Verification of Imperative Programs: The VeriFast Approach. A Draft Course Text

Bart Jacobs *Jan Smans* *Frank Piessens*

Report CW 578, March 2010

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verification of Imperative Programs: The VeriFast Approach. A Draft Course Text

Bart Jacobs *Jan Smans* *Frank Piessens*

Report CW 578, March 2010

Department of Computer Science, K.U.Leuven

Abstract

This draft course text presents a formalization and soundness proof of a core subset of the VeriFast approach for verification of imperative programs.

Verification of Imperative Programs: The VeriFast Approach

A Draft Course Text

Bart Jacobs, Jan Smans, and Frank Piessens
DistriNet, Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract

This draft course text presents the formal, mathematical underpinnings of the VeriFast approach, an approach for verifying certain correctness properties of computer programs written in an imperative programming language. We will elaborate the approach for a simple C-like language, but it is equally applicable to C, C++, Java, or C#. The approach enables programmers to ascertain the absence of invalid memory accesses, including null pointer dereferences and out-of-bounds array accesses, as well as compliance with programmer-specified routine preconditions and postconditions. The approach performs modular verification: each routine is verified using only the contracts, not the implementations, of other routines. Verification proceeds by symbolic execution, i.e. execution using symbolic states instead of concrete states. Values are represented using logical symbols and terms; memory is represented as a bag of symbolic heap elements.

The text first introduces the syntax and the semantics, i.e. the run-time behavior, of the programming language. The run-time behavior is expressed in terms of *concrete executions*. Then, the syntax and the meaning of preconditions and postconditions, and other annotations, is presented, and the notion of *abstracted execution* is defined. Finally, in the third part, symbolic execution is introduced.

If during a given concrete execution, a program performs an illegal memory access, we say the concrete execution *goes wrong*. The goal of verification is to check that no concrete execution goes wrong. However, a program may have an infinite number of concrete executions, and each concrete execution may be infinitely long. Therefore, this cannot be checked directly. Abstracted execution deals with the problem of infinite-length concrete executions by using routine preconditions and postconditions to execute each routine in isolation, and by using loop invariants to execute each loop iteration in isolation. As a result, even though there are still infinitely many abstracted executions due to an infinite state space, the length of abstracted routine executions is linear in the size of the routine. Abstracted execution further has the advantage of being modular. In a final step, the problem of the infinite state space is solved by moving from abstracted executions to symbolic executions. A single symbolic state can be used to represent an infinite number of abstracted states. Using this property, we can verify a program by checking a finite number of finite-length symbolic executions.

This text assumes readers already have some intuitive understanding of the VeriFast approach; for this, we refer them to the VeriFast Tutorial on the VeriFast website. This text does not cite related work; a brief overview of related work is given in the tool paper *The VeriFast Program Verifier*, TR CW-520, 2008.

1 Programs and concrete execution

In this section, we define the syntax and the meaning of the simple C-like programming language used in this text to explain the verification approach. Section 1.1 defines the syntax; Section 1.2 defines the meaning.

1.1 Syntax

The syntax of arithmetic expressions e, boolean expressions b, commands c, routine definitions def, and programs *program* is given by the following grammar:

x	::=	*one of* $\mathsf{i}, \mathsf{counter}, \mathsf{result}, \ldots$	program variable name
n	::=	*one of* $0, 1, -1, 2, -2, 3, -3, \ldots$	integer literal
r	::=	*one of* $\mathsf{increment}, \mathsf{swap}, \ldots$	routine name
e	::=		arithmetic expression
		x	program variable expression
		$\mid n$	literal expression
		$\mid (e + e)$	addition expression
		$\mid (e - e)$	subtraction expression
b	::=		boolean expression
		$e = e$	equality expression
		$\mid e < e$	inequality expression
		$\mid (b \wedge b)$	conjunction expression
		$\mid (b \vee b)$	disjunction expression
		$\mid \neg b$	negation expression
c	::=		command
		$x := e$	assignment command
		$\mid (c; c)$	sequential composition command
		$\mid$ **if** b **then** c **else** c	conditional command
		$\mid$ **while** b **do** c	loop command
		$\mid x :=$ **cons**$(e, \ldots, e)$	allocation command
		$\mid x := [e]$	lookup command
		$\mid [e] := e$	mutation command
		$\mid$ **dispose** e	de-allocation command
		$\mid$ **skip**	skip command
		$\mid x := r(e, \ldots, e)$	routine call command
def	::=	$r(x, \ldots, x) := c$	routine definition
program	::=	*def* ... *def* c	program

The grammar defines the *grammar symbols* x, n, r, e, b, c, *def*, and *program*. We say a text string is an *instance* of a grammar symbol if it matches the grammar for the symbol. We call the *extent* $||\sigma||$ of a grammar symbol σ the set of text strings that match it, i.e. the set of its instances:

$$
\begin{aligned}
||x|| &= \{\mathsf{i}, \mathsf{counter}, \mathsf{result}, \ldots\} \\
||n|| &= \mathbb{Z} \\
||r|| &= \{\mathsf{increment}, \mathsf{swap}, \ldots\} \\
||e|| &= \{\mathsf{i}, \mathsf{counter}, 5, -7, (\mathsf{i} + 5), (7 - (\mathsf{i} + \mathsf{counter})), \ldots\} \\
||b|| &= \{5 = 7, \mathsf{i} = 3, \mathsf{counter} < -7, (5 = 7 \wedge \mathsf{i} = 3), (5 = 7 \vee \mathsf{i} = 3), \neg 5 = 7, \ldots\} \\
||c|| &= \{\mathsf{i} := 5, \mathsf{i} := (\mathsf{i} + 1), (\mathsf{i} := 5; \mathsf{i} := (\mathsf{i} + 1)), \\
&\quad \textbf{if } \mathsf{i} < 0 \textbf{ then } \mathsf{i} := (0 - \mathsf{i}) \textbf{ else skip}, \\
&\quad \textbf{while } \mathsf{i} < 10 \textbf{ do } (\mathsf{i} := (\mathsf{i} + 1); \mathsf{counter} := (\mathsf{counter} + 2)), \\
&\quad \mathsf{point} := \textbf{cons}(2, 3), \mathsf{x} := [\mathsf{point}], \mathsf{y} := [(\mathsf{point} + 1)], \\
&\quad [\mathsf{point}] := (\mathsf{x} + 1), [(\mathsf{point} + 1)] := (\mathsf{y} + 1), \textbf{dispose } \mathsf{point}, \\
&\quad \mathsf{result} := \mathsf{increment}(\mathsf{counter}), \ldots\} \\
||\mathit{def}|| &= \{\mathsf{increment}(\mathsf{ctr}) := (\mathsf{v} := [\mathsf{ctr}]; ([\mathsf{ctr}] := (\mathsf{v} + 1); \mathsf{result} := (\mathsf{v} + 1))), \ldots\} \\
||\mathit{program}|| &= \{\mathsf{double}(\mathsf{a}) := \mathsf{result} := (\mathsf{a} + \mathsf{a})\ \mathsf{b} := \mathsf{double}(10), \ldots\}
\end{aligned}
$$

We will use subscripted grammar symbols $\sigma_1, \sigma_2, \ldots$ and even σ itself to denote instances of grammar symbol σ. For example, e_1 and e denote arbitrary expressions.

We will use the shorthand $e_1 + e_2 - e_3 + e_4$ for $(((e_1 + e_2) - e_3) + e_4)$. That is, we treat $+$ and $-$ as mutually *left-associative*. We will use the shorthand $b_1 \wedge b_2 \wedge b_3 \vee b_4 \vee b_5 \wedge b_6$ for $((((b_1 \wedge b_2) \wedge b_3) \vee b_4) \vee (b_5 \wedge b_6))$. That is, we treat conjunction and disjunction as left-associative, but conjunction *binds more tightly* than disjunction. Finally, we will use the shorthand $c_1; c_2; c_3$ for $(c1; (c2; c3))$. That is, we treat sequential composition as *right-associative*.

1.2 Meaning

The meaning, or *semantics*, of a program can be given by giving the set of its *executions*. An execution can be defined as a finite or infinite sequence of *configurations*, each related to the next by an *execution step*. A configuration is given by a *state* and a *continuation*. A state consists of a *store* and a *heap*. The store specifies the values of the program variables. The heap specifies the values of the memory cells. The continuation specifies what happens next. It is either a *command continuation*, a *return continuation*, or a *done continuation*.

1.2.1 Configurations

We define the set of stores *Stores* as the set of total functions from program variable names to integers:

$$\mathit{Stores} = ||x|| \to \mathbb{Z}$$

A function is *total* if it is defined for all arguments; we require that each store assigns a value to each program variable name.

We use the following shorthand notation for stores that assign the value 0 to all but a finite number of program variable names:

$$\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \_ \mapsto 0\}(x) = \begin{cases} v_1 & \text{if } x = x_1 \\ \cdots & \\ v_n & \text{if } x = x_n \\ 0 & \text{otherwise} \end{cases}$$

Example 1. *The store* $\{\mathsf{i} \mapsto 1, \mathsf{counter} \mapsto 5, \_ \mapsto 0\}$ *assigns the value 1 to program variable name* i*, 5 to* counter*, and 0 to all other names:*

$$\begin{gathered} \{\mathsf{i} \mapsto 1, \mathsf{counter} \mapsto 5, \_ \mapsto 0\}(\mathsf{i}) = 1 \\ \{\mathsf{i} \mapsto 1, \mathsf{counter} \mapsto 5, \_ \mapsto 0\}(\mathsf{counter}) = 5 \\ \{\mathsf{i} \mapsto 1, \mathsf{counter} \mapsto 5, \_ \mapsto 0\}(\mathsf{a}) = 0 \\ \{\mathsf{i} \mapsto 1, \mathsf{counter} \mapsto 5, \_ \mapsto 0\}(\mathsf{b}) = 0 \\ \cdots \end{gathered}$$

We define the set of *addresses* as the set of positive integers:

$$\mathit{Addresses} = \{n \in \mathbb{Z} \mid 0 < n\}$$

We define the set of *heaps* as the set of finite partial functions from addresses to values:

$$\mathit{Heaps} = \mathit{Addresses} \xrightarrow{\text{fin}} \mathbb{Z}$$

A *partial function* is not necessarily defined for all arguments. If f is a partial function, then $\operatorname{dom} f$ denotes the set of arguments for which the function is defined. A *finite partial function* is a partial function whose domain is a finite set.

If h is a heap, then $\operatorname{dom} h$ is the set of *allocated addresses*.

Example 2. *The heap where only addresses 5 and 10 are allocated, and which assigns value 7 to address 5 and value 12 to address 10 is denoted as*

$$\{5 \mapsto 7, 10 \mapsto 12\}$$

We have:

$$\begin{gathered} \operatorname{dom}\{5 \mapsto 7, 10 \mapsto 12\} = \{5, 10\} \\ \{5 \mapsto 7, 10 \mapsto 12\}(5) = 7 \\ \{5 \mapsto 7, 10 \mapsto 12\}(10) = 12 \end{gathered}$$

We define the *update* $f[x := y]$ of a function f at argument x with value y as follows:

$$f[x := y](z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

We have $\mathrm{dom}\,(f[x := y]) = \mathrm{dom}\, f \cup \{x\}$.

We define the *removal* $f - x$ of argument x from partial function f as

$$(f - x)(z) = f(z) \qquad \text{if } z \neq x$$

We have $\mathrm{dom}\,(f - x) = \mathrm{dom}\, f \setminus \{x\}$.

Example 3. *Suppose* $f = \{1 \mapsto 10, 2 \mapsto 20\}$. *Then* $f[1 := 100] = \{1 \mapsto 100, 2 \mapsto 20\}$ *and* $f[3 := 30] = \{1 \mapsto 10, 2 \mapsto 20, 3 \mapsto 30\}$. *Furthermore,* $f - 1 = \{2 \mapsto 20\}$.

The states are the pairs of stores and heaps:

$$\mathit{States} = \mathit{Stores} \times \mathit{Heaps}$$

For example:

$$\mathit{States} = \{(\{\mathsf{i} \mapsto 3, \_ \mapsto 0\}, \{5 \mapsto 10\}), \ldots\}$$

The set of continuations *Conts* is defined inductively as follows:

- **done** is a continuation; it is a *done continuation*
- If κ is a continuation and c is a command, then $c; \kappa$ is a continuation; it is a *command continuation with command* c *and nested continuation* κ
- If κ is a continuation, x is a program variable name, and s is a store, then $\mathbf{ret}(s, x, \kappa)$ is a continuation; it is a *return continuation with store* s *and nested continuation* κ

An *inductive definition* consists of a set of *axioms* and a set of *inference rules*. For example, in the above inductive definition of set *Conts*, the first bullet point is an axiom and the other two are inference rules. An element is in an inductively defined set if its presence can be derived using a finite number of axiom and inference rule applications.

We interpret $c_1; c_2; \kappa$ as $c_1; (c_2; \kappa)$. That is, command continuation binds more tightly than sequential composition.

Example 4. *The following are continuations:*

$$\mathbf{done}$$
$$\mathsf{x} := 5; \mathbf{done}$$
$$\mathsf{x} := [\mathsf{p}]; [\mathsf{p}] := \mathsf{x} + 1; \mathbf{done}$$
$$\mathsf{result} := 3; \mathbf{ret}(\{\mathsf{i} \mapsto 5, \_ \mapsto 0\}, \mathsf{i}, [\mathsf{p}] := \mathsf{i}; \mathbf{done})$$

The configurations are the pairs of states and continuations, plus the special error configuration:

$$\mathit{Configs} = \mathit{States} \times \mathit{Conts} \cup \{\mathbf{error}\}$$

We enclose a non-error configuration with angle brackets:

$$\mathit{Configs} = \{\langle(\{\_ \mapsto 0\}, \emptyset), \mathbf{done}\rangle, \mathbf{error}, \ldots\}$$

1.2.2 Evaluation

The *evaluation* $[\![e]\!]_s$ of an arithmetic expression e in a store s is defined as follows:

$$[\![x]\!]_s = s(x) \qquad [\![n]\!]_s = n \qquad [\![e + e']\!]_s = [\![e]\!]_s + [\![e']\!]_s \qquad [\![e - e']\!]_s = [\![e]\!]_s - [\![e']\!]_s$$

In words: the evaluation of a variable expression equals the value assigned to the variable by the store; the evaluation of a literal expression equals the literal value; and the evaluation of a sum or difference expression equals the sum or difference, respectively, of the evaluations of the operand expressions.

Example 5. *Let* $s = \{\mathsf{a} \mapsto 5, \mathsf{b} \mapsto 7, \_ \mapsto 0\}$. *Then* $[\![\mathsf{a}]\!]_s = 5$ *and* $[\![\mathsf{a} + \mathsf{b}]\!]_s = 12$.

The evaluation $[\![b]\!]_s$ of a boolean expression b in a store s is defined analogously.

Example 6. *Let* s *be as above. We have*

$$[\![\mathsf{a} = \mathsf{b}]\!]_s = \mathbf{false}$$
$$[\![\mathsf{a} = \mathsf{b} \vee \mathsf{a} < \mathsf{b}]\!]_s = \mathbf{true}$$
$$[\![\mathsf{a} = \mathsf{b} \wedge \mathsf{a} < \mathsf{b}]\!]_s = \mathbf{false}$$

1.2.3 Execution

We define a *step relation* between configurations, denoted by the symbol $\rightsquigarrow$:

$$\rightsquigarrow \subseteq \mathit{Configs} \times \mathit{Configs}$$

The relation is defined using the rules in Figure 1.

Notice that rule Step-Call uses the set *RoutineDefs*. We define this set as

$$\mathit{RoutineDefs} = \{\mathit{def}_1, \ldots, \mathit{def}_n\}$$

if the program under consideration is $\mathit{def}_1 \ \ldots \ \mathit{def}_n \ c$.

In words, the rules say the following:

- Execution of a skip command does nothing.
- Execution of an assignment command $x := e$ updates the store at x with the value of e in the current store.
- Execution of a sequential composition command $c; c'$ first executes c and then executes c'.
- If boolean expression b evaluates to **true**, execution of an if command **if** b **then** c **else** c' executes the **then** branch c.
- If boolean expression b evaluates to **false**, execution of an if command **if** b **then** c **else** c' executes the **else** branch c'.
- If boolean expression b evaluates to **true**, execution of a loop command **while** b **do** c first executes c and then executes the same loop command again.
- If boolean expression b evaluates to **false**, execution of a loop command does nothing.
- Execution of an allocation command $x := \mathbf{cons}(e_1, \ldots, e_n)$ picks an arbitrary address ℓ such that ℓ through $\ell + n - 1$ are not allocated, then updates the heap at ℓ through $\ell + n - 1$ with the values of e_1 through e_n in the current store, and finally updates the store at x to ℓ.
- If the value of e is an allocated address, execution of a lookup command $x := [e]$ updates the store at x with the value of the memory cell at this address.
- If the value of e is an allocated address, execution of a mutation command $[e] := e'$ updates the heap at this address with the value of e'.
- If the value of e is an allocated address, execution of a de-allocation command **dispose** e removes the memory cell at address e from the heap.
- If the value of e is not an allocated address, execution of a lookup command $x := [e]$, a mutation command $[e] := e'$, and a de-allocation command **dispose** e *goes wrong*, i.e., results in an **error** configuration.

We define the *initial configuration* γ_0 for program $\mathit{def}_1 \ \ldots \ \mathit{def}_n \ c$ as $\langle(\{\_ \mapsto 0\}, \emptyset), c; \mathbf{done}\rangle$.

