Programming Languages and Computation

Problem Sheet 8: Verifying and Compiling While Programs

We first reason about given While programs, and prove some simple properties.
We then compile While programs down to the Abstract Machine, and beyond.

Compiling While Programs

** 1. Consider the While programs from the Week 5 worksheet (also used in Week 6). Manually compile them to the Abstract Machine language.

Solution

(a)
```
LOAD n; PUSH 2; ADD; STORE n;
```

(b)
```
PUSH 0; LOAD n; LE; IF(LOAD n; STORE r;, PUSH 0; LOAD n; SUB; STORE r;);
```

(c)
```
LOAD a; LOAD b; ADD; STORE b;
LOAD b; LOAD a; SUB; STORE a;
LOAD b; LOAD a; SUB; STORE b;
```

(d)
```
LOOP(LOAD n; PUSH 0; LE; NOT;,
   LOAD n; STORE r;
   LOAD n; PUSH 2; SUB; STORE n;)
```

(e)
```
LOOP(PUSH true;,
   LOAD n; PUSH 1; ADD; STORE n;)
```

You can test your attempts with the `bam` program provided in the labcode archive. Don't forget to initialize the variables meant to serve as inputs before running tests.

At the moment, `bam` reads its input (an Abstract Machine program) from `stdin` and prints any output (final state, or error messages) to `stdout`. In order to invoke it, run:

```
$ cat <filepath> | cabal run bam:bam
```

or

```
$ echo "program" | cabal run bam:bam
```

You can combine `echo` and `cat` to prepend any state initialisation required to pass arguments. For example, the following command will run the factorial program in a state with variable `n` initialised to 12.

```
$ echo "PUSH 12; STORE n;" | cat - bam/test/factn.am | cabal run bam:ba
```

Making Haskell Compile While Programs

In this section, we compile While programs down to the abstract machine's language. (Or rather, we make Haskell do it for us.) Our compiler will translate While ASTs to AM ASTs in a syntax-directed way.

We do so in file while/src/WhileC.hs.

We first write a function to compile arithmetic expressions (type `AExpr`) down to Abstract Machine code (type `Code`).

An Abstract Machine program is just a list of instructions (although some of them currently have more structure), so we can construct them mostly using operations over type `List`. However, those are most efficient when constructing lists from right to left. (Another story of left recursion vs right recursion.)

To allow us to keep thinking about the compilation problem from left to right without losing the (very minor) performance gains of constructing the list from right to left, we write our translator using an accumulator. (In essence, we construct the AM program on our way back up from the recursive calls instead of constructing its beginning, then recursing down to construct the end.)

As an example, you may have already seen how the following naive implementation of list reversal (which has quadratic complexity)

```
rev :: [a] → [a]
rev [] = []
rev (x : xs) = (rev xs) ++ x
```

can be implemented much more efficiently (with linear complexity) as

```
rev :: [a] → [a]
rev = aux []
 where
   aux acc [] = acc
   aux acc (x : xs) = aux (x : acc) xs
```

** 2. The following makes you write the code in its most efficient right-to-left version directly. Feel free to first implement the compiler directly as specified in the reference material, then consider how to make it more efficient by avoiding repeated concatenations with increasingly long prefixes.

(a) Write function `arithToInstrs :: AExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material). When its second argument is some list `c` of AM instructions, then its

result is simply prepended to `c`.

(b) Write function `boolToInstrs :: BExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).

(c) Write function `stmtToInstrs :: Stmt → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).

Solution

See code. The straightforward version has no tricks and is literally a direct implementation of the specification!

Compiling in Restricted Settings

Now that you understand compilation enough to implement it effectively, we can consider what happens if we try to compile to a slightly more restricted language.

