Probabilistic Assertions: Extended Semantics and Proof

Adrian Sampson Pavel Panchekha Todd Mytkowicz
Kathryn S. McKinley Dan Grossman Luis Ceze

This addendum expands the semantics section in the paper *Expressing and Verifying Probabilistic Assertions* in PLDI 2014 and gives the full proof of the associated theorem.

1 Semantics

This section formalizes a simple probabilistic imperative language, ProbCore, and Mayhap's distribution extraction process. We describe ProbCore's syntax, a *concrete semantics* for nondeterministic run-time execution, and a *symbolic semantics* for distribution extraction. Executing a ProbCore program under the symbolic semantics produces a Bayesian network for a passert statement. We prove this extracted distribution is equivalent to the original program under the concrete semantics, demonstrating the soundness of Mayhap's core analysis.

1.1 Core Language

ProbCore is an imperative language with assignment, conditionals, and loops. Programs use probabilistic behavior by sampling from a distribution and storing the result, written $v \leftarrow D$. Without loss of generality, a program is a sequence

$$
\begin{aligned}
&P \equiv S \,;; \textbf{ passert } C \\
&C \equiv E < E \mid E = E \mid C \wedge C \mid C \vee C \mid \neg C \\
&E \equiv E + E \mid E * E \mid E \div E \mid R \mid V \\
&S \equiv V := E \mid V \leftarrow D \mid S \,;\, S \mid \textbf{skip} \mid \textbf{if } C\ S\ S \mid \textbf{while } C\ S \\
&R \in \mathbb{R},\ V \in \text{Variables},\ D \in \text{Distributions}
\end{aligned}
$$

Figure 1: Syntax of ProbCore.

of statements followed by a single passert, since we may verify a passert at any program point by examining the program prefix leading up to a passert.

Figure 1 defines ProbCore's syntax for programs denoted P, which consist of conditionals C, expressions E, and statements S.

1.2 Concrete Semantics

The concrete semantics for ProbCore reflect a straightforward execution in which each sampling statement $V \leftarrow D$ draws a new value. To represent distributions and sampling, we define distributions as functions from a sufficiently large set of *draws* $\mathcal{S}$. The draws are similar to the seed of a pseudorandom number generator: a sequence Σ of draws dictates the probabilistic behavior of ProbCore programs.

We define a large-step judgment $(H, e) \Downarrow_c v$ for expressions and conditions and a small-step semantics $(\Sigma, H, s) \rightarrow_c (\Sigma', H', s')$ for statements. In the small-step semantics, the heap H consists of the variable-value bindings (queried with $H(v)$) and Σ is the sequence of draws (destructed with $\sigma : \Sigma'$). The result of executing a program is a Boolean declaring whether or not the condition in the passert was satisfied at the end of this particular execution.

The rules for most expressions and statements are standard. The rules for addition and assignment are representative:

$$\textsc{plus}\ \frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 + e_2) \Downarrow_c v_1 + v_2} \qquad \textsc{assign}\ \frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \rightarrow_c (\Sigma, (v \mapsto x) : H, \mathbf{skip})}$$

Figure 2 gives the full set of rules for the concrete semantics. The rule for the sampling statement, $V \leftarrow D$, consumes a draw σ from the head of the sequence Σ. It uses the draw to compute the sample, $d(\sigma)$.

$$\textsc{sample}\ \frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H, \mathbf{skip})}$$

The result of an execution under the concrete semantics is the result of the passert condition after evaluating the program body. We use the standard definition of $\rightarrow_c^*$ as the reflexive, transitive closure of the small step judgment:

$$\textsc{passert}\ \frac{(\Sigma, H_0, s) \rightarrow_c^* (\Sigma', H', \mathbf{skip}) \qquad (H', c) \Downarrow_c b}{(\Sigma, H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_c b}$$

1.3 Symbolic Semantics

While the concrete semantics describe ProbCore program execution, the symbolic semantics describe Mayhap's distribution extraction. Values in the symbolic semantics are expression trees that represent Bayesian networks. The result of a symbolic execution is the expression tree corresponding to the passert condition, as opposed to a Boolean.

