Hopping Proofs of Expectation-Based Properties: Applications to Skiplists and Security Proofs

MARTIN AVANZINI, Centre Inria d'Université Côte d'Azur, France
GILLES BARTHE, MPI-SP, Germany and IMDEA Software Institute, Spain
BENJAMIN GRÉGOIRE, Centre Inria d'Université Côte d'Azur, France
GEORG MOSER, University of Innsbruck, Austria
GABRIELE VANONI, IRIF, CNRS, Université Paris Cité, France

We propose, implement, and evaluate a hopping proof approach for proving expectation-based properties of probabilistic programs. Our approach combines eHL, a syntax-directed proof system for reducing proof goals of a program to proof goals of simpler programs, with a "hopping" proof rule for reducing proof goals of an original program to proof goal of a different program which is suitably related (by means of pRHL, a relational program logic for probabilistic program) to the original program. We prove that eHL is sound for a core language with procedure calls and adversarial computations, and complete for the adversary-free fragment of the language. We also provide an implementation of eHL into EasyCrypt, a proof assistant tailored for reasoning about relational properties of probabilistic programs. We provide a tight integration of eHL with other program logics supported by EasyCrypt, and in particular probabilistic Relational Hoare Logic (pRHL). Using this tight integration, we give mechanized proofs of expected complexity of in-place implementations of randomized quickselect and skip lists. We also sketch applications of our approach to cryptographic proofs and discuss the broader impact of eHL in the EasyCrypt proof assistant.

Additional Key Words and Phrases: probabilistic programs, Hoare logic, formal verification

1 INTRODUCTION

There is a long line of work that develops rigorous approaches for proving properties of probabilistic programs. These approaches generalize to the probabilistic setting the classic notions of pre- and post-conditions and of invariants. A fundamental difference is that in the probabilistic setting these notions are quantitative. Assertions are expectations, i.e. functions that map states to extended positive reals. The use of expectations was pioneered by Kozen [Kozen 1985], systematized by Morgan, McIver and Seidel [Morgan et al. 1996], and still prevails to date.

Unfortunately, these approaches are often difficult to use. One main reason is that proofs of probabilistic programs do not always follow their control flow. Another reason is that once the target program property is fixed, it is often very convenient to reason about more abstract or refactored programs. From the theoretical perspective, none of these concerns is an issue, since in general these approaches are complete. However, more flexible approaches are desirable when verifying concrete examples, in particular when building mechanized proofs.

Problem statement and contributions. The main goal of this paper is to support flexible computer-aided verification of probabilistic programs, and in particular to develop an approach that allows breaking away from the control flow of programs, and change program representation during verification. Our target is to use our approach on relatively small but challenging probabilistic

Authors' addresses: Martin Avanzini, Centre Inria d'Université Côte d'Azur, Route des Lucioles - BP 93, Sophia Antipolis, 06902, France, martin.avanzini@inria.fr; Gilles Barthe, MPI-SP, Bochum, 44799, Germany and IMDEA Software Institute, Pozuelo de Alarcon, Madrid, 28223, Spain, gilles.barthe@mpi-sp.org; Benjamin Grégoire, Centre Inria d'Université Côte d'Azur, Route des Lucioles - BP 93, Sophia Antipolis, 06902, France, benjamin.gregoire@inria.fr; Georg Moser, Department of Computer Science, University of Innsbruck, Technikerstraße 21a, Innsbruck, 6020, Austria, georg.moser@uibk.ac.at; Gabriele Vanoni, IRIF, CNRS, Université Paris Cité, Paris, France, gabriele.vanoni@irif.fr.

2018. ACM 2475-1421/2018/1-ART1
https://doi.org/

programs drawn from the theory of randomized algorithms and from cryptography. The choice of application domains naturally delineates the choice of the pWhile language, a core probabilistic language with sampling from discrete distributions, (non-recursive) procedures and adversaries. Informally, an adversary is an unspecified quantified procedure with constraints on the variables it can read and write, and on the procedures it can call. Thus the main challenge with adversaries is to devise proof principles that are sound w.r.t. all possible instantiations of the adversary. We note that in contrast with many other works in this realm, pWhile explicitly (and purposedly) does not support conditioning, concurrency and non-determinism, which do not have a central role in our applications.

We achieve our goals in three steps. First, we define a program logic, called eHL, to reason about expectation-based properties of pWhile programs. Judgments of eHL are of the form $\{ f \}\ \mathtt{C}\ \{ g \}$ where C is a statement and f and g are maps from program states to extended positive reals. Informally, a judgment is valid if the expected value of g on the output memory is upper bounded by the value of f on the initial memory. The proof system for eHL closely matches the pGCL pre-expectation calculus [Morgan et al. 1996], except for loops, procedure and adversary calls:

- our rule for loops uses the notion of upper invariant from the literature;
- our rule for procedures uses auxiliary variables. It is folklore that complete proof rules for procedures—even in the deterministic setting—require the use of auxiliary variables, cf. [Kleymann 1998, 1999; Nipkow 2002a,b]. We show that auxiliary variables also allow to recover completeness in the probabilistic setting.
- our proof rule for adversaries is new. The main challenge is to devise useful and sound proof rules based exclusively on the aforementioned adversary constraints.

In addition, our program logic features a "hopping"[1] proof rule to reduce the proof of a probabilistic program $\mathtt{C}'$ to a proof of a probabilistic program C. Hopping proofs subsume the "abstract and verify" or "refactor and verify" paradigms that are commonly used in verification by allowing the possibility to perform arbitrary long interleavings of verification steps with abstraction/refactoring steps. They have been previously used in interactive and automated program verification, including [Lammich and Tuerk 2012; Magill et al. 2010; Nipkow et al. 2020; Tassarotti and Harper 2019]. In our case, programs are probabilistic, so we use the relational program logic pRHL [Barthe et al. 2009] (we defer to subsequent sections for the definition of the pRHL judgment $\vdash \{ P \}\ \mathtt{C}' \sim \mathtt{C}\ \{ Q \}$) in the following way:

$$\frac{\vdash \{ f' \}\ \mathtt{C}'\ \{ g' \} \quad \vdash \{ P \}\ \mathtt{C}' \sim \mathtt{C}\ \{ Q \} \quad \text{(condition ommitted)}}{\vdash \{ f \}\ \mathtt{C}\ \{ g \}}\ [\textsc{prhl}]$$

This rule brings hopping proofs to the realm of expectation-based properties. Our case studies use the rule to switch to a more abstract representation of probabilistic programs, and to switch to a different probabilistic program, e.g. one whose control flow follows the reasoning.

Our logic also features a proof rule inspired by the *frame* rule, also known as rule of *constancy*, from classical Hoare Logic. Indispensable in practice, the rule improves upon modularity and compositionality of the calculus, by allowing one to focus only on those parts of assertions that are potentially affected during evaluation. Leveraging the reverse Jensen's inequality, the rule takes the form

$$\frac{\vdash_Z \{ f \}\ \mathtt{C}\ \{ g \} \quad F \perp \mathsf{Mod}_{\mathtt{C}} \quad F \text{ concave and monotone}}{\vdash_Z \{ F[f] \}\ \mathtt{C}\ \{ F[g] \}}\ [\textsc{frame}]$$

In short, it permits the extension of judgments to arbitrary contexts F (i) depending only on the memory not modified by the statement C that is (ii) concave (e.g. linear or sublinear) and monotone,

[1] This style of proof is commonly used in cryptographic proofs under the name game-hopping (probabilistic programs with adversary calls are known as games in this realm). We simply use the name "hopping" here.

when seen as function. For instance, the rule allows to deduce $\vdash_Z \{ \log(2x) + y \}\ \mathtt{C}\ \{ \log(x) + y \}$ from $\vdash_Z \{ 2x \}\ \mathtt{C}\ \{ x \}$ by taking $F[\square] = \log \square + y$, whenever C leaves y unchanged.

Second, we implement our program logic in the EasyCrypt proof assistant [Barthe et al. 2013], an existing tool for the verification of probabilistic programs and cryptographic proofs. Our implementation is carefully crafted to leverage some key features of EasyCrypt, including some weak forms of weakest precondition and SMT-based support. Concretely, we define and implement another set of proof rules that make deductive verification more practical. This set of proof rules is obtained by adapting classic approaches to turn Hoare logics into deductive verification tools, e.g. chaining applications of construct-specific rules with applications of sequential composition and non-structural rules. In order to reason effectively about pre-expectations in the ambient logic of EasyCrypt, we have also developed a library of mathematical definitions and facts about extended positive reals. This library is used critically in our case studies.

Finally, we use our framework to mechanize proofs of several examples. Our main examples are proofs of expected cost for in-place implementations of randomized quickselect and skip lists. Both examples leverage the full power of the framework and go beyond the reach of previous approaches. In particular, our proof is the first to establish a logarithmic bound for skip lists implementations—prior works either establish a logarithmic bound for an abstract description of skip lists [Haslbeck and Eberl 2020] or a linear bound for a (concurrent) implementation of 2-level skiplists [Tassarotti and Harper 2019]. In addition, we illustrate how our framework can be used beneficially in the context of cryptographic proofs. In contrast to the expected cost examples, which target real examples, we consider a synthetic example of cryptographic proofs, inspired from concurrent work [Barbosa et al. 2023] that uses our implementation of eHL to prove security of Dilithium, a post-quantum signature scheme recently standardized by the NIST (National Institute of Standards and Technology). The goal of our example is to illustrate how eHL can be used to obtain simpler proofs with tighter security bounds. However, potential uses of eHL are not limited to such use cases. We also discuss informally how eHL can be used to verify previously axiomatized techniques for reasoning about failure events, and to prove probability bounds in place of the existing logic implemented in EasyCrypt.

In summary, our main contributions are:

- the design, theoretical study and implementation of eHL;
- the application of eHL to expected cost analysis of randomized quickselect and skip lists;
- an illustration of the benefits of eHL in cryptographic proofs.

Artifact. The implementation of eHL, the library of expectations and the formally verified case studies will be submitted as an artifact. The case studies themselves are also available in source form as supplementary material. As an indication, the implementation of eHL proof system and associated libraries represents about 3,000 lines of OCaml code and 1,000 lines of EasyCrypt code. The proof of the quickselect example represent 70 lines for the programs, 300 lines for a library on partition, 110 lines for the equivalence proof in pRHL (concrete version versus the abstract one) and 70 lines for the proof bounding the expected cost in eHL. For skip lists, the proof consists of 2600 lines of EasyCrypt code, about 500 lines for bounding the expectation, the remaining part is mostly concerned with the equivalence proofs and functional correctness. The implementation and case studies will also be made publicly available in GitHub.

Outline. This paper is structured as follows. In Section 3, we provide a bird's eye view on the contributions of this work. Sections 4 and 5 formally establish the expectation logic eHL and its integration with pRHL, while in Section 6 we employ our framework to obtain a fully formalized average case complexity analysis of (a natural and realistic implementation of) *skip lists*, a randomized data structure of interest for practitioners. In Section 7 we extend eHL to a

setting permitting adversarial code and demonstrates the usefulness of the logic for carrying out cryptographic proofs. Section 8 we describe the integration of eHL into EasyCrypt and provide further details on the formal verification of the case studies. In Section 2 we consider related work, and we finally conclude in Section 9.

2 RELATED WORK

There is a large body of work on formal verification of probabilistic programs and resource analysis. For space reasons, we mention only closely related work.

Verification of probabilistic programs. Expectation-based reasoning can be traced back to the seminal work of Kozen [Kozen 1985], who developed a sound and complete propositional dynamic logic for a core probabilistic programming language. It was further developed by Morgan, McIver and Seidel [Morgan et al. 1996], who introduced and studied extensively probabilistic predicate transformers for a core probabilistic language with non-deterrinism. These approaches were recently extended to recursive procedures [Olmedo et al. 2016] and conditioning [Olmedo et al. 2018]. eHL inherits many technical tools from this line of work, in particular the use of upper invariants. However, eHL makes several (minor but practically important) technical contributions: it embeds pRHL into expectation-based reasoning; it supports adversary calls; it features a non-structural rule to simplify expectations (to our best knowledge, no such rule has been considered before); it recasts in the setting of probabilistic programs existing approaches to achieve completeness of Hoare logic in presence of procedures. For the latter, we follow the approach of [Kleymann 1998, 1999; Nipkow 2002a,b].

Complexity analysis of probabilistic of programs. There is also a huge body of work related to complexity analysis of probabilistic programs. Related to probabilistic predicate transformers, Kaminski et al. [2018] define an expected runtime transformer ert for a core probabilistic programming language with non-determinism. Subsequent works extend the expected runtime transformer with recursive procedures [Olmedo et al. 2016], amortized reasoning Batz et al. [2023], or to higher-order functions [Avanzini et al. 2021]. Related to this line of work, several automated tools have emerged Avanzini et al. [2020b, 2023]; Ngo et al. [2018]. Also martingale theory has been successfully tailored towards the analysis of complexity related properties of imperative programs [Agrawal et al. 2018; Barthe et al. 2016; Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2017; Takisaka et al. 2018; Wang et al. 2019]. These notions correspond close to that of Lyapunov ranking functions for proving (positive almost-sure) termination, and for deriving bounds on the runtime [Avanzini et al. 2020a; Bournez and Garnier 2005]. For functional languages, type-based approaches to complexity analysis turned out useful [Avanzini et al. 2019; Leutgeb et al. 2022; Wang et al. 2020].

Mechanized analyses of probabilistic programs. Haslbeck [2021] implements a Hoare style calculus related to the ert calculus of Kaminski et al. within the Isabelle/HOL proof assistant. Interestingly, his work contains a frame rule which can be interpreted as a special case of the one we give. Related, Hurd et al. [2004] formalized a weakest pre-condition calculus for probabilistic programs within HOL, and proof several interesting meta-theoretical properties of the calculus. Program verification is aided through the extraction of recurrence relations to Prolog. Both works include proofs of soundness and completeness of the transformer. In contrast, our core logical rules are part of the trusted computing base. This is in line with the approach inEasyCrypt, where proof rules for program verification are not verified—in other words, EasyCrypt does not use a shallow nor a deep embedding of programs, but rather a hardwired embedding.

Van der Weegen and McKinna [2008] was probably the first to formalize quicksort in a proof assistant, more precisely in Coq. They used a shallow embedding and analyzed the average case

complexity of more high-level, functional version of quicksort. In a similar spirit, Eberl et al. [2020] use the Isabelle/HOL proof assistant to reason via a shallow embedding about the average case complexity of algorithms on binary tree structures. Notably, their analysis covers (the functional variant of) quicksort. Tassarotti and Harper [2018] study quantitative properties of concrete randomized algorithms, focusing on the formal verification of tail bounds. For example they handle (a functional version of) quicksort, again using a monadic embedding. Their analysis is formalized in Coq.

The average complexity analysis of skiplists is rather intricate, rendering skip lists a prime example to evaluate the expresivity and usability of proof assistants. Haslbeck and Eberl [2020] formalise the relationship between the expected height and expected length of search paths within the proof assistant Isabelle/HOL, leading also to the formalisation of a considerable amount of results of probability theory. Wheareas the starting point of Haslbeck and Eberl is a formal but abstract specification, here, we study a concrete algorithm resembling the reference implementation given by Pugh [1990b]. This explains our focus on program logics, rather than the formalisation of mathematical results. Relying on the extensive library underlying EasyCrypt, our formalisation effort is mostly concerned with laws on expectations (such as, linearity or Jensen's inequality). Strongly related to our formal complexity analysis of skip lists is the work by Tassarotti and Harper [2019] on concurrent skip lists. Their Coq formalization extends Iris [Jung et al. 2015] with probabilistic coupling, conceptually in line with our use of eHL in conjunction with pRHL. Their very impressive formalization is orthogonal to our results. On the one hand, the focus is on the verification of quantitative program behaviour in the context of concurrency, while our analysis only concerns sequential evaluation. On the other hand, the notion of skip lists is restricted to two levels and the obtained upper bound on the expected search length is linear, while we consider skip lists in their original definition and re-obtain the original logarithmic bound, in expectation. This latter aspect requires a more involved encoding of our non-concurrent version and conclusively a more sophisticated verification.

Comparison with EasyCrypt. EasyCrypt is an interactive proof assistant targetted to formal verification of cryptographic proofs. Its main component pRHL is used to support game-hopping proofs. In addition, EasyCrypt features a program logic called phoare for reasoning about the probability of events. In contrast to eHL, phoare judgments are of the form $\vdash_{\diamond p} \{\,\phi\,\}\ c\ \{\,\psi\,\}$, where ϕ, ψ are boolean-valued assertions and $\diamond$ is either $\leq$, $\geq$, or $=$; unfortunately, it is difficult to build sound, complete, and practical proof systems for such judgments. Moreover, the proof rules of phoare, and in particular the rule for loops, require programs to be certainly terminating. In general, it would seem beneficial to deprecate phoare and use eHL instead.

EasyCrypt also provides a cost logic for adversarial programs [Barbosa et al. 2021]. The purpose of the cost logic is to upper bound the the complexity of constructed adversaries, i.e. programs with adversary calls that formalize the security reduction from security of a cryptographic scheme to hardness assumptions or assumptions about primitives. One main rule of the logic is an instatiation rule, which allows to reason about the cost of a program where the adversary is instantiated by another program—it can either be a concrete program but also a so-called constructed adversary,. The instantiation rule is required to upper bound constructed adversaries for complex cryptographic systems that are built from several components. The logic is focused on worst-case cost. An interesting direction for future work is to adapt this logic to expected cost.

3 A BIRD'S-EYE VIEW ON OUR METHODOLOGY

In what follows, we introduce our methodology on Tony Hoare's quickselect [Hoare 1961]: a non-trivial, (possibly) non-recursive, randomized algorithm.

```
var ct;

proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then
      i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

proc qselect(a, k)
  ct ← 0;
  (l, h) ← (0, size(a) − 1);
  while l < h do
    i ← rpartition(a, l, h);
    if i = k then
      l ← i; h ← i // exit loop
    elseif i < k then
      l ← i + 1 // descent right
    else
      h ← i − 1 // descent left
  return a[k]
```

(a) Quickselect.

```
var ct;
// l ≤ h | ct + (h − l) + 1/(h−l+1) Σ_{i=l}^{h} f(i)
proc rpartition_abs(l, h)
  // l ≤ h | ct + (h − l) + 1/(h−l+1) Σ_{i=l}^{h} f(i)
  // E_{unif(l,h)} [λi. ct + (h − l) + f(i)]
  ct ← ct + (h − l);
  // E_{unif(l,h)} [λi. ct + f(i)]
  i ←$ unif(l, h);
  // ct + f(i)
  return i
// ct + f(res)

// 0 ≤ k < n | 4(n − 1)
proc qselect_abs(n, k)
  // 0 ≤ k < n | 4(n − 1)
  ct ← 0;
  // 0 ≤ k < n | ct + 4(n − 1)
  (l, h) ← (0, n − 1);
  // 0 ≤ l ≤ k ≤ h | ct + 4(h − l)
  while l < h do
    // l < h ∧ 0 ≤ l ≤ k ≤ h | ct + 4(h − l)        (⋆)
    // 0 ≤ l ≤ k ≤ h | ct + (h − l) + 1/(h−l+1) Σ_{i=l}^{h} g(i, k, l, h)
    i ← rpartition_abs(l, h);
    // 0 ≤ l ≤ k ≤ h | ct + g(i, k, l, h)
    if i = k then l ← i; h ← i
    elseif i < k then l ← i + 1
    else h ← i − 1
    // 0 ≤ l ≤ k ≤ h | ct + 4(h − l)
  // h ≤ l ∧ 0 ≤ l ≤ k ≤ h | ct + 4(h − l)
  // ct
  return ()
// ct
```

(b) Size abstraction.

