FLOLAC 2020

Programming Language Theory

Imperative Language Constructs

Materials are based on Charguéraud (2020)

游書泓

2020 邏輯、語言與計算暑期研習營

Imperative Programming

Imperative Programming

Mutable references:

- `ref x`, `newIORef x` (creating a new mutable cell),
- `!r`, `readIORef r` (reading a mutable cell) and
- `r := e`, `writeIORef r e` (mutating the cell content)

Sequencing and Looping:

Example

```
let a = !x in
x := !y;
y := a
```

Example

```
while !n <> 0 do
  prod := !prod * !n;
  n := !n - 1
done
```

The Idea

We have been using a single piece of program term to model the execution of programs. The meaning of a program is directly given by the value it reduces to.

$$\to \subset \text{Term} \times \text{Term}$$
$$e = e_1 \to e_2 \to \cdots \to e_{n-1} \to e_n = v$$

To model mutable states, we introduce the notion of stores, which are finite maps from locations to values. The reduction relation now operates on pairs of program terms and stores, and the meaning of a program is given by the value it reduces to and the content of the final store.

$$\sigma : (l \in \text{Loc}) \longmapsto v$$
$$(e, \sigma) = (e_1, \sigma_1) \to (e_2, \sigma_2) \to \cdots \to (e_n, \sigma_n) = (v, \sigma')$$

The Basic Language

e	$::=$	$c \mid e \odot e \mid x \mid e\, e \mid \lambda x.\, e$	Expressions
		$\mid$ if e then e else e	
		$\mid$ let $x = e$ in e	
v	$::=$	$c \mid \lambda x.\, e$	Values; must be closed.
$\odot$	$::=$	$+ \mid - \mid \times \mid = \mid \neq$	Primitive Operators
c	$::=$	$() \mid \text{true} \mid \text{false} \mid n$	Constants: unit, booleans and integers ($n \in \mathbb{Z}$)
σ	$:$	$l \longmapsto v$	Store

We specifically build let expressions into the language to simplify the reasoning of the programs. Conceptually, they behave the same as immediate function applications.

$$(\text{let } x = e_1 \text{ in } e_2) \text{ is like } (\lambda x.\, e_2)\, e_1$$

Imperative Programming

Big-step Semantics

Imperative Programming Constructs

Axiomatic Semantics

Big-step Semantics (Natural Semantics)

Big-step semantics is a natural deduction style operational semantics that directly associates a program with its meaning (value). Its judgment has the following form.

$$e, \sigma \Downarrow v, \sigma'$$

The judgment means that the evaluation of the program e with the initial store σ *terminates* with value v and final store σ'.

In contrast, the structural operational semantics we have been using proceeds by making a series of reduction steps until reaching a value.

$$(e, \sigma) \to \cdots \to (v, \sigma')$$

Evaluating Values and Primitive Operators

$$\frac{}{c, \sigma \Downarrow c, \sigma}\ [\text{E-VAL}] \qquad \frac{}{(\lambda x.\, e), \sigma \Downarrow (\lambda x.\, e), \sigma}\ [\text{E-FUN}]$$

$$\frac{e_1, \sigma \Downarrow v_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v_2, \sigma_2 \qquad \delta(\odot, v_1, v_2) = v}{e_1 \odot e_2, \sigma \Downarrow v, \sigma_2}\ [\text{E-OP}]$$

Constants and functions evaluate to themselves. We give meanings to the primitive operators through the δ-function.

Example

Let D denote

$$\frac{\dfrac{}{5, \sigma \Downarrow 5, \sigma} \qquad \dfrac{}{3, \sigma \Downarrow 3, \sigma} \qquad \delta(+, 5, 3) = 8}{5 + 3, \sigma \Downarrow 8, \sigma}$$

then

$$\frac{D \qquad \dfrac{}{8, \sigma \Downarrow 8, \sigma} \qquad \delta(=, 8, 8) = \text{true}}{5 + 3 = 8, \sigma \Downarrow \text{true}, \sigma}$$

Evaluating Function Applications

$$\frac{e_1, \sigma \Downarrow (\lambda x.\, e), \sigma_1 \qquad e_2, \sigma_1 \Downarrow v, \sigma_2 \qquad e[v/x], \sigma_2 \Downarrow v', \sigma'}{(e_1\, e_2), \sigma \Downarrow v', \sigma'}\ [\text{E-App}]$$

The application rule resembles β-reduction in structural operational semantics except the subexpressions are first evaluated to values.