PLUS
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 + e_2) \Downarrow_c v_1 + v_2}$$

MULT
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 * e_2) \Downarrow_c v_1 v_2}$$

DIVD
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 \div e_2) \Downarrow_c v_1 / v_2}$$

REAL
$$\frac{}{(H, r) \Downarrow_c r}$$

VARB
$$\frac{}{(H, v) \Downarrow_c H(v)}$$

LT
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 < e_2) \Downarrow_c v_1 < v_2}$$

EQ
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 = e_2) \Downarrow_c v_1 = v_2}$$

AND
$$\frac{(H, c_1) \Downarrow_c b_1 \qquad (H, c_2) \Downarrow_c b_2}{(H, c_1 \wedge c_2) \Downarrow_c b_1 \wedge b_2}$$

OR
$$\frac{(H, c_1) \Downarrow_c b_1 \qquad (H, c_2) \Downarrow_c b_2}{(H, c_1 \vee c_2) \Downarrow_c b_1 \vee b_2}$$

NEG
$$\frac{(H, c) \Downarrow_c b}{(H, \neg c) \Downarrow_c \neg b}$$

ASSIGN
$$\frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \to_c (\Sigma, (v \mapsto x) : H, \mathbf{skip})}$$

SAMPLE
$$\frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \to_c (\Sigma', (v \mapsto d(\sigma)) : H, \mathbf{skip})}$$

PROGN
$$\frac{(\Sigma, H, s_1) \to_c (\Sigma', H', s_1')}{(\Sigma, H, s_1; s_2) \to_c (\Sigma', H', s_1'; s_2)}$$

PROG1
$$\frac{}{(\Sigma, H, \mathbf{skip}; s_2) \to_c (\Sigma, H, s_2)}$$

WHEN
$$\frac{(H, c) \Downarrow_c \mathbf{true}}{(\Sigma, H, \mathbf{if}\ c\ s_1\ s_2) \to_c (\Sigma, H, s_1)}$$

UNLESS
$$\frac{(H, c) \Downarrow_c \mathbf{false}}{(\Sigma, H, \mathbf{if}\ c\ s_1\ s_2) \to_c (\Sigma, H, s_2)}$$

WHILE
$$\frac{}{(\Sigma, H, \mathbf{while}\ c\ s) \to_c (\Sigma, H, \mathbf{if}\ c\ (s; \mathbf{while}\ c\ s)\ \mathbf{skip})}$$

PASSERT
$$\frac{(\Sigma, H_0, s) \to_c^* (\Sigma', H', \mathbf{skip}) \qquad (H', c) \Downarrow_c b}{(\Sigma, H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_c b}$$

Figure 2: The concrete semantics. We use a big-step operational semantics for conditions and expressions, and a small-step operational semantics for statements and programs. Both use a heap H, which stores variable-value bindings. The small-step operational semantics uses a stream Σ of draws.

The language for expression trees includes conditions denoted C_o, real-valued expressions E_o, constants, and distributions:

$$C_o \equiv E_o < E_o \mid E_o = E_o \mid C_o \wedge C_o \mid C_o \vee C_o \mid \neg C_o$$
$$E_o \equiv E_o + E_o \mid E_o * E_o \mid E_o \div E_o \mid R \mid \langle D, E_o \rangle \mid \textbf{if } C_o\ E_o\ E_o$$
$$R \in \mathbb{R},\ D \in \text{Distributions}$$

Instead of the stream of draws Σ used in the concrete semantics, the symbolic semantics tracks a stream offset and the distribution D for every sample. Different branches of an **if** statement can sample a different number of times, so the stream offset may depend on a conditional; thus, the stream offset in $\langle d, n \rangle$ is an expression in E_o and not a simple natural number. The symbolic semantics does not evaluate distributions, so the draws themselves are not required. Expression trees do not contain variables; these are eliminated during distribution extraction.

The symbolic semantics again has big-step rules $\Downarrow_s$ for expressions and conditions and small-step rules $\rightarrow_s$ for statements. Instead of real numbers, however, expressions evaluate to expression trees in E_o and the heap H maps variables to expression trees. For example, the rules for addition and assignment are:

PLUS
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}}$$

ASSIGN
$$\frac{(H, e) \Downarrow_s \{x\}}{(n, H, v := e) \rightarrow_s (n, (v \mapsto \{x\}) : H, \textbf{skip})}$$

The syntax $\{x\}$ represents an expression in E_o, with the brackets intended to suggest quotation or suspended evaluation. Figure 3 lists the full set of rules.

