A Calculus for Amortized Expected Runtimes

KEVIN BATZ*, RWTH Aachen University, Germany
BENJAMIN LUCIEN KAMINSKI, Saarland University, Saarland Informatics Campus, Germany and University College London, United Kingdom
JOOST-PIETER KATOEN*, RWTH Aachen University, Germany
CHRISTOPH MATHEJA, Technical University of Denmark, Denmark
LENA VERSCHT*, RWTH Aachen University, Germany

We develop a weakest-precondition-style calculus à la Dijkstra for reasoning about *amortized expected runtimes of randomized algorithms with access to dynamic memory* — the aert calculus. Our calculus is truly quantitative, i.e. instead of Boolean valued predicates, it manipulates real-valued functions.

En route to the aert calculus, we study the ert calculus for reasoning about *expected runtimes* of Kaminski et al. [2018] extended by capabilities for handling dynamic memory, thus enabling *compositional* and *local* reasoning about *randomized data structures*. This extension employs *runtime separation logic*, which has been foreshadowed by Matheja [2020] and then implemented in Isabelle/HOL by Haslbeck [2021]. In addition to Haslbeck's results, we further prove soundness of the so-extended ert calculus with respect to an operational Markov decision process model featuring countably-branching nondeterminism, provide extensive intuitive explanations, and provide proof rules enabling separation logic-style verification for upper bounds on expected runtimes. Finally, we build the so-called *potential method* for amortized analysis into the ert calculus, thus obtaining the aert calculus. Soundness of the aert calculus is obtained from the soundness of the ert calculus and some probabilistic form of telescoping.

Since one needs to be able to handle *changes in potential* which can in principle be both positive or negative, the aert calculus needs to be — essentially — capable of handling certain signed random variables. A particularly pleasing feature of our solution is that, unlike e. g. Kozen [1985], we obtain a loop rule for our signed random variables, and furthermore, unlike e. g. Kaminski and Katoen [2017], the aert calculus makes do without the need for involved technical machinery keeping track of the integrability of the random variables.

Finally, we present case studies, including a formal analysis of a randomized delete-insert-find-any set data structure [Brodal et al. 1996], which yields a constant expected runtime per operation, whereas no deterministic algorithm can achieve this.

CCS Concepts: • **Theory of computation → Probabilistic computation**; **Invariants**; **Program specifications**; **Pre- and post-conditions**; **Program verification**; **Denotational semantics**; **Separation logic**.

Additional Key Words and Phrases: quantitative verification, randomized data structures, amortized analysis

*Batz, Katoen, and Verscht are supported by the ERC AdG 787914 FRAPPANT.

Authors' addresses: Kevin Batz, RWTH Aachen University, Germany, kevin.batz@cs.rwth-aachen.de; Benjamin Lucien Kaminski, Saarland University, Saarland Informatics Campus, Germany and University College London, United Kingdom, kaminski@cs.uni-saarland.de; Joost-Pieter Katoen, RWTH Aachen University, Germany, katoen@cs.rwth-aachen.de; Christoph Matheja, Technical University of Denmark, Denmark, chmat@dtu.dk; Lena Verscht, RWTH Aachen University, Germany, lena.verscht@rwth-aachen.de.

2475-1421/2023/1-ART67
https://doi.org/10.1145/3571260

ACM Reference Format:
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, Article 67 (January 2023), 30 pages. https://doi.org/10.1145/3571260

1 INTRODUCTION

Amortized analysis [Tarjan 1985] is a well-established method to analyze the runtime complexity of algorithms, in particular of those who manipulate dynamic data structures such as dynamically-sized lists, self-balancing trees, and so forth. The essence of amortized analysis is to average the runtime of a single operation *Op* over a long sequence of *Op*'s. Why is this useful? Suppose *Op* has *large worst-case runtime* and *small "normal-case" runtime*. A worst-case analysis of *Op* would tell us that *Op* performs poorly. However, when executing a *long sequence* of consecutive *Op*'s, it may be that the worst case inevitably occurs only very *seldomly*. The large runtimes of the small number of worst cases thus amortize over the large number of small runtimes of the normal cases. *On average*, the runtime of a single execution of *Op* is thus actually small. Amortized analysis hence yields results that are more realistic than worst-case analyses. Notice that amortized analysis is *not* the same as so-called *average case analysis*. The latter assumes a *probability distribution over all possible inputs* to *Op* and averages *Op*'s runtime over this distribution.

One popular technique for amortized analysis is the *potential method* (aka *physicist's method*) [Cormen et al. 2009; Tarjan 1985]. We will introduce this method in some detail in Section 5 and plot out how to make it probabilistic. For that, we will develop a calculus for reasoning about the *amortized runtime complexity of randomized algorithms*. Various randomized algorithms use dynamic data structures such as randomized meldable heaps, randomized splay trees and randomized search trees. An amortized analysis gives a detailed account of the expected runtime of a randomized algorithm and extends (read: refines) existing runtime analysis techniques for probabilistic programs [Avanzini et al. 2019; Kaminski et al. 2018; Ngo et al. 2018]. For instance, an amortized analysis of the complexity of the randomized delete-insert-find-any set data structure [Brodal et al. 1996] yields a constant expected runtime per operation, whereas no deterministic algorithm can achieve this. The aim of this paper is to develop a *systematic, calculational method for carrying out an amortized runtime analysis of randomized algorithms on source code level*. Our method is in the spirit of weakest-precondition style reasoning. That is to say, we present a syntax-oriented technique to determine the amortized expected time of randomized algorithms by applying backward reasoning. Our technique yields amortized upper bounds on the expected runtime complexity.

Our starting point is the ert-calculus for reasoning about expected runtimes of probabilistic pointer programs by Haslbeck [2021], which extends the ert-calculus of Kaminski et al. [2018] by the principles of separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002]. The main challenge here is that classical separation logic connectives do not admit a frame rule — the key to compositional and local reasoning — for runtime *over*-approximations. Based on a suggestion in [Matheja 2020, Chapter 9], Haslbeck [2021] investigated runtime analogues to these separating connectives, separating sum and (its adjoint) separating monus, thus obtaining a real-valued "logic" — *runtime separation logic* (RSL) — upon which the ert-calculus is built.

Haslbeck [2021] has mechanized the ert-calculus in Isabelle/HOL and has proven various properties such as the validity of the frame rule. In addition to Haslbeck's results, we further prove *soundness* of ert by establishing a strong correspondence to a simple operational cost model defined in terms of Markov decision processes (MDPs) [Puterman 2005]. This resembles the approach adopted for quantitative separation logic (QSL) [Batz et al. 2019], a version of separation logic to reason about the correctness (not the runtime) of probabilistic pointer programs. The treatment of rewards in the operational model is rather different, however, as time may elapse at arbitrary

steps in a program execution. This contrasts the situation for QSL where rewards corresponding to post"conditions" are only collected in states indicating successful program termination. The proof principle to establish the correspondence between RSL and the operational MDP interpretation is new and relies on Bellman equations [Puterman 2005] and Blackwell's theorem [Blackwell 1967].