We consider a knock-off Abstract Machine (the Knock-Off Machine, below) that can only hold two elements on its stack. We give (part of) its structural operational semantics as a transition relation $\Rightarrow_{\mathsf{KO}}$ (Figure 1).

$$\begin{array}{rll}
\langle \mathtt{PUSH}\ v; c, s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, v : s, \sigma\rangle & & \text{if } |s| \leq 1 \\
\langle \mathtt{ADD}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 + z_2, \sigma\rangle & & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \mathtt{MUL}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 * z_2, \sigma\rangle & & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \mathtt{SUB}; c, z_2 : z_1, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, z_1 - z_2, \sigma\rangle & & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \mathtt{LOAD}\ \mathtt{x}; c, s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, \sigma(\mathtt{x}) : s, \sigma\rangle & & \text{if } |s| \leq 1 \\
\langle \mathtt{STORE}\ \mathtt{x}; c, z : s, \sigma\rangle \Rightarrow_{\mathsf{KO}} \langle c, s, [\mathtt{x} \mapsto z]\sigma\rangle & & \text{if } z \in \mathbb{Z}
\end{array}$$

Figure 1: Semantics for the Knock-Off Machine's arithmetic instructions

*** 3. (From last year's exam.) Define and argue correct a translation function $\mathcal{CA}_{\mathsf{KO}}[\![\cdot]\!]$ from While's *arithmetic expressions only* to the Abstract Machine's bytecode language.
When the source expression a has semantics $[\![\mathtt{a}]\!]_{\mathcal{A}}\,\sigma$ in a given state σ, executing the translated program $\mathcal{CA}_{\mathsf{KO}}[\![a]\!]$ with the empty evaluation stack in state σ should yield a configuration with an empty program, a stack containing exactly one element whose value is $[\![\mathtt{a}]\!]_{\mathcal{A}}\,\sigma$, and a state that must be larger than σ in the sense that it is defined and equal to σ wherever σ is defined, and may also give values to variables not defined in σ. You may choose to extend the domain of the state to inputs that are not valid While variables.
Your translation function may rely on an environment.

Solution

The central idea is to offload overflow stack elements to the state where necessary. In the below, we only use the stack for immediate operations (storing values as soon as possible, and fetching values as late as possible); this is likely not optimal, but optimality was not asked for. Other concrete solutions are possible.

$$\begin{array}{rcl}
\mathscr{C}\mathscr{A}_{\mathsf{KO}}[\![n]\!](\gamma) & = & (\texttt{PUSH}\ n;\texttt{STORE stack}_\gamma, \gamma+1)\\
\mathscr{C}\mathscr{A}_{\mathsf{KO}}[\![\mathtt{x}]\!](\gamma) & = & (\texttt{LOAD x};\texttt{STORE stack}_\gamma, \gamma+1)\\
\mathscr{C}\mathscr{A}_{\mathsf{KO}}[\![a_1+a_2]\!](\gamma) & = & \text{let } (c_1,\gamma_1) = \mathscr{C}\mathscr{A}_{\mathsf{KO}}[\![a_1]\!](\gamma) \text{ in}\\
 & & \text{let } (c_2,\gamma_2) = \mathscr{C}\mathscr{A}_{\mathsf{KO}}[\![a_2]\!](\gamma_1) \text{ in}\\
 & & (c_1;c_2;\texttt{LOAD stack}_{\gamma_1};\texttt{LOAD stack}_{\gamma_2};\texttt{ADD};\texttt{STORE stack}_\gamma,\gamma_2)
\end{array}$$

In this concrete translation (multiplication and subtraction are similar to addition), the $\texttt{stack}_i$ variables are variables not used in the rest of the program, and the environment γ simply keeps track of the stack's depth. Translating an expression simply calls the translation function with 0 as environment, and removes the final environment (which will be 0 as well) before outputting the final program. Note that all intermediate programs produced in fact leave the stack entirely empty.

**** 4. (Optional.) The code repository also contains code for `blam`, an interpreter for the Bristol Less Abstract Machine (BLAM). The BLAM's language is similar to the AM's language, but eschews structured control-flow. Unlike the jump-based abstract machine seen in W8V6, the BLAM does not use labels, instead using `GOTO` and `GOTOF` to (conditionally) jump to the specified line number. Line numbers are 0-indexed (this is just to be slightly annoying).

Implement a compiler from While to BLAM. You can do so either by adapting the existing `whilec` compiler (from Q2), or by implementing a compiler from BAM to BLAM and composing compilation from While to BAM with compilation from BAM to BLAM.