Example 7. *Consider the following program.*

$$\begin{array}{l}\mathsf{increment(p)} := \\ \quad \mathsf{tmp} := [\mathsf{p}];\ [\mathsf{p}] := \mathsf{tmp} + 1\end{array}$$

$$\mathsf{x} := \mathbf{cons}(5);\ \mathsf{y} := \mathsf{increment(x)};\ \mathbf{dispose}\ \mathsf{x}$$

Then we have

$$\gamma_0 = \langle(\{\_ \mapsto 0\}, \emptyset), \mathsf{x} := \mathbf{cons}(5);\ \mathsf{y} := \mathsf{increment(x)};\ \mathbf{dispose}\ \mathsf{x};\ \mathbf{done}\rangle$$

We define $\rightsquigarrow^n$, for arbitrary natural number n, as follows:

- $\rightsquigarrow^0 = \{(\gamma, \gamma) \mid \gamma \in \mathit{Configs}\}$

STEP-SKIP
$$\langle (s,h), \mathbf{skip}; \kappa \rangle \rightsquigarrow \langle (s,h), \kappa \rangle$$

STEP-ASSIGN
$$\langle (s,h), x := e; \kappa \rangle \rightsquigarrow \langle (s[x := [\![e]\!]_s], h), \kappa \rangle$$

STEP-SEQ
$$\langle (s,h), (c; c'); \kappa \rangle \rightsquigarrow \langle (s,h), c; (c'; \kappa) \rangle$$

STEP-IF-TRUE
$$\frac{[\![b]\!]_s = \mathbf{true}}{\langle (s,h), \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa \rangle \rightsquigarrow \langle (s,h), c; \kappa \rangle}$$

STEP-IF-FALSE
$$\frac{[\![b]\!]_s = \mathbf{false}}{\langle (s,h), \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'; \kappa \rangle \rightsquigarrow \langle (s,h), c'; \kappa \rangle}$$

STEP-WHILE-TRUE
$$\frac{[\![b]\!]_s = \mathbf{true}}{\langle (s,h), \mathbf{while}\ b\ \mathbf{do}\ c; \kappa \rangle \rightsquigarrow \langle (s,h), c; \mathbf{while}\ b\ \mathbf{do}\ c; \kappa \rangle}$$

STEP-WHILE-FALSE
$$\frac{[\![b]\!]_s = \mathbf{false}}{\langle (s,h), \mathbf{while}\ b\ \mathbf{do}\ c; \kappa \rangle \rightsquigarrow \langle (s,h), \kappa \rangle}$$

STEP-CONS
$$\frac{\ell, \ldots, \ell + n - 1 \in \mathit{Addresses} - \operatorname{dom} h}{\langle (s,h), x := \mathbf{cons}(e_1, \ldots, e_n); \kappa \rangle \rightsquigarrow \langle (s[x := \ell], h[\ell := [\![e_1]\!]_s, \ldots, \ell + n - 1 := [\![e_n]\!]_s]), \kappa \rangle}$$

STEP-READ-OK
$$\frac{[\![e]\!]_s \in \operatorname{dom} h}{\langle (s,h), x := [e]; \kappa \rangle \rightsquigarrow \langle (s[x := h([\![e]\!]_s)], h), \kappa \rangle}$$

STEP-READ-ERROR
$$\frac{[\![e]\!]_s \notin \operatorname{dom} h}{\langle (s,h), x := [e]; \kappa \rangle \rightsquigarrow \mathbf{error}}$$

STEP-WRITE-OK
$$\frac{[\![e]\!]_s \in \operatorname{dom} h}{\langle (s,h), [e] := e'; \kappa \rangle \rightsquigarrow \langle (s, h[[\![e]\!]_s := [\![e']\!]_s]), \kappa \rangle}$$

STEP-WRITE-ERROR
$$\frac{[\![e]\!]_s \notin \operatorname{dom} h}{\langle (s,h), [e] := e'; \kappa \rangle \rightsquigarrow \mathbf{error}}$$

STEP-DISPOSE-OK
$$\frac{[\![e]\!]_s \in \operatorname{dom} h}{\langle (s,h), \mathbf{dispose}\ e; \kappa \rangle \rightsquigarrow \langle (s, h - [\![e]\!]_s), \kappa \rangle}$$

STEP-DISPOSE-ERROR
$$\frac{[\![e]\!]_s \notin \operatorname{dom} h}{\langle (s,h), \mathbf{dispose}\ e; \kappa \rangle \rightsquigarrow \mathbf{error}}$$

STEP-CALL
$$\frac{r(x_1, \ldots, x_n) := c \in \mathit{RoutineDefs}}{\langle (s,h), x := r(e_1, \ldots, e_n); \kappa \rangle \rightsquigarrow \langle (\{x_1 \mapsto [\![e_1]\!]_s, \ldots, x_n \mapsto [\![e_n]\!]_s, \_ \mapsto 0\}, h), c; \mathbf{ret}(s, x, \kappa) \rangle}$$

STEP-RETURN
$$\frac{}{\langle (s,h), \mathbf{ret}(s', x, \kappa) \rangle \rightsquigarrow \langle (s'[x := s(\mathsf{result})], h), \kappa \rangle}$$

Figure 1: Step rules

- $\rightsquigarrow^{n+1} = \rightsquigarrow^{n} \rightsquigarrow$

We define $\rightsquigarrow^*$ as $\bigcup_{k \in \mathbb{N}} \rightsquigarrow^k$.

We say a configuration γ is *reachable* if $\gamma_0 \rightsquigarrow^* \gamma$. We say a program *goes wrong* if the error configuration is reachable.

Example 8. *Consider the program of Example 7. The program declares a routine* increment *which is called from the main command. One possible execution of this program is shown below.*

$\langle(\{\_ \mapsto 0\}, \emptyset), \mathsf{x} := \mathbf{cons}(5);\ \mathsf{y} := \mathsf{increment}(\mathsf{x});\ \mathbf{dispose}\ \mathsf{x};\ \mathbf{done}\rangle$
$\rightsquigarrow$ (Step-Cons)
$\langle(\{\mathsf{x} \mapsto 44, \_ \mapsto 0\}, \{44 \mapsto 5\}), \mathsf{y} := \mathsf{increment}(\mathsf{x});\ \mathbf{dispose}\ \mathsf{x};\ \mathbf{done}\rangle$
$\rightsquigarrow$ (Step-Call)
$\langle(\{\mathsf{p} \mapsto 44, \_ \mapsto 0\}, \{44 \mapsto 5\}), \mathsf{tmp} := [\mathsf{p}];\ [\mathsf{p}] := \mathsf{tmp} + 1;\ \mathbf{ret}(\{\mathsf{x} \mapsto 44, \_ \mapsto 0\}, \mathsf{y}, \mathbf{dispose}\ \mathsf{x};\ \mathbf{done})\rangle$
$\rightsquigarrow$ (Step-Read-OK)
$\langle(\{\mathsf{p} \mapsto 44, \mathsf{tmp} \mapsto 5, \_ \mapsto 0\}, \{44 \mapsto 5\}), [\mathsf{p}] := \mathsf{tmp} + 1;\ \mathbf{ret}(\{\mathsf{x} \mapsto 44, \_ \mapsto 0\}, \mathsf{y}, \mathbf{dispose}\ \mathsf{x};\ \mathbf{done})\rangle$
$\rightsquigarrow$ (Step-Write-OK)
$\langle(\{\mathsf{p} \mapsto 44, \mathsf{tmp} \mapsto 5, \_ \mapsto 0\}, \{44 \mapsto 6\}), \mathbf{ret}(\{\mathsf{x} \mapsto 44, \_ \mapsto 0\}, \mathsf{y}, \mathbf{dispose}\ \mathsf{x};\ \mathbf{done})\rangle$
$\rightsquigarrow$ (Step-Return)
$\langle(\{\mathsf{x} \mapsto 44, \mathsf{y} \mapsto 0, \_ \mapsto 0\}, \{44 \mapsto 6\}), \mathbf{dispose}\ \mathsf{x};\ \mathbf{done}\rangle$
$\rightsquigarrow$ (Step-Dispose-OK)
$\langle(\{\mathsf{x} \mapsto 44, \mathsf{y} \mapsto 0, \_ \mapsto 0\}, \emptyset), \mathbf{done}\rangle$

In this particular execution, the **cons** *command allocates memory at address 44. This is a valid choice, since 44 is not in the domain of the heap. However, note that other executions of this program may select a different address. In fact, for every address $\ell \in$ Addresses, there is an execution of the example program where the* **cons** *command allocates memory at address ℓ. It follows that this program has infinitely many executions.*

Exercise 1. *What is the value of the variable* i *at the end of the following program?*

$$\mathsf{p} := \mathbf{cons}(1, 2, 3);\ \mathsf{i} := [\mathsf{p} + 1];\ [\mathsf{p} + \mathsf{i}] := \mathsf{i};\ \mathsf{j} = [\mathsf{p} + i];\ \mathsf{i} = \mathsf{i} + \mathsf{j}$$

Exercise 2. *Write down the configurations in some execution of the following program.*

$$\begin{array}{l}\mathsf{swap}(\mathsf{x}, \mathsf{y}) := \\ \quad \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}\end{array}$$

$$\mathsf{i} := \mathbf{cons}(5, 8);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i} + 1);\ \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i} + 1$$

Exercise 3. *What happens if we change the main command in Exercise 2 as follows?*

$$\mathsf{i} := \mathbf{cons}(5);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i} + 1);\ \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i} + 1$$

Exercise 4. *What happens if we change the main command in Exercise 2 as follows?*

$$\mathsf{i} := \mathbf{cons}(5, 8);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i} + 1);\ \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}$$

Exercise 5. *What happens if we change the* swap *routine in Exercise 2 as follows?*

$$\begin{array}{l}\mathsf{swap}(\mathsf{x}, \mathsf{y}) := \\ \quad \mathsf{i} := 37\end{array}$$

2 Annotations and Abstracted Execution

The verification approach under consideration is a *modular* approach. This means that each routine is verified separately, using only the contracts of other routines, not their implementations. A routine contract consists of a precondition and a postcondition. A routine's precondition specifies the expectations the routine has about the store and the heap on entry to the routine. A routine's postcondition specifies the guarantees the routine offers about the store and the heap on exit from the routine. Preconditions and postconditions are expressed in the form of *assertions*.

Example 9. *Consider the routine* r*.*

$$
\begin{array}{l}
\mathsf{r}(\mathsf{a}, \mathsf{b}) \\
\quad \textbf{requires } \mathsf{a} = \mathsf{b} \\
\quad \textbf{ensures } \mathsf{result} = 0 \\
:= \mathsf{result} := \mathsf{a} - \mathsf{b}
\end{array}
$$

The precondition of r *requires the value of* a *to equal the value of* b*. The postcondition of* r *ensures that the result of* r *equals* 0*, provided the precondition was satisfied on entry to the routine.*

In order to preserve information hiding, routine contracts must be able to express constraints over the heap in an abstract way. For this purpose, the approach uses *predicates*. A predicate is simply a named, parameterized assertion. Specifically, a *predicate definition* is of the form

$$\textbf{predicate } p(x_1, \ldots, x_n) := a$$

where p is a predicate name, $x_1, \ldots, x_n$ are program variable names, and a is an assertion. $x_1, \ldots, x_n$ are called the parameters of p and a is called the body of p. n is called the *arity* of p. We say p is an n-ary predicate.

Using predicates, an abstract heap can be derived from a concrete heap. An abstract heap contains not just memory cells, but *predicate instances* as well. If p is an n-ary predicate, then $p(v_1, \ldots, v_n)$ is a predicate instance, where $v_1, \ldots, v_n \in \mathbb{Z}$. An abstract heap is obtained from a concrete heap by *closing* predicate instances. Closing a predicate instance replaces the part of the heap described by the predicate instance's body with the predicate instance itself. Conversely, *opening* a predicate instance replaces the predicate instance with a heap fragment that matches its body. We say that a given abstract heap *abstracts* a given concrete heap if the concrete heap can be obtained from the abstract heap through a finite number of *open* operations.

Example 10. *Consider the predicate* equal *and the routine* r′*.*

$$
\begin{array}{l}
\textbf{predicate } \mathsf{equal}(\mathsf{x}, \mathsf{y}) := \mathsf{x} = \mathsf{y} \\
\\
\mathsf{r}'(\mathsf{a}, \mathsf{b}) \\
\quad \textbf{requires } \mathsf{equal}(\mathsf{a}, \mathsf{b}) \\
\quad \textbf{ensures } \mathsf{equal}(\mathsf{result}, 0) \\
:= \textbf{open } \mathsf{equal}(\mathsf{a}, \mathsf{b});\ \textbf{result} := \mathsf{a} - \mathsf{b};\ \textbf{close } \mathsf{equal}(\mathsf{result}, 0)
\end{array}
$$

The predicate equal(x, y) *is a shorthand for the assertion* x = y*. The contract of* r′ *uses this shorthand both in its pre- and postcondition. The routine's body starts with an open statement to convert the predicate instance* equal(a, b) *to its definition* a = b*. Vice versa, the body ends with a close statement to convert the assertion* result = 0 *to the predicate instance* equal(result, 0)*.*

In this section, we introduce the syntax of assertions and other program annotations, and their meaning, by defining the notions of *consumption* and *production* of assertions. We then use these concepts to define *abstracted execution* of annotated commands, routines, and programs. Finally, we show a relationship between abstracted execution and concrete execution: if abstracted execution of a program succeeds, then the program does not go wrong. We call this property the *soundness* of abstracted execution.

2.1 Syntax

The syntax of annotated programs is given by the following grammar:

x	::=	*one of* $\mathsf{i}, \mathsf{counter}, \mathsf{result}, \ldots$	program variable name
n	::=	*one of* $0, 1, -1, 2, -2, 3, -3, \ldots$	integer literal
p	::=	*one of* $\mathsf{cell}, \mathsf{account}, \ldots$	predicate name
r	::=	*one of* $\mathsf{increment}, \mathsf{swap}, \ldots$	routine name
e	::=		arithmetic expression
		x	program variable expression
		$\mid n$	literal expression
		$\mid (e + e)$	addition expression
		$\mid (e - e)$	subtraction expression
b	::=		boolean expression
		$e = e$	equality expression
		$\mid e < e$	inequality expression
		$\mid (b \wedge b)$	conjunction expression
		$\mid (b \vee b)$	disjunction expression
		$\mid \neg b$	negation expression
π	::=		pattern
		e	expression pattern
		$\mid ?x$	variable pattern
a	::=		assertion
		$e \mapsto ?x$	points-to assertion
		$\mid p(\pi, \ldots, \pi)$	predicate assertion
		$\mid b$	boolean assertion
		$\mid a * a$	separate conjunction assertion
		$\mid$ **if** b **then** a **else** a	conditional assertion
c	::=		command
		$x := e$	assignment command
		$\mid (c; c)$	sequential composition command
		$\mid$ **if** b **then** c **else** c	conditional command
		$\mid$ **while** b **inv** a **do** c	loop command
		$\mid x :=$ **cons**$(e, \ldots, e)$	allocation command
		$\mid x := [e]$	lookup command
		$\mid [e] := e$	mutation command
		$\mid$ **dispose** e	de-allocation command
		$\mid$ **skip**	skip command
		$\mid x := r(e, \ldots, e)$	routine call command
		$\mid$ **open** $p(e, \ldots, e)$	open command
		$\mid$ **close** $p(e, \ldots, e)$	close command
pdef	::=	**predicate** $p(x, \ldots, x) := a$	predicate definition
def	::=	$r(x, \ldots, x)$ **requires** a **ensures** a $:= c$	routine definition
program	::=	*pdef* ... *pdef def* ... *def c*	program

The open and close commands are the *ghost commands*.

Given an annotated program, we can obtain an un-annotated program by removing all predicate definitions, routine contracts, loop invariants, and ghost commands. We call this program the *erasure* of the annotated program.

2.2 Meaning of Assertions

Assertions are interpreted with respect to an *abstract heap*. We define the set of *abstract heap elements* *AbsHeapElems* as the union of the *points-to elements* and the *predicate instances*:

$$\mathit{AbsHeapElems} = \{\ell \mapsto v \mid \ell \in \mathit{Addresses}, v \in \mathbb{Z}\} \cup \{p(v_1, \ldots, v_n) \mid v_1, \ldots, v_n \in \mathbb{Z}\}$$

Example 11. *The following are abstract heap elements:*

$$1 \mapsto 5 \qquad \mathsf{cell}(3, 7) \qquad 10 \mapsto 8 \qquad \mathsf{interval}(9, 11)$$

We define the set of *abstract heaps* *AbsHeaps* as the set of finite multisets of abstract heap elements:

$$AbsHeaps = AbsHeapElems \xrightarrow{\text{fin}} \mathbb{N}_0$$

A *multiset* is like a set, except that elements may occur more than once. Formally, a multiset of elements of some set A is a total function from A to the natural numbers. We say a *finite multiset* of elements of A is a finite partial function from A to the positive natural numbers.

We will denote a multiset containing elements $a_1, \ldots, a_n$ (not necessarily distinct) as $\{\!\{a_1, \ldots, a_n\}\!\}$. We will use operators $+$ and $-$ to denote sum and difference of multisets.

Example 12. *Consider some set A and distinct elements $a_1, a_2, a_3 \in A$. Then the multiset $\{\!\{a_1, a_1, a_2\}\!\}$ can also be written as the finite partial function $\{a_1 \mapsto 2, a_2 \mapsto 1\}$, since element a_1 occurs twice and element a_2 occurs once. Furthermore, we have*

$$\{\!\{a_1, a_2\}\!\} + \{\!\{a_1, a_3\}\!\} = \{\!\{a_1, a_1, a_2, a_3\}\!\}$$
$$\{\!\{a_1, a_1, a_2, a_3\}\!\} - \{\!\{a_1, a_3\}\!\} = \{\!\{a_1, a_2\}\!\}$$

We say an abstract heap is *well-formed* if it does not contain two points-to elements whose addresses are equal:

$$\mathsf{wf}\ H \triangleq (\forall \ell_1, v_1, \ell_2, v_2, H' \bullet H = \{\!\{\ell_1 \mapsto v_1, \ell_2 \mapsto v_2\}\!\} + H' \Rightarrow \ell_1 \neq \ell_2)$$

Example 13. *The abstract heap $\{\!\{1 \mapsto 5, 2 \mapsto 5\}\!\}$ is well-formed, while $\{\!\{1 \mapsto 5, 1 \mapsto 5\}\!\}$ and $\{\!\{1 \mapsto 5, 1 \mapsto 6\}\!\}$ are not.*

We define the set of *abstract states* *AbsStates* as the pairs of stores and abstract heaps:

$$AbsStates = Stores \times AbsHeaps$$

2.2.1 Consumption of Assertions

We define the operation of *consuming* an assertion. Consuming an assertion in a given abstract state checks that there exists a fragment of the heap that matches the assertion, and then removes this fragment from the heap.

Consumption performs *pattern matching*. A points-to assertion and a predicate assertion may contain *variable patterns*, of the form $?x$. A variable pattern matches any value, and binds the value to the variable (called the *pattern variable*). Matching occurs from left to right; pattern variable bindings are visible in the remainder of the assertion.

Below we define function *consume*. $consume(s, H, a, Q)$ denotes that consumption of assertion a in abstract state (s, H) succeeds, and the post-state satisfies consumption postcondition Q, which is a function from abstract states to booleans.

For consuming points-to assertions and predicate assertions, function *consume* uses two auxiliary functions, *match_pattern* and *match_patterns*. $match\_pattern(s, v, \pi)$ attempts to match value v against pattern π in store s. If the match succeeds, the function returns a singleton set containing the resulting store, i.e. the store s after performing any pattern variable bindings; otherwise, it returns an empty set. Similarly, function $match\_patterns(s, \overline{v}, \overline{\pi})$ attempts to match the list of values $\overline{v}$ against the list of patterns $\overline{\pi}$. If the match succeeds, the function returns a singleton set containing the resulting store; otherwise, it returns an empty set.

A *list* of elements of a set A is either the *empty list* ϵ, or the *constructed list* $a::\overline{a}$, where $a \in A$ and $\overline{a}$ is a list of elements of A. a is called the *head* and $\overline{a}$ is called the *tail* of $a::\overline{a}$. In general, if a ranges over A, then $\overline{a}$ ranges over lists of elements of A.

Example 14. *The following are lists of integers: $\epsilon, 1::\epsilon, 1::2::\epsilon, 10::9::8::\epsilon$.*

We use the notation $\textbf{lambda}\ x_1, \ldots, x_n \bullet E$ to denote the function of n arguments that maps arguments $x_1, \ldots, x_n$ to the value given by E. In other words, we have

$$f = \textbf{lambda}\ x_1, \ldots, x_n \bullet E \Leftrightarrow \forall x_1, \ldots, x_n \bullet f(x_1, \ldots, x_n) = E$$

The lambda notation makes it possible to write functions without having to give them a name and without having to define them in a separate place.

