Reasoning about Recursive Probabilistic Programs

Federico Olmedo  Benjamin Lucien Kaminski  Joost-Pieter Katoen  Christoph Matheja

RWTH Aachen University, Germany
{federico.olmedo, benjamin.kaminski, katoen, matheja}@cs.rwth-aachen.de

Abstract
This paper presents a wp-style calculus for obtaining expectations on the outcomes of (mutually) recursive probabilistic programs. We provide several proof rules to derive one- and two-sided bounds for such expectations, and show the soundness of our wp-calculus with respect to a probabilistic pushdown automaton semantics. We also give a wp-style calculus for obtaining bounds on the expected runtime of recursive programs that can be used to determine the (possibly infinite) time until termination of such programs.

Categories and Subject Descriptors F.3.1 [Logics and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs.

Keywords recursion  probabilistic programming  program verification  weakest pre-condition calculus  expected runtime.

1. Introduction
Uncertainty is nowadays more and more pervasive in computer science. Applications have to process inexact data from, e.g., unreliable sources such as wireless sensors, machine learning methods, or noisy biochemical reactors. Approximate computing saves reliable sources such as wireless sensors, machine learning methods, or noisy biochemical reactors. Approximate computing saves unreliable hardware, circuits that every now and then (deliberately) produce incorrect results [4]. Probabilistic programming [28] is a key technique for dealing with uncertainty. Put in a nutshell, a probabilistic program takes a (prior) probability distribution as input and obtains a (posterior) distribution. Probabilistic programs are not new at all; they have been investigated by Kozen [20] and others in the early eighties. In the last years, the interest in these programs has rapidly grown. In particular, the incentive by the AI community to use probabilistic programs for describing complex Bayesian networks has boosted the field of probabilistic programming [10]. Probabilistic programs are used in, amongst others, machine learning, systems biology, security, planning and control, quantum computing, and software-defined networks. Indeed almost all programming languages, either being functional, object-oriented, logical, or imperative, in the meanwhile have a probabilistic variant.

This paper focuses on recursive probabilistic programs. Recursion in Bayesian networks where a variable associated with a particular domain entity can depend probabilistically on the same variable associated to a different entity, is “common and natural” [29]. Recursive probability models occur in gene regulatory networks where a variable associated with a particular domain entity can depend probabilistically on the same variable. The first calculus is an extension of McIver and Mor-

[Copyright notice will appear here once ‘preprint’ option is removed.]
gan’s calculus [23] for non–recursive programs and enables obtaining expectations on the outcomes of (mutually) recursive probabilistic programs. Compared to an existing extension with recursion [22], our approach provides a clear separation between syntax and semantics. We prove the soundness of our wp–calculus with respect to a probabilistic pushdown automaton semantics. This is complemented by a set of proof rules to derive one– and two–sided bounds for expected outcomes of recursive programs. We illustrate the usage of these proof rules by analyzing the termination probability of the example program above. Subsequently, we provide a variant of our wp–style calculus for obtaining bounds on the expected runtime of probabilistic programs. This extends our recent approach [17] towards treating recursive programs. The application of this calculus includes proving positive almost–sure termination, i.e., does a program terminate with probability one in finite expected runtime of this calculus includes proving positive almost–sure termination behavior: If in established a (well–known) relationship between the expected run–

tion, i.e., does a program terminate with probability one in finite run–

tion. This extends our recent approach [17] towards treating recursive programs. The application of this calculus includes proving positive almost–sure termination, i.e., does a program terminate with probability one in finite run–
tion. We provide a set of proof rules for expected runtimes and show the applicability of our approach by proving several correctness properties as well as the expected runtime of the ‘Sherwood’ variant of binary search.

Organization of the paper. Section 2 presents our probabilistic programming language with recursion. Section 3 presents the wp–style semantics for reasoning about program correctness. Section 4 introduces several proof rules for reasoning about the correctness of recursive programs. Section 5 presents the expected runtime transformer together with proof rules for recursive programs. Section 6 describes an operational probabilistic pushdown automata semantics and relates it to the wp–style semantics. Section 7 discusses some extensions of the results presented in the previous sections. Section 8 presents a detailed analysis of the ‘Sherwood’ variant of binary search. Finally, Section 9 discusses related work and Section 10 concludes. Detailed proofs are provided in the appendix, which is added for the convenience of the reviewer, and will not be part of the final version (if accepted).

2. Programming Model

To model our probabilistic recursive programs we consider a simple imperative language à la Dijkstra’s Guarded Command Language (GCL) [7] with two additional features: First, a (binary) probabilistic choice operator to endow our programs with a probabilistic behavior. For instance, the program

\[ \{ x := x+1 \} \frac{1}{3} \{ x := x-1 \} \]

either increases \( x \) with probability \( \frac{1}{3} \) or decreases it with probability \( \frac{2}{3} \). Second, we allow for procedure calls. For simplicity, our development assumes the presence of only a single procedure, say \( P \). We defer the treatment of multiple (possibly mutually recursive) procedures to Section 7.

Formally, a command of our language, coined pRGCL, is defined by the following grammar:

\[
C ::= \text{skip} \quad \text{no–op} \\
V := E \quad \text{assignment} \\
\text{abort} \quad \text{abortion} \\
\text{if} (E) \{ C \} \text{ else } \{ C \} \quad \text{conditional branching} \\
\{ C \} [p] \{ C \} \quad \text{probabilistic choice} \\
\text{call } P \quad \text{procedure call} \\
C; C \quad \text{sequential composition}
\]

We assume a set \( V \) of program variables and a set \( E \) of expressions over program variables. As usual, we assume that program states are variable valuations, i.e. mappings from variables to values; let \( S \) be the set of program states. Finally, we also assume an interpretation function \( \llbracket \cdot \rrbracket \) for expressions that maps program states to values.

No–op, assignments, conditionals and sequential composition are standard. \( \{ C \} [p] \{ C \} \) represents a probabilistic choice: it behaves as \( C_1 \) with probability \( p \) and as \( C_2 \) with probability \( 1−p \). Finally call \( P \) makes a (possibly recursive) call to procedure \( P \).

For our development we assume that procedure \( P \) manipulates the global program state and we thus dispense with parameters and return statements for passing information across procedure calls. The declaration of \( P \) consists then of its body and we use \( P \triangleright c \) to denote that \( c \in C \) is the body of \( P \). We say that a command is closed if it contains no procedure calls.

A pRGCL program is then given by a pair \( (c, \triangleright) \), where \( c \in C \) is the “main” command and \( \triangleright : \{ P \} \to C \) is the declaration of \( P \).

In order not to clutter the notation, when \( c \) is closed we simply write \( c \) for program \( (c, \triangleright) \), for any declaration \( \triangleright \).

Example 1. To illustrate the use of our language consider the following declaration of a (faulty) recursive procedure for computing the factorial of a natural number stored in \( x \):

\[
P_{\text{fact}} \triangleright \begin{cases}
\text{if } (x \leq 0) \{ y := 1 \} & \\
\{ x := x−1; \text{ call } P_{\text{fact}}; x := x+1 \} & \left[ \frac{9}{6} \right] \\
\{ x := x−2; \text{ call } P_{\text{fact}}; x := x+2 \}; y := y \cdot x &
\end{cases}
\]

In each recursive call \( x \) is decreased either by one or two, with probability \( \frac{9}{6} \) and \( \frac{1}{6} \), respectively. Therefore some factors might be missing in the computation of the factorial of \( x \).

As a final remark, observe that the language does not support guarded loops in a native way because they can be simulated. Concretely, the usual guarded loop while \( (E) \) do \{ \( c \) \} is simulated by the recursive procedure \( P_{\text{while}} \triangleright \begin{cases}
\text{if } (E) \{ c; \text{ call } P_{\text{while}} \} & \\
\text{else } (\text{skip}) &
\end{cases}
\)

3. Weakest Pre–Expectation Semantics

Inspired by Kozen [20], McIver and Morgan [22] generalized Dijkstra’s weakest pre–condition semantics to (a variant of) pRGCL. In particular, they defined the semantics of recursive programs using fixed point techniques. In this section we present a different approach where the behavior of a recursive program is defined as the limit of its finite approximations (or truncations) and prove it equivalent to their definition based on fixed points.

3.1 Definition

The wp-semantics over pRGCL generalizes Dijkstra’s weakest pre-condition semantics over GCL twofold: First, instead of being predicates over program states, pre– and post–conditions are now (non-negative) real–valued functions over program states. Secondly, instead of merely evaluating a (boolean–valued) post–condition in the final state(s) of a program, we now measure the expected value of a (real–valued) post–condition w.r.t. the distribution of final states.

Formally, if \( f : S \to \mathbb{R}^\geq \) we let

\[
\text{wp}[c, \triangleright](f) \triangleq \lambda s. E_{c, \triangleright}(s)(f)
\]

where \( [c, \triangleright](s) \) denotes the distribution of final states from executing \( (c, \triangleright) \) in initial state \( s \) and \( E_{c, \triangleright}(s)(f) \) denotes the expected value of \( f \) w.r.t. the distribution of final states \( [c, \triangleright](s) \). Consider for instance program

\[
c_{\text{coins}} : \begin{cases}
\{ x := 0 \} \left[ \frac{1}{2} \right] \{ y := 1 \} & \\
\{ y := 0 \} \left[ \frac{1}{3} \right] \{ y := 1 \} &
\end{cases}
\]

that flips a pair of fair and biased coins. We have

\[
\text{wp}[c_{\text{coins}}](f) = \lambda s. \frac{1}{2} f(s[x,y/0,0]) + \frac{1}{3} f(s[x,y/0,1])
\]

\(^{1}\)We chose the declaration of \( P \) to be a mapping from a singleton and not the mere body of \( P \) because this minimizes the changes to accommodate the subsequent treatment to multiple procedures.
\[ + \frac{1}{3} f(s[x, y/1, 0]) + \frac{1}{3} f(s[x, y/1, 1]) \]

where \( s[x_1, \ldots, x_n, v_1, \ldots, v_n] \) represents the state obtained by updating in \( s \) the value of variables \( x_1, \ldots, x_n \) to \( v_1, \ldots, v_n \), respectively. As above, when \( c \) is closed, we usually write \( \text{wp}[c] \) instead of \( \text{wp}[c, \mathcal{D}] \), as a declaration \( \mathcal{D} \) plays no role.

Observe that, in particular, if \( [A] \) denotes the indicator function of a predicate \( A \) over program states, \( \text{wp}[c, \mathcal{D}](\{A\}) \) gives the probability of (terminating and) establishing \( A \) after executing \( (c, \mathcal{D}) \) from state \( s \). For instance, we can determine the probability that the above program \( \text{coins} \) establishes \( x = y \) from state \( s \) through

\[ \text{wp}[\text{coins}](\{x = y\}) = \frac{1}{2} + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{5}{6} \]

Moreover, for a deterministic program \( c \) that from state \( s \) terminates in state \( s' \), \( [c, \mathcal{D}](s) \) is the Dirac distribution that concentrates all its mass in \( s' \) and \( \text{wp}[c, \mathcal{D}](\{A\}) \) reduces to \( 1\cdot[A](s') \), which is closed, and 0 otherwise. This yields the classical weakest pre-conditions semantics of ordinary sequential programs.

To reason about partial program correctness, PRGCL also admits a liberal version of the transformer \( \text{wp}[\cdot \cdot \cdot] \), namely \( \text{wp}[\cdot \cdot \cdot] \). In the same vein as for ordinary sequential programs, \( \text{wp}[c, \mathcal{D}](\{A\}) \) gives the probability that program \( (c, \mathcal{D}) \) terminates and establishes event \( A \) from state \( s \), while \( \text{wp}[c, \mathcal{D}](\{A\}) \) gives the probability that \( (c, \mathcal{D}) \) terminates and establishes \( A \), or diverges.

Formally, the transformer \( \text{wp} \) operates on unbounded, so-called expectations in \( \mathbb{E} \triangleq \{ f \mid f : \mathcal{S} \to [0, \infty] \} \), while the transformer \( \text{wp} \) operates on bounded expectations in \( \mathbb{E}_{<1} \triangleq \{ f \mid f : \mathcal{S} \to [0, 1] \} \). Our expectation transformers thus have type \( \text{wp} : \mathbb{E} \to \mathbb{E} \) and \( \text{wp} : \mathbb{E}_{<1} \to \mathbb{E}_{<1} \). In the probabilistic setting pre- and post-conditions are thus referred to as pre- and post-expectations.

**Notation.** We use boldface for constant expectations, e.g. \( \mathbf{1} \) denotes the constant expectation \( \lambda s. 1 \). Given an arithmetical expression \( E \) over program variables we write \( E \) for the expectation that in states \( s \) returns \( [E](s) \). Given a Boolean expression \( G \) over program variables let \( [G] \) denote the \( \{ 0, 1 \} \)-valued expectation that on state \( s \) returns 1 if \( [G](s) = \text{true} \) and 0 if \( [G](s) = \text{false} \). Finally, given variable \( x \), expression \( E \) and expectation \( f \) we use \( f[x/E] \) to denote the expectation that on state \( s \) returns \( f(s[x/E](s)) \).

Moreover, \( \preceq \) denotes the pointwise order between expectations, i.e. \( f_1 \preceq f_2 \) iff \( f_1(s) \leq f_2(s) \) for all states \( s \in \mathcal{S} \).

### 3.2 Inductive Characterization

McIver and Morgan [22] showed that the expectation transformers \( \text{wp} \) and \( \text{wp} \) can be defined by induction on the program’s structure. We now recall their result, taking an alternative approach to handle recursion: While McIver and Morgan use fixed point techniques, we follow e.g. Hehner [12] and define the semantics of a recursive procedure as the limit of an approximation sequence. We believe that this approach is sometimes more intuitive and closer to the operational view of programs.

In the same way as the semantics of loops is defined as the limit of their finite unrollings, we define the semantics of recursive procedures as the limit of their finite inlinings. Formally, the \( n \)-th inlining call\( \mathcal{P} \) of procedure \( P \) w.r.t. declaration \( \mathcal{D} \) is defined inductively by

\[
\begin{align*}
\text{call}_{0}^{P} & \quad = \text{abort} \\
\text{call}_{n+1}^{P} & \quad = \mathcal{D}(P) \text{call } P/\text{call}_{n}^{P} \\
\end{align*}
\]

where \( c(\text{call } P/c') \) denotes the syntactic replacement of every occurrence of \( \text{call } P \) in \( c \) by \( c' \). The family of commands call\( \mathcal{P} \) defines a sequence of approximations to call \( P \) where call\( \mathcal{P} \) is the “poorest” approximation, while the larger the \( n \), the more precise the approximation becomes. Observe that, in general, call\( \mathcal{P} \) mimics the exact behavior of call \( P \) for all executions that finish after at most \( n \) recursive calls.

The expectation transformer semantics over PRGCL is provided in Figure 1. The action of transformers on procedure calls is defined as the limit of their action over the \( n \)-th inlining of the procedures. For the rest of the language constructs, we follow McIver and Morgan [22]. Let us briefly explain each of the rules. \( \text{wp}[\text{skip}, \mathcal{D}] \) behaves as the identity since skip has no effect. The pre- and post-expectation of an assignment is obtained by updating the program state and then applying the post-expectation, i.e. \( \text{wp}[x := E, \mathcal{D}] \) takes post-expectation \( f \) to pre-expectation \( f[x/E] = \lambda s. f(s[x/E](s)) \) wp[abort, \mathcal{D}] maps any post-expectation to the constant pre-expectation \( \text{wp}[\text{skip}, \mathcal{D}] \) is the probabilistic counterpart of predicate false. \( \text{wp}[\text{false}, \mathcal{D}] \) behaves either as \( \text{wp}[\text{call } c_1, \mathcal{D}] \) if \( \text{wp}[c_2, \mathcal{D}] \) is provided as a convex combination of \( \text{wp}[\text{call } c_1, \mathcal{D}] \) and \( \text{wp}[\text{call } c_2, \mathcal{D}] \), weighted according to \( p \). wp[call \( P, \mathcal{D} \) behaves as the limit of wp on the sequence of finite truncations (or inlinings) of \( P \). We take the supremum because the sequence is increasing. Observe that we adverently include no declaration in \( \text{wp}[\text{call } P^{n}, \mathcal{D}] \) because \( \text{call } P^{n} \) is a closed command for every \( n \). Finally, \( \text{wp}[\text{call } c_1, \mathcal{D}] \) is obtained as the functional composition of \( \text{wp}[\text{call } c_1, \mathcal{D}] \) and \( \text{wp}[\text{call } c_2, \mathcal{D}] \). The wp transformer follows the same rules as wp, except for the abort statement and procedure calls. wp[abort, \mathcal{D}] takes any post-expectation to pre-expectation 1. (Expectation 1 is the probabilistic counterpart of predicate true.) wp[\text{call } P, \mathcal{D}] also behaves as the limit of wp on the sequence of finite truncations of \( P \). This time we take the infimum because the sequence is decreasing.

**Example 2.** Reconsider \( \text{coins} = \text{call } c_1 \cup \text{call } c_2 \) from Section 3.1 with \( c_1 : \{ x : 0 \} \cup \{ x : 1 \} \) and \( c_2 : \{ y : 0 \} \cup \{ y : 1 \} \). We use our weakest pre-expectation calculus to formally determine the probability that the outcome of the two coins coincide:

\[
\begin{align*}
\text{wp}[\text{coins}](\{x=y\}) & = \text{wp}[c_1](\text{wp}[c_2](\{x=y\})) \\
& = \text{wp}[c_1](\frac{1}{3} \cdot \text{wp}[y = 0](\{x=y\}) + \frac{2}{3} \cdot \text{wp}[y = 1](\{x=y\})) \\
& = \text{wp}[c_1](\frac{1}{3} \cdot \frac{1}{4} \cdot \frac{1}{4} + \frac{2}{3} \cdot \frac{3}{4}) \\
& = \frac{1}{2} + \text{wp}[x := 0](\frac{1}{4} \cdot \frac{1}{4} + \frac{2}{3} \cdot \frac{3}{4})
\end{align*}
\]
\[
\frac{1}{2} \cdot \wp[x \leftarrow 1] (\frac{1}{2} \cdot [x=0] + \frac{3}{4} \cdot [x=1]) = \frac{1}{2} \cdot (\frac{1}{2} \cdot [x=0] + \frac{1}{2} \cdot [x=1]) = \frac{1}{2} \cdot \delta(x) + \frac{1}{2} \cdot \delta(x) = \frac{1}{2} \\
\]

The transformers \(\wp\) and \(\wlp\) enjoy several appealing algebraic properties, which we summarize below.