Exercise. Let T denote $\lambda f x.\, f\,(f\,x)$ and G denote $\lambda z.\, (z \times 2)$. Derive $TG3, \sigma \Downarrow 12, \sigma$.

$$\frac{\dfrac{\vdots}{TG, \sigma \Downarrow \lambda x.\, G(Gx), \sigma} \qquad \dfrac{}{3, \sigma \Downarrow 3, \sigma} \qquad \dfrac{\vdots}{G(G3), \sigma \Downarrow 12, \sigma}}{TG3, \sigma \Downarrow 12, \sigma}$$

Question. How does the term $(\lambda x.\, xx)\,(\lambda x.\, xx)$ evaluate?

Evaluating Let Expressions

Find the evaluation rule for call-by-value let expressions.

$$\frac{e_1, \sigma \Downarrow v_1, \sigma_1 \qquad e_2[v_1/x], \sigma_1 \Downarrow v_2, \sigma_2}{(\text{let } x = e_1 \text{ in } e_2), \sigma \Downarrow v_2, \sigma_2} \text{[E-Let]}$$

Consider the following typing rule that inlines the expression e_1 in e_2, supporting polymorphic types naturally:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2[e_1/x] : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

Is this rule sound in the arithmetic language? What about the current imperative language?

Evaluatings Conditional Expressions

$$\frac{e, \sigma \Downarrow \text{true}, \sigma' \qquad e_1, \sigma' \Downarrow v, \sigma''}{(\text{if } e \text{ then } e_1 \text{ else } e_2), \sigma \Downarrow v, \sigma''} \text{ [E-IF1]}$$

$$\frac{e, \sigma \Downarrow \text{false}, \sigma' \qquad e_2, \sigma' \Downarrow v, \sigma''}{(\text{if } e \text{ then } e_1 \text{ else } e_2), \sigma \Downarrow v, \sigma''} \text{ [E-IF2]}$$

Conditional expressions have two evaluation rules, corresponding to the two branching choices.

Example

$$\dfrac{\dfrac{\dfrac{}{5, \sigma \Downarrow 5, \sigma} \quad \dfrac{}{3, \sigma \Downarrow 3, \sigma} \quad \delta(=, 5, 3) = \text{false}}{5 = 3, \sigma \Downarrow \text{false}, \sigma} \quad \dfrac{}{\text{true}, \sigma \Downarrow \text{true}, \sigma}}{(\text{if } 5 = 3 \text{ then false else true}), \sigma \Downarrow \text{true}, \sigma}$$

References

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{ref}\, e \mid !e \mid e := e \mid l \\ v & ::= & \ldots \mid l \end{array}$$

We extend the syntax of programs and the definition of values to include locations ($l \in \mathrm{Loc}$).

$$\frac{e, \sigma \Downarrow v, \sigma' \qquad l \notin \mathrm{dom}(\sigma') \qquad \sigma'' = \sigma'[l := v]}{(\mathsf{ref}\, e), \sigma \Downarrow l, \sigma''}\ [\text{E-Ref}]$$

The evaluation of the expression ref e allocates a new mutable cell, rendered as the generation of a fresh location in the store.

The expression $\sigma'[l := v]$ denotes functional store update. It creates a new store σ'' with domain $\mathrm{dom}(\sigma') \cup \{l\}$ such that

$$\sigma''(l') = \begin{cases} v & \text{if } l' = l \\ \sigma'(l') & \text{otherwise} \end{cases}$$

References

$$\frac{e, \sigma \Downarrow l, \sigma' \qquad \sigma'(l) = v}{!e, \sigma \Downarrow v, \sigma'} \text{ [E-Get]}$$

$$\frac{e_1, \sigma \Downarrow l, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v, \sigma_2 \qquad l \in \mathrm{dom}(\sigma_1) \;\; \sigma' = \sigma_2[l := v]}{(e_1 := e_2), \sigma \Downarrow (), \sigma'} \text{ [E-Set]}$$

Reading and writing mutable cells correspond to querying and updating the store, respectively.