You may want to start by defining semantics for BLAM (you can take a look at the source code to figure it out; we implement configurations of the form $\langle c, pc, s, \sigma\rangle$ by keeping track of the program's code as two lists that are split just before the next instruction to execute. This makes jumps a bit costly, but most linear code a lot cheaper.

Solution

Let's consider a couple of rules of the semantics $\Rightarrow_\downarrow$ for BLAM. We focus only on one simple instruction, and one of the new unstructured control-flow instructions. Configurations have a program, a program counter, a stack and a state.

$$\begin{array}{lll}
\langle c, pc, s, \sigma\rangle & \Rightarrow_\downarrow \langle c, pc+1, z : s, \sigma\rangle & \text{if } c[pc] = \texttt{PUSH}\ z\\
\langle c, pc, \bot : s, \sigma\rangle & \Rightarrow_\downarrow \langle c, pc', s, \sigma\rangle & \text{if } c[pc] = \texttt{GOTOF}\ pc'
\end{array}$$

Compilation is roughly equivalent in difficulty whether it's done from While or AM. Translating from While means we're duplicating quite a lot of the While to AM compilation code, but it does make discussing control-flow slightly easier, so we do this below. To compile from AM, just note that a lot of the counting for straightline components is done, and we just need to do sums on them.

The compilation is only difficult for `if` and `while` statements: we need to know where to jump! In order to know where jump targets should be, we parameterize the compilation function by the line number of the next line we'll produce, and make sure that the number of the next available line is returned by recursive calls. (We essentially keep an instruction counter.)

For arithmetic and boolean expressions, everything is relatively straightforward: we simply count the number of instructions we've emitted so far.

$$\begin{aligned}
\mathscr{C}\mathscr{A}_{\downarrow}[\![n]\!](\ell) &= (\texttt{PUSH}\ n, \ell+1) \\
\mathscr{C}\mathscr{A}_{\downarrow}[\![\ldots]\!](\ell) &= \ldots \\
\mathscr{C}\mathscr{A}_{\downarrow}[\![e_1+e_2]\!](\ell) &= (c_1; c_2; \texttt{ADD}, \ell_2+1) \quad \text{where } (c_1,\ell_1) = \mathscr{C}\mathscr{A}_{\downarrow}[\![e_1]\!](\ell), (c_2,\ell_2) = \mathscr{C}\mathscr{A}_{\downarrow}[\![e_2]\!](\ell_1)
\end{aligned}$$

For assignments, we do the same thing: the code we emit is as before, and we know we emit the instructions to evaluate the right-hand side, plus one for the `STORE`. Things get a bit more exciting for conditionals and loops, but W8V6 is helpful: it shows us what the jumps should look like! We'll just need to be a bit careful when counting: we need to insert jumps, but we don't insert labels. (If it helps, you can always insert `NOOP` instructions to help you count.)

For example, to translate `if` b `then` s_t `else` s_e starting at line ℓ, we therefore:

1. translate the boolean expression b starting at line ℓ, which gives us a program c_b and a new line number ℓ_b; (and we have $\ell_b = \ell + |c_b|$, which is not really relevant, but may help understand what's going on)

2. recursively translate the statement s_t starting at line $\ell_b + 1$ (that's the end of c_b, plus one line to accommodate the conditional jump we need to stick in there!), which gives us a program c_e and a new line number ℓ_t;

3. recursively translate the statement s_e starting at line $\ell_e + 1$ (that's the end of c_e, plus one line to accommodate the unconditional jump over the else branch), which gives us a program c_t and a new line number ℓ_e;

4. output the program c_b; `GOTOF` ℓ_t; c_t; `GOTO` ℓ_e; c_e, and the line number ℓ_e (which is such that $\ell_e = \ell + |c_b| + |c_t| + |c_e| + 2$).

Translating `while` loops is slightly simpler, since there's only one branch. (But still two jumps!)

Instead of implementing this directly, the code solution compiles from BAM to BLAM. This makes things easier in two ways: 1) it avoids duplicating the code that translates *most* of While down to BAM; 2) it makes an efficient implementation possible.