The rule for samples produces an expression tree that captures the distribution and the current stream offset:

SAMPLE
$$\frac{}{(n, H, v \leftarrow d) \rightarrow_s (n + 1, (v \mapsto \{\langle d, n \rangle\}) : H, \textbf{skip})}$$

Each sample statement increments the stream offset, uniquely identifying a sample expression tree. This enumeration is crucial. For example, enumerating samples distinguishes the statement $x \leftarrow d;\ y := x + x$ from a similar program using two samples: $x_1 \leftarrow d;\ x_2 \leftarrow d;\ y := x_1 + x_2$.

The symbolic semantics must consider both sides of an **if** statement. For each **if** statement, we need to merge updates from both branches and form conditional expression trees for conflicting updates. We introduce a function `merge`, which takes two heaps resulting from two branches of an **if** along with the condition and produces a new combined heap. Each variable that does not match across the two input heaps becomes an $\{\textbf{if } c\ a\ b\}$ expression tree in the output heap. The definition of `merge` is straightforward and its post-conditions are:

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a \neq b}{\texttt{merge}(H_t, H_f, \{x\})(v) = \{\textbf{if } x\ a\ b\}} \qquad \frac{H_t(v) = a \qquad H_f(v) = b \qquad a = b}{\texttt{merge}(H_t, H_f, \{x\})(v) = a}$$

PLUS
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}}$$

MULT
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 * e_2) \Downarrow_s \{x_1 * x_2\}}$$

DIVD
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 \div e_2) \Downarrow_s \{x_1 \div x_2\}}$$

REAL
$$\frac{}{(H, r) \Downarrow_s \{r\}}$$

VARB
$$\frac{}{(H, v) \Downarrow_s H(v)}$$

LT
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 < e_2) \Downarrow_s \{x_1 < x_2\}}$$

EQ
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 = e_2) \Downarrow_s \{x_1 = x_2\}}$$

AND
$$\frac{(H, c_1) \Downarrow_s x_1 \qquad (H, c_2) \Downarrow_s x_2}{(H, c_1 \wedge c_2) \Downarrow_s \{x_1 \wedge x_2\}}$$

OR
$$\frac{(H, c_1) \Downarrow_s x_1 \qquad (H, c_2) \Downarrow_s x_2}{(H, c_1 \vee c_2) \Downarrow_s \{x_1 \vee x_2\}}$$

NEG
$$\frac{(H, c) \Downarrow_s x}{(H, \neg c) \Downarrow_s \{\neg x\}}$$

ASSIGN
$$\frac{(H, e) \Downarrow_s \{x\}}{(n, H, v := e) \rightarrow_s (n, (v \mapsto \{x\}) : H, \mathbf{skip})}$$

SAMPLE
$$\frac{}{(\{n\}, H, v \leftarrow d) \rightarrow_s (\{n+1\}, (v \mapsto \{\langle d, n\rangle\}) : H, \mathbf{skip})}$$

PROGN
$$\frac{(n, H, s_1) \rightarrow_s (n', H', s_1')}{(n, H, s_1; s_2) \rightarrow_s (n', H', s_1' \,;\, s_2)}$$

PROG1
$$\frac{}{(n, H, \mathbf{skip}; s_2) \rightarrow_s (n, H, s_2)}$$

IF
$$\frac{(H, c) \Downarrow_s \{x\} \qquad \begin{array}{c}(n, H, b_t) \rightarrow_s^* (m_t, H_t, \mathbf{skip}) \qquad (n, H, b_f) \rightarrow_s^* (m_f, H_f, \mathbf{skip})\end{array}}{(n, H, \mathbf{if}\ c\ b_t\ b_f) \rightarrow_s (\{\mathbf{if}\ x\ m_t\ m_f\})\mathtt{merge}(H_t, H_f, \{x\}), \mathbf{skip})}$$

WHILE
$$\frac{}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{if}\ c\ (\mathbf{while}\ c\ s))}$$

WHILE0
$$\frac{(H, c) \Downarrow_s \{x\} \qquad \forall \Sigma, (\Sigma, \{x\}) \Downarrow_o \mathbf{false}}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{skip})}$$

PASSERT
$$\frac{(0, H_0, s) \rightarrow_s^* (n, H', \mathbf{skip}) \qquad (H', c) \Downarrow_s \{x\}}{(H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_s \{x\}}$$

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a \neq b}{\mathtt{merge}(H_t, H_f, \{x\})(v) = \{\mathbf{if}\ x\ a\ b\}} \qquad \frac{H_t(v) = a \qquad H_f(v) = b \qquad a = b}{\mathtt{merge}(H_t, H_f, \{x\})(v) = a}$$

Figure 3: The symbolic semantics produce an expression tree. We use a big-step style for conditions and expressions, and small-step style for statements. Each big step has the form $(H, e) \Downarrow_s \{s_e\}$ or $(H, c) \Downarrow_s \{s_c\}$, where $e \in E$, $c \in C$, and $s_e \in E_o$, and $s_c \in C_o$. H maps variables to expressions in E_o.