In a second step, we extend ert to aert — a calculus for reasoning about *amortized expected runtimes using the potential method*. We show that aert recovers what is essentially the *telescoping* property of the classical potential method in the probabilistic setting. That is, for probabilistic program C and potential function π:

$$\overbrace{\mathsf{aert}_\pi\,[\![C]\!]\,(0)}^{\text{amortized expected runtime of } C} \quad = \quad \underbrace{\mathsf{ert}\,[\![C]\!]\,(\pi) \;-\; \pi}_{\text{expected runtime of } C \,+\, \text{expected change of the potential caused by } C}$$

This result enables us to derive a frame rule for local reasoning about amortized expected runtimes. As indicated above, an integral part of the potential method is reasoning about differences in potential when performing an operation. Such differences can potentially become negative. This seems rather innocent, but technically it is not. Existing weakest precondition reasoning rules for probabilistic programs restrict random variables to be non-negative (at least for loops) [Kozen 1985; McIver and Morgan 2005]. This is for a good reason as it avoids issues with integrability of expected values. Extensions which can handle signed random variables [Kaminski and Katoen 2017] are technically involved. (Indeed a naïve approach would have been to extend the classical ert calculus with RSL and with these signed random variables.) This paper shows that this complicated machinery is not needed to treat amortization. Another interesting result is that our framework recovers a classical result from amortized complexity analysis over sequences of programs (Theorem 5.5).

We illustrate our amortized runtime calculus on a few examples such as a randomized dynamic list as well as an analysis of the insert-delete-find-any set data structure from [Brodal et al. 1996]. The latter example is of interest as it only has a constant amortized runtime per operation under randomization. To the best of our knowledge, our analysis is the first such analysis using the potential method and on source code level.

To summarize, the main contributions of this paper are:

- a compositional, weakest-precondition-style calculus to reason about amortized expected runtimes of randomized algorithms that features local reasoning,
- invariant-based reasoning for loops and proof rules enabling the separation logic-style verification of such runtimes,
- soundness of our methods by providing a close correspondence to an operational model based on countably-branching Markov decision processes, and
- a source-code-level analysis based on the potential method for amortized complexity on the insert-delete-find-any data structure [Brodal et al. 1996].

Structure of the paper. Section 2 introduces our model programming language, where Section 2.3 defines its operational semantics. Section 3 studies and explains runtime separation logic. In Section 4, we present the ert calculus for expected runtimes alongside, where Section 4.4 features its soundness proof. We present the aert calculus for reasoning about *amortized* expected runtimes in Section 5. We consider related work in Section 6. Proofs and details on the case studies are found in an extended version of this paper, which is available online [Batz et al. 2022b].

2 PROBABILISTIC POINTER PROGRAMS

We will employ an imperative model language à la Dijkstra's guarded commands language adapted from [Batz et al. 2019] with three main features: (1) *probabilistic choices*, (2) a *customizable runtime model*, and (3) statements for *accessing and manipulating dynamic memory*.

2.1 Program States

Program states have two components: (1) a *stack* assigning values to program variables and (2) a *heap* modeling the dynamic memory which stores values at (dynamically) allocated locations.

Stacks. A stack $\mathfrak{s}$ is a mapping from *variables* taken from a finite set Vars to values taken from a set Vals; in our case values are natural numbers, i.e. Vals $= \mathbb{N}$. Hence, the set of *stacks* is given by

$$\mathsf{Stacks} \;=\; \{\, \mathfrak{s} \mid \mathfrak{s} \colon \mathsf{Vars} \to \mathsf{Vals} \,\} \;.$$

Heaps. A heap $\mathfrak{h}$ maps finitely many *memory locations* taken from the set $\mathsf{Loc} = \mathbb{N}_{>0}$ to values; the value 0 is *not* a valid location and represents the null pointer. Hence, the set of *heaps* is given by

$$\mathsf{Heaps} \;=\; \{\, \mathfrak{h} \mid \mathfrak{h} \colon L \to \mathsf{Vals}, \quad L \subseteq \mathsf{Loc}, \quad L \text{ finite} \,\} \;.$$

For a given heap $\mathfrak{h} \colon L \to \mathsf{Vals}$, we denote by $\mathsf{dom}\,(\mathfrak{h})$ its *domain*, i.e. $\mathsf{dom}\,(\mathfrak{h}) = L$. We write $\mathfrak{h}_1 \perp \mathfrak{h}_2$ to indicate that the domains of heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are disjoint, i.e.

$$h_1 \perp h_2 \quad \text{iff} \quad \mathsf{dom}\,(h_1) \cap \mathsf{dom}\,(h_2) = \emptyset \;.$$

For heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ with disjoint domains, i.e. $\mathfrak{h}_1 \perp \mathfrak{h}_2$, their *union* $\mathfrak{h}_1 \star \mathfrak{h}_2$ is given by

$$\mathfrak{h}_1 \star \mathfrak{h}_2 \colon \quad \mathsf{dom}\,(\mathfrak{h}_1) \cup \mathsf{dom}\,(\mathfrak{h}_2) \to \mathsf{Vals}, \quad \ell \mapsto \begin{cases} \mathfrak{h}_1(\ell), & \text{if } \ell \in \mathsf{dom}\,(\mathfrak{h}_1) \\ \mathfrak{h}_2(\ell), & \text{if } \ell \in \mathsf{dom}\,(\mathfrak{h}_2) \end{cases} .$$

If the domains of $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are not disjoint, $\mathfrak{h}_1 \star \mathfrak{h}_2$ is undefined.

We denote by $\mathfrak{h}_\emptyset$ the *empty heap* with $\mathsf{dom}\,(\mathfrak{h}_\emptyset) = \emptyset$. Moreover, $\{\ell \mapsto v\}$ denotes the heap $\mathfrak{h}$ that consists of a single memory location ℓ which stores value v, i.e. $\mathsf{dom}\,(\mathfrak{h}) = \{\ell\}$ and $\mathfrak{h}(\ell) = v$. Note that $\mathfrak{h} \star \mathfrak{h}_\emptyset = \mathfrak{h}_\emptyset \star \mathfrak{h} = \mathfrak{h}$ for any heap $\mathfrak{h}$, whereas $\{\ell \mapsto v\} \star \{\ell \mapsto w\}$ is always undefined.

Program states. The set States of *program states* consists of all stack-heap pairs, i.e.

$$\mathsf{States} \;=\; \{\, (\mathfrak{s}, \mathfrak{h}) \mid \mathfrak{s} \in \mathsf{Stacks},\ \mathfrak{h} \in \mathsf{Heaps} \,\} \;.$$

Given a program state $(\mathfrak{s}, \mathfrak{h})$, we denote by $\mathfrak{s}(e)$ the evaluation of an arithmetic expression e in stack $\mathfrak{s}$, i.e. the value that is obtained by evaluating expression e after replacing every occurrence of every variable x in e by its assigned value $\mathfrak{s}(x)$. Analogously, we denote by $\mathfrak{s} \models \varphi$ that the boolean expression φ evaluates to true in stack $\mathfrak{s}$. We require that both arithmetic and boolean expressions are *pure*, meaning that they only depend on variables in Vars and *not* on the heap. Evaluating an expression thus never causes any side effects, such as dereferencing an unallocated location.