In particular, we cannot use the accumulator tricks from Q2 to avoid quadratic behaviour: this requires compiling the end of the program before compiling the beginning, but we need to know the size of the prefix in order to be able to aim the jumps right. We provide a first implementation which doesn't even try to be clever performance-wise (as `lower` in file `bam/src/BamC.hs`), just to get the ideas right first.

The second implementation makes two *local* passes over the BAM code when translating a control-flow instruction: a first pass that computes the size of the BLAM code that will be emitted when compiling it down, and a second pass that carries out the translation. Note that, in a standard language, this may still lead to quadratic behaviour on some pathological programs (with deeply nested control-flow), but Haskell's laziness will save us here: any size that's already been computed will be recomputed for free as we go down the AST—in reality, we would probably want to enrich the AST structure with this auxiliary information so we rely a bit less on Haskell's tricks.

Reasoning about While Programs

Consider the While program example from Week 1.

```
x := x * x - 1
while !(x = 0) {
  x := x - 2
  y := 2 * y - 1
}
```

* 5.

(a) Specialize the semantics of While to consider only configurations reachable from initial configurations whose statement component is the While program example.

(b) Compare the resulting set of transitions to the transition system described in the original example. (In such a situation, we say that the transition systems *simulate* each other.)

Solution

(a) We start unrolling the semantics with abstract states.

$$\left\langle \begin{array}{l} \texttt{x := x * x - 1} \\ \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \Rightarrow \left\langle \begin{array}{l} \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \left[\mathrm{x} \mapsto \sigma(\mathrm{x})^2 - 1\right]\sigma \right\rangle$$

$$\left\langle \begin{array}{l} \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \Rightarrow \left\langle \begin{array}{l} \texttt{x := x - 2} \\ \texttt{y := 2 * y - 1} \\ \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \qquad \text{when } \sigma(\mathrm{x}) \neq 0$$

$$\left\langle \begin{array}{l} \texttt{x := x - 2} \\ \texttt{y := 2 * y - 1} \\ \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \Rightarrow \left\langle \begin{array}{l} \texttt{y := 2 * y - 1} \\ \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, [\mathrm{x} \mapsto \sigma(\mathrm{x}) - 2]\sigma \right\rangle$$

$$\left\langle \begin{array}{l} \texttt{y := 2 * y - 1} \\ \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \Rightarrow \left\langle \begin{array}{l} \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \left[\mathrm{y} \mapsto 2 \cdot \sigma(\mathrm{y}) - 1\right]\sigma \right\rangle$$

$$\left\langle \begin{array}{l} \texttt{while !(x = 0)} \\ \quad\texttt{x := x - 2} \\ \quad\texttt{y := 2 * y - 1} \end{array}, \sigma \right\rangle \Rightarrow \langle \texttt{skip}, \sigma \rangle \qquad \text{when } \sigma(\mathrm{x}) = 0$$

(b) We can build a one-to-one relation between the (abstract) configurations above (each of which really corresponds to a set of configurations with various states) and the configurations that appear in Steven's transition system in such a way that two related configurations can only transition to related configurations.
In this case, this relation will:

- Replace the statement component with the line number of the first instruction in the initial statement;

- Map every state σ down to only the values $x = \sigma(\mathtt{x})$ and $y = \sigma(\mathtt{y})$ of x and y in that state, ignoring all other variables.

For example, the (concrete) configuration $\left\langle \begin{array}{l}\mathtt{y := 2 * y - 1}\\ \mathtt{while\ !(x = 0)}\\ \quad\mathtt{x := x - 2}\\ \quad\mathtt{y := 2 * y - 1}\end{array}, \begin{bmatrix}\mathtt{x} \mapsto 2\\ \mathtt{y} \mapsto 18\\ \mathtt{a} \mapsto 42\end{bmatrix}\right\rangle$ will be related to the "Steven configuration" $\langle 4, 2, 18\rangle$.

