Programming Languages and Computation

Week 8: Big Steps, Liveness and Correctness

We first spend some more time practicing with the While language and its big step semantics with some exercises on programming, on derivations and with some proofs on the language as a whole. We then practice safety and correctness reasoning looking at the properties of some interesting While programmes.

Finally, we consider an extension of the language which "goes wrong" in different ways, and consider what that means for the While language semantics.

Familiarity with the mechanics of While derivations is a must, so keep practicing those until you confidently get them. Proofs on the language, and proofs of safety and correctness, are useful to master to reach for first class marks. Understanding how programmes written in the WhileDiv language go wrong will be useful only for very top achievers.

While practice

In the following, feel free to make use of past results—especially on associativity of sequential composition—when they make derivations less unwieldy. When doing so, be explicit about the fact that you are applying a known result.

* 1. Write a programme exp in the While language such that, for any state σ with $0 \leq \sigma(\mathtt{n})$ we have

$$\left\langle \mathtt{exp}, \sigma \right\rangle \Downarrow \sigma'$$

for some state σ' such that $\sigma'(\mathtt{x}) = \sigma(\mathtt{x})$, $\sigma'(\mathtt{n}) = \sigma(\mathtt{n})$ and $\sigma'(\mathtt{r}) = \sigma(\mathtt{x})^{\sigma(\mathtt{n})}$.
You do not (yet) need to prove termination or correctness. Choose one initial state σ_{in} that meets the condition above and one initial state σ_{out} that doesn't, and construct derivations for $\left\langle \mathtt{exp}, \sigma_{in} \right\rangle \Downarrow \sigma'$ and $\left\langle \mathtt{exp}, \sigma_{out} \right\rangle \Downarrow \sigma'$ if they exist. (You may wish to carefully select the initial values of n.)

Solution

The following programme works.

```
r ← 1;
m ← n;
while (1 <= m)
  r ← r * x;
  m ← m - 1
```

Consider $\sigma_{in} = [\mathtt{n} \mapsto 0]$, with the following derivation.

$$\dfrac{\dfrac{}{\langle \texttt{r} \leftarrow \texttt{1}, \sigma_{in}\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]} \quad \dfrac{\dfrac{}{\langle \texttt{m} \leftarrow \texttt{n}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]} \quad \dfrac{[\![\texttt{1 <= m}]\!]^{\mathcal{B}}([\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]) = \bot}{\left\langle \begin{array}{l}\texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}}{\left\langle \begin{array}{l}\texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}}{\left\langle \begin{array}{l}\texttt{r} \leftarrow \texttt{1;}\\ \texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, \sigma_{in}\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}$$

Consider $\sigma_{out} = [\texttt{n} \mapsto -10]$, with the following derivation.

$$\dfrac{\dfrac{}{\langle \texttt{r} \leftarrow \texttt{1}, \sigma_{in}\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]} \quad \dfrac{\dfrac{}{\langle \texttt{m} \leftarrow \texttt{n}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]} \quad \dfrac{[\![\texttt{1 <= m}]\!]^{\mathcal{B}}([\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]) = \bot}{\left\langle \begin{array}{l}\texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]}}{\left\langle \begin{array}{l}\texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]}}{\left\langle \begin{array}{l}\texttt{r} \leftarrow \texttt{1;}\\ \texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (1 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * x;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, \sigma_{in}\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]}$$

Note the amount of laziness involved in selecting the initial values of n considered, and feel free to emulate it in time-limited settings.

** 2. Write a programme `factorial` in the While language such that, for any state σ with $0 \leq \sigma(\texttt{n})$ we have

$$\langle \texttt{factorial}, \sigma\rangle \Downarrow \sigma'$$

for some state σ' such that $\sigma'(\texttt{n}) = \sigma(\texttt{n})$ and $\sigma'(\texttt{r}) = \sigma(\texttt{n})!$.
You do not (yet) need to prove termination or correctness. Choose one initial state σ_{in} that meets the condition above and one initial state σ_{out} that doesn't, and construct derivations for $\langle \texttt{factorial}, \sigma_{in}\rangle \Downarrow \sigma'$ and $\langle \texttt{factorial}, \sigma_{out}\rangle \Downarrow \sigma'$ if they exist. (You may wish to carefully select the initial values of n.)