Using the `merge` function, we write the rule for **if** statements:

$$\text{IF}\quad \frac{(H, c) \Downarrow_s \{x\} \qquad (H, b_t) \to_s^* (H_t, \mathbf{skip}) \qquad (H, b_f) \to_s^* (H_f, \mathbf{skip})}{(H, \mathbf{if}\ c\ b_t\ b_f) \to_s (\mathtt{merge}(H_t, H_f, \{x\}), \mathbf{skip})}$$

Our symbolic semantics assumes terminating **while** loops. Symbolic execution of potentially-unbounded loops is a well-known problem and, accordingly, our formalism only handles loops with non-probabilistic conditions. A simple but insufficient rule for **while** is:

$$\text{WHILE}\quad \frac{}{(n, H, \mathbf{while}\ c\ s) \to (n, H, \mathbf{if}\ c\ (\mathbf{while}\ c\ s))}$$

This rule generates infinite expression trees and prevents the analysis from terminating. We would like our analysis to exit a loop if it can prove that the loop condition is false—specifically, when the condition does not depend on any probability distributions. To capture this property, we add the following rule:

$$\text{WHILE0}\quad \frac{(H, c) \Downarrow_s \{x\} \qquad \forall \Sigma, (\Sigma, \{x\}) \Downarrow_o \mathbf{false}}{(n, H, \mathbf{while}\ c\ s) \to (n, H, \mathbf{skip})}$$

Here, the judgment $(\Sigma, \{x\}) \Downarrow_o v$ denotes evaluation of the expression tree $\{x\}$ under the draw sequence Σ. This rule applies when MAYHAP proves that an expression tree evaluates to **false** independent of the random draws.

We can now define the symbolic evaluation of programs:

$$\text{PASSERT}\quad \frac{(0, H_0, s) \to_s^* (n, H', \mathbf{skip}) \qquad (H', c) \Downarrow_s \{x\}}{(H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_s \{x\}}$$

To evaluate the resulting expression tree requires a sequence of draws Σ but no heap. Figure 4 shows the full set of rules. As an example, the rules for addition and sampling are representative:

$$\text{PLUS}\quad \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 + e_2) \Downarrow_o v_1 + v_2} \qquad \text{SAMPLE}\quad \frac{}{(\Sigma, \langle d, k \rangle) \Downarrow_o d(\sigma_k)}$$

2 Theorem and Proof

Theorem 1. *Let $(0, H_0, p) \Downarrow_s \{x\}$, where x is a finite program. Then $(\Sigma, H_0, p) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$.*

Intuitively, this theorem is true because the distribution extraction $\Downarrow_s$ is just a call-by-need lazy evaluation, and $\Downarrow_o$ is the projection of $\Downarrow_c$ over this lazy evaluation. We prove the theorem formally here.

The proof of this theorem proceeds by structural induction on p. First, a few lemmas establish corresponding properties for conditionals, expressions, then statements, and finally programs.

Lemma 1. *For $e \in E$, let $(H_s, e) \Downarrow_s \{x\}$, and suppose that for every variable a, $(\Sigma, H_s(a)) \Downarrow_e H_c(a)$. Then $(H_c, e) \Downarrow_c v$ if and only if $(\Sigma, x) \Downarrow_o v$.*

Proof. The proof is by induction on e. The condition on H_s and H_c is necessary because H_s maps variables to expressions in E_o, while H_c maps variables to real numbers. Note that Σ is unbound; this is because, while Σ is necessary for sampling distributions in E_o, expressions in E do not involve sampling. We examine each of five cases individually.