Practicing Backwards Reasoning. Let us now practice reasoning about programs, as illustrated in W8V1. We do so on the implementation S_{div} of Euclidean division shown in Figure 2.

```
q := 0
while (b <= a) {
  q := q + 1
  a := a - b
}
r := a
```

Figure 2: The Euclidean division program S_{div}.

Given an initial state σ such that $0 \leq \sigma(\mathtt{a})$ and $0 <\leq \sigma(\mathtt{b})$, there always exists a state σ' such that $\langle S_{\mathsf{div}}, \sigma\rangle \Rightarrow \langle \mathtt{skip}, \sigma'\rangle$, $\sigma(\mathtt{a}) = \sigma'(\mathtt{q}) \cdot \sigma(\mathtt{b}) + \sigma'(\mathtt{r})$, and $0 \leq \sigma'(\mathtt{r}) < \sigma(\mathtt{b})$.

*** 6. (Optional.) Using the backwards reasoning technique illustrated in the lecture, prove that the correctness property stated above holds.

Ask the lecturers for a loop invariant if you're having trouble finding it, but recall that a loop invariant must:

- Hold at the start of the first loop iteration;
- Be preserved by the loop; and
- Be sufficiently strong after the last loop iteration to ensure correctness of the whole program.

It is often useful to consider the property implied on the loop exit configuration, consider how it is transformed by one loop iteration (propagating backwards as seen in the lecture), and attempting to generalize from there.

Solution

We start by specialising the semantics to describe only transitions relevant to initial configurations whose program component is S_{div}.

$$\left\langle pc_1 = \begin{array}{l}\texttt{q := 0}\\ \texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \Rightarrow \left\langle pc_2 = \begin{array}{l}\texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, [q \mapsto 0]\sigma \right\rangle$$

$$\left\langle pc_2 = \begin{array}{l}\texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \Rightarrow \left\langle pc_3 = \begin{array}{l}\texttt{q := q + 1}\\ \texttt{a := a - b}\\ \texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \qquad \text{when } \sigma(\texttt{b}) \leq \sigma(\texttt{a})$$

$$\left\langle pc_3 = \begin{array}{l}\texttt{q := q + 1}\\ \texttt{a := a - b}\\ \texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \Rightarrow \left\langle pc_4 = \begin{array}{l}\texttt{a := a - b}\\ \texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \left[\texttt{q} \mapsto \sigma(\texttt{q}) + 1\right]\sigma \right\rangle$$

$$\left\langle pc_4 = \begin{array}{l}\texttt{a := a - b}\\ \texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \Rightarrow \left\langle pc_2 = \begin{array}{l}\texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, [\texttt{a} \mapsto \sigma(\texttt{a}) - \sigma(\texttt{b})]\sigma \right\rangle$$

$$\left\langle pc_2 = \begin{array}{l}\texttt{while (b <= a)}\\ \quad\texttt{q := q + 1}\\ \quad\texttt{a := a - b}\\ \texttt{r := a}\end{array}, \sigma \right\rangle \Rightarrow \left\langle pc_5 = \texttt{ r := a }, \sigma \right\rangle \qquad \text{when } \sigma(\texttt{a}) < \sigma(\texttt{b})$$

$$\left\langle pc_5 = \texttt{ r := a }, \sigma \right\rangle \Rightarrow \left\langle \texttt{skip}, [\texttt{r} \mapsto \sigma(\texttt{a})]\sigma \right\rangle$$

In this discussion, we'll look at a slightly weaker correctness property, ignoring the requirement that the remainder be between 0 and b, for simplicity. The reasoning is similar there, but finding the loop invariant requires slightly more creativity.

We are looking for a property Φ of the initial memory (a *precondition*) such that, for any $\sigma_0 \in \Phi$, and for any $\sigma_f \in \Phi$, such that $\langle pc_1, \sigma_0 \rangle \Rightarrow \langle \texttt{skip}, \sigma_f \rangle$, the following property (the *postcondition*, call it Ψ) holds.

$$\sigma_0(\texttt{a}) = \sigma_0(\texttt{b}) \cdot \sigma_f(\texttt{q}) + \sigma_f(\texttt{r})$$

It's worth noting that the precondition $\Phi = \emptyset$ works here. But we're really interested in the *weakest precondition* (i.e. the largest property) such that the postcondition holds.