Example

let $x =$ ref false in
let $y =$ ref false in
(if (!*secret*) then x else y) $:=$ true

Question. In [E-Set], is the condition $l \in \mathrm{dom}(\sigma_1)$ necessary? Will $l \in \mathrm{dom}(\sigma_2)$?

Sequencing

$$e \quad ::= \quad \ldots \mid e; e$$

$$\frac{e_1, \sigma \Downarrow v_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v_2, \sigma_2}{(e_1; e_2), \sigma \Downarrow v_2, \sigma_2} \text{ [E-Seq]}$$

The sequencing operator evaluates the two subexpressions in order, discarding the value of the first subexpression and returns the value of the second subexpression.

Exercise. Evaluate the following term. The term n is any integer. The symbol $\cdot$ denotes the empty store.

$$\left(\begin{array}{l} \text{let } r = \text{ref } n \text{ in} \\ (r := !r + 1);\ !r \end{array}\right), \cdot \Downarrow n + 1, \cdot[l := n + 1]$$

While Loops

$$e \quad ::= \quad \ldots \mid \text{while } e \text{ do } e$$

$$\frac{(\text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else } ()), \sigma \Downarrow v, \sigma'}{(\text{while } e_1 \text{ do } e_2), \sigma \Downarrow v, \sigma'} \text{ [E-While]}$$

The evaluation of while loops unfolds one iteration at a time, duplicating the body of the loop.

Exercise. Prove that the following rule is admissible.

$$\frac{e_1, \sigma \Downarrow \text{true}, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v, \sigma_2 \qquad (\text{while } e_1 \text{ do } e_2), \sigma_2 \Downarrow v', \sigma'}{(\text{while } e_1 \text{ do } e_2), \sigma \Downarrow v', \sigma'}$$

Axiomatic Semantics

Pre-conditions and Post-conditions

Let the notation "$(l_{\mathsf{i}} \hookrightarrow v_1) \star (l_{\mathsf{prod}} \hookrightarrow v_2)$" denote a *predicate* asserting that the store maps the location l_{i} to the value v_1, the location l_{prod} to the value v_2 and nothing else.

Precondition: $[n > 0] \star (l_{\mathsf{i}} \hookrightarrow n) \star (l_{\mathsf{prod}} \hookrightarrow 1)$
while $!l_{\mathsf{i}} \neq 0$ do
$(l_{\mathsf{prod}} := !l_{\mathsf{prod}} \times !l_{\mathsf{i}};$
$l_{\mathsf{i}} := !l_{\mathsf{i}} - 1)$
Postcondition: $[n > 0] \star (l_{\mathsf{i}} \hookrightarrow 0) \star (l_{\mathsf{prod}} \hookrightarrow n!)$

The behavior of the factorial program can then be characterized by asserting the status of the store *before* and *after* running the program.

Axiomatic Semantics

The Idea and Informal Reasoning

A More Formal Account

Store Predicate Combinators

Store predicates accept the stores that satisfy the described properties. We will use the following language to express store predicates.

The variable H ranges over store predicates.

$[]$	The store is empty.
$[P]$	The store is empty. Moreover, the proposition P holds.
$(l \hookrightarrow v)$	The store maps l to v and nothing else.
$H_1 \star H_2$	The store can be split into two disjoint pieces. The first piece satisfies H_1 while the second piece satisfies H_2.
$\forall x.\, H$	For all x, H is a store predicate.
$\exists x.\, H$	There exists x such that H is a store predicate.

Annotating Programs for Informal Reasoning

Between every lines in the program, we annotate the store predicate that should hold at the point and follow several informal rules to check if the annotations make sense.