$e_1 + e_2$ Let $(H_s, e_1) \Downarrow_s \{x_1\}$ and $(H_s, e_2) \Downarrow_s \{x_2\}$. Also let $(H_c, e_1) \Downarrow_c v_1$ and $(H_c, e_2) \Downarrow_c v_2$, so that $(H_c, e_1 + e_2) \Downarrow_c v_1 + v_2 = v$. By the definition of $\Downarrow_s$, $(H_s, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}$, and by induction $(\Sigma, x_1) \Downarrow_o v_1$ and $(\Sigma, x_2) \Downarrow_o v_2$. Then by the definition of $\Downarrow_o$, $(\Sigma, x) = (\Sigma, x_1 + x_2) \Downarrow_o v_1 + v_2 = v$. Thus this case is established.

$e_1 * e_2$ Analogous to $e_1 + e_2$.

$e_1 \div e_2$ Analogous to $e_1 + e_2$.

$e_1 \div e_2$ Analogous to $e_1 + e_2$.

r $(H_s, r) \Downarrow_s \{r\}$ and $(\Sigma, r) \Downarrow_c r$; on the other hand, $(H_c, r) \Downarrow_c r$. Thus this case is established.

v $(H_s, v) \Downarrow_s H_s(v)$, while $(H_c, v) \Downarrow_c H_c(v)$. But by hypothesis, $(\Sigma, H_s(v)) \Downarrow_e H_c(a)$, so this case is established.

These are all the cases present in the definition of E, so the lemma is complete.

□

Lemma 2. *For $c \in C$, let $(H_s, c) \Downarrow_s \{x\}$, and suppose that for every variable a, $(\Sigma, H_s(a)) \Downarrow_e H_c(a)$. Then $(H_c, c) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$.*

Proof. We again use induction, on c. We examine each of five cases individually.

$e_1 < e_2$ By the definition of $\Downarrow_s$, $\{x\} = \{x_1 + x_2\}$. Let $(H_c, e_1) \Downarrow_c v_1$ and $(H_c, e_2) \Downarrow_c v_2$, so that $b = [v_1 < v_2]$. By lemma 1, $(\Sigma, x_1) \Downarrow_o v_1$ and $(\Sigma, x_2) \Downarrow_o v_2$, so $(\Sigma, x) \Downarrow_o [v_1 < v_2] = b$. Thus this case is established.

$e_1 = e_2$ Analogous to $e_1 < e_2$.

$c_1 \wedge c_2$ Let $(H_s, c_1) \Downarrow_s \{x_1\}$ and $(H_s, c_2) \Downarrow_s \{x_2\}$. Also let $(H_c, c_1) \Downarrow_c b_1$ and $(H_c, c_2) \Downarrow_c b_2$, so that $(H_c, c_1 \wedge c_2) \Downarrow_c b_1 \wedge b_2 = v$. By the definition of $\Downarrow_s$, $(H_s, c_1 \wedge c_2) \Downarrow_s \{x_1 \wedge x_2\}$, and by induction $(\Sigma, x_1) \Downarrow_o b_1$ and $(\Sigma, x_2) \Downarrow_o b_2$. Then by the definition of $\Downarrow_o$, $(\Sigma, x) = (\Sigma, x_1 \wedge x_2) \Downarrow_o b_1 + b_2 = b$. Thus this case is established.

$c_1 \vee c_2$ Analogous to $c_1 \wedge c_2$.

$\neg c_1$ Let $(H_s, c_1) \Downarrow_s \{x_1\}$ and $(H_c, c_1) \Downarrow_c b_1$, so that $(H_c, \neg c_1) \Downarrow_c \neg b_1$. By the definition of $\Downarrow_s$, $(H_s, \neg c_1) \Downarrow_s \{\neg x_1\}$, and by induction $(\Sigma, x_1) \Downarrow_o b_1$, so that $(\Sigma, x) \Downarrow_o \neg b_1 = b$. Thus this case is established.

These are all the cases present in the definition of C, so the lemma is complete.

□

We now prove a lemma which establishes equivalence for statements that do not contain **while** loops.

Lemma 3. *Let $(n, H_s, s) \to_s (m, H'_s, s')$, where s contains no **while** statements. Also suppose that $(\Sigma, n) \Downarrow_o l$ and $(\Sigma, m) \Downarrow_o l + k$. Furthermore let H_c be such that $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$ for all variables v. Then $(\Sigma, H_c, s) \to_c^* (\Sigma', H'_c, s')$, where $\Sigma = \sigma_1 : \sigma_2 : \cdots : \sigma_k : \Sigma'$. Furthermore, $(\Sigma, H'_s(v)) \Downarrow_o H'_c(v)$ for all v.*

Proof. A few difficulties arise when attempting a naive induction:

- While $\Downarrow_c$ and $\to_c$ consume an element of Σ, $\Downarrow_s$ and $\to_s$ simply increment an offset. Our induction must show that this offset is correctly handled.
- While $\to_c$ only evaluates one side of an **if** statement, $\to_s$ evaluates both. Proving that this is sound requires proving that the "merge" function correctly unifies the two branches.
- Non-terminating while loops, especially those involving sampling, are difficult to handle in the induction. The statement of the lemma guarantees that the while loop must terminate (since $\to_s^*$ requires a finite number of steps), but the possibility for while loops to not terminate still complicates the proof.