About termination. We note in passing that the theorem is carefully expressed so we don't have to prove termination, this is known as *partial correctness*. To prove termination, we will usually rely on well-ordering arguments. Here, if $0 < \sigma_0(\texttt{b}) \leq \sigma_0(\texttt{a})$, then we can prove (using backwards reasoning) that the value of a in the state decreases with each loop iteration, and is bounded below by 0. This means the number of iterations has to be finite (or we'd have an infinite stream of strictly decreasing non-negative integers) and the program terminates—the value of a is known as a *loop variant*. (The program also terminates when $\sigma_0(\texttt{a}) < \sigma_0(\texttt{b})$, since it never enters the loop.)

Back to correctness. But what we wanted to do was prove correctness, and we do so by backwards reasoning.

- Any execution that reaches configuration $\langle \texttt{skip}, \sigma_f \rangle$ has to do so from a configuration of the form $\langle pc_5, \sigma' \rangle$, with $\sigma_f = [\texttt{r} \mapsto \sigma'(\texttt{a})]$.

  Therefore, any execution that reaches configuration $\langle \texttt{skip}, \sigma_f \rangle$ with $\sigma_f \in \Psi$ has to do so from a configuration of the form above where $\sigma' \in \Psi'$ is such that the following property holds.

$$\Psi'(\sigma) = \sigma_0(\texttt{a}) = \sigma_0(\texttt{b}) \cdot \sigma(\texttt{q}) + \sigma(\texttt{a})$$

- There are two transitions that could have led to a configuration of the form $\langle pc_2, \sigma \rangle$.

  Let's first look at the one that comes from the loop, and trace it back all the way to the previous configuration of the form $\langle pc_2, \sigma \rangle$. Tracing back through the two transitions in the loop body, we get that any execution that reachs a configuration $\langle pc_2, \sigma \rangle$ with $\Psi'(\sigma)$ was previously in a configuration $\langle pc_2, \sigma' \rangle$ such that $\Psi''(\sigma')$, where Ψ'' is defined below.

$$\Psi''(\sigma) = \sigma_0(\texttt{a}) = \sigma_0(\texttt{b}) \cdot (\sigma(\texttt{q}) - 1) + \sigma(\texttt{a}) + \sigma(\texttt{b})$$

  We note that this is exactly equivalent to Ψ', which makes it a pretty good candidate invariant:

  - It is initially true when we enter the loop (anticipating slightly, we'll later show that we enter the loop in a configuration whose state σ_1 is such that $\sigma_1(\texttt{q}) = 0$);
  - We've just shown that it is invariant under the loop body: if we start the loop body in a state where Ψ'' holds, then we complete its execution in a state where Ψ'' holds; and
  - It is sufficient to complete the proof, since proving that Ψ' holds when we exit the loop is sufficient to prove that Ψ holds in σ_f (as per the previous bullet).

  We do note that this is not an inductive invariant of the transition system: the configuration between both instructions in the loop body breaks it. To make it inductive, we'd need to strengthen it quite a bit.

- We now know that all execution traces that end in a configuration $\langle \texttt{skip}, \sigma_f \rangle$ such that $\Psi(\sigma_f)$ are such that, all the configurations of the form $\langle pc_2, \sigma \rangle$ that appear on the trace have $\Psi'(\sigma)$. Let us now consider the *first* such configuration: it is reached by a transition $\langle pc_1, \sigma_0 \rangle \Rightarrow \langle pc_2, \sigma \rangle$ (otherwise, there is an earlier instance of pc_2 along the trace), with $\sigma = [\texttt{q} \mapsto 0]\sigma_0$.

  And then we have $\Psi''(\sigma) \Leftrightarrow \Phi(\sigma_0)$, when

$$\Phi(\sigma) = \top$$

That is, whenever the program terminates, and from *any* initial state, it terminates in a state where Ψ holds.

(Now, adding the condition that the final remainder must be between 0 and b does change this quite a bit, and I'm happy to discuss it. But writing it up is a right PITA.)