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`

Solution

The results can be obtained easily by doing a simple substitution and evaluating. The interest of asking for a derivation based on the formal definitions is to force thinking about associativity and the correspondence between the abstract syntax and the ASTs.

I only expand the first. The second has similar associativity.

(a)

$$\begin{aligned}
[\![\texttt{a + x * 14 - b}]\!]_{\mathscr{A}}(\sigma) &= [\![\texttt{a + x * 14}]\!]_{\mathscr{A}}(\sigma) - [\![\texttt{b}]\!]_{\mathscr{A}}(\sigma) \\
&= [\![\texttt{a}]\!]_{\mathscr{A}}(\sigma) + [\![\texttt{x * 14}]\!]_{\mathscr{A}}(\sigma) - [\![\texttt{b}]\!]_{\mathscr{A}}(\sigma) \\
&= [\![\texttt{a}]\!]_{\mathscr{A}}(\sigma) + [\![\texttt{x}]\!]_{\mathscr{A}}(\sigma) * 14 - [\![\texttt{b}]\!]_{\mathscr{A}}(\sigma) \\
&= \sigma(a) + \sigma(x) * 14 - \sigma(b) \\
&= 42 + 11 * 14 - 154 \\
&= 42
\end{aligned}$$

(b)

$$[\![\texttt{x * (a - x) * b}]\!]_{\mathscr{A}}(\sigma) = 52514$$

* 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`

Solution

The only additional difficulty is that you need to remember that looking up a variable we have not explicitly defined yields value 0.

(a)

$$[\![\texttt{a + x * 14 - b}]\!]_{\mathscr{A}}(\sigma) = 142$$

(b)

$$[\![\texttt{b - g * x + h}]\!]_{\mathscr{A}}(\sigma) = 86$$

* 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`

Solution

No additional difficulty here again. Recall that `!` binds stronger than $\wedge$.

(a)

$$[\![\texttt{a + x * 14 - b} \leq \texttt{b + x} \wedge \texttt{true}]\!]_{\mathscr{B}}(\sigma) = \top$$

(b)

$$[\![\texttt{!x * (a - x) * b = b + x} \wedge \texttt{x = 11}]\!]_{\mathscr{B}}(\sigma) = \top$$

(c)

$$[\![\texttt{b - g * x + h} \leq \texttt{0}]\!]_{\mathscr{B}}(\sigma) = \bot$$

Execution Traces

** 4. Consider the While programs you wrote in Question 1 in Week 5. (You may use those from the solution.) Give their execution trace in the following state σ when the trace is finite.

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

Solution

(a) `n := n + 2`

$$\langle \texttt{n := n + 2}, \sigma \rangle \Rightarrow \left\langle \texttt{skip}, \begin{bmatrix} \texttt{n} \mapsto 5 \\ \texttt{a} \mapsto 5 \end{bmatrix} \right\rangle$$

(b) $S =$ `if (0 <= n) then r := n else r := 0 - n`

$$\begin{aligned} \langle S, \sigma \rangle &\Rightarrow \langle \texttt{r := n}, \sigma \rangle \\ &\Rightarrow \left\langle \texttt{skip}, \begin{bmatrix} \texttt{n} \mapsto 3 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 3 \end{bmatrix} \right\rangle \end{aligned}$$

(c) $S =$ `b := a + b; a := b - a; b := b - a`

$$\begin{aligned} \langle S, \sigma \rangle &\Rightarrow \left\langle \texttt{a := b - a; b := b - a}, \begin{bmatrix} \texttt{n} \mapsto 3 \\ \texttt{a} \mapsto 5 \\ \texttt{b} \mapsto 5 \end{bmatrix} \right\rangle \\ &\Rightarrow \left\langle \texttt{b := b - a}, \begin{bmatrix} \texttt{n} \mapsto 3 \\ \texttt{a} \mapsto 0 \\ \texttt{b} \mapsto 5 \end{bmatrix} \right\rangle \\ &\Rightarrow \left\langle \texttt{skip}, \begin{bmatrix} \texttt{n} \mapsto 3 \\ \texttt{a} \mapsto 0 \\ \texttt{b} \mapsto 5 \end{bmatrix} \right\rangle \end{aligned}$$