Solution

The following programme works.

```
r ← 1;
m ← n;
while (2 <= m)
│ r ← r * m;
└ m ← m - 1
```

Consider $\sigma_{in} = [\texttt{n} \mapsto 0]$, with the following derivation.

$$\dfrac{\dfrac{}{\langle \texttt{r} \leftarrow \texttt{1}, \sigma_{in}\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]} \quad \dfrac{\dfrac{}{\langle \texttt{m} \leftarrow \texttt{n}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]} \quad \dfrac{[\![\texttt{1 <= m}]\!]^{\mathcal{B}}([\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]) = \bot}{\left\langle \begin{array}{l}\texttt{while (2 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * m;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}}{\left\langle \begin{array}{l}\texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (2 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * m;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto 0]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}}{\left\langle \begin{array}{l}\texttt{r} \leftarrow \texttt{1;}\\ \texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (2 <= m)}\\ \begin{array}{|l}\texttt{r} \leftarrow \texttt{r * m;}\\ \texttt{m} \leftarrow \texttt{m - 1}\end{array}\end{array}, \sigma_{in}\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto 0;\texttt{n}\mapsto 0]}$$

Consider $\sigma_{out} = [\texttt{n} \mapsto -10]$, with the following derivation.

$$
\dfrac{
\dfrac{}{\langle \texttt{r} \leftarrow \texttt{1}, \sigma_{in}\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]}
\qquad
\dfrac{
\dfrac{}{\langle \texttt{m} \leftarrow \texttt{n}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]}
\quad
\dfrac{[\![\texttt{1 <= m}]\!]^{\mathscr{B}}([\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]) = \bot}{\left\langle \begin{array}{l}\texttt{while (2 <= m)}\\ \quad \texttt{r} \leftarrow \texttt{r * m;}\\ \quad \texttt{m} \leftarrow \texttt{m - 1}\end{array}, [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]}
}{
\left\langle \begin{array}{l}\texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (2 <= m)}\\ \quad \texttt{r} \leftarrow \texttt{r * m;}\\ \quad \texttt{m} \leftarrow \texttt{m - 1}\end{array}, [\texttt{r}\mapsto 1;\texttt{n}\mapsto -10]\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]
}
}{
\left\langle \begin{array}{l}\texttt{r} \leftarrow \texttt{1;}\\ \texttt{m} \leftarrow \texttt{n;}\\ \texttt{while (2 <= m)}\\ \quad \texttt{r} \leftarrow \texttt{r * m;}\\ \quad \texttt{m} \leftarrow \texttt{m - 1}\end{array}, \sigma_{in}\right\rangle \Downarrow [\texttt{r}\mapsto 1;\texttt{m}\mapsto -10;\texttt{n}\mapsto -10]
}
$$

Note the *extreme* amount of laziness involved in selecting both the programme and the initial values of `n` considered, and feel free to emulate it in time-limited settings.

** 3. Consider the following derivation "shapes". For each, find two distinct programmes `s` and `s'` and states σ and σ' such that the derivations for $\langle \texttt{s}, \sigma\rangle \Downarrow \sigma'$ and $\langle \texttt{s'}, \sigma\rangle \Downarrow \sigma'$ match the given shape. In each case, give the relation or fact that appears in the spot marked with "?" in the derivation shape.

1. $$\text{(BSeq)}\ \dfrac{\text{(BAss)}\ \dfrac{}{\cdots} \qquad \text{(BWhile}_\bot)\ \dfrac{?}{\cdots}}{\cdots}$$

2. $$\text{(BSeq)}\ \dfrac{\text{(BSkip)}\ \dfrac{}{?} \qquad \text{(BAss)}\ \dfrac{}{\cdots}}{\cdots}$$

** 4. Prove that, for any statement s and any states σ and σ', we have

$$\langle s, \sigma\rangle \Downarrow \sigma' \Leftrightarrow \langle s\,\texttt{;}\,, \sigma\rangle \Downarrow \sigma'$$

(Recall that s`;` is linear notation for the tree `;`$(s, \texttt{skip})$.)

Solution

We prove the following:

For every statement s and any states σ and σ', if $\langle s, \sigma\rangle \Downarrow \sigma'$ then $\langle s\,\texttt{;}\,, \sigma\rangle \Downarrow \sigma'$.

Let s be a statement, and σ and σ' be states such that $\langle s, \sigma\rangle \Downarrow \sigma'$. There must therefore exist a derivation D such that

$$\dfrac{D}{\langle s, \sigma\rangle \Downarrow \sigma'}$$

Then the following is a valid derivation.

$$\dfrac{\dfrac{D}{\langle s, \sigma\rangle \Downarrow \sigma'} \qquad \dfrac{}{\langle \texttt{skip}, \sigma'\rangle \Downarrow \sigma'}}{\langle s\,\texttt{;}\,, \sigma\rangle \Downarrow \sigma'}$$

The converse implication holds by a similar argument.