We write $\mathfrak{s}\,[x \leftarrow v]$ for stack $\mathfrak{s}$ in which the value of variable x has been updated to $v \in \mathsf{Vals}$, i.e.[1]

$$\mathfrak{s}\,[x \leftarrow v] \;=\; \lambda y.\, \begin{cases} v, & \text{if } y = x \\ \mathfrak{s}(y), & \text{if } y \neq x \;. \end{cases}$$

[1]We use λ–expressions to construct functions: $\lambda\xi.\ \epsilon$ stands for the function that, when applied to an argument α, evaluates to ϵ in which every occurrence of ξ is replaced by α.

Likewise, we write $\mathfrak{h}\,[\ell \leftarrow v]$ for the heap $\mathfrak{h}$ in which the value stored at location ℓ has been updated to v. Formally, if ℓ is allocated in $\mathfrak{h}$, i.e. if $\ell \in \mathrm{dom}\,(\mathfrak{h})$ (otherwise $\mathfrak{h}\,[\ell \leftarrow v]$ is undefined),

$$\mathfrak{h}\,[\ell \leftarrow v] \;=\; \lambda\,\ell'.\; \begin{cases} v, & \text{if } \ell' = \ell \\ \mathfrak{h}(\ell'), & \text{if } \ell' \neq \ell\,. \end{cases}$$

2.2 The Heap-Manipulating Probabilistic Guarded Command Language

We now present the syntax of our model programming language and briefly discuss its intuitive semantics and runtime model; a formal semantics and runtime model are provided in Section 2.3.

Syntax. Programs written in the *heap-manipulating probabilistic guarded command language*, denoted hpGCL, are given by the context-free grammar

$C \longrightarrow$	$\texttt{tick}\,(e)$	(time consumption)	$\mid$	$x := e$	(variable assignment)
	$\mid\; x := \texttt{alloc}\,(e)$	(memory allocation)	$\mid$	$\langle e \rangle := e'$	(heap mutation)
	$\mid\; x := \langle e \rangle$	(heap lookup)	$\mid$	$\texttt{free}(e)$	(memory deallocation)
	$\mid\; \{\,C\,\}\,[p]\,\{\,C\,\}$	(probabilistic choice)	$\mid$	$\texttt{if}\,(\,\varphi\,)\,\{\,C\,\}\,\texttt{else}\,\{\,C\,\}$	(conditional choice)
	$\mid\; C \fatsemi C$	(sequential composition)	$\mid$	$\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$	(while loop)

where $x \in \mathrm{Vars}$, $e, e_1, \ldots, e_n$ are arithmetic expressions over variables that evaluate to values in $\mathrm{Vals} = \mathbb{N}$, and φ is a Boolean expression over variables. Moreover, p is an arithmetic expression over variables that evaluates to a rational probability, i.e. $\mathfrak{s}(p) \in [0,1] \cap \mathbb{Q}$ holds for all stacks $\mathfrak{s}$.

Intuitive semantics. Assignments, sequential composition, conditionals, and loops are standard. The probabilistic choice $\{\,C_1\,\}\;[p]\;\{\,C_2\,\}$ executes C_1 with probability $\mathfrak{s}(p)$ and C_2 with probability $1 - \mathfrak{s}(p)$. $\texttt{tick}\,(e)$ does not affect the program state but takes e units of time; see below.

The remaining statements access or manipulate the dynamic memory. $x := \texttt{alloc}\,(e)$ allocates a block of e consecutive, previously unallocated, and nondeterministically chosen memory locations, initializes their contents with zero,[2] and assigns to x the first of those locations; attempting to allocate an empty block, e. g. via $x := \texttt{alloc}\,(0)$, only assigns a *nondeterministic* value to x but does not affect the heap. Since we have an infinite reservoir of locations, *memory allocation never fails.*

The mutation statement $\langle e \rangle := e'$ changes the value at location e to e'. Mutation *can* fail: If location e is not allocated, we encounter *undefined behavior* (most likely a crash) due to a *memory fault.* The lookup statement $x := \langle e \rangle$ assigns the value stored at location e to variable x if location e is allocated and otherwise causes a memory fault. Finally, the deallocation statement $\texttt{free}(e)$ disposes of location e if allocated and causes a memory fault otherwise.

Runtime model. Our ultimate goal is to reason about (amortized) expected runtimes of hpGCL programs. To deal with a variety of runtime models, we do *not* assign particular runtimes to individual statements. Rather, we model runtime using `tick` statements; executing $\texttt{tick}\,(e)$ takes e units of time (where e is evaluated in the current program state). All other hpGCL statements have a runtime of zero with one exception: whenever we encounter a memory fault, this constitutes for us *undefined behavior* — anything can happen, including non-termination. Hence, our runtime model for memory faults is that they have unbounded, i.e. infinite, runtime.

Modified variables. We denote by $\mathrm{Mod}\,(C) \subseteq \mathrm{Vars}$ the set of variables that are potentially *modified* by program C, i.e. occur on the left-hand side of a variable assignment $x := e$ in C.

[2] Similarly to C's `calloc`.

2.3 Formal Operational Semantics

We give operational semantics to programs by (1) defining a *small-step execution relation* $\to$ describing how (and how probable) statements manipulate program states, (2) constructing a *Markov decision process* (MDP) based on $\to$, and (3) introducing a *reward function modeling runtimes*. Expected runtimes of program executions will then be the expected rewards of a corresponding MDP.

Configurations. The set of *program configurations* is given by

$$\mathsf{Conf} \quad = \quad \big(\mathsf{hpGCL} \cup \{\mathit{term}, \mathit{fault}\}\big) \times \mathsf{Stacks} \times \mathsf{Heaps} \quad \cup \quad \{\mathit{sink}\}\,.$$

A configuration is an hpGCL program C, or *term* indicating fault-free termination, or *fault*, indicating a memory fault, together with a program state $(\mathfrak{s}, \mathfrak{h})$. For technical reasons, we also add a *sink* configuration which we enter after program termination.