**Lemma 3.1** (Basic properties of \(\wp(\cdot,p)\)). For every program \((c, \theta)\), every \(f_1, f_2, \text{ and increasing } \omega\)-chain \(f_0 \preceq f_1 \preceq \cdots \in \mathbb{E}, g_1, g_2, \text{ and every decreasing } \omega\)-chain \(g_0 \succeq g_1 \succeq \cdots \in \mathbb{E}_{\leq 1},\) and scalars \(\alpha_1, \alpha_2 \in \mathbb{R}_{\geq 0}\) it holds:

- **Continuity:** \(\sup_n \wp[c, \theta](f_n) = \wp[c, \theta](\sup_n f_n)\)
- **Infinitary:** \(\inf_n \wp[c, \theta](g_n) = \wp[c, \theta](\inf_n g_n)\)
- **Monotonicity:** \(f_1 \preceq f_2 \implies \wp[c, \theta](f_1) \preceq \wp[c, \theta](f_2)\)
- **Linearity:** \(\wp[c, \theta](\alpha_1 \cdot f_1 + \alpha_2 \cdot f_2) = \alpha_1 \cdot \wp[c, \theta](f_1) + \alpha_2 \cdot \wp[c, \theta](f_2)\)
- **Preservation of 0.1:** \(\wp[c, \theta](0) = 0\) and \(\wp[c, \theta](1) = 1\)

**Proof.** See Appendix A.1. \(\square\)

**Program termination.** Since the termination behavior of a program is given by the probability that it establishes true, we can readily use the transformer \(\wp\) to reason about program termination. It suffices to consider the weakest pre-\(\wp\) of the program w.r.t. post-\(\wp\) true \(= 1\). Said otherwise, \(\wp[c, \theta](1)\) gives the termination probability of program \((c, \theta)\) from state \(s\). In particular, if the program terminates with probability 1, we say that it terminates almost-surely.

### 3.3 Characterization based on Fixed Points

Next we use a continuity argument on the transformer \(\wp(\cdot,p)\) to prove that its action on recursive procedures can also be defined using fixed point techniques. This alternative characterization rests on a subsidiary transformer \(\wp(\cdot,p)_{\ell, \alpha}\), which is a slight variant of \(\wp(\cdot,p)\). The main difference between these transformers is the mechanism that they use to give semantics to procedure calls: \(\wp(\cdot,p)\) relies on a declaration \(\theta\), while \(\wp(\cdot,p)_{\ell, \alpha}\) relies on a so-called (liberal) semantic environment \(\theta: \mathbb{E} \rightarrow \mathbb{E}(\theta: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1})\) which is meant to directly encode the semantics of procedure calls. Then \(\wp(\cdot,p)(P_f)\) gives \(\theta(f)\), while for all other program constructs \(c\), \(\wp(\cdot,p)(c)(f)\) agrees with \(\wp(c)(f)\); see Figure 8 in Section A.2 for details. For technical reasons, in the remainder of our development we will consider only continuous semantic environments in \(\mathbb{SEnv} = \{f \mid f: \mathbb{E} \rightarrow \mathbb{E} \text{ is upper continuous}\}\) and \(\mathbb{LSEnv} = \{f \mid f: \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1} \text{ is lower continuous}\}\). The fixed points above are taken w.r.t. the pointwise order \(\subseteq\) over semantic environments: given \(\theta_1, \theta_2 \in \mathbb{SEnv}\) (resp. \(\theta_1, \theta_2 \in \mathbb{LSEnv}\)), \(\theta_1 \subseteq \theta_2\) iff \(\theta_1(f) \subseteq \theta_2(f)\) for all \(f \in \mathbb{E}\) (resp. \(f \in \mathbb{E}_{\leq 1}\)).

**Theorem 3.1** (Fixed point characterization for procedure calls). Given a declaration \(\theta: \mathbb{P} \rightarrow \mathbb{C}\) for procedure \(P\),

\[
\wp[call P, \theta] = \inf_{\theta: \mathbb{SEnv}} \left( \lambda \theta: \mathbb{SEnv}. \wp[call P(\theta)] \right) \\
\wp[call P, \theta] = \sup_{\theta: \mathbb{LSEnv}} \left( \lambda \theta: \mathbb{LSEnv}. \wp[call P(\theta)] \right)
\]

**Proof.** See Appendix A.2. \(\square\)

### 4. Correctness of Recursive Programs

In this section we introduce some proof rules for effectively reasoning about the behavior of recursive programs. For that we require the notion of constructive derivability. Given logical formulae \(A\) and \(B\), we use \(A \vdash B\) to denote that \(B\) can be derived assuming \(A\). In particular, we will consider claims of the form

\[
\wp[call P](f) \preceq g \vdash \wp[\theta](f) \preceq \wp[\theta](g) \\quad \text{ [wp-rec]}
\]

So for proving that a procedure \(call\) satisfies a specification (given by \(f, g\)), it suffices to show that the procedure’s body satisfies the specification, assuming that the recursive calls in the body do, too.

**Example 3.** Reconsider the procedure \(P_{\text{rec}}\) with declaration

\[
\forall P_{\text{rec}}: \{\text{skip}\} \rightarrow \{\text{call } P_{\text{rec}}\} \vdash \{\text{call } P_{\text{rec}}\}
\]

presented in the introduction. We prove that it terminates with probability at most \(\varphi = \frac{\sqrt{5} - 1}{2}\) from any initial state. Formally, this is captured by \(\wp[call P, \theta](1) \preceq \varphi\). To prove this, we apply rule [wp-rec]. We must then establish the derivability claim

\[
\wp[call P](1) \preceq \varphi \vdash \wp[\theta](P_{\text{rec}})(1) \preceq \varphi
\]

The derivations goes as follows:

\[
\wp[\theta](P_{\text{rec}})(1) = \{\text{def. of } \wp[\theta]\} \varphi = \frac{1}{2} \cdot \wp[\text{skip}](1) + \frac{1}{2} \cdot \wp[\text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}](1) = \{\text{def. of } \wp[\theta]\} \varphi
\]

\[
\frac{1}{2} + \frac{1}{2} \cdot \wp[\text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}](\wp[\text{call } P_{\text{rec}}](1)) = \{\text{assumption, monot. of } \wp[\theta]\} \varphi
\]

\[
\frac{1}{2} + \frac{1}{2} \cdot \wp[\text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}](\varphi) = \{\text{def. of } \wp[\theta], \text{scabal. of } \wp[\theta]\} \varphi
\]

\[
\frac{1}{2} + \frac{1}{2} \cdot \wp[\text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}](1) = \{\text{assumption, monot. of } \wp[\theta]\} \varphi
\]

\[
\frac{1}{2} + \frac{1}{2} \cdot \wp[\text{call } P_{\text{rec}}; \text{call } P_{\text{rec}}]^3(1) = \{\text{scalar of } \wp[\theta]\} \varphi^3
\]

\[
\varphi
\]

\(\triangle\)
An appealing feature of our approximation semantics is to prove the following soundness result we do not need to resort to a continuity argument on the expectation transformers.

**Theorem 4.1** (Soundness of rules $\text{wp}(\text{p-rec})$). Rules $\text{wp}(\text{p-rec})$ and $\text{wlp}(\text{p-rec})$ are sound w.r.t. the $\text{wp}(\text{l})$p semantics in Figure 1.

**Proof.** See Appendix A.3. \hfill $\square$

Rules $\text{wp}(\text{l})p$-rec allow deriving only one-sided bounds for the weakest (liberal) pre-expectation of a procedure call. It is also possible to derive two-sided bounds by means of the following rules:

\[
\begin{align*}
  l_0 &= 0, \quad u_0 = 0, \\
  l_n &\leq \text{wp}(\text{call } P)(f) \leq u_n \quad \text{if } l_{n+1} \leq \text{wp}(\text{d}(P))(f) \leq u_{n+1} & \text{[wp-rec \_\_]} \\
  l_0 &= 1, \quad u_0 = 1, \\
  l_n &\leq \text{wp}(\text{call } P)(f) \leq u_n \quad \text{if } l_{n+1} \leq \text{wp}(\text{d}(P))(f) \leq u_{n+1} & \text{[wlp-rec \_\_]} \\
  \inf_n l_n &\leq \text{wp}(\text{call } P, d)(f) \leq \sup_n u_n & \text{[wp-rec \_\_]} \\
  \sup_n l_n &\leq \text{wp}(\text{call } P, d)(f) \leq \inf_n u_n & \text{[wlp-rec \_\_]} \\
\end{align*}
\]

In contrast to rules $\text{wp}(\text{l})p$-rec, these rules require exhibiting two sequences of expectations $\langle l_n \rangle$ and $\langle u_n \rangle$ rather than a single expectation $g$ to bound the weakest (liberal) pre-expectation of a procedure call. Intuitively $l_n$ ($u_n$) represents a lower (upper) bound for the weakest pre-expectation of the $n$-inlining of the procedure, i.e. from the premises of the rules we will have $l_n \leq \text{wp}(\text{call } P)(f) \leq u_n$ for all $n \in \mathbb{N}$.

Observe that both rules can be specialized to reason about one-sided bounds. For instance, by setting $u_{n+1} = \infty$ in [wp-rec \_] we can reason about lower bounds of $\text{wp}(\text{call } P, d)(f)$, which is not supported by rule [wp-rec \_]. Similarly, by taking $l_n = 0$ in rule [wlp-rec \_] we can reason about upper bounds of $\text{wp}(\text{call } P, d)(f)$.

**Example 4.** Reconsider the procedure $P_{\text{rec}}$ from Example 3. Now we prove that the procedure terminates with probability at least $\varphi = \frac{\sqrt{5} - 1}{2}$ from any initial state. To this end, we rely on the fact that $\varphi$ can be characterized by the asymptotic behavior of the sequence $\langle \varphi_n \rangle$, where $\varphi_0 = 0$ and $\varphi_{n+1} = \frac{1}{2} + \frac{1}{2} \varphi_n$. In symbols, $\varphi = \sup_n \varphi_n$. We wish then to prove that

\[
\sup_n \varphi_n \leq \text{wp}(\text{call } P_{\text{rec}}, d)(1). 
\]

To establish this formula we apply the one side variant of rule [wp-rec \_] to reason about lower bounds of $\text{wp}(\text{call } P_{\text{rec}}, d)(1)$, that is, we implicitly take $u_{n+1} = \infty$. We must then establish

\[
\varphi_n \leq \text{wp}(\text{call } P_{\text{rec}})(1) \quad \text{if } \varphi_{n+1} \leq \text{wp}(\text{d}(P_{\text{rec}}))(1). 
\]

The derivation follows the same steps as those taken in Example 3 to give upper bounds on $\text{wp}(\text{call } P_{\text{rec}}, d)(1)$. Combining the result proved with that in Example 3, we conclude that $\varphi = \frac{\sqrt{5} - 1}{2}$ is the exact termination probability of $(\text{call } P_{\text{rec}}, d)$. \hfill $\square$

Lastly, we can establish the correctness of our rules.

**Theorem 4.2** (Soundness of rules $\text{wp}(\text{l})p$-rec). Rules $\text{wp}(\text{l})p$-rec are sound w.r.t. the $\text{wp}(\text{l})p$ semantics in Figure 1.

**Proof.** See Appendix A.3. \hfill $\square$

To conclude the section we would like to point out that the rules [wp-rec \_] is related to previous work on proof rules. It can be viewed as a generalization of Jones’s loop rule [15] to the case of recursion (even though Jones originally presented a one-sided version) and as an adaptation of Audebaud and Paulin-Mohring’s rule [1] to our weakest pre-expectation semantics. The counterpart of the rule for partial correctness, on the other hand, is, to the best of our knowledge, novel.

**5. The Expected Runtime of Programs**

To further our study of recursive probabilistic programs we now develop a calculus for reasoning about the expected or average runtime of PRGCL programs. This calculus builds upon our previous work in [17] and is able to handle recursive procedures.

**5.1 The Expected Runtime Transformer $\text{ert}$**

We assume a runtime model where executing a skip statement, an assignment, evaluating the guard in a conditional branching and invoking a procedure consumes one unit of time. On the other hand, combining two programs by means of a sequential composition or a probabilistic choice consumes no additional time other than that consumed by the original programs. Likewise, halting a program execution with an abort statement consumes no unit of time.

Since the runtime of a program varies according to the initial state from which it is executed, our aim is to associate to each program $(c, d)$ a mapping that takes each state $s$ to the expected time until $(c, d)$ terminates on $s$. Such mappings will range over the set of runtimes $\mathcal{T} \triangleq \{ t \mid t : S \rightarrow [0, \infty) \}$.

To associate each program to its runtime we use a continuation passing style formalized by the transformer

\[
\text{ert}[\cdot] : \mathcal{T} \rightarrow \mathcal{T}.
\]

If $t \in \mathcal{T}$ represents the runtime of the computation that follows program $(c, d)$, then $\text{ert}[c, d][t]$ represents the overall runtime of $(c, d)$, plus the computation following $(c, d)$. Runtime $t$ is usually referred to as the continuation of $(c, d)$. In particular, by setting the continuation of a program to zero we recover the runtime of the plain program. That is, for every initial state $s$,

\[
\text{ert}[c, d][0](s)
\]

gives the expected runtime of program $(c, d)$ from state $s$.

The transformer $\text{ert}[c, d]$ is defined by induction on the structure of $c$, following the rules in Figure 2.

The rules are defined so as to correspond to the aforementioned runtime model. That is, $\text{ert}[c, d][0]$ captures the expected number of assignments, guard evaluations, procedure calls and skip statements in the execution of $(c, d)$. Most rules are self-explanatory. $\text{ert}[\text{skip}, d]$ adds one unit of time to the continuation since skip does not modify the program state and its execution takes one unit of time. $\text{ert}[x := E, d]$ also adds one unit of time, but to the continuation evaluated in the state resulting from the assignment. $\text{ert}[\text{abort}, d]$ yields always the infinite runtime.

3 Loosely speaking, the overall runtime of a procedure call is then one plus the runtime of executing the procedure’s body.

4 Strictly speaking, the set of runtimes $\mathcal{T}$ coincides with the set of unbounded expectations $\mathcal{E}$ but we prefer to distinguish the two sets since they are to represent different objects. We will, however, keep the same notations for runtimes as for expectations, for example $t[x/E]$, $t_1 \leq t_2$, etc.
constant runtime 0 since abort aborts any subsequent program execution (making their runtime irrelevant) and consumes no time.  
\[ \eta = \text{if } \{G\} \{c_1\} \text{ else } \{c_2\}, D \] adds one unit of time to the runtime of either of its branches, depending on the value of the guard.  
\[ \eta = \{\{c_3\}\} \{\{c_4\}\} \{\{c_5\}\} \{\{c_6\}\} \} \] gives the weighted average between the runtime of its branches, each of them weighted according to its probability. \[ \eta = \{c_1 ; c_2 , D \} \] first applies \[ \eta = \{c_2 , D \} \] to the continuation and then \[ \eta = \{c_1 , D \} \] to the resulting runtime of this application. Finally, \[ \eta = \{c_1 , D \} \] is defined using fixed point techniques.

To understand the intuition behind the definition of \[ \eta = \{c_1 , D \} \] recall that \[ \eta = \{c_1 \} \] consumes one unit of time more than the body of \[ \eta = \{c_2 \} \]. To capture this fact we make use of the auxiliary runtime transformer \[ \eta = \{c_1 \} \] computing the runtime of the former and \[ \eta = \{c_2 \} \] computing the runtime of the latter.

**Proof.** For every program \( \eta = \{c \} \) and initial state \( s \) of the program,
\[ \eta = \{c \} (0) (s) < \infty \implies \eta = \{c \} (1) (s) = 1 . \]

**Theorem 5.2** allows giving a very short proof of a well–known result relating expected runtimes and termination probabilities: If a program has finite expected runtime, it terminates almost surely.

**Theorem 5.3.** For every abort–free program \( \eta = \{c \} \) and initial state \( s \) of the program,
\[ \eta = \{c \} (0) (s) < \infty \implies \eta = \{c \} (1) (s) = 1 . \]

**Proof.** By instantiating Theorem 5.2 with \( t = 1 \) and using the propagation of constants property of \( \eta = \{c \} \) (Theorem 5.1) to decompose \[ \eta = \{c \} (1) \] as \( 1 + \eta = \{c \} (0) \).

Observe that in Theorem 5.3 we cannot drop the abort–free requirement on the program. To see this, consider the program \( \eta = \{\text{skip} \} \{\text{if } y = 1 \{\text{abort} \} \} \). The program has a finite runtime \( \eta = \{\text{c} \} (0) = 1/2 < \infty \) and terminates, however, with probability less than one \( \{\text{wp} \} (1) = 1/2 < 1 \). Moreover, observe that Theorem 5.3 is only valid on the stated direction: A probabilistic program can terminate almost–surely and require, still, an expected infinite time to reach termination. This phenomenon is illustrated, for instance, by the one dimensional random walk; see e.g. [17, 7].

Even though Theorem 5.3 constitutes a well–known and natural result on probabilistic programs, our contribution here is to give the first fully formal proof of such a result.

### 5.2 Proof Rules for Recursive Programs

The runtime of procedure calls, which includes, in particular, recursive programs, is defined using fixed points. To avoid reasoning about fixed points we propose some proof rules based on invariants.

We show that an adaptation of the proof rules for procedure calls from our \( \eta = \{ \text{wp} \} \)–calculus is sound for the \( \eta = \{ \text{rec} \} \)–calculus. The rules are:

- \[ \eta = \{\text{rec} \} (t) \leq 1 + u \quad \implies \quad \eta = \{\text{rec} \} (u) \]  
- \[ \eta = \{\text{call} \} (t) \leq 1 + v \quad \implies \quad \eta = \{\text{call} \} (u) \]

Compares to the proof rules from the \( \eta = \{ \text{wp} \} \)–calculus, these proof rules require incrementing by one unit some of the bounds. Loosely speaking, this is because the runtime of a procedure call is one plus the runtime of its body, whereas the semantics of a procedure call fully agrees with the semantics of its body.

**Example 5.** To illustrate the use of the rules, consider the faulty factorial procedure with declaration
\[ \eta = \{\text{fact} \} : \{x \preceq 0 \} \{y = 1 \} \text{ else } \{c_1 \} \{y = 0 \} ; \]

where \( c_1 = x = x - 1 \) and \( c_2 = x = x - 2 \) and \( \eta = \{\text{call} \} \{x \preceq 0 \} \{y = 1 \} \text{ else } \).  

We prove that on input \( x \) the expected runtime of the procedure is \( 2 + \alpha_k \), where \( \alpha_k = \frac{1}{49} \left( 121 + 210k + 432 \left[ -\frac{3}{2} \right]^{k+1} \right) \).

Since the term \( 432 \left[ -\frac{3}{2} \right]^{k+1} \) is negligible, we can approximate the procedure’s runtime by \( 4.5 + 4.3k \).  

We can formally capture our exact runtime assertion by
\[ \eta = \{\text{call} \} (0) (s) = 1 + \sup_n t_n \]

where \( t_n = 1 + [x < 0] + 0 \leq x \leq n] \alpha_x + [x > n] \alpha_{n+1} \).  

To see this, observe that the sequence \( \alpha_k \) is increasing and therefore \( \sup_n t_n = 1 + [x < 0] + 1 \leq x \) \( \alpha_x \). We prove the runtime assertion using rule \( \text{rec} \) with instantiations \( t = 0 \) and \( t_n = \alpha_n = t_n \) for \( n \geq 1 \). We have to discharge the premise
\[ \eta = \{\text{call} \} (0) (s) = 1 + t_n \implies \eta = \{\text{rec} \} (1) (s) = 1 + t_{n+1} \]
Since some simple calculations yield
\[
\text{ert}[P_\text{fact}](\mathbf{0}) = 1 + [x \leq 0] \cdot 1 \\
+ [x > 0] \cdot \left(\frac{1}{2} \cdot \text{ert}[c_1](1) + \frac{1}{6} \cdot \text{ert}[c_2](1)\right),
\]
our next step is to compute \(\text{ert}[c_1](1)\) (the calculations are identical for \(\text{ert}[c_2](1)\)). To do so, we rely on assumption \(\text{ert}[\text{call } P](\mathbf{0}) = 1 + t_n\), and the propagation of constants property of \(\text{ert}\).
\[
\text{ert}[c_1](1) = \text{ert}[x := x - 1; \text{call } P_\text{fact}](\text{ert}[x := x + 1](1)) = 2 + \text{ert}[x := x - 1; \text{call } P_\text{fact}](\mathbf{0}) = 2 + \text{ert}[x := x - 1](1 + t_n) = 4 + t_n[x/x + 1]
\]
The derivation then concludes by showing that
\[
t_{n+1} = 1 + [x \leq 0] \cdot 1 \\
+ [x > 0] \cdot \left(\frac{1}{2} (4 + t_n[x/x + 1]) + \frac{1}{6} (4 + t_n[x/x + 2])\right),
\]
which after some term reordering reduces to proving that \(c_{01} = 1\), \(c_{11} = 7\) and \(c_{12} = 5 + 2^c_{14} + 5c_{12}\).

We conclude the section establishing the soundness of the rules.

**Theorem 5.4** (Soundness of rules \([\text{ett-rec}], [\text{ett-rec}_{\text{c}}]\)). Rules \([\text{ett-rec}]\) and \([\text{ett-rec}_{\text{c}}]\) are sound w.r.t. the \(\text{ert}\)-calculus in Figure 2.

**Proof.** See Appendix A.8.

6. **Operational Semantics**

We provide an operational semantics for pRGCL programs in terms of pushdown Markov chains with rewards (PRMC) [3] and prove the transformer \(\wp\) to be sound with respect to this semantics. Due to space limitations, this section contains an informal introduction only. Corresponding formal definitions are found in Appendix A.9.

For simplicity, we assume a canonical labeling for each command \(c \in C\) together with auxiliary functions \(\text{init}, \text{succ}_1, \text{succ}_2\) and \(\text{stmt}\) determining the initial location, the first and second successor of a location and the program statement corresponding to a label. As an example, the labels attached to each statement of program \(c\) from Example 3 are as follows:
\[
c = \text{\{skip\}} [\text{\{if\}2] \text{\{call } P^3\text{; call } P^4\text{; call } P^5\}.
\]

The definition of the auxiliary functions is straightforward. For instance, we have \(\text{init}(c) = 2\), \(\text{succ}_1(1) = \downarrow\), \(\text{succ}_2(2) = 3\), and \(\text{stmt}(2) = c\), where \(\downarrow\) is a special symbol indicating termination of a procedure. Moreover, label \(\text{Term}\) stands for termination of the whole program.

Our operational semantics of pRGCL programs is given as an execution relation, where each step is of the form
\[
(\ell, s) \overset{\gamma, P, \gamma'}{\rightarrow} (\ell', s').
\]

Here, \(\ell, \ell'\) are program labels, \(s, s' \in S\) are program states, \(\gamma\) is a program label being popped from and \(\gamma'\) a finite sequence of labels being pushed on the stack, respectively. \(P \in \{0, 1\}\) denotes the probability of executing this step.

This execution relation corresponds to the transition relation of a PRMC, where each pair \((\ell, s)\) is a state and the stack alphabet is given by the set of all labels of a given PRGCL program. Moreover, given \(f \in E\), a reward of \(f(s)\) is assigned to each state of the form \((\text{Term}, s)\). Otherwise, the reward of a state is 0. Figure 3 shows the rules defining the operational semantics of pRGCL programs. The rules in Figure 3 are self-explanatory. In case of a procedure call, the calls successor label is pushed on the stack and execution continues with the called procedure. Whenever a procedure terminates, i.e. reaches a state \((\downarrow, s)\), and the stack is non-empty, a return address is popped from and execution continues at this address.

Figure 4 shows the PRMC of example program \(c\). The initial state is 2 (the probabilistic choice). Say the right branch is chosen; we move to 3. The statement at 3 is a call, and the address after the call is 4; so 4 is pushed and the procedure body is reentered. Say now the left branch is chosen: we move to 1 (the skip) and then terminate, i.e. we move to \(\downarrow\). Recall that return address 4 is on top of the stack; 4 is popped, we move to 4 to continue execution.

The expected reward that PRMC \(\mathcal{M}\) associated to program \((c, D)\) reaches a set of target states \(T\) from initial state \((\ell, s)\) is defined as
\[
\text{ExpRew}^{\mathcal{M}}[c, D](T) = \sum_{\pi \in \Omega(\ell, s, T)} \text{Prob}^\mathcal{M}(\pi) \cdot \text{rew}(\pi),
\]
where \(\pi\) is a path from \((\ell, s)\) to some target state, \(\text{Prob}^\mathcal{M}(\pi)\) is the probability of \(\pi\) and \(\text{rew}(\pi)\) is the reward collected along \(\pi\).

We are now in a position to state the relationship between the operational model and the denotational semantics:

**Theorem 6.1** (Correspondence Theorem). Let \(c \in C, f \in E\), and \(T = \{(\text{Term}, s) | s \in S\}\). Then for each \(s \in S\), we have
\[
\text{ExpRew}^{\mathcal{M}}[c, D](T) = \wp(c, D)(f)(s).
\]

**Proof.** See Appendix A.10.

In the spirit of [11] a similar result can be obtained for \(\wp\). For that one needs a liberal expected reward being defined as the expected reward plus the probability of not reaching the target states at all. One can then show a similar correspondence to \(\wp\).

7. **Extensions**

**Mutual recursion.** Both our \(\wp\)– and \(\text{ert}\)–calculus can be extended to handle multiple procedures. Say we want to handle \(m\) (possibly mutually recursive) procedures \(P_1, \ldots, P_m\) with declaration \(D \in C^m\). The definition of \(\wp(\text{call } P_i, D)\) remains the same, we only need to adapt the definition of the \(n\)-inlining call_{\text{in}}^n P_i of procedure \(P_i\) as to inline the calls of all procedures:
\[
call_{\text{in}}^n P_i = \mathcal{D}(P_i)\text{call } P_i/\text{call } P_i, \ldots, \text{call } P_i/\text{call}_{\text{in}}^n P_i.
\]

As for the \(\text{ert}\)-calculus, a runtime environment is now a tuple \(\eta = (\eta_1, \ldots, \eta_m)\), where \(\eta_i\) is meant to provide the behavior of procedure \(P_i\) in \(\text{ert}[\bullet]_\eta^\bullet\), i.e. \(\text{ert}[\text{call } P_i]_{\text{in}}^\bullet = \eta_i\). The action of \(\text{ert}\) on procedure calls is then defined simultaneously as\(^9\)
\[
\text{ert}(\text{call } P_i, D), \ldots, \text{ert}(\text{call } P_m, D) = \text{ifp}\left(\lambda \eta, \left(1 + \text{ert}[\mathcal{D}(P_i)]_{\text{in}}^\bullet, \ldots, 1 + \text{ert}[\mathcal{D}(P_m)]_{\text{in}}^\bullet\right)\right).
\]

The proof rules for reasoning about procedure calls in both calculi are easily adapted. We show only the case of \([\wp\text{-rec}]\); the others admit a similar adaptation.
\[
\wp(\text{call } P_i)(f_i) \leq g_i, \ldots, \wp(\text{call } P_m)(f_m) \leq g_m \iff \wp(\mathcal{D}(P_i))(f_i) \leq g_i, \ldots, \wp(\mathcal{D}(P_m))(f_m) \leq g_m, \text{ for } i = 1, \ldots, m
\]

The rule reasons about all the procedures simultaneously. Roughly speaking, the rule premise requires deriving the specification \(g_i\).

\(^7\) \(T\) denotes the set of states representing successful termination of the pushdown automaton.

\(^8\) For determining the \(\text{least}\) fixed point, environments are compared component-wise, i.e. \((\eta_1, \ldots, \eta_m) \sqsubseteq (\nu_1, \ldots, \nu_m)\) iff \(\eta_i \subseteq \nu_i\) for all \(i = 1 \ldots m\).
The algorithm we analyze searches for value \( \text{val} \) in array \( a[\text{left .. right}] \). It is encoded by procedure \( B \) with declaration \( \mathcal{D} \) presented in Figure 5. We use random assignment \( \text{mid} \leftarrow \text{uniform}(\text{left}, \text{right}) \) to model the random election of the pivot. For simplicity, we assume that the random assignment is performed in constant time if \( \text{left} \leq \text{right} \) and that it diverges if \( \text{left} > \text{right} \).

**Partial correctness.** We verify the following partial correctness property: When \( B \) is invoked in a state where \( \text{left} \leq \text{right}, a[\text{left .. right}] \) is sorted, and \( \text{val} \) occurs in \( a[\text{left .. right}] \), then the invocation of \( B \) stores \( \text{mid} \) in the index where \( \text{val} \) lies. Formally,

\[
g \leq \text{wlp}[\text{call } B(\mathcal{D})](f), \quad \text{with} \quad g = \left[ \text{left} \leq \text{right} \right] \cdot \left[ \text{sorted}(\text{left}, \text{right}) \right] \cdot \left[ \exists x \in \text{left, right} : a[x] = \text{val} \right] \cdot \left[ a[\text{mid}] = \text{val} \right].
\]

where \( \left[ \text{sorted}(y, z) \right] \) is the indicator function of \( a[y .. z] \) being sorted. In order to prove \( g \preceq \text{wlp}[\text{call } B(f)] \) we apply rule \( \text{wlp-rec} \). We are then left to prove

\[
g \preceq \text{wlp}[\text{call } B(f)](f) \iff g \preceq \text{wlp}[\text{call } B(f)](f).
\]

The way in which we propagate post-expectation \( f \) from the exit point of the procedure till its entry point, obtaining pre-expectation \( g \), is fully detailed in Figure 5. To do so we use assumption \( g \preceq \text{wlp}[\text{call } B(f)] \) and monotonicity of \( \text{wlp} \).

Dually, we can verify that when \( \text{val} \) is not in the array, the value of \( a[\text{mid}] \) after termination of \( B \) is different from \( \text{val} \). A detailed derivation of this property is provided in Appendix A.11, Figure 9.

**Expected runtime.** We perform a runtime analysis of the algorithm for those inputs where \( \text{val} \) does not occur in the array. Under this assumption we can distinguish two cases: either \( \text{val} \) is smaller than every element in the array or larger than all of them.

For the first case we show that the expected runtime of the algorithm is upper bounded by \( 1 + u \), with

\[
u = \left[ \text{left} < \text{right} \right] \cdot \infty + 3 + \left[ \text{left} < \text{right} \right] \cdot \left( 5 \cdot H_{\text{right} - \text{left} + 1} - 5/2 \right),
\]

and \( H_k \) being the \( k \)-th harmonic number. Formally, we show that

\[
\text{ert}[\text{call } B(\mathcal{D})](\emptyset) \preceq 1 + u\text{ applying rule } [\text{ert-rec}] \text{. We must then establish}
\]

\[
\text{ert}[\text{call } B](0) \preceq 1 + u \iff \text{ert}[\mathcal{D}](0) \preceq u.
\]

The details of this derivation are provided in Figure 6.

Similarly, when \( \text{val} \) is greater than every element in the array, the expected runtime is upper bounded by \( 1 + u \), with

\[
u = \left[ \text{left} > \text{right} \right] \cdot \infty + 3 + \left[ \text{left} < \text{right} \right] \cdot \left( 6 \cdot H_{\text{right} - \text{left} + 1} - 3 \right).
\]
The verification for this case is analogous therefore omitted. Combining the two cases we conclude that when the sought—after value does not occur in the array, the algorithm terminates in expected time in \( \Theta(\log n) \), where \( n = \text{right} - \text{left} + 1 \) is the size of the array, since \( H_k \in \Theta(\log k) \).

9. Related Work

**wp-style reasoning for recursive programs.** Recursion has been treated for non–probabilistic programs. Hesselink [13] provided several proof rules for recursive procedures, both for total and partial correctness. Our first two proof rules are extensions of his rules to the probabilistic setting. Predicate transformer semantics for recursive non–deterministic procedures has been provided by Bon Sangue and Kok [2] and Hesselink [13]. Nipkow [27] provides an operational semantics and a Hoare logic for recursive (parameterless) non–deterministic procedures. Zhang et al. [33] establishes the equivalence between an operational semantics and a weakest pre–condition semantics for recursive programs in Coq. To some extent our transfer theorem between probabilistic pushdown automata and the wp–semantics can be considered as a probabilistic extension of this work.

**Deductive reasoning for recursive probabilistic programs.** Jones provided several proof rules for recursive probabilistic programs in her Ph.D. dissertation [15]. One of our proof rules is a generalisation of Jones’ proof rule to general recursion. McIver and Morgan [22] also provide a wp–semantics of probabilistic recursive programs. While [22] use fixed point techniques, we follow low e.g. Hehner [12] and define the semantics of a recursive procedure as the limit of an approximation sequence. In contrast to our approach based on procedures, [22] introduced recursion through the language constructor \( \text{rec } B \), where \( B \) is a program–semantics transformer. (Intuitively \( B \) encodes how the recursive procedure defined (and invoked) by \( \text{rec } B \) transforms the outcome of its recursive calls). Our approach provides a strict separation between program syntax and semantics. Moreover our approach based on procedure calls can model mutual recursion in a natural way (see Section 7), while the approach in [22] approach does not accommodate so naturally to such cases. Audebaud and Paulin-Mohring [1] present a mechanized method for proving properties of randomized algorithms in the Coq proof assistant. Their approach is based on higher–order logic, in particular using a monadic interpretation of programs as probabilistic distributions. Our proof rule for obtaining two–sided bounds on recursive programs is directly adapted from their work. They however do neither relate their work to an operational model nor support the analysis of expected runtimes.

**Semantics of recursive probabilistic programs.** Gupta et al. consider the interplay between constraints, probabilistic choice, and recursion in the context of a (concurrent) constraint–based probabilistic programming language. They provide an operational semantics using labeled transition systems and (weak) bisimulation as well as a denotational semantics. Recursion is treated operationally by considering the limit of syntactic finite approximations. In the denotational semantics, the mixture of probabilities and constraints...
violates basic monotonicity properties for a standard treatment of recursion. Their main result is that the transition system semantics modulo weak bisimulation is fully abstract with respect to the input–output relation of processes. They do neither consider non-determinism nor reasoning about recursive probabilistic programs. Pfeffer and Koller [29] provide a measure–theoretic semantics of recursive Bayesian networks and show that every recursive probabilistic relational database has a probability measure as model. This is complemented by an inference algorithm that obtains approximations by basically unfolding the recursive Bayesian network. Recently, Toronto et al. [30] provided a measure–theoretic semantics for a probabilistic programming language with recursion. Their interpretation of recursive programs is however restricted to (almost surely) terminating programs.

Probabilistic pushdown automata. The analysis of probabilistic pushdown automata, which correspond to the model of recursive Markov chains, has been well–investigated. Key computational problems for analyzing classes of these models can be reduced to computing the least fixed point solution of corresponding classes of monotone polynomial systems of non–linear equations. For subclasses of these models termination probabilities, ω–regular properties, and expected runtimes can be algorithmically obtained. Recent surveys are provided by Etessami [8] and Brazdil et al. [3]. Our transfer theorem indicates that (some of) these results are transferable to obtaining weakest pre–expectations for recursive probabilistic programs having a finite–control probabilistic push–down automata. A detailed study is outside the scope of this paper and left for future work.

10. Conclusion

We have presented two wp-calculi: one for reasoning about correctness, and one for analysing expected run–times of recursive probabilistic programs. The wp-calculi have been related, equipped with proof rules, and exemplified by analysing a Sherwood version of binary search. A relation with a straightforward operational interpretation using pushdown Markov chains has been established. We believe that this work provides a good basis for the automation of the analysis of recursive probabilistic programs. Future work consists of applying our calculi to other recursive randomized algorithms (such as quick sort with random pivot selection). Other future work includes investigating a generalisation of Colussi’s technique [5] to transform a recursive program and its correctness proof into a non–recursive program with its accompanying correctness proof. This would allow to transfer—typically simpler—correctness proofs of the recursive probabilistic programs to non-recursive ones.

References

A. Appendix

For our proofs about transformer \(\text{wp}\), we observe that "\(\preceq\)" endows the set of unbounded expectations \(E\) with the structure of an upper \(\omega\)-\(\text{cpo}\), where the supremum of an increasing \(\omega\)-chain \(f_0 \preceq f_1 \preceq \cdots\) is given pointwise, i.e. \((\sup_n f_n)(s) \triangleq \sup_n f_n(s)\). Likewise, "\(\preceq\)" endows the set of bounded expectations \(E_{<1}\) with the structure of a lower \(\omega\)-\(\text{cpo}\), where the infimum of a decreasing \(\omega\)-chain \(f_0 \succeq f_1 \succeq \cdots\) is given pointwise, i.e. \((\inf_n f_n)(s) \triangleq \inf_n f_n(s)\). Upper \(\omega\)-\(\text{cpo}\) \((E, \preceq)\) has as bottom element the constant expectation \(0\), while lower \(\omega\)-\(\text{cpo}\) \((E_{<1}, \preceq)\) has as top element the constant expectation \(1\).

In what follows, we usually refer to the set of upper continuous expectation transformers\(^1\) over \((E, \preceq)\) and the set of lower continuous expectation transformers over \((E_{<1}, \preceq)\). We use \(E \xrightarrow{\text{upper}} E\) and \(E_{<1} \xrightarrow{\text{lower}} E_{<1}\) to denote such sets.

A.1 Basic Properties of the \(w(l)p\)-Transformer

**Proof of Continuity.** We prove continuity by induction on the program structure. Let \(f_0 \preceq f_1 \preceq f_2 \preceq \cdots\) and \(g_0 \succeq g_1 \succeq g_2 \succeq \cdots\). For the base cases we have:

**skip:**

\[
\text{wp}[\text{skip}, D]\left(\sup f_n\right) = \sup n f_n = \sup n \text{wp}[\text{skip}, D](f_n)
\]

And

\[
\text{wp}[\text{skip}, D]\left(\inf g_n\right) = \inf g_n = \inf \text{wp}[\text{skip}, D](g_n)
\]

**x := E:**

\[
\text{wp}[x := E, D]\left(\sup f_n\right) = \left(\sup_n f_n\right)[x/E] = \sup_n f_n[x/E] = \sup_n \text{wp}[x := E, D](f_n)
\]

And

\[
\text{wp}[x := E, D]\left(\inf g_n\right) = \left(\inf g_n\right)[x/E] = \inf g_n[x/E] = \inf \text{wp}[x := E, D](g_n)
\]

**abort:**

\[
\text{wp}[\text{abort}, D]\left(\sup f_n\right) = 0 = \sup n 0 = \sup n \text{wp}[\text{abort}, D](f_n)
\]

And

\[
\text{wp}[\text{abort}, D]\left(\inf g_n\right) = 1 = \inf 1 = \inf \text{wp}[\text{abort}, D](g_n)
\]

For the induction hypothesis we assume that for any two programs \(c_1\) and \(c_2\) continuity holds. Then we can perform the induction step:

\(^{10}\)Given a binary relation \(\preceq\) over a set \(A\), we say that \((A, \preceq)\) is an upper (resp. lower) \(\omega\)-\(\text{cpo}\) if \(\preceq\) is reflexive, transitive and antisymmetric, and every increasing \(\omega\)-chain \(a_0 \preceq a_1 \preceq \cdots\) (resp. decreasing \(\omega\)-chain \(a_0 \succeq a_1 \succeq \cdots\)) has a supremum (resp. infimum) \(\sup_n a_n\) (resp. \(\inf_n a_n\)) in \(A\).

\(^{11}\)A function \(f : A \rightarrow B\) between two upper (resp. lower) \(\omega\)-\(\text{cpo}\)s \((A, \preceq_A)\) and \((B, \preceq_B)\) is upper (resp. lower) continuous iff for every increasing \(\omega\)-chain \(a_0 \preceq a_1 \preceq \cdots\) (resp. decreasing \(\omega\)-chain \(a_0 \succeq a_1 \succeq \cdots\)), \(\sup_n f(a_n) = f(\sup_n a_n)\) (resp. \(\inf_n f(a_n) = f(\inf_n a_n)\)).

if \((G)\) \(\{c_1\}\) else \(\{c_2\}\):

\[
\text{wp}[if (G)\{c_1\} else \{c_2\}, D]\left(\sup f_n\right) = \left[|G| \cdot \text{wp}[c_1, D]\left(\sup f_n\right) + \neg|G| \cdot \text{wp}[c_2, D]\left(\sup f_n\right)\right]
\]

And

\[
\text{wp}[if (G)\{c_1\} else \{c_2\}, D]\left(\inf g_n\right) = \left[|G| \cdot \text{wp}[c_1, D]\left(\inf g_n\right) + \neg|G| \cdot \text{wp}[c_2, D]\left(\inf g_n\right)\right]
\]

\(\{c_1\}\) \([p]\) \(\{c_2\}\):

\[
\text{wp}[\{c_1\}[p] \{c_2\}, D]\left(\sup f_n\right) = \left(p \cdot \text{wp}[c_1, D]\left(\sup f_n\right) + (1 - p) \cdot \text{wp}[c_2, D]\left(\sup f_n\right)\right)
\]

And

\[
\text{wp}[\{c_1\}[p] \{c_2\}, D]\left(\inf g_n\right) = \left(p \cdot \text{wp}[c_1, D]\left(\inf g_n\right) + (1 - p) \cdot \text{wp}[c_2, D]\left(\inf g_n\right)\right)
\]

\(c_1; c_2:\)

\[
\text{wp}[c_1; c_2, D]\left(\sup f_n\right) = \text{wp}[c_1, D]\left(\text{wp}[c_2, D]\left(\sup f_n\right)\right)
\]

And

\[
\text{wp}[c_1; c_2, D]\left(\inf g_n\right) = \text{wp}[c_1, D]\left(\text{wp}[c_2, D]\left(\inf g_n\right)\right)
\]
call $P$:

$$\text{wp}[^{\text{call}}_n P, \vartheta] \left( \sup_{n} f_n \right) = \sup_{k} \text{wp}[^{\text{call}}_n P \left( \sup_{n} f_n \right)$$

and

$$\text{wp}[^{\text{call}}_n P, \vartheta] \left( \inf_{n} g_n \right) = \inf_{n} \text{wp}[^{\text{call}}_n P \left( \inf_{n} g_n \right)$$

Since $[^{\text{call}}_n P$ is call–free for every $n$ and we have already proven continuity for all call–free programs, we have

$$\text{wp}[^{\text{call}}_n P \left( \sup_{n} f_n \right) = \sup_{n} \text{wp}[^{\text{call}}_n P \left( f_n \right)$$

and

$$\text{wp}[^{\text{call}}_n P \left( \inf_{n} g_n \right) = \inf_{n} \text{wp}[^{\text{call}}_n P \left( g_n \right)$$

for every $n$ and hence

$$\text{wp}[^{\text{call}}_n P, \vartheta] \left( \sup_{n} f_n \right) = \sup_{n} \text{wp}[^{\text{call}}_n P \left( f_n \right)$$

and

$$\text{wp}[^{\text{call}}_n P, \vartheta] \left( \inf_{n} g_n \right) = \inf_{n} \text{wp}[^{\text{call}}_n P \left( g_n \right)$$

Proof of Monotonicity. Assume $f_1 \leq f_2$. Then

$$\text{wp}[^{\text{call}}_n P \left( \sup_{n} f_1 \right) = \sup_{n} \text{wp}[^{\text{call}}_n P \left( f_1 \right)$$

which implies $\text{wp}[^{\text{call}}_n P \left( f_1 \right) \leq \text{wp}[^{\text{call}}_n P \left( f_2 \right)$, and

$$\text{wp}[^{\text{call}}_n P \left( f_1 \right) = \sup_{n} \text{wp}[^{\text{call}}_n P \left( f_1 \right)$$

which implies $\text{wp}[^{\text{call}}_n P \left( f_1 \right) \leq \text{wp}[^{\text{call}}_n P \left( f_2 \right)$.

Proof of Linearity. We prove linearity by induction on the program structure. For the base cases we have:

**skip:**

$$\text{wp}[\text{skip}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2$$

$$= \alpha_1 \cdot \text{wp}[\text{skip}, \vartheta] \left( f_1 \right) + \alpha_2 \cdot \text{wp}[\text{skip}, \vartheta] \left( f_2 \right)$$

**x := E:**

$$\text{wp}[x := E, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \cdot \alpha \cdot \text{wp}[x := E, \vartheta] \left( f_1 \right) + \alpha_2 \cdot \text{wp}[x := E, \vartheta] \left( f_2 \right)$$

**abort:**

$$\text{wp}[\text{abort}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= 0$$

$$= \alpha_1 \cdot 0 + \alpha_2 \cdot 0$$

$$= \alpha_1 \cdot \text{wp}[\text{abort}, \vartheta] \left( f_1 \right) + \alpha_2 \cdot \text{wp}[\text{abort}, \vartheta] \left( f_2 \right)$$

For the induction hypothesis we assume that for any two programs $c_1$ and $c_2$ linearity holds. Then we can perform the induction step:

if $(G \{ c_1 \})$ else $(c_2)$:

$$\text{wp}[if (G \{ c_1 \}) \text{ else } (c_2), \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= \left\{ \begin{array}{l}
\text{wp}[G \{ c_1 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) + \left\{ \begin{array}{l}
\text{wp}[G \{ c_2 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)
\end{array} \right.
\end{array} \right.$$

$$= \left\{ \begin{array}{l}
\text{wp}[G \{ c_1 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) + \left\{ \begin{array}{l}
\text{wp}[G \{ c_2 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)
\end{array} \right.
\end{array} \right.$$

$$= \alpha_1 \cdot \text{wp}[G \{ c_1 \}, \vartheta] \left( f_1 \right) + \alpha_2 \cdot \text{wp}[G \{ c_2 \}, \vartheta] \left( f_2 \right)$$

$$\text{wp}[c_1] \{ p \} \{ c_2 \}$$

$$= \left\{ \begin{array}{l}
\text{wp}[c_1 \{ p \} \{ c_2 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) + \left\{ \begin{array}{l}
\text{wp}[c_2 \{ p \} \{ c_2 \}, \vartheta] \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right) \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)
\end{array} \right.
\end{array} \right.$$

$$= \alpha_1 \cdot \text{wp}[c_1 \{ p \} \{ c_2 \}, \vartheta] \left( f_1 \right) + \alpha_2 \cdot \text{wp}[c_2 \{ p \} \{ c_2 \}, \vartheta] \left( f_2 \right)$$

Since $[^{\text{call}}_n P$ is call–free for every $n$ and we have already proven linearity for all call–free programs, we have

$$\text{wp}[[^{\text{call}}_n P \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= \alpha_1 \cdot \text{wp}[[^{\text{call}}_n P \left( f_1 \right) + \alpha_2 \cdot \text{wp}[[^{\text{call}}_n P \left( f_2 \right)$$

for every $n$ and hence

$$\sup_{n} \text{wp}[[^{\text{call}}_n P \left( \alpha_1 \cdot f_1 + \alpha_2 \cdot f_2 \right)$$

$$= \sup_{n} \alpha_1 \cdot \text{wp}[[^{\text{call}}_n P \left( f_1 \right) + \sup_{n} \alpha_2 \cdot \text{wp}[[^{\text{call}}_n P \left( f_2 \right)$$

$$= \alpha_1 \cdot \text{wp}[[^{\text{call}}_n P \left( f_1 \right) + \alpha_2 \cdot \text{wp}[[^{\text{call}}_n P \left( f_2 \right)$$

Proof of Preservation of 0 and 1. We prove preservation of 0 and 1 by induction on the program structure. For the base cases we have:

**skip:**

$$\text{wp}[\text{skip}, \vartheta] \left( 0 \right) = 0$$

and

$$\text{wp}[\text{skip}, \vartheta] \left( 1 \right) = 1$$
Reasoning about Recursive Probabilistic Programs

For the induction hypothesis we assume that for any two programs \( c_1 \) and \( c_2 \) preservation of 0 and 1 holds. Then we can perform the induction step:

\[\text{if } (G) \{ c_1 \} \text{ else } \{ c_2 \}:\]

\[
wp*[\{c_1\} \text{ else } \{c_2\}](0) = \wp[c_1, \mathcal{E}](0) + \wp[c_2, \mathcal{E}](0) = 0
\]

\[
wp*[\{c_1\} \text{ else } \{c_2\}](1) = \wp[c_1, \mathcal{E}](1) + \wp[c_2, \mathcal{E}](1) = 1
\]

\[c_1; c_2:\]

\[
wp[c_1; c_2, \mathcal{E}](0) = \wp[c_1, \mathcal{E}](wp[c_2, \mathcal{E}](0)) = \wp[c_1, \mathcal{E}](0) = 0
\]

\[
wp[c_1; c_2, \mathcal{E}](1) = \wp[c_1, \mathcal{E}](wp[c_2, \mathcal{E}](1)) = \wp[c_1, \mathcal{E}](1) = 1
\]

\[\text{call } P:\]

\[
wp[\text{call } P, \mathcal{E}](0) = \sup_n \wp[\text{call}^n_P, \mathcal{E}](0)
\]

\[
wp[\text{call } P, \mathcal{E}](1) = \inf_n \wp[\text{call}^n_P, \mathcal{E}](1)
\]

Since \( \text{call}^n_P \) is call-free for every \( n \) and we have already proven preservation of 0 and 1 for all call-free programs, we have

\[
wp[\text{call}^n_P, \mathcal{E}](0) = 0
\]

and

\[
wp[\text{call}^n_P, \mathcal{E}](1) = 1
\]

for every \( n \) and hence

\[
wlp[\text{call } P, \mathcal{E}](0) = \sup_n \wp[\text{call}^n_P, \mathcal{E}](0) = 0
\]

and

\[
wlp[\text{call } P, \mathcal{E}](1) = \inf_n \wp[\text{call}^n_P, \mathcal{E}](1) = 1
\]

\[\Box\]

\[\text{A.2 Fixed Point Characterization of Recursive Procedures}\]

Establishing the results from Theorem 3.1 requires a subsidiary result connecting \( \wp[\text{call } P, \mathcal{E}] \) with \( \wp[\text{call } P, \mathcal{E}]^\uparrow \) in the presence of non–recursive procedure calls.

**Lemma A.1.** For every command \( c \) and closed command \( c' \),

\[
\wp[\text{call}^n_P, c'](f) = \wp[\text{call}^n_P, c'](f)
\]

\[\Box\]

**Proof.** By induction on the structure of \( c \). Except for procedure calls, the proof for all other program constructs follows immediately from definition of \( \wp[\cdot] \), \( \wp[\cdot]^\uparrow \), and the inductive hypotheses in the case of compound instructions. For the case of procedure calls, the proof relies on the fact that as \( c' \) is a closed command, \( \text{call}^n_P \circ c' \) is \( c' \) for all \( n \geq 1 \). Concretely, we reason as follows:

\[
wp[\text{call}^n_P, c'](f) = \wp[\text{call}^n_P, c'](f)
\]

\[\Box\]

Now we are in a position to prove Theorem 3.1. Consider first the case of fixed point characterization

\[
wp[\text{call } P, \mathcal{E}] = \lfp_{\mathcal{E}} \left( \lambda \theta : \text{Env}. \wp[P, \mathcal{E}]^{\uparrow}_\theta \right).
\]

Its proof comprises two major steps:

1. **Use the continuity of \( F_\mathcal{E} : (\text{Env}, \subseteq) \to (\text{Env}, \subseteq) \) established by Lemma A.6 to conclude that**

\[
lfp_{\mathcal{E}}(F) = \sup_n F^n(\bot_{\text{Env}}),
\]

where \( F^n \) denotes the composition of \( F \) with itself \( n \) times (i.e., \( F^0 = id \) and \( F^{n+1} = F \circ F^n \)) and \( \bot_{\text{Env}} = \lambda f : \mathcal{E}. 0 \) is the constantly 0 environment.

2. **Show that**

\[
\forall f : \mathcal{E}. F^n(\bot_{\text{Env}})(f) = \wp[\text{call}^n_P, f]
\]

for all \( n \geq 0 \).

Then the proof follows immediately since by definition of \( \wp \), we have

\[
\Box\]
wp[call P, D] (f) = sup_n wp[call^n P] (f) 
= sup_n F^n (⊥_{SEnv}) (f) = lfp (F) (f) .

We now consider each of these two steps in details. Step 1 follows immediately from an application of Kleene’s Fixed Point Theorem. Step 2 proceeds by induction on n. The base case is straightforward:

\[ F^0 (⊥_{SEnv}) (f) = ⊥_{SEnv} (f) = 0 \]

= wp[abort] (f) = wp[call^n P] (f) .

For the inductive case we have

\[ F^{n+1} (⊥_{SEnv}) (f) = \{ \text{def. of } F^{n+1} \} \]

\[ F(F^n (⊥_{SEnv})) (f) = \{ \text{def. of } F \} \]

\[ wp[D(P)]^{\geq n} (⊥_{SEnv}) (f) = \{ \text{I.H.} \} \]

\[ wp[D(P)]^{\geq n} wp[call P] (f) = \{ \text{Lemma A.1} \} \]

\[ wp[D(P), P ⊡ call^n P] (f) = \{ \text{Lemma A.4} \} \]

\[ wp[D(P)]^{\geq n} [call P] (f) = \{ \text{def. n-inl.} \} \]

\[ wp[call^{n+1} P] (f) \]

Now we turn to the fixed point characterization

\[ wp[call P, D] = gfp (\theta : \text{LSEnv. } wp[D(P)]^{\geq n}) . \]

The proof follows a dual argument. We first apply Kleene’s Fixed Point Theorem to show that

\[ gfp (G) = \inf_n G^n (⊥_{SEnv}) , \]

where \( ⊥_{SEnv} = \lambda f : E \leq 1, 1 \) is the constantly 1 environment. Next we show by induction on n that

\[ \forall f : E \leq 1, G^n (⊥_{SEnv}) (f) = wp[call^n P] (f) \]

The proof concludes combining these two results since

\[ wp[call P, D] (f) = \inf_n wp[call^n P] (f) \]

= \inf_n G^n (⊥_{SEnv}) (f) = gfp (G) (f) . \]

**Lemma A.2.** [32, p. 127] Suppose \( a_{n,m} \) are elements of upper \( ω\)-cpo \( (A, ≤) \) with the property that \( a_{n,m} ≤ a_{n′,m′} \) whenever \( n ≤ n′ \) and \( m ≤ m′ \). Then,

\[ \sup_n (\sup_n a_{n,m}) = \sup_n (\sup_n a_{n,m}) = sup_1 a_{n,m} . \]

**Lemma A.3** (Monotone Sequence Theorem). If \( (a_n) \) is a monotonic increasing sequence in a closed interval \([L, U] \subseteq [0, +∞], \)

then the supremum \( \sup_n a_n \) coincides with \( \lim_{n \to +∞} a_n \). Dually, if \( (a_n) \) is a monotonic decreasing sequence in a closed interval \([L, U] \subseteq [0, +∞], \)

the infimum \( \inf_n a_n \) coincides with \( \lim_{n \to +∞} a_n \). 

**A.3 Soundness of \( wp[\text{lp}] \) Rules**

**Fact A.1.** To carry on the proofs we use the fact that from

\[ wp[\text{lp}] [call P] (f_1) \Join g_1 \vdash wp[\text{lp}] [c] (f_2) \Join g_2 , \]

it follows that for all environment \( D^* \),

\[ wp[\text{lp}] [call P, D^*] (f_1) \Join g_1 \Rightarrow wp[\text{lp}] [c, D^*] (f_2) \Join g_2 . \]

We provide detailed proofs for rules \([wp-\text{rec}]\) and \([wp-\text{rec}_m]\); the proof of rules \([wp-\text{rec}]\) and \([wp-\text{rec}_m]\) follows a dual argument.

**Soundness of rule** \([wp-\text{rec}]\). Since by definition, \( wp[\text{call P, D}] (f) = sup_n wp[call^n P, D] (f) \), to establish the conclusion of the rule it suffices to show that

\[ ∀ n, wp[call^n P] (f) ≤ g , \]

which we do by induction on n. The base case is immediate since \( call^n P = \text{abort} \) and \( wp[\text{abort}] (f) = 0 \). For the inductive case, we reason as follows:

\[ wp[call^{n+1} P] (f) ≤ g \quad \{ \text{def. n-inl.} \} \]

\[ ⇒ wp[D(P)] [call P, call^n P] (f) ≤ g \quad \{ \text{Lemma A.4} \} \]

\[ ⇒ wp[D(P), P ⊡ call^n P] (f) ≤ g \quad \{ \text{rule prem. Fact A.1} \} \]

\[ ⇒ wp[D(P)] [call P, call^n P] (f) ≤ g \quad \{ \text{Lemma A.4} \} \]

\[ ⇒ wp[call P, call^n P] (f) ≤ g \quad \{ \text{def. subst.} \} \]

\[ ⇒ wp[call^n P] (f) ≤ g \quad \{ \text{I.H.} \} \]

**Soundness of rule** \([wp-\text{rec}_m]\). We prove that the rule’s premises entail \( l_n ≤ wp[call^n P] (f) ≤ u_n \) for all \( n \in N \). The conclusion of the rule then follows immediately by taking the supremum over \( n \) on the three sides of the equation. We proceed by induction on n. The base case is trivial since by definition, \( wp[call^n P] (f) = wp[\text{abort}] (f) = 0 \) and by the rule’s premise, \( l_0 = u_0 = 0 \). For the inductive case we reason as follows:

\[ l_{n+1} ≤ wp[call^{n+1} P] (f) ≤ u_{n+1} \quad \{ \text{def. n-inl.} \} \]

\[ ⇒ wp[D(P)] [call P, call^n P] (f) ≤ u_{n+1} \quad \{ \text{Lemma A.4} \} \]

\[ ⇒ wp[D(P)] [call P, call^n P] (f) ≤ u_{n+1} \quad \{ \text{rule prem. Fact A.1} \} \]

\[ l_{n+1} ≥ wp[D(P)] [call P, call^n P] (f) ≤ u_{n+1} \quad \{ \text{Lemma A.4} \} \]

\[ l_{n+1} ≥ wp[D(P)] [call P, call^n P] (f) ≤ u_{n+1} \quad \{ \text{I.H.} \} \]

true

**A.4 Substitution of Procedure Calls**

\[
\begin{array}{c|c|c}
\text{c} & \text{c[call P / c']} & \text{wp[ \text{c[call P / c']} ]} \\
\hline
\text{skip} & \text{skip} & \text{wp[ \text{c[call P / c']} ]} \\
\text{x := E} & \text{x := E} & \text{wp[ \text{c[call P / c']} ]} \\
\text{abort} & \text{abort} & \text{wp[ \text{c[call P / c']} ]} \\
\text{call P} & \text{c'} & \text{wp[ \text{c[call P / c']} ]} \\
\text{if (G) \{c_1\} else \{c_2\} } & \text{if (G) \{c_1[call P / c']\} else \{c_2[call P / c']\} } & \text{wp[ \text{c[call P / c']} ]} \\
\text{\{c_1\} \[p\] \{c_2\} } & \text{\{c_1[call P / c’] \} \[p\] \{c_2[call P / c’] \} } & \text{wp[ \text{c[call P / c']} ]} \\
\text{\{c_1\} \{c_2\} } & \text{\{c_1[call P / c’] \} \{c_2[call P / c’] \} } & \text{wp[ \text{c[call P / c']} ]} \\
\end{array}
\]

**Figure 7.** Syntactic replacement of procedure calls.

**Lemma A.4.** For every command c and closed command c’,

\[ \text{wp[ \text{c[call P / c']} ]} = \text{wp[ \text{c, P ⊡ c’} ]} . \]

**Proof.** By induction on the structure of c. Except for procedure calls, the proof for all other program constructs follows from de definition of wp and some simple calculations (and the inductive hypotheses in the case of compound instructions). For the case of...
procedure calls, the proof relies on the fact that as \( c' \) is a closed command, \( \text{call}^{ \omega \rightarrow c'} \mathcal{P} = c' \) for all \( n \geq 1 \). Concretely, we reason as follows:

\[
\begin{align*}
\text{wp}[\text{call} \mathcal{P} [\text{call} / c']](f) &= \{\text{def. subst.}\} \\
\text{wp}[c'](f) &= \{\text{sup. of a constant sequence}\} \\
\sup_n \text{wp}[c'](f) &= \{\text{observation above}\} \\
\sup_n \text{wp} \left[ \text{call}^{ \omega \rightarrow c'} \mathcal{P} \right](f) &= \{\text{wp}[\text{call}^{ \omega \rightarrow c'} \mathcal{P}](f) = 0\} \\
\sup_n \text{wp} \left[ \text{call}^{ \omega \rightarrow c'} \mathcal{P} \right](f) &= \{\text{wp}[\text{call} \mathcal{P} [\text{call} / c']](f)\} \\
\text{wp}[\text{call} \mathcal{P}, \mathcal{P} \triangleright c'] &= \{\text{def. wp[\cdot]}\} \\
\end{align*}
\]

A.5 Continuity of Transformer \( \text{wp}[\cdot]_\theta^\mathcal{P} \)

<table>
<thead>
<tr>
<th>( c )</th>
<th>( \text{wp}[c]_\theta^\mathcal{P} (f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{skip}</td>
<td>( f )</td>
</tr>
<tr>
<td>( x := E )</td>
<td>( f[x/E] )</td>
</tr>
<tr>
<td>\text{abort}</td>
<td>( 0 )</td>
</tr>
<tr>
<td>if (( G )) { ( c_1 ) } else { ( c_2 ) }</td>
<td>( [\mathcal{G}] \cdot \text{wp}[c_1]<em>\theta^\mathcal{P} (f) + [-\mathcal{G}] \cdot \text{wp}[c_2]</em>\theta^\mathcal{P} (f) )</td>
</tr>
<tr>
<td>{ ( c_1 ) } ( p ) { ( c_2 ) }</td>
<td>( p \cdot \text{wp}[c_1]<em>\theta^\mathcal{P} (f) + (1 - p) \cdot \text{wp}[c_2]</em>\theta^\mathcal{P} (f) )</td>
</tr>
<tr>
<td>\text{call} \mathcal{P}</td>
<td>( \theta(f) )</td>
</tr>
<tr>
<td>( c_1 \triangleright c_2 )</td>
<td>( \text{wp}[c_1]<em>\theta^\mathcal{P} (\text{wp}[c_2]</em>\theta^\mathcal{P} (f)) )</td>
</tr>
</tbody>
</table>

\( \text{wp}[c]_\theta^\mathcal{P} (f) \) differs from \( \text{wp}[c]_\theta^\mathcal{P} \) only in abort instructions.

As a preliminary step to discuss the continuity of \( \text{wp}[\cdot]_\theta^\mathcal{P} \), we observe that order relation \( \sqsubset \) (see paragraph below Theorem 3.1) endows the set of environments \( \mathcal{E} \) with the structure of an upper \( \omega \)-cpo with bottom element \( \bot_{\mathcal{E}} = \lambda f : \mathcal{E} : 0 \), where the supremum of an increasing \( \omega \)-chain \( \theta_0 \sqsubset \cdots \sqsubset \theta_1 \sqsubset \cdots \) is given pointwise, i.e. \( \sup_n \theta_i(f) = \sup_n \theta_i(f) \). Likewise, \( \sqsubset \) endows the set of liberal environments \( \mathcal{LSEnv} \) with the structure of a lower \( \omega \)-cpo with top element \( \top_{\mathcal{LSEnv}} = \lambda f : \mathcal{LSEnv} : \mathbb{E}_1 : 1 \), where the infimum of a decreasing \( \omega \)-chain \( \theta_0 \sqsupset \cdots \sqsupset \theta_1 \sqsupset \cdots \) is given pointwise, i.e. \( \inf_n \theta_i(f) = \inf_n \theta_i(f) \).

We will discuss two kind of continuity results for \( \text{wp}[\cdot]_\theta^\mathcal{P} \).

First, we show that for every environment \( \theta \), expectation transformer \( \text{wp}[\cdot]_\theta^\mathcal{P} \) is continuous, or equivalently, that

\[
\text{wp}[c]_\theta^\mathcal{P} : (\mathcal{E}) \rightarrow (\mathcal{E})
\]

\[
\text{wp}[c]_\theta^\mathcal{P} : (\mathcal{LSEnv}) \rightarrow (\mathcal{LSEnv})
\]

This result will be established in Lemma A.5. Second, we show that the above environment transformers are themselves continuous, i.e. that

\[
\text{wp}[c]_\theta^\mathcal{P} : (\mathcal{E}) \rightarrow (\mathcal{E})
\]

\[
\text{wp}[c]_\theta^\mathcal{P} : (\mathcal{LSEnv}) \rightarrow (\mathcal{LSEnv})
\]

This result will be established in Lemma A.6.

Lemma A.5. Let \( \theta \in \mathcal{SEnv} \) and \( f_0 \preceq f_1 \preceq \cdots \) be an ascending \( \omega \)-chain of expectations in \( \mathbb{E} \). Then for every command \( c \),

\[
\text{wp}[c]_\theta^\mathcal{P} (\sup_n f_n) = \sup_n \text{wp}[c]_{\theta}^\mathcal{P} (f_n) .
\]

Analogously, if \( f_0 \succeq f_1 \succeq \cdots \) is a descending \( \omega \)-chain of expectations in \( \mathbb{E}_\leq, 1 \),

\[
\text{wp}[c]_{\inf_n} (f_n) = \inf_n \text{wp}[c]_{\theta}^\mathcal{P} (f_n) .
\]

Proof. By induction on the structure of \( c \). Except for procedure calls, all program constructs use the same proof argument as for the continuity of plain transformer \( \text{wp}[\cdot], \) which has already been dealt with in e.g. [11]. For procedure calls we reason as follows.

\[
\text{wp}[\text{call} \mathcal{P} \triangleright (\sup_n f_n)] = \{\text{def. wp[\cdot]}\}
\]

\[
\theta (\sup_n f_n) = \{\text{\( \theta \) is continuous by hypothesis}\}
\]

\[
\sup_n \theta(f_n) = \{\text{def. wp[\cdot]}\}
\]

\[
\sup_n \text{wp} \left[ \text{call}^{ \omega \rightarrow c'} \mathcal{P} \right](f_n) .
\]

The reasoning to show that

\[
\text{wp}[\text{call} \mathcal{P} \triangleright (\inf_n f_n)] = \inf_n \text{wp}[\text{call} \mathcal{P} \triangleright (f_n)]
\]

is analogous.

Lemma A.6. Let \( \theta_0 \sqsubset \theta_1 \sqsubset \cdots \) be an ascending \( \omega \)-chain in \( \mathcal{SEnv} \). Then for every command \( c \),

\[
\text{wp}[c]_{\sup_n} = \sup_n \text{wp}[c]_{\theta}^\mathcal{P} .
\]

Analogously, if \( \theta_0 \sqsupset \theta_1 \sqsupset \cdots \) is a descending \( \omega \)-chain in \( \mathcal{LSEnv} \),

\[
\text{wp}[c]_{\inf_n} = \inf_n \text{wp}[c]_{\theta}^\mathcal{P} .
\]

Proof. By induction on the structure of \( c \). We consider only the case of \( \text{wp}[c]_{\sup_n} \); the case of \( \text{wp}[c]_{\inf_n} \) is analogous. For the three basic instructions \( c = \text{skip}, \ c = x := E \) and \( c = \text{abort} \) the proof is straightforward since the action of transformer \( \text{wp}[\cdot]_\theta^\mathcal{P} \) on these instructions is independent of the semantic environment at stake (i.e. constant functions are always continuous). For the remaining program constructs we reason as follows:

**Procedure Call:**

\[
\text{wp}[\text{call} P_{\sup_n} (f_n)] = \{\text{def. wp[\cdot]}\}
\]

\[
\sup_n \theta(f_n) = \{\text{def. sup_n \theta_n}\}
\]

\[
\sup_n \text{wp} \left[ \text{call}^{ \omega \rightarrow c'} \mathcal{P} \right](f_n) .
\]
Sequential Composition:

\[
\begin{align*}
\text{wp}[c_1; c_2]_{\sup_n} &= \{ \text{def. wp}[c_2]_{\sup_n} \} \\
\text{wp}\{c_1\cdot c_2\}_{\sup_n} &= \{ \text{IH on } c_2 \} \\
\text{wp}[c_1]_{\sup_n} \eta_n &\leq \text{wp}[c_2]_{\sup_n} \eta_n (f) \quad (\text{Lemma A.5}) \\
\sup_n \text{wp}[c_1]_{\sup_n} \eta_n (wp[c_2]_{\sup_n} \eta_n (f)) &= \{ \text{Lemma A.2} \} \\
\sup_n \text{wp}[c_1]_{\sup_n} \eta_n (wp[c_2]_{\sup_n} \eta_n (f)) &= \{ \text{def. wp}[c_2]_{\sup_n} \} \\
\sup_n \text{wp}[c_1; c_2]_{\sup_n} (f) &= \{ \text{Lemma A.8} \} \\
\end{align*}
\]

For applying Lemma A.2 in step (4) we have to show that

\[
\text{wp}[c_1]_{\sup_n} (wp[c_2]_{\sup_n} \eta_n (f)) \leq \text{wp}[c_1]_{\sup_n} (wp[c_2]_{\sup_n} \eta_n (f)) \text{ whenever } n \leq n' \text{ and } m \leq m'.
\]

To prove Equation (1) we first apply the I.H. on \(c_2\). Since continuity entails monotonicity, we obtain \(\text{wp}[c_2]_{\sup_n} \eta_n \subset \text{wp}[c_2]_{\sup_n} \eta_n\), which itself gives \(\text{wp}[c_2]_{\sup_n} \eta_n\). We are left to show that \(\text{wp}[c_1]_{\sup_n} \eta_n\) is monotonic, which follows by its continuity guaranteed by Lemma A.5. To prove Equation (2), we apply the I.H. on \(c_1\). Again, since the continuity of \(\text{wp}[c_1]_{\sup_n} \eta_n\) implies its monotonicity, we obtain \(\text{wp}[c_1]_{\sup_n} \eta_n \subset \text{wp}[c_1]_{\sup_n} \eta_n\), which establishes Equation (2).

Conditional Branching:

\[
\begin{align*}
\text{wp}[\text{call if } G \{ c_1 \} \text{ else } c_2]_{\sup_n} &= \{ \text{def. wp}[c_2]_{\sup_n} \} \\
\text{wp}[G \cdot \text{wp}[c_1]_{\sup_n} \eta_n (f)] &= \{ \text{IH on } c_1, c_2 \} \\
\text{wp}[G \cdot \text{wp}[c_1]_{\sup_n} \eta_n (f)] &= \{ \text{Lemma A.3} \} \\
\text{wp}[G \cdot \text{wp}[c_1]_{\sup_n} \eta_n (f)] &= \{ \text{Lemma A.3} \} \\
\sup_n \{ \text{wp}[G \cdot \text{wp}[c_1]_{\sup_n} \eta_n (f)]\} &= \{ \text{Lemma A.3} \} \\
\end{align*}
\]

A.6 Basic Properties of Transformer \(\text{ert}\)

We begin by presenting some preliminary results that will be necessary for establishing the main results about the \(\text{ert}\) transformer.

Fact A.2 ([\(\text{RtEnv}, \sqsubseteq\))] is an \(\omega\)-cpo). Let \(\sqsubseteq\) denote the pointwise order between runtime environments, i.e., for \(\eta_1, \eta_2 \in \text{RtEnv}\), \(\eta_1 \sqsubseteq \eta_2\) iff \(\eta_1(t) \leq \eta_2(t)\) for every \(t \in T\). Relation \(\sqsubseteq\) endows the set of runtime environments \(\text{RtEnv}\) with the structure of an upper \(\omega\)-cpo with bottom \(\bot\).

Lemma A.7 (Continuity of \(\text{ert}\[\cdot\]_{\sup_n}\). Let \(\eta_0 \sqsubseteq \eta \implies \) be an ascending \(\omega\)-chain in \(\text{RtEnv}\). Then for every command \(c\),

\[
\text{ert}[c]_{\sup_n} \eta_n = \sup_n \text{ert}[c]_{\sup_n} \eta_n.
\]

Proof. The proof follows the same argument as that for establishing the continuity of transformer \(wp\) (see Lemma A.6).

Lemma A.8 ([\(\text{ert}\[\cdot\]_{\sup_n}\). For every command \(c\) and every (upper continuous) runtime environment \(\eta \in \text{RtEnv}\), \(\text{ert}[c]_{\sup_n}\) is a continuous runtime transformer in \(T\) \(\text{wp}^\omega\).

Proof. By induction on the program structure. For every program constructs different from a procedure call, the reasoning is similar to that used in Lemma A.5 to prove the same property for \(wp[c]_{\sup_n}\). For a procedure call the statement follows immediately since \(\eta\) is continuous by hypothesis.

Lemma A.9 (Alternative characterization of \(\text{ert}\[\cdot\]_{\sup_n}\). Let \(F(\eta) = \bot \implies \text{ert}[\cdot]_{\sup_n} F^n(\bot_{\text{RtEnv}})\), where \(\bot_{\text{RtEnv}} = \exists: T, 0\) and \(F^n(\bot_{\text{RtEnv}})\) denotes the repeated application of \(F\) from \(\bot_{\text{RtEnv}}\) \(n\) times (i.e. \(F^0(\bot_{\text{RtEnv}}) = \text{id} = F^n(\bot_{\text{RtEnv}})\)).

Proof. Using Lemma A.7 one can show that \(F\) is an (upper) continuous runtime transformer. The result then follows from a direct application of Kleene’s Fixed Point Theorem and Fact A.2.

To present the following lemma we use the notion of expanding runtime environments. Given \(\eta_0, \eta, \theta \in \text{RtEnv}\) and \(k, \Delta \in \mathbb{R}_{\geq 0}\) we say that \(\eta_0, \eta, \theta\) are \((k, \Delta)\)-expanding iff \(t_0 - t_0 \geq k \cdot (1 - f) + \Delta\) implies \(\eta_1(t_0) - \eta_0(t_0) \geq k \cdot (1 - \theta(f)) + \Delta\) for all \(t_0, t_1 \in T\) and \(f \in \mathbb{R}_{\leq 1}\).

Lemma A.10. Let \(\langle \eta_0, \eta, \theta \rangle\) be \((k, \Delta)\)-expanding environments and \(c\) be an abort-free command. Then

\[
t_0 - t_0 \geq k \cdot (1 - f) + \Delta \implies \text{ert}[c]_{\sup_n} (t_1) - \text{ert}[c]_{\sup_n} (t_0) \geq k \cdot (1 - \text{wp}[c]_{\sup_n} (f)) + \Delta \text{ for all } t_0, t_1 \in T \text{ and } f \in \mathbb{R}_{\leq 1}.
\]

Proof. By induction on the structure of \(c\).

\[12\] See paragraph above.
No-op:
\[
\text{ert}[\text{skip}]^p_{\eta_1}(t_1) - \text{ert}[\text{skip}]^p_{\eta_0}(t_0) 
\geq k \cdot (1 - \text{wp}[\text{skip}]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{def. of } \text{ert}[\cdot]^p_{\eta_0}, \text{wp}[\cdot]^p_{\eta_0}\}
\]
\[
(1 + t_1) - (1 + t_0) \geq k \cdot (1 - f) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{hypothesis}\}
\]
true

Assignment:
\[
\text{ert}[x := E]^p_{\eta_1}(t_1) - \text{ert}[x := E]^p_{\eta_0}(t_0) 
\geq k \cdot (1 - \text{wp}[x := E]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{def. of } \text{ert}[\cdot]^p_{\eta_0}, \text{wp}[\cdot]^p_{\eta_0}\}
\]
\[
(1 + t_1)[x/E] - (1 + t_0)[x/E] \geq k \cdot (1 - f)[x/E] + \Delta
\]
\[
\Leftrightarrow \quad \{\text{algebra}\}
\]
\[
(t_1 - t_0)[x/E] \geq (k \cdot (1 - f) + \Delta)[x/E]
\]
\[
\Leftrightarrow \quad \{\text{hypothesis}\}
\]
true

Procedure Call:
\[
\text{ert}[\text{call } P]^p_{\eta_1}(t_1) - \text{ert}[\text{call } P]^p_{\eta_0}(t_0) 
\geq k \cdot (1 - \text{wp}[\text{call } P]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{def. of } \text{ert}[\cdot]^p_{\eta_0}, \text{wp}[\cdot]^p_{\eta_0}\}
\]
\[
\eta_1(t_1) - \eta_0(t_0) \geq k \cdot (1 - \theta(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\{\eta_1, \eta_0, \theta\} \text{ is } (k, \Delta)-\text{expanding}\}
\]
\[
t_1 - t_0 \geq k \cdot (1 - f) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{hypothesis}\}
\]
true

Probabilistic Choice:
\[
\text{ert}[\{c_1\} [p] \{c_2\}]^p_{\eta_1}(t_1) - \text{ert}[\{c_1\} [p] \{c_2\}]^p_{\eta_0}(t_0) 
\geq k \cdot (1 - \text{wp}[\{c_1\} [p] \{c_2\}]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{def. of } \text{ert}[\cdot]^p_{\eta_0}, \text{wp}[\cdot]^p_{\eta_0}\}
\]
\[
p \cdot (\text{ert}[\{c_1\}^p_{\eta_1}(t_1) - \text{ert}[\{c_1\}]^p_{\eta_0}(t_0))
\]
\[
+ (1 - p) \cdot (\text{ert}[\{c_2\}^p_{\eta_1}(t_1) - \text{ert}[\{c_2\}]^p_{\eta_0}(t_0))
\]
\[
\geq k \cdot (1 - p \cdot \text{wp}[\{c_1\}]^p_{\eta_0}(f) + (1 - p) \cdot \text{wp}[\{c_2\}]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{IH on } c_1, c_2\}
\]
\[
p \cdot (k \cdot (1 - \text{wp}[\{c_1\}]^p_{\eta_0}(f) + \Delta)
\]
\[
+ (1 - p) \cdot (k \cdot (1 - \text{wp}[\{c_2\}]^p_{\eta_0}(f) + \Delta)
\]
\[
\geq k \cdot (1 - p \cdot \text{wp}[\{c_1\}]^p_{\eta_0}(f) + (1 - p) \cdot \text{wp}[\{c_2\}]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{algebra (equality holds)}\}
\]
true

Conditional Branching: analogous to the case of probabilistic choice.

Sequential Composition:
\[
\text{ert}[c_1 ; c_2]^p_{\eta_1}(t_1) - \text{ert}[c_1 ; c_2]^p_{\eta_0}(t_0) 
\geq k \cdot (1 - \text{wp}[c_1 ; c_2]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{def. of } \text{ert}[\cdot]^p_{\eta_0}, \text{wp}[\cdot]^p_{\eta_0}\}
\]
\[
\text{ert}[c_1]^p_{\eta_1}(t_1) - \text{ert}[c_1]^p_{\eta_0}(t_0)
\geq k \cdot (1 - \text{wp}[c_1]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{IH on } c_1\}
\]
\[
\text{ert}[c_2]^p_{\eta_1}(t_1) - \text{ert}[c_2]^p_{\eta_0}(t_0)
\geq k \cdot (1 - \text{wp}[c_2]^p_{\eta_0}(f)) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{IH on } c_2\}
\]
\[
t_1 - t_0 \geq k \cdot (1 - f) + \Delta
\]
\[
\Leftrightarrow \quad \{\text{hypothesis}\}
\]
true

Lemma A.11. Let P be an abort–free procedure with declaration D. Then for every runtime t,
\[
\text{ert}[\text{call } P](t) \geq \sup_n n \cdot (1 - \text{wp}[\text{call } P, D](1)) .
\]

Proof. Let \( F(\eta) = 1 \oplus \text{ert}[\eta(P)]^p_{\eta_0} \). Since by Lemma A.9, \( \text{ert}[\text{call } P, D] = \sup_n F^n(\bot) \), the result follows from showing that for all \( n \geq 0 \),
\[
F^{n+1}(\bot)(t) \geq (n + 1) \cdot (1 - \text{wp}[\text{call } P, D](1)) .
\]
To establish this, we first prove by induction on i that whenever \( t_1 - t_0 \geq 0 \),
\[
F^{i+1}(\bot)(t_1) - F^i(\bot)(t_0) \geq 1 - \text{wp}[\text{call } P, D](1) ,
\]
and then conclude using a telescopic sum argument as follows:
\[
F^{n+1}(\bot)(t) = F^0(\bot)(t) + \sum_{i=0}^n F^{i+1}(\bot)(t) - F^i(\bot)(t)
\]
\[
\geq \sum_{i=0}^n (n + 1) \cdot (1 - \text{wp}[\text{call } P, D](1)) .
\]
For the inductive proof we reason as follows. For the base case we have
\[
F^1(\bot)(t_1) - F^0(\bot)(t_0) \geq 1 - \text{wp}[\text{call } P, D](1)
\]
\[
\Leftrightarrow \quad \{\text{def. of } F^n, \bot\}
\]
\[
1 + \text{ert}[\eta(P)]^p_{\eta_1}(t_1) - \bot(t_0) \geq 1 - \text{wp}[\text{call } P, D](1)
\]
\[
\Leftrightarrow \quad \{\text{def. of } \bot\}
\]
\[
1 + \text{ert}[\eta(P)]^p_{\eta_1}(t_1) \geq 1 - \text{wp}[\text{call } P, D](1)
\]
\[
\Leftrightarrow \quad \{\text{wp}[\text{call } P, D](1) \geq 0\}
\]
true
while for the inductive case we have,
\[ F^{i+2}(\bot)(t_1) - F^{i+1}(\bot)(t_0) \geq 1 - \wp[c\{P, D\}](1) \]
\[ \vdash \text{def. of } F^n \]
\[ (1 + \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_1)) - (1 + \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_0)) \]
\[ \geq 1 - \wp[c\{P, D\}](1) \]
\[ \vdash \text{lemma} \]
\[ \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_1) - \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_0) \]
\[ \geq 1 \cdot (1 - \wp[c\{P, D\}](1)) + 0 \]
\[ \vdash \text{lemma} \]
\[ \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_1) - \text{ert}[D(P)]_{F^{i+1}(\bot)}(t_0) \]
\[ \geq 1 \cdot (1 - \wp[D(P)]_{\text{call}[P, \pi]}(1)) + 0 \]
\[ \vdash \text{lemma} \]
\[ t_1 - t_0 \geq 1 \cdot (1 - 1) + 0 \]
\[ \langle F^{i+1}(\bot), F^i(\bot), \wp[c\{P, D\}] \rangle \text{ are } \langle 1, 0 \rangle - \text{expanding} \]
\[ \vdash \text{lemma} \]
\[ \langle F^{i+1}(\bot), F^i(\bot), \wp[c\{P, D\}] \rangle \text{ are } \langle 1, 0 \rangle - \text{expanding} \]
\[ \vdash \text{IH} \]
\[ \text{true} \]

Lemma A.12. For every constant \( k \in \mathbb{R}_{\geq 0} \) and abort–free program \( \langle c, D \rangle \),
\[ \text{ert}[c, D](\{k\}) \geq k. \]

Proof. By induction on the structure of \( c \). Except for the case of procedure calls, all other program constructs pose no difficulty. For the case of a procedure call, we make a case distinction on the termination behaviour of the procedure. If from state \( s \) the procedure terminates almost surely, i.e. \( \wp[c\{P, D\}](1)(s) = 1 \), the result follows from Theorem 5.2 and the linearity of \( \wp[\cdot] \) (see Lemma A.11) since
\[ \text{ert}[c\{P, D\}](\{k\})(s) = \text{ert}[c\{P, D\}](0)(s) + \wp[c\{P, D\}](\{k\})(s) \]
\[ \geq \wp[c\{P, D\}](\{k\})(s) \]
\[ = k \cdot \wp[c\{P, D\}](1)(s) = k \]
If, on the contrary, the procedure terminates with probability strictly less than 1 from state \( s \), we conclude applying Lemma A.11 since
\[ \text{ert}[c\{P, D\}](\{k\})(s) \]
\[ \geq \sup_{n_0} n \cdot (1 - \wp[c\{P, D\}](1)(s)) \]
\[ = \infty \geq k. \]

For stating the following lemma we use the notion of “constant separable” runtime environment. We say that \( \eta \in \text{RtEnv} \) is constant separable into \( v \in \text{RtEnv} \) iff for all \( k \in \mathbb{R}_{\geq 0} \) and \( t \in T \), \( \eta(k + t) = \eta(k) + \eta(t) \).

Lemma A.13. Let \( \eta \) be a runtime environment constant separable\(^{13}\) into \( v \). Then for all command \( c \),
\[ \text{ert}[c, D](\{k\}) = k + \text{ert}[c](\{k\}). \]

Proof. By induction on the structure of \( c \).

\(^{13}\)See paragraph above.
2. \( f_1(\bot_1) \leq f_2(\uparrow_F(\bot_1)) \) and \( f_2(\bot_2) \leq f_1(\uparrow_F(\bot_2)) \), and

3. \( h_1(f_2(\uparrow_F(\bot_2))) \leq f_2(\uparrow_F(\bot_2)) \) and \( h_2(f_1(\uparrow_F(\bot_2))) \leq f_1(\uparrow_F(\bot_2)) \).

then

\[ f_1(\uparrow_F(\bot_1)) = f_2(\uparrow_F(\bot_2)). \]

**Proof:**

\[ f_1(\uparrow_F(\bot_1)) = f_2(\uparrow_F(\bot_2)) \]

\[ \{\text{"\( \leq \)" is a partial order over \( D \)}\}

\[ f_1(\uparrow_F(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \land f_2(\uparrow_F(\bot_2)) \leq f_1(\uparrow_F(\bot_1)) \]

\[ \{\text{Kleene’s Fixed Point Theorem, \( F_1, F_2 \) continuous}\}

\[ f_1(\sup_n F_1^n(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \]

\[ \land f_2(\sup_n F_2^n(\bot_2)) \leq f_1(\uparrow_F(\bot_1)) \]

\[ \{f_1, f_2 \text{ continuous}\}

\[ \sup_n f_1(\sup_n F_1^n(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \]

\[ \land \sup_n f_2(\sup_n F_2^n(\bot_2)) \leq f_1(\uparrow_F(\bot_1)) \]

\[ \{\forall n, a_s \leq S \implies \sup_n a_n \leq S\}

\[ \forall n, f_1(F_1^n(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \]

\[ \land \forall n, f_2(F_2^n(\bot_2)) \leq f_1(\uparrow_F(\bot_1)) \]

We prove the above pair of inequalities by induction on \( n \). We exhibit the details only for the first one; the second one follows a similar argument. The base case \( f_1(F_1^0(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \) follows from hypothesis 2. For the inductive case \( f_1(F_1^{n+1}(\bot_1)) \leq f_2(\uparrow_F(\bot_2)) \) we reason as follows:

\[ f_1(F_1^{n+1}(\bot_1)) = \text{def. of } F_1^{n+1} \]

\[ f_1(F_1^n(\bot_1)) \]

\[ \leq \text{hyp. 1} \]

\[ h_1(f_1(F_1^n(\bot_1))) \]

\[ \leq \{\text{I.H., monot. of } h_1\}

\[ h_1(f_2(\uparrow_F(\bot_2))) \]

\[ \leq \text{hyp. 3} \]

\[ f_2(\uparrow_F(\bot_2)) \]

Proof of **Theorem 5.1.** The proof of all properties proceeds by induction on the program structure. Except for the case of probabilistic choice and procedure calls, all other programs construct the have already been dealt with in [17, 18]. For probabilistic choice we follow the same reasoning as for conditional branches. We are left to analyze the only case of procedure calls. For each of the properties we reason as follows:

**Continuity.** Let \( F(\eta) = \text{1} \oplus \text{ert}[\Pi(P)]^2_{\eta} \)

\[ \text{ert}[\text{call } P, D](\langle \sup_n t_n \rangle) \]

\[ = \{\text{Lemma A.9}\}

\[ \sup_n F^m(D_{\text{RefEnv}}(\langle \sup_n t_n \rangle)) \]

\[ = \{F^m(D_{\text{RefEnv}}) \text{ continuous; see below}\}

\[ \sup_n \sup_m F^m(D_{\text{RefEnv}})(t_n) \]

\[ = \{\text{Lemma A.2}\}

\[ \sup_n \sup_m F^m(D_{\text{RefEnv}})(t_n) \]

\[ = \{\text{Lemma A.9}\}

We are only left to prove that \( F^m(D_{\text{RefEnv}}) \) is continuous for all \( m \in \mathbb{N} \). We prove this by induction on \( m \). The base case is immediate since \( F^0(D_{\text{RefEnv}}) = D_{\text{RefEnv}} \) and \( D_{\text{RefEnv}} \) is continuous. For the inductive case we have \( F^{m+1}(D_{\text{RefEnv}}) = F(F^m(D_{\text{RefEnv}})) \). The continuity of \( F^{m+1}(D_{\text{RefEnv}}) \) follows from the I.H. and the fact that \( F \) preserves continuity, i.e. \( \eta \) continuous implies \( F(\eta) \) continuous (see **Lemma A.8**).

**Propagation of constants.** By letting \( F(\eta) = \text{1} \oplus \text{ert}[\Pi(P)]^2_{\eta} \), we can recast the property as \( \text{ert}(F(\eta))(k + t) = k + \text{ert}(F(\eta))(t) \), or equivalently, as

\[ \{(\lambda \eta^*, \lambda^*, \eta^*(k + t)) | F(\eta)(F(\eta))(\eta^*(k + t)) \}

To prove this equation, we apply **Lemma A.14** with instantiations

\[ F_1 = F_2 = F \]

\[ f_1 = \lambda \eta^*, \lambda^*, \eta^*(k + t) \]

\[ f_2 = \lambda \eta^*, \lambda^*, \kappa + \eta^*(t^*) \]

\[ h_1 = \lambda \eta^*, \lambda^*, k + 1 + \text{ert}[\Pi(P)]^2_{\lambda \eta^*, \lambda^*(\eta^* - k)}(k + t^*) \]

\[ h_2 = \lambda \eta^*, \lambda^*, k + 1 + \text{ert}[\Pi(P)]^2_{\lambda \eta^*, \lambda^*(\eta^* - k)}(t^*) \]

and \( \omega\text{-cpos } (D_1, \leq_1) = (D_2, \leq_2) = (D, \leq) = \text{RtEnv, } \) and bottom elements \( \bot_1 = \bot_2 = \bot = \bot_{\text{RefEnv}} \). The application of **Lemma A.14** requires the continuity of \( F \) which follows from **Lemma A.7**, the continuity of \( f_1 \) and \( f_2 \), which holds because runtime environments are continuous by definition, and finally the monotonicity of \( h_1 \) and \( h_2 \). This latter fact, together with the fact that \( h_1 \) and \( h_2 \) are effectively well-defined (i.e. have type \( \text{RtEnv} \rightarrow \text{RtEnv} \)) can be proved with an inductive argument (on the structure of \( D(P) \)).

We are left to discharge hypotheses 1–3 of **Lemma A.14**. A simple unfolding of the involved functions yields \( f_1(F(\eta)) \subseteq h_1(f_2(\eta)) \) and \( f_2(F(\eta)) \subseteq h_2(f_2(\eta)) \) for all \( \eta \in \text{RtEnv}; \) this establishes hypothesis 1. As for hypothesis 2, \( f_1(\bot_{\text{RefEnv}}) \subseteq f_2(\uparrow_F(\bot_2)) \) holds because \( f_1(\bot_{\text{RefEnv}}) = \bot_{\text{RefEnv}} \) and \( f_2(\uparrow_F(\bot_2)) \) reduces to \( k \leq \text{ert}(\text{call } P, D)(k + t) \), which holds in view of the monotonicity of transformer \( \text{ert} \) and **Lemma A.12**. Finally, to discharge hypothesis 3 we reason as follows:

\[ h_1(f_2(\uparrow_F(\eta)(\eta))(t) \leq f_2(\uparrow_F(\eta))(t) \]

\[ \{\text{def. of } h_1, f_2, F; \text{let } \eta(t^*) = k + \text{ert}(\text{call } P, D)(t^*-k) \}

\[ 1 + \text{ert}[\Pi(P)]^2_{\eta}(k + t) \leq k + \text{ert}[\text{call } P, D](t) \]

\[ \{\text{def. of } \eta \}

\[ \{\text{def. of } F \}

\[ k + F(\text{ert}(\text{call } P, D))(t) \leq k + \text{ert}[\text{call } P, D](t) \]

\[ \{\text{def. of } \text{ert} \}

\[ k + \text{ert}(\uparrow_F(\eta))(t) \leq k + \text{ert}(\uparrow_F(\eta))(t) \]

\[ \{\text{def. of } \text{ert} \}

\[ k + \text{ert}(\uparrow_F(\eta))(t) \leq k + \text{ert}(\uparrow_F(\eta))(t) \]

\[ \{\"\leq\" \text{ is a partial order}\}

true
This result relies on the notion of
By the monotonicity of
Proof. \(\{\text{def. of } h_2, f_1, F, \text{ let } v(t') = \text{ert}[P, D](t' + k) - k\}\)
\[ k + 1 + \text{ert}[D(P)](t) \leq \text{ert}[P, D](k + t) \]
\[ \{\text{ert}[P, D] \text{ is constant separable into } v; \text{ Lemma A.13}\} \]
\[ F(\text{ert}[P, D])(k + t) \leq \text{ert}[P, D](k + t) \]
\[ \{\text{def. of } F\} \]
\[ F(\text{lfp}(F))(k + t) \leq \text{lfp}(F)(k + t) \]
\[ \{\text{def. of lfp}\} \]
\[ \text{lfp}(F)(k + t) \leq \text{lfp}(F)(k + t) \]
\[ \{\text{"\leq" is a partial order} \}
true

Preservation of infinity. By the monotonicity of \(\text{ert}[c, D]\) and
Lemma A.12, we have
\[ \text{ert}[c, D](\infty) \geq k \quad \forall k \in \mathbb{R}_{\geq 0} , \]
which itself entails \(\text{ert}[c, D](\infty) = \infty\).

A.7 Relation between Transformers ert and wp
To establish Theorem 5.2 we make use of a subsidiary result. This result relies on the notion of separable runtime environment. We say that a runtime environment \(\eta\) is separable into runtimes environments \(\eta_1\) and \(\eta_2\) if we have \(\eta(t_1 + t_2) = \eta_1(t_1) + \eta_2(t_2)\) for every two runtimes \(t_1\) and \(t_2\).

Lemma A.15. For every command \(c\) and runtime environment \(\eta\) separable into \(\eta_1\) and \(\eta_2\),
\[ \text{ert}[c, \eta](t_1 + t_2) = \text{ert}[c, \eta_1](t_1) + \text{wp}[c, \eta_2](t_2) . \]

Proof. For the basic instructions (skip, abort and assignment), the statement follows immediately from the definitions of ert and wp. For the remaining program constructs we reason as follows:

Conditional Branching:
\[ \text{ert}[\text{if } (G) \{c_1\} \text{ else } \{c_2\}]\eta(t_1 + t_2) = \{\text{def. of } \text{ert}[\{c_1\}]\eta\} \]
\[ \{1 + [G] \cdot \text{ert}[c_1]\eta_1(t_1 + t_2) + [-G] \cdot \text{ert}[c_2]\eta_2(t_1 + t_2)\} \]
\[ = \{\text{I.H. on } c_1, c_1\} \]
\[ 1 + [G] \cdot \text{ert}[c_1]\eta_1(t_1 + t_2) + \text{wp}[c_1]\eta_2(t_2) \]
\[ + [-G] \cdot \text{ert}[c_2]\eta_1(t_1 + t_2) + \text{wp}[c_2]\eta_2(t_2) \]
\[ = \{\text{algebra}\} \]
\[ 1 + [G] \cdot \text{ert}[c_1]\eta_1(t_1) + [-G] \cdot \text{ert}[c_2]\eta_1(t_1) \]
\[ + [G] \cdot \text{wp}[c_1]\eta_2(t_2) + [\text{wp}[c_2]\eta_2(t_2) \]
\[ = \{\text{def. of } \text{ert}[\{c_1\}]\eta_1, \text{wp}[\{c_2\}]\eta_2\} \]
\[ \text{ert}[\text{if } (G) \{c_1\} \text{ else } \{c_2\}]\eta_1(t_1) \]
\[ + \text{wp}[\text{if } (G) \{c_1\} \text{ else } \{c_2\}]\eta_2(t_2) \]

Probabilistic Choice: analogous to the conditional branching case.

Sequential Composition:
\[ \text{ert}[c_1; c_2]_\eta(t_1 + t_2) = \{\text{def. of } \text{ert}[\{c_1; c_2\}]\eta\} \]
\[ \text{ert}[c_1]_{\eta_1}(\text{ert}[c_2]_{\eta_2}(t_1 + t_2)) = \{\text{I.H. on } c_2\} \]
\[ \text{ert}[c_1]_{\eta_1}(\text{ert}[c_2]_{\eta_2}(t_1) + \text{wp}[c_2]_{\eta_2}(t_2)) \]
\[ = \{\text{I.H. on } c_1\} \]
\[ \text{ert}[c_1]_{\eta_1}(\text{ert}[c_2]_{\eta_2}(t_1)) + \text{wp}[c_2]_{\eta_2}(\text{wp}[c_2]_{\eta_2}(t_2)) \]
\[ = \{\text{def. of } \text{ert}[\{c_1\}]\eta_1, \text{wp}[\{c_2\}]\eta_2\} \]
\[ \text{ert}[c_1; c_2]_{\eta_1}(t_1) + \text{wp}[c_1; c_2]_{\eta_2}(t_2) \]

Procedure Call:
\[ \text{ert}[\text{call } P]_{\eta}(t_1 + t_2) = \{\text{def. of } \text{ert}[\{\text{call } P\}]\eta\} \]
\[ \eta(t_1 + t_2) = \{\text{I.H. sep. into } \eta_1, \eta_2\} \]
\[ \eta_1(t_1) + \eta_2(t_2) = \{\text{def. of } \text{ert}[\{\text{call } P\}]\eta_1, \text{wp}[\{\text{call } P\}]\eta_2\} \]
\[ \text{ert}[\text{call } P]_{\eta_1}(t_1) + \text{wp}[\text{call } P]_{\eta_2}(t_2) \]

Proof of Theorem 5.2. The proof proceeds by induction on the program structure, but for the inductive reasoning to work we need to consider a stronger statement, namely
\[ \text{ert}[c, D](t_1 + t_2) = \text{ert}[c, D](t_1) + \text{wp}[c, D](t_2) . \]
(3) (We recover the original statement by taking \(t_1 = 0\). For all program constructs \(c\) different from a procedure call, establishing Equation 3 follows exactly the same argument as that used in Lemma A.15 for establishing
\[ \text{ert}[c, \eta](t_1 + t_2) = \text{ert}[c, \eta_1](t_1) + \text{wp}[c, \eta_2](t_2) \]
since \(\text{ert}[\{\}]\eta\) and \(\text{ert}[\{\}]\eta\) obey the same definition rule for such program constructs.
For the case of a procedure call we have to prove that
\[ \text{ert}[\text{call } P, D](t_1 + t_2) = \text{ert}[\text{call } P, D](t_1) + \text{wp}[\text{call } P, D](t_2) . \]
Since
\[ \text{ert}[\text{call } P, D] = \text{lfp}(F) \quad \text{where } F(\eta) = 0 + \text{ert}[D(P)]\eta \]
\[ + \text{wp}[\text{call } P, D] = \text{lfp}(F) \quad \text{where } G(\theta) = \text{wp}[D(P)]\theta \]
and both \(F\) and \(G\) are continuous (see Lemma A.6 and Lemma A.7), by Kleene’s Fixed Point Theorem our statement can be recast as
\[ \sup_n F^n(\bot_{\text{RefEnv}})(t_1 + t_2) = \sup_n F^n(\bot_{\text{RefEnv}})(t_1) + \sup_n G^n(\bot_{\text{RefEnv}})(t_2) , \]
where \(\bot_{\text{RefEnv}} = \lambda \eta : E . 0, \bot_{\text{RefEnv}} = \lambda \lambda : T . 0, F^n(\bot_{\text{RefEnv}}) = F(...(F(\bot_{\text{RefEnv}}))...)\) denotes the repeated application of \(F\) from \(\bot_{\text{RefEnv}} n\) times and likewise for \(G^n(\bot_{\text{RefEnv}})\). Since a standard property of complete partial orders ensures that \(F^n(\bot_{\text{RefEnv}})\) and \(G^n(\bot_{\text{RefEnv}})\) are monotonic w.r.t. \(n\), we can use the Monotone Sequence Theorem (Lemma A.3) to replace \(\sup_n\) with \(\lim_{n \to \infty}\) in the above equation and this way “merge” the two limits in the RHS into a single limit. The above equation is then entailed by formula
\[ \forall n, F^n(\bot_{\text{RefEnv}})(t_1 + t_2) = F^n(\bot_{\text{RefEnv}})(t_1) + G^n(\bot_{\text{RefEnv}})(t_2) , \]
which we prove by induction on \(n\). The base case is immediate since for every runtime \(t, F^n(\bot_{\text{RefEnv}})(t) = G^n(\bot_{\text{RefEnv}})(t) = 0.\)
For the inductive case we reason as follows:
The reasoning for the original—two–side rule—is analogous. The validity of the above rule follows from the following reasoning:

\[ 1 + \sup_{n} l_n \preceq \# \eta \text{ [ call } P, \#(t) \]

implies that for every runtime environment \( \eta \),

\[ \eta(t) \preceq u \implies \# \eta(t) \preceq u_2. \]

The result remain valid if we reverse all inequalities.

Soundness of rule [eet-rec]. Let runtime environment \( \eta^* \) map \( t \) to \( u \) and all other runtimes to \( 0 \) (the constant runtime) \( \infty \). The validity of the rule follows from the following reasoning:

\[ \text{eet} \left[ \text{call } P, \#(t) \right] \preceq 1 + u \]

\[ \{ \text{def. eet (Figure 2)} \}

\[ 1 + \# \eta \text{ [ call } P, \#(t) \] \preceq 1 + u \]

\[ \{ \text{def. } \eta^* \} \]

\[ \# \eta \text{ [ call } P, \#(t) \] \preceq 1 + u \]

\[ \{ \text{Park's Lemma}^{14}, \text{Fact A.2, Lemma A.7} \} \]

\[ \text{eet} \left[ \text{call } P, \#(t) \right] \preceq 1 + u \]

\[ \{ \text{def. } \eta^* \} \]

\[ \text{eet} \left[ \text{call } P, \#(t) \right] \preceq 1 + u \]

\[ \{ \text{rule premise, def } F \} \]

\[ \text{eet} \left[ \text{call } P, \#(t) \right] \preceq 1 + u \]

\[ \{ \text{I.H.} \} \]

\[ \text{true} \]

\[ 1 + l_n \preceq F^{n+1}(\#(t)) \]

\[ \{ \text{def. eet (Figure 2)} \}, \text{F(n)} = 1 \]

\[ 1 + l_n \preceq \# \eta \text{ [ call } P, \#(t) \] \preceq 1 + u \]

\[ \{ \text{rule premise, def } F \} \]

\[ \text{true} \]

\[ 1 + l_n \preceq F^{n+1}(\#(t)) \]

\[ \{ \text{rule premise, def } F \} \]

\[ 1 + l_n \preceq F^{n+1}(\#(t)) \]

\[ \{ \text{I.H.} \} \]

\[ \text{true} \]

\[ 1 + l_n \preceq F^{n+1}(\#(t)) \]

The reasoning for the original—two–side rule—is analogous. The validity of the above rule follows from the following reasoning:

\[ 1 + \sup_{n} l_n \preceq \# \eta \text{ [ call } P, \#(t) \] \preceq 1 + u \]

\[ \{ \text{def. eet (Figure 2)} \}, \text{F(n)} = 1 \]

\[ 1 + \sup_{n} l_n \preceq \# \eta \text{ [ call } P, \#(t) \] \preceq 1 + u \]

\[ \{ \text{Kleene's Fixed Point Thm, Lemma A.7} \} \]

\[ 1 + \sup_{n} l_n \preceq F^{n+1}(\#(t)) \]

Since \( F^n(\#(t)) \) is monotonic w.r.t \( \eta \), \( \sup_{n} F^n(\#(t)) = F^{\infty}(\#(t)) \) and the reasoning continues as follows:

\[ 1 + \sup_{n} l_n \preceq F^{n+1}(\#(t)) \]

\[ \{ \text{I.H.} \} \]

\[ \text{true} \]

\[ 1 + l_n \preceq F^{n+1}(\#(t)) \]

We prove the above statement by induction on \( n \). For the base case we have

\[ 1 + l_0 \preceq F^1(\#(t)) \]

\[ \{ \text{rule premise, def } F \} \]

\[ 1 \leq 1 + \# \eta \left[ \text{call } P, \#(t) \right] \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

For the inductive case we have

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{def. } F^{n+2}(\#(t)) \} \]

\[ 1 + l_{n+1} \leq 1 + \# \eta \left[ \text{call } P, \#(t) \right] \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]

\[ 1 + l_{n+1} \preceq F^{n+2}(\#(t)) \]

\[ \{ \text{rule premise} \} \]

\[ \text{true} \]
Reasoning about Recursive Probabilistic Programs

We assume a given labeling for each program $c \in C$ that specifies the control flow of $c$ as illustrated in Section A.9. Let $\text{Lab}_c$ denote the finite set of labels used in a given program $C$. We assume a special symbol $\downarrow$ to denote successful termination of a program. Furthermore, we make use of the following operations between statements and labels.

- $\text{init} : C \rightarrow \text{Lab}_C$ gives the label corresponding to the beginning of a given program.
- $\text{stmt} : \text{Lab}_C \rightarrow (\mathcal{C} \cup \{\downarrow\})$ gives the statement associated to a label used in a program.
- $\text{succ}_1, \text{succ}_2 : \text{Lab}_C \rightarrow (\text{Lab}_C \cup \{\downarrow\})$ give the first and second successor label of a given program label. In case $\ell \in \text{Lab}_C$ has no such successor, we define $\text{succ}_1(\ell) = \downarrow$ and $\text{succ}_2(\ell) = \downarrow$, respectively.

Definition A.2 (Operational PRMCs). Let $\sigma_0 \in S$ and $f \in \Sigma$. The operational PRMC of program $(c, D)$ starting in initial state $\sigma_0$ with respect to post-expected $f$ is given by $\Psi_0[c, D] = (Q, q_{\text{init}}, \Gamma, \Delta, \tau, \text{rew})$ where

- $Q = \{(\ell, \sigma) | \ell \in \text{Lab}_C \cup \{\downarrow, \text{Term}\}, \sigma \in S\}$,
- $q_{\text{init}} = (\text{init}(c), \sigma_0)$,
- $\Gamma = \text{Lab}_C \cup \{\gamma_0\}$,
- $\Delta$ is given by the least partial function satisfying the rules provided in Figure 3,
- rew$(\langle \text{Term}, \sigma \rangle) = f(\sigma)$ for each $\sigma \in S$ and rew$(q) = 0$ if $q$ is not of the form $(\text{Term}, \sigma)$.

A.10 Soundness of Transformer wp

Proof of Theorem 6.1. For simplicity in the remainder we will assume the program declaration $D$ fixed and therefore, omit it. Consider first an automaton $n(\Psi^f_c[c])$ that behaves exactly the same as $\Psi^f_c[c]$, but counts the number of symbols that currently lie on top of $\gamma_0$ on the stack and which self-loops if that number is exactly $n$ and $\Psi^f_c[c]$ would perform another push onto the stack. It is evident that

$$\begin{align*}
\text{ExpRew}^n(\Psi^f_c[c]) (T) &= \sup_{n \in \mathbb{N}} \text{ExpRew}^n(\Psi^f_c[c]) (T),
\end{align*}$$

since $n(\Psi^f_c[c])$ exhibits a partial behavior of $\Psi^f_c[c]$ in the sense that every path of $n(\Psi^f_c[c])$ that reaches $T$ is (up to renaming) also a path of $\Psi^f_c[c]$. In the other direction, every path $\pi$ of $\Psi^f_c[c]$ that reaches $T$ can be implemented with finite stack size. Therefore, there exists an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ the path $\pi$ is also a path of $n(\Psi^f_c[c])$.

Consider now that by Theorem 3.1 and its proof we can conclude that

$$\begin{align*}
\sup_{n \in \mathbb{N}} \text{wp}^n_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}(f) &= \text{wp}^n_{\Psi^f_c[c]}(f).
\end{align*}$$

It is therefore only left to show that the missing link

$$\lambda, \text{ExpRew}^n(\Psi^f_c[c]) (T) = \text{wp}^n_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}(f)$$

holds for all $n \in \mathbb{N}$. The proof of this equality proceeds by induction on $n$:

The base case $n = 0$: We have to show that

$$\lambda, \text{ExpRew}^0(\Psi^f_c[c]) (T) = \text{wp}^0_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}(f)$$

holds. Whenever the automaton $\Psi^f_c[c]$ would perform the push action associated with a procedure call, the automaton $\Psi^f_c[c]$ immediately self-loops as no push to the stack whatsoever is allowed in this restricted automaton. Therefore, we can syntactically replace every call in $c$ by an abort and still obtain the same behavior for the corresponding restricted automaton. Formally,

$$\text{ExpRew}^0(\Psi^f_c[c]) (T) = \text{ExpRew}^0(\Psi^f_c[c | \text{call P/abort}]) (T).$$

Now, since syntactically call $P$ = abort we have

$$\text{wp}^0_{\Psi^f_c[c | \text{call P/abort}]}(f) = \text{wp}^0_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}(f)$$

and therefore, it is left to show that

$$\lambda, \text{ExpRew}^0(\Psi^f_c[c | \text{call P/abort}]) (T) = \text{wp}^0_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}(f)$$

holds. The proof of this equality proceeds by structural induction on $c$: For the base cases we have:

- **The effectless program skip**: On the denotational side, we have

$$\text{wp}[^0_{\Psi^f_c[c] \Rightarrow \Psi^f_c[c]}](f) = f(\sigma).$$

On the operational side we have $\text{skip}(\text{call P/abort}) = \text{skip}$. Let init(abort) = $\ell$, stmt(abort) = $\downarrow$, and succ(abort) = $\downarrow$. The only path of $\Psi^f_c[c | \text{skip}]$ reaching $T$ is

$$\rho = (\langle \ell, \sigma \rangle, \gamma_0) \leftarrow (\downarrow, \sigma) \leftarrow (\langle \text{Term}, \sigma \rangle, \gamma_0)$$

and its reward is

$$1 \cdot 1 \cdot (0 + 0 + f(\sigma)) = f(\sigma).$$

As $\rho$ is the only path reaching $T$, we have

$$\text{ExpRew}^0(\Psi^f_c[c | \text{skip}]) (T) = f(\sigma) = \text{wp}[\text{skip}]_{\text{abort}}(f) \sigma.$$
The inductive hypothesis on $c_1$ and $c_2$: We now assume that for arbitrary but fixed programs $c_i$, with $i \in \{1, 2\}$, holds
\[
\lambda_\sigma, \text{ExpRew}_{i}^{0}(c_1 [c_2 \triangleright \text{call P/abort}]) (T) = \wp[c_2]_{\text{abort}} (f) .
\]
We can then proceed with the inductive steps:

The sequential composition $c_1 ; c_2$: On the denotational side, we have
\[
\wp[c_1 ; c_2]_{\text{abort}} (f) (\sigma) = \wp[c_1]_{\text{abort}} (\wp[c_2]_{\text{abort}} (f) (\sigma)) .
\]
Operationally, we have
\[
(c_1 ; c_2) [\text{call P/abort}] = c_1 [\text{call P/abort}] ; c_2 [\text{call P/abort}] .
\]
We furthermore observe that any path of the automaton
\[
0 \langle \Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}] ; c_2 [\text{call P/abort}]] \rangle
\]
reaching $T$ is of the form
\[
\rho = (\langle \text{init}[c_1 [\text{call P/abort}]), \sigma \rangle, \gamma_0) \xrightarrow{a_1} \ldots \xrightarrow{a_k} (\langle \text{init}[c_2 [\text{call P/abort}]), \sigma' \rangle, \gamma_0) \xrightarrow{a_{k'}} (\langle \text{Term}, \sigma'' \rangle, \gamma_0)
\]
and any such a path’s reward is given by
\[
\prod_{i=1}^{k} (a_i) \cdot (0 + \ldots + 0 + f(\sigma')) + \prod_{i=k+2}^{k'} (a_i) \cdot (0 + \ldots + f(\sigma'')).
\]
Next, we observe that for any such path $\rho$ a suffix of it, namely
\[
(\langle \text{init}[c_2 [\text{call P/abort}]), \sigma' \rangle, \gamma_0) \xrightarrow{a_{k+2}} \ldots \xrightarrow{a_k} (\langle \text{Term}, \sigma'' \rangle, \gamma_0)
\]
is a path of $0 \langle \Psi^f_{\sigma} \triangleright [c_2 [\text{call P/abort}]] \rangle$ reaching $T$ with reward
\[
\prod_{i=k+2}^{k'} (a_i) \cdot (0 + \ldots + f(\sigma'')) = \prod_{i=k+2}^{k'} (a_i) \cdot f(\sigma'').
\]
Moreover, we can think of the expected reward of
\[
0 \langle \Psi^f_{\sigma} \triangleright [c_2 [\text{call P/abort}]] \rangle
\]
as an expectation
\[
\lambda_\sigma', \text{ExpRew}_{i}^{0}(c_2 [\text{call P/abort}]) (T) ,
\]
which by the inductive hypothesis on $c_2$ is equal to
\[
\wp[c_2]_{\text{abort}} (f) .
\]
Therefore,
\[
0 \langle \Psi^f_{\sigma} \triangleright [c_2 [\text{call P/abort}]] \rangle
\]
and
\[
0 \langle \Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}] ; c_2 [\text{call P/abort}]] \rangle
\]
have the same expected reward, as in the former all paths reaching $T$ have the form
\[
(\langle \text{init}[c_2 [\text{call P/abort}]), \sigma \rangle, \gamma_0) \xrightarrow{a_1} \ldots \xrightarrow{a_k} (\langle \text{Term}, \sigma' \rangle, \gamma_0)
\]
and reward
\[
\prod_{i=1}^{k} (a_i) \cdot (0 + \ldots + 0 + \wp[c_2 [\text{call P/abort}]]_{\text{abort}} (f) (\sigma')) = \wp[c_2 [\text{call P/abort}]]_{\text{abort}} (f) (\sigma').
\]
Keeping that in mind and applying the inductive hypothesis to $c_1$ now yields the desired statement:
\[
\text{ExpRew}_{i}^{0}(c_1 [c_2 [\text{call P/abort}]) (T) = \text{ExpRew}_{i}^{0}(c_1 [\text{call P/abort}]) (T) = \wp[c_1]_{\text{abort}} (\wp[c_2]_{\text{abort}} (f) (\sigma)) \quad \text{(I.H. on } c_1) \]
\[
= \wp[c_1 ; c_2]_{\text{abort}} (f) (\sigma).
\]
The conditional choice if $(G) \{c_1\}$ else $\{c_2\}$: We distinguish two cases:
In Case 1 we have $\sigma = G$. Then on the denotational side, we have
\[
\wp[\text{if } (G) \{c_1\} \text{ else } \{c_2\} ]_{\text{abort}} (f) (\sigma) = ([G] \cdot \wp[c_1]_{\text{abort}} (f) + [-G] \cdot \wp[c_2]_{\text{abort}} (f)) (\sigma)
\]
\[
\wp[c_1]_{\text{abort}} (f) (\sigma) \quad \text{(I.H. on } c_1) \]
\[
\wp[c_1 ; c_2]_{\text{abort}} (f) (\sigma).
\]
Regarding the control flow, let the following hold:
\[
\text{init}[if (G) \{c_1\} \text{ else } \{c_2\}]) = \ell, \quad \text{stmt } (\ell) = \text{if } (G) \{c_1\} \text{ else } \{c_2\}.
\]
\[
\text{succ}_{1} (\ell) = \text{init}[c_1 [\text{call P/abort}]) , \quad \text{and finally } \text{succ}_{2} (\ell) = \text{init}[c_2 [\text{call P/abort}]).
\]
We observe that any path of $0 \langle \Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}]] \rangle$ finally reaching $T$ is of the form
\[
\rho = (\langle \ell, \sigma \rangle, \gamma_0) \xrightarrow{a_1} (\langle \text{init}[c_1 [\text{call P/abort}]), \sigma \rangle, \gamma_0) \xrightarrow{a_2} \ldots \xrightarrow{a_k} (\langle \text{Term}, \sigma' \rangle, \gamma_0)
\]
and it’s reward is given by
\[
1 \cdot \prod_{i=2}^{k} (a_i) \cdot (0 + \ldots + 0 + f(\sigma')) = \prod_{i=2}^{k} (a_i) \cdot f(\sigma').
\]
Next, observe that removing from any such path $\rho$ the initial segment, i.e. removing $\langle \ell, \sigma \rangle$, gives a path of the form
\[
(\langle \text{init}[c_1 [\text{call P/abort}]), \sigma \rangle, \gamma_0) \xrightarrow{a_2} \ldots \xrightarrow{a_k} (\langle \text{Term}, \sigma' \rangle, \gamma_0),
\]
which is a path of $0 \langle \Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}]] \rangle$ reaching $T$ with reward
\[
\prod_{i=2}^{k} (a_i) \cdot (0 + \ldots + f(\sigma')) = \prod_{i=2}^{k} (a_i) \cdot f(\sigma').
\]
Notice that if we remove the initial segments from every path in $\text{Paths}^{0}(\Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}])$, we obtain exactly the set $\text{Paths}^{0}(\Psi^f_{\sigma} \triangleright [c_1 [\text{call P/abort}])$. Thus
\[
0 \langle \Psi^f_{\sigma} \triangleright [\text{if } (G) \{c_1\} \text{ else } \{c_2\} [\text{call P/abort}]] \rangle
\]
as well as $^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\}\rangle$ have the same expected reward. This immediately yields the desired statement:

\[
\text{ExpRew}^0\langle\Psi'_{\ell}\{\langle\ell \; (G) \; (c_1) \; \text{else } (c_2)\} \; \text{call } P/\text{abort}\}\rangle (T) = \text{ExpRew}^0\langle\Psi'_{\ell}\{\langle\ell \; (G) \; (c_1) \; \text{call } P/\text{abort}\} \; \text{else } (c_2)\} \; \text{call } P/\text{abort}\}\rangle (T) = \text{ExpRew}^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\}\rangle (T)
\]

\[
= \text{wp}[c_1]^\ell_{\text{abort}} (f)(\sigma) \quad \text{(I.H. on } c_1) = \text{wp}[\text{if } (G) \{ c_1 \} \text{else } c_2]^\ell_{\text{abort}} (f)(\sigma)
\]

The reasoning for Case 2, i.e. $\sigma \not\in G$, is completely analogous using the inductive hypothesis on $c_2$.

The probabilistic choice $\{c_1\} \cdot \{c_2\}$: On the denotational side, we have

\[
\text{wp}[\{c_1\} \cdot \{c_2\}]^\ell_{\text{abort}} (f)(\sigma) = (p \cdot \text{wp}[c_1]^\ell_{\text{abort}} (f)(\sigma)) + (1-p) \cdot \text{wp}[c_2]^\ell_{\text{abort}} (f)(\sigma)
\]

On the operational side we have

\[
\langle \{c_1\} \cdot \{c_2\} \rangle [\text{call } P/\text{abort}] = \langle c_1 \rangle [\text{call } P/\text{abort}] \langle p \cdot \{c_2\} \rangle [\text{call } P/\text{abort}]
\]

Let $\text{init}(\{c_1\} [\text{call } P/\text{abort}] \langle p \cdot \{c_2\} [\text{call } P/\text{abort}] \rangle) = (\ell, \text{stmt } (\ell) = \{c_1\} [\text{call } P/\text{abort}] \langle p \cdot \{c_2\} [\text{call } P/\text{abort}] \rangle$, let $\text{succ}_1(\ell) = \text{init}(\{c_1\} [\text{call } P/\text{abort}])$, and let $\text{succ}_2(\ell) = \text{init}(\{c_2\} [\text{call } P/\text{abort}])$. We observe that any path of $^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\} \cdot \{c_2[\text{call } P/\text{abort}]\}\rangle$ reaching $T$ is either of the form

\[
\rho_1 = ((\ell, \sigma), \gamma_0) \overset{p}{\rightarrow} \langle \text{init}(c_1[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle \overset{a_k}{\rightarrow} \cdots \overset{a_k}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle
\]

and its reward is given by

\[
p \cdot \left(0 + \sum_{i=2}^{k} (a_i) \cdot (0 + \cdots + 0 + f(\sigma')) \right) = p \cdot \sum_{i=2}^{k} (a_i) \cdot f(\sigma')
\]

or it is of the form

\[
\rho_2 = ((\ell, \sigma), \gamma_0) \overset{1-p}{\rightarrow} \langle \text{init}(c_2[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle \overset{a_k}{\rightarrow} \cdots \overset{a_k}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle
\]

and its reward is given by

\[
(1-p) \cdot \left(0 + \sum_{i=2}^{k'} (a_i) \cdot (0 + \cdots + 0 + f(\sigma'')) \right) = (1-p) \cdot \sum_{i=2}^{k'} (a_i) \cdot f(\sigma'')
\]

Notice that there is a possibility to partition the set

\[
\text{Paths}^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\} \cdot \{c_2[\text{call } P/\text{abort}]\}\rangle
\]

into two sets $P_1$ containing those paths starting with

\[
((\ell, \sigma), \gamma_0) \overset{p}{\rightarrow} \langle \text{init}(c_1[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle, \text{ and a set }
\]

$P_1 \cdot \{\text{call } P/\text{abort}\}$ containing those paths starting with

\[
((\ell, \sigma), \gamma_0) \overset{1-p}{\rightarrow} \langle \text{init}(c_2[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle.
\]

Next, observe that removing from any path in $P_1$ the initial segment, i.e. removing $((\ell, \sigma), \gamma_0) \overset{1}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle$, gives exactly the set $\text{Paths}^0\{c_1[\text{call } P/\text{abort}]\}$. The paths of $^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\}\rangle$ reaching $T$ are of the form

\[
\langle \text{init}(c_2[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle \overset{a_k}{\rightarrow} \cdots \overset{a_k}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle
\]

and have reward

\[
\sum_{i=2}^{k} (a_i) \cdot (0 + \cdots + 0 + f(\sigma')) = \sum_{i=2}^{k'} (a_i) \cdot f(\sigma').
\]

Dually, removing from any path in $P_1 \cdot \{\text{call } P/\text{abort}\}$ the initial segment, i.e. removing $((\ell, \sigma), \gamma_0) \overset{1}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle$, gives exactly the set $\text{Paths}^0\{c_2[\text{call } P/\text{abort}]\}$. The paths of $^0\langle\Psi'_{\ell}\{c_2[\text{call } P/\text{abort}]\}\rangle$ reaching $T$ are of the form

\[
\langle \text{init}(c_2[\text{call } P/\text{abort}]), \sigma), \gamma_0 \rangle \overset{a_k}{\rightarrow} \cdots \overset{a_k}{\rightarrow} \langle \text{Term}, \sigma''\rangle, \gamma_0 \rangle
\]

and have reward

\[
\sum_{i=2}^{k'} (a_i) \cdot (0 + \cdots + 0 + f(\sigma'')) = \sum_{i=2}^{k} (a_i) \cdot f(\sigma'').
\]

Since $P_1$ and $P_1 \cdot \{\text{call } P/\text{abort}\}$ was a partition of the path set $\text{Paths}^0\{\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\} \cdot \{c_2[\text{call } P/\text{abort}]\}\}$, we can conclude:

\[
\text{ExpRew}^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\} \cdot \{c_2[\text{call } P/\text{abort}]\}\rangle (T) = p \cdot \text{ExpRew}^0\langle\Psi'_{\ell}\{c_2[\text{call } P/\text{abort}]\}\rangle (T)
\]

\[
+ (1-p) \cdot \text{ExpRew}^0\langle\Psi'_{\ell}\{c_1[\text{call } P/\text{abort}]\}\rangle (T)
\]

\[
= p \cdot \text{wp}[c_1]^\ell_{\text{abort}} (f)(\sigma) + (1-p) \cdot \text{wp}[c_2]^\ell_{\text{abort}} (f)(\sigma)
\]

\[
\quad \text{(I.H. on } c_1 \text{ and } c_2).
\]

This ends the proof for the base case of the induction on $n$ and we can now state the inductive hypothesis:

**Inductive hypothesis on $n$:** We assume that for an arbitrary but fixed $n \in \mathbb{N}$ holds

\[
\lambda \sigma. \text{ExpRew}^n\langle\Psi'_{\ell}\{c\}\rangle (T) = \text{wp}[c]^n_{\text{wp}} (f)
\]

for all programs $c$. We can then proceed with the inductive step:

**Inductive step $n \rightarrow n+1$:** We now have to show that

\[
\lambda \sigma. \text{ExpRew}^{n+1}\langle\Psi'_{\ell}\{c\}\rangle (T) = \text{wp}[c]^{n+1}_{\text{wp}} (f)
\]

holds assuming the inductive hypothesis on $n$. The proof of this equality proceeds quite analogously, again by structural induction on $c$:

The base cases skip, abort, $x \equiv E$: The proofs for these base cases are completely analogous to the proofs conducted in the base case $n = 0$.

**The procedure call call P:** The procedure call is technically a base case in the structural induction on $c$ as it is an atomic statement. It does, however, require using the inductive hypothesis on $n$. The proof goes as follows: By an argument on the transition relation $\Delta^{n+1}\langle\Psi'_{\ell}\{c\}\rangle$ we see that

\[
\text{ExpRew}^{n+1}\langle\Psi'_{\ell}\{c\}\rangle (T) = \text{ExpRew}^n\langle\Psi'_{\ell}\{c\}\rangle (T).
\]
To the right hand side, we can apply the inductive hypothesis on $n$ and then obtain the desired result:
\[
\lambda \sigma. \text{ExpRew}^{n+1} (\psi[^1_c]_{c_1: c_2 = 1}) (T)
\]
\[
= \lambda \sigma. \text{ExpRew}^n (\psi[^1_c]_{\text{call} P}) (T)
\]
\[
= \text{wp}[\pi^P]_{\text{call} P}^\rho (f) \quad \text{(I.H. on } n)\]
\[
= \text{wp}[\text{call} P]_{\text{call} P}^\rho (f)
\]

**Inductive hypothesis and all inductive steps:** The inductive hypothesis and the proofs for the inductive steps are completely analogous to the inductive hypothesis and the proofs conducted in the base case $n = 0$. Exemplarily, we shall sketch the proof for the sequential composition: By a lengthy argument and application of the inductive hypothesis on $c_2$ (completely analog to the base case for $n = 0$) one arrives at
\[
\text{ExpRew}^{n+1} (\psi[^1_c]_{c_1: c_2 = 1}) (T)
\]
\[
= \text{ExpRew}^n (\psi[^1_c]_{\text{call} P}^\rho_{c_2 = 1}) (T)
\]

Applying the inductive hypothesis on $c_1$ then yields the desired result:
\[
\lambda \sigma. \text{ExpRew}^{n+1} (\psi[^1_c]_{c_1: c_2 = 1}) (T)
\]
\[
= \text{ExpRew}^n (\psi[^1_c]_{\text{call} P}^\rho_{c_2 = 1}) (T)
\]
\[
= \text{wp}[c_1]_{\text{wp}[\text{call} P]_{c_2 = 1}}^\rho (f)
\]
\[
= \text{wp}[c_1; c_2]_{\text{wp}[\text{call} P]_{c_2 = 1}}^\rho (f)
\]

A.11 Case Study

The omitted details for proving the second partial correctness property are provided in Figure 9.

![Figure 9](image-url)