*** 5. Write a programme `pingala` in the While programming language, which computes the nth Pingala number $p(n)$—with p as defined below, where n is the value of variable `n` in the initial state.[a] The value of $p(n)$ should be, in the final state, stored in variable `r`. You may, if needed, assume that the initial value of n is non-negative. You may also, if needed, use variables other than n and r without worry.

$$p(n) = \begin{cases} 0 & \text{if } n \leq 0 \\ 1 & \text{if } n = 1 \\ p(n-2) + p(n-1) & \text{otherwise} \end{cases}$$

You do not (yet) need to prove termination or correctness. Choose one initial state σ_{in} that meets the condition above and one initial state σ_{out} that doesn't, and construct derivations for $\langle \texttt{pingala}, \sigma_{in} \rangle \Downarrow \sigma'$ and $\langle \texttt{pingala}, \sigma_{out} \rangle \Downarrow \sigma'$ if they exist. (You may wish to carefully select the initial values of n.)
Hint: First work out the logic of turning the recursive definition into an iterative programme, then write it as a programme. Our language is restricted enough that it's a bit painful to do directly. ("A bit" is a bit of an understatement.)

[a]These are commonly known as Fibonacci number. Indian mathematician and poet Pingala first wrote about them about 2000 years before Fibonacci.

Solution

```
if (n <= 0) { r ← 0 }
else if (n = 1) { r ← 1 }
else {
 a ← 0; b ← 1; i ← n;
 while (1 <= i) {
  t ← a; a ← b; b ← t + b; i ← i - 1
 };
 r ← a + b
}
```

There are several ways of writing this programme. The one given above uses the loop to find the values of $p(n-2)$ and $p(n-1)$ (in a and b, respectively), then returns their sum as the final result. You could also shift the loop so that one of the two variables contains the final result instead.

*** 6. Prove that our $\Downarrow$ relation is a partial function.
Namely, show that, for any statement s and any states σ_0, σ and σ', if we have both $\langle s, \sigma_0 \rangle \Downarrow \sigma$ and $\langle s, \sigma_0 \rangle \Downarrow \sigma'$, then $\sigma = \sigma'$.
Hint: If induction over derivation trees is too intimidating, you may find that a mathematical induction over the depth/height of the derivation will do.

Solution

We prove the following by structural induction on the derivation of $\langle s, \sigma_0 \rangle \Downarrow \sigma$. (Note the quantifications!)

> For every statement s and every triple of states σ_0, σ and σ', if $\langle s, \sigma_0 \rangle \Downarrow \sigma$ and $\langle s, \sigma_0 \rangle \Downarrow \sigma'$, then $\sigma = \sigma'$.

Case (BSkip)**:** if

$$\text{(BSkip)}\ \frac{}{\langle s, \sigma_0 \rangle \Downarrow \sigma}$$

then it must be that $s = \texttt{skip}$ and $\sigma = \sigma_0$.

(BSkip) is the only rule that constructs a derivation for a configuration whose programme component is empty, and we must therefore have $\sigma' = \sigma_0 = \sigma$.

Case (BAss): if

$$(\text{BAss})\ \frac{}{\langle s, \sigma_0 \rangle \Downarrow \sigma}$$

then it must be that $s = \texttt{x} \leftarrow A$ and $\sigma = \sigma_0[\texttt{x} \mapsto [\![\texttt{A}]\!]^{\mathscr{A}}(\sigma_0)$.

(BAss) is the only rule that constructs a derivation for a configuration whose programme component is an assignment, and we must therefore also have $\sigma' = \sigma_0[\texttt{x} \mapsto [\![\texttt{A}]\!]^{\mathscr{A}}(\sigma_0)]$. Since the semantics of arithmetic expressions is a function, we also have $\sigma' = \sigma$.

Case (BSeq): if

$$(\text{BSeq})\ \frac{D_1 \qquad D_2}{\langle s, \sigma_0 \rangle \Downarrow \sigma}$$

then it must be that $s = s_1\,;\, s_2$ and there exists a state σ_1 such that D_1 derives $\langle s_1, \sigma_0 \rangle \Downarrow \sigma_1$ and D_2 derives $\langle s_2, \sigma_1 \rangle \Downarrow \sigma$.

(BSeq) is the only rule that constructs a derivation for a configuration whose programme component is a sequential composition, so there must also exist (if $\langle s, \sigma_0 \rangle \Downarrow \sigma'$) a state σ_1' such that $\langle s_1, \sigma_0 \rangle \Downarrow \sigma_1'$ and $\left\langle s_2, \sigma_1' \right\rangle \Downarrow \sigma'$.

