) Identify the next configuration for the following two programs, in some abstract state σ, justifying it fully using the rules of the semantics for While:

1. `x := y; (z := x; y := z)`, and
2. `(x := y; z := x); y := z`.

(b) Show that, if $\langle S_1; S_2, \sigma \rangle \Rightarrow^k \langle \texttt{skip}, \sigma'' \rangle$, then there must exist some state σ' and some natural number $k_1 \leq k$ such that $\langle S_1, \sigma \rangle \Rightarrow^{k_1} \langle \texttt{skip}, \sigma' \rangle$ and $\langle S_2, \sigma' \rangle \Rightarrow^{k-k_1} \langle \texttt{skip}, \sigma'' \rangle$.
(In other words, there is a suffix of the complete execution trace for $S_1; S_2$ in state σ that is an execution trace for S_2 in some state σ' that happens to be the terminal state when executing S_1 in $sigma$.)
This proof is by mathematical induction on k. You'll want to isolate the *first* transition in the inductive step of the proof, and do the appropriate case analysis.

(c) Show that, if $\langle S_1, \sigma \rangle \Rightarrow^k \langle \texttt{skip}, \sigma' \rangle$, then $\langle S_1; S_2, \sigma \rangle \Rightarrow^k \langle S_2, \sigma' \rangle$.
As before, this proof is by mathematical induction on k, isolating the first transition.

(d) Show that, if $\langle S_1; (S_2; S_3), \sigma \rangle \Rightarrow^* \langle \texttt{skip}, \sigma_3 \rangle$, then $\langle (S_1; S_2); S_3, \sigma \rangle \Rightarrow^* \langle \texttt{skip}, \sigma_3 \rangle$.
(The converse also holds, but the proof is roughly the same, and quite tedious.)

*** 9. We now show that the semantics of arithmetic expressions depends only on the value of variables that actually appear in the expression. We call these the *free variables* of an expression, and first define that set. As always, we do so inductively on the structure of expressions—and avoid some verbosity by combining all three "binary operations" cases into a single case where $\diamond \in \{+, -, *\}$.

$$FV(i) = \emptyset$$
$$FV(\mathtt{x}) = \{\mathtt{x}\}$$
$$FV(e_1 \diamond e_2) = FV(e_1) \cup FV(e_2)$$

(a) Show that, if for any $\mathtt{x} \in FV(e)$, $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$, then $[\![e]\!]_{\mathscr{A}}(\sigma_1) = [\![e]\!]_{\mathscr{A}}(\sigma_2)$.