(d) $S =$ `while ! n <= 0 r := n; n := n - 2`

$$\langle S, \sigma \rangle \Rightarrow \langle \texttt{r := n; n := n - 2;}\ S, \sigma \rangle$$
$$\Rightarrow^2 \left\langle S, \begin{bmatrix} \texttt{n} \mapsto 1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 3 \end{bmatrix} \right\rangle$$
$$\Rightarrow \left\langle \texttt{r := n; n := n - 2;}\ S, \begin{bmatrix} \texttt{n} \mapsto 1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 3 \end{bmatrix} \right\rangle$$
$$\Rightarrow^2 \left\langle S, \begin{bmatrix} \texttt{n} \mapsto \text{-}1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 1 \end{bmatrix} \right\rangle$$
$$\Rightarrow \left\langle \texttt{skip}, \begin{bmatrix} \texttt{n} \mapsto \text{-}1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 1 \end{bmatrix} \right\rangle$$

(e) The last program does not terminate.

*** 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.)

Solution

The following program suffices: the statement component of the configuration only takes two different values, but the state component changes without repeating. This is true for any initial state.

```
while (true) x := x + 1
```

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.

Solution

(a) The set $\mathbb{N}$ contains 0, and for any integer $n \in \mathbb{N}$, we also have $n+2 \in \mathbb{N}$. (This is because $n+1 \in \mathbb{N}$ by definition of $\mathbb{N}$, and therefore also $n+2 = n+1+1 \in \mathbb{N}$.)
The key here is that $\mathbb{N}$ contains 1, which cannot be constructed by adding a multiple of 2 to 0.

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]

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.

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

[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.

** 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 $1 \leq \mathsf{depth}(t)$ and $1 \leq \mathsf{size}(t)$.

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

Solution

The structural induction principle for $\mathbb{T}$ says we have to prove the property for $\bullet$, and then, assuming it holds for two trees ℓ and r also prove that it must hold for $N(\ell, r)$.

We write these proofs in slightly more detail than we'd expect in an exam setting.

(a) By structural induction on t, we prove that for any $t \in \approx$, we have $1 \leq \mathsf{depth}(t)$ and $1 \leq \mathsf{size}(t)$.

Case $t = \bullet$**:** In this case, we have $\mathsf{size}(\bullet) = \mathsf{depth}(\bullet) = 1$.

Case $t = N(\ell, r)$**:** We have two induction hypotheses (one for each of the subtrees):

1. $1 \leq \mathsf{depth}(\ell)$ and $1 \leq \mathsf{size}(\ell)$, and
2. $1 \leq \mathsf{depth}(r)$ and $1 \leq \mathsf{size}(r)$.

Therefore, we have $1 \leq 2 \leq 1 + \max(\mathsf{depth}(\ell), \mathsf{depth}(r)) = \mathsf{depth}(N(\ell, r))$, and $1 \leq 2 \leq \mathsf{size}(\ell) + \mathsf{size}(r) = \mathsf{size}(N(\ell, r))$.

(b) By structural induction on t, we prove that for any $t \in \mathbb{T}$ we have $\mathsf{depth}(t) \leq \mathsf{size}(t)$.

Case $t = \bullet$**:** In this case, we have $\mathsf{size}(\bullet) = 1 = \mathsf{depth}(\bullet)$, and the property holds.

Case $t = N(\ell, r)$**:** We have two induction hypotheses (one for each of the subtree):

1. $\mathsf{depth}(\ell) \leq \mathsf{size}(\ell)$, and
2. $\mathsf{depth}(r) \leq \mathsf{size}(r)$.

By definition of size and depth, we have $\mathsf{depth}(N(\ell, r)) = 1 + \max(\mathsf{depth}(\ell), \mathsf{depth}(r))$ and $\mathsf{size}(N(\ell, r)) = \mathsf{size}(\ell) + \mathsf{depth}(r)$.

By the induction hypotheses, we know that $1 + \max(\mathsf{depth}(\ell), \mathsf{depth}(r)) \leq 1 + \max(\mathsf{size}(\ell), \mathsf{size}(r))$.

Further, since $1 \leq \mathsf{size}(\ell)$ and $1 \leq \mathsf{size}(r)$, we have $1 + \max(\mathsf{size}(\ell), \mathsf{size}(r)) \leq \mathsf{size}(\ell) + \mathsf{size}(r)$, and the property holds.

It's worth noting that it's in fact easy to prove both properties in a single go, using $1 \leq \mathsf{depth}(t) \leq \mathsf{size}(t)$ as the inductive property.

Let us define the set $\mathbb{Z} \subseteq \mathbb{T}$ of *zig-zags* as the smallest set such that:

- $\bullet \in \mathbb{Z}$;
- if $t \in \mathbb{Z}$, then $N(t, \bullet) \in \mathbb{Z}$; and
- if $t \in \mathbb{Z}$, then $N(\bullet, t) \in \mathbb{Z}$.