The first problem is avoided by the statement of the lemma: we require that the symbolic semantics increment the sequence offset by exactly as many elements as the concrete semantics consumes. The second problem requires a careful analysis of the "merge" function. This is also why we assume a single step in $\to_s$ but a number of steps in $\to_c^*$. Finally, the last problem is avoided by a nested induction over the number of times the **while** loop is unrolled. Since we assume the symbolic semantics terminate, the loop must at some point unroll fully, so the induction is founded.

As mentioned, we induct over the number of steps taken by $\to_s^*$. At each step, we assume that the future steps will satisfy the statement of the lemma. We consider each case individually.

$v := e$ Assume that $(H_c, e) \Downarrow_c x_c$, so that $(\Sigma, H_c, v := e) \to_c (\Sigma, (v \mapsto x) : H_c, \mathbf{skip})$. Furthermore, suppose $(H_s, e) \Downarrow_s \{x_s\}$, so that $(n, H_s, v := e) \to_s (n, (v \mapsto x_s) : H_s, \mathbf{skip})$. By lemma 1, $(\Sigma, x_s) \Downarrow_o x_s$. But then, for all variables v, we have $(\Sigma, ((v \mapsto x_s) : H_s)(v')) \Downarrow_o ((v \mapsto x_c) : H_c)(v')$ for all v'. If we set $\Sigma' = \Sigma$ and $k = 0$, we find that in this case our theorem is proven.

$v \leftarrow d$ Let $\Sigma = \sigma : \Sigma'$. Then $(\Sigma, H_c, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H_c, \textbf{skip})$ On the other hand, in the symbolic semantics, $(\{n\}, H_s, v \leftarrow d) \rightarrow_s (\{n+1\}, (v \mapsto \langle d, n\rangle) : H_s, \textbf{skip})$ We can see that if $(\Sigma, \{n\}) \Downarrow_o l$, then $(\Sigma, \{n+1\}) \Downarrow_o l + 1$, forcing $k = 1$. Indeed, $\Sigma = \sigma_1 : \Sigma'$. Furthermore, since $(\Sigma, \langle d, n\rangle) \Downarrow_o d(\sigma_1) = d(\sigma)$, we know that for all v', $(\Sigma, ((v \mapsto \langle d, n\rangle) : H_s)(v')) \Downarrow_o ((v \mapsto d(\sigma)) : H_c)(v')$. So this case is established.

skip Since there are no symbolic steps for **skip**, the lemma is vacuously true.

$s_1; s_2$ This statement has two cases: where s_1 is **skip**, and where it is not. If s_1 is **skip**, the case is trivial, so suppose s_1 is not **skip**. Furthermore, let $(n, H_s, s_1) \rightarrow_s (m', H_s', s_1')$. By induction, we also have $(\Sigma, H_c, s_1) \rightarrow_c^* (\Sigma'', H_c', s_1')$, with the expected properties relating Σ' and k, and H_c' and H_s'. But then since $(n, H_s, s_1; s_2) \rightarrow_s (m', H_s', s_1'; s_2)$ and $(\Sigma, H_c, s_1; s_2) \rightarrow_c^* (\Sigma'', H_c', s_1'; s_2)$, this case is established with $m = m'$ and $\Sigma' = \Sigma''$. (We omit the lemma that $s_1 \rightarrow^* s_1'$ implies $s_1; s_2 \rightarrow^* s_1'; s_2$, with the expected behavior of the other parameters.)

if $c\ s_1\ s_2$ First, consider that per lemma 2, we know that if $(H_c, c) \Downarrow_c b$, and $(H_s, c) \Downarrow_s \{x_s\}$, then $(\Sigma, x_s) \Downarrow_o b$. Now consider two sub-cases: b is true, and b is false. If b is true, then for all expressions y_t and y_f, the expression **if** $x_s\ y_t\ y_f$ must evaluate to the same result as y_t; otherwise if b is false, to the same result as y_f.