Example 15. *Consider the function f defined by $f(x) = x + 1$. Then we have $f =$* **lambda** $x \bullet x + 1$. *Therefore, we have* $(\textbf{lambda}\ x \bullet x + 1)(5) = 6, (\textbf{lambda}\ x \bullet x + 1)(6) = 7$, *etc.*

In words, function *consume* operates as follows:

- Consumption of points-to assertions and predicate assertions first compiles a set of matching elements, and then checks that there is a match such that the postcondition holds in the updated abstract state, where the store has been updated with the appropriate pattern variable bindings and the matched element has been removed from the heap.
- Consumption of a pure assertion checks that the boolean expression is true in the current store and that the postcondition holds in the current state.
- Consumption of a separate conjunction $a_1 * a_2$ first consumes a_1 in the current state and then consumes a_2 in the resulting state.
- Consumption of a conditional assertion consumes the **then** branch if the condition is true; otherwise, it consumes the **else** branch.

$$
\begin{array}{l}
\mathit{match\_pattern}(s, v, e) \equiv \\
\quad \textbf{if } [\![e]\!]_s = v \textbf{ then } \{s\} \textbf{ else } \emptyset \\
\mathit{match\_pattern}(s, v, ?x) \equiv \\
\quad \{s[x := v]\} \\
\\
\mathit{match\_patterns}(s, \epsilon, \epsilon) = \{s\} \\
\mathit{match\_patterns}(s, v{::}\overline{v}, \pi{::}\overline{\pi}) \equiv \\
\quad \{s'' \mid s' \in \mathit{match\_pattern}(s, v, \pi), s'' \in \mathit{match\_patterns}(s', \overline{v}, \overline{\pi})\} \\
\\
\mathit{consume}(s, H, e \mapsto ?x, Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \{\ell \mapsto v \mid \ell = [\![e]\!]_s \wedge \ell \mapsto v \in H\} \textbf{ in} \\
\quad \exists \ell \mapsto v \in \mathit{matches} \bullet Q(s[x := v], H - \{\!\{\ell \mapsto v\}\!\}) \\
\mathit{consume}(s, H, p(\pi_1, \ldots, \pi_n), Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \\
\qquad \{(s', (v_1, \ldots, v_n)) \mid \\
\qquad\quad p(v_1, \ldots, v_n) \in H, s' \in \mathit{match\_patterns}(s, v_1{::}\cdots{::}v_n{::}\epsilon, \pi_1{::}\cdots{::}\pi_n{::}\epsilon)\} \textbf{ in} \\
\quad \exists (s', (v_1, \ldots, v_n)) \in \mathit{matches} \bullet Q(s', H - \{\!\{p(v_1, \ldots, v_n)\}\!\}) \\
\mathit{consume}(s, H, b, Q) \equiv \\
\quad ([\![b]\!]_s = \textbf{true}) \wedge Q(s, H) \\
\mathit{consume}(s, H, a_1 * a_2, Q) \equiv \\
\quad \mathit{consume}(s, H, a_1, (\textbf{lambda } s', H' \bullet \mathit{consume}(s', H', a_2, Q))) \\
\mathit{consume}(s, H, \textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2, Q) \equiv \\
\quad \textbf{if } [\![b]\!]_s \textbf{ then } \mathit{consume}(s, H, a_1, Q) \textbf{ else } \mathit{consume}(s, H, a_2, Q)
\end{array}
$$

Example 16. *Consider the store $s = \{\mathsf{p} \mapsto 5, \_ \mapsto 0\}$. Then we have:*

$$\mathit{match\_pattern}(s, 10, \mathsf{p}) = \emptyset \qquad \mathit{match\_pattern}(s, 5, \mathsf{p}) = \{s\} = \{\{\mathsf{p} \mapsto 5, \_ \mapsto 0\}\}$$

$$\mathit{match\_pattern}(s, 7, ?\mathsf{x}) = \{s[\mathsf{x} := 7]\} = \{\{\mathsf{p} \mapsto 5, \mathsf{x} \mapsto 7, \_ \mapsto 0\}\}$$

$$\mathit{match\_pattern}(s, 7, ?\mathsf{p}) = \{s[\mathsf{p} := 7]\} = \{\{\mathsf{p} \mapsto 7, \_ \mapsto 0\}\} \qquad \mathit{match\_patterns}(s, 10{::}\epsilon, \mathsf{p}{::}\epsilon) = \emptyset$$

$$\mathit{match\_patterns}(s, 5{::}7{::}\epsilon, \mathsf{p}{::}?\mathsf{x}{::}\epsilon) = \{s[\mathsf{x} := 7]\} \qquad \mathit{match\_patterns}(s, 5{::}7{::}\epsilon, ?\mathsf{p}{::}(\mathsf{p} + 2){::}\epsilon) = \{s[\mathsf{p} := 5]\}$$

Example 17. *Assume the following predicate definition:*

$$\textbf{predicate } \mathsf{cell}(\mathsf{c}, \mathsf{v}) := \ldots$$

Suppose the store is $\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}$ *and the abstract heap equals* $\{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}$*. What is the result of consuming the assertion* $\mathsf{i} \mapsto ?\mathsf{j} * \mathsf{j} = 5$*?*

$$
\begin{array}{l}
\mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{j} * \mathsf{j} = 5, Q) \\
\Leftrightarrow\ \mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{j}, \textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{j} = 5, Q)) \\
\Leftrightarrow\ (\textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{j} = 5, Q))(\{\mathsf{i} \mapsto 7, \mathsf{j} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}) \\
\Leftrightarrow\ \mathit{consume}(\{\mathsf{i} \mapsto 7, \mathsf{j} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}, \mathsf{j} = 5, Q) \\
\Leftrightarrow\ Q(\{\mathsf{i} \mapsto 7, \mathsf{j} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\})
\end{array}
$$

The heap element $7 \mapsto 5$ *is consumed by the assertion, and the store contains an additional variable binding* $\mathsf{j} \mapsto 5$*. What's the result of consuming the assertion* $\mathsf{i} \mapsto ?\mathsf{x} * \mathsf{i} \mapsto ?\mathsf{y}$*?*

$$
\begin{array}{l}
\mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{x} * \mathsf{i} \mapsto ?\mathsf{y}, Q) \\
\Leftrightarrow \mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{x}, \textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{i} \mapsto ?\mathsf{y}, Q)) \\
\Leftrightarrow (\textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{i} \mapsto ?\mathsf{y}, Q))(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}) \\
\Leftrightarrow \mathit{consume}(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{y}, Q) \\
\Leftrightarrow \exists \ell \mapsto v \in \emptyset \bullet Q(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \mathsf{y} \mapsto v, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\} - \{\!|\ell \mapsto v|\!\}) \\
\Leftrightarrow \textbf{false}
\end{array}
$$

One can think of a points-to element $\ell \mapsto v$ *as a ticket that gives permission to access the memory location* ℓ*. It is not possible to duplicate the ticket. What is the result of consuming the assertion* $\mathsf{i} \mapsto ?\mathsf{x} * \mathsf{cell}(\mathsf{x}, ?\mathsf{v})$*?*

$$
\begin{array}{l}
\mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{x} * \mathsf{cell}(\mathsf{x}, ?\mathsf{v}), Q) \\
\Leftrightarrow \mathit{consume}(\{\mathsf{i} \mapsto 7, \_ \mapsto 0\}, \{\!|7 \mapsto 5, \mathsf{cell}(5,1)|\!\}, \mathsf{i} \mapsto ?\mathsf{x}, \textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{cell}(\mathsf{x}, ?\mathsf{v}), Q)) \\
\Leftrightarrow (\textbf{lambda}\ s', H' \bullet \mathit{consume}(s', H', \mathsf{cell}(\mathsf{x}, ?\mathsf{v}), Q))(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}) \\
\Leftrightarrow \mathit{consume}(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \_ \mapsto 0\}, \{\!|\mathsf{cell}(5,1)|\!\}, \mathsf{cell}(\mathsf{x}, ?\mathsf{v}), Q) \\
\Leftrightarrow Q(\{\mathsf{i} \mapsto 7, \mathsf{x} \mapsto 5, \mathsf{v} \mapsto 1, \_ \mapsto 0\}, \{\!|\,|\!\})
\end{array}
$$

Exercise 6. *Suppose the store is* $\{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 11, \_ \mapsto 0\}$ *and the abstract heap equals* $\{\!|\mathsf{cell}(11,0), 1 \mapsto 2, \mathsf{cell}(3,4)|\!\}$*. Try consuming the following assertions:*

1. $\mathsf{x} = 1$
2. $\mathsf{x} = 2$
3. $\mathsf{y} \mapsto ?\mathsf{k} * \mathsf{cell}(\mathsf{k}, \mathsf{x} - 1)$
4. $\mathsf{cell}(11, ?\mathsf{v}) * (\mathsf{v} + \mathsf{x}) \mapsto ?\mathsf{w} * \mathsf{w} = 2$

We have the following properties.

Lemma 1 (Consumption Postcondition Weakening)**.** *If consumption of an assertion with respect to a given postcondition succeeds, then consumption with respect to a weaker postcondition succeeds also.*

$$\forall s, H, a, Q, Q' \bullet \mathit{consume}(s, H, a, Q) \Rightarrow (\forall s', H' \bullet Q(s', H') \Rightarrow Q'(s', H')) \Rightarrow \mathit{consume}(s, H, a, Q')$$

Lemma 2 (Consumption Framing)**.** *If consumption of an assertion succeeds, then it also succeeds if elements are added to the abstract heap, with respect to a postcondition that states that the post-state abstract heap still contains these added elements, and after removing these added elements the original postcondition holds.*

$$\forall s, H, a, Q, H_{\mathrm{F}} \bullet \mathit{consume}(s, H, a, Q) \Rightarrow \mathit{consume}(s, H + H_{\mathrm{F}}, a, (\textbf{lambda}\ s', H' \bullet H_{\mathrm{F}} \leq H' \wedge Q(s', H' - H_{\mathrm{F}})))$$

We say an abstract state *satisfies* an assertion if consumption of the assertion in the abstract state succeeds and the resulting abstract heap is empty.

$$s, H \vDash a \Leftrightarrow \mathit{consume}(s, H, a, (\textbf{lambda}\ s', H' \bullet H' = \emptyset))$$

We have the following properties.

Lemma 3 (Consumption Soundness Helper)**.** *If consumption of an assertion a succeeds in a given abstract state, then the abstract heap can be split up into a fragment H_A such that consumption of a in H_A succeeds and leads to an empty heap and a store s'', and a fragment H'' that together with s'' satisfies the postcondition.*

$$\forall s, H, a, Q \bullet \mathit{consume}(s, H, a, Q) \Rightarrow \\ \exists H_A, s'', H'' \bullet \mathit{consume}(s, H_\mathrm{A}, a, (\textbf{lambda } s', H' \bullet s' = s'' \wedge H' = \emptyset)) \wedge Q(s'', H'') \wedge H = H_\mathrm{A} + H''$$

Proof. By induction on the structure of assertion a. □

Lemma 4 (Consumption Soundness)**.** *If consumption of an assertion succeeds in a given abstract state, then the abstract heap can be split into a fragment that satisfies the assertion and a fragment that satisfies the postcondition.*

$$\forall s, H, a, Q \bullet \mathit{consume}(s, H, a, Q) \Rightarrow \exists H_\mathrm{A}, s', H' \bullet s, H_\mathrm{A} \vDash a \wedge Q(s', H') \wedge H = H_\mathrm{A} + H'$$

Proof. Follows from Lemma 3. □

2.2.2 Production of Assertions

The operation of *producing* an assertion is the inverse of consuming it: producing an assertion in a given abstract state extends the heap with an arbitrary heap fragment that satisfies the assertion.

Specifically, function $\mathit{produce}(s, H, a, Q)$ denotes that postcondition Q holds in all states obtained by extending abstract state (s, H) with some heap fragment that satisfies assertion a.

Auxiliary function $\mathit{produce\_pattern}(s, \pi, Q)$ denotes that postcondition Q holds for each pair (s', v) such that value v matches pattern π in store s and s' is s updated with the appropriate pattern variable bindings. Similarly, auxiliary function $\mathit{produce\_patterns}(s, \overline{\pi}, Q)$ denotes that postcondition Q holds for each pair $(s', \overline{v})$ such that list of values $\overline{v}$ matches list of patterns $\overline{\pi}$ in store s and s' is s updated with the appropriate pattern variable bindings.

$$\begin{array}{l}
\mathit{produce_pattern}(s, e, Q) \equiv Q(s, [\![e]\!]_s) \\
\mathit{produce_pattern}(s, ?x, Q) \equiv (\forall v \in \mathbb{Z} \bullet Q(s[x := v], v))
\end{array}$$

$$\begin{array}{l}
\mathit{produce_patterns}(s, \epsilon, Q) \equiv Q(s, \epsilon) \\
\mathit{produce_patterns}(s, \pi{::}\overline{\pi}, Q) \equiv \\
\quad \mathit{produce_pattern}(s, \pi, (\textbf{lambda } s', v \bullet \mathit{produce_patterns}(s', \overline{\pi}, (\textbf{lambda } s'', \overline{v} \bullet Q(s'', v{::}\overline{v})))))
\end{array}$$

$$\begin{array}{l}
\mathit{produce}(s, H, e \mapsto ?x, Q) \equiv \\
\quad \forall v \bullet \mathsf{wf}\,(H + \{\!\{[\![e]\!]_s \mapsto v\}\!\}) \Rightarrow Q(s[x := v], H + \{\!\{[\![e]\!]_s \mapsto v\}\!\}) \\
\mathit{produce}(s, H, p(\pi_1, \ldots, \pi_n), Q) \equiv \\
\quad \mathit{produce_patterns}(s, \pi_1{::}\cdots{::}\pi_n{::}\epsilon, (\textbf{lambda } s', v_1{::}\cdots{::}v_n{::}\epsilon \bullet Q(s', H + \{\!\{p(v_1, \ldots, v_n)\}\!\}))) \\
\mathit{produce}(s, H, b, Q) \equiv \\
\quad [\![b]\!]_s = \textbf{true} \Rightarrow Q(s, H) \\
\mathit{produce}(s, H, a_1 * a_2, Q) \equiv \\
\quad \mathit{produce}(s, H, a_1, (\textbf{lambda } s', H' \bullet \mathit{produce}(s', H', a_2, Q)) \\
\mathit{produce}(s, H, \textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2, Q) \equiv \\
\quad \textbf{if } [\![b]\!]_s \textbf{ then } \mathit{produce}(s, H, a_1, Q) \textbf{ else } \mathit{produce}(s, H, a_2, Q)
\end{array}$$

Example 18. *Suppose $s = \{\mathsf{p} \mapsto 5, \_ \mapsto 0\}$. Then we have:*

$$\begin{array}{l}
\mathit{produce_pattern}(s, 0, Q) \Leftrightarrow Q(s, 0) \\
\mathit{produce_pattern}(s, \mathsf{p} + 4, Q) \Leftrightarrow Q(s, 9) \\
\mathit{produce_pattern}(s, ?\mathsf{q}, Q) \Leftrightarrow \forall v \in \mathbb{Z} \bullet Q(\{\mathsf{p} \mapsto 5, \mathsf{q} \mapsto v, _ \mapsto 0\}, v) \\
\mathit{produce_patterns}(s, \mathsf{p}{::}?\mathsf{p}{::}(\mathsf{p} + 4){::}\epsilon, Q) \Leftrightarrow \forall v \bullet Q(\{\mathsf{p} \mapsto v, _ \mapsto 0\}, 5{::}v{::}(v + 4){::}\epsilon)
\end{array}$$

Example 19. *Suppose $s = \{\mathsf{p} \mapsto 5, \_ \mapsto 0\}$. Elaborate the production of assertion $\mathsf{p} \mapsto ?\mathsf{x} * \mathsf{Cell}(\mathsf{x}, 9)$.*

$$\begin{array}{ll}
& \mathit{produce}(s, \emptyset, \mathsf{p} \mapsto ?\mathsf{x} * \mathsf{Cell}(\mathsf{x}, 9), Q) \\
\Leftrightarrow & \forall v \in \mathbb{Z} \bullet \mathit{produce}(\{\mathsf{p} \mapsto 5, \mathsf{x} \mapsto v, _ \mapsto 0\}, \{\!\{5 \mapsto v\}\!\}, \mathsf{Cell}(\mathsf{x}, 9), Q) \\
\Leftrightarrow & \forall v \in \mathbb{Z} \bullet Q(\{\mathsf{p} \mapsto 5, \mathsf{x} \mapsto v, _ \mapsto 0\}, \{\!\{5 \mapsto v, \mathsf{Cell}(v, 9)\}\!\})
\end{array}$$

The postcondition Q must hold for all heaps that satisfy the assertion.

Elaborate the production of assertion $\mathsf{p} = 6 * \mathsf{Cell}(\mathsf{p}, 10)$.

$$\begin{array}{l} \mathit{produce}(s, \emptyset, \mathsf{p} = 6 * \mathsf{Cell}(\mathsf{p}, 10), Q) \\ \Leftrightarrow (5 = 6 \Rightarrow \mathit{produce}(s, \emptyset, \mathsf{Cell}(\mathsf{p}, 10), Q)) \\ \Leftrightarrow \mathbf{true} \end{array}$$

If a boolean assertion is false, production succeeds trivially.

Exercise 7. *Suppose the store is* $\{\mathsf{x} \mapsto 5, \_ \mapsto 0\}$. *Elaborate the production of the following assertions.*

- $\mathsf{cell}(?\mathsf{x}, ?\mathsf{x}) * \mathsf{x} \mapsto ?\mathsf{x}$
- $\mathsf{x} < 10 * \mathsf{x} + 5 \mapsto ?\mathsf{y}$
- $\mathsf{cell}(?\mathsf{x}, ?\mathsf{y}) * \mathsf{x} \mapsto ?\mathsf{xv} * \mathsf{y} \mapsto ?\mathsf{yv}$
- $\mathsf{x} \mapsto ?\mathsf{v} * \mathbf{if}\ \neg(\mathsf{v} = 0)\ \mathbf{then}\ \mathsf{v} \mapsto ?\mathsf{w}\ \mathbf{else}\ \mathsf{v} = \mathsf{v}$

We have the following properties.

Lemma 5 (Production Postcondition Weakening)**.** *If production of an assertion succeeds, then it also succeeds with a weaker postcondition.*

$$\forall s, H, a, Q, Q' \bullet \mathit{produce}(s, H, a, Q) \Rightarrow (\forall s', H' \bullet Q(s', H') \Rightarrow Q'(s', H')) \Rightarrow \mathit{produce}(s, H, a, Q')$$

Lemma 6 (Production Heap Irrelevance)**.** *Producing an assertion in a given abstract state is equivalent to producing the same assertion in an empty heap and adding the initial heap to the post-state heap.*

$$\mathit{produce}(s, H, a, Q) \Leftrightarrow \mathit{produce}(s, \emptyset, a, (\mathbf{lambda}\ s', H' \bullet \mathsf{wf}\ (H' + H) \Rightarrow Q(s', H' + H)))$$

Lemma 7 (Production Soundness Helper)**.** *If consumption succeeds, then in each post-state, if production of the same assertion succeeds with respect to some production postcondition, then the original heap satisfies the production postcondition.*

$$\begin{array}{l} \forall s, H, a, Q \bullet \mathsf{wf}\ H\ \ \Rightarrow \\ \quad \mathit{consume}(s, H, a, Q) \Rightarrow \mathit{consume}(s, H, a, (\mathbf{lambda}\ s', H' \bullet \forall Q' \bullet \mathit{produce}(s, H', a, Q') \Rightarrow Q'(s', H))) \end{array}$$

Lemma 8 (Production Soundness)**.** *If a given abstract state satisfies an assertion, and production of this assertion succeeds in the empty heap, then the production postcondition holds for the given heap.*

$$\forall s, H, a, Q \bullet \mathsf{wf}\ H \Rightarrow s, H \vDash a \Rightarrow \mathit{produce}(s, \emptyset, a, Q) \Rightarrow \exists s' \bullet Q(s', H)$$

2.3 Heap abstraction

We say an abstract heap H *directly abstracts* an abstract heap H', denoted $H \triangleleft H'$, if H' can be obtained by removing a predicate instance $p(v_1, \ldots, v_n)$ from H and adding some set of abstract heap elements that satisfy the body of $p(v_1, \ldots, v_n)$.

$$\begin{array}{l} H \triangleleft H' \ \Leftrightarrow \\ \quad \exists H_0, p(v_1, \ldots, v_n), H_\mathrm{P} \bullet \\ \qquad \mathbf{predicate}\ p(x_1, \ldots, x_n) := a \in \mathit{PredDefs} \\ \qquad \wedge\ \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \_ \mapsto 0\}, H_\mathrm{P} \vDash a \\ \qquad \wedge\ H = H_0 + \{\!|p(v_1, \ldots, v_n)|\!\} \wedge H' = H_0 + H_\mathrm{P} \end{array}$$

We say an abstract heap H *abstracts* an abstract heap H', denoted $H \triangleleft^* H'$, if there exists a natural number n such that $H \triangleleft^n H'$.