By induction hypothesis, we have $\sigma_1 = \sigma_1'$, and therefore also $\sigma' = \sigma$.

Case ($\text{BIf}_\perp$): if

$$(\text{BIf}_\perp)\ \frac{F \qquad D}{\langle s, \sigma_0 \rangle \Downarrow \sigma}$$

then it must be that $s = \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$, $F = ([\![\texttt{B}]\!]^{\mathscr{B}}(\sigma_0) = \perp)$ and D derives $\langle S_2, \sigma_0 \rangle \Downarrow \sigma$.

($\text{BIf}_\perp$) is the only rule that constructs a derivation for a configuration $\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma \rangle$ such that $[\![\texttt{B}]\!]^{\mathscr{B}}(\sigma) = \perp$ (by the fact that $[\![\cdot]\!]^{\mathscr{B}}$ is functional) and we conclude by induction hypothesis.

Other cases are similar.

Safety and Correctness

** 7. For as many of the programmes from questions 1, 2 and 5 as you feel is necessary for you to practice this, find:

1. if it exists, a configuration that leads to non-termination, and a loop invariant that proves non-termination; (recall that it needs to hold when we enter the loop, be preserved by the loop, and imply the loop condition)
2. a configuration that leads to a terminating execution, and a loop variant that proves termination. (Recall that its value needs to be decreased strictly by the loop's body, and must be bounded from below when decreased by the loop.)

Solution

1. Non-termination:

   `exp` The programme given terminates in all configurations. If we had used, for example `while (!1 = m)` as the loop condition, then any state in which `m` has a non-positive value would lead to non-termination, with a loop invariant of the form `m < 1`.

   `factorial` The programme given terminates in all configurations. If we had used, for example `while (!2 = m)` as the loop condition, then any state in which `m` has a non-positive value would lead to non-termination, with a loop invariant of the form `m < 1`.

   `pingala` The programme given terminates in all configurations. other valid answers may not terminate when execution starts in a state where `n` has a negative value—since the question allowed you to assume the value of `n` was initially non-negative.

2. Termination: In all cases, configurations leading to terminating executions were already given in your original answers.

   `exp` $V(\sigma) = [\![\texttt{m - 1}]\!]^{\mathscr{A}}(\sigma)$ is a variant proving termination, lower-bounded by 0. It is also true that $V(\sigma) < 0 \Rightarrow [\![\texttt{1 <= m}]\!]^{\mathscr{B}} = \bot$.

   `factorial` $V(\sigma) = [\![\texttt{m - 2}]\!]^{\mathscr{A}}(\sigma)$ is a variant proving termination, lower-bounded by 0. It is also true that $V(\sigma) < 0 \Rightarrow [\![\texttt{2 <= m}]\!]^{\mathscr{B}} = \bot$.

   `pingala` $V(\sigma) = [\![\texttt{i - 1}]\!]^{\mathscr{A}}(\sigma)$ is a variant proving termination, lower-bounded by 0. Note in particular that `i` takes the initial value of `n` *only if* `n` is initially greater than 1, and it is also true that $1 < [\![\texttt{i}]\!]^{\mathscr{A}}(\sigma) \Rightarrow V(\sigma) < 0 \Rightarrow [\![\texttt{1 <= i}]\!]^{\mathscr{B}}(\sigma) = \bot$

*** 8. Generalise your answers from the previous question as much as possible—identifying *sets* of initial states that lead to non-termination and termination, keeping your loop invariants and variants in mind.

Solution

For the programmes as given, and as already noted, the set of programmes leading to terminating executions is State, which is clear from the fact that our invariants do not rely on the chosen initial values of variables.

*** 9. For as many of the programmes from question 1, 2 and 5 as you feel is necessary for you to practice this skill, find a loop invariant that can be used to prove (partial) correctness of the programme for all initial states identified in the relevant questions.

Solution

Note that the following answers refer to the solutions from the sample solution, not necessarily yours!