Execution relation. The steps of our operational semantics are given by an execution relation

$$\to \quad \subseteq \quad \mathsf{Conf} \times \mathsf{Prob} \times \mathsf{Vals} \times \mathsf{Conf}\,,$$

where Prob is the set of transition probabilities[3] and Vals are the allocation values which are chosen nondeterministically; if the step is not an allocation, we default to 0. Hence, $c \xrightarrow[p]{v} c'$ (denoting $(c, p, v, c') \in \to$) indicates that our program performs *one step* from c to c' with probability p while allocation value v has been chosen. To avoid cluttering, we omit p if $p = 1$ and v if $v = 0$. $\to$ is given by the inference rules in Figure 1 which match the intuitive semantics of Section 2.2. E. g., the rule for x := `alloc`(e) chooses a location v from e consecutive unallocated locations. These locations are added to the heap with their content initialized to 0. In particular, allocation never fails (steps into $(\mathit{fault}, \ldots)$) and causes infinite branching over all such memory locations v.

Markov Decision Processes. We formalize the expected runtime of programs as expected rewards of MDPs. While we broadly adhere to Baier and Katoen [2008, Chapter 10], we consider MDPs with infinitely many states, infinite branching (actions), and (non-discounted) rewards. A thorough discussion of such MDPs is found in [Puterman 2005, Chapter 7]. Intuitively, an MDP is a transition system that assigns to every state one or more[4] probability distributions (distinguished by an action) over successor states. Moreover, whenever we *leave* a state, we collect a reward.

Definition 2.1 (Markov Decision Process). A *Markov decision process* $\mathcal{M}$ is a tuple

$$\mathcal{M} \quad = \quad (\mathcal{S}, \mathsf{Act}, \mathsf{prob}, \sigma_{\text{init}}, \mathsf{rew})\ ,$$

where $\mathcal{S}$ is a countable set of *states*, Act is a countable set of *actions*, $\mathsf{prob}\colon \mathcal{S} \times \mathsf{Act} \times \mathcal{S} \to [0,1]$ is the *transition probability function*[5], $\sigma_{\text{init}} \in \mathcal{S}$ is the *initial state*, and $\mathsf{rew}\colon \mathcal{S} \to \mathbb{R}^{\infty}_{\geq 0}$ assigns to every state a reward that is collected when leaving a state. △

Consider until further notice a fixed MDP $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, \mathsf{prob}, \sigma_{\text{init}}, \mathsf{rew})$. Our goal is to determine the maximal expected reward collected over all possible paths that start in σ_{init}. For that, we first resolve the nondeterminism, arising from multiple actions being enabled, by a *scheduler* $\mathfrak{S}\colon \mathcal{S}^{+} \to \mathsf{Act}$ which chooses an action for every history of states. We denote by Sched the set of all schedulers.

[3] Formally, we set $\mathsf{Prob} = ([0,1] \cap \mathbb{Q}) \cup \{1 - 1/2\}$, where $1 - 1/2 \neq 1/2$ is a *formal value* that allows us to represent two distinguishable steps from a configuration c both to the same configuration c', each with probability $1/2$.
[4] due to nondeterminism
[5] i.e., (1) for all states $\sigma \in \mathcal{S}$ and actions $a \in \mathsf{Act}$, $\mathrm{dist}(\sigma, a) = \sum_{\sigma' \in \mathcal{S}} \mathsf{prob}(\sigma, a, \sigma') \in \{0, 1\}$ and (2) for all $\sigma \in \mathcal{S}$ there exists an action $a \in \mathsf{Act}$ such that $\mathrm{dist}(\sigma, a) = 1$. We call the actions a with $\mathrm{dist}(\sigma, a) = 1$ the *enabled actions* of state σ.

$$\texttt{tick}\,(e)\,,\,\mathfrak{s},\,\mathfrak{h}\;\to\;\mathit{term},\,\mathfrak{s},\,\mathfrak{h}\qquad \mathit{term},\,\mathfrak{s},\,\mathfrak{h}\;\to\;\mathit{sink}\qquad \mathit{sink}\to\mathit{sink}\qquad \mathit{fault},\,\mathfrak{s},\,\mathfrak{h}\to\mathit{sink}$$

$$\{\,C_1\,\}\;[p]\;\{\,C_2\,\}\,,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[\mathfrak{s}(p)]{}\;C_1,\,\mathfrak{s},\,\mathfrak{h}\qquad\qquad \{\,C_1\,\}\;[p]\;\{\,C_2\,\}\,,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[1-\mathfrak{s}(p)]{}\;C_2,\,\mathfrak{s},\,\mathfrak{h}$$

$$x := e,\,\mathfrak{s},\,\mathfrak{h}\;\to\;\mathit{term},\,\mathfrak{s}\,[x\leftarrow\mathfrak{s}(e)]\,,\,\mathfrak{h}\qquad \texttt{if}\;(\,\varphi\,)\;\{\,C_1\,\}\;\texttt{else}\;\{\,C_2\,\}\,,\,\mathfrak{s},\,\mathfrak{h}\;\to\;\begin{cases} C_1,\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}\models\varphi\\ C_2,\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}\not\models\varphi\end{cases}$$

$$x := \mathit{alloc}\,(e)\,,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow{v}\;\begin{cases}\mathit{term},\,\mathfrak{s}\,[x\leftarrow v]\,,\,\mathfrak{h}\star\mathfrak{h}' & \text{if } \mathfrak{s}(e)=n>0 \text{ and } \mathfrak{h}\perp\mathfrak{h}'\\ & \quad\text{and}\quad \mathfrak{h}'=\{v\mapsto 0\}\star\ldots\star\{v+n-1\mapsto 0\}\\ \mathit{term},\,\mathfrak{s}\,[x\leftarrow v]\,,\,\mathfrak{h} & \text{if } \mathfrak{s}(e)=0 \text{ and } v\in\mathrm{Vals}\end{cases}$$

$$x := \langle e\rangle,\,\mathfrak{s},\,\mathfrak{h}\;\to\;\begin{cases}\mathit{term},\,\mathfrak{s}\,[x\leftarrow\mathfrak{h}(v)]\,,\,\mathfrak{h} & \text{if } \mathfrak{s}(e)=v\in\mathrm{dom}\,(\mathfrak{h})\\ \mathit{fault},\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}(e)\notin\mathrm{dom}\,(\mathfrak{h})\end{cases}$$

$$\langle e\rangle := e',\,\mathfrak{s},\,\mathfrak{h}\;\to\;\begin{cases}\mathit{term},\,\mathfrak{s},\,\mathfrak{h}\,[v\leftarrow\mathfrak{s}(e')] & \text{if } \mathfrak{s}(e)=v\in\mathrm{dom}\,(\mathfrak{h})\\ \mathit{fault},\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}(e)\notin\mathrm{dom}\,(\mathfrak{h})\end{cases}$$

$$\mathit{free}(e),\,\mathfrak{s},\,\mathfrak{h}\;\to\;\begin{cases}\mathit{term},\,\mathfrak{s},\,\mathfrak{h}' & \text{if } \mathfrak{h}=\mathfrak{h}'\star\{\mathfrak{s}(e)\mapsto v\} \text{ for some } v\in\mathrm{Vals}\\ \mathit{fault},\,\mathfrak{s},\,\mathfrak{h} & \text{otherwise}\end{cases}$$