We say an abstract heap H *abstracts* a concrete heap h, denoted $H \triangleleft^* h$, if H abstracts the abstract heap that contains each points-to element specified by h once.

$$H \triangleleft^* h \Leftrightarrow H \triangleleft^* \{\!|\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n|\!\} \quad \text{where } h = \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}$$

Example 20. *Assume the following predicate definitions:*

$$\textbf{predicate } \mathsf{p(i)} := \mathsf{i} \mapsto ?\mathsf{x} * 0 < \mathsf{x}$$
$$\textbf{predicate } \mathsf{equal(a, b)} := \mathsf{a} = \mathsf{b}$$

The abstract heap $H = \{\!|\mathsf{equal}(1,1), \mathsf{equal}(1,1), \mathsf{p}(44), 30 \mapsto 50, \mathsf{p}(45)|\!\}$ *can be reached from the abstract heap* $\{\!|44 \mapsto 5, 30 \mapsto 50, 45 \mapsto 10|\!\}$ *by closing* $\mathsf{p}(44)$ *and* $\mathsf{p}(45)$*, and closing* $\mathsf{equal}(1,1)$ *twice. H can also be reached from the abstract heap* $\{\!|44 \mapsto 145, 30 \mapsto 50, 45 \mapsto 78|\!\}$*, by closing the same predicate instances. It follows that H abstracts both concrete heaps. We further have:*

$$\{\!|\mathsf{equal}(1,1), \mathsf{equal}(1,1), \mathsf{p}(44), 30 \mapsto 50, \mathsf{p}(45)|\!\} \lhd^* \{\!|44 \mapsto 5, 30 \mapsto 50, 45 \mapsto 10|\!\}$$
$$\{\!|\mathsf{equal}(1,1), \mathsf{equal}(1,1), \mathsf{p}(44), 30 \mapsto 50, \mathsf{p}(45)|\!\} \lhd^* \{\!|44 \mapsto 145, 30 \mapsto 50, 45 \mapsto 78|\!\}$$
$$\{\!|\mathsf{p}(42), 42 \mapsto 100|\!\} \not\lhd^* \{\!|42 \mapsto 100|\!\}$$
$$\{\!|\mathsf{p}(42)|\!\} \not\lhd^* \{\!|42 \mapsto 100, 42 \mapsto 100|\!\}$$

Exercise 8. *Consider the following predicate definitions:*

$$\textbf{predicate } \mathsf{p(i)} := \mathsf{i} \mapsto ?\mathsf{x} * 0 < \mathsf{x}$$
$$\textbf{predicate } \mathsf{cell(x, y)} := \mathsf{x} \mapsto \mathsf{y}$$
$$\textbf{predicate } \mathsf{weird()} := \mathsf{cell}(?\mathsf{a}, ?\mathsf{b})$$

Which of the following statements are true?

1. $\{\!|\mathsf{p}(7)|\!\} \lhd^* \{\!|7 \mapsto 5|\!\}$
2. $\{\!|\mathsf{p}(7)|\!\} \lhd^* \{\!|7 \mapsto 6|\!\}$
3. $\{\!|\mathsf{p}(7)|\!\} \lhd^* \{\!|7 \mapsto -5|\!\}$
4. $\{\!|\mathsf{weird}(), 1 \mapsto 10|\!\} \lhd^* \{\!|1 \mapsto 10, 2 \mapsto 20|\!\}$
5. $\{\!|\mathsf{weird}(), 2 \mapsto 20|\!\} \lhd^* \{\!|1 \mapsto 10, 2 \mapsto 20|\!\}$

2.4 Abstracted execution

We now define *abstracted execution* of a command, of a routine, and of a program. Specifically, function $\mathit{valid}(s, H, c, Q)$, defined in Figure 2, denotes that abstracted execution of c in abstract state (s, H) succeeds, and each possible post-state satisfies postcondition Q. We also say that command c is *valid* with respect to abstract state (s, H) and postcondition Q.

Example 21. *Consider the program*

```
swap(x, y)
  requires x ↦ ?v * y ↦ ?w
  ensures x ↦ w * y ↦ v
  := x2 := [x]; y2 := [y]; [x] := y2; [y] := x2

i := cons(5, 8); m := swap(i, i + 1); dispose i; dispose i + 1
```

Suppose $s_0 = \{\_ \mapsto 0\}$*,* $H_0 = \emptyset$ *and* $Q_0 = \textbf{lambda } s', H' \bullet \textbf{true}$*. Write down the abstracted execution of*

$$
\begin{array}{l}
\mathit{havoc}(s, \{x_1, \ldots, x_n\}, Q) \equiv \forall v_1, \ldots, v_n \bullet Q(s[x_1 := v_1, \ldots, x_n := v_n]) \\
\\
\mathit{valid}(s, H, \textbf{skip}, Q) \equiv Q(s, H) \\
\mathit{valid}(s, H, x := e, Q) \equiv Q(s[x := [\![e]\!]_s], H) \\
\mathit{valid}(s, H, c_1; c_2, Q) \equiv \mathit{valid}(s, H, c_1, (\textbf{lambda } s', H' \bullet \mathit{valid}(s', H', c_2, Q))) \\
\mathit{valid}(s, H, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, Q) \equiv \\
\quad \textbf{if } [\![b]\!]_s \textbf{ then } \mathit{valid}(s, H, c_1, Q) \textbf{ else } \mathit{valid}(s, H, c_2, Q) \\
\mathit{valid}(s, H, \textbf{while } b \textbf{ inv } a \textbf{ do } c, Q) \equiv \\
\quad \mathit{consume}(s, H, a, (\textbf{lambda } s_1, H_1 \bullet \\
\qquad \mathit{havoc}(s, \mathit{modifies}(c), (\textbf{lambda } s_2 \bullet \\
\qquad\quad \mathit{produce}(s_2, \emptyset, a, (\textbf{lambda } s_3, H_3 \bullet \\
\qquad\qquad \textbf{if } [\![b]\!]_{s_2} \textbf{ then} \\
\qquad\qquad\quad \mathit{valid}(s_2, H_3, c, (\textbf{lambda } s_4, H_4 \bullet \\
\qquad\qquad\qquad \mathit{consume}(s_4, H_4, a, (\textbf{lambda } s_5, H_5 \bullet H_5 = \emptyset)))) \\
\qquad\qquad \textbf{else } Q(s_2, H_1 + H_3))))))) \\
\mathit{valid}(s, H, x := \textbf{cons}(e_1, \ldots, e_n), Q) \equiv \\
\quad \forall \ell \in \mathit{Addresses} \bullet Q(s[x := \ell], H + \{\!\{\ell \mapsto [\![e_1]\!]_s, \ldots, \ell + n - 1 \mapsto [\![e_n]\!]_s\}\!\}) \\
\mathit{valid}(s, H, x := [e], Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \{v \mid [\![e]\!]_s \mapsto v \in H\} \textbf{ in} \\
\quad \exists v \in \mathit{matches} \bullet Q(s[x := v], H) \\
\mathit{valid}(s, H, [e] := e', Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \{v \mid [\![e]\!]_s \mapsto v \in H\} \textbf{ in} \\
\quad \exists v \in \mathit{matches} \bullet Q(s, H - \{\!\{[\![e]\!]_s \mapsto v\}\!\} + \{\!\{[\![e]\!]_s \mapsto [\![e']\!]_s\}\!\}) \\
\mathit{valid}(s, H, \textbf{dispose } e, Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \{v \mid [\![e]\!]_s \mapsto v \in H\} \textbf{ in} \\
\quad \exists v \in \mathit{matches} \bullet Q(s, H - \{\!\{[\![e]\!]_s \mapsto v\}\!\}) \\
\mathit{valid}(s, H, \textbf{open } p(e_1, \ldots, e_n), Q) \equiv \\
\quad \textbf{let predicate } p(x_1, \ldots, x_n) := a \in \mathit{PredicateDefs} \textbf{ in} \\
\quad \textbf{let } v_1 = [\![e_1]\!]_s, \ldots, v_n = [\![e_n]\!]_s \textbf{ in} \\
\quad \textbf{let } s' = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \_ \mapsto 0\} \textbf{ in} \\
\quad p(v_1, \ldots, v_n) \in H \wedge \mathit{produce}(s', H - \{\!\{p(v_1, \ldots, v_n)\}\!\}, a, (\textbf{lambda } s_2, H_2 \bullet Q(s, H_2))) \\
\mathit{valid}(s, H, \textbf{close } p(e_1, \ldots, e_n), Q) \equiv \\
\quad \textbf{let predicate } p(x_1, \ldots, x_n) := a \in \mathit{PredicateDefs} \textbf{ in} \\
\quad \textbf{let } v_1 = [\![e_1]\!]_s, \ldots, v_n = [\![e_n]\!]_s \textbf{ in} \\
\quad \mathit{consume}(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \_ \mapsto 0\}, H, a, (\textbf{lambda } s', H' \bullet Q(s, H' + \{\!\{p(v_1, \ldots, v_n)\}\!\}))) \\
\mathit{valid}(s, H, x := r(e_1, \ldots, e_n), Q) \equiv \\
\quad \textbf{let } r(x_1, \ldots, x_n) \textbf{ requires } a_{\mathrm{pre}} \textbf{ ensures } a_{\mathrm{post}} \in \mathit{RoutineSpecs} \textbf{ in} \\
\quad \textbf{let } s_1 = \{x_1 \mapsto [\![e_1]\!]_s, \ldots, x_n \mapsto [\![e_n]\!]_s, \_ \mapsto 0\} \textbf{ in} \\
\quad \mathit{consume}(s_1, H, a_{\mathrm{pre}}, (\textbf{lambda } s_2, H_2 \bullet \\
\qquad \forall v \bullet \mathit{produce}(s_2[\mathsf{result} := v], H_2, a_{\mathrm{post}}, (\textbf{lambda } s_3, H_3 \bullet Q(s[x := v], H_3))))))
\end{array}
$$

Figure 2: Abstracted execution

the program's main command:

$$
\begin{array}{ll}
& \mathit{valid}(s_0, H_0, \mathsf{i} := \mathbf{cons}(5,8);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i}+1);\ \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \mathit{valid}(\{\mathsf{i} \mapsto \ell, \_ \mapsto 0\}, \{\!|\ell \mapsto 5, \ell+1 \mapsto 8|\!\}, \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i}+1);\ \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \mathit{consume}(\{\mathsf{x} \mapsto \ell, \mathsf{y} \mapsto \ell+1, \_ \mapsto 0\}, \{\!|\ell \mapsto 5, \ell+1 \mapsto 8|\!\}, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \mathbf{lambda}\ s_1, H_1 \bullet \\
& \qquad \forall v \in \mathbb{Z} \bullet \mathit{produce}(s_1[\mathsf{result} := v], H_1, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \mathbf{lambda}\ s_2, H_2 \bullet \\
& \qquad\quad \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, H_2, \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \mathit{consume}(\{\mathsf{x} \mapsto \ell, \mathsf{y} \mapsto \ell+1, \mathsf{v} \mapsto 5, \_ \mapsto 0\}, \{\!|\ell+1 \mapsto 8|\!\}, \mathsf{y} \mapsto ?\mathsf{w}, \mathbf{lambda}\ s_1, H_1 \bullet \\
& \qquad \forall v \in \mathbb{Z} \bullet \mathit{produce}(s_1[\mathsf{result} := v], H_1, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \mathbf{lambda}\ s_2, H_2 \bullet \\
& \qquad\quad \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, H_2, \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \forall v \in \mathbb{Z} \bullet \mathit{produce}(\{\mathsf{x} \mapsto \ell, \mathsf{y} \mapsto \ell+1, \mathsf{v} \mapsto 5, \mathsf{w} \mapsto 8, \mathsf{result} \mapsto v, \_ \mapsto 0\}, \emptyset, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad \mathbf{lambda}\ s_2, H_2 \bullet \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, H_2, \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \forall v \in \mathbb{Z} \bullet \mathit{produce}(\{\mathsf{x} \mapsto \ell, \mathsf{y} \mapsto \ell+1, \mathsf{v} \mapsto 5, \mathsf{w} \mapsto 8, \mathsf{result} \mapsto v, \_ \mapsto 0\}, \{\!|\ell \mapsto 8|\!\}, \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad \mathbf{lambda}\ s_2, H_2 \bullet \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, H_2, \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \forall v \in \mathbb{Z} \bullet \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \{\!|\ell \mapsto 8, \ell+1 \mapsto 5|\!\}, \mathbf{dispose}\ \mathsf{i};\ \mathbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses} \bullet \\
& \quad \forall v \in \mathbb{Z} \bullet \mathit{valid}(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \{\!|\ell+1 \mapsto 5|\!\}, \mathbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \forall \ell \in \mathit{Addresses}, v \in \mathbb{Z} \bullet Q_0(\{\mathsf{i} \mapsto \ell, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \emptyset) \\
\Leftrightarrow & \mathbf{true}
\end{array}
$$

We have the following properties.

Lemma 9 (Command Validity Postcondition Weakening)**.** *If a command is valid with respect to a given postcondition, then it is valid with respect to a weaker postcondition.*

$$\forall s, H, c, Q, Q' \bullet \mathit{valid}(s, H, c, Q) \Rightarrow (\forall s', H' \bullet Q(s', H') \Rightarrow Q'(s', H')) \Rightarrow \mathit{valid}(s, H, a, Q')$$

Lemma 10 (Command Validity Framing)**.** *If a command is valid in a given abstract state and a given postcondition, then it is valid if elements are added to the abstract heap, and these elements are still present in each post-state, and the original postcondition holds after removing these elements.*

$$\forall s, H, c, Q, H_\mathrm{F} \bullet \mathit{valid}(s, H, c, Q) \Rightarrow \mathit{valid}(s, H + H_\mathrm{F}, c, (\mathbf{lambda}\ s', H' \bullet H_\mathrm{F} \leq H' \wedge Q(s', H' - H_\mathrm{F})))$$

Function *valid_routine*(*def*) states that abstracted execution of the routine defined by *def* succeeds, or that the routine is *valid*. We say that abstracted execution of a program succeeds, or the program is *valid*, if abstracted execution of each routine succeeds, and abstracted execution of the main command succeeds in the initial store and the empty heap:

$$
\begin{array}{l}
\mathit{valid\_routine}(r(x_1, \ldots, x_n)\ \mathbf{requires}\ a_\mathrm{pre}\ \mathbf{ensures}\ a_\mathrm{post} := c) \equiv \\
\quad \forall v_1, \ldots, v_n \bullet \\
\qquad \mathbf{let}\ s = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \_ \mapsto 0\}\ \mathbf{in} \\
\qquad \mathit{produce}(s, \emptyset, a_\mathrm{pre}, (\mathbf{lambda}\ s_1, H_1 \bullet \\
\qquad\quad \mathit{valid}(s, H_1, c, (\mathbf{lambda}\ s_2, H_2 \bullet \\
\qquad\qquad \mathit{consume}(s_1[\mathsf{result} := s_2(\mathsf{result})], H_2, a_\mathrm{post}, (\mathbf{lambda}\ s_3, H_3 \bullet H_3 = \emptyset))))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{valid\_program}(\mathit{def}_1\ \cdots\ \mathit{def}_n\ c) \equiv \\
\quad \mathit{valid\_routine}(\mathit{def}_1) \wedge \cdots \wedge \mathit{valid\_routine}(\mathit{def}_n) \\
\quad \wedge\ \mathit{valid}(\{\_ \mapsto 0\}, \emptyset, c, (\mathbf{lambda}\ s', H' \bullet \mathbf{true}))
\end{array}
$$

Example 22. *Consider the program of Example 21. Write down the abstracted execution of routine* swap.

$$
\begin{array}{ll}
& \mathit{valid\_routine}(\mathsf{swap}(\mathsf{x},\mathsf{y})\cdots) \\
\Leftrightarrow & \forall v_1, v_2 \bullet \\
& \quad \textbf{let } s = \{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \_ \mapsto 0\} \textbf{ in} \\
& \quad\quad \mathit{produce}(s, \emptyset, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } s_1, H_1 \bullet \\
& \quad\quad\quad \mathit{valid}(s, H_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad\quad \mathit{consume}(s_1[\mathsf{result} := s_2(\mathsf{result})], H_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset))) \\
\Leftrightarrow & \forall v_1, v_2 \bullet \\
& \quad \mathit{produce}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \_ \mapsto 0\}, \emptyset, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } s_1, H_1 \bullet \\
& \quad\quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \_ \mapsto 0\}, H_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(s_1[\mathsf{result} := s_2(\mathsf{result})], H_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset))) \\
\Leftrightarrow & \forall v_1, v_2, v_3 \bullet \\
& \quad \mathit{produce}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_3|\!\}, \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } s_1, H_1 \bullet \\
& \quad\quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \_ \mapsto 0\}, H_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(s_1[\mathsf{result} := s_2(\mathsf{result})], H_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset))) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_3, v_2 \mapsto v_4|\!\}, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad\quad \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}[\mathsf{result} := s_2(\mathsf{result})], H_2, \\
& \quad\quad\quad\quad \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset)) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{x2} \mapsto v_3, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_3, v_2 \mapsto v_4|\!\}, \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad\quad \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}[\mathsf{result} := s_2(\mathsf{result})], H_2, \\
& \quad\quad\quad\quad \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset)) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{x2} \mapsto v_3, \mathsf{y2} \mapsto v_4, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_3, v_2 \mapsto v_4|\!\}, [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad\quad \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}[\mathsf{result} := s_2(\mathsf{result})], H_2, \\
& \quad\quad\quad\quad \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset)) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{valid}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{x2} \mapsto v_3, \mathsf{y2} \mapsto v_4, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_4, v_2 \mapsto v_4|\!\}, [\mathsf{y}] := \mathsf{x2}, \\
& \quad\quad \textbf{lambda } s_2, H_2 \bullet \\
& \quad\quad\quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}[\mathsf{result} := s_2(\mathsf{result})], H_2, \\
& \quad\quad\quad\quad \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset)) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}, \{\!|v_1 \mapsto v_4, v_2 \mapsto v_3|\!\}, \\
& \quad\quad \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \\
& \quad \mathit{consume}(\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2, \mathsf{v} \mapsto v_3, \mathsf{w} \mapsto v_4, \_ \mapsto 0\}, \{\!|v_2 \mapsto v_3|\!\}, \\
& \quad\quad \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } s_3, H_3 \bullet H_3 = \emptyset) \\
\Leftrightarrow & \forall v_1, v_2, v_3, v_4 \bullet v_1 \neq v2 \Rightarrow \emptyset = \emptyset \\
\Leftrightarrow & \textbf{true}
\end{array}
$$

It follows from the abstracted execution of swap *in this example and the abstracted execution of the main command in Example 21 that the program in Example 21 is valid.*

Exercise 9. *Elaborate the abstracted execution of the following routine. Is it valid?*

```
add(a, b)
  requires −1 < a
  ensures result = a + b
:=
  result := b;
  k := 0;
  while k < a
    inv −1 < k ∧ k − 1 < a ∧ result − k = b
  do (k := k + 1; result := result + 1)
```

Exercise 10. *Elaborate the abstracted execution of the following routine. Is it valid?*

$$
\begin{array}{l}
\mathsf{copy}(\mathsf{p},\mathsf{q}) \\
\quad \textbf{requires } \mathsf{p} \mapsto ?\mathsf{v} * \mathsf{q} \mapsto ?\mathsf{w} \\
\quad \textbf{ensures } \mathsf{p} \mapsto ?\mathsf{v2} * \mathsf{q} \mapsto ?\mathsf{w2} * \mathsf{v2} = \mathsf{w} \wedge \mathsf{w2} = \mathsf{w} \\
:= \mathsf{x} := [\mathsf{q}]; [\mathsf{p}] := \mathsf{x}
\end{array}
$$

Exercise 11. *Elaborate the abstracted execution of the following program. Is it valid?*

$$\textbf{predicate } \mathsf{list}(\mathsf{p}) := \textbf{if } \mathsf{p} = 0 \textbf{ then } \mathsf{p} = 0 \textbf{ else } (\mathsf{p} \mapsto ?\mathsf{value} * \mathsf{p} + 1 \mapsto ?\mathsf{next} * \mathsf{list}(\mathsf{next}))$$

```
add(p, v)
  requires list(p)
  ensures list(result)
:=
  open list(p);
  if p = 0 then
    result := cons(v, 0)
  else (
    next := [p + 1];
    next := add(next, v);
    [p + 1] := next;
    result := p
  );
  close list(result)
```

$$\mathsf{myList} := 0; \mathsf{myList} := \mathsf{add}(\mathsf{myList}, 5); \mathsf{myList} := \mathsf{add}(\mathsf{myList}, 10)$$

2.5 Soundness of abstracted execution

In this section, we characterize the relationship between abstracted executions and concrete executions. Specifically, we prove that if abstracted execution of an annotated program succeeds, then the erasure of this program does not go wrong. We do so by defining the notion of a *valid configuration*, and by proving that in case of a valid program the initial configuration is a valid configuration, and that execution steps preserve configuration validity. Since the error configuration is not a valid configuration, it follows directly that valid programs do not go wrong.