Precondition: $\{\, (l_x \hookrightarrow v_1) \star (l_y \hookrightarrow v_2) \star (l_{tmp} \hookrightarrow \_) \,\}$

$l_{tmp} := !l_x;$

$\{\, (l_x \hookrightarrow v_1) \star (l_y \hookrightarrow v_2) \star (l_{tmp} \hookrightarrow v_1) \,\}$

$l_x := !l_y;$

$\{\, (l_x \hookrightarrow v_2) \star (l_y \hookrightarrow v_2) \star (l_{tmp} \hookrightarrow v_1) \,\}$

$l_y := !l_{tmp}$

Postcondition: $\{\, (l_x \hookrightarrow v_2) \star (l_y \hookrightarrow v_1) \star (l_{tmp} \hookrightarrow v_1) \,\}$

Simple Expressions and Assignments

$$a \quad ::= \quad n \mid a \odot a \mid !l$$

For expressions restricted to the form of a, it is relatively easy to compute their values. Given any such expression a and store predicate H, we use the notation $[\![a]\!]_H$ to denote the value of a under *any* store satisfying H. It is an error if H cannot prove that a has a single value.

We can now write down the shape of annotations for assignment expressions.

$$\begin{array}{l} \{\,(l \hookrightarrow v) \star H'\,\} \\ l := a \\ \{\,(l \hookrightarrow [\![a]\!]_H) \star H'\,\} \end{array}$$

where $H = (l \hookrightarrow v) \star H'$.

Conditional Expressions

$$a \quad ::= \quad n \mid a \odot a \mid !l$$

For conditional expressions, the value of the test expression is known inside the two branches.

$$
\begin{array}{l}
\quad \{\, H \,\} \\
(\text{if } a \\
\quad \text{then} \quad (\{\, [\![a]\!]_H = \text{true}] \star H \,\} \\
\qquad\qquad\quad e_1 \\
\qquad\qquad\quad \{\, H' \,\}) \\
\quad \text{else} \quad (\{\, [\![a]\!]_H = \text{false}] \star H \,\} \\
\qquad\qquad\quad e_2 \\
\qquad\qquad\quad \{\, H' \,\})) \\
\quad \{\, H' \,\}
\end{array}
$$

While Loops

$$a \quad ::= \quad n \mid a \odot a \mid !l$$

Since while expressions can loops for an unknown number of times, we need to find an *invariant* that remains true across the loop to see that the program is correct under *any* number of iterations.

We use I to denote loop invariants. Currently, they are just store predicates.

$$\begin{array}{l} \{\, I \,\} \\ (\text{while } a \text{ do} \\ \qquad (\{\, [\![a]\!]_I = \text{true}] \star I \,\} \\ \qquad\; e \\ \qquad\; \{\, I \,\})) \\ \{\, [\![a]\!]_I = \text{false}] \star I \,\} \end{array}$$

While Loops

Example

In the following program, we can use the invariant I_{fact}.

$$I_{\text{fact}} \equiv$$

$$\exists m.\, [0 \leq m \leq n] \star [n > 0] \star (l_{\text{i}} \hookrightarrow m) \star \left(l_{\text{prod}} \hookrightarrow \frac{n!}{m!} \right)$$

Precondition: $[n > 0] \star (l_{\text{i}} \hookrightarrow n) \star (l_{\text{prod}} \hookrightarrow 1)$

while $!l_{\text{i}} \neq 0$ do

$(l_{\text{prod}} := !l_{\text{prod}} \times !l_{\text{i}};$

$l_{\text{i}} := !l_{\text{i}} - 1)$

Postcondition: $[n > 0] \star (l_{\text{i}} \hookrightarrow 0) \star (l_{\text{prod}} \hookrightarrow n!)$

Axiomatic Semantics

The Idea and Informal Reasoning

A More Formal Account

The Hoare Triple

Generally, since our imperative language is based on expressions, the postconditions will be store predicates that additionally accept the evaluation result. We use Q to range over postconditions.

- Preconditions: $H : \text{Store} \to \text{Set}$
- Postconditions: $Q : \text{Value} \to (\text{Store} \to \text{Set})$

The Hoare triple is defined as the logic proposition

$$\{\!\{H\}\!\}\ e\ \{\!\{Q\}\!\} \equiv$$
$$\forall \sigma.\, (H\,\sigma \ \to\ (\exists v \sigma'.\, (e, \sigma \Downarrow v, \sigma') \wedge Q\,v\,\sigma'))$$

If the Hoare triple $\{\!\{H\}\!\}\ e\ \{\!\{Q\}\!\}$ is proved, the evaluation of e under any store satisfying H will terminate with some value v and store σ' such that $Q\,v\,\sigma'$ holds.