Now, depending on b, either $(\Sigma, H_c, \textbf{if} c s_1 s_2) \rightarrow (\Sigma', H_c', s_1)$ or $(\Sigma, H_c, \textbf{if} c s_1 s_2) \rightarrow (\Sigma', H_c', s_2)$. We know that $(n, H_s, s_1) \rightarrow_s^* (m_t, H_{st}, \textbf{skip})$ and $(n, H_s, s_2) \rightarrow_s^* (m_f, H_{sf}, \textbf{skip})$. But then by induction, we know that $(\Sigma, H_c, s_1) \rightarrow_c^* (\Sigma_t, H_{ct}, \textbf{skip})$ or $(\Sigma, H_c, s_2) \rightarrow_c^* (\Sigma_f, H_{cf}, \textbf{skip})$, where the relationship of Σ_t to m_t, of Σ_f to m_f, of H_{ct} to H_{st}, and of H_{cf} to H_{sf} are as expected. Thus, when $(n, H_s, \textbf{if}\ c\ s_1\ s_2) \rightarrow (m, H_s', \textbf{skip})$, we know that $(\Sigma, H_c, \textbf{if}\ c\ s_1\ s_2) \rightarrow (\Sigma', H_c', \textbf{skip})$ as required, where Σ' is Σ_t or Σ_f depending on the condition, and where H_c' is H_{ct} or H_{cf}, again depending on the loop condition.

All that remains to prove is that the symbolic inference rule for the **if** rule correctly combines H_{st} and H_{sf}, and likewise correctly combines m_t and m_f. Recall that b is the value of the loop condition, and the loop conditional evaluates symbolically to x_s. We do a case analysis on b. First, suppose b is true. Then $\Sigma' = \Sigma_t$, so we know that $\Sigma' = \sigma_1 : \cdots \sigma_k : \Sigma'$ where $(\Sigma, m) = (\Sigma, \textbf{if}\, x_s\, m_t\, m_f) \Downarrow_o k$. Similarly, since $H_s' = \texttt{merge}(H_{sf}, H_{st}, x_s)$, we know that for all variable v, $(\Sigma, H_s'(v)) = (\Sigma, \texttt{merge}(H_{st}, H_{sf}, x_s)(v))$ This is equal to either $(\Sigma, \textbf{if}\ x_s\ (H_{st}(v))\ (H_{sf}(v)))$ or $(\Sigma, H_{st}(v))$, both of which evaluate to $H_{ct}(v) = H_c'(v)$ because x_s evaluates to b which is true, and because $(\Sigma, H_{st}(v)) = H_{ct}(v)$ by induction. The case where b is false is analogous to the case where b is true.

Thus this case is established.

This was the last remaining case (we assume that s_1 contains no **while** statements), so the lemma is done. □

We now extend the equivalence to programs that contain while loops. We require that the symbolic evaluation terminate.

Lemma 4. *Let* $(n, H_s, s) \to_s^* (m, H'_s, \mathbf{skip})$. *Further suppose that for all variables* v, $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$. *Then* $(\Sigma, H_c, s) \to_c^* (\Sigma', H'_c, \mathbf{skip})$, *and furthermore for all variables* v, $(\Sigma, H'_s(v)) = H'_c(v)$ *and also* $\Sigma = \sigma_1 : \cdots : \sigma_k : \Sigma'$, *where* $(\Sigma, m) \Downarrow_o l + k$ *(where* $(\Sigma, n) \Downarrow_o l$*).*

Proof. We proceed by structural induction on s.

$v := e$ There are no **while** loops in this statement, so it follows from lemma 3.

$v \leftarrow d$ Analogous to $v := e$.

skip Analogous to $v := e$.