We define a valid configuration as a configuration that is abstracted by some valid abstracted configuration. The abstracted configurations are the pairs of abstract states and annotated continuations, and the error configuration.

The set *AnnConts* of annotated continuations is defined inductively as follows:

- **done** is an annotated continuation
- if c is an annotated command and K is an annotated continuation, then $c; K$ is an annotated continuation
- if s is a store, x is a program variable name, and K is an annotated continuation, then $\mathbf{ret}(s, x, K)$ is an annotated continuation

$$\mathit{AbsConfigs} = \mathit{AbsStates} \times \mathit{AnnConts} \cup \{\mathbf{error}\}$$

An abstracted configuration is *valid* if it is not an error configuration and its annotated continuation is valid with respect to the abstract state. Validity of annotated continuations is defined by the following rules.

$$
\begin{array}{l}
\mathit{valid\_ann\_cont}(s, H, \mathbf{done}) \equiv \mathbf{true} \\
\mathit{valid\_ann\_cont}(s, H, c; K) \equiv \mathit{valid}(s, H, c, (\mathbf{lambda}\ s', H' \bullet \mathit{valid\_ann\_cont}(s', H', K))) \\
\mathit{valid\_ann\_cont}(s, H, \mathbf{ret}(s', x, K)) \equiv \mathit{valid\_ann\_cont}(s'[x := s(\mathsf{result})], H, K)
\end{array}
$$

$$
\begin{array}{l}
\mathit{valid\_abs\_config}(\langle (s, H), K \rangle) \equiv \mathit{valid\_ann\_cont}(s, H, K) \\
\mathit{valid\_abs\_config}(\mathbf{error}) \equiv \mathbf{false}
\end{array}
$$

Example 23. *We have:*

$$\mathit{valid\_abs\_cont}(\{\mathsf{p} \mapsto 5, \_ \mapsto 0\}, \{\!\{5 \mapsto 7\}\!\}, \mathsf{result} := \mathsf{p}; \mathbf{ret}(\{\_ \mapsto 0\}, \mathsf{x}, [\mathsf{x}] := 8; \mathbf{done}))$$
$$\neg \mathit{valid\_abs\_cont}(\{\mathsf{p} \mapsto 6, \_ \mapsto 0\}, \{\!\{5 \mapsto 7\}\!\}, \mathsf{result} := \mathsf{p}; \mathbf{ret}(\{\_ \mapsto 0\}, \mathsf{x}, [\mathsf{x}] := 8; \mathbf{done}))$$
$$\neg \mathit{valid\_abs\_config}(\langle(\{\mathsf{p} \mapsto 6, \_ \mapsto 0\}, \{\!\{5 \mapsto 7\}\!\}), \mathsf{result} := \mathsf{p}; \mathbf{ret}(\{\_ \mapsto 0\}, \mathsf{x}, [\mathsf{x}] := 8; \mathbf{done})\rangle)$$

We say an abstract configuration *abstracts* a concrete configuration if the stores are equal, the abstract heap abstracts the concrete heap, and the concrete continuation is the erasure of the annotated continuation.

$$\langle (s, H), K\rangle \triangleleft \langle (s', h), \kappa\rangle \Leftrightarrow s = s' \wedge H \triangleleft^* h \wedge \kappa = \mathit{erasure}(K)$$

Example 24. *Assume the predicate definition* **predicate** $\mathsf{Cell}(\mathsf{x}, \mathsf{y}) := \mathsf{x} \mapsto \mathsf{y}$. *We then have:*

$$\langle(\{\mathsf{p} \mapsto 5, \_ \mapsto 0\}, \{\!\{\mathsf{Cell}(5, 3)\}\!\}), \mathbf{open}\ \mathsf{Cell}(\mathsf{p}, 3); [\mathsf{p}] := 4; \mathbf{done}\rangle \triangleleft \langle(\{\mathsf{p} \mapsto 5, \_ \mapsto 0\}, \{\!\{5 \mapsto 3\}\!\}), [\mathsf{p}] := 4; \mathbf{done}\rangle$$

A concrete configuration is *valid* if it is abstracted by some valid abstracted configuration.

$$\mathit{valid\_config}(\gamma) \equiv \exists C \in \mathit{AbsConfigs} \bullet C \triangleleft \gamma \wedge \mathit{valid\_abs\_config}(C)$$

Lemma 11. *Consider a concrete configuration γ and a valid abstracted configuration C. If C abstracts γ, then there exists a valid abstracted configuration C' that abstracts γ and whose continuation does not start with a ghost command.*

Proof. By induction on the length of the annotated continuation. □

Lemma 12. *The initial configuration of a valid program is a valid configuration.*

$$\mathit{valid\_program}(\mathit{program}) \Rightarrow \mathit{valid\_config}(\gamma_0)$$

Proof. Follows immediately from the definition of program validity. □

Lemma 13 (Preservation)**.** *If γ is a configuration of a concrete execution of a valid program, and γ is valid, and there is an execution step from γ to γ', then γ' is valid.*

$$\mathit{valid\_program}(\mathit{program}) \Rightarrow \mathit{valid\_config}(\gamma) \Rightarrow \gamma \rightsquigarrow \gamma' \Rightarrow \mathit{valid\_config}(\gamma')$$

Proof. By case analysis on the step rule. □

Theorem 1 (Soundness of Abstracted Execution)**.** *If an annotated program is valid, then its erasure does not go wrong.*

$$\mathit{valid\_program}(\mathit{program}) \Rightarrow \gamma_0 \not\rightsquigarrow^* \mathbf{error}$$

Proof. By combining Lemmas 12 and 13, we obtain that each reachable configuration is valid. It follows that the error configuration is not reachable. □

3 Logic and symbolic execution

The goal of verifying a program is to check that no concrete execution of the program goes wrong. However, a typical program has infinitely many executions, and furthermore, each execution may be arbitrarily long, or even non-terminating. Therefore, it is impossible in general to check each concrete execution directly. In the last section, we defined abstracted execution. Abstracted execution in effect reduces the infinite set of unbounded-length concrete program executions to an infinite set of abstracted routine executions whose length is linear in the size of the routine. However, this set is still infinite. Symbolic execution then performs the final step: it reduces the infinite set of abstracted routine executions to a finite set of symbolic routine executions, by performing the execution using symbolic values instead of concrete values. This is possible because a single symbolic value may represent an infinite number of concrete values. The number of symbolic executions performed for a routine is exponential in the number of branch points in the routine (i.e., the number of conditional assertions and conditional commands involved), but it is finite.

In the remainder of this section, we first introduce the syntax and semantics of the logical terms and formulae which we use to represent symbolic execution states. Then we introduce symbolic states and we define the set of abstract states represented by a symbolic state, and the well-formedness of a symbolic state. In Sections 3.4 through 3.6, we define symbolic consumption, symbolic production, and symbolic command execution, respectively. Finally, in the last subsection we define routine verification and program verification, we state the soundness theorem, and we sketch a proof.

3.1 Logic: Syntax and semantics

Symbolic execution states are represented using logical symbols, terms, and formulae, the syntax of which is given by the following grammar.

σ	::=	*one of* $\mathsf{a}, \mathsf{b}, \mathsf{i}, \mathsf{j}, \mathsf{x}, \mathsf{y}, \ldots$	symbol
t	::=		term
		σ	symbol term
		n	literal term
		$(t + t)$	addition term
		$(t - t)$	subtraction term
ϕ	::=		formula
		$t = t$	equality formula
		$t < t$	inequality formula
		$(\phi \wedge \phi)$	conjunction formula
		$(\phi \vee \phi)$	disjunction formula
		$\neg\phi$	negation formula

Notice that we print symbols using a slanted sans-serif font, to distinguish them from program variables, which are printed using an upright sans-serif font.

The meaning of a symbol, a term, or a formula depends on the *interpretation* used. An interpretation is a total function from symbols to values:

$$\mathit{Interpretations} = ||\sigma|| \to \mathbb{Z}$$

The *evaluation* $[\![t]\!]_I$ of a term t under an interpretation I is defined as follows:

$$[\![\sigma]\!]_I = I(\sigma) \qquad [\![n]\!]_I = n \qquad [\![t_1 + t_2]\!]_I = [\![t_1]\!]_I + [\![t_2]\!]_I \qquad [\![t_1 - t_2]\!]_I = [\![t_1]\!]_I - [\![t_2]\!]_I$$

Similarly, the *truth* $I \vDash \phi$ of a formula ϕ under an interpretation I is defined as follows:

$$I \vDash t_1 = t_2 \Leftrightarrow [\![t_1]\!]_I = [\![t_2]\!]_I \qquad I \vDash t_1 < t_2 \Leftrightarrow [\![t_1]\!]_I < [\![t_2]\!]_I \qquad I \vDash \phi_1 \wedge \phi_2 \Leftrightarrow (I \vDash \phi_1) \wedge (I \vDash \phi_2)$$

$$I \vDash \phi_1 \vee \phi_2 \Leftrightarrow (I \vDash \phi_1) \vee (I \vDash \phi_2)$$

Example 25. *Suppose* $I = \{\mathsf{x} \mapsto 5, \mathsf{y} \mapsto 10, \_ \mapsto 0\}$. *Then we have:*

$$[\![\mathsf{x} + \mathsf{y}]\!]_I = 15 \qquad I \vDash \mathsf{x} = \mathsf{y} - 5 \wedge \mathsf{x} + \mathsf{x} = \mathsf{y}$$

We denote by $\mathsf{symbs}\, t$ and by $\mathsf{symbs}\, \phi$ the set of symbols that appear in term t and formula ϕ, respectively.

The *powerset* $\wp(A)$ of a set A is the set of subsets of A:

$$\wp(A) = \{B \mid B \subseteq A\}$$

A *theory* is a set of formulae.

$$\mathit{Theories} = \wp(\mathit{Formulae})$$

A theory is *true* under an interpretation if all formulae in the theory are true under the interpretation.

$$I \vDash \Sigma \Leftrightarrow (\forall \phi \in \Sigma \bullet I \vDash \phi)$$

In this case, we say the interpretation is a *model* of the theory. A formula is a *valid consequence* of a theory if the formula is true under all interpretations under which the theory is true, i.e., under all models of the theory. We also say the formula is *valid under the theory.*

$$\Sigma \vDash \phi \Leftrightarrow (\forall I \bullet I \vDash \Sigma \Rightarrow I \vDash \phi)$$

Example 26. *Suppose*

$$I_1 = \{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 3, \_ \mapsto 0\} \qquad I_2 = \{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1, \_ \mapsto 0\} \qquad \Sigma_1 = \{\neg(\mathsf{x} = \mathsf{y}), \neg(\mathsf{y} = \mathsf{z})\}$$

$$\Sigma_2 = \{\mathsf{x} = \mathsf{y}, \mathsf{y} = \mathsf{z}\}$$

We then have

$$I_1 \vDash \Sigma_1 \qquad I_2 \vDash \Sigma_1 \qquad \Sigma_2 \vDash \mathsf{x} = \mathsf{z} \qquad \Sigma_1 \nvDash \neg(\mathsf{x} = \mathsf{z})$$

A *theorem prover* is a program that accepts a theory and a formula as input, and attempts to determine whether the formula is a valid consequence of the theory. Its output is either Valid or Unknown. We write $\Sigma \vdash_{\mathrm{R}} \phi$ to denote that theorem prover R outputs Valid when given input Σ and ϕ. We say a theorem prover is *sound* if it outputs Valid only when the input formula is a valid consequence of the input theory.

$$\text{sound}\, R \Leftrightarrow (\forall \Sigma, \phi \bullet \Sigma \vdash_R \phi \Rightarrow \Sigma \vDash \phi)$$

We say a theorem prover is *complete* if it outputs Valid whenever the input formula is a valid consequence of the input theory.

$$\text{complete}\, R \Leftrightarrow (\forall \Sigma, \phi \bullet \Sigma \vDash \phi \Rightarrow \Sigma \vdash_R \phi)$$

3.2 Symbolic states

The main components of a symbolic execution state (or *symbolic state*) are the *symbolic store* and the *symbolic heap*, which are similar to the store and abstract heap of abstracted execution, except that logical terms are used instead of values. Specifically, the symbolic store maps program variable names to logical terms, and the symbolic heap is a finite multiset of *symbolic heap elements*. A symbolic heap element is a *symbolic points-to element* or a *symbolic predicate instance*.

$$\begin{gathered}
\mathit{SymbolicStores} = ||x|| \to \mathit{Terms} \\
\mathit{SymbolicPointsToElements} = \{\hat{\ell} \mapsto \hat{v} \mid \hat{\ell}, \hat{v} \in \mathit{Terms}\} \\
\mathit{SymbolicPredicateInstances} = \{p(\hat{v_1}, \ldots, \hat{v_n}) \mid \hat{v_1}, \ldots, \hat{v_n} \in \mathit{Terms}\} \\
\mathit{SymbolicHeapElements} = \mathit{SymbolicPointsToElements} \cup \mathit{SymbolicPredicateInstances} \\
\mathit{SymbolicHeaps} = \mathit{SymbolicHeapElements} \overset{\text{fin}}{\rightharpoonup} \mathbb{N}_0
\end{gathered}$$

There are two additional components: the *used set* and the *path condition*. The used set is the set of logical symbols that have already been used during symbolic execution. When symbolic execution requires a fresh symbol, it chooses a symbol that is not in the used set and then adds it to the used set. The path condition is a set of logical formulae. It is the set of constraints that hold over the interpretations of the symbols on the current symbolic execution path. The notation $\wp(A)$ denotes the *power set* of set A, i.e., the set of all subsets of A.

$$\begin{gathered}
\mathit{UsedSets} = \wp(\mathit{Symbols}) \\
\mathit{PathConditions} = \wp(\mathit{Formulae}) \\
\mathit{SymbolicStates} = \mathit{UsedSets} \times \mathit{PathConditions} \times \mathit{SymbolicStores} \times \mathit{SymbolicHeaps}
\end{gathered}$$

We use the following notational convention: U ranges over used sets, Σ ranges over path conditions, $\hat{s}$ ranges over symbolic stores, and $\hat{H}$ ranges over symbolic heaps. In general, we denote the symbolic counterpart of some concept X as $\hat{X}$. Specifically, $\hat{\ell}$ and $\hat{v}$ range over logical terms.

We say a symbolic state *represents* an abstract state if there is an interpretation of the symbols used in the symbolic state under which the path condition is true and the *value* of the symbolic store equals the concrete store and the *value* of the symbolic heap equals the abstract heap. We denote the set of abstract states represented by a given symbolic state $\hat{S}$ by $\text{rep}\, \hat{S}$. A path condition is true under an interpretation if all formulae it contains are true under the interpretation. The value of a symbolic store under an interpretation is a function that maps a program variable x to some value v if the symbolic store maps x to some term t and the value of t under the interpretation is v. The value of a symbolic heap under an interpretation is the set of the values of the symbolic heap's elements under the interpretation. The value of a symbolic heap element is the heap element obtained by replacing all terms by their value under the interpretation.

$$\begin{gathered}
[\![\hat{s}]\!]_I = \mathbf{lambda}\ x \bullet [\![\hat{s}(x)]\!]_I \\
[\![\hat{H}]\!]_I = \{\!|[\![\hat{\ell}]\!]_I \mapsto [\![\hat{v}]\!]_I \mid \hat{\ell} \mapsto \hat{v} \in \hat{H}|\!\} \cup \{\!|p([\![\hat{v}_1]\!]_I, \ldots, [\![\hat{v}_n]\!]_I) \mid p(\hat{v}_1, \ldots, \hat{v}_n) \in \hat{H}|\!\} \\
\text{rep}\,(U, \Sigma, \hat{s}, \hat{H}) = \{([\![\hat{s}]\!]_I, [\![\hat{H}]\!]_I) \mid I \vDash \Sigma\}
\end{gathered}$$

We say a symbolic state is *well-formed* if all logical symbols that appear in the path condition, the symbolic store, or the symbolic heap, are in the used set.

$$\mathsf{wf}\ (U, \Sigma, \hat{s}, \hat{H}) \Leftrightarrow \mathsf{symbs}\, \Sigma \cup \mathsf{symbs}\, \hat{s} \cup \mathsf{symbs}\, \hat{H} \subseteq U$$

We have the property that if a symbolic state is well-formed, then for any interpretation I that satisfies the path condition, and for any symbol σ that is not in the used set, and for any value v, it holds that the updated interpretation $I[\sigma := v]$ satisfies the path condition, and the value of the symbolic store and the symbolic heap under $I[\sigma := v]$ equals their value under I.

Example 27. *Consider the symbolic store* $\hat{s} = \{\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} \mapsto \mathsf{a}+1, \_ \mapsto 0\}$*, the symbolic heap* $\hat{H} = \{\!\{\mathsf{a} \mapsto 10, \mathsf{cell}(\mathsf{a}+2, \mathsf{b})\}\!\}$*, and the interpretation* $I = \{\mathsf{a} \mapsto 7, \_ \mapsto 0\}$*. Then we have*

$$[\![\hat{s}]\!]_I = \{\mathsf{x} \mapsto 7, \mathsf{y} \mapsto 8, \_ \mapsto 0\}$$
$$[\![\hat{H}]\!]_I = \{\!\{7 \mapsto 10, \mathsf{cell}(9, 0)\}\!\}$$

Example 28. *We have*

$$(\{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \_ \mapsto 0\}, \emptyset) \in \mathrm{rep}\,(\{x, y\}, \{x < y\}, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \emptyset)$$
$$(\{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 1, \_ \mapsto 0\}, \emptyset) \notin \mathrm{rep}\,(\{x, y\}, \{x < y\}, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \emptyset)$$
$$(\{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \_ \mapsto 0\}, \emptyset) \in \mathrm{rep}\,(\{x, y\}, \{x < y\}, \{\mathsf{x} \mapsto y, \mathsf{y} \mapsto x, \_ \mapsto 0\}, \emptyset)$$
$$(\{\mathsf{x} \mapsto 5, \_ \mapsto 0\}, \{\!\{5 \mapsto 10, \mathsf{Cell}(10, 3)\}\!\}) \in \mathrm{rep}\,(\{p, q\}, \emptyset, \{\mathsf{x} \mapsto p, \_ \mapsto 0\}, \{\!\{p \mapsto q, \mathsf{Cell}(q, p-2)\}\!\})$$

Exercise 12. *Consider used set* $U = \{p, q\}$*, symbolic store* $\hat{s} = \{\mathsf{a} \mapsto p+q, \mathsf{b} \mapsto p-q, \_ \mapsto 0\}$*, symbolic heap* $\hat{H} = \{\!\{p-q-q \mapsto 10\}\!\}$*, and path conditions* $\Sigma_1 = \{0 < p, q < 10\}$ *and* $\Sigma_2 = \{p < 0, q < 10\}$*. Which of the following statements are true?*

$$(\{\mathsf{a} \mapsto 3, \mathsf{b} \mapsto 5, \_ \mapsto 0\}, \{\!\{6 \mapsto 10\}\!\}) \in \mathrm{rep}\,(U, \Sigma_1, \hat{s}, \hat{H})$$
$$(\{\mathsf{a} \mapsto 3, \mathsf{b} \mapsto 5, \_ \mapsto 0\}, \{\!\{6 \mapsto 10\}\!\}) \in \mathrm{rep}\,(U, \Sigma_2, \hat{s}, \hat{H})$$

3.3 Symbolic execution

Abstracted execution involves consumption and production of assertions; analogously, symbolic execution involves symbolic consumption and symbolic production of assertions. Symbolic execution is very similar to abstracted execution, with the following main differences:

- Instead of comparing values, symbolic execution queries a *theorem prover* to see if a given equality between two terms is a valid consequence of the path condition.
- In case of ambiguous matches, instead of backtracking, symbolic consumption fails.
- Instead of evaluating boolean expressions, symbolic execution queries the theorem prover for the validity of a formula under the path condition.
- When executing a conditional construct, instead of choosing one branch or the other based on the value of the condition, symbolic execution proceeds along both paths, while adding the condition or its negation to the path condition.
- Instead of universally quantifying over all values, symbolic execution picks a fresh logical symbol.