*** 8.

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

Solution

(a) By structural induction on t, we show that any t such that $\mathsf{depth}(t) = \mathsf{size}(t)$ is a zig-zag.

Case $t = \bullet$**:** t is a zig-zag by definition.

Case $t = N(\ell, r)$**:** By induction hypothesis, we have:

1. if $\mathsf{depth}(\ell) = \mathsf{size}(\ell)$, then $\ell \in \mathbb{Z}$;
2. if $\mathsf{depth}(r) = \mathsf{size}(r)$, then $r \in \mathbb{Z}$.

We have $\mathsf{size}(N(\ell, r)) = \mathsf{size}(\ell) + \mathsf{size}(r) = 1 + \max(\mathsf{depth}(\ell), \mathsf{depth}(r)) = \mathsf{depth}(N(\ell, r))$.
Let us consider the case where $\max(\mathsf{depth}(\ell), \mathsf{depth}(r)) = \mathsf{depth}(\ell)$.
In this case, we have $\mathsf{size}(\ell) + \mathsf{size}(r) = \mathsf{depth}(\ell) + 1$. Since $1 \leq \mathsf{depth}(\ell) \leq \mathsf{size}(\ell)$ and $1 \leq \mathsf{size}(r)$, it must be that $1 = \mathsf{size}(r)$ and $\mathsf{size}(\ell) = \mathsf{depth}(\ell)$.
We therefore have $r = \bullet$ and $\ell \in \mathbb{Z}$.[2] and t is therefore a zig-zag.
The case $\mathsf{depth}(N(\ell, r)) = \mathsf{depth}(r)$ gives rise to a symmetrical argument which constructs a zig-zag that starts on the right.

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.

**** 9. 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.

(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 $\langle S_1; S_2, \sigma\rangle \Rightarrow^k \langle \mathtt{skip}, \sigma''\rangle$, then there must exist some state σ' and some natural number $k_1 \leq k$ such that $\langle S_1, \sigma\rangle \Rightarrow^{k_1} \langle \mathtt{skip}, \sigma'\rangle$ and $\langle S_2, \sigma'\rangle \Rightarrow^{k-k_1} \langle \mathtt{skip}, \sigma''\rangle$.
(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 $\langle S_1, \sigma\rangle \Rightarrow^k \langle \mathtt{skip}, \sigma'\rangle$, then $\langle S_1; S_2, \sigma\rangle \Rightarrow^k \langle S_2, \sigma'\rangle$.

[2] You may want to prove $\mathsf{size}(t) = 1 \Rightarrow t = \bullet$ also, by induction. But it's obvious enough I wouldn't hold it against you if you didn't.

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.)

Solution

(a) The point here is to have you reflect on what it takes to identify the next configuration (which will look very similar in linear notation but would be different if we drew out the AST).

1. In the first case, the sequence rule where the first statement executes in one step gives us the next configuration right away.
   Since we have $\langle \texttt{x := y},\sigma\rangle \Rightarrow \langle \texttt{skip},\sigma'\rangle$, we also have $\langle \texttt{x := y; (z := x; y := z)},\sigma\rangle \Rightarrow \langle \texttt{z := x; y := x},\sigma'\rangle$.
2. In the second case, the reason why this (same) transition is possible requires two levels of justification.
   Since we have $\langle \texttt{x := y},\sigma\rangle \Rightarrow \langle \texttt{skip},\sigma'\rangle$, we also have $\langle \texttt{x := y; z := x},\sigma\rangle \Rightarrow \langle \texttt{z := x},\sigma'\rangle$. and therefore we have $\langle \texttt{(x := y; z := x); y := z},\sigma\rangle \Rightarrow \langle \texttt{z := x; y := z},\sigma'\rangle$.

Now imagine if an entire n-line program is parsed into a left-associative way: justifying its first transition take $n-1$ reasoning steps.

(b) By mathematical induction over the length k of the trace.

Case $k=0$**:** $\langle S_1,\sigma\rangle = \langle \texttt{skip},\sigma''\rangle$, and we therefore have $\langle S_1;S_2,\sigma\rangle = \langle \texttt{skip};S_2,\sigma\rangle \Rightarrow \langle S_2,\sigma\rangle$ and we conclude with $k_1=0$ and $\sigma'=\sigma$.