$$\texttt{while}\;(\,\varphi\,)\,\{\,C\,\},\,\mathfrak{s},\,\mathfrak{h}\;\to\;\begin{cases}C\fatsemi\texttt{while}\;(\,\varphi\,)\,\{\,C\,\},\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}\models\varphi\\ \mathit{term},\,\mathfrak{s},\,\mathfrak{h} & \text{if } \mathfrak{s}\not\models\varphi\end{cases}$$

$$\frac{C_1,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;C_1',\,\mathfrak{s}',\,\mathfrak{h}'}{C_1\fatsemi C_2,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;C_1'\fatsemi C_2,\,\mathfrak{s}',\,\mathfrak{h}'}\qquad \frac{C_1,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;\mathit{term},\,\mathfrak{s}',\,\mathfrak{h}'}{C_1\fatsemi C_2,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;C_2,\,\mathfrak{s}',\,\mathfrak{h}'}\qquad \frac{C_1,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;\mathit{fault},\,\mathfrak{s},\,\mathfrak{h}}{C_1\fatsemi C_2,\,\mathfrak{s},\,\mathfrak{h}\;\xrightarrow[p]{v}\;\mathit{fault},\,\mathfrak{s},\,\mathfrak{h}}$$

Fig. 1. The inference rules determining the execution relation $\to$ of our operational semantics. Here, $c \xrightarrow[p]{v} c'$ indicates a step from c to c' with probability p in which allocation value v has been chosen. To avoid clutter, we omit p and v if they equal their default values $p = 1$ and $v = 0$. Hence, $c \to c'$ means $c \xrightarrow[1]{0} c'$.

Once a scheduler $\mathfrak{S}$ is fixed, a *path* is a sequence $\sigma_0\sigma_1\sigma_2\ldots$ of states starting in $\sigma_0 = \sigma_{\text{init}}$ such that there is non-0 probability (under the distribution determined by $\mathfrak{S}$) of moving from σ_n to σ_{n+1}. Formally, the set $\mathsf{Paths}_{=n}(\mathfrak{S})$ of *paths of length* $n \in \mathbb{N}$ *induced by scheduler* $\mathfrak{S}$ is given by

$$\mathsf{Paths}_{=n}(\mathfrak{S}) \;=\; \{\;\sigma_0\ldots\sigma_{n-1} \mid \sigma_0=\sigma_{\text{init}},\;\forall i\in\{1,\ldots,n-1\}\colon \mathsf{prob}(\sigma_{i-1},\mathfrak{S}(\sigma_0\ldots\sigma_{i-1}),\sigma_i)>0\;\}\;.$$

The total probability of taking a path $\sigma_0\ldots\sigma_{n-1}$ and the total reward collected along that path are then obtained by multiplying transition probabilities and summing up rewards, that is,

$$\mathsf{prob}(\sigma_0\ldots\sigma_{n-1}) \;=\; \prod_{i=1}^{n-1}\mathsf{prob}(\sigma_{i-1},\mathfrak{S}(\sigma_0\ldots\sigma_{i-1}),\sigma_i) \quad\text{and}\quad \mathsf{rew}(\sigma_0\ldots\sigma_{n-1}) \;=\; \sum_{i=0}^{n-2}\mathsf{rew}(\sigma_i)\;.$$

The *total expected reward* $\mathsf{ExpRew}\,(\mathcal{M})$ of MDP $\mathcal{M}$ is then the maximal[6] (over all schedulers $\mathfrak{S}$ and path lengths n) accumulated reward collected along all paths of length n and induced by scheduler $\mathfrak{S}$ weighted by each path's probability. Formally, $\mathsf{ExpRew}\,(\mathcal{M})$ is given by

$$\mathsf{ExpRew}\,(\mathcal{M}) \;=\; \sup_{\mathfrak{S}\in\mathrm{Sched}}\;\sup_{n\in\mathbb{N}}\;\sum_{\pi\in\mathsf{Paths}_{=n}(\mathfrak{S})}\mathsf{prob}(\pi)\cdot\mathsf{rew}(\pi)\;.$$

[6] since the reward is used for modeling the runtime

The Operational Markov Decision Process. We will now construct an MDP

$$\mathcal{M}[\![C, \mathfrak{s}, \mathfrak{h}]\!] \quad = \quad \big(\,\mathsf{Conf},\ \mathsf{Vals},\ \mathit{prob},\ (C,\ \mathfrak{s},\ \mathfrak{h}),\ \mathit{rew}\,\big)$$

whose expected reward captures the expected runtime of executing program C on state $(\mathfrak{s}, \mathfrak{h})$. We call $\mathcal{M}[\![C, \mathfrak{s}, \mathfrak{h}]\!]$ the *operational MDP* of C and $(\mathfrak{s}, \mathfrak{h})$. This MDP's set of states are the program configurations Conf, its action set are the allocation values Vals, and its initial state is the configuration $(C, \mathfrak{s}, \mathfrak{h})$. The transition probability function *prob* is obtained from the execution relation $\rightarrow$ in Figure 1 by accumulating the probability of all steps from c to c' which choose the same value v, i.e.

$$\mathit{prob}\colon \mathsf{Conf} \times \mathsf{Vals} \times \mathsf{Conf}, \qquad (c, v, c') \mapsto \sum_{p\colon c \xrightarrow[p]{v} c'} p\ .$$

By construction of our execution relation, *prob* is a well-defined transition probability function:

(1) for all $c \in \mathsf{Conf}$ and $v \in \mathsf{Vals}$, we have $\sum_{c' \in \mathsf{Conf}} \mathit{prob}(c, v, c') \in \{0, 1\}$ and
(2) for all $c \in \mathsf{Conf}$, there exists $v \in \mathsf{Vals}$ such that $\sum_{c' \in \mathsf{Conf}} \mathit{prob}(c, v, c') \ = \ 1$.

(2) follows from the fact that in $\rightarrow$ every configuration has a successor ($\mathit{sink} \rightarrow \mathit{sink}$ if necessary).

Finally, the reward function *rew* reflects our runtime model, where collected reward corresponds to accumulated runtime: we collect reward e whenever we execute $\texttt{tick}\,(e)$ and reward ∞ whenever we encounter a memory fault. Executing any other hpGCL statement consumes no runtime and thus reward 0 is collected. Hence, the reward function *rew* is given by

$$\mathit{rew}\colon \mathsf{Conf} \to \mathbb{R}^{\infty}_{\geq 0}\,, \qquad c \mapsto \begin{cases} \mathfrak{s}(e), & \text{if } c = (\texttt{tick}\,(e)\,,\ \mathfrak{s},\ \mathfrak{h}) \text{ or } c = (\texttt{tick}\,(e)\ ;\ C,\ \mathfrak{s},\ \mathfrak{h}) \\ \infty, & \text{if } c = (\mathit{fault},\ \mathfrak{s},\ \mathfrak{h}) \\ 0, & \text{otherwise.} \end{cases}$$

Put together, we define the *expected runtime* of hpGCL program C on initial program state $(\mathfrak{s}, \mathfrak{h})$ as the expected reward $\mathsf{ExpRew}\big(\mathcal{M}[\![C, \mathfrak{s}, \mathfrak{h}]\!]\big)$ of the operational MDP of C and $(\mathfrak{s}, \mathfrak{h})$.