We denote the theorem prover used for symbolic execution by R. We assume R is sound. (We do not assume that it is complete.)

Evaluation $[\![e]\!]_{\hat{s}}$ of an arithmetic expression e under a symbolic store $\hat{s}$ yields a logical term; it is defined as follows:

$$[\![x]\!]_{\hat{s}} = \hat{s}(x) \qquad [\![n]\!]_{\hat{s}} = n \qquad [\![e_1 + e_2]\!]_{\hat{s}} = [\![e_1]\!]_{\hat{s}} + [\![e_2]\!]_{\hat{s}} \qquad [\![e_1 - e_2]\!]_{\hat{s}} = [\![e_1]\!]_{\hat{s}} - [\![e_2]\!]_{\hat{s}}$$

We have the following property.

Lemma 14 (Correctness of Symbolic Evaluation of Arithmetic Expressions)**.** *The value v under an interpretation I of a term t obtained by evaluating an expression e under a symbolic store $\hat{s}$ equals the value of e under the concrete store s that is the value of the symbolic store $\hat{s}$ under interpretation I.*

$$[\![[\![e]\!]_{\hat{s}}]\!]_I = [\![e]\!]_{[\![\hat{s}]\!]_I}$$

Evaluation $[\![b]\!]_{\hat{s}}$ of a boolean expression b under a symbolic store $\hat{s}$ yields a formula; it is defined as follows:

$$[\![e_1 = e_2]\!]_{\hat{s}} = ([\![e_1]\!]_{\hat{s}} = [\![e_2]\!]_{\hat{s}}) \qquad [\![e_1 < e_2]\!]_{\hat{s}} = ([\![e_1]\!]_{\hat{s}} < [\![e_2]\!]_{\hat{s}}) \qquad [\![b_1 \wedge b_2]\!]_{\hat{s}} = ([\![b_1]\!]_{\hat{s}} \wedge [\![b_2]\!]_{\hat{s}})$$

$$[\![b_1 \vee b_2]\!]_{\hat{s}} = ([\![b_1]\!]_{\hat{s}} \vee [\![b_2]\!]_{\hat{s}}) \qquad [\![\neg b]\!]_{\hat{s}} = \neg [\![b]\!]_{\hat{s}}$$

We have the following property.

Lemma 15 (Correctness of Symbolic Evaluation of Boolean Expressions)**.** *If evaluation of a boolean expression b under a symbolic store $\hat{s}$ yields a formula ϕ, then ϕ is true under some interpretation I if and only if b evaluates to* **true** *under the concrete store that is the value of $\hat{s}$ under I.*

$$(I \vDash [\![b]\!]_{\hat{s}}) \Leftrightarrow ([\![b]\!]_{[\![\hat{s}]\!]_I} = \mathbf{true})$$

Example 29. *Suppose* $\hat{s} = \{\mathsf{x} \mapsto p + 3, \mathsf{y} \mapsto 9, \_ \mapsto 0\}$*. We then have*

$$[\![\mathsf{x} + \mathsf{y}]\!]_{\hat{s}} = (p + 3) + 9 \qquad [\![\mathsf{x} - 3 < \mathsf{y}]\!]_{\hat{s}} = ((p + 3) - 3 < 9)$$

The next three subsections zoom in on symbolic consumption, symbolic production, and symbolic command execution.

Symbolic execution involves performing *case splits*. Case splitting is based on the following property:

Lemma 16 (Case Splitting Property)**.** *If a given disjunction holds in a given symbolic state, then the symbolic state may be split up into two symbolic states, where one of the disjuncts is added to the path condition of each state.*

$$\Sigma \vDash \phi_1 \vee \phi_2 \Rightarrow \mathrm{rep}\,(U, \Sigma, \hat{s}, \hat{H}) = \mathrm{rep}\,(U, \Sigma \cup \{\phi_1\}, \hat{s}, \hat{H}) \cup \mathrm{rep}\,(U, \Sigma \cup \{\phi_2\}, \hat{s}, \hat{H})$$

Symbolic execution performs *infeasibility checks*. This is based on the following property.

Lemma 17 (Infeasibility Check Property)**.** *If a given formula is false in a given symbolic state, then adding that formula to the path condition yields a symbolic state that does not represent any abstract states.*

$$\Sigma \vDash \neg\phi \Rightarrow \mathrm{rep}\,(U, \Sigma \cup \{\phi\}, \hat{s}, \hat{H}) = \emptyset$$

3.4 Symbolic consumption

Symbolic consumption of an assertion is analogous to abstracted consumption: it checks that some fragment of the current symbolic heap satisfies the assertion, and then removes that fragment from the symbolic heap. Specifically, function *sconsume* takes a path condition, a symbolic store, a symbolic heap, an assertion, and a *symbolic consumption postcondition*, and returns **true** if symbolic consumption of the assertion under the given symbolic state succeeds and all post-states satisfy the postcondition. There are multiple post-states if the assertion contains conditional assertions. Function *sconsume* does not take the used set as an argument since symbolic consumption does not involve choosing fresh symbols and therefore does not involve the used set.

$$\begin{aligned} &\mathit{sconsume}: \\ &\quad \mathit{PathConditions} \times \mathit{SymbolicStores} \times \mathit{SymbolicHeaps} \times ||a|| \\ &\quad \times (\mathit{PathConditions} \times \mathit{SymbolicStores} \times \mathit{SymbolicHeaps} \to \mathbb{B}) \to \mathbb{B} \end{aligned}$$

Function *sconsume* uses the auxiliary functions *smatch_pattern* and *smatch_patterns*. These are the symbolic versions of functions *match_pattern* and *match_patterns* defined in Section 2.2.1. *smatch_pattern*-$(\Sigma, \hat{s}, \hat{v}, \pi)$ attempts to match symbolic value $\hat{v}$ against pattern π under path condition Σ and symbolic store $\hat{s}$. If the match succeeds, the function returns a singleton set containing the symbolic store updated with the required pattern variable bindings. Otherwise, the function returns an empty set. Similarly, function *smatch_patterns*$(\Sigma, \hat{s}, \overline{\hat{v}}, \overline{\pi})$ attempts to match the list of symbolic values $\overline{\hat{v}}$ against the list of patterns $\overline{\pi}$. If each value matches the corresponding pattern, the function returns a singleton set containing the symbolic store updated with the required pattern variable bindings. Otherwise, the function returns an empty set.

Symbolic consumption is defined by the following rules:

$$
\begin{array}{l}
\mathit{smatch\_pattern}(\Sigma, \hat{s}, \hat{v}, e) \equiv \\
\quad \textbf{if } \Sigma \vdash_{\mathrm{R}} [\![e]\!]_{\hat{s}} = \hat{v} \textbf{ then } \{\hat{s}\} \textbf{ else } \emptyset \\
\mathit{smatch\_pattern}(\Sigma, \hat{s}, \hat{v}, ?x) \equiv \\
\quad \{\hat{s}[x := \hat{v}]\} \\
\\
\mathit{smatch\_patterns}(\Sigma, \hat{s}, \epsilon, \epsilon) = \{\hat{s}\} \\
\mathit{smatch\_patterns}(\Sigma, \hat{s}, \hat{v}::\overline{\hat{v}}, \pi::\overline{\pi}) \equiv \\
\quad \{\hat{s}'' \mid \hat{s}' \in \mathit{smatch\_pattern}(\Sigma, \hat{s}, \hat{v}, \pi), \hat{s}'' \in \mathit{smatch\_patterns}(\Sigma, \hat{s}', \overline{\hat{v}}, \overline{\pi})\} \\
\\
\mathit{sconsume}(U, \Sigma, \hat{s}, \hat{H}, e \mapsto ?x, Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \{\hat{\ell} \mapsto \hat{v} \in \hat{H} \mid \Sigma \vdash_{\mathrm{R}} \hat{\ell} = [\![e]\!]_{\hat{s}}\} \textbf{ in} \\
\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(\Sigma, \hat{s}[x := \hat{v}], \hat{H} - \{\!|\hat{\ell} \mapsto \hat{v}|\!\}) \\
\mathit{sconsume}(U, \Sigma, \hat{s}, \hat{H}, p(\pi_1, \ldots, \pi_n), Q) \equiv \\
\quad \textbf{let } \mathit{matches} = \\
\quad\quad \{(\hat{s}', (\hat{v}_1, \ldots, \hat{v}_n)) \mid \\
\quad\quad\quad p(\hat{v}_1, \ldots, \hat{v}_n) \in \hat{H}, \hat{s}' \in \mathit{smatch\_patterns}(\Sigma, \hat{s}, \hat{v}_1::\cdots::\hat{v}_n::\epsilon, \pi_1::\cdots::\pi_n::\epsilon)\} \textbf{ in} \\
\quad \exists (\hat{s}', (\hat{v}_1, \ldots, \hat{v}_n)) \bullet \mathit{matches} = \{(\hat{s}', (\hat{v}_1, \ldots, \hat{v}_n))\} \wedge Q(\Sigma, \hat{s}', \hat{H} - \{\!|p(\hat{v}_1, \ldots, \hat{v}_n)|\!\}) \\
\mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, b, Q) \equiv \\
\quad (\Sigma \vdash_{\mathrm{R}} [\![b]\!]_{\hat{s}}) \wedge Q(\hat{s}, \hat{H}) \\
\mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, a_1 * a_2, Q) \equiv \\
\quad \mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, a_1, (\textbf{lambda } \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\hat{s}', \hat{H}', a_2, Q))) \\
\mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, \textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2, Q) \equiv \\
\quad (\Sigma \nvdash_{\mathrm{R}} \neg[\![b]\!]_{\hat{s}} \Rightarrow \mathit{sconsume}(\Sigma \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, a_1, Q)) \\
\quad \wedge\ (\Sigma \nvdash_{\mathrm{R}} [\![b]\!]_{\hat{s}} \Rightarrow \mathit{sconsume}(\Sigma \cup \{\neg[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, a_2, Q))
\end{array}
$$

In words, the rules say the following:

- Symbolically matching an expression pattern e against a symbolic value $\hat{v}$ under a path condition Σ and symbolic store $\hat{s}$ queries the theorem prover to see if the equality between $\hat{v}$ and the symbolic value of e under $\hat{s}$ follows from Σ. If the theorem prover outputs Valid, the pattern is considered to match the symbolic value and a singleton set containing the unchanged symbolic store is returned. Otherwise, the pattern is considered to not match the symbolic value and an empty set is returned.

- Symbolically matching a variable pattern $?x$ against a symbolic value $\hat{v}$ always succeeds; the function returns the symbolic store updated to bind variable x to value $\hat{v}$.

- Symbolically matching the empty list of values against the empty list of patterns always succeeds; the symbolic store is returned unchanged.

- Symbolically matching a nonempty list of values against a nonempty list of patterns first attempts to match the first value against the first pattern. If this match succeeds, the remaining list of values is matched against the remaining list of patterns under the updated symbolic store.

- Symbolically consuming a points-to assertion first compiles the set of matching symbolic points-to elements in the symbolic heap. A symbolic points-to element is considered to match the points-to assertion if the theorem prover proves that the symbolic address of the points-to element equals the symbolic value of the address expression of the points-to assertion under the path condition. If exactly one points-to element matches the assertion, the postcondition is checked with a symbolic store updated to bind the pattern variable to the symbolic value of the matching points-to element, and a symbolic heap where the matching points-to element has been removed. If no element matches the assertion, or more than one element matches the assertion (a situation called an *ambiguous match*), symbolic consumption fails.

- Symbolically consuming a predicate assertion is analogous: first, a set of matching symbolic predicate instances is compiled, where each match is accompanied by the resulting symbolic store. If there is exactly one match, the postcondition is checked under the updated symbolic store and the symbolic

heap obtained by removing the matching symbolic predicate instance; if there is no match or there are multiple matching elements, symbolic consumption fails.

- Symbolically consuming a boolean assertion checks with the theorem prover that the truth of the assertion under the symbolic store follows from the path condition. If the theorem prover verifies this, the postcondition is checked under an unchanged symbolic state; otherwise, symbolic consumption fails.
- Symbolically consuming a separate conjunction first consumes the left-hand side and then consumes the right-hand side in the resulting symbolic state.
- Symbolically consuming a conditional assertion performs a *case split*: the **then** branch is symbolically consumed under a path condition where the symbolic value of the condition under the current symbolic store is added; additionally, the **else** branch is symbolically consumed under a path condition where the negation of the symbolic value of the condition under the current symbolic store is added. Consumption of both branches must succeed for consumption of the conditional assertion to succeed. However, before a branch is consumed, an *infeasibility check* is first performed. A symbolic state $\hat{S}$ is said to be *infeasible* if it does not represent any abstracted states, i.e., if $\text{rep}\,\hat{S} = \emptyset$. The theorem prover is asked to attempt to prove the negation of the formula that is being added to the path condition; if the theorem prover succeeds in proving this, we know the resulting symbolic state is infeasible and therefore there is no point in performing the consumption. Only if the theorem prover does not succeed, the branch is consumed.

Example 30. *We have*

$$\begin{gathered}
\mathit{smatch_pattern}(\{p = q\}, \{\mathsf{x} \mapsto \mathsf{p}, _ \mapsto 0\}, q, \mathsf{x}) = \{\{\mathsf{x} \mapsto \mathsf{p}, _ \mapsto 0\}\} \\
\mathit{smatch_pattern}(\emptyset, \{\mathsf{x} \mapsto \mathsf{p}, _ \mapsto 0\}, q, \mathsf{x}) = \emptyset \\
\mathit{smatch_patterns}(\{p = q\}, \{\mathsf{x} \mapsto \mathsf{p}, _ \mapsto 0\}, q{::}p{::}\epsilon, ?\mathsf{x}{::}\mathsf{x}{::}\epsilon) = \{\{\mathsf{x} \mapsto \mathsf{q}, _ \mapsto 0\}\}
\end{gathered}$$

Example 31. *Suppose the path condition is* $\{q = 5\}$*, the symbolic store is* $\{\mathsf{i} \mapsto p, \_ \mapsto 0\}$ *and the abstract heap is* $\{\!\{p \mapsto q, \mathsf{cell}(q, 1)\}\!\}$*. What is the result of symbolically consuming the assertion* $\mathsf{i} \mapsto ?\mathsf{j} * \mathsf{j} = 5$*?*

$$\begin{aligned}
&\mathit{sconsume}(\{q = 5\}, \{\mathsf{i} \mapsto p, _ \mapsto 0\}, \{\!\{p \mapsto q, \mathsf{cell}(q, 1)\}\!\}, \mathsf{i} \mapsto ?\mathsf{j} * \mathsf{j} = 5, Q) \\
\Leftrightarrow\ &\mathit{sconsume}(\{q = 5\}, \{\mathsf{i} \mapsto p, \mathsf{j} \mapsto q, _ \mapsto 0\}, \{\!\{\mathsf{cell}(q, 1)\}\!\}, \mathsf{j} = 5, Q) \\
\Leftrightarrow\ &\{q = 5\} \vdash_R q = 5 \Rightarrow Q(\{q = 5\}, \{\mathsf{i} \mapsto p, \mathsf{j} \mapsto q, _ \mapsto 0\}, \{\!\{\mathsf{cell}(q, 1)\}\!\}) \\
\Leftrightarrow\ &Q(\{q = 5\}, \{\mathsf{i} \mapsto p, \mathsf{j} \mapsto q, _ \mapsto 0\}, \{\!\{\mathsf{cell}(q, 1)\}\!\})
\end{aligned}$$

Example 32. *Suppose the path condition is* $\{p = q \vee p = r\}$*, the symbolic store is* $\{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \mathsf{z} \mapsto r, \_ \mapsto 0\}$*, and the symbolic heap is* $\{\!\{q \mapsto 5, r \mapsto 10\}\!\}$*. Symbolically consume the assertion* $\mathsf{x} \mapsto ?\mathsf{v}$*.*

$$\begin{aligned}
&\mathit{sconsume}(\{p = q \vee p = r\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, _ \mapsto 0\}, \{\!\{q \mapsto 5, r \mapsto 10\}\!\}, \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
\Leftrightarrow\ &\textbf{let } \mathit{matches} = \{\hat{\ell} \mapsto \hat{v} \in \{\!\{q \mapsto 5, r \mapsto 10\}\!\} \mid \{p = q \vee p = r\} \vdash_R \hat{\ell} = p\} \textbf{ in} \\
&\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(\Sigma, \hat{s}[\mathsf{v} := \hat{v}], \{\!\{q \mapsto 5, r \mapsto 10\}\!\} - \{\!\{\hat{\ell} \mapsto \hat{v}\}\!\}) \\
\Leftrightarrow\ &\textbf{let } \mathit{matches} = \{q \mapsto 5 \mid \{p = q \vee p = r\} \vdash_R q = p\} \cup \{r \mapsto 10 \mid \{p = q \vee p = r\} \vdash_R r = p\} \textbf{in} \\
&\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(\Sigma, \hat{s}[\mathsf{v} := \hat{v}], \{\!\{q \mapsto 5, r \mapsto 10\}\!\} - \{\!\{\hat{\ell} \mapsto \hat{v}\}\!\}) \\
\Leftrightarrow\ &\exists \hat{\ell} \mapsto \hat{v} \bullet \emptyset = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(\Sigma, \hat{s}[\mathsf{v} := \hat{v}], \{\!\{q \mapsto 5, r \mapsto 10\}\!\} - \{\!\{\hat{\ell} \mapsto \hat{v}\}\!\}) \\
\Leftrightarrow\ &\textbf{false}
\end{aligned}$$

What is the result of symbolically consuming the assertion $(\textbf{if } \mathsf{x} = \mathsf{y} \textbf{ then } \mathsf{x} = \mathsf{x} \textbf{ else } \mathsf{x} = \mathsf{x}) * \mathsf{x} \mapsto ?\mathsf{v}$*?*

$$
\begin{array}{ll}
& \mathit{sconsume}(\{p = q \vee p = r\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \\
& \quad (\textbf{if } \mathsf{x} = \mathsf{y} \textbf{ then } \mathsf{x} = \mathsf{x} \textbf{ else } \mathsf{x} = \mathsf{x}) * \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
\Leftrightarrow & \mathit{sconsume}(\{p = q \vee p = r\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \\
& \quad \textbf{if } \mathsf{x} = \mathsf{y} \textbf{ then } \mathsf{x} = \mathsf{x} \textbf{ else } \mathsf{x} = \mathsf{x}, \\
& \quad \textbf{lambda } \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\Sigma', \hat{s}', \hat{H}', \mathsf{x} \mapsto ?\mathsf{v}, Q)) \\
\Leftrightarrow & \\
& \quad (\{p = q \vee p = r\} \not\vdash_{\mathrm{R}} \neg(p = q) \Rightarrow \\
& \qquad \mathit{sconsume}(\{p = q \vee p = r, p = q\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} = \mathsf{x}, \\
& \qquad\quad \textbf{lambda } \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\Sigma', \hat{s}', \hat{H}', \mathsf{x} \mapsto ?\mathsf{v}, Q))) \\
& \wedge \\
& \quad (\{p = q \vee p = r\} \not\vdash_{\mathrm{R}} p = q \Rightarrow \\
& \qquad \mathit{sconsume}(\{p = q \vee p = r, \neg(p = q)\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} = \mathsf{x}, \\
& \qquad\quad \textbf{lambda } \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\Sigma', \hat{s}', \hat{H}', \mathsf{x} \mapsto ?\mathsf{v}, Q))) \\
\Leftrightarrow & \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, p = q\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} = \mathsf{x}, \\
& \qquad \textbf{lambda } \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\Sigma', \hat{s}', \hat{H}', \mathsf{x} \mapsto ?\mathsf{v}, Q)) \\
& \wedge \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, \neg(p = q)\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} = \mathsf{x}, \\
& \qquad \textbf{lambda } \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sconsume}(\Sigma', \hat{s}', \hat{H}', \mathsf{x} \mapsto ?\mathsf{v}, Q)) \\
\Leftrightarrow & \\
& \quad \{p = q \vee p = r, p = q\} \vdash_R p = p \ \Rightarrow \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, p = q\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
& \wedge \\
& \quad \{p = q \vee p = r, \neg(p = q)\} \vdash_R p = p \ \Rightarrow \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, \neg(p = q)\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
\Leftrightarrow & \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, p = q\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
& \wedge \\
& \quad \mathit{sconsume}(\{p = q \vee p = r, \neg(p = q)\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \_ \mapsto 0\}, \{\!|q \mapsto 5, r \mapsto 10|\!\}, \mathsf{x} \mapsto ?\mathsf{v}, Q) \\
\Leftrightarrow & \\
& \quad Q(\{p = q \vee p = r, p = q\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \mathsf{v} \mapsto 5, \_ \mapsto 0\}, \{\!|r \mapsto 10|\!\}) \\
& \wedge \\
& \quad Q(\{p = q \vee p = r, \neg(p = q)\}, \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q, \mathsf{v} \mapsto 10, \_ \mapsto 0\}, \{\!|q \mapsto 5|\!\})
\end{array}
$$

This example shows that in the verification approach described in this text, it is sometimes necessary to insert a conditional construct solely for the purpose of causing the verification tool to perform a case split*. In general terms,* case splitting *on a disjunction $\phi_1 \vee \phi_2$ means replacing a symbolic execution $\mathit{sexec}(\Sigma)$ with a conjunction $\mathit{sexec}(\Sigma \cup \{\phi_1\}) \wedge \mathit{sexec}(\Sigma \cup \{\phi_2\})$, where ϕ_1 is added to the path condition of one branch and ϕ_2 is added to the path condition of the other branch. The new symbolic executions are referred to as branches since a symbolic execution where case splits occur forms a tree. Each path in the tree from the root (the initial symbolic execution configuration) to a leaf (a final symbolic execution configuration) is called a symbolic execution path.*

Symbolic consumption is *sound*; that is, it correctly mimics consumption on abstracted states. Specifically, if symbolic consumption succeeds with respect to a given postcondition, then consumption of the same assertion in a corresponding abstracted state succeeds and yields post-states that are represented by some symbolic state that satisfies the original postcondition.