Store Predicate Combinators

The store predicates are synonyms of the functions on the right hand side.

$$
\begin{array}{lcl}
[] & \equiv & \lambda\sigma.\,(\sigma = \cdot) \\
[P] & \equiv & \lambda\sigma.\,P \wedge (\sigma = \cdot) \\
(l \hookrightarrow v) & \equiv & \lambda\sigma.\,\mathrm{dom}(\sigma) = \{l\} \wedge \sigma(l) = v \\
H_1 \star H_2 & \equiv & \lambda\sigma.\,\exists\sigma_1\sigma_2.\,\sigma = \sigma_1 \uplus \sigma_2 \wedge H_1\,\sigma_1 \wedge H_2\,\sigma_2 \\
\forall\!\!\forall x.\,H & \equiv & \lambda\sigma.\,(\forall x.\,H\,\sigma) \\
\exists\!\!\exists x.\,H & \equiv & \lambda\sigma.\,(\exists x.\,H\,\sigma) \\
Q \star_{\mathrm{v}} H & \equiv & \lambda r.\,(Q\,r \star H)
\end{array}
$$

The disjoint sum of the store $\sigma = \sigma_1 \uplus \sigma_2$ means that $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)$, $\mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$ and

$$
\sigma(l) = \begin{cases} \sigma_1(l) & \text{if } l \in \mathrm{dom}(\sigma_1) \\ \sigma_2(l) & \text{if } l \in \mathrm{dom}(\sigma_2) \end{cases}
$$

Consequence Rule

Formally, the judgment $\Gamma \vdash \{\!\{H\}\!\}\ e\ \{\!\{Q\}\!\}$ asserts that the Hoare triple is provable in the context Γ. Here, the context can contain any other variable bindings and propositions as in Agda.

As a simplest example, the following consequence rule is provable and allows us to strengthen the precondition and weaken the postcondition.

$$\frac{\Gamma \vdash (H \vdash_s H') \qquad \Gamma \vdash \{\!\{H'\}\!\}\ e\ \{\!\{Q'\}\!\} \qquad \Gamma \vdash (Q' \vdash_v Q)}{\Gamma \vdash \{\!\{H\}\!\}\ e\ \{\!\{Q\}\!\}}\ [\textsc{Conseq}]$$

The entailment relation $\vdash_s$ and $\vdash_v$ are again abbreviations.

- Store predicate entailment: $H_1 \vdash_s H_2 \ \equiv\ \forall \sigma.\, H_1\, \sigma \to H_2\, \sigma$
- Postcondition entailment: $Q_1 \vdash_v Q_2 \ \equiv\ \forall v.\, Q_1\, v \vdash_s Q_2\, v$

Primitive Operators and Let Expressions

To simplify the reasoning process, we only consider programs where the computation results of all expressions are being bound by let expressions. For example, the expression $(a + b) \times (c \times d)$ can be transformed into

$$\text{let } x = a + b \text{ in let } y = c \times d \text{ in } x \times y$$

Now, the reasoning rule of Hoare triples can be given as

$$\frac{}{\Gamma \vdash \{\!\{H\}\!\}\ v_1 \odot v_2\ \{\!\{\lambda r.\, [r = v_1 \odot v_2] \star H\}\!\}}\ [\textsc{Op}]$$

$$\frac{\Gamma \vdash \{\!\{H\}\!\}\ e_1\ \{\!\{Q'\}\!\} \qquad \Gamma \vdash \forall v.\, \{\!\{Q'\ v\}\!\}\ e_2[v/x]\ \{\!\{Q\}\!\}}{\Gamma \vdash \{\!\{H\}\!\}\ \text{let } x = e_1 \text{ in } e_2\ \{\!\{Q\}\!\}}\ [\textsc{Let}]$$

The rule for let expressions is of particular importance to threading values in expressions. It allows store predicates to depend on the evaluation results.