3 RUNTIME SEPARATION LOGIC

We will now study *runtime separation logic* (RSL), a real-valued "logic" in the spirit of [Batz et al. 2019], suitable for use in reasoning about upper bounds on expected runtimes of randomized algorithms that manipulate dynamic data structures. Its key ingredient are two separating connectives, $\oplus$ and $\mathrel{-\!\!\ominus}$, which replace the standard separation logic connectives $\star$ and $\mathrel{-\!\!*}$. Though rediscovered independently by us, RSL has been proposed for future investigation by Matheja [2020] and then investigated by Haslbeck [2021], with almost-exclusive focus on its meta-theory.

3.1 Runtimes

Classical program verification employs *logical predicates* which evaluate to true or false for reasoning about program correctness. Our goal is to reason about a program's *expected runtime*, i.e. the average (possibly unbounded) number of time units it takes to execute the program. To this end, we use genuine *quantities*, which map states to *numbers* instead of truth values.

Definition 3.1 (Runtimes). The set of *runtimes* is given by

$$\mathbb{T} \quad = \quad \big\{\, f \;\big|\; f\colon \mathsf{States} \to \mathbb{R}^{\infty}_{\geq 0} \,\big\}\ .$$

We use metavariables f, g, u, and variations for runtimes. Together with the order

$$f \ \preceq\ g \qquad \text{iff} \qquad \forall\, (\mathfrak{s}, \mathfrak{h}) \in \mathsf{States}\colon \quad f(\mathfrak{s}, \mathfrak{h}) \ \leq\ g(\mathfrak{s}, \mathfrak{h})$$

the set of runtimes forms a complete lattice.

We call a runtime f *finite* if $\forall\, (\mathfrak{s}, \mathfrak{h}) \in \mathsf{States}\colon f(\mathfrak{s}, \mathfrak{h}) < \infty$ and denote this by $f \prec \infty$. △

Table 1. Standard separation logic atoms and their semantics.

φ	$(\mathfrak{s},\mathfrak{h}) \models \varphi$ iff ...
emp	$\mathrm{dom}\,(\mathfrak{h}) = \emptyset$
$e \mapsto -$	$\mathrm{dom}\,(\mathfrak{h}) = \{\mathfrak{s}(e)\}$
$e \mapsto e'$	$\mathrm{dom}\,(\mathfrak{h}) = \{\mathfrak{s}(e)\}$ and $\mathfrak{h}(\mathfrak{s}(e)) = \mathfrak{s}(e')$

For any constant $r \in \mathbb{R}^{\infty}_{\geq 0}$, we simply write r for the *constant runtime* $\lambda(\mathfrak{s},\mathfrak{h}).\ r$. Similarly, we write x for the runtime $\lambda(\mathfrak{s},\mathfrak{h}).\ \mathfrak{s}(x)$. We write size for the runtime corresponding to the number of allocated memory locations on the heap, i.e. $\mathrm{size} = \lambda(\mathfrak{s},\mathfrak{h}).\ |\mathrm{dom}\,(\mathfrak{h})\,|$.

3.2 Truth vs. Runtimes

It is not overly helpful, in general, to think of runtimes as many-valued truth values. However, if one subscribes to the fairly widespread conception that *truth is something desirable* and *falsehood is something undesirable*, then one could make the following analogy: In the world of runtimes — usually — *a runtime of* 0 *is something desirable*, *higher numbers are less and less desirable*, and ∞ *is something undesirable*. In that sense, one can well draw a connection between the undesirable false and ∞ on the one hand, and less well a connection between the desirable true and all finite runtimes. This analogy is to some extent also reflected in our runtime model of the heap-manipulating hpGCL constructs: Memory faults (certainly undesirable!) cause infinite runtime.

3.3 Gatekeeper Brackets

We can turn Boolean predicates $\varphi\colon \mathrm{States} \to \{\mathrm{true}, \mathrm{false}\}$ into runtimes in a way that preserves the above analogy: The *gatekeeper bracket* $\wr\varphi\wr$ (reminiscent of *Iverson brackets*) of a predicate φ is defined as the function

$$\wr\varphi\wr\colon \quad \mathrm{States} \to \{0, \infty\}, \quad \wr\varphi\wr(\mathfrak{s},\mathfrak{h}) \;=\; \begin{cases} 0, & \text{if } (\mathfrak{s},\mathfrak{h}) \models \varphi \\ \infty, & \text{if } (\mathfrak{s},\mathfrak{h}) \not\models \varphi. \end{cases}$$

$\wr\varphi\wr$ can be understood as a gatekeeper which checks whether the documents we present to them (i.e. the current state $(\mathfrak{s},\mathfrak{h})$) are in accordance with their internal guidelines (i.e. the predicate φ). In the desirable case that our documents check out (i.e. if $(\mathfrak{s},\mathfrak{h}) \models \varphi$), they will let us pass with zero further delay: $\wr\varphi\wr(\mathfrak{s},\mathfrak{h}) = 0$. In the undesirable case that our documents do not check out (i.e. if $(\mathfrak{s},\mathfrak{h}) \not\models \varphi$), the gatekeeper will hold us up for infinitely long: $\wr\varphi\wr(\mathfrak{s},\mathfrak{h}) = \infty$.

3.4 Separation Logic Atoms

We will often specify memory-safety constraints using the standard (Boolean) separation logic atoms collected in Table 1 (cf., [Ishtiaq and O'Hearn 2001; Reynolds 2002]). The *empty heap predicate* **emp** specifies that *no* memory location is allocated; the predicate $e \mapsto -$ specifies that exactly one location, namely e, is allocated on the heap, and the *points-to predicate* $e \mapsto e'$ specifies also that precisely location e is allocated on the heap and moreover that it stores content e'. For a separation logic atom α, we write $\wr\alpha\wr$ to obtain its gatekeeper bracket.

3.5 Standard Connectives on Runtimes

3.5.1 Addition. We define standard mathematical operations (addition, multiplication, minimum, etc.) on runtimes pointwise, e. g. $f + g = \lambda(\mathfrak{s}, \mathfrak{h}).\ f(\mathfrak{s}, \mathfrak{h}) \ + \ g(\mathfrak{s}, \mathfrak{h})$. *Addition* (+) aggregates undesirableness, namely the runtime f *and* the runtime g. In that sense, addition can be thought of as the runtime analogue to *logical conjunction* aka *logical and* ($\wedge$) which also aggregates undesirableness, namely *falsehood*: If either a or b are false, $a \wedge b$ aggregates this falsehood and becomes itself false. This interpretation is also compatible when using gatekeepers: $\llbracket \varphi \wedge \psi \rrbracket = \llbracket \varphi \rrbracket + \llbracket \psi \rrbracket$.