`exp` Here, we are only concerned with executions starting in states where the value of `n` is initially non-negative. In those cases and where the initial value of `n` is n and the initial value of `x` is x, the invariant

$$I_{\texttt{exp}} = \left\{\sigma \mid \sigma(\texttt{r}) \cdot \sigma(\texttt{x})^{\sigma(\texttt{m})} \wedge x - \sigma(\texttt{x})\right\}$$

allows us to prove correctness. Indeed, the invariant:

- is initially true when we first enter the loop: with $\sigma(\texttt{r}) = 1$ and $\sigma(\texttt{m}) = n$);
- is preserved by the loop body when the loop condition holds: executing the loop's body starting in a state $\sigma \in \left\{\sigma \in I_{\texttt{exp}} \mid [\![\texttt{1 <= m}]\!]^{\mathscr{B}}(\sigma)\right\}$ terminates in state $\sigma[\texttt{r} \mapsto \sigma(\texttt{r}) \cdot \sigma(\texttt{x}); \texttt{m} \mapsto \sigma(\texttt{m}) - 1]$ and we have $(\sigma(\texttt{r}) \cdot \sigma(\texttt{x})) \cdot \sigma(\texttt{x})^{\sigma(\texttt{m - 1})} = \sigma(\texttt{r}) \cdot \sigma(\texttt{x})^{\sigma(\texttt{m})}$ when $1 \leq \sigma\texttt{m}$; and
- is sufficient to conclude once the loop exits.

`factorial` Here, we are only concerned with executions starting in states where the value of `n` is initially non-negative. In those cases and where the initial value of `n` is n, the invariant

$$I_{\texttt{factorial}} = \{\sigma \mid \sigma(\texttt{r}) \cdot \sigma(\texttt{m})! = n!\}$$

allows us to prove correctness. Indeed, the invariant:

- is initially true when we first enter the loop: with $\sigma(\texttt{r}) = 1$ and $\sigma(\texttt{m}) = n$);
- is preserved by the loop body when the loop condition holds: executing the loop's body starting in a state $\sigma \in \left\{\sigma \in I_{\texttt{factorial}} \mid [\![\texttt{2 <= m}]\!]^{\mathscr{B}}(\sigma)\right\}$ terminates in state $\sigma[\texttt{r} \mapsto \sigma(\texttt{r}) \cdot \sigma(\texttt{m}); \texttt{m} \mapsto \sigma(\texttt{m}) - 1]$ and we have $(\sigma(\texttt{r}) \cdot \sigma(\texttt{m})) \cdot \sigma(\texttt{m - 1})! = \sigma(\texttt{r}) \cdot \sigma(\texttt{m})!$ when $1 \leq \sigma\texttt{m}$; and
- is sufficient to conclude once the loop exits.

`pingala` Here, we are only concerned with executions starting in states where the value of `n` is initially non-negative. The cases: $n = 0$ and $n = 1$ are dealt with separately and are simply correct by definition. We now consider all cases where $2 \leq n$. in those cases, the invariant

$$I_{\texttt{pingala}} = \{\sigma \mid \sigma(\texttt{a}) = p(\sigma(\texttt{n}) - \sigma(\texttt{i})) \wedge \sigma(\texttt{b}) = p(\sigma(\texttt{n}) - \sigma(\texttt{i})) \wedge \sigma(\texttt{n}) = n\}$$

allows us to prove correctness. Justifying this one is a bit more tedious but was also not asked for. (The main additional difficulty in the justification is that the loop is not the final step in the programme!)

Extending the While language

We now extend the While language with arithmetic operators for division (÷) and remainder (%). We call this new language WhileDiv. However, we hate dividing by zero as much as the next language designer.

* 10. Define the syntax of arithmetic and boolean expressions, and that of statements in the WhileDiv language. The only meaningful change will be the addition of the two binary operators (÷ and %) to the syntax or arithmetic expressions. These are left-associative and have the same priority as `*`.

** 11. Define, using derivation rules or functional notation, the semantics of WhileDiv's arithmetic and boolean expressions as a total function in $\mathscr{A} \to \text{State} \to \mathbb{Z} \cup \{✺\}$, where ✺—a big explosion, obviously—is the value of any expression that includes a division (or remainder) by 0, and the value of any expression that does not include a division by 0 is defined as expected.
The semantics of ÷ and % with negative numerators or denominators should be defined, but is up to you. Some definitions make more sense than others. (For example, it is often convenient to have $[\![(\text{-}A_1) \div (\text{-}A_2)]\!]^{\mathscr{B}}(\sigma) = [\![A_1 \div A_2]\!]^{\mathscr{B}}(\sigma)$.)

*** 12. Define, using derivation rules or functional notation, the semantics of WhileDiv's statements as a partial function in $\mathscr{S} \to \mathsf{State} \to \mathbb{Z} \cup \{✷\}$, where ✷ denotes the fact that an execution that tries to evaluate a division by 0 "goes wrong" (or "explodes"). Executions that do not divide by 0 should never terminate with ✷.

**** 13. Show that any WhileDiv programme in which all division and remainder operations have non-zero integer literals as denominators never explodes.