Fig. 1. Implementation of `qselect` (left) of quickselect and its eHL annotated abstraction `qselect_abs` (right). The term `g(i, k, l, h)` abbreviates `if i = k then 0 elseif i < k then 4(h − (l + 1)) else 4(h − 1 − l)`.

Quickselect. *Sorting* and *searching* are arguably the most studied algorithmic problems in computer science.[2] Quickselect is a selection algorithm to find the kth smallest element in a given (unordered) array. Quickselect operates similar to quicksort, by partitioning the array around a chosen pivot. However, the recursive call is performed just on the partition actually containing the element one is looking for. This observation allows one to perform a tail-call optimization of recursive quickselect, which produces an iterative algorithm. As for quicksort, performance degrades if bad pivots are consistently chosen. By choosing a pivot uniformly at random at each stage, it can be shown that quickselect expected runtime, often more interesting than worst-case complexity when randomness plays a role, is in $O(n)$.[3] The code of quickselect with random pivot selection is given in Figure 1a. Arrays are indexed from 0, for instance, `qselect([4, 6, 2, 8], 1) = 4`

[2]This is for example witnessed by the fact that Donald Knuth dedicated an entire volume of his celebrated series *The Art of Computer Programming* [Knuth 1973] just to these two problems.

[3]Actually, choosing the pivot uniformly at random turns out to allow the same average-case complexity analysis of deterministic quickselect with uniformly distributed inputs.

since 4 occurs at index 1 in the sorted input array [2, 4, 6, 8]. Randomized partitioning of an array a (within indices l and h) is implemented with rpartition(a, l, h). The instruction unif(l, h), used to choose the pivot index p, samples at random an integer between l and h. It is the only point of the code where randomness actually plays a role. Partitioning is then carried out via partition following the Lomuto partition scheme, expecting the pivot at the final index.

Informal Complexity Analysis. The classic textbook proof on the average case complexity of quickselect can be found in [Cormen et al. 2009]. It is based on a sequence of lemmas that are proved looking at the source code in a quite abstract way, through some high-level reasoning.

An important observation is that for each input, if the pivot is chosen uniformly at random from the interval [l, h], then so is its rank (the position of an element in the sorted array) i. Thus, partitioning with pivot of rank i has probability $1/h-l+1$ and, depending on i, the resulting parts of the partition have sizes i − l and h − i, respectively. The procedure qselect loops over just one of the parts, the one actually containing the element we are looking for. In particular, if i < k, the right partition of size h − i is explored, likewise, if i > k, then the left partition of size i − l is explored. In the remaining case i = k the k-th element has been found. Averaging over all the h − l + 1 possible partitions and noting that the number of comparisons performed inside partition is h − l, the average number of comparisons can be estimated accurately by solving the following recurrence relation:

$$C(l,h) = (h-l) + \frac{1}{h-l+1}\left(\sum_{i=l}^{k-1} C(i+1,h) + \sum_{i=k+1}^{h} C(l,i-1)\right) \qquad (\dagger)$$

Then, it is not difficult to prove that $C(l,h) \leq 4(h-l)$. Since l is initialized to 0 and h to size(a) − 1, we obtain the well-known linear bound of O(size(a)), in expectation.

Towards a Formal Analysis. The complexity analysis sketched above is still informal. In particular, the recurrence relation is obtained by a high-level analysis of the code, and through informal reasoning involving probabilities, sizes of partitions, etc. How can we be sure that all of this is correct?

In this paper, we propose a formal *end-to-end methodology* that is able to provide upper-bounds on the complexity of randomized programs, based on the general methodology of Hoare logic [Hoare 1969]. Towards this formalization, we first have to endow a cost model, i.e., be precise in exactly what to measure. A generic way to do so is to simply instrument the program with a cost counter, as we have already done in Figure 1a. Notice how the global variable ct takes account of the total number of comparisons—the usual cost metric for sorting and selection algorithms—performed by qselect. Our objective now turns into bounding the value that ct takes on average after execution, in terms of the size of the input.[4]

From here, a fully formalized complexity analysis of qselect is certainly possible, however, unnecessarily complicated. As we have already seen in the informal proof, a priori we do not really have to reason about the full program. Some parts of it can be abstracted, so that the complexity analysis becomes easier. This is exactly what we have done when we have claimed that partition does h − l comparisons. Indeed, program abstraction is a useful tool in program analysis (see e.g. [Magill et al. 2010]). Consider the procedure qselect_abs, depicted in Figure 1b, giving a complexity preserving skeleton of qselect. Ignoring gray annotations for now, in essence arrays a are abstracted by their size n. While the skeleton of quickselect remains identical, partitioning

[4] The attentive reader may have noticed that since we are about to measure a counter after execution, the correspondence hinges on (almost-sure) termination. If one is interested in the analysis of non-terminating programs and the cost incurred by infinite executions, one way to overcome the discrepancy is to externalize the cost counter in the program semantics (see e.g. [Kaminski et al. 2018]).

of the array becomes superfluous. In rpartition_abs a cost of $\mathtt{h} - \mathtt{l}$ is incurred directly and the rank i, rather than the pivot p, is sampled.

Naturally, the claim about the complexity equivalence of the two programs has to be made formal. To this end, *relational program logics* such as *probabilistic relational Hoare logic* (pRHL) provide a suitable solution [Barthe et al. 2015, 2012, 2017]. Moreover, support for pRHL is readily available in the proof assistant EasyCrypt. In pRHL, judgments take the form of (relational) Hoare triples

$$\{\, P \,\}\ \mathtt{C} \sim \mathtt{D}\ \{\, Q \,\}$$

where P and Q are predicates over the joint program states of C and D, with the informal meaning that on inputs related by P, the programs C and D produce an output (distribution) related by Q.[5] Referring with $(\cdot)^{\langle 1\rangle}$ and $(\cdot)^{\langle 2\rangle}$ to the state of the left- and right program the triple

$$\vdash \{\, \mathtt{unique}(\mathtt{a}^{\langle 1\rangle}) \land \mathtt{size}(\mathtt{a}^{\langle 1\rangle}) = \mathtt{n}^{\langle 2\rangle} \land \mathtt{k}^{\langle 1\rangle} = \mathtt{k}^{\langle 2\rangle} \,\}\ \mathtt{qselect}(\mathtt{a},\mathtt{k}) \sim \mathtt{qselect\_abs}(\mathtt{n},\mathtt{k})\ \{\, \mathtt{ct}^{\langle 1\rangle} = \mathtt{ct}^{\langle 2\rangle} \,\} \qquad \text{(equiv\_qselect)}$$

asserts that if the inputs are related in the obvious way, then the (distributions of) cost counters ct are identical after execution.[6] The main crux of the proof lies in proving a related statement on partitioning:

$$\begin{aligned} &\vdash \{\, \mathtt{unique}(\mathtt{a}^{\langle 1\rangle}) \land (\mathtt{l},\mathtt{h})^{\langle 1\rangle} = (\mathtt{l},\mathtt{h})^{\langle 2\rangle} \,\} \qquad \text{(equiv\_rpartition)}\\ &\quad \mathtt{rpartition}(\mathtt{a},\mathtt{l},\mathtt{h}) \sim \mathtt{rpartition\_abs}(\mathtt{a},\mathtt{l},\mathtt{h})\\ &\quad \{\, \mathtt{ct}^{\langle 1\rangle} = \mathtt{ct}^{\langle 2\rangle} \land \mathtt{res}^{\langle 1\rangle} = \mathtt{res}^{\langle 2\rangle} \,\} \end{aligned}$$

where res refers to the return value of the procedure. Comparing the two procedures, in effect this statement formalizes that (i) partitioning itself performs $\mathtt{h} - \mathtt{l}$ comparisons ($\mathtt{ct}^{\langle 1\rangle} = \mathtt{ct}^{\langle 2\rangle}$) and that (ii) the rank of the pivot lies uniformly in the interval $[\mathtt{l},\mathtt{h}]$ ($\mathtt{res}^{\langle 1\rangle} = \mathtt{res}^{\langle 2\rangle}$). While the former point is quite trivial to prove, the latter property essentially states that pivot positions and ranks are in a bijective relationship, a property that rests on functional correctness of partition and uniqueness.

Formal Reasoning about Expectations. Through the correspondence (equiv_qselect) we have achieved a separation of concerns, as functional correctness properties relevant to the complexity analysis have been dealt with. Knowing that qselect_abs is a cost-preserving abstraction of qselect, we can thus focus on the core of the complexity analysis, as carried out in the informal analysis above.

For this, we use a Hoare logic for reasoning about expectations. This logic, dubbed *Expectation Hoare Logic* (eHL), constitutes a *sound and complete* logic for reasoning about judgments of the form

$$\vdash \{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$$

where f, g are (non-negative) real-valued functions over the program state of C, also referred to as *pre-* and *post-expectation*, respectively. Informally, this judgment states the expected value of g after execution of C is bounded by f. More formally, this judgment is valid if $\mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \le f\ m$ for any initial program memory m, where the left-hand side denotes the expected value of g on the (sub)distribution $[\![\mathtt{C}]\!]_m$ of memories obtained after evaluating C on m.

Coming back to quickselect, the judgment

$$\vdash \{\, 0 \le \mathtt{k} < \mathtt{n} \mid 4(\mathtt{n}-1) \,\}\ \mathtt{qselect\_abs}(\mathtt{n},\mathtt{k})\ \{\, \mathtt{ct} \,\} \qquad \text{(qselect\_abs\_cost)}$$

[5] To be more precise, the judgment guarantees a *probabilistic coupling* of the output distributions within relation Q, as detailed in Section 5.

[6] To slightly simplify proofs, we assume via predicate $\mathtt{unique}(a^{\langle 1\rangle})$ that the input array a to the left procedure contains no duplicate elements.

bounds the expected value of ct after execution by $4(\mathtt{n}-1)$. The guard $0 \le \mathtt{k} < \mathtt{n}$ in the pre-expectation should be understood as a classical pre-condition, for details see Section 5. We have decorated the code of Figure 1b with the corresponding eHL assertions at each line of the listing. The proof of this statement relies again on an auxiliary statement on partitioning, namely,

$$\vdash \{\, \mathtt{l} \le \mathtt{h} \mid \mathtt{ct} + (\mathtt{h}-\mathtt{l}) + \tfrac{1}{\mathtt{h}-\mathtt{l}+1} \textstyle\sum_{\mathtt{i}=\mathtt{l}}^{\mathtt{h}} f(\mathtt{i}) \,\} \ \mathtt{rpartition\_abs}(\mathtt{l},\mathtt{h}) \ \{\, \mathtt{ct} + f(\mathtt{res}) \,\} \qquad (\mathtt{rpartition\_abs\_cost})$$

Here, the free variable f should be understood as a universally quantified, *logical* (function) variable, and as above, res refers to the return value of rpartition_abs. Notice how this statement reflects that the cost counter is advanced by $\mathtt{h}-\mathtt{l}$, and that the return value is sampled uniformly from the interval $[\mathtt{l},\mathtt{h}]$; eHL is in many aspects reminiscent of classical HL. Indeed, the core rules—when restricted to predicates—are identical. As such it transfers Hoare-style backward reasoning to probabilistic programs. Where eHL does depart from HL is the support of sampling instructions S, embodied by the axiom $\vdash \{\, \mathbb{E}_{\mathtt{S}}[\lambda v.\, f[\mathtt{x}/v]] \,\} \ \mathtt{x} \xleftarrow{\$} \mathtt{S} \ \{\, f \,\}$, generalising the the usual axiom for assignments $\vdash \{\, f[x/\mathtt{E}] \,\} \ \mathtt{x} \leftarrow \mathtt{E} \ \{\, f \,\}$. Also the rule of consequence,

$$\frac{f \ge f' \quad \{\, f' \,\} \ \mathtt{C} \ \{\, g' \,\} \quad g' \ge g}{\{\, f \,\} \ \mathtt{C} \ \{\, g \,\}}$$

extends naturally from HL to eHL, implications turn into inequalities. The two axioms together with the consequence rule, tacitly employed before the first statement within the procedure's body, should be sufficient to comprehend the annotations given in Figure 1b around the definition of rpartition_abs.

In a similar fashion, the annotations of qselect_abs can be traced from bottom to top. As in classical HL, the treatment of loop rests on finding a suitable invariant, here it is given by $0 \le \mathtt{l} \le \mathtt{k} \le \mathtt{h} \mid \mathtt{ct} + 4(\mathtt{h}-\mathtt{l})$. Within the loop, the guard $\mathtt{l} < \mathtt{h}$ can be additionally assumed, the guard is falsified immediately after the loop. Concerning the nested conditional in the loop, the term $\mathtt{ct} + \mathtt{g}(\mathtt{i},\mathtt{k},\mathtt{l},\mathtt{h})$ is computed syntactically as the weakest pre-expectation given post-expectation $\mathtt{ct} + 4(\mathtt{h}-\mathtt{l})$. (See the caption for the precise definition of g.) Concerning the call $\mathtt{rpartition\_abs}(\mathtt{l},\mathtt{h})$ the logical variable f is instantiated by the function $\mathtt{i} \mapsto \mathtt{g}(\mathtt{i},\mathtt{k},\mathtt{l},\mathtt{h})$, since the result of the call is bound to i. Interestingly, one recovers, in a formal and syntax-directed way, the recurrence relation of the previous paragraph through the weakening performed in (⋆). Indeed, the (approximate) solution of the recurrence (†) becomes the invariant of the main while loop.

The combination of (equiv_qselect) and (qselect_abs_cost) yields

$$\vdash \{\, \mathtt{unique}(\mathtt{a}) \wedge 0 \le \mathtt{k} < \mathtt{size}(\mathtt{a}) \mid 4(\mathtt{size}(\mathtt{a}) - 1) \,\} \ \mathtt{qselect}(\mathtt{a},\mathtt{k}) \ \{\, \mathtt{ct} \,\} \qquad (\mathtt{qselect\_cost})$$

confirming the linear bound—$\mathrm{O}(\mathtt{size}(a))$—on the expected cost of qselect, derived above in the informal analysis.

Integration within EasyCrypt. The here presented case study on quickselect clarifies the effectiveness of our verification methodology. Relational reasoning provided by pRHL—in particular that employed to guarantee functional correctness—and quantitative reasoning provided by eHL—formalizing the original (informal) complexity proof—work together in a synergistic way. As mentioned, the development is fully formalized (within EasyCrypt), rendering heightened assurance that none of the (necessary) intricacies of a complexity analysis of a randomized algorithm have been overlooked. To this end, EasyCrypt has been extended with support for eHL, see Section 8.

4 A PROBABILISTIC PROGRAMMING LANGUAGE

We consider here a simple imperative probabilistic programming `pWhile` capturing the core language of `EasyCrypt` without adversaries. This language follows the spirit of Dijkstra's *Guarded Command Language* but including (non-recursive) procedures and a separation of global and (statically scoped) local variables. The language will be consecutively extended to permit adversarial code in Section 7, when we discuss applications to cryptography.

Syntax. Let $\mathsf{Fun} = \{\mathsf{f}, \mathsf{g}, \dots\}$ be a *set of procedure names*, and $\mathsf{Var} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \dots\}$ a set of *variables*, partitioned into local variables LVar and global variables GVar. The set Stmt of statements is defined by the following syntax:

$$\mathsf{C}, \mathsf{D} ::= \mathtt{skip} \mid \mathsf{x} \leftarrow \mathsf{E} \mid \mathsf{x} \xleftarrow{\$} \mathsf{S} \mid \mathsf{x} \leftarrow \mathtt{f}(\mathsf{E}) \mid \mathsf{C}; \mathsf{D} \mid \mathtt{if}\ \mathsf{B}\ \mathtt{then}\ \mathsf{C}\ \mathtt{else}\ \mathsf{D} \mid \mathtt{while}\ \mathsf{B}\ \mathtt{do}\ \mathsf{C}$$

Here, $\mathsf{E} \in \mathsf{Expr}$ is drawn from a set of *expressions*, $\mathsf{B} \in \mathsf{BExpr}$ is a *Boolean* expression, and $\mathsf{S} \in \mathsf{SExpr}$ a *sampling expression*. The statements are mostly standard. The statement $\mathsf{x} \leftarrow \mathsf{E}$ gives the usual, deterministic assignment, whereas $\mathsf{x} \xleftarrow{\$} \mathsf{S}$ samples a value from S, and thereby makes the language probabilistic. Statement $\mathsf{x} \leftarrow \mathtt{f}(\mathsf{E})$ calls a procedure with argument E and assigns its return value to x. Zero or more than one argument can be passed to procedures as tuples. We require that x is a local variable. A *procedure* is declared through a *procedure definition* of the form

$$\mathtt{proc}\ \mathtt{f}(\mathsf{x})\ \mathsf{C}; \mathtt{return}\ \mathsf{E},$$

where $\mathsf{x} \in \mathsf{LVar}$ is the *formal parameter*, $\mathsf{C} \in \mathsf{Stmt}$ the *body* and $\mathsf{E} \in \mathsf{Expr}$ the *return expression* of $\mathtt{f}$. Global variables should be understood as implicit input and output to procedures, whereas local ones are statically scoped. A *program* $\mathsf{P} \in \mathsf{Prog}$ is a finite sequence of (mutually exclusive) *procedure definitions*.

Monadic Denotational Semantics. Semantics of imperative programs can be given in many ways. Here, we endow the language with a denotational (monadic) style semantics, lending itself better to the proofs of soundness and completeness of our logic. Since programs are probabilistic, we interpret them as functions from states to subdistributions of states, rather than as mere (partial) state transformers.