3.5.2 Monus. The (pointwise) *monus* operation $g \dot{-} f = \max\{g - f, 0\}$ can be read as first carving out the runtime f from the runtime g and then measuring only the remaining runtime. Monus is the adjoint of addition, satisfying $g \preceq u + f$ iff $g \dot{-} f \preceq u$. Consequently, monus is the runtime analogue to logical implication ($\rightarrow$) in that $\alpha \dot{-} \beta$ corresponds to $\beta \rightarrow \alpha$. Keeping in mind that non-zero runtime is undesired, implication also carves out undesirableness, namely falsehood: For $a \rightarrow b$, whatever falsehood a carries is carved out from b. Indeed, if a is false, then we carve out *all* the falsehood from b (since we are in a Boolean realm). Thus, there cannot remain any falsehood and $a \rightarrow b$ is true in this case. Dually, if a is true, then there is no falsehood to be carved out from b. Thus, there remains only whichever falsehood b already carries and $a \rightarrow b$ is just b in this case.

Compatibility for gatekeepers is given by $\llbracket \varphi \rightarrow \psi \rrbracket = \llbracket \varphi \rrbracket \dot{-} \llbracket \psi \rrbracket$ when using the convention $\infty \dot{-} \infty = 0$. Even $\infty \dot{-} \infty = 0$ fits with our intuition of carving out undesirableness: ∞ is the most undesired, and $\infty \dot{-} \infty$ would thus carve out *all* undesirableness out of the most undesired. What remains is no undesirableness whatsoever: 0.

3.5.3 Minimum. The *minimum* of two runtimes, denoted $f \sqcap g$, is the runtime analogue of logical disjunction ($\vee$). Applied to arbitrary runtimes, we can read $f \sqcap g$ as a preference for smaller, i.e. more desirable, runtimes, thus reflecting that we ultimately wish to reason about as tight as possible upper bounds. In particular, we prefer a finite runtime over an infinite one indicating undesired behavior. Analogously, $a \vee b$ prefers the more desirable (more true) truth value. Compatibility for gatekeepers is given by $\llbracket \varphi \vee \psi \rrbracket = \llbracket \varphi \rrbracket \sqcap \llbracket \psi \rrbracket$.

3.5.4 Multiplication. We typically use runtime multiplication $f \cdot g$ in two restricted forms: Firstly, we write $p \cdot f$ for the runtime f scaled by some probability $p \in [0, 1]$. Throughout this paper, we adapt the convention that $0 \cdot \infty = \infty \cdot 0 = 0$. Secondly, we write $[\varphi] \cdot f$ to specify a *conditional runtime* f that only amounts to f if the predicate φ holds and otherwise to 0. Here, the *Iverson bracket* $[\varphi]$ (defined as $[\varphi]\,(\mathfrak{s}, \mathfrak{h}) = 1$ if $(\mathfrak{s}, \mathfrak{h}) \models \varphi$ and $[\varphi]\,(\mathfrak{s}, \mathfrak{h}) = 0$ otherwise) acts as a logical guard that "activates" the runtime f if and only if φ holds. Conditional runtimes $[\varphi] \cdot f$ can also be expressed with gatekeeper brackets, since $[\varphi] \cdot f = \llbracket \neg\varphi \rrbracket \sqcap f = f \dot{-} \llbracket \varphi \rrbracket$.

3.6 Separating Connectives on Runtimes

To enable *local reasoning* about expected runtimes of randomized *heap-manipulating* programs, we derive quantitative versions of separation logic's characteristic connectives — the separating conjunction $\varphi \star \psi$ and the separating implication $\varphi \mathrel{-\!\!\star} \psi$. We will obtain them from our runtime analogues for conjunction and implication, namely addition $f + g$ and monus $g \dot{-} f$.

3.6.1 Separating Addition. Recall from [Ishtiaq and O'Hearn 2001] that

$$(\mathfrak{s}, \mathfrak{h}) \ \models \ \varphi \star \psi \qquad \text{iff} \qquad \exists\, \mathfrak{h}_1, \mathfrak{h}_2 \text{ with } \mathfrak{h} \ = \ \mathfrak{h}_1 \star \mathfrak{h}_2 \colon \quad (\mathfrak{s}, \mathfrak{h}_1) \ \models \ \varphi \ \text{ and } \ (\mathfrak{s}, \mathfrak{h}_2) \ \models \ \psi \,,$$

i.e. the *separating conjunction* $\varphi \star \psi$ is true for a state $(\mathfrak{s}, \mathfrak{h})$ if, among all partitionings of the heap $\mathfrak{h}$ into $\mathfrak{h}_1$ and $\mathfrak{h}_2$, there exists one such that φ is true for $(\mathfrak{s}, \mathfrak{h}_1)$ *and* ψ is true for $(\mathfrak{s}, \mathfrak{h}_2)$. Notice that the "and" aggregates *undesirableness* (falsehood, cf. Section 3.5.1), whereas the $\exists$ quantifier, by choosing a heap partitioning, optimizes globally for the *most desirable* outcome (truth).

Towards connecting runtimes f and g in a similar fashion aggregating as little undesirableness as possible, it is natural to replace the falsehood aggregator "and" by its runtime analogue +. As for the $\exists$ quantifier which governs the choice of partitioning, it is natural to replace this with a min, so that we aggregate as little runtime as possible. This leads us to the following definition:

Definition 3.2 (Quantitative Separating Addition [Haslbeck 2021; Matheja 2020]*).* The *quantitative separating addition* $f \oplus g$ of two runtimes $f, g \in \mathbb{T}$ is defined as

$$f \oplus g \quad = \quad \lambda(\mathfrak{s}, \mathfrak{h}).\ \min_{\mathfrak{h}_1, \mathfrak{h}_2} \{\, f(\mathfrak{s}, \mathfrak{h}_1) + g(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \star \mathfrak{h}_2 \,\}\ . \qquad \triangle$$

Example 3.3. We typically use $f \oplus g$ to cut off parts of the heap (specified by f) before evaluating the runtime in g. For example, to evaluate $\wr 7 \mapsto 3 \wr \oplus$ size, we first attempt to cut off the single memory location 7 such that $\wr 7 \mapsto 3 \wr$ evaluates to 0 and then measure the number of locations in the remaining heap. As a truly quantitative example, size $\oplus$ size = size. $\triangle$

3.6.2 Separating Monus. Recall from [Ishtiaq and O'Hearn 2001] that

$$(\mathfrak{s}, \mathfrak{h}) \ \models\ \varphi \mathrel{-\!\!*} \psi \qquad \text{iff} \qquad \forall\, \mathfrak{h}' \text{ with } \mathfrak{h}' \perp \mathfrak{h}\colon \quad (\mathfrak{s}, \mathfrak{h}') \models \varphi \ \text{ implies } \ (\mathfrak{s}, \mathfrak{h} \star \mathfrak{h}') \models \psi,$$

i.e. the state $(\mathfrak{s}, \mathfrak{h})$ satisfies the *separating implication* $\varphi \mathrel{-\!\!*} \psi$ iff for every well-defined heap extension $\mathfrak{h}'$ of $\mathfrak{h}$ (i.e. $\mathfrak{h}' \perp \mathfrak{h}$) specified by φ (i.e. $(\mathfrak{s}, \mathfrak{h}') \models \varphi$), the combined state $(\mathfrak{s}, \mathfrak{h} \star \mathfrak{h}')$ satisfies ψ. In other words, ψ must hold for the worst (for satisfying ψ) heap extensions admitted by φ.