Lemma 18 (Soundness of symbolic consumption)**.** *Consider a path condition Σ, a symbolic store $\hat{s}$, a symbolic heap $\hat{H}$, an assertion a, and a symbolic consumption postcondition Q. Further consider an interpretation I. Let s and H be the value of $\hat{s}$ and $\hat{H}$ under I. If symbolic consumption of a succeeds under $\Sigma, \hat{s}, \hat{H}$ with respect to postcondition Q, then consumption of a in abstract state (s, H) succeeds with respect to a postcondition that is true for a given post-state (s', H') if there is a path condition Σ', a symbolic store $\hat{s}'$, and a symbolic heap $\hat{H}'$ such that s' is the value of $\hat{s}'$ under I, $\hat{H}'$ is the value of $\hat{H}$*

under I, $Q(\Sigma', \hat{s}', \hat{H}')$ holds, and all symbols used by Σ', $\hat{s}'$, or $\hat{H}'$ are also used by Σ, $\hat{s}$, or $\hat{H}$.

$$
\begin{array}{l}
\forall \Sigma, \hat{s}, \hat{H}, a, Q, I, s, H \bullet \\
\quad \Sigma \vDash I \wedge s = [\![\hat{s}]\!]_I \wedge H = [\![\hat{H}]\!]_I \Rightarrow \\
\quad \mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, a, Q) \Rightarrow \\
\quad \mathit{consume}(s, H, a, (\textbf{lambda } s', H' \bullet \\
\qquad \exists \Sigma', \hat{s}', \hat{H}' \bullet \\
\qquad\quad \Sigma' \vDash I \wedge s' = [\![\hat{s}']\!]_I \wedge H' = [\![\hat{H}']\!]_I \\
\qquad\quad \wedge\ Q(\Sigma', \hat{s}', \hat{H}') \\
\qquad\quad \wedge\ \mathsf{symbs}\,\Sigma' \cup \mathsf{symbs}\,\hat{s}' \cup \mathsf{symbs}\,\hat{H}' \subseteq \mathsf{symbs}\,\Sigma \cup \mathsf{symbs}\,\hat{s} \cup \mathsf{symbs}\,\hat{H}))
\end{array}
$$

Proof. By induction on the structure of the assertion a. □

3.5 Symbolic production

Symbolic production of an assertion is analogous to abstracted production: it checks the postcondition for all symbolic heaps obtained by extending the current symbolic heap with a heap fragment that satisfies the assertion. Specifically, given a used set U, a path condition Σ, a symbolic store $\hat{s}$, a symbol heap $\hat{H}$, an assertion a, and a symbolic production postcondition Q, $\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, a, Q)$ denotes that $Q(U', \Sigma', \hat{s}', \hat{H}')$ holds, for each symbolic state $(U', \Sigma', \hat{s}', \hat{H}')$ obtained by extending $\hat{H}$ with a symbolic heap fragment that satisfies a.

$$\mathit{sproduce} : \mathit{SymbolicStates} \times ||a|| \times\ (\mathit{SymbolicStates} \to \mathbb{B}) \to \mathbb{B}$$

The function is defined by the following rules:

$$
\begin{array}{l}
\mathit{sproduce\_pattern}(U, \hat{s}, e, Q) \equiv Q(U, \hat{s}, [\![e]\!]_{\hat{s}}) \\
\mathit{sproduce\_pattern}(U, \hat{s}, ?x, Q) \equiv \exists \sigma \in \mathit{Symbols} \setminus U \bullet Q(U \cup \{\sigma\}, \hat{s}[x := \sigma], \sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathit{sproduce\_patterns}(U, \hat{s}, \epsilon, Q) \equiv Q(U, \hat{s}, \epsilon) \\
\mathit{sproduce\_patterns}(U, \hat{s}, \pi :: \overline{\pi}, Q) \equiv \\
\quad \mathit{sproduce\_pattern}(\hat{s}, \pi, (\textbf{lambda } U', \hat{s}', \hat{v} \bullet \\
\qquad \mathit{sproduce\_patterns}(U', \hat{s}', \overline{\pi}, (\textbf{lambda } U'', \hat{s}'', \overline{\hat{v}} \bullet Q(U'', \hat{s}'', \hat{v} :: \overline{\hat{v}})))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, e \mapsto ?x, Q) \equiv \\
\quad \exists \sigma \in \mathit{Symbols} \setminus U \bullet Q(U \cup \{\sigma\}, \Sigma, \hat{s}[x := \sigma], \hat{H} + \{\!\{[\![e]\!]_{\hat{s}} \mapsto \sigma\}\!\}) \\
\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, p(\pi_1, \ldots, \pi_n), Q) \equiv \\
\quad \mathit{sproduce\_patterns}(U, \hat{s}, \pi_1 :: \cdots :: \pi_n :: \epsilon, (\textbf{lambda } U', \hat{s}', \hat{v}_1 :: \cdots :: \hat{v}_n :: \epsilon \bullet \\
\qquad Q(U', \Sigma, \hat{s}', \hat{H} + \{\!\{p(\hat{v}_1, \ldots, \hat{v}_n)\}\!\}))) \\
\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, b, Q) \equiv \\
\quad \Sigma \nvdash_{\mathrm{R}} \neg [\![b]\!]_{\hat{s}} \Rightarrow Q(U, \Sigma \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}) \\
\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, a_1 * a_2, Q) \equiv \\
\quad \mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, a_1, (\textbf{lambda } U', \Sigma', \hat{s}', \hat{H}' \bullet \mathit{sproduce}(U', \Sigma', \hat{s}', \hat{H}', a_2, Q)) \\
\mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, \textbf{if } b \textbf{ then } a_1 \textbf{ else } a_2, Q) \equiv \\
\quad (\Sigma \nvdash_{\mathrm{R}} \neg [\![b]\!]_{\hat{s}} \Rightarrow \mathit{sproduce}(U, \Sigma \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, a_1, Q)) \\
\quad \wedge\ (\Sigma \nvdash_{\mathrm{R}} [\![b]\!]_{\hat{s}} \Rightarrow \mathit{sproduce}(U, \Sigma \cup \{\neg [\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, a_2, Q))
\end{array}
$$

Example 33. *Suppose $\hat{s} = \{\mathsf{x} \mapsto \mathsf{p}, \_ \mapsto 0\}$. Then we have*

$$
\begin{array}{c}
\mathit{sproduce\_pattern}(\{\mathsf{p}\}, \hat{s}, \mathsf{x}, Q) \Leftrightarrow Q(\{\mathsf{p}\}, \hat{s}, \mathsf{p}) \\
\mathit{sproduce\_pattern}(\{\mathsf{p}\}, \hat{s}, ?\mathsf{x}, Q) \Leftrightarrow Q(\{\mathsf{p}, \mathsf{q}\}, \{\mathsf{x} \mapsto \mathsf{q}, \_ \mapsto 0\}, \mathsf{q}) \\
\mathit{sproduce\_patterns}(\{\mathsf{p}\}, \hat{s}, ?\mathsf{x} :: \mathsf{x} :: \epsilon, Q) \Leftrightarrow Q(\{\mathsf{p}, \mathsf{q}\}, \{\mathsf{x} \mapsto \mathsf{q}, \_ \mapsto 0\}, \mathsf{q} :: \mathsf{q} :: \epsilon)
\end{array}
$$

Example 34. *Suppose $\hat{s} = \{\mathsf{p} \mapsto \mathsf{p}, \_ \mapsto 0\}$. Elaborate the symbolic production of assertion $\mathsf{p} \mapsto ?\mathsf{x} * \mathsf{Cell}(\mathsf{x}, 9)$.*

$$
\begin{array}{ll}
 & \mathit{sproduce}(\{\mathsf{p}\}, \emptyset, \hat{s}, \emptyset, \mathsf{p} \mapsto ?\mathsf{x} * \mathsf{Cell}(\mathsf{x}, 9), Q) \\
\Leftrightarrow & \mathit{sproduce}(\{\mathsf{p}, \mathsf{x}\}, \emptyset, \{\mathsf{p} \mapsto \mathsf{p}, \mathsf{x} \mapsto \mathsf{x}, \_ \mapsto 0\}, \{\!\{\mathsf{p} \mapsto \mathsf{x}\}\!\}, \mathsf{Cell}(\mathsf{x}, 9), Q) \\
\Leftrightarrow & Q(\{\mathsf{p}, \mathsf{x}\}, \emptyset, \{\mathsf{p} \mapsto \mathsf{p}, \mathsf{x} \mapsto \mathsf{x}, \_ \mapsto 0\}, \{\!\{\mathsf{p} \mapsto \mathsf{x}, \mathsf{Cell}(\mathsf{x}, 9)\}\!\})
\end{array}
$$

The postcondition Q must hold for a symbolic heap that represents all abstract heaps that satisfy the assertion.

Elaborate the symbolic production of assertion $\mathsf{p} = 6 * \mathsf{Cell}(\mathsf{p}, 10)$.

$$\begin{array}{l} \mathit{sproduce}(\{\mathsf{p}\}, \emptyset, \hat{s}, \emptyset, \mathsf{p} = 6 * \mathsf{Cell}(\mathsf{p}, 10), Q) \\ \Leftrightarrow (\emptyset \nvdash_{\mathrm{R}} \neg(\mathsf{p} = 6) \Rightarrow \mathit{sproduce}(\{\mathsf{p}\}, \{\mathsf{p} = 6\}, \hat{s}, \emptyset, \mathsf{Cell}(\mathsf{p}, 10), Q))) \\ \Leftrightarrow Q(\{\mathsf{p}\}, \{\mathsf{p} = 6\}, \hat{s}, \{\!|\mathsf{Cell}(\mathsf{p}, 10)|\!\}) \end{array}$$

Production of a boolean assertion adds the evaluation of the boolean expression under the symbolic store to the path condition.

Exercise 13. *Suppose the used set is* $\{\mathsf{a}\}$*, the path condition is* $\{5 < \mathsf{a}\}$*, and the symbolic store is* $\{\mathsf{x} \mapsto \mathsf{a}, \_ \mapsto 0\}$*. Elaborate the symbolic production of the following assertions.*

- $\mathsf{cell}(?\mathsf{x}, ?\mathsf{x}) * \mathsf{x} \mapsto ?\mathsf{x}$
- $\mathsf{x} < 10 * \mathsf{x} + 5 \mapsto ?\mathsf{y}$
- $\mathsf{cell}(?\mathsf{x}, ?\mathsf{y}) * \mathsf{x} \mapsto ?\mathsf{xv} * \mathsf{y} \mapsto ?\mathsf{yv}$
- $\mathsf{x} \mapsto ?\mathsf{v} * \textbf{if } \neg(\mathsf{v} = 0) \textbf{ then } \mathsf{v} \mapsto ?\mathsf{w} \textbf{ else } \mathsf{v} = \mathsf{v}$

Symbolic production is *sound*; that is, it correctly mimics production on abstract states.

Lemma 19 (Soundness of Symbolic Production)**.** *Consider a well-formed symbolic state* $(U, \Sigma, \hat{s}, \hat{H})$*, an assertion* a*, and a symbolic production postcondition* Q*. Further consider an interpretation* I*. Let* s *and* H *be the value of* $\hat{s}$ *and* $\hat{H}$ *under* I*. If production of* a *succeeds in symbolic state* $(U, \Sigma, \hat{s}, \hat{H})$ *with respect to postcondition* Q*, then production of* a *succeeds in abstract state* (s, H)*, with respect to a postcondition that holds for a post-state* (s', H') *if there exists a well-formed symbolic state* $(U', \Sigma', \hat{s}', \hat{H}')$ *and an interpretation* I' *such that* s' *is the value of* $\hat{s}'$ *under* I*,* H' *is the value of* $\hat{H}'$ *under* I*, the postcondition* Q *holds for* $(U', \Sigma', \hat{s}', \hat{H}')$*, the new used set* U' *includes the old used set* U*, and the new interpretation* I' *coincides with the old one* I *on the old used set* U*.*

$$\begin{array}{l} \forall U, \Sigma, \hat{s}, \hat{H}, a, Q, I, s, H \bullet \\ \quad \mathsf{wf}\ (U, \Sigma, \hat{s}, \hat{H}) \Rightarrow \Sigma \vDash I \Rightarrow s = [\![\hat{s}]\!]_I \Rightarrow H = [\![\hat{H}]\!]_I \Rightarrow \\ \quad \mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H}, a, Q) \Rightarrow \\ \quad \mathit{produce}(s, H, a, (\textbf{lambda } s', H' \bullet \\ \qquad \exists U', \Sigma', \hat{s}', \hat{H}' \bullet \mathsf{wf}\ (U', \Sigma', \hat{s}', \hat{H}') \wedge U \subseteq U' \wedge I'|_U = I|_U \\ \qquad\quad \wedge\ \Sigma' \vDash I' \wedge s' = [\![\hat{s}']\!]_{I'} \wedge H' = [\![\hat{H}']\!]_{I'} \wedge Q(U', \Sigma', \hat{s}', \hat{H}'))) \end{array}$$

Proof. By induction on the structure of assertion a. □

3.6 Symbolic execution of commands

Symbolic execution of commands is very similar to abstracted execution of commands. For a given symbolic state $(U, \Sigma, \hat{s}, \hat{H})$, command c, and symbolic execution postcondition Q, $\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, c, Q)$ denotes that symbolic execution of c in state $(U, \Sigma, \hat{s}, \hat{H})$ succeeds and for each post-state $(U', \Sigma', \hat{s}', \hat{H}')$, $Q(U', \Sigma', \hat{s}', \hat{H}')$ holds.

$$\mathit{verify} : \mathit{SymbolicStates} \times ||c|| \times (\mathit{SymbolicStates} \to \mathbb{B}) \to \mathbb{B}$$

Function *verify* is defined by the rules in Figure 3.

Example 35. *Consider the program*

$$\begin{array}{l} \mathsf{swap}(\mathsf{x}, \mathsf{y}) \\ \quad \textbf{requires } \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w} \\ \quad \textbf{ensures } \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v} \\ \quad := \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2} \\ \\ \mathsf{i} := \textbf{cons}(5, 8);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i} + 1);\ \textbf{dispose } \mathsf{i};\ \textbf{dispose } \mathsf{i} + 1 \end{array}$$