Functions and Values

$$\frac{\Gamma \vdash (H \vdash_s (Q\, v))}{\Gamma \vdash \{\!\{H\}\!\}\, v\, \{\!\{Q\}\!\}}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash v_1 = \lambda x.\, e \qquad \Gamma \vdash \{\!\{H\}\!\}\, e[v_2/x]\ \{\!\{Q\}\!\}}{\Gamma \vdash \{\!\{H\}\!\}\, v_1\, v_2\, \{\!\{Q\}\!\}}\ [\textsc{App}]$$

The rule for functions is a direct translation of its big-step semantics.

References

$$\frac{}{\Gamma \vdash \{\!\{(l \hookrightarrow v) \star H\}\!\}\ !l\ \{\!\{\lambda r.\, [r = v] \star (l \hookrightarrow v) \star H\}\!\}}\ [\textsc{Get}]$$

$$\frac{}{\Gamma \vdash \{\!\{(l \hookrightarrow v) \star H\}\!\}\ l := v'\ \{\!\{\lambda \_.\, (l \hookrightarrow v') \star H\}\!\}}\ [\textsc{Set}]$$

$$\frac{}{\Gamma \vdash \{\!\{H\}\!\}\ \mathrm{ref}\ v\ \{\!\{\lambda r.\, \exists l.\, [r = l] \star (l \hookrightarrow v) \star H\}\!\}}\ [\textsc{Ref}]$$

The inference rules for reading and writing references are also straightforward. In the inference rule for reference allocation, we use the existential quantifier and the star operator to capture the freshness of the location.

Conditional Expression

$$\frac{\begin{array}{l}\Gamma \vdash \{\!\{H\}\!\}\ e\ \{\!\{\lambda r.\,(\exists b.\,[r = b] \star Q'\,b)\}\!\} \\ \Gamma \vdash \{\!\{Q'\,\mathsf{true}\}\!\}\ e_1\ \{\!\{Q\}\!\} \\ \Gamma \vdash \{\!\{Q'\,\mathsf{false}\}\!\}\ e_2\ \{\!\{Q\}\!\}\end{array}}{\Gamma \vdash \{\!\{H\}\!\}\ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \{\!\{Q\}\!\}}\ [\textsc{If}]$$

In the inference rule for conditional expressions, the postcondition Q' depends on the evaluation result of the test expression e.

The very same postcondition becomes the precondition for the subexpressions e_1 and e_2, but with the evaluation result of the test expression supplied.

Sequencing and While Loops

$$\frac{\Gamma \vdash \{\!\{H\}\!\}\ e_1\ \{\!\{\lambda\_.\, H'\}\!\} \qquad \Gamma \vdash \{\!\{H'\}\!\}\ e_2\ \{\!\{Q\}\!\}}{\Gamma \vdash \{\!\{H\}\!\}\ e_1; e_2\ \{\!\{Q\}\!\}}\ [\textsc{Seq}]$$

The inference rule for the sequencing operator discards the evaluation result of the first subexpression.

$$\frac{\begin{array}{l}\Gamma \vdash \forall bn.\, \{\!\{I\ b\ n\}\!\}\ e_1\ \{\!\{\lambda r.\, (\exists b'.\, [r = b'] \star I\ b'\ n)\}\!\} \\ \Gamma \vdash \forall n.\, \{\!\{I\ \mathsf{true}\ n\}\!\}\ e_2\ \{\!\{\lambda\_.\, (\exists b' n'.\, [n' \leq n] \star I\ b'\ n')\}\!\}\end{array}}{\Gamma \vdash \{\!\{\exists bn.\, I\ b\ n\}\!\}\ \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \{\!\{\lambda\_.\, (\exists n.\, I\ \mathsf{false}\ n)\}\!\}}\ [\textsc{While}]$$

- Loop invariant: $I : \mathbb{B} \to \mathbb{N} \to (\mathrm{Store} \to \mathrm{Set})$

In while loops the loop invariant I takes extra arguments to accept the evaluation result of the test expression as in [If]. Further more, I depends on a natural number in order to ensure the termination of the while loop.

References

1. Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). In *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (August 2020), 43 pages. https://doi.org/10.1145/3408998