As we saw in Section 3.5.2, the "implies" carves out the *undesirableness* (falsehood) of its left operand from its right one, whereas the $\forall$ quantifier, by considering *every* heap extension, optimizes globally for the *least desirable* outcome (falsehood). Towards connecting runtimes f and g in a similar fashion carving out as little undesirableness as possible, it is natural to replace "implies" by its runtime analogue $\dot{-}$. As for the $\forall$ quantifier which optimizes for falsehood, it is natural to replace this with a quantitative analogue that also optimizes for most undesirable: sup,[7] so that we aggregate as much runtime as possible. This leads us to the following definition:

Definition 3.4 (Quantitative Separating Monus [Haslbeck 2021]*).* The *quantitative separating monus* $f \mathrel{-\!\!\ominus} g$ of two runtimes $f, g \in \mathbb{T}$ is defined as

$$f \mathrel{-\!\!\ominus} g \quad = \quad \lambda(\mathfrak{s}, \mathfrak{h}).\ \sup_{\mathfrak{h}'} \{\, g(\mathfrak{s}, \mathfrak{h} \star \mathfrak{h}') \mathbin{\dot{-}} f(\mathfrak{s}, \mathfrak{h}') \mid \mathfrak{h}' \perp \mathfrak{h} \,\}\ ,$$

where $\infty \mathbin{\dot{-}} \infty = 0$. $\triangle$

Example 3.5. We typically use $f \mathrel{-\!\!\ominus} g$ to extend the heap before evaluating g on the extended heap. For example, to evaluate $\wr 7 \mapsto 3 \wr \mathrel{-\!\!\ominus}$ size, we first extend the heap $\mathfrak{h}$ by $\{7 \mapsto 3\}$ and then count the number of allocated locations. If location 7 is not allocated in $\mathfrak{h}$, the result is $|\mathrm{dom}\,(\mathfrak{h})| + 1$; otherwise, $\wr 7 \mapsto 3 \wr$ is ∞ for every heap extension and the overall result is "something" $\dot{-}\, \infty = 0$.$\triangle$

3.6.3 Properties of $\oplus$ and $\mathrel{-\!\!\ominus}$. Haslbeck [2021, Chapter 4] showed that the separating addition $\oplus$ and the separating monus $\mathrel{-\!\!\ominus}$ enjoy most desirable properties of the classical separating connectives collected by Reynolds [2002]. In particular, $\oplus$ and $\mathrel{-\!\!\ominus}$ are *adjoint*, i.e. for all runtimes $f, g, u \in \mathbb{T}$,

$$u \ \preceq\ f \oplus g \qquad \text{iff} \qquad g \mathrel{-\!\!\ominus} u \ \preceq\ f\ .$$

Adjointness immediately yields the *modus ponens* property: subtracting and adding the same runtime f from and to a runtime g overapproximates g, i.e. $g \ \preceq\ f \oplus (f \mathrel{-\!\!\ominus} g)$. Moreover, $(\mathbb{T}, \oplus, \wr \mathbf{emp} \wr)$ forms a commutative monoid, i.e. $\oplus$ is associative ($f \oplus (g \oplus u) = (f \oplus g) \oplus u$), $\wr \mathbf{emp} \wr$ is the neutral element ($f \oplus \wr \mathbf{emp} \wr = \wr \mathbf{emp} \wr \oplus f = f$), and $\oplus$ is commutative ($f \oplus g = g \oplus f$). Many properties of standard addition naturally carry over to the separating addition. For example, $\oplus$ is monotone and

[7] Not a max as there are potentially infinitely many extensions $\mathfrak{h}'$.

multiplication distributes over $\oplus$, i.e. $p \cdot (f \oplus g) = p \cdot f \oplus p \cdot g$. Furthermore, standard addition and separating addition are sub-distributive: $f + (g \oplus u) \;\geq\; (f \oplus g) \;+\; (f \oplus u)$.

3.7 Runtime Specifications

Pure runtimes. A runtime f is *pure* if it does not depend on the heap, i.e. $f(\mathfrak{s}, \mathfrak{h}) = f(\mathfrak{s}, \mathfrak{h}')$ for all $\mathfrak{h}, \mathfrak{h}' \in$ Heaps. Examples of pure runtimes include 5, $x + y$, and $[\![x = y]\!]$, but not size or $[\![\mathbf{emp}]\!]$.

In a separating addition $f \oplus g$ where f is pure, heap portions that would increase g will always be evaluated in f (thus not at all), since $\oplus$ tries to minimize the overall runtime. In particular, we have $f \preceq f \oplus g$ and $0 \oplus g \preceq g$, where $0 \oplus g$ is the runtime analogue to the smallest intuitionistic extension true $\star\, \varphi$ of predicate φ (cf. [Reynolds 2002]). Notice that $f \preceq f \oplus g$ does not hold for arbitrary f: E. g., size $\not\preceq$ size $\oplus\, [\![x \mapsto -]\!]$, since the LHS can become 1 unit bigger than the RHS.

To explicitly prohibit such effects, we denote by $\underline{f}$ the runtime f (f arbitrary) that is required to be evaluated in the empty heap (otherwise it is ∞), i.e. we require f *and* the empty heap:

$$\underline{f} \;\coloneqq\; f \;+\; [\![\mathbf{emp}]\!]\,.$$

If, additionally, f is pure, then $f + g = \underline{f} \oplus g$ holds for all runtimes g.

Example 3.6. Consider the runtime f given by

$$f \quad = \quad [\![a \mapsto b]\!] \oplus [\![b \mapsto c]\!] \oplus [\![c \mapsto d]\!] \oplus \underline{[\![a = d]\!]} \;\oplus\; \underline{42}\,.$$

It evaluates to 42 for every state whose heap contains a circle $a \mapsto b \mapsto c \mapsto a$ and nothing else; otherwise, it evaluates to ∞. We can think of f as having two components: the gatekeeper brackets impose safety constraints to avoid undefined behavior (which would lead to ∞) and $\underline{42}$ represents the time units consumed if all safety constraints are met. △

Quantifiers. In the Boolean case, $\exists$ optimizes for the most desirable (truth), whereas $\forall$ optimizes for the least desirable (falsehood). In RSL, smaller runtimes are more desirable than larger