A *subdistribution* over a set A is a function $d : A \to [0,1]$ such that $\sum_{a \in A} d(a) \leq 1$, with $\mathsf{D}\,A$ we denote the set of all subdistributions over A. For $d : \mathsf{D}\,A$, the *support* $\mathsf{supp}(d) \subseteq A$ is given by the collection of elements $a \in A$ with $d(a) > 0$. Throughout the following, we consider only discrete subdistributions, that is, where the set A is countable. Let $\mathbb{R}^{+\infty}$ denote the non-negative reals extended with top element ∞. Given function $f : A \to \mathbb{R}^{+\infty}$ and a distribution $d : \mathsf{D}\,A$ we denote by $\mathbb{E}_d[f] \triangleq \sum_{a \in \mathsf{supp}(d)} f(a) \cdot d(a)$ the *expected value of* f *on* d. By the Monotone Convergence Theorem, this value always lies within $\mathbb{R}^{+\infty}$. The subdistribution functor D forms a monad. The *unit* $\mathbf{dunit} : A \to \mathsf{D}\,A$ returns on $a \in A$ the Dirac distribution δ_a (where $\delta_a(b) \triangleq 1$ if $a = b$ and $\delta_a(b) \triangleq 0$ otherwise). The *bind* $\mathbf{dbind} : \mathsf{D}\,A \to (A \to \mathsf{D}\,B) \to \mathsf{D}\,B$ is defined as $\mathbf{dbind}\,d\,f \triangleq \lambda b.\, \sum_{a \in \mathsf{supp}(d)} d(a) \cdot f\,a\,b : \mathsf{D}\,B$. To ease notation, we may write $\mathbf{dlet}\ a \leftarrow d\ \mathbf{in}\ f(a)$ for $\mathbf{dbind}\,d\,(\lambda a.\, f(a))$. With $\mathbf{fail} : \mathsf{D}\,A$ we denote the subdistribution with empty support.

We model program *memories* as mappings $m \in \mathsf{Mem} \triangleq \mathsf{Var} \to \mathsf{Val}$ from variables to (a discrete set of) values Val. Each memory m can be partitioned into a *global memory* $m_g : \mathsf{GMem} \triangleq \mathsf{GVar} \to \mathsf{Val}$ and a *local memory* $m : \mathsf{LMem} \triangleq \mathsf{LVar} \to \mathsf{Val}$. We write $m[\mathsf{x}/v]$ for the memory obtained from m by updating x to v. We suppose that expressions $\mathsf{E} \in \mathsf{Expr}$, Boolean expressions $\mathsf{B} \in \mathsf{BExpr}$ and sampling expressions $\mathsf{S} \in \mathsf{SExpr}$ are equipped with semantics $[\![\mathsf{E}]\!]_{(\cdot)} : \mathsf{Mem} \to \mathsf{Val}$, $[\![\mathsf{B}]\!]_{(\cdot)} : \mathsf{Mem} \to \mathbb{B}$ and $[\![\mathsf{S}]\!]_{(\cdot)} : \mathsf{Mem} \to \mathsf{D}\,\mathsf{Val}$, respectively. Statements C are then interpreted as functions $[\![\mathsf{C}]\!]_{(\cdot)} : \mathsf{Mem} \to \mathsf{D}\,\mathsf{Mem}$, see Figure 2. The definition is mostly standard. Noteworthy,

$\mathtt{C} \in \mathsf{Stmt}$	$[\![\mathtt{C}]\!]_m$
`skip`	$\mathbf{dunit}\, m$
$\mathtt{x} \leftarrow \mathtt{E}$	$\mathbf{dunit}\, m[\mathtt{x}/[\![\mathtt{E}]\!]_m]$
$\mathtt{x} \xleftarrow{\$} \mathtt{S}$	$\mathbf{dlet}\ v \leftarrow [\![\mathtt{S}]\!]_m\ \mathbf{in}\ \mathbf{dunit}\, m[\mathtt{x}/v]$
$\mathtt{x} \xleftarrow{\$} \mathtt{f}(\mathtt{E})$	$\mathbf{dlet}\ (m'_g, r) \leftarrow [\![\mathtt{f}]\!]_{(m_g, [\![\mathtt{E}]\!]_m)}\ \mathbf{in}\ \mathbf{dunit}\, (m'_g \uplus m_l)[\mathtt{x}/r]$
`C; D`	$\mathbf{dlet}\ m' \leftarrow [\![\mathtt{C}]\!]_m\ \mathbf{in}\ [\![\mathtt{D}]\!]_{m'}$
`if B then C else D`	$\begin{cases} [\![\mathtt{C}]\!]_m & \text{if } [\![\mathtt{B}]\!]_m, \\ [\![\mathtt{D}]\!]_m & \text{otherwise.} \end{cases}$
`while B do C`	$\sup_{i \in \mathbb{N}} [\![\mathtt{while}^{(i)}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m$ *where*
$[\![\mathtt{while}^{(0)}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m$	$\triangleq \mathbf{fail}$
$[\![\mathtt{while}^{(i+1)}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m$	$\triangleq \begin{cases} \mathbf{dlet}\ m' \leftarrow [\![\mathtt{C}]\!]_m\ \mathbf{in}\ [\![\mathtt{while}^{(i)}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_{m'} & \text{if } [\![\mathtt{B}]\!]_m, \\ \mathbf{dunit}\, m & \text{otherwise.} \end{cases}$

Fig. 2. Semantics of statements $[\![\cdot]\!]_{(\cdot)} : \mathsf{Stmt} \to \mathsf{Mem} \to \mathsf{D}\,\mathsf{Mem}$.

each procedure `f` is interpreted as a function in $\mathsf{GMem} \times \mathsf{Val} \to \mathsf{D}\,(\mathsf{GMem} \times \mathsf{Val})$, parameterised by the global memory before execution and a value—the formal parameter—and yielding as output a subdistribution of modified global memories and return values. Upon invocation, the local memory is initialised to an *initial memory* m_l^0 assigning to each variable $\mathtt{x} \in \mathsf{LVar}$ a default value, and the formal parameter `x` is bound by the argument. Upon completion, the return value is evaluated and returned, together with the potentially modified global memory. Precisely, we interpret a declaration by

$$[\![\mathtt{proc\ f(x)\ C; return\ E}]\!]_{(m_g, v)} \triangleq \mathbf{dlet}\ m' \leftarrow [\![\mathtt{C}]\!]_{(m_g \uplus m_l^0[\mathtt{x}/v])}\ \mathbf{in}\ \mathbf{dunit}\, (m'_g, [\![\mathtt{E}]\!]_{m'})\,.$$

and we use $[\![\mathtt{f}]\!]_{(m_g, v)}$ as a short-hand when `f` is declared in the program as above.

5 EXPECTATION HOARE LOGIC

In this section, we now present the *Expectation Hoare Logic* (eHL) formally, starting with the core logic and then integrating relational reasoning towards the end of the section.

As seen in Section 3, eHL is designed for reasoning reason about judgments of the form $\{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$, where `C` is a `pWhile` statement and f and g, dubbed *pre-* and *post-expectations* respectively, are functions from states to (non-negative) extended reals. In effect, eHL manipulates slightly more complex judgments in order to address a well-known issue with completeness of proof rules for procedures. In a nutshell, the standard proof systems for procedures aim to achieve modularity by proving for each procedure a *procedure specification*. These are triples of the form $\{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$. For instance, in (rpartition_abs_cost) on page 8, we have employed the specification

$$\{\, l \le h \mid ct + (h - l) + \tfrac{1}{h-l+1} \textstyle\sum_{i=l}^{h} f(i) \,\}\ \mathtt{rpartition\_abs}\ \{\, ct + f(res) \,\}.$$

In this specification, the pre-expectation is parameterised in the argument—here, a tuple (l, h)—whereas the post-expectation is parameterised in the return value *res*. Both may reference global variables like the counter *ct* above—they are implicit input and output of the procedure. Then, modularity is achieved by using the procedure specification every time the procedure is called. Unfortunately, a naive realization of this approach does not achieve completeness. Incompleteness arises because the specification of a function is independent of its call site. Since independence

in itself is desirable for reducing proof effort, the standard compromise is to provide users with a means to adapt a declaration to specific call-sites, to reason about properties potentially involving local state. To this end, we borrow the notion of *auxiliary* (or *logical*) variables from Kleymann [1998]. Auxiliary variables may occur in pre- and post-expectations and are (implicitly) unversally quantified. Effectively, they turn declarations into schemata, where auxiliary variables can be freely instantiated. For instance, in the above specification of `rpartition_abs` we used an auxiliary variable f, with the intended meaning that the triple holds for any concrete instantiation of f. As for Kleymann, auxiliary variables yield a conceptual simple solution to recover (relative) completeness of our logic. With this in mind, we can embark of defining eHL. Our presentation follows closely the presentation of (classical) Hoare Logic HL given by Nipkow [2002b], with pre- and post-expectations given by semantic objects parameterized by a type Z of auxiliary variables, rather than terms or expressions. In eHL, judgments now take one of two forms, namely

$$\vdash_Z \{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \qquad \text{or} \qquad \vdash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\},$$

where $f, g : Z \to \mathsf{Mem} \to \mathbb{R}^{+\infty}$ and $p, q : Z \to \mathsf{GMem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}$. As indicated above, pre- and post-expectations of procedures are parametric only in the global memory. In the pre-expectation p, the additional value argument refers to the formal parameter of `f`, whereas in the post-expectation q it refers to the returned value. To avoid notational overhead, in examples, we will continue to write pre- and post-expectations as expressions, potentially referring to extra auxiliary variables besides program variables. For instance, $Z = \mathbb{Z} \times \mathbb{Z}$ admits two integer valued extra variables, say x and y. If v is a program variable, an expression such as $x + y + v$ formally represents $\lambda(x, y)\, m.\ x + y + m(v)$. In a similar vain, we will use variables *arg* and *res* to refer to the formal parameter and return value within procedure specifications.

eHL is tailored to proving upper-bounds f on the value that a function g takes, in expectation, after running a program. This meaning is made precise through the notion of *validity*.

Definition 5.1 (Validity of Judgments).

(1) A triple $\{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$ is *valid*, in notation $\vDash_Z \{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$, if $\mathbb{E}_{[\![\mathtt{C}]\!]_m}[g\, z] \leq f\, z\, m$ holds for all $z \in Z$ and initial memories $m \in \mathsf{Mem}$, and,

(2) a procedure specification $\{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$ is *valid*, in notation $\vDash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$, if $\mathbb{E}_{[\![\mathtt{f}]\!]_{(m_g, v)}}[q\, z] \leq p\, z\, (m_g, v)$ holds for all $z \in Z$, initial memories $m_g \in \mathsf{GMem}$ and parameters $v \in \mathsf{Val}$.

Finally, through the binary operator $(\mid)$ that we have already used when reasoning about quickselect, we can also combine classical with probabilistic reasoning. Semantically, the operator is defined such that $(\mathtt{true} \mid r) \triangleq r$ and $(\mathtt{false} \mid r) \triangleq \infty$ for any real value $r \in \mathbb{R}^{+\infty}$, and extended to pre- and post-expectations in the obvious way. This way, $\{\, Q \mid f \,\}\ \mathtt{C}\ \{\, P \mid g \,\}$ for instance asserts validity of $\{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$ *under* pre-condition P, guaranteeing post-condition Q.

The Core Rules. Figure 3 presents the core rules of eHL. Interestingly, and what we believe makes the logic in particular usable, is that the core rules are in essence identical in shape to that of classical HL. This is in particular visible in the rules (SKIP), (SEQ) and (ASSIGN). In Rule (ASSIGN), $f[\mathtt{x}/\mathtt{E}]$ is shorthand for $\lambda z\, m.\ f\, z\, m[\mathtt{x}/[\![\mathtt{E}]\!]_m]$. Rule (SAMPLE) generalizes the usual assignment rule to sampling instructions: the pre-expectation $\mathbb{E}_{\mathtt{S}}[\lambda v.f[\mathtt{x}/v]] \triangleq \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{S}]\!]_m}[\lambda v.f\, z\, m[\mathtt{x}/v]]$, is the weakest one binding post-expectation f when $\mathtt{x}$ is sampled from $\mathtt{S}$. For instance, $\vdash_Z \{\, 0.5 \,\}\ \mathtt{x} \xleftarrow{\$} \mathtt{unif}([0,1])\ \{\, x \,\}$ states that in expectation the value of $\mathtt{x}$ is given by 0.5, when sampled uniformly from $\{0, 1\}$.

Rule (IF) is the mere adaptation of the equivalent classical HL rule. The rule descends into the then- and else-branches, where one can additionally assume that the guard and its negation holds, respectively. Concerning loops, rule (WHILE) requires establishing an invariant f on the loops body.

Structural Rules

$$\frac{}{\vdash_Z \{\, f \,\}\ \texttt{skip}\ \{\, f \,\}}\ [\textsc{skip}] \qquad \frac{}{\vdash_Z \{\, f[x/\mathsf{E}] \,\}\ \mathsf{x} \leftarrow \mathsf{E}\ \{\, f \,\}}\ [\textsc{assign}]$$

$$\frac{}{\vdash_Z \{\, \mathbb{E}_{\mathsf{S}}[\lambda v.\, f[\mathsf{x}/v]] \,\}\ \mathsf{x} \xleftarrow{\$} \mathsf{S}\ \{\, f \,\}}\ [\textsc{sample}] \qquad \frac{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, h \,\} \quad \vdash_Z \{\, h \,\}\ \mathsf{D}\ \{\, g \,\}}{\vdash_Z \{\, f \,\}\ \mathsf{C}; \mathsf{D}\ \{\, g \,\}}\ [\textsc{seq}]$$

$$\frac{\vdash_Z \{\, \mathsf{B} \mid f \,\}\ \mathsf{C}\ \{\, g \,\} \quad \vdash_Z \{\, \neg\mathsf{B} \mid f \,\}\ \mathsf{D}\ \{\, g \,\}}{\vdash_Z \{\, f \,\}\ \texttt{if}\ \mathsf{B}\ \texttt{then}\ \mathsf{C}\ \texttt{else}\ \mathsf{D}\ \{\, g \,\}}\ [\textsc{if}] \qquad \frac{\vdash_Z \{\, \mathsf{B} \mid f \,\}\ \mathsf{C}\ \{\, f \,\}}{\vdash_Z \{\, f \,\}\ \texttt{while}\ \mathsf{B}\ \texttt{do}\ \mathsf{C}\ \{\, \neg\mathsf{B} \mid f \,\}}\ [\textsc{while}]$$

$$\frac{\vdash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}}{\vdash_Z \{\, \lambda z\, m.\ p\, z\, (m_g, [\![\mathsf{E}]\!]_m) \,\}\ \mathsf{x} \xleftarrow{\$} \mathtt{f}(\mathsf{E})\ \{\, \lambda z\, m.\ q\, z\, (m_g, m\, \mathsf{x}) \,\}}\ [\textsc{call}]$$

Procedure Declarations

$$\frac{(\texttt{proc}\ \mathtt{f}(\mathsf{x})\ \mathsf{C}; \texttt{return}\ \mathsf{E}) \in \mathsf{P} \qquad \vdash_Z \{\, \lambda z\, m.\ m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}] \mid p\, z\, (m_g, m\, \mathsf{x}) \,\}\ \mathsf{C}\ \{\, \lambda z\, m.\ q\, z\, (m_g, [\![\mathsf{E}]\!]_m) \,\}}{\vdash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}}\ [\textsc{proc}]$$

Logical Rules

$$\frac{\vdash_{Z'} \{\, f' \,\}\ \mathsf{C}\ \{\, g' \,\} \quad \forall m\, d.\ (\forall z' \in Z'.\ \mathbb{E}_d[g'\, z'] \leq f'\, z'\, m) \Rightarrow (\forall z \in Z.\ \mathbb{E}_d[g\, z] \leq f\, z\, m)}{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}}\ [\textsc{conseq}]$$

$$\frac{\vdash_{Z \times \mathsf{Val}} \{\, \lambda(z, v)\, m.\ m\, \mathsf{x} = v \mid f\, z\, m \,\}\ \mathsf{C}\ \{\, \lambda(z, v)\, m.\ g\, z\, m[\mathsf{x}/v] \,\} \quad \mathsf{x} \notin \mathsf{Mod}_{\mathsf{C}}}{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}}\ [\textsc{nmod}]$$

Fig. 3. Kernel rules of eHL.

As in classical HL, the invariant needs to be established only on initial memories making the guard evaluate to true. The rule also establishes that the guard evaluates to false after exiting the loop.

The rule (CALL) allows one to use a procedure specification $\{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$ to reason about a call-site $\mathsf{x} \leftarrow \mathtt{f}(\mathsf{E})$. Recall that p and q are parameterized, beside auxiliary variables and global state, by the formal argument and return value of f, respectively. The rule adapts these to the call-site, by substituting value of the argument E for the formal argument in p, and by identifying the return value of f with that of the assigned variable x within q. Dual to (CALL), rule (PROC) establishes that a procedure $\texttt{proc}\ \mathtt{f}(\mathsf{x})\ \mathsf{C}; \texttt{return}\ \mathsf{E}$ satisfies a specification $\{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$. Here, one essentially has to validate that the procedures body $\mathsf{C}; \texttt{return}\ \mathsf{E}$ adheres to the specification. Following the semantics of procedure calls, the pre-condition $m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}]$ permits one to restrict attention to memories whose local variables are initialised by m_l^0, apart from the formal argument x which ranges over an arbitrary value. This completes the definition of all structural rules.

The final two logical rules deal with auxiliary variables and approximate reasoning, through a *rule of consequence*. A natural candidate for the latter is the rule we have seen in Section 3, corresponding to the *law of monotonicity* in pre-expectation transformers [McIver and Morgan 2005]. Alas, ignoring extra variables, the rule is too weak and its addition alone would render our logic incomplete. Rather, our rule (CONSEQ) is an embodiment of the one of Nipkow [2002b] which is strictly more powerful in the presence of local variables. Observe how the additional premise is just enough to lift validity $\vDash_{Z'} \{\, f' \,\}\ \mathsf{C}\ \{\, g' \,\}$ of the premise to that of the conclusion. Although a bit cumbersome, its generality allows one to derive various rules more useful in practice, such as the simple rule from Section 3. It also encompasses book-keeping rules on auxiliary variables such as

$$\frac{\vdash_Z \{\, f' \,\}\ \mathsf{D}\ \{\, g' \,\} \quad \vDash \{\, P \,\}\ \mathsf{D} \sim \mathsf{C}\ \{\, Q \,\} \qquad \begin{array}{l} \forall z\, m.\ f\, z\, m \neq \infty \Rightarrow \exists m'.\ f'\, z\, m' \leq f\, z\, m \wedge P\, m'\, m \\ \forall z\, m'\, m.\ Q\, m'\, m \Rightarrow g\, z\, m \leq g'\, z\, m' \end{array}}{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}}\ [\textsc{prhl}]$$

Fig. 4. Integration of Relational Hoare Logic

the *instantiation*, or *substitution*, rule

$$\frac{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}}{\vdash_{Z'} \{\, f[z/t] \,\}\ \mathsf{C}\ \{\, g[z/t] \,\}}\ [\textsc{inst}]$$

where t is itself an expression over Z'.

The final rule (NMOD) captures the observation that if a variable is not touched by statement C, it remains constant through evaluation, and can thereby be regarded as an auxiliary variable. In the rule, $\mathsf{Mod}_{\mathsf{C}}$ denotes the set of *variables modified by* $\mathsf{C} \in \mathsf{Stmt}$.[7] The rule gives a mean to internalise the local memory across procedure calls, indispensable in our setup since procedure specifications reference only global memories. This rule, together with the rule of consequence is powerful enough to derive e.g. a *framing rule* based on Jensen's inequality. We elaborate more on that in Section 8.