$$
\begin{array}{l}
\mathit{shavoc}(U, \hat{s}, \{x_1, \ldots, x_n\}, Q) \equiv \\
\quad \exists \{\sigma_1, \ldots, \sigma_n\} \subseteq \mathit{Symbols} \setminus U \bullet Q(U \cup \{\sigma_1, \ldots, \sigma_n\}, \hat{s}[x_1 := \sigma_1, \ldots, x_n := \sigma_n]) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{skip}, Q) \equiv Q(U, \Sigma, \hat{s}, \hat{H}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, x := e, Q) \equiv Q(U, \Sigma, \hat{s}[x := [\![e]\!]_{\hat{s}}], \hat{H}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, c_1; c_2, Q) \equiv \\
\quad \mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, c_1, (\mathbf{lambda}\ U', \Sigma', \hat{s}', \hat{H}' \bullet \mathit{verify}(U', \Sigma', \hat{s}', \hat{H}', c_2, Q))) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, Q) \equiv \\
\quad (\Sigma \nvdash_{\mathrm{R}} \neg [\![b]\!]_{\hat{s}} \Rightarrow \mathit{verify}(U, \Sigma \cup \{[\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, c_1, Q)) \\
\quad \wedge\ (\Sigma \nvdash_{\mathrm{R}} [\![b]\!]_{\hat{s}} \Rightarrow \mathit{verify}(U, \Sigma \cup \{\neg [\![b]\!]_{\hat{s}}\}, \hat{s}, \hat{H}, c_2, Q)) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{while}\ b\ \mathbf{inv}\ a\ \mathbf{do}\ c, Q) \equiv \\
\quad \mathit{sconsume}(\Sigma, \hat{s}, \hat{H}, a, (\mathbf{lambda}\ \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
\quad\quad \mathit{shavoc}(U, \hat{s}, \mathit{modifies}(c), (\mathbf{lambda}\ U_2, \hat{s}_2 \bullet \\
\quad\quad\quad \mathit{sproduce}(U_2, \Sigma_1, \hat{s}_2, \emptyset, a, (\mathbf{lambda}\ U_3, \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \\
\quad\quad\quad\quad \Sigma \nvdash_{\mathrm{R}} \neg [\![b]\!]_{\hat{s}_2} \Rightarrow \\
\quad\quad\quad\quad \mathit{verify}(U_3, \Sigma_3 \cup \{[\![b]\!]_{\hat{s}_2}\}, \hat{s}_2, \hat{H}_3, c, (\mathbf{lambda}\ U_4, \Sigma_4, \hat{s}_4, \hat{H}_4 \bullet \\
\quad\quad\quad\quad\quad \mathit{sconsume}(\Sigma_4, \hat{s}_4, \hat{H}_4, a, (\mathbf{lambda}\ \Sigma_5, \hat{s}_5, \hat{H}_5 \bullet \hat{H}_5 = \emptyset)))) \\
\quad\quad\quad\quad \wedge\ (\Sigma \nvdash_{\mathrm{R}} [\![b]\!]_{\hat{s}_2} \Rightarrow Q(U_3, \Sigma_3 \cup \{\neg [\![b]\!]_{\hat{s}_2}\}, \hat{s}_2, \hat{H}_1 + \hat{H}_3)))))))) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, x := \mathbf{cons}(e_1, \ldots, e_n), Q) \equiv \\
\quad \exists \sigma \in \mathit{Symbols} - U \bullet Q(U \cup \{\sigma\}, \Sigma, \hat{s}[x := \sigma], \hat{H} + \{\!\{\sigma \mapsto [\![e_1]\!]_{\hat{s}}, \ldots, \sigma + n - 1 \mapsto [\![e_n]\!]_{\hat{s}}\}\!\}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, x := [e], Q) \equiv \\
\quad \mathbf{let}\ \mathit{matches} = \{\hat{\ell} \mapsto \hat{v} \in \hat{H} \mid \Sigma \vdash_{\mathrm{R}} \hat{\ell} = [\![e]\!]_{\hat{s}}\}\ \mathbf{in} \\
\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(U, \Sigma, \hat{s}[x := \hat{v}], \hat{H}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, [e] := e', Q) \equiv \\
\quad \mathbf{let}\ \mathit{matches} = \{\hat{\ell} \mapsto \hat{v} \in \hat{H} \mid \Sigma \vdash_{\mathrm{R}} \hat{\ell} = [\![e]\!]_{\hat{s}}\}\ \mathbf{in} \\
\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(U, \Sigma, \hat{s}, \hat{H} - \{\!\{\hat{\ell} \mapsto \hat{v}\}\!\} + \{\!\{\hat{\ell} \mapsto [\![e']\!]_{\hat{s}}\}\!\}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{dispose}\ e, Q) \equiv \\
\quad \mathbf{let}\ \mathit{matches} = \{\hat{\ell} \mapsto \hat{v} \in \hat{H} \mid \Sigma \vdash_{\mathrm{R}} \hat{\ell} = [\![e]\!]_{\hat{s}}\}\ \mathbf{in} \\
\quad \exists \hat{\ell} \mapsto \hat{v} \bullet \mathit{matches} = \{\hat{\ell} \mapsto \hat{v}\} \wedge Q(U, \Sigma, \hat{s}, \hat{H} - \{\!\{\hat{\ell} \mapsto \hat{v}\}\!\}) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{open}\ p(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathbf{let\ predicate}\ p(x_1, \ldots, x_n) := a \in \mathit{PredicateDefs}\ \mathbf{in} \\
\quad \mathbf{let}\ \hat{v}_1 = [\![e_1]\!]_{\hat{s}}, \ldots, \hat{v}_n = [\![e_n]\!]_{\hat{s}}\ \mathbf{in} \\
\quad \mathbf{let}\ \hat{s}' = \{x_1 \mapsto \hat{v}_1, \ldots, x_n \mapsto \hat{v}_n, \_ \mapsto 0\}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{matches} = \{p(\hat{v}'_1, \ldots, \hat{v}'_n) \in \hat{H} \mid \Sigma \vdash_{\mathrm{R}} \hat{v}_1 = \hat{v}'_1 \wedge \cdots \wedge \hat{v}_n = \hat{v}'_n\}\ \mathbf{in} \\
\quad \exists\ p(\hat{v}'_1, \ldots, \hat{v}'_n) \bullet \mathit{matches} = \{p(\hat{v}'_1, \ldots, \hat{v}'_n)\} \wedge \\
\quad \mathit{sproduce}(U, \Sigma, \hat{s}, \hat{H} - \{p(\hat{v}'_1, \ldots, \hat{v}'_n)\}, a, (\mathbf{lambda}\ U', \Sigma', \hat{s}', \hat{H}' \bullet Q(U', \Sigma', \hat{s}, \hat{H}'))) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, \mathbf{close}\ p(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathbf{let\ predicate}\ p(x_1, \ldots, x_n) := a \in \mathit{PredicateDefs}\ \mathbf{in} \\
\quad \mathbf{let}\ \hat{v}_1 = [\![e_1]\!]_{\hat{s}}, \ldots, \hat{v}_n = [\![e_n]\!]_{\hat{s}}\ \mathbf{in} \\
\quad \mathit{sconsume}(\Sigma, \{x_1 \mapsto \hat{v}_1, \ldots, x_n \mapsto \hat{v}_n, \_ \mapsto 0\}, \hat{H}, a, (\mathbf{lambda}\ \Sigma', \hat{s}', \hat{H}' \bullet \\
\quad\quad Q(U, \Sigma', \hat{s}, \hat{H}' + \{\!\{p(\hat{v}_1, \ldots, \hat{v}_n)\}\!\}))) \\
\mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, x := r(e_1, \ldots, e_n), Q) \equiv \\
\quad \mathbf{let}\ r(x_1, \ldots, x_n)\ \mathbf{requires}\ a_{\mathrm{pre}}\ \mathbf{ensures}\ a_{\mathrm{post}} \in \mathit{RoutineSpecs}\ \mathbf{in} \\
\quad \mathbf{let}\ \hat{s}_1 = \{x_1 \mapsto [\![e_1]\!]_{\hat{s}}, \ldots, x_n \mapsto [\![e_n]\!]_{\hat{s}}, \_ \mapsto 0\}\ \mathbf{in} \\
\quad \mathit{sconsume}(\Sigma, \hat{s}_1, \hat{H}, a_{\mathrm{pre}}, (\mathbf{lambda}\ \Sigma', \hat{s}_2, \hat{H}_2 \bullet \\
\quad\quad \exists \sigma \in \mathit{Symbols} \setminus U \bullet \mathit{sproduce}(U, \Sigma', \hat{s}_2[\mathsf{result} := \sigma], \hat{H}_2, a_{\mathrm{post}}, \\
\quad\quad\quad (\mathbf{lambda}\ U_3, \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet Q(U_3, \Sigma_3, \hat{s}[x := \sigma], \hat{H}_3)))))
\end{array}
$$

Figure 3: Symbolic execution rules

Suppose $\hat{s}_0 = \{\_ \mapsto 0\}$, $\hat{H}_0 = \emptyset$ *and* $Q_0 = \textbf{lambda}\ U', \Sigma', \hat{s}', \hat{H}' \bullet \textbf{true}$. *Write down the symbolic execution of the program's main command:*

$$
\begin{array}{ll}
& \mathit{verify}(\emptyset, \emptyset, \hat{s}_0, \hat{H}_0, \mathsf{i} := \textbf{cons}(5,8);\ \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i}+1);\ \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \mathit{verify}(\{i\}, \emptyset, \{\mathsf{i} \mapsto i, \_ \mapsto 0\}, \{\!|i \mapsto 5, i+1 \mapsto 8|\!\}, \mathsf{m} := \mathsf{swap}(\mathsf{i}, \mathsf{i}+1);\ \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0) \\
\Leftrightarrow & \mathit{sconsume}(\emptyset, \{\mathsf{x} \mapsto i, \mathsf{y} \mapsto i+1, \_ \mapsto 0\}, \{\!|i \mapsto 5, i+1 \mapsto 8|\!\}, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda}\ \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
& \quad \mathit{sproduce}(\{i, v\}, \Sigma_1, \hat{s}_1[\mathsf{result} := v], \hat{H}_1, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda}\ U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \quad\quad \mathit{verify}(U_2, \Sigma_2, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \hat{H}_2, \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \mathit{sconsume}(\emptyset, \{\mathsf{x} \mapsto i, \mathsf{y} \mapsto i+1, \mathsf{v} \mapsto 5, \_ \mapsto 0\}, \{\!|i+1 \mapsto 8|\!\}, \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda}\ \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
& \quad \mathit{sproduce}(\{i, v\}, \Sigma_1, \hat{s}_1[\mathsf{result} := v], \hat{H}_1, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda}\ U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \quad\quad \mathit{verify}(U_2, \Sigma_2, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \hat{H}_2, \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \mathit{sproduce}(\{i, v\}, \emptyset, \{\mathsf{x} \mapsto i, \mathsf{y} \mapsto i+1, \mathsf{v} \mapsto 5, \mathsf{w} \mapsto 8, \mathsf{result} \mapsto v, \_ \mapsto 0\}, \emptyset, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \quad \textbf{lambda}\ U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \quad\quad \mathit{verify}(U_2, \Sigma_2, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \hat{H}_2, \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \mathit{sproduce}(\{i, v\}, \emptyset, \{\mathsf{x} \mapsto i, \mathsf{y} \mapsto i+1, \mathsf{v} \mapsto 5, \mathsf{w} \mapsto 8, \mathsf{result} \mapsto v, \_ \mapsto 0\}, \{\!|i \mapsto 8|\!\}, \mathsf{y} \mapsto \mathsf{v}, \\
& \quad \textbf{lambda}\ U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \quad\quad \mathit{verify}(U_2, \Sigma_2, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \hat{H}_2, \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \mathit{verify}(\{i, v\}, \emptyset, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \{\!|i \mapsto 8, i+1 \mapsto 5|\!\}, \textbf{dispose}\ \mathsf{i};\ \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \mathit{verify}(\{i, v\}, \emptyset, \{\mathsf{i} \mapsto i, \mathsf{m} \mapsto v, \_ \mapsto 0\}, \{\!|i+1 \mapsto 5|\!\}, \textbf{dispose}\ \mathsf{i}+1, Q_0)) \\
\Leftrightarrow & \textbf{true}
\end{array}
$$

Consider a function $f : A \to B$, and a set $C \subseteq A$. Then we define the *restriction* $f|_C$ of f to C as the partial function that coincides with f on elements of C, but that is not defined on elements outside of C.

$$f|_C(x) = f(x) \quad \text{if } x \in C \qquad\qquad \operatorname{dom} f|_C = \operatorname{dom} f \cap C$$

Command verification is *sound*. Specifically, if a command verifies in a given symbolic state and with respect to a given postcondition, then this command is valid in any abstract state that is represented by the symbolic state and with respect to a postcondition that is true for a given abstract post-state if there is a symbolic post-state that satisfies the original postcondition and that represents the abstract post-state.

Lemma 20 (Soundness of command verification)**.** *Consider a well-formed symbolic state* $(U, \Sigma, \hat{s}, \hat{H})$*, a command* c*, and a symbolic execution postcondition* Q*. Further consider an interpretation* I*. Let* s *and* H *be the value of* $\hat{s}$ *and* $\hat{H}$ *under* I*. If symbolic execution of* c *succeeds in symbolic state* $(U, \Sigma, \hat{s}, \hat{H})$ *with respect to postcondition* Q*, then abstracted execution of* c *succeeds in abstract state* (s, H)*, with respect to a postcondition that holds for a post-state* (s', H') *if there exists a well-formed symbolic state* $(U', \Sigma', \hat{s}', \hat{H}')$ *and an interpretation* I' *such that* s' *is the value of* $\hat{s}'$ *under* I*,* H' *is the value of* $\hat{H}'$ *under* I*, the postcondition* Q *holds for* $(U', \Sigma', \hat{s}', \hat{H}')$*, the new used set* U' *includes the old used set* U*, and the new interpretation* I' *coincides with the old one* I *on the old used set* U*.*

$$
\begin{array}{l}
\forall U, \Sigma, \hat{s}, \hat{H}, c, Q, I, s, H \bullet \\
\quad \mathsf{wf}\ (U, \Sigma, \hat{s}, \hat{H}) \Rightarrow \Sigma \vDash I \Rightarrow s = [\![\hat{s}]\!]_I \Rightarrow H = [\![\hat{H}]\!]_I \Rightarrow \\
\quad \mathit{verify}(U, \Sigma, \hat{s}, \hat{H}, c, Q) \Rightarrow \\
\quad \mathit{valid}(s, H, c, (\textbf{lambda}\ s', H' \bullet \\
\quad\quad \exists U', \Sigma', \hat{s}', \hat{H}' \bullet \mathsf{wf}\ (U', \Sigma', \hat{s}', \hat{H}') \wedge U \subseteq U' \wedge I'|_U = I|_U \\
\quad\quad\quad \wedge\ \Sigma' \vDash I' \wedge s' = [\![\hat{s}']\!]_{I'} \wedge H' = [\![\hat{H}']\!]_{I'} \wedge Q(U', \Sigma', \hat{s}', \hat{H}')))
\end{array}
$$

Proof. By induction on the structure of command c. □

3.7 Verification of routines and programs

Verification of a routine is similar to abstracted execution of a routine.

$$
\begin{array}{l}
\mathit{verify\_routine}(r(x_1, \ldots, x_n)\ \textbf{requires}\ a_{\text{pre}}\ \textbf{ensures}\ a_{\text{post}} := c) \equiv \\
\quad \exists \{\sigma_1, \ldots, \sigma_n\} \in \mathit{Symbols} \bullet \\
\qquad \textbf{let}\ \hat{s} = \{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n, \_ \mapsto 0\}\ \textbf{in} \\
\qquad \textbf{let}\ U = \{\sigma_1, \ldots, \sigma_n\}\ \textbf{in} \\
\qquad \mathit{sproduce}(U, \emptyset, \hat{s}, \emptyset, a_{\text{pre}}, (\textbf{lambda}\ U_1, \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
\qquad\quad \mathit{verify}(U_1, \Sigma_1, \hat{s}, \hat{H}_1, c, (\textbf{lambda}\ U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
\qquad\qquad \mathit{sconsume}(\Sigma_2, \hat{s}_1[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, a_{\text{post}}, (\textbf{lambda}\ \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset))))))
\end{array}
$$

Routine verification is *sound*.

Lemma 21 (Soundness of routine verification)**.** *If a routine verifies, it is valid.*

$$\mathit{verify\_routine}(\mathit{def}) \Rightarrow \mathit{valid\_routine}(\mathit{def})$$

Proof. Follows directly from the soundness of symbolic consumption, symbolic production, and symbolic command execution. □

Verification of a program is similar to abstracted execution of a program.

$$
\begin{array}{l}
\mathit{verify\_program}(\mathit{def}_1\ \cdots\ \mathit{def}_n\ c) \equiv \\
\quad \mathit{verify\_routine}(\mathit{def}_1) \wedge \cdots \wedge \mathit{verify\_routine}(\mathit{def}_n) \\
\quad \wedge\ \mathit{verify}(\emptyset, \emptyset, \{\_ \mapsto 0\}, \emptyset, c, (\textbf{lambda}\ U', \Sigma', s', H' \bullet \textbf{true}))
\end{array}
$$

Example 36. *Consider the program of Example 35. Write down the symbolic execution of routine* swap.

$$
\begin{array}{ll}
& \mathit{verify\_routine}(\mathsf{swap}(\mathsf{x},\mathsf{y})\cdots) \\
\Leftrightarrow & \textbf{let } \hat{s} = \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\} \textbf{ in} \\
& \quad \textbf{let } U = \{x, y\} \textbf{ in} \\
& \quad \mathit{sproduce}(U, \emptyset, \hat{s}, \emptyset, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } U_1, \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
& \qquad \mathit{verify}(U_1, \Sigma_1, \hat{s}, \hat{H}_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad\quad \mathit{sconsume}(\Sigma_2, \hat{s}_1[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset))) \\
\Leftrightarrow & \mathit{sproduce}(\{x, y\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \emptyset, \mathsf{x} \mapsto ?\mathsf{v} * \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } U_1, \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
& \quad \mathit{verify}(U_1, \Sigma_1, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \hat{H}_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \qquad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad\quad \mathit{sconsume}(\Sigma_2, \hat{s}_1[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset))) \\
\Leftrightarrow & \mathit{sproduce}(\{x, y, v\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \_ \mapsto 0\}, \{\!\{x \mapsto v\}\!\}, \mathsf{y} \mapsto ?\mathsf{w}, \textbf{lambda } U_1, \Sigma_1, \hat{s}_1, \hat{H}_1 \bullet \\
& \quad \mathit{verify}(U_1, \Sigma_1, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \hat{H}_1, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \qquad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad\quad \mathit{sconsume}(\Sigma_2, \hat{s}_1[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset))) \\
\Leftrightarrow & \mathit{verify}(\{x, y, v, w\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \_ \mapsto 0\}, \{\!\{x \mapsto v, y \mapsto w\}\!\}, \mathsf{x2} := [\mathsf{x}]; \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad \mathit{sconsume}(\Sigma_2, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad\quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset)) \\
\Leftrightarrow & \mathit{verify}(\{x, y, v, w\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{x2} \mapsto v, \_ \mapsto 0\}, \{\!\{x \mapsto v, y \mapsto w\}\!\}, \mathsf{y2} := [\mathsf{y}];\ [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad \mathit{sconsume}(\Sigma_2, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad\quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset)) \\
\Leftrightarrow & \mathit{verify}(\{x, y, v, w\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{x2} \mapsto v, \mathsf{y2} \mapsto w, \_ \mapsto 0\}, \{\!\{x \mapsto v, y \mapsto w\}\!\}, [\mathsf{x}] := \mathsf{y2};\ [\mathsf{y}] := \mathsf{x2}, \\
& \quad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad \mathit{sconsume}(\Sigma_2, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad\quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset)) \\
\Leftrightarrow & \mathit{verify}(\{x, y, v, w\}, \emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{x2} \mapsto v, \mathsf{y2} \mapsto w, \_ \mapsto 0\}, \{\!\{x \mapsto w, y \mapsto w\}\!\}, [\mathsf{y}] := \mathsf{x2}, \\
& \quad \textbf{lambda } U_2, \Sigma_2, \hat{s}_2, \hat{H}_2 \bullet \\
& \qquad \mathit{sconsume}(\Sigma_2, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}[\mathsf{result} := \hat{s}_2(\mathsf{result})], \hat{H}_2, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \qquad\quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset)) \\
\Leftrightarrow & \mathit{sconsume}(\emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}, \{\!\{x \mapsto w, y \mapsto v\}\!\}, \mathsf{x} \mapsto \mathsf{w} * \mathsf{y} \mapsto \mathsf{v}, \\
& \quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset) \\
\Leftrightarrow & \mathit{sconsume}(\emptyset, \{\mathsf{x} \mapsto x, \mathsf{y} \mapsto y, \mathsf{v} \mapsto v, \mathsf{w} \mapsto w, \_ \mapsto 0\}, \{\!\{y \mapsto v\}\!\}, \mathsf{y} \mapsto \mathsf{v}, \\
& \quad \textbf{lambda } \Sigma_3, \hat{s}_3, \hat{H}_3 \bullet \hat{H}_3 = \emptyset) \\
\Leftrightarrow & \textbf{true}
\end{array}
$$

Exercise 14. *Elaborate the symbolic execution of the following routine. Does it verify?*

$$
\begin{array}{l}
\mathsf{add}(\mathsf{a}, \mathsf{b}) \\
\quad \textbf{requires } -1 < \mathsf{a} \\
\quad \textbf{ensures } \mathsf{result} = \mathsf{a} + \mathsf{b} \\
:= \\
\quad \mathsf{result} := \mathsf{b}; \\
\quad \mathsf{k} := 0; \\
\quad \textbf{while } \mathsf{k} < \mathsf{a} \\
\quad\quad \textbf{inv } -1 < \mathsf{k} \wedge \mathsf{k} - 1 < \mathsf{a} \wedge \mathsf{result} - \mathsf{k} = \mathsf{b} \\
\quad \textbf{do } (\mathsf{k} := \mathsf{k} + 1; \mathsf{result} := \mathsf{result} + 1)
\end{array}
$$

Exercise 15. *Elaborate the symbolic execution of the following routine. Does it verify?*

$$
\begin{array}{l}
\mathsf{copy}(\mathsf{p}, \mathsf{q}) \\
\quad \textbf{requires } \mathsf{p} \mapsto ?\mathsf{v} * \mathsf{q} \mapsto ?\mathsf{w} \\
\quad \textbf{ensures } \mathsf{p} \mapsto ?\mathsf{v2} * \mathsf{q} \mapsto ?\mathsf{w2} * \mathsf{v2} = \mathsf{w} \wedge \mathsf{w2} = \mathsf{w} \\
:= \mathsf{x} := [\mathsf{q}]; [\mathsf{p}] := \mathsf{x}
\end{array}
$$

Exercise 16. *Elaborate the symbolic execution of the following program. Does it verify?*

```
predicate list(p) := if p = 0 then p = 0 else (p ↦?value * p + 1 ↦?next * list(next))

add(p, v)
  requires list(p)
  ensures list(result)
:=
  open list(p);
  if p = 0 then
    result := cons(v, 0)
  else (
    next := [p + 1];
    next := add(next, v);
    [p + 1] := next;
    result := p
  );
  close list(result)

myList := 0; myList := add(myList, 5); myList := add(myList, 10)
```

Program verification is sound with respect to program validity.

Lemma 22. *If a program verifies, then it is valid.*

$$verify\_program(program) \Rightarrow valid\_program(program)$$

Proof. Follows directly from the soundness of routine verification and the soundness of command verification. □

The main property of the verification approach is the following:

Theorem 2 (Soundness)**.** *If a program verifies, it does not go wrong.*

$$verify\_program(program) \Rightarrow \gamma_0 \not\rightsquigarrow^* \textbf{error}$$

Proof. Follows directly from the soundness of program verification with respect to program validity, and the soundness of program validity. □