$s_1; s_2$ We must have $(n, H_s, s_1) \to_s^* (n', H''_s, \mathbf{skip})$, so by induction $(\Sigma, H_c, s_1) \to_c^* (\Sigma'', H''_c, \mathbf{skip})$, with the usual relation between Σ'' and n', and between H''_s and H''_c. But then again by induction $(n', H''_s, s_2) \to_s^* (m, H'_s, \mathbf{skip})$ implies $(\Sigma'', H''_c, s_2) \to_c^* (\Sigma', H'_c, \mathbf{skip})$. Thus, $(\Sigma, H_c, s_1; s_2) \to_c^* (\Sigma', H'_c, \mathbf{skip})$, and this case is established.

if $c\ s_1\ s_2$ If $(n, H_s, \mathbf{if} c s_1 s_2) \to^* (n', H'_s, \mathbf{skip})$, we must have $(n, H_s, s_1) \to^* (n_t, H'_{st}, \mathbf{skip})$ and $(n, H_s, s_2) \to^* (n_f, H'_{sf}, \mathbf{skip})$. Then, analogously to the argument in lemma 3, this case can be established.

while $c\ s$ There are two inference rules concerning the symbolic semantics of **while** loops, so we must prove that both are sound.

First consider the rule WHILE0. If it applies, we must have $(\Sigma, x) \Downarrow_o$ **false** for $(H_s, c) \Downarrow_s \{x\}$, and thus (by lemma 2) $(H_c, c) \Downarrow_c$ **false**. Then $(\Sigma, H_c, \mathbf{while} c s) \to^* (\Sigma, H_c, \mathbf{skip})$. But by assumption, $(n, H_s, \mathbf{while} c s) \to (n, H_s, \mathbf{skip})$, so the inductive statement holds.

Second, consider the rule WHILE in the symbolic semantics. It is identical to the corresponding rule for **while**, so by induction this case is established.

These are all the cases for S, so the lemma is proven.

□

Finally, we can prove our Theorem 1.

Theorem 2. *Let* $(0, H_0, p) \Downarrow_s \{x\}$, *where* x *is a finite program. Then* $(\Sigma, H_0, p) \Downarrow_c b$ *if and only if* $(\Sigma, x) \Downarrow_o b$.

Proof. First, note that $H_c = H_s = H_0$, so that $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$ for all v (by the rule for constants).

Let the program p be s ; ; $\mathsf{passert}\ c$. If $(0, H_0, p) \Downarrow_s \{x\}$, then $(0, H_0, s) \to_s^* (n, H_s, \mathbf{skip})$. Then by lemma 4, $(\Sigma, H_0, s) \to_s^* (\Sigma', H_c, \mathbf{skip}$, with the expected relation between H_c and H_s. But then due to this relation, if $(H_s, c) \Downarrow_s \{y\}$,

$(H_c, c) \Downarrow_c b$ if and only if $(\Sigma, y) \Downarrow_o b$ (the lemma to prove this would be a straightforward induction over y).

Thus, $(\Sigma, H_0, p) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$, and our theorem is proven. $\square$

While complex, this theorem shows that the distribution extraction performed by MAYHAP is sound.

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 + e_2) \Downarrow_o v_1 + v_2} \qquad \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 * e_2) \Downarrow_o v_1 * v_2}$$

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 \div e_2) \Downarrow_o v_1 \div v_2} \qquad \frac{}{(\Sigma, r) \Downarrow_o r} \qquad \frac{(\Sigma, n) \Downarrow_o k}{(\Sigma, \langle d, n \rangle) \Downarrow_o d(\sigma_k)}$$

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 < e_2) \Downarrow_o v_1 < v_2} \qquad \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 = e_2) \Downarrow_o v_1 = v_2}$$

$$\frac{(\Sigma, c_1) \Downarrow_o b_1 \qquad (\Sigma, c_2) \Downarrow_o b_2}{(\Sigma, c_1 \wedge c_2) \Downarrow_o b_1 \wedge b_2} \qquad \frac{(\Sigma, c_1) \Downarrow_o b_1 \qquad (\Sigma, c_2) \Downarrow_o b_2}{(\Sigma, c_1 \vee c_2) \Downarrow_o b_1 \vee b_2}$$

$$\frac{(\Sigma, c) \Downarrow_o b}{(\Sigma, \neg c) \Downarrow_o \neg b} \qquad \frac{(\Sigma, c) \Downarrow_o \textbf{true} \qquad (\Sigma, e_1) \Downarrow_o v}{(\Sigma, \textbf{if } c\, e_1\, e_2) \Downarrow_o v}$$

$$\frac{(\Sigma, c) \Downarrow_o \textbf{false} \qquad (\Sigma, e_2) \Downarrow_o v}{(\Sigma, \textbf{if } c\, e_1\, e_2) \Downarrow_o v}$$

Figure 4: The semantics for our simple expression language. Σ is a stream of draws, and σ_k is the k-th element of Σ.