Case $k=n+1$**:** $\langle S_1,\sigma\rangle \Rightarrow \langle S_1',\sigma'\rangle \Rightarrow^n \langle \texttt{skip},\sigma''\rangle$

Case $S_1' = \texttt{skip}$**:** Then we have $\langle S_1,\sigma\rangle \Rightarrow^1 \langle \texttt{skip},\sigma'\rangle$, and therefore also $\langle S_1;S_2,\sigma\rangle \Rightarrow^1 \langle S_2,\sigma'\rangle \Rightarrow^n \langle \texttt{skip},\sigma''\rangle$, and we can conclude with $k_1=1$ and σ' as defined.

Case $S_1' \neq \texttt{skip}$**:** Then we have $\langle S_1,\sigma\rangle \Rightarrow^1 \langle S_1',\sigma'\rangle$, and therefore also $\langle S_1;S_2,\sigma\rangle \Rightarrow^1 \langle S_1';S_2,\sigma'\rangle \Rightarrow^n \langle \texttt{ski[},\sigma''\rangle$. By induction hypothesis, there exists a state σ_1 and some integer $k_1' \leq n$ such that $\langle S_1',\sigma'\rangle \Rightarrow^{k_1'} \langle \texttt{skip},\sigma_1\rangle$ and $\langle S_2,\sigma_1\rangle \Rightarrow^{n-k_1'} \langle \texttt{skip},\sigma''\rangle$. Therefore, we have $\langle S_1,\sigma\rangle \Rightarrow^{k_1'+1} \langle \texttt{skip},\sigma_1\rangle$, $\langle S_2,\sigma_1\rangle \Rightarrow^{n-k_1'-1} \langle \texttt{skip},\sigma''\rangle$ and $k_1'+1 \leq n+1$, and we conclude.

(c) By mathematical induction on k.

Case $k=0$**:** By definition of $\Rightarrow$.

Case $k=n+1$**:** $\langle S_1,\sigma\rangle \Rightarrow^1 \langle S_1',\sigma''\rangle \Rightarrow^n \langle \texttt{skip},\sigma'\rangle$

Case $S_1' = \texttt{skip}$**:** Then $n=0$ and we conclude by definition of $\Rightarrow$.

Case $S_1' \neq \texttt{skip}$**:** Then we have $\langle S_1;S_2,\sigma\rangle \Rightarrow \langle S_1';S_2,\sigma''\rangle$ (by definition of $\Rightarrow$) and $\langle S_1';S_2,\sigma''\rangle \Rightarrow^n \langle S_2,\sigma'\rangle$ (by induction hypothesis), and we conclude.

(d) Use (b) to deconstruct the trace into its three discrete components. THen use (c) to reconstruct the trace with the other associativity. It obviously also works the other way.

*** 10. 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 $[\![e]\!]_{\mathscr{A}}(\sigma_1) = [\![e]\!]_{\mathscr{A}}(\sigma_2)$.

Solution

(a) By structural induction on e.

Case $e = i$**:** $[\![\mathtt{i}]\!]_{\mathscr{A}}(\sigma_1) = i = [\![\mathtt{i}]\!]_{\mathscr{A}}(\sigma_2)$, which closes this case.

Case $e = \mathtt{x}$**:** Then we have $\mathtt{x} \in FV(e)$, and therefore $[\![\mathtt{x}]\!]_{\mathscr{A}}(\sigma_1) = \sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x}) = [\![\mathtt{x}]\!]_{\mathscr{A}}(\sigma_2)$.

Case $e = e_1 \diamond e_2$**:** Then we have $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$ for all $\mathtt{x} \in FV(e_1 \diamond e_2) = FV(e_1) \cup FV(e_2)$ and we get, by induction hypothesis, that $[\![e_1]\!]_{\mathscr{A}}(\sigma_1) = [\![e_1]\!]_{\mathscr{A}}(\sigma_2)$ and that $[\![e_2]\!]_{\mathscr{A}}(\sigma_1) = [\![e_2]\!]_{\mathscr{A}}(\sigma_2)$. Therefore, $[\![e_1 \diamond e_2]\!]_{\mathscr{A}}(\sigma_1) = [\![e_1]\!]_{\mathscr{A}}(\sigma_1) \diamond [\![e_2]\!]_{\mathscr{A}}(\sigma_1) = [\![e_1]\!]_{\mathscr{A}}(\sigma_2) \diamond [\![e_2]\!]_{\mathscr{A}}(\sigma_2) = [\![e_1 \diamond e_2]\!]_{\mathscr{A}}(\sigma_2)$.