Theorem 5.2 (Soundness and Completeness). *For all procedures* f,

$$\vdash_Z \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \qquad \Leftrightarrow \qquad \vDash_Z \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$$

The proof is this theorem is given in the appendix.

Relational Reasoning. Formally reasoning about the complexity of intricate programs can be very hard. However, complexity can often be studied on *simplified* (but complexity preserving) versions of the original programs with much less burden. *Probabilistic relational Hoare logic* (pRHL for short) allows one to formally relate two programs that behave *the same* [Barthe et al. 2015, 2012, 2017]. Judgments have the following form:

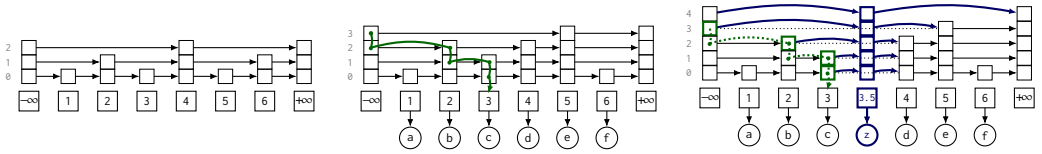
$$\vdash \{\, P \,\}\ \mathsf{C} \sim \mathsf{D}\ \{\, Q \,\},$$

where $P, Q \subseteq \mathsf{Mem} \times \mathsf{Mem}$ are both assertions that relate memories of C and D. The intuitive meaning behind this judgment is that, when programs C and D are run on initial memories related by P, the resulting output-distributions are coupled via relation Q. Probabilistic coupling is formalised via the notion of *relational lifting* of Q to a relation $Q^\dagger \colon \mathsf{D}\,(\mathsf{Mem}) \times \mathsf{D}\,(\mathsf{Mem})$. Precisely, $d_1\ Q^\dagger\ d_2$ iff there exists a (sub)distribution $d \in \mathsf{D}\,(\mathit{Mem} \times \mathsf{Mem})$ such that (i) the marginal (sub)distributions of d are d_1 and d_2; and (ii) $\mathsf{supp}(d) \subseteq Q$. We are now ready to state the definition of *validity* of a pRHL judgment:

Definition 5.3 (Validity of pRHL *Judgments).* Judgment $\vdash \{\, P \,\}\ \mathsf{C} \sim \mathsf{C}'\ \{\, Q \,\}$ is *valid*, in notation $\vDash \{\, P \,\}\ \mathsf{C} \sim \mathsf{C}'\ \{\, Q \,\}$, if for all memories $m_1, m_2 \in \mathsf{Mem}$ such that $m_1\ P\ m_2$, then $[\![\mathsf{C}]\!]_{m_1}\ Q^\dagger\ [\![\mathsf{C}']\!]_{m_2}$.

The proof system underlying pRHL is extensively described in the literature [Barthe et al. 2015, 2012, 2017]. Noteworthy, an implementation is available in EasyCrypt. Here, the notion of validity is sufficient to relate eHL with pRHL. Indeed, we would like to transfer eHL properties from one program C' to a potentially more complex one C. The rule in Figure 4 allows for just that. Concerning post-expectations, the second side-condition is sufficient to establish $\mathbb{E}_d[g] \leq \mathbb{E}_{d'}[g']$ for any coupling $d\ Q^\dagger\ d'$. Through the pRHL judgement, this holds in particular for the output distributions of C and C', on any pair of initial memories m and m' related by P. The first side-condition now essentially demands that each initial m of C can be paired with a memory m' of C' related through P, but also through the pre-expectations. From here, soundness is not difficult to establish.

[7] To be precise, a variable x is modified by C if $m\,\mathsf{x} \neq m'\,\mathsf{x}$ for some initial memory m and final memory $m' \in \mathsf{supp}([\![\mathsf{C}]\!]_m)$.



(a) List with two extra levels of balanced forward pointers.

(b) Searching for value associated to key k = 3 in random skip list.

(c) Inserting z with key k at sampled height 5 in (b).

Fig. 5. Several representations of skip lists over elements $[1, \ldots, 6]$. Figure (a) depicts a perfectly balanced skip list. Figure (b) depicts the dictionary $\{1 \mapsto \mathsf{a}, \ldots, 6 \mapsto \mathsf{f}\}$ implemented on top of a (random) skip list. The search path for value c with key 3 is indicated as a solid green arrow. Figure (c) is obtained from (b) by inserting an element with key 3.5, with a sampled height of ht = 5. The dotted green arrow indicates the search path followed by `insert(3.5, z)` to determine the position of the new node, it is identical to the path from (b). The search path array sp is outlined with thick green borders, note that it is given by those nodes on the search path where search proceeded downwards. The bended thick blue arrows indicate new pointers.

Proposition 5.4. *Rule (prhl) is sound.*

6 AVERAGE CASE COMPLEXITY OF SKIP LISTS

In this section, we demonstrate the flexibility of our framework via a complexity analysis of the *skip list* data structure. Skip lists have been introduced in [Pugh 1990b] as a randomized alternative to balanced binary trees that is easier to implement and maintain.[8] Being a probabilistic data structure, their formal average case complexity analysis is intricate. A skip list can be thought of as an ordered linked list, where nodes may have additional forward pointers skipping several nodes ahead so as to facilitate a more efficient search. Forward pointers are organised in levels, each level skipping ahead (ideally) half of the nodes found in the level below. By introducing $\log_2(n)$ levels for a list of n elements, search becomes effectively a $O(\log_2(n))$ operation. For illustration, Figure 5a shows a (perfectly balanced) skip list with three levels of forward pointers, organized as a stack above keys. Element $-\infty$ and $+\infty$, acting as head and terminator of the list, respectively.

Maintaining perfect balance of forward pointers when elements are inserted or deleted is a costly operation. In a skip list, forward pointers are chosen at random, so that when inserting a node, it is assigned a forward pointer at level $l+1$ with a certain probability p, in case there is a forward pointer also on level l. For $p = 1/2$ (the standard case, which we fix below for simplicity), this means that level $l+1$ will have around half the nodes of level l; maintaining a balancing of forward pointers similar to that of the perfectly balanced case in the common case. This almost perfect balancing is precisely the reason why skip lists have good complexity properties, *on average*.

6.1 A Dictionary Implementation on Top of Skip Lists

Dictionaries are prime examples of data structures implemented on top of skip lists. Figure 6a shows such an implementation, for brevity we consider only search and insertion. This implementation maintains an explicit, global memory `nodes` of nodes `node(key, data, fwd)` consisting of a key `key`, a datum `data` and an array of forward pointers `fwd`. We assume keys are equipped with a partial order. As indicated above, we always assume the presence of two nodes with minimal key $-\infty$ and maximal key $+\infty$. As in the quickselect example, a global counter `ct` is used to measure costs.

[8] In quote "Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space." [Pugh 1990a].

```
var ct;
var nodes;
proc find(k)
  (p, lvl) ← (ptr_{-∞}, height(ptr_{-∞}));
  while 0 ≤ lvl do
    q ← get_fwd(p, lvl);
    if k < get_key(q)
    then lvl−− else p ← q;
    ct++
  if get_key(p) = k
  then return get_data(p)
  else return null

proc insert(k, d, hk)
  (spf, k′) ← find_path(k);
  if k = k′ then set_data(k′, d)
  else
    fwd ← new_array(hk, ptr_{+∞});
    q ← fresh_ptr();
    h ← height(ptr_{-∞});
    for k = 0 to min(h, hk) − 1 do
      fwd[k] ← get_fwd(spf[k], k);
      set_fwd(spf[k], k, q)
    set_ptr(q, node(k, d, fwd));
    set_fwds(ptr_{-∞}, [h..hk − 1], q);

proc from_list(lst)
  nodes ← empty();
  while lst ≠ nil do
    (k, d)::lst ← lst;
    hk ←$ geo(1/2) + 1;
    insert(k, d, hk)

proc find_cost(lst, k)
  from_list(lst);
  ct ← 0; find(k);
  return ct
```

(a) Concrete implementation.

```
var hts;
proc path_len_to(k)
  keys ← decr(keys(hts));
  (len, l) ← (0, −1);
  while keys ≠ nil do
    _::keys ← keys;
    if l < hts[k] ∧ head(keys) ≤ k
    then
      len ← len + hts[k] − l;
      l ← hts[k] − 1
  return (len + (hts[−∞] − 1) − l)

proc insert_h(k, hk)
  if k ∈ dom(hts) then skip
  else
    hts[k] ← hk;
    hts[−∞] ← max(hts[−∞], hk);

proc from_list_h(keys)
  hts ← { −∞ ↦ 1; +∞ ↦ 0};
  while keys ≠ nil do
    k::keys ← keys;
    hk ←$ geo(1/2) + 1;
    insert_h(k, hk)

proc find_cost_h(lst, k)
  from_list(keys(lst));
  ct ← path_len_to(k);
  return ct
```

(b) Height abstraction.

```
// 2 · log_2(size(lst) + 1) + 4
proc find_cost_d(lst, k)
  // 2 · log_2(size(lst) + 1) + 4
  // (⋆_1)
  // Δ_l(m) + Δ_0(m) − 1
  (len, l, h) ← (0, −1, 1);
  keys ← decr_uniq(keys(lst));
  // φ | len + (h − 1 − l) + Δ_h(n) + Δ_{l+1}(n)
  while keys ≠ nil do
    // keys ≠ nil ∧ φ | len + (h − 1 − l)
    //                + Δ_h(n) + Δ_{l+1}(n)
    _::keys ← keys;
    // φ | len + (h − 1 − l)
    //     + Δ_h(n + 1) + Δ_{l+1}(n + 1)
    // (⋆_2)
    // E_δ[λhk.
    //     φ | len + (max(h, hk + 1) − 1 − l)
    //       + Δ_{max(h,hk+1)}(n)
    //       + if l < hk ∧ head(keys) then
    //           Δ_{hk+1}(n) + 1 else Δ_{l+1}(n)
    hk ←$ geo(1/2) + 1;
    h ← max(h, hk);
    // φ | len + (h − 1 − l) + Δ_h(n)
    //     + if l < hk ∧ head(keys) then
    //         Δ_{hk+1}(n) + 1 else Δ_{l+1}(n)
    if l < hk ∧ head(keys) ≤ k then
      len ← len + hk − l;
      l ← hk − 1
    // φ | len + (h − 1 − l) + Δ_h(n) + Δ_{l+1}(n)
  // keys = nil ∧ φ | len + (h − 1 − l) + Δ_h(n)
  //                + Δ_{l+1}(n)
  // (⋆_3)
  // 0 ≤ len + (h − 1 − l) | len + (h−1−l)
  return len + (h − 1 − l)
// 0 ≤ res | res
```

(c) Final cost function, annotated.

Fig. 6. Skip list implementation of insertion and search, its height abstraction and final cost function. In (c), $n \triangleq$ `size(keys)`, $m \triangleq$ `size(uniq_decr(keys(lst)))`, and δ is the geometric distribution with parameter $1/2$. In the invariant attributed to the loop, $\phi \triangleq 0 \le \mathtt{len} \wedge -1 \le \mathtt{l} < \mathtt{h} \wedge 1 \le \mathtt{h} \wedge 0 \le \mathtt{n}$; and $\Delta_\ell(n) \triangleq \log_2(n+1) - \ell$ if $\ell \le \lceil \log_2(n+1) \rceil$ and $\Delta_\ell(n) \triangleq \frac{n}{2^{\ell-1}}$ otherwise.

Search. The procedure `find(k)` implements lookup of a datum `d` associated to `k`, returning `null` if no such datum can be found. Search proceeds from the top level at $-\infty$, following forward pointers as long as not overshooting, decrementing the level otherwise. Search eventually reaches level `lvl` $= -1$, and stops at the last entry whose key is still bounded by `k`. Figure 5b illustrates the search of key 3, highlighting the *search path* traversed by `find(3)`. The implementation increments the cost counter `ct`, whenever a comparisons of keys—`k` < `get_key(q)`—is performed. Observe how this cost measure is directly related to the length of the search path.

Insertion. Inserting a datum `d` with key `k` involves finding first the location for the given `k`, and linking a new node within skip list in case the key `k` is not present. Figure 5c illustrates insertion of a datum `z` with unoccupied key 3.5. The size of the array of forward pointers `fwd` is drawn at random. Pre-existing pointers that would "skip through the new column" are separated in two, pointing now to and from the new node, respectively. Finally, the forward pointers of $-\infty$ are extended, in case insertion increases the maximal level, as in the Figure. The full implementation of insertion is given by procedure `proc insert(k, d)` in Figure 6a. It uses a variation `find_path` of `find` that returns an array `spf` of forward pointers on the search path where search took a

downward turn—those that will link to a newly inserted entry—together with the key k′ where search terminated. Insertion incurs no cost, as we will be interested in the complexity of search.

Average search complexity. In what follows, we outline our formalization on the *search complexity*—the number of comparisons performed within a search—of skip lists. To this end, our starting point is the function `find_cost(lst, k)` which searches for key k in a skip list, built from the provided list of key/value pairs `lst`. The procedure returns the cost counter, storing the number of comparisons performed by `find`. Since the skip list is constructed at random through the implementation of `insert`, the expectation of the return value `ct` reflects precisely the average search complexity.

6.2 Outline of the Formalization

Height abstraction. Since pointers in a skip list always point forward to the first node of sufficient height, the structure of a skip list is fully determined by the height of nodes, i.e. the size of their array of forward pointers. This, in turn, justifies to abstract nodes by there height, specifically, we have the following mapping in mind:

$$\texttt{hts(nodes)} \triangleq \{\texttt{k} \mapsto \texttt{size(fwd)} \mid \texttt{p} \in \texttt{dom(nodes)}, \texttt{nodes(p)} = \texttt{node(k, d, fwd)}\}$$

As `insert` is mostly concerned with managing the pointer structure after update, this abstraction considerable simplifies its implementation, see Figure 6b. Correctness of this abstraction is justified by the (classical) Hoare judgment[9]

$$\vdash_{\mathsf{HL}} \{\, \texttt{wf(nodes)} \land \texttt{hs} = \texttt{hts(nodes)} \,\}\ \texttt{insert(k, d, hk)}\ \{\, \texttt{wf(nodes)} \land \texttt{hts} = \texttt{upd(hs, k, hk)} \,\} \tag{insert_spec}$$

where `upd(hs, k, hk)` updates the height of k to hk in hs only in the case when $\texttt{k} \notin \texttt{dom(hs)}$. The predicate `wf(nodes)` collects several well-formedness conditions expressing that `nodes` forms a skip list (eg, keys are ordered, pointers reference the first larger key, etc). Reasoning inductively, this auxiliary result establishes the following correspondence:

$$\begin{array}{l}\vdash \{\, \texttt{keys(lst}^{\langle 1\rangle}\texttt{)} = \texttt{keys}^{\langle 2\rangle} \,\} \\ \quad \texttt{from\_list(lst)} \sim \texttt{from\_list\_h(keys)} \\ \quad \{\, \texttt{wf(nodes}^{\langle 1\rangle}\texttt{)} \land \texttt{hts(nodes}^{\langle 1\rangle}\texttt{)} = \texttt{hts}^{\langle 2\rangle} \,\}\end{array} \tag{equiv_from_list}$$

As we have alluded to already above, the search complexity corresponds to the length of the search path. Formally, this statement is expressed by the (classical) Hoare triple

$$\vdash_{\mathsf{HL}} \{\, \texttt{wf(nodes)} \land \texttt{cost} = \texttt{ct} \,\}\ \texttt{find(k)}\ \{\, \texttt{ct} = \texttt{cost} + \texttt{path\_len(hts(nodes), k)} \,\} \tag{find_spec}$$

where `cost` refers to the value of the cost counter before execution, and where `path_len(hts, k)` expresses the length of the search path to key k. It is worth mentioning that the proof of this judgment depends crucially on `wf(nodes)`. For instance, would `nodes` contain a cycle, `find(k)` would potentially loop and no bound on `ct` could be derived. This explains why we have proven preservation of well-formedness—in essence functional correctness—of insertion. Indeed, this turned out to be the most delicate part in the proof of (equiv_from_list).

By (find_spec), to analyze the search complexity it is sufficient to bound the length of the search path `path_len(hts(nodes), k)`, which in turn is computable within the abstraction. The procedure `path_len_to(k)`, given in Figure 6b, gives an explicit definition of the search path length. To give some intuition about the definition, reconsider the search path for key $\texttt{k} = 3$ depicted in Figure 5b. The procedure starts by scanning keys in reverse-order, pictorially from right to left, until it reaches

[9]Here and below, we denote Hoare judgments that should be interpreted in the classical sense by $\vdash_{\mathsf{HL}} \{\, \phi \,\}\ \mathtt{C}\ \{\, \psi \,\}$. We have also done a similar judgment establishing the functional correctness of the data part, i.e that the skip list data structure can be used as a dictionary.

`head(keys) = 3`. From now on, the procedure traverses the search path in reverse-order, starting at level `l = −1`. Observe that search reaches a new key always through the top-most incoming forward pointer. Correspondingly, the backward traversal moves up by raising the level `l` to the maximal level and by incrementing the length `len` of the path traversed so far, accounting for the upward moves and the move to the left. From here, the procedure iterates. In the example, at key `3` the procedure moves this way to level `l = 1` advancing `len = 0` to `len = 3`, accounting for the upward two moves and the move to the left. The procedure then iterates, to key `2`, skipping along key `1` not on the search path (due to the condition `l < hts[k]`), until finally arriving at $-\infty$. The final increment in the return statement accounts for the final move upwards on key $-\infty$, in the example from level 2 to level 3. With this intuition in mind, functional correctness

$$\vdash_{\mathsf{HL}} \{\, \mathtt{hs} = \mathtt{hts} \,\}\ \mathtt{path\_len\_to(k)}\ \{\, \mathtt{res} = \mathtt{path\_len(hs, k)} \,\} \qquad \text{(path\_len\_to\_spec)}$$

is easily provable in classical Hoare logic. Summing up, the following relational Hoare judgment state correctness of the abstraction with respect to search complexity.

$$\vdash \{\, \mathtt{lst}^{\langle 1\rangle} = \mathtt{lst}^{\langle 2\rangle} \wedge \mathtt{k}^{\langle 1\rangle} = \mathtt{k}^{\langle 2\rangle} \,\}\ \mathtt{find\_cost(lst, k)} \sim \mathtt{find\_cost\_h(lst, k)}\ \{\, \mathtt{res}^{\langle 1\rangle} = \mathtt{res}^{\langle 2\rangle} \,\} \qquad \text{(equiv\_find\_cost\_h)}$$

It is a direct consequence of (equiv_from_list), (find_spec) and (path_len_to_spec).

Estimation of the path length through the height abstraction. The judgment (equiv_find_cost_h) formally justifies that we analyze the search complexity through its height abstraction given in Figure 6b. The crux in proving the latter directly is to find a suitable upper invariant for the loop in `from_list_h`. In effect, this requires expressing the search path length after inserting a column, in terms of of the search path length before the insertion. At the same time, this invariant has to lead to a sufficiently tight bound in the size of `keys`. However, this technicality can be avoided altogether, by sampling `hts` on-demand, rather than eagerly. The procedure `find_cost_d`, given in Figure 6c, is obtained by inlining `path_len_to` within `find_cost_d` from Figure 6b. Heights `hk` corresponding to `hts[k]` are sampled on demand-within the path traversal, rendering the call to `from_list_h` obsolete. The auxiliary variable `h` refers to the maximal sampled height, viz. the height of $-\infty$. One can prove that semantically, `find_cost_h` and `find_cost_d` coincide:

$$\vdash \{\, \mathtt{lst}^{\langle 1\rangle} = \mathtt{lst}^{\langle 2\rangle} \wedge \mathtt{k}^{\langle 1\rangle} = \mathtt{k}^{\langle 2\rangle} \,\}\ \mathtt{find\_cost\_h(lst, k)} \sim \mathtt{find\_cost\_d(lst, k)}\ \{\, \mathtt{res}^{\langle 1\rangle} = \mathtt{res}^{\langle 2\rangle} \,\} \qquad \text{(equiv\_find\_cost\_d)}$$

Notice that the left program inserts keys in the order they occur in `lst`, whereas the right program processes keys in reverse-sorted order, removing duplicates. Thus, a rather involved key step towards this equivalence is proving that path length is independent of the order of insertions.

Final cost analysis via eHL. The judgments (equiv_find_cost_h) and (equiv_find_cost_d) establish

$$\vdash \{\, \mathtt{lst}^{\langle 1\rangle} = \mathtt{lst}^{\langle 2\rangle} \wedge \mathtt{k}^{\langle 1\rangle} = \mathtt{k}^{\langle 2\rangle} \,\}\ \mathtt{find\_cost(lst, k)} \sim \mathtt{find\_cost\_d(lst, k)}\ \{\, \mathtt{res}^{\langle 1\rangle} = \mathtt{res}^{\langle 2\rangle} \,\} \qquad \text{(equiv\_find\_cost)}$$

witnessing that the complexity of searching for a key `k` in an arbitrary skip list build from `lst` is computed by `find_cost_d(lst, k)`. The final puzzle piece lies now in bounding this result, in expectation. To this end, we make use of eHL, compare the assertions in Figure 6c. The gist of the proof lies in finding an invariant for the loop. As the definition and the related weakening proofs are quite technically involved, we have relegated further discussion to the Appendix. Very briefly, terms $\Delta_h(n)$ and $\Delta_{l+1}(n)$ are used to account for changes to the path length, through vertical and horizontal steps, respectively. Concerning horizontal steps for instance, in the common case where

the current height h does not exceed the (average) logarithmic overall height, $\Delta_h(n) = \log_2(\frac{n+1}{2^h})$ measures the expected height increase of completing the loop in terms of the $\frac{n+1}{2^h}$ nodes found at current height h. The invariant turns slightly more complicated, to also account for the final difference $\mathsf{h} - 1 - \mathsf{l}$ contributing to the result of the procedure (see weakening $(\star_1)$). Once carried over the initialisation statements (see weakening $(\star_2)$), the invariant gives the final logarithmic bound $2 \cdot \log_2(\mathtt{size}(lst)+1)+4$. Apart from defining the invariant, the most delicate step concerned the proof of the weakening step $(\star_2)$. Towards this proof, we have build a considerate library on laws of expectations, such as the law of linearity, Jensen's inequality, etc.

Concluding Remarks. Splitting the correctness proof, done via pRHL, from the complexity analysis, carried on via eHL, seems essential to achieve our goal. The modularity provided by our framework has allowed us to develop the proof step-by-step, in a compositional way, which would not have been possible without the EasyCrypt implementation.

7 ADVERSARIES AND APPLICATIONS TO CRYPTOGRAPHIC PROOFS

In this section, we extend our programming language and logic with adversary calls, and illustrate how the extended logic can be used to reason about cryptographic proofs. Our example is inspired from a recent work by [Barbosa et al. 2023], which uses our implementation of eHL for proving security of Dilithium [Ducas et al. 2017], a post-quantum signature scheme recently standardized by the NIST (National Institute of Standards and Technology).

Extension of the language. We now extend the language to adversarial code by permitting adversary calls $\mathsf{x} \leftarrow \mathcal{A}_\mathsf{o}(\mathsf{E})$, where $\mathcal{A}$ is drawn from a set $\mathsf{Adv} = \{\mathcal{A}, \mathcal{B}, \dots\}$ of *adversary names*. Each adversary call is parameterised by an *oracle*, i.e., a pre-defined procedure $\mathsf{o} \in \mathsf{Fun}$.[10] Adversaries $\mathcal{A}$ refer to arbitrary procedures, granted only partial access to the global memory through a set $\mathsf{Write}_\mathcal{A} \subseteq \mathsf{GVar}$ of *writable global variables*. In a call to $\mathcal{A}_\mathsf{o}$, the adversary may modify variables outside $\mathsf{Write}_\mathcal{A}$ only by invoking the oracle o. To model adversarial code in the semantics, we index the interpretation of program statements by an *adversary environment* γ. This environments maps each $\mathcal{A} \in \mathsf{Adv}$ to a declaration

$$\gamma(\mathcal{A}) = o \mapsto (\mathtt{proc}\ \mathcal{A}(\mathsf{x})\ \mathsf{C}_o\mathtt{;}\ \mathtt{return}\ \mathsf{E}),$$

indexed by an oracle o. Note that the code of the adversary is parametric in the oracle. The body C_o may contain *oracle calls* $\mathsf{x} \leftarrow o(\mathsf{E})$. Invocation of $\mathcal{A}_\mathsf{o}$ executes the procedure $\gamma(\mathcal{A})(\mathsf{o}) = \mathtt{proc}\ \mathcal{A}(\mathsf{x})\ \mathsf{C}_\mathsf{o}\mathtt{;}\ \mathtt{return}\ \mathsf{E}$, where in the body the meta-variable o has been substituted by the provided oracle o. We require that adversary environments are consistent with writeable variables, i.e., the body of $\gamma(\mathcal{A})(\mathsf{o})$, nor any of its subprocedures except the oracle o, modifies the memory outside of $\mathsf{Write}_\mathcal{A}$. For instance, if the adversary executes an instruction $\mathsf{x} \leftarrow \mathsf{E}$, then $\mathsf{x} \in \mathsf{Write}_\mathcal{A}$. In contrast to the notion of modified variables Mod_C, which is semantic, $\mathsf{Write}_\mathcal{A}$ is a syntactic notion with subtle differences. The memory content of a variable $\mathsf{x} \notin \mathsf{Write}_\mathcal{A}$ may change during an invocation, but only through invocations of the oracle. The semantics of an adversarial call are now identical to ordinary procedure calls, just, the declaration of the adversary is provided by the adversary environment γ, that is, we let $[\![\mathcal{A}_\mathsf{o}]\!]^\gamma = [\![\gamma(\mathcal{A})(\mathsf{o})]\!]$, but treat a call $\mathsf{x} \leftarrow \mathcal{A}_\mathsf{o}(\mathsf{E})$ otherwise identical to an ordinary procedure call.

Extension of eHL. To extend the logic for programs with adversarial code, the notion of judgment can remain identical, apart from the fact that program statements now may contain adversarial calls. However, judgments will now be *valid* if validity in the original sense holds *independent*

[10] In practice, we permit $\mathcal{A}$ to be parameterised by more than one oracle. Here, the restriction helps us avoid notational overhead.

```
⫽ 1/δ + size(log)
proc rsample()
  var t, r;
  ⫽ 1/δ + size(log)
  t ← false;
  while ¬t do
    ⫽ ¬t | ¬t/δ + size(log)
    ⫽ E_sample[λr. (¬test(r))/δ + size(r::log)]
    r ←$ sample();
    log ← r::log;
    t ← test(r);
    ⫽ ¬t/δ + size(log)
  ⫽ t | ¬t/δ + size(log)
⫽ size(log)
```

(a) Logged rejection sampling.

```
⫽ φ | if Q ≤ c then bad else F
proc o()
  var r;
  ⫽ φ | if Q ≤ c then bad else F
  c ← c + 1;
  ⫽ φ | if Q < c then bad else ε/δ + F
  rsample();
  ⫽ φ | if Q < c then bad else F
  ⫽ if c = Q then E_sample[λr.r ∈ log]
    else φ | if Q ≤ c then bad else F
  if c = Q then
    r* ←$ sample();
    bad ← r* ∈ log;
⫽ φ | if Q ≤ c then bad else F
```

(b) Oracle.

```
⫽ ε · Q/δ
proc game()
  ⫽ ε · Q/δ
  ⫽ true | if Q ≤ 0 then bad
        else ε · 0 + ε/δ · (Q − 0)
  bad ← false;
  c ← 0;
  log ← nil;
  ⫽ φ | if Q ≤ c then bad else F
  A_o()
  ⫽ φ | if Q ≤ c then bad else F
⫽ bad
```

(c) Main program

Fig. 7. Rejection sampling with bad. Variables c, log and bad are global. Here, $0 \le \mathtt{Q}$ is a constant, $\delta \triangleq \Pr[\mathtt{sample} : \mathtt{test}] > 0$ is the probability of event test on the distribution given by sample, $\Pr[\mathtt{sample} : \mathbf{1}_v] \le \epsilon$ is an upper-bound on the probability of sampling a value v; $\phi \triangleq \mathsf{bad} \Rightarrow \mathtt{Q} \le \mathtt{c}$ and $F \triangleq \epsilon \cdot (\mathtt{size}(\mathtt{log}) + \frac{\mathtt{Q}-\mathtt{c}}{\delta})$.

of the adversarial code, that is, $\vDash_Z \{ f \}\ \mathtt{C}\ \{ g \}$ if $\mathbb{E}_{[\![\mathtt{C}]\!]^\gamma_m}[g\, z] \le f\, z\, m$ holds for all z, m and *all adversary environments* γ. Similar, validity for procedure declarations is defined by quantifying over all adversary environments.

The following now gives our adversarial rule, for $f : Z \to \mathsf{GMem} \to \mathbb{R}^{+\infty}$ depending only on the *global memory*.

$$\frac{f \perp \mathsf{Write}_{\mathcal{A}} \quad F = \lambda z\, (m_g, \_).\, f\, z\, m_g \quad \vDash_Z \{ F \}\ \mathtt{o}\ \{ F \}}{\vDash_Z \{ F \}\ \mathcal{A}_{\mathtt{o}}\ \{ F \}}\ [\textsc{adv}]$$

This rule lifts invariants on oracles to that of adversaries. The hypothesis $f \perp \mathsf{Write}_{\mathsf{Adv}}$, stating that f is independent of writable variables by the adversary, ensures that F remains invariant throughout complete invocation of the adversary.

Theorem 7.1. *Rule (adv) is sound.*

Example. We illustrate how eHL can be used to upper bound the probability of bad events in rejection sampling. The example captures the essence of a key step in the security proof of the Dilithium signature scheme, formalized in Barbosa et al. [2023] using our implementation of eHL. Our goal is to provide an upper-bound on the probability that a fresh, random value appears in the history of samplings performed during rejection sampling. This stage of the proof is represented, in slightly simplified form, in Figure 7. Procedure rsample (Figure 7a) performs rejection sampling from distribution sample with predicate test. The global variable log keeps track of all sampled values. Each invocation of the oracle o, later provided to the adversary, performs rejection sampling and thereby populates log. A global counter c keeps track of the number of oracle invocations. Once the counter reaches $0 \le \mathtt{Q}$, a bad event is signaled through setting the global variable bad, precisely when log contains a randomly sampled value $\mathtt{r}^*$. The main program (Figure 7c) consists simply of a call to the adversary $\mathcal{A}_{\mathtt{o}}$, with global auxiliary global variables initialised correspondingly. The adversary has access to the global variables only through the oracle, that is, $\mathsf{Write}_{\mathcal{A}} = \varnothing$. Our goal is to bind the probability of the Boolean variable bad—its expectation—within this program.

Figure 7 is annotated with the corresponding eHL proof. The central proof step lies in annotating the oracle in Figure 7b with an invariant binding the value of bad. Being initialized to false, this variable is only set once the invocation counter c of the oracle reaches Q, and then only when a fresh sampled value $\mathtt{r}^*$ collides with a previously sampled value in log. In turn, the probability of a

$$\frac{g \leq f}{\vdash_Z \{\, f \,\}\ \texttt{skip}\ \{\, g \,\}}\ [\textsc{skipEc}] \qquad \frac{\vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, g[\texttt{x}/\texttt{E}] \,\}}{\vdash_Z \{\, f \,\}\ \texttt{C; x} \leftarrow \texttt{E}\ \{\, g \,\}}\ [\textsc{assignEc}] \qquad \frac{\vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, \texttt{wp(D,}\, g\texttt{)} \,\}}{\vdash_Z \{\, f \,\}\ \texttt{C; D}\ \{\, g \,\}}\ [\textsc{wpEc}]$$

$$\frac{\vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, g \,\} \quad \forall z\, v.\ (\lambda m.\ F\, z\, m\, v) \perp \mathsf{Mod}_{\texttt{C}} \quad \forall z\, m.\ F\, z\, m \text{ concave, non-decreasing}}{\vdash_Z \{\, \lambda z\, m.F\, z\, m\, (f\, z\, m) \,\}\ \texttt{C}\ \{\, \lambda z\, m.F\, z\, m\, (g\, z\, m) \,\}}\ [\textsc{frameEc}]$$

$$\frac{\begin{array}{c}\vdash_Z \{\, p \,\}\ \texttt{f}\ \{\, q \,\} \\ \vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, \lambda z\, m.\ (\forall r\, \vec{v}.\ g\, z\, m[\texttt{x}/r][\mathsf{Mod}_{\texttt{f}}/\vec{v}] \leq q\, z\, (m_g[\mathsf{Mod}_{\texttt{f}}/\vec{v}], r)) \mid p\, z\, (m_g, [\![\texttt{E}]\!]_m) \,\}\end{array}}{\vdash_Z \{\, f \,\}\ \texttt{C; x} \leftarrow \texttt{f(E)}\ \{\, g \,\}}\ [\textsc{callEc}]$$

Fig. 8. Excerpt of derived rules implemented in EasyCrypt.

collision $\texttt{r}^* \in \texttt{log}$ is bounded from above by $\epsilon \cdot \texttt{size(log)}$, for ϵ an upper-bound on probabilities of sample. This, in effect, allows us to estimate the value of bad in terms of the size of log when c reaches Q. To this end, let $0 < \delta$ be the probability that a sample satisfies the predicate test. As indicated in Figure 7a, rejection sampling increases the length of log, on average, by $\frac{1}{\delta}$.

The invariant given in the specification (see Figure 7b) lifts this observation to the oracle. In the term $F = \epsilon \cdot (\texttt{size(log)} + \frac{\texttt{Q}-\texttt{c}}{\delta})$, the factor ϵ stems from the approximation of bad in terms of the size of log, the fraction $\frac{\texttt{Q}-\texttt{c}}{\delta}$ accounts the potential size increase of log until the invocation counter reaches the limit Q. Once the counter is reached, the invariant simply refers to the value of bad. The overall program is now treated essentially by an application of the adversary rule, using the invariant on the oracle as provided, see Figure 7c. The weakening at the end follows from the classical invariant ϕ. The derived bound $\epsilon \cdot \frac{\texttt{Q}}{\delta}$ is obtained by simplification of the invariant with global variables initialised correspondingly.

The proof hinges essentially on the fact that, on average, the size of log is bounded, although rsample is potentially non-terminating and may produce a log of arbitrary size. Lacking capabilities for expection based reasoning, this renders a proof using the phoare logic present in EasyCrypt significantly more involved. The most natural way here is to proceed via an approximation of rejection sampling so that the number of iterations is bounded, say by a constant K. Thereby, within the Q invocations of the oracle, the size of log becomes bounded by $\texttt{Q} \cdot K$, worst case. On the so transformed, certainly terminating, program, one can then obtain a bound $\texttt{Q} \cdot K \cdot \epsilon$ on the probability of bad being set. The approximation itself however, introduces an additional error rate, leading to the overall bound of $\texttt{Q} \cdot K \cdot \epsilon + \texttt{Q} \cdot \delta^K$.

In contrast, the use of eHL not only significantly reduced proof effort, it also lead to a more preferable bound. The complete formal proof in EasyCrypt takes in total only 48 lines. The frame rule (detailed in the next section) turned out particularly useful. It allowed us to lift the specification of rsample, talking only about the expected size increase of log, to the call within the oracle o.

8 IMPLEMENTATION

We have implemented eHL in the EasyCrypt proof assistant [Barthe et al. 2013]. EasyCrypt is a natural choice to implement eHL, since it is specially tailored to reason about probabilistic programs. Informally, EasyCrypt combines a proof engine for an ambient higher-order logic (HOL) with several program logics for proving properties of probabilistic programs. Judgments of the program logics are terms of the ambient logic, and proofs in the program logics are carried by means of (proof) *tactics*. In essence, a tactic implements a rule of the logic, by turning the conclusion into its hypotheses. This way, proofs are build gradually from the conclusion, upwards, ending in the axioms of the logic.

In order to support expectation-based reasoning, we have added eHL judgments as assertions of the ambient logic, and built support to reason about such judgments. In particular, we have:

- added tactics for core and several derived eHL rules. The core rules are in the trusted computing base (TCB) of the tool. However, the derived rules are designed to generate sequences of core tactics, in order to minimize the TCB as much as possible;
- added a library to reason about expectations. The library is required to discharge the many ambient logic goals generated by applying eHL tactics.

Derived proof rules. The proof rules in Section 5 follow the conventional presentation of program logics but are tedious to use in practice. For instance, reasoning about a sequence of instructions would first require a sequence of applications of rule (SEQ) to split the sequence apart, and then use the syntax-directed rules, possibly combined with non-structural rules, on the individual program instructions. Even more tedious, working towards a triple this way would entail that in many situations the intermediate pre-/post-expectations would need to be supplied by the user, as these cannot be inferred in general. To overcome these complications and to enhance usability of the logic, in the implementation we composing core syntax-directed rules with sequential composition and structural rules. An excerpt of derived rules can be found in Figure 8.

Rules (SKIPEC) is a variation of the ordinary rule (SKIP), combined with rule (CONSEQ) to make it applicable to the usual scenario where pre- and post-expectations differ. Rule (ASSIGNEC), the combination ofrules (SEQ) and (ASSIGN), embodies the backward style kind of analysis commonly found across the implementations of different logics in EasyCrypt, close to traditional weakest pre-condition reasoning. Generalising on this idea with rule (WPEC), our implementation provides a tactic wp computing the weakest pre-expectation, $\mathtt{wp}(\mathtt{D}, f)$, for a tail D neither containing loops nor procedure calls. To illustrate the advantage of these derived rules, note that the proof of rpartition_abs in Figure 1b is completely automated by the tactic wp, apart from the initial weakening step. This would not have been possible otherwise.

Among the more interesting derived rules is the final rule (FRAMEEC). In classical Hoare logic the *frame* rule, also known as rule of *constancy* constancy, takes the form

$$\frac{\vdash_{\mathsf{HL}} \{\, P \,\}\ \mathtt{C}\ \{\, Q \,\} \quad R \perp \mathsf{Mod}_{\mathtt{C}}}{\vdash_{\mathsf{HL}} \{\, P \wedge R \,\}\ \mathtt{C}\ \{\, Q \wedge R \,\}}$$

It is indispensable in practice, since it allows one to focus only on the relevant parts of an assertion, namely only the one that is potentially altered by C. But how to transfer this rule to our quantitative logic, in particular, how to translate logical conjunction? Here are three valid rules, all derivable from rules (CONSEQ) and (NMOD):

$$\frac{\vdash_Z \{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \quad P \perp \mathsf{Mod}_{\mathtt{C}}}{\vdash_Z \{\, P \mid f \,\}\ \mathtt{C}\ \{\, P \mid g \,\}}\ [\textsc{frame1}] \qquad \frac{\vdash_Z \{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \quad h \perp \mathsf{Mod}_{\mathtt{C}}}{\vdash_Z \{\, f \cdot h \,\}\ \mathtt{C}\ \{\, g \cdot h \,\}}\ [\textsc{frame2}] \qquad \frac{\vdash_Z \{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \quad h \perp \mathsf{Mod}_{\mathtt{C}}}{\vdash_Z \{\, f + h \,\}\ \mathtt{C}\ \{\, g + h \,\}}\ [\textsc{frame3}]$$

Rather than imposing a concrete choice, our rule (FRAMEEC) abstracts over the choice, and permits placing pre- and post-expectations in an arbitrary context $F \perp \mathsf{Mod}_{\mathtt{C}}$, that is concave[11] and non-decreasing (i.e. monotone), when seen as function $F : \mathbb{R}^{+\infty} \to \mathbb{R}^{+\infty}$. For instance, this rule has been applied in the previous section, adapting the function specification of rsample to the call site within o (see Figure 7). In the application of the rule, F is given by the context

$$\phi \mid \mathtt{if}\ \mathtt{Q} < \mathtt{c}\ \mathtt{then}\ \mathsf{bad}\ \mathtt{else}\ \epsilon \cdot (\square + \tfrac{\mathtt{Q}-\mathtt{c}}{\delta}). \tag{$\times$}$$

Seen as function in the hole $\square$, this term can be proven monotone and concave, as demanded by the third premise in rule (FRAMEEC). Since it mentions only local, unmodified variables, the second premise is easy to discharge. The rule itself is derivable by a composition of (NMOD) and

[11] i.e. $\forall t,\ 0 \le t \le 1 \Rightarrow \forall x\ y,\ tF(x) + (1-t)F(y) \le F(tx + (1-t)y)$

(CONSEQ). To see this, assume $\vdash_Z \{ f \}\ \mathtt{C}\ \{ g \}$. Rule (CONSEQ) deduces $\vdash_Z \{ F[f] \}\ \mathtt{C}\ \{ F[g] \}$ since $\mathbb{E}_d[F[g]] \leq F(\mathbb{E}_d[g]) \leq F[f]$ holds for all m, d with $\mathbb{E}_d[g] \leq f$. The first inequality effectively imposes concavity of F (it is then a consequence from the reverse Jensen's inequality), the second imposes that F is non-decreasing. Allowing F to depend on part of the memory that is not modified by C explains why the rule relies on (NMOD) to be justified.

To ease the application of the frame rule, we have proven a list of lemmas showing that functions like identity, multiplication by a constant or log satisfy those properties. EasyCrypt is then able to automatically/recursively apply those lemmas to prove the last premises. Furthermore this list of lemmas is user extensible. This way, for instance, EasyCrypt can automatically discharge the premises related to the context ($\times$) in the proof mentioned above.

The final rule, rule (CALLEC) implemented by tactic `call`, allows to compute the weakest pre-expectation of a procedure call, given a specification. The specification itself is usually already proven by a lemma in EasyCrypt. It implicitly features an application of rule (FRAMEEC), more precisely its instance (FRAME1) given in the motivation above, to internalise an implicit weakening of the post-expectations within the pre-expectation. This aides usability in connection with `wp`, which will for instance automatically propagate variable initialisation within the internalised weakening. The tactic `call` also takes a further context F (adhering to the restrictions imposed by rule (FRAMEEC)) as optional argument, in order to lift a procedure specification directly to its use at a call site.

Last but not least, in addition to these derived rules, we have extended already existing tactics that do not change the semantics of programs to deal with eHL judgments, such as the `inline` tactic that replaces a procedure call by its body.

Libraries of extended positive reals and expectations. We have developed a library to reason about extended positive reals and expectations. The library formalizes the type of positive reals $\mathbb{R}^+$ as a subtype of $\mathbb{R}$ and the type of extended positive reals as a disjoint union of $\mathbb{R}^+$ and $+\infty$. The library establishes that both positive and extended positive reals form additive monoids, which allows to instantiate the EasyCrypt library on big-operators. This library, inspired from [Bertot et al. 2008], provides a wealth of facts to reason about indexed sums—via the mathematical operator Σ. Using big-operators, it is thus relatively simple to define the notion of expectation, and to prove elementary facts about expectations. These facts are used to discharge many proof obligations automatically. At the time of writing, the library weights in at around 1.100 lines of proof scripts.

9 CONCLUSION

We have proposed a proof hopping approach for reasoning about expectation-based properties of (adversarial) probabilistic programs, and extended the EasyCrypt proof assistant to support our approach. In addition, we have shown that our approach is useful for reasoning about expected cost of randomized algorithms and for cryptographic proofs. Our implementation of eHL has been integrated into the EasyCrypt proof assistant. Future directions include extending eHL to quantum adversaries and quantum programs, and to further develop and capture formally the use of expectation-based properties in cryptography.

ACKNOWLEDGMENTS

We would like the thank the anonymous reviewers for their work and invaluable suggestions, which greatly improved our presentation. This work is partly supported by the ANR Project PPS: "Probabilistic Program Semantics" and the Agence Nationale de la Recherche (ANR, French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006. Further it is partly

supported by the FWF Project AUTOSARD: "Automated Sublinear Amortised Resource Analysis of Data Structures".

REFERENCES

S. Agrawal, K. Chatterjee, and P. Novotný. 2018. Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs. *PACMPL* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3385412.3386002

Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On continuation-passing transformations and expected cost analysis. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473592

M. Avanzini, U. Dal Lago, and A. Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Proc. of 34th LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785725

M. Avanzini, U. Dal Lago, and A. Yamada. 2020a. On Probabilistic Term Rewriting. *SCP* 185 (2020), 102338. https://doi.org/10.1016/j.scico.2019.102338

Martin Avanzini, Georg Moser, and Michael Schaper. 2020b. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

Martin Avanzini, Georg Moser, and Michael Schaper. 2023. Automated Expected Value Analysis of Recursive Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1050–1072. https://doi.org/10.1145/3591263

Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. 2023. Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 14085)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, 358–389. https://doi.org/10.1007/978-3-031-38554-4_12

Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2541–2563. https://doi.org/10.1145/3460120.3484548

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 43–61. https://doi.org/10.1007/978-3-319-41528-4_3

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 387–401. https://doi.org/10.1007/978-3-662-48899-7_27

G. Barthe, B. Grégoire, and S. Z. Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Proc. of 36th POPL*. ACM, 90–101. https://doi.org/10.1145/1480881.1480894

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7342)*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer, 1–6. https://doi.org/10.1007/978-3-642-31113-0_1

Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proc. of 44th POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 161–174. https://doi.org/10.1145/3009837.3009896

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986. https://doi.org/10.1145/3571260

Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. 2008. Canonical Big Operators. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 86–101. https://doi.org/10.1007/978-3-540-71067-7_11

O. Bournez and F. Garnier. 2005. Proving Positive Almost-Sure Termination. In *Proc. of 16th RTA (LNCS, Vol. 3467)*. Springer, 323–337. https://doi.org/10.1142/S0129054112400588

A. Chakarov and S. Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Proc. of 25th CAV (LNCS, Vol. 8044)*. Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

K. Chatterjee, H. Fu, and A. Murhekar. 2017. Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds. In *Proc. of 29th CAV (LNCS, Vol. 10426)*. Springer, 118–139. https://doi.org/10.1007/978-3-319-63387-9_6

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, , and Damien Stehlé. 2017. CRYSTALS–Dilithium: Algorithm Specification and Supporting Documentation. Round-1 submission to the NIST Post-Quantum Cryptography Standardization Project. https://cryptojedi.org/papers/#dilithium.

Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2020. Verified Analysis of Random Binary Tree Structures. *J. Autom. Reason.* 64, 5 (2020), 879–910. https://doi.org/10.1007/s10817-020-09545-0

Maximilian Paul Louis Haslbeck. 2021. *Verified Quantitative Analysis of Imperative Algorithms*. Ph. D. Dissertation. Technische Universität München.

Max W. Haslbeck and Manuel Eberl. 2020. Skip Lists. *Arch. Formal Proofs* 2020 (2020). https://www.isa-afp.org/entries/Skip_Lists.html

C. A. R. Hoare. 1961. Algorithm 65: find. *Commun. ACM* 4, 7 (1961), 321–322. https://doi.org/10.1145/366622.366647

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Joe Hurd, Annabelle McIver, and Carroll Morgan. 2004. Probabilistic Guarded Commands Mechanized in *HOL*. In *Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages, QAPL 2004, Barcelona, Spain, March 27-28, 2004 (Electronic Notes in Theoretical Computer Science, Vol. 112)*, Antonio Cerone and Alessandra Di Pierro (Eds.). Elsevier, 95–111. https://doi.org/10.1016/j.entcs.2004.01.021

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proc. of 42nd POPL*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *JACM* 65, 5 (2018), 30:1–30:68. https://doi.org/10.1145/3208102

Thomas Kleymann. 1998. *Hoare logic and VDM : machine-checked soundness and completeness proofs*. Ph. D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/387

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Formal Aspects Comput.* 11, 5 (1999), 541–566. https://doi.org/10.1007/s001650050057

Donald Knuth. 1973. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley.

D. Kozen. 1985. A Probabilistic PDL. *JCSC* 30, 2 (1985), 162 – 178. https://doi.org/10.1016/0022-0000(85)90012-1

Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7406)*, Lennart Beringer and Amy P. Felty (Eds.). Springer, 166–182. https://doi.org/10.1007/978-3-642-32347-8_12

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Proc. of 34th CAV (LNCS, Vol. 13372)*. 70–91. https://doi.org/10.1007/978-3-031-13188-2_4

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 211–222. https://doi.org/10.1145/1706299.1706326

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media.

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (1996), 325–353. https://doi.org/10.1145/229542.229547

N. C. Ngo, Q. Carbonneaux, and J. Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. of 39th PLDI*. ACM, 496–512. https://doi.org/10.1145/3296979.3192394

Tobias Nipkow. 2002a. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2471)*, Julian C. Bradfield (Ed.). Springer, 103–119. https://doi.org/10.1007/3-540-45793-3_8

Tobias Nipkow. 2002b. *Hoare Logics in Isabelle/HOL*. Springer Netherlands, Dordrecht, 341–367. https://doi.org/10.1007/978-94-010-0413-8_11

Tobias Nipkow, Manuel Eberl, and Maximilian P. L. Haslbeck. 2020. Verified Textbook Algorithms - A Biased Survey. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October*

19-23, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12302), Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 25–53. https://doi.org/10.1007/978-3-030-59152-6_2

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 4:1–4:50. https://doi.org/10.1145/3156018

F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proc. of 31nd LICS*. ACM, 672–681. https://doi.org/10.1145/2933575.2935317

William Pugh. 1990a. *Concurrent Maintenance of Skip Lists*. Technical Report. USA. http://hdl.handle.net/1903/542

William Pugh. 1990b. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977

Eric Schlechter. 1996. *Handbook of Analysis and Its Foundations*. Academic Press.

T. Takisaka, Y. Oyabu, N. Urabe, and I. Hasuo. 2018. Ranking and Repulsing Supermartingales for Reachability in Probabilistic Programs. In *Proc. of 16th ATVA (LNCS, Vol. 11138)*. Springer, 476–493. https://doi.org/10.1007/978-3-030-01090-4_28

Joseph Tassarotti and Robert Harper. 2018. Verified Tail Bounds for Randomized Programs. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 560–578. https://doi.org/10.1007/978-3-319-94821-8_33

Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 64:1–64:30. https://doi.org/10.1145/3290377

Eelis Van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5497)*, Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-642-02444-3_16

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *PACM on Programming Languages* 4, ICFP (2020), 110:1–110:31. https://doi.org/10.1145/3408992

P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi. 2019. Cost Analysis of Nondeterministic Probabilistic Programs. In *Proc. of 40th PLDI*. ACM, 204–220. https://doi.org/10.1145/3314221.3314581

$\mathtt{C} \in \mathsf{Stmt}$	$\mathbb{E}_{[\![\mathtt{C}]\!]_m}[f]$
$\mathtt{skip}$	$f\, m$
$\mathtt{x} \leftarrow \mathtt{E}$	$f\, m[\mathtt{x}/[\![\mathtt{E}]\!]_m]$
$\mathtt{x} \xleftarrow{\$} \mathtt{S}$	$\mathbb{E}_{[\![\mathtt{S}]\!]_m}[\lambda v. f\, m[\mathtt{x}/v]]$
$\mathtt{x} \xleftarrow{\$} \mathtt{f}(\mathtt{E})$	$\mathbb{E}_{[\![\mathtt{f}]\!]_{m_g}\,[\![\mathtt{E}]\!]_m}[\lambda(m'_g, r).\, f\,(m'_g \uplus m_l)[\mathtt{x}/r]]$
$\mathtt{C}_1\,;\,\mathtt{C}_2$	$\mathbb{E}_{[\![\mathtt{C}_1]\!]_m}[\lambda m'.\mathbb{E}_{[\![\mathtt{C}_2]\!]_{m'}}[f]]$
$\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2$	$\begin{cases}\mathbb{E}_{[\![\mathtt{C}_1]\!]_m}[f] & \text{if } [\![\mathtt{B}]\!]_m,\\ \mathbb{E}_{[\![\mathtt{C}_2]\!]_m}[f] & \text{otherwise.}\end{cases}$
$\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{D}$	$\sup_{i\in\mathbb{N}}(F^{(i)}\, m)$ *where*
	$F_f^{(\cdot)} : \mathbb{N} \to \mathsf{Mem} \to \mathbb{R}^{+\infty}$
	$F_f^{(0)}\ m \triangleq 0$
	$F_f^{(i+1)} m \triangleq \begin{cases} f\, m & \text{if } [\![\neg\mathtt{B}]\!]_m,\\ \mathbb{E}_{[\![\mathtt{D}]\!]_m}[F_f^{(i)}] & \text{otherwise.}\end{cases}$

Fig. 9. Structural expectation rules.

A SOUNDNESS AND COMPLETENESS PROOF OF EXPECTATION HOARE LOGIC

In this section, we proof soundness and completeness of the expectation logic.

Throughout the following, we fix a program $\mathtt{P}$. Let $\triangleright_{\mathtt{P}}$ denote the smallest relation on $\mathsf{Fun} \cup \mathsf{Stmt}$ such that (i) $\mathtt{C} \triangleright_{\mathtt{P}} \mathtt{D}$ if $\mathtt{D}$ is a direct sup-program of $\mathtt{C}$; and (ii) $\mathtt{f} \triangleright_{\mathtt{P}} \mathtt{C}$ for $\mathtt{C}$ the body of $\mathtt{f}$; and (iii) $\mathtt{x} \leftarrow \mathtt{f}(\mathtt{E}) \triangleright_{\mathtt{P}} \mathtt{f}$. Since procedures in $\mathtt{P}$ are assumed non-recursive, this relation is well-founded. The relation justifies well-definedness of our program semantics, since the semantics follows along $\triangleright_{\mathtt{P}}$. It also yields an induction principle on $\mathsf{Fun} \cup \mathsf{Stmt}$; that we dub *definitional induction* for brevity: The base cases are given by the atomic statements, except procedure calls. The inductive cases extend structural induction (i) on commands by the cases (ii) and (iii), where the property has to be shown to hold for procedures and procedure calls, assuming that it holds for the considered procedure's body and procedure, respectively.

We start with some observations of the expectation of f wrt. to program semantics.

Proposition A.1. *The following properties hold:*

(1) continuity: $\mathbb{E}_{\sup_i d_i}[f] = \sup_i(\mathbb{E}_{d_i}[f])$ *for every* ω*-chain* $d_0 \le d_1 \le d_2 \le \ldots$;
(2) monotony: $f \le g \Rightarrow \mathbb{E}_d[f] \le \mathbb{E}_d[g]$;
(3) bind: $\mathbb{E}_{\mathbf{dbind}\, d\, h}[f] = \mathbb{E}_d[\lambda a.\mathbb{E}_{h\, a}[f]]$;
(4) unit: $\mathbb{E}_{\mathbf{dunit}\, a}[f] = f\, a$.

The first point is a discrete version of Lebesgue's Monotone Convergence Theorem [Schlechter 1996, Theorem 21.38], implying monotony. The bind equation can be proven by unfolding and re-arranging sums. The unit law follows by definition. These laws are sufficient to prove the characterisation of $\mathbb{E}_{[\![\mathtt{C}]\!]_m}[f]$ given in Figure 9, which we tacitly employ throughout the proofs of soundness and completeness.

Proposition A.2. *All equalities depicted in Figure 9 hold.*

Formally this proposition can be proven by definitional induction. The only equality worth mentioning is that given wrt. to while loops, all remaining ones follow directly from Proposition A.1.

The crucial step in the proof of this equality lies in proving $F_f^{(i)}\ m = \mathbb{E}_{[\![\texttt{while}^{(i)}\ \texttt{B}\ \texttt{do}\ \texttt{C}]\!]_m}[f]$, which can be easily verified by induction on i. From there, the equality follows essentially by continuity of expectations (Proposition A.1.(1)).

The Call Rule. The rule (CALL), as we have introduced it in Section 5, requires the variable to which the resulting value of the procedure is assigned to be local. We prove soundness and complete/ness of our logic eHL in the more general case in which the assignment can be done also to a global variable. In this case, the rule (CALL) becomes the following:

$$\frac{\vdash_Z \{\, p \,\}\ \texttt{f}\ \{\, \lambda z\,(m_g, r).\ q\,z\,(m_g[\texttt{x}/r], r) \,\}}{\vdash_Z \{\, \lambda z\, m.\ p\,z\,(m_g, [\![\texttt{E}]\!]_m) \,\}\ \texttt{x} \xleftarrow{\$} \texttt{f}(\texttt{E})\ \{\, \lambda z\, m.\ q\,z\,(m_g, m\,\texttt{x}) \,\}}\ [\textsc{callG}]$$

where $\lambda z\,(m_g, r).\ q\,z\,(m_g[x/r], r)$ just reduces to q when x is local, resulting in our rule (CALL). This implies that under our convention that calls assign only to local variables, the rule (CALL) retains completeness.

A.1 Soundness

THEOREM A.3 (SOUNDNESS). *For every* $\texttt{f} \in \mathsf{Fun}$ *and* $p, q : Z \to \mathsf{Mem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}$,

$$\vdash_Z \{\, p \,\}\ \texttt{f}\ \{\, q \,\} \qquad \Rightarrow \qquad \vDash_Z \{\, p \,\}\ \texttt{f}\ \{\, q \,\}$$

PROOF. We prove simultaneously

(1) $\vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, g \,\}$ implies $\mathbb{E}_{[\![\texttt{C}]\!]_m}[g\,z] \le f\,z\,m$ for all $z \in Z$ and $m \in \mathsf{Mem}$; and
(2) $\vdash_Z \{\, p \,\}\ \texttt{f}\ \{\, q \,\}$ implies $\mathbb{E}_{[\![\texttt{f}]\!]_{m_g,v}}[q\,z] \le p\,z\,(m_g, v)$ for all $z \in Z$, $m_g \in \mathsf{GMem}$ and $v \in \mathsf{Val}$.

The proof is by induction on the derivation of $\vdash_Z \{\, p \,\}\ \texttt{f}\ \{\, q \,\}$.

- CASE (SKIP), (ASSIGN), (SAMPLE). Soundness of all three rules follows by definition.
- CASE (SEQ). Suppose $\vdash_Z \{\, f \,\}\ \texttt{C}; \texttt{D}\ \{\, g \,\}$ by Rule (SEQ) since (i) $\vdash_Z \{\, f \,\}\ \texttt{C}\ \{\, h \,\}$ and (ii) $\vdash_Z \{\, h \,\}\ \texttt{D}\ \{\, g \,\}$. Then, for any $m \in \mathsf{Mem}$ and $z \in Z$,

  $$\mathbb{E}_{[\![\texttt{C};\ \texttt{D}]\!]_m}[q\,z] = \mathbb{E}_{[\![\texttt{C}]\!]_m}[\lambda m'.\mathbb{E}_{[\![\texttt{D}]\!]_{m'}}[g\,z]] \le \mathbb{E}_{[\![\texttt{C}]\!]_m}[h\,z] \le f\,z$$

  where the first inequality follows by induction hopothesis on (ii) and monotony (Proposition A.1.2) and the second by induction hypothesis on (i).
- CASE (IF). Suppose $\vdash_Z \{\, f \,\}\ \texttt{if B then C else D}\ \{\, g \,\}$ since $\vdash_Z \{\, \texttt{B} \mid f \,\}\ \texttt{C}\ \{\, g \,\}$ and $\vdash_Z \{\, \neg\texttt{B} \mid f \,\}\ \texttt{D}\ \{\, g \,\}$. Fix a memory $m \in \mathsf{Mem}$ and $z \in Z$. Suppose first that m satisfies B. In this case

  $$\mathbb{E}_{[\![\texttt{if B then C else D}]\!]_m}[q\,z] = \mathbb{E}_{[\![\texttt{C}]\!]_m}[q\,z] \le \texttt{B}\,m \mid f\,z\,m = f\,z\,m$$

  where the equalities follow by assumption on m, and the inequality is given by the induction hypothesis. The case where m does not satisfy B follows by identical reasoning.
- CASE (WHILE). Suppose $\vdash_Z \{\, f \,\}\ \texttt{while B do C}\ \{\, \neg\texttt{B} \mid f \,\}$ since $\vdash_Z \{\, \texttt{B} \mid f \,\}\ \texttt{C}\ \{\, f \,\}$.
  To prove the case, it is sufficient to show

  $$F_g^{(i)}\ m \le f\,z\,m$$

  for all $i \in \mathbb{N}$, $m \in \mathsf{Mem}$ and $z \in Z$, with F as defined in Figure 9 and $g \triangleq \lambda m'.\ [\![\neg\texttt{B}]\!]\,m' \mid f\,z\,m'$. The proof is by (side) induction on i. The base case is trivial. In the inductive case, we separate two cases. First, suppose m satisfies B. Then

  $$F_g^{(i+1)}\ m = \mathbb{E}_{[\![\texttt{C}]\!]_m}[F_g^{(i)}] \le \mathbb{E}_{[\![\texttt{C}]\!]_m}[(f\,z)] \le [\![\texttt{B}]\!]_m \mid f\,z\,m = f\,z\,m$$

  where the first inequality follows by monotony of expectations (Proposition A.1.2) and side induction hypothesis, and the second by the outer induction hypothesis In the remaining case,

where m does not satisfy B, we directly have

$$F_g^{(i+1)}\, m = [\![\neg \mathsf{B}]\!]_m \mid f\, z\, m = f\, z\, m$$

- Case (call). Suppose $\vdash_Z \{\, \lambda z\, m.\ p\, z\, (m_g, [\![\mathsf{E}]\!]_m) \,\}\ \mathsf{x} \xleftarrow{\$} \mathtt{f}(\mathsf{E})\ \{\, \lambda z\, m.\ q\, z\, (m_g, m\,\mathsf{x}) \,\}$ by Rule (call). Soundness follows as for any $z \in Z$ and $m \in \mathsf{Mem}$,

$$\mathbb{E}_{[\![\mathsf{x} \xleftarrow{\$} \mathtt{f}(\mathsf{E})]\!]_m}[\lambda m'.\ q\, z\, (m'_g, m'\,\mathsf{x})] = \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, [\![\mathsf{E}]\!]_m}}[\lambda (m', r).\ q\, z\, (m'[\mathsf{x}/r], r)] \le p\, z\, (m_g,\ [\![\mathsf{E}]\!] m)$$

by induction hypothesis.

- Case (proc). For the inverse, suppose $\vdash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$ for $(\mathtt{proc\ f(x)\ C;\ return\ E}) \in \mathsf{P}$. Soundness follows as for any $z \in Z$, $m_g \in \mathsf{Mem}$ and arguments $v \in \mathsf{Val}$,

$$\begin{aligned}
\mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, v}}[q\, z] &= \mathbb{E}_{[\![\mathsf{C}]\!]_{m_g \uplus m_l^0[\mathsf{x}/v]}}[\lambda m'.\ q\, z\, (m'_g, [\![\mathsf{E}]\!]_{m'})] \\
&\quad \textit{(induction hypothesis)} \\
&\le (\lambda m.\ m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}] \mid p\, z\, (m_g, m\,\mathsf{x}))\ (m_g \uplus m_l^0[\mathsf{x}/v]) \\
&= p\, z\, (m_g,\ v)
\end{aligned}$$

- Case (conseq). Suppose $\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$ was derived from $\vdash_{Z'} \{\, f' \,\}\ \mathsf{C}\ \{\, g' \,\}$ by Rule (conseq). Fix a memory $m \in \mathsf{Mem}$ and $z \in Z$. Using the induction $\forall z' \in Z'.\ \mathbb{E}_{[\![\mathsf{C}]\!]_m}[g'\, z'] \le f'\, z'\, m$ to discharge the assumption of the side-condition in Rule (conseq) yields $\forall z \in Z.\ \mathbb{E}_{[\![\mathsf{C}]\!]_m}[g\, z] \le f\, z\, m$, which is precisely what we have to show in this case.
- Case (nmod). Fix $m \in \mathsf{Mem}$ and $z \in Z$, and let $v \triangleq m\,\mathsf{x}$. Then

$$\mathbb{E}_{[\![\mathsf{C}]\!]_m}[g] = \mathbb{E}_{[\![\mathsf{C}]\!]_m}[\lambda m'.\ g\, z\, m'[\mathsf{x}/v]] \le m\,\mathsf{x} = v \mid f\, z\, m = f\, z\, m$$

where the first equality holds as $m' \in \mathrm{supp}([\![\mathsf{C}]\!]_m)$ implies $m'\, x = v$ due to the side-condition $\mathsf{x} \notin \mathsf{Mod}_{\mathsf{C}}$, and the inequality follows by induction hypothesis.

- Case (prhl).

$$\frac{\begin{array}{ll} \vdash_Z \{\, f' \,\}\ \mathsf{C}'\ \{\, g' \,\} & \forall z\, m.\ f\, z\, m \neq \infty \Rightarrow \exists m'.\ f'\, z\, m' \le f\, z\, m \wedge P\, m'\, m \quad (3) \\ \vdash \{\, P \,\}\ \mathsf{C}' \sim \mathsf{C}\ \{\, Q \,\} & \forall z\, m'\, m.\ Q\, m'\, m \Rightarrow g\, z\, m \le g'\, z\, m' \quad (4) \end{array}}{\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}}\ [\textsc{prhl}]$$

We have the following hypothesis:

(1) By i.h. $\vdash_Z \{\, f' \,\}\ \mathsf{C}\ \{\, g' \,\}$, i.e. for each $z \in Z, m' \in \mathsf{Mem}$, $\mathbb{E}_{[\![\mathsf{C}']\!]_{m'}}[(g'z)] \le f'zm'$.

(2) By soundness of pRHL, $\vDash \{\, P \,\}\ \mathsf{C}' \sim \mathsf{C}\ \{\, Q \,\}$, i.e. for each $m_1, m_2 \in \mathsf{Mem}$, $Pm_1m_2 \Rightarrow Q^\dagger\ [\![\mathsf{C}']\!]_{m_1} [\![\mathsf{C}]\!]_{m_2}$.

If $f\, z\, m = \infty$, we are done. Then we consider $f\, z\, m \neq \infty$. By (1) and (3) we have:

$$\forall z\, m. \exists m'.\ \mathbb{E}_{[\![\mathsf{C}']\!]_{m'}}[g'z] \le f\, z\, m \wedge P\, m'\, m$$

Then, by (2):

$$\forall z\, m. \exists m'.\ \mathbb{E}_{[\![\mathsf{C}']\!]_{m'}}[g'z] \le f\, z\, m \wedge\ Q^\dagger\ [\![\mathsf{C}']\!]_{m'} [\![\mathsf{C}]\!]_m \qquad (+)$$

By definition of $Q^\dagger$, there exists a distribution d on $\mathsf{Mem} \times \mathsf{Mem}$ such that (i) if $d(m_1, m_2) > 0$, then Qm_1m_2 holds, and (ii) which has $[\![\mathsf{C}']\!]_{m'}$ and $[\![\mathsf{C}]\!]_m$ as marginals. Then, using (ii), for each z we have:

$$\begin{aligned}
\mathbb{E}_{[\![\mathsf{C}]\!]_m}[gz] - \mathbb{E}_{[\![\mathsf{C}']\!]_{m'}}[g'z] &= \mathbb{E}_d[\lambda m_1 m_2.g\, z\, m_1] - \mathbb{E}_d[\lambda m_1 m_2.g'\, z\, m_2] \\
&= \mathbb{E}_d[\lambda m_1 m_2.g\, z\, m_1 - g'\, z\, m_2]
\end{aligned}$$

Then, by (i) and (4), we have:

$$\mathbb{E}_d[\lambda m_1 m_2.g\, z\, m_1 - g'\, z\, m_2] \le 0$$

and thus, reading back the chain of equalities:

$$\mathbb{E}_{[\![\mathsf{C}]\!]_m}[g\,z] \leq \mathbb{E}_{[\![\mathsf{C}']\!]_{m'}}[g'\,z]$$

Finally, combining with (+)

$$\forall z\,m.\ \mathbb{E}_{[\![\mathsf{C}]\!]_m}[g\,z] \leq f\,z\,m$$

□

Adversary rule. We now prove that the adversary rule is correct. To this end, let $\mathsf{Write}_{\mathsf{C}} \subseteq \mathsf{GMem}$ and $\mathsf{Write}_{\mathsf{f}} \subseteq \mathsf{GMem}$ denote the *writeable variables* of statement C and procedure $\mathtt{f}$. In particular, $\mathsf{Write}_{\mathsf{f}} = \mathsf{Write}_{\mathsf{C}}$ for C the body of $\mathtt{f}$. On statements C, $\mathsf{Write}_{\mathsf{C}}$ is defined so that $\mathsf{x} \in \mathsf{Write}_{\mathsf{C}}$ if C contains any instruction $\mathsf{x} \leftarrow e$ overwriting the value of x. If e is a call $\mathtt{f}(\mathsf{E})$ then additionally $\mathsf{Write}_{\mathtt{f}} \subseteq \mathsf{Write}_{\mathsf{C}}$. Finally, for a statement C_o with abstract oracle call $\mathsf{x} \leftarrow o(\mathsf{E})$, we also require $\mathsf{x} \in \mathsf{Write}_{\mathsf{C}_o}$.

Let $f : Z \to \mathsf{GMem} \to \mathbb{R}^{+\infty}$ be the invariant. To avoid notational overhead, we may also write f for the pre-/post-expectations $\lambda m.\ f\,m_g$ and $\lambda(m_g, \_).\ f\,m_g$ within triples of statements and procedure calls, respectively. The following is the main technical lemma behind soundness of the adversarial rule.

Lemma A.4. *Let C_o be the body of an adversary and let $f : Z \to \mathsf{GMem} \to \mathbb{R}^{+\infty}$ with $f \perp \mathsf{Write}_{\mathsf{C}_o}$. Then*

$$\vdash_Z \{\, f \,\}\ \mathtt{o}\ \{\, f \,\} \qquad \Rightarrow \qquad \vdash_Z \{\, f \,\}\ \mathsf{C}_{\mathtt{o}}\ \{\, f \,\}$$

where $\mathsf{C}_{\mathtt{o}}$ is obtained from C_o by instantiating oracles o by $\mathtt{o}$.

Proof. Suppose $\vdash_Z \{\, f \,\}\ \mathtt{o}\ \{\, f \,\}$. The proof is by definitional induction on the program P, extending the relation $\triangleright_{\mathsf{P}}$ to adversary code as expected. Suppose $\vdash_Z \{\, f \,\}\ \mathtt{o}\ \{\, f \,\}$, we prove $\vdash_Z \{\, f \,\}\ \mathsf{C}\ \{\, f \,\}$ for all $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathsf{C}$ and $\vdash_Z \{\, f \,\}\ \mathtt{g}\ \{\, f \,\}$ for all $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathtt{g}$.

- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathsf{C}$ with C one of $\mathtt{skip}$, $\mathsf{x} \xleftarrow{\$} \mathsf{S}$; or $\mathsf{x} \xleftarrow{\$} \mathsf{E}$. The claim follows directly from applying rules (skip), (sample),or (assign)
- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathsf{x} \leftarrow \mathtt{g}(\mathsf{E})$ with $\mathtt{g} \neq \mathtt{f}$. We conclude by application of rule (call)

$$\frac{\vdash_Z \{\, \lambda z\,(m_g, \_).\ f\,z\,m_g \,\}\ \mathtt{g}\ \{\, \lambda z\,(m_g, \_).\ f\,z\,m_g \,\}}{\vdash_Z \{\, \lambda z\,m.\ f\,z\,m_g \,\}\ \mathsf{x} \leftarrow \mathtt{g}(\mathsf{E})\ \{\, \lambda z\,m.\ f\,z\,m_g \,\}}\ [\textsc{call}]$$

  with the premise given by induction hypothesis. Concerning the pre-expectation, notice that we employed, $\lambda z\,m.\ (\lambda z\,(m_g, \_).\ f\,z\,m_g)\,z\,(m_g, [\![\mathsf{E}]\!]_m) = \lambda z\,m.\ f\,z\,m_g$, similar for the post-expectation.
- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathsf{x} \leftarrow \mathtt{f}(\mathsf{E})$. The case is identical to above by rule (call), but the premise stems from the assumption.
- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathsf{C}\,;\,\mathsf{D}$. The case is given by induction hypotheses and rule (seq).
- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathtt{if}\ \mathsf{B}\ \mathtt{then}\ \mathsf{C}\ \mathtt{else}\ \mathsf{D}$. Then we conclude as follows

$$\frac{\dfrac{\{\, f \,\}\ \mathsf{C}\ \{\, f \,\} \quad (\dagger)}{\{\, \mathsf{B} \mid f \,\}\ \mathsf{C}\ \{\, f \,\}}\ [\textsc{conseq}] \qquad \dfrac{\{\, f \,\}\ \mathsf{C}\ \{\, f \,\} \quad (\ddagger)}{\{\, \neg\mathsf{B} \mid f \,\}\ \mathsf{D}\ \{\, f \,\}}\ [\textsc{conseq}]}{\{\, f \,\}\ \mathtt{if}\ \mathsf{B}\ \mathtt{then}\ \mathsf{C}\ \mathtt{else}\ \mathsf{D}\ \{\, f \,\}}\ [\textsc{if}]$$

  where to discharge the side-conditions (†) and (‡) we use $f \leq \mathsf{B} \mid f$ and $f \leq \neg\mathsf{B} \mid f$, respectively. The premises are given by induction hypothesis.
- Case $\mathsf{C}_o \triangleright_{\mathsf{P}}^* \mathtt{while}\ \mathsf{B}\ \mathtt{do}\ \mathsf{C}$. Then we conclude as follows

$$\frac{\dfrac{\dfrac{\{\, f \,\}\ \mathsf{C}\ \{\, f \,\} \quad (\dagger)}{\{\, \mathsf{B} \mid f \,\}\ \mathsf{C}\ \{\, f \,\}}\ [\textsc{conseq}]}{\{\, f \,\}\ \mathtt{while}\ \mathsf{B}\ \mathtt{do}\ \mathsf{C}\ \{\, \neg\mathsf{B} \mid f \,\}}\ [\textsc{while}] \qquad (\ddagger)}{\{\, f \,\}\ \mathtt{while}\ \mathsf{B}\ \mathtt{do}\ \mathsf{C}\ \{\, f \,\}}\ [\textsc{conseq}]$$

where we use again $f \le \mathsf{B} \mid f$ and $f \le \neg\mathsf{B} \mid f$ to discharge the side-conditions (†) and (‡), respectively, and where the premise is given by induction hypothesis.

- Case $\mathsf{C}_o \triangleright^*_{\mathsf{P}} g$.

$$\dfrac{\dfrac{\vdash_Z \{\, \lambda z\, m.\ f\, z\, m_g \,\}\ \mathsf{C}\ \{\, \lambda z\, m.\ f\, z\, m_g \,\} \quad (\dagger)}{\vdash_Z \{\, \lambda z\, m.\ m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}] \mid f\, z\, m_g \,\}\ \mathsf{C}\ \{\, \lambda z\, m.\ f\, z\, m_g \,\}}\,[\textsc{conseq}]}{\vdash_Z \{\, (\lambda z\, (m_g, \_).\ f\, z\, m_g) \,\}\ \mathtt{f}\ \{\, (\lambda z\, (m_g, \_).\ f\, z\, m_g) \,\}}\,[\textsc{proc}]$$

where $\mathtt{proc\ f(x)\ C; return\ E}$ and (†) derived from $\lambda z\, m.\ f\, z\, m_g \le \lambda z\, m.\ m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}] \mid f\, z\, m_g$.

□

Theorem A.5. *Rule (adv) is sound.*

Proof. The proof follows precisely the case of rule (proc) in the soundness proof Theorem A.3. Rather than the induction hypothesis, we make use of Lemma A.4. Note how the premises of the lemma correspond to that of the rule. □

A.2 Completeness

Theorem A.6 (Completeness). *For every* $\mathtt{f} \in \mathsf{Fun}$ *and* $p, q : Z \to \mathsf{Mem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}$,

$$\vDash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\} \qquad \Rightarrow \qquad \vdash_Z \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}$$

Proof. To prove this theorem, we prove the stronger claims:

(1) $\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathsf{C}]\!]_m}[g\, z] \,\}\ \mathsf{C}\ \{\, g \,\}$ for all $\mathsf{C} \in \mathsf{Stmt}$ and $g : Z \to \mathsf{Mem} \to \mathbb{R}^{+\infty}$; and
(2) $\vdash_Z \{\, \lambda z\, (m_g, v).\ \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, v}}[q\, z] \,\}\ \mathtt{f}\ \{\, q \,\}$ for all $\mathtt{f} \in \mathsf{Fun}$ and $q : Z \to \mathsf{GMem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}$.

The proof is by definitional induction on P. Again, all base cases follow by definition. Concerning the inductive step, we continue by case analysis:

- Case $\mathsf{x} \xleftarrow{\$} \mathtt{f(E)}$. Fix $g : Z \to \mathsf{Mem} \to \mathbb{R}^{+\infty}$. For a memory $m \in \mathsf{Mem}$ and variables $\vec{\mathsf{x}}$, let $m_{\vec{\mathsf{x}}}$ denote the projection of m to $\vec{\mathsf{x}}$. Let $\vec{l}$ and $\vec{g}$ denote a sequence of local and global variables, respectively, without x, thus in particular every $m \in \mathsf{Mem}$ can be written as $m_{\vec{g}} \uplus \{\mathsf{x} \mapsto v\} \uplus m_{\vec{l}}$, for some value v. Using this notation, unfolding definitions we see that

$$\mathbb{E}_{[\![\mathsf{x} \xleftarrow{\$} \mathtt{f(E)}]\!]_m}[g\, z] = \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, [\![\mathsf{E}]\!]_m}}[\lambda(m', r).\ g\, z\, (m'_{\vec{g}} \uplus \{\mathsf{x} \mapsto r\} \uplus m_{\vec{l}})]$$

By repeated application of Rule (nmod), the case thus follows if

$$\begin{aligned}\vdash_{Z \times \mathsf{Val}^k}\ & \{\, \lambda(z, \vec{v})\, m.\ m\, \vec{l} = \vec{v} \mid \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, [\![\mathsf{E}]\!]_m}}[\lambda(m', r).\ g\, z\, (m'_{\vec{g}} \uplus \{\mathsf{x} \mapsto r\} \uplus \{\vec{l} \mapsto \vec{v}\})] \,\} \\ & \mathsf{x} \xleftarrow{\$} \mathtt{f(E)} \\ & \{\, \lambda(z, \vec{v})\, m.\ g\, z\, (m_{\vec{g}} \uplus \{\mathsf{x} \mapsto m\,\mathsf{x}\} \uplus \{\vec{l} \mapsto \vec{v}\}) \,\}\end{aligned}$$

Let $q\, (z, \vec{v})\, (m, r) \triangleq g\, z\, (m_{\vec{g}} \uplus \{\mathsf{x} \mapsto r\} \uplus \{\vec{l} \mapsto \vec{v}\})$, thus, this judgment is thus equivalent to

$$\vdash_{Z \times \mathsf{Val}^k} \{\, \lambda(z, \vec{v})\, m.\ m\, \vec{l} = \vec{v} \mid \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, [\![\mathsf{E}]\!]_m}}[(q\, (z, \vec{v}))] \,\}\ \mathsf{x} \xleftarrow{\$} \mathtt{f(E)}\ \{\, \lambda(z, \vec{v})\, m.\ q\, (z, \vec{v})\, (m_g, m_g\, \mathsf{x}) \,\}$$

which by weakening the pre-expectation using Rule (conseq) follows from

$$\vdash_{Z \times \mathsf{Val}^k} \{\, \lambda z'\, m.\ \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, [\![\mathsf{E}]\!]_m}}[q\, z'] \,\}\ \mathsf{x} \xleftarrow{\$} \mathtt{f(E)}\ \{\, \lambda z'\, m.\ q\, z'\, (m_g, m_g\, \mathsf{x}) \,\}$$

Notice that $q\, z\, (m_g[\mathsf{x}/r], r) = q\, z\, (m_g, r)$, thus we can now apply Rule (call) so as to precisely yield the induction hypothesis:

$$\vdash_{Z \times \mathsf{Val}^k} \{\, \lambda z'\, (m_g, w).\ \mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, w}}[q\, z'] \,\}\ \mathtt{f}\ \{\, q\, z' \,\}$$

- Case $\mathtt{C}\,\mathtt{;}\,\mathtt{D}$. Induction hypothesis on $\mathtt{C}$ and $\mathtt{D}$, respectively, yield

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g\, z] \,\}\ \mathtt{C}\ \{\, g \,\} \qquad\qquad \vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{D}]\!]_m}[f\, z] \,\}\ \mathtt{D}\ \{\, f \,\}$$

  for any Z and pre-expectations f and g. Thus, by substituting $(\lambda z m.\mathbb{E}_{[\![\mathtt{D}]\!]_m}[f\, z])$ for g in the first judgments, an application of Rule (seq) yields,

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{C}]\!]_m}[\lambda m.\mathbb{E}_{[\![\mathtt{D}]\!]_m}[f\, z]] \,\}\ \mathtt{C}\,\mathtt{;}\ \mathtt{D}\ \{\, f \,\}$$

  As by Figure 9 the pre-expectation just corresponds to $\lambda z\, m.\ \mathbb{E}_{[\![\mathtt{C}\,\mathtt{;}\ \mathtt{D}]\!]_m}[f\, z]$, the case follows.
- Case $\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}$. Fix post-expectation f. Observe that for any $m \in \mathsf{Mem}$ and $z \in Z$,

$$\mathbb{E}_{[\![\mathtt{C}]\!]_m}[f\, z] \le [\![\mathtt{B}]\!]_m \mid \mathbb{E}_{[\![\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}]\!]_m}[f\, z]$$

  holds. This inequality can be validated by case analysis on $[\![\mathtt{B}]\!]_m$ and unfolding the definition of $[\![\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}]\!]_m$, see Figure 9. Induction hypothesis on $\mathtt{C}$ together with Rule (conseq), then proves

$$\vdash_Z \{\, \lambda z\, m.\ [\![\mathtt{B}]\!]_m \mid \mathbb{E}_{[\![\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}]\!]_m}[f\, z] \,\}\ \mathtt{C}\ \{\, f \,\}$$

  Similar, induction hypothesis on $\mathtt{D}$ and one application of Rule (conseq) proves

$$\vdash_Z \{\, \lambda z\, m.\ [\![\neg\mathtt{B}]\!]_m \mid \mathbb{E}_{[\![\mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}]\!]_m}[f\, z] \,\}\ \mathtt{D}\ \{\, f \,\}$$

  The claim now follows by one application of Rule (if).
- Case $\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}$. Fix post-expectation f. By instantiating the post-expectation of the induction hypothesis on $\mathtt{C}$ with $\lambda z\, m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z]$, we see that

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{C}]\!]_m}[\lambda m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z]] \,\}\ \mathtt{C}\ \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\},$$

  is derivable. Notice that

$$\mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] = \mathbb{E}_{[\![\mathtt{C}]\!]_m}[\lambda m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z]],$$

  for all memories $m \in \mathsf{Mem}$ on which $\mathtt{B}$ evaluates to true, compare Figure 9. This permits us to strengthen the pre-expectation via one application of Rule (conseq), deriving

$$\vdash_Z \{\, \lambda z\, m.\ [\![\mathtt{B}]\!]_m \mid \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\}\ \mathtt{C}\ \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\}.$$

  An application of Rule (while) thus yields

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\}\ \mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}\ \{\, \lambda z\, m.\ [\![\neg\mathtt{B}]\!]_m \mid \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\}.$$

  Exploiting that, by Figure 9, $\mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] = f\, z\, m$ for all memories $m \in \mathsf{Mem}$ on which $\mathtt{B}$ evaluates to false, another application of Rule (conseq) allows us to weaken the post-expectation, yielding

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}]\!]_m}[f\, z] \,\}\ \mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathtt{C}\ \{\, f\, z \,\},$$

  and thereby concluding the case.
- Case $\mathtt{proc}\ \mathtt{f}(\mathtt{x})\ \mathtt{C}\,\mathtt{;}\ \mathtt{return}\ \mathtt{E}$. Fix $q : Z \to \mathsf{GMem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}$. By induction hypothesis,

$$\vdash_Z \{\, \lambda z\, m.\ \mathbb{E}_{[\![\mathtt{C}]\!]_m}[\lambda m'.\ q\, z\, (m'_g, [\![\mathtt{E}]\!]_{m'})] \,\}\ \mathtt{C}\ \{\, \lambda z\, m.\ q\, z\, (m_g, [\![\mathtt{E}]\!]_m) \,\},$$

  which by Rule (conseq) gives

$$\vdash_Z \{\, \lambda z\, m.\ m_l = m_l^0[\mathtt{x}/m\,\mathtt{x}] \mid \mathbb{E}_{[\![\mathtt{C}]\!]_{m_g \uplus m_l^0[\mathtt{x}/m\,\mathtt{x}]}}[\lambda m'.\ q\, z\, (m'_g, [\![\mathtt{E}]\!]_{m'})] \,\}\ \mathtt{C}\ \{\, \lambda z\, m.\ q\, z\, (m_g, [\![\mathtt{E}]\!]_m) \,\},$$

  and thus

$$\vdash_Z \{\, \lambda z\, (m_g, v).\ \mathbb{E}_{[\![\mathtt{C}]\!]_{m_g \uplus m_l^0[\mathtt{x}/v]}}[\lambda m'.\ q\, z\, (m'_g, [\![\mathtt{E}]\!]_{m'})] \,\}\ \mathtt{f}\ \{\, q \,\},$$

  by Rule (proc). This establishes the case, since

$$\mathbb{E}_{[\![\mathtt{f}]\!]_{m_g, v}}[q\, z] = \mathbb{E}_{[\![\mathtt{C}]\!]_{m_g \uplus m_l^0[\mathtt{x}/v]}}[\lambda m'.\ q\, z\, (m'_g, [\![\mathtt{E}]\!]_{m'})],$$

$$\frac{(Z\colon \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}) \in \Gamma}{\Gamma \vdash Z\colon \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}}\ [\textsc{ass}]$$

$$\frac{(\mathtt{proc}\ \mathtt{f}(\mathsf{x})\ \mathsf{C};\ \mathtt{return}\ \mathsf{E}) \in \mathsf{P} \qquad \Gamma;\ Z\colon \{\, p \,\}\ \mathtt{f}\ \{\, q \,\} \vdash Z\colon \{\, \lambda z\, m.\ m_l = m_l^0[\mathsf{x}/m\,\mathsf{x}] \mid p\, z\, (m_g, m\,\mathsf{x}) \,\}\ \mathsf{C}\ \{\, \lambda z\, m.\ q\, z\, (m_g, [\![\mathsf{E}]\!]_m) \,\}}{\Gamma \vdash \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}}\ [\textsc{proc}]$$

$$\frac{\Gamma \vdash Z'\colon \{\, p' \,\}\ \mathtt{f}\ \{\, q' \,\} \qquad \forall m_g\, v\, d.\ (\forall z' \in Z'.\ \mathbb{E}_d[(q'\, z')] \le p'\, z'\, (m_g, v)) \Rightarrow (\forall z \in Z.\ \mathbb{E}_d[q\, z] \le p\, z\, (m_g, v))}{\Gamma \vdash Z\colon \{\, p \,\}\ \mathtt{f}\ \{\, q \,\}}\ [\textsc{conseqP}]$$

Fig. 10. Changes to eHOARE to account for recursion.

by Figure 9. □

B RECURSION

Semantics. To incorporate recursion, the semantics of statements $\mathsf{C} \in \mathsf{Stmt}$ now take the form $[\![\mathsf{C}]\!]^\eta_m$ where

$$\eta : \mathsf{Fun} \to \mathsf{GMem} \times \mathsf{Val} \to \mathsf{D}\,(\mathsf{GMem} \times \mathsf{Val})$$

is a *procedure* environment. The interpretation of statements remains as is, procedures though are interpreted via a lookup within the environment

$$[\![\mathtt{f}]\!]^\eta_{m_g}\, v \triangleq \eta\, \mathtt{f}\, (m_g, v)$$

A program P can now be interpreted as a procedure environment, defined through its approximations

$$[\![\mathsf{P}]\!] \triangleq \sup_{i \in \mathbb{N}} \eta^{(i)}$$

where

$$\eta^{(0)}\, \mathtt{f} \triangleq \lambda(m_g, v).\ \mathbf{fail}$$

$$\eta^{(i+1)}\, \mathtt{f} \triangleq \lambda(m_g, v).\mathbf{dlet}\ m' \leftarrow ([\![\mathsf{C}]\!]^{\eta^{(i)}}_{m_g \uplus m_l^0[\mathsf{x}/v]})\ \mathbf{in}\ \mathbf{dunit}\ (m'_g, [\![\mathsf{E}]\!]_{m'})$$

Note that $\eta^{(i)}$ corresponds to the semantics that permit i unfoldings of procedures, failing when the threshold i has been reached. When $\eta = [\![\mathsf{P}]\!]$ we abbreviate $[\![\mathsf{C}]\!]^\eta$ and $[\![\mathtt{f}]\!]^\eta$ by $[\![\mathsf{C}]\!]$ and $[\![\mathtt{f}]\!]$, respectively.

Logic. Judgements take now the form

$$\Gamma \vdash Z\colon \{\, p \,\}\ \mathtt{f}\ \{\, q \,\} \quad \text{and} \quad \Gamma \vdash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$$

where $\Gamma = Z_1\colon \{\, p_1 \,\}\ \mathtt{f}_1\ \{\, q_1 \,\};\ \ldots;\ Z_k\colon \{\, p_k \,\}\ \mathtt{f}_k\ \{\, q_k \,\}$ is a sequence of judgements on Fun. The rules of the logic remain mostly the same, with Γ remaining constant. What changes

is Rule (PROC) which now permits assuming the judgement that is to be derived. Further, we add an assumption rule, and a rule of consequence for procedures. The latter is strictly speaking not necessary (for completeness), but convenient and slightly simplifies the presentation of proofs.

With an eye on proofs, we parameterise validity of judgements by a procedure environment

$$\eta \vDash Z\colon \{\, p \,\}\ \mathsf{C}\ \{\, q \,\} \qquad :\Leftrightarrow \qquad \forall m\, z.\ \mathbb{E}_{[\![\mathsf{C}]\!]^{\eta}_{m}}[q\, z] \le f\, z\, m$$
$$\eta \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \qquad :\Leftrightarrow \qquad \forall m_g\, v\, z.\ \mathbb{E}_{[\![\mathsf{f}]\!]^{\eta}_{m_g}\, v}[(q\, z)] \le p\, z\, (m_g, v)$$

When $\eta = [\![\mathsf{P}]\!]$, we may omit η from the left-hand side; recovering the expected notion of validity. Note that entailment is antitone in the environment, in the following sense.

Lemma B.1.

$$\eta \le \zeta \wedge \zeta \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \Rightarrow \eta \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$$

Proof. If $\eta \le \zeta$ then $[\![f]\!]^{\eta}_{m_g}\, v$ is a sub-distribution of $[\![f]\!]^{\zeta}_{m_g}\, v$, for all $m_g \in \mathsf{GMem}$ and $v \in \mathsf{Val}$. Consequently, $\mathbb{E}_{[\![\mathsf{f}]\!]^{\eta}_{m_g}\, v}[q\, z] \le \mathbb{E}_{[\![\mathsf{f}]\!]^{\zeta}_{m_g}\, v}[q\, z] \le p\, z\, (m_g, v)$. □

We extend validity to contexts by stipulating $\vDash \Gamma$ iff $\vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$ for all $(Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}) \in \Gamma$ and finally define

$$\Gamma \vDash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\} \qquad :\Leftrightarrow \qquad \vDash \Gamma \Rightarrow\ \vDash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$$
$$\Gamma \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \qquad :\Leftrightarrow \qquad \vDash \Gamma \Rightarrow\ \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$$

Soundness. Again we define soundness as derivability implies validity, now however, for any context Γ. The central technical lemma states soundness wrt. to approximations.

Lemma B.2. *Let $\eta^{(i)}$ be the i-th approximant of $[\![\mathsf{P}]\!]$. For all $i \in \mathbb{N}$, $\eta^{(i)} \vDash \Gamma$, then*
(1) $\Gamma \vdash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \Rightarrow \eta^{(i)} \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$.
(2) $\Gamma \vdash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\} \Rightarrow \eta^{(i)} \vDash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$; *and*

Proof. We prove the two statements simultaneously. To this end, suppose $\Gamma \vdash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$ or $\Gamma \vdash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$, we prove $\eta^{(i)} \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$ and $\eta^{(i)} \vDash Z\colon \{\, f \,\}\ \mathsf{C}\ \{\, g \,\}$, respectively. The proof is by induction on the size of the deriviation plus i. □

Theorem B.3 (Soundness).

$$\Gamma \vdash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \qquad \Rightarrow \qquad \Gamma \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$$

Proof. □

Completeness. For $\mathsf{f} \in \mathsf{Fun}$, we define

$$\mathsf{MGJ}\ \mathsf{f} \triangleq \mathsf{GMem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}\colon \{\, \lambda h\, (m_g, v).\ \mathbb{E}_{[\![\mathsf{f}]\!]_{m_g}\, v}[h] \,\}\ \mathsf{f}\ \{\, \lambda h\, (m_g, v).\ h\, (m_g, v) \,\}$$

This judgement is *most general*, in the following sense:

Lemma B.4.

$$\Gamma \vdash \mathsf{MGJ}\ \mathsf{f} \wedge \vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\} \Rightarrow \Gamma \vdash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$$

Proof. Suppose $\Gamma \vdash \mathsf{MGJ}\ \mathsf{f}$ and $\vDash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$. We conclude $\Gamma \vdash Z\colon \{\, p \,\}\ \mathsf{f}\ \{\, q \,\}$ by one application of Rule **??**. To this end, we discharge the condition

$$\forall m_g\, v\, d.\ (\forall h \in \mathsf{GMem} \times \mathsf{Val} \to \mathbb{R}^{+\infty}.\ \mathbb{E}_d[h] \le \mathbb{E}_{[\![\mathsf{f}]\!]_{m_g}\, v}[h]) \Rightarrow (\forall z \in Z.\ \mathbb{E}_d[(q\, z)] \le p\, z\, (m_g, v))$$

using

$$\mathbb{E}_d[q\, z] \le \mathbb{E}_{[\![\mathsf{f}]\!]_{m_g}\, v}[q\, z] \le p\, z\, (m_g, v)$$

specializing h in the premise to $q\, z$. □

Lemma B.5. $\vdash \mathsf{MGJ}\ \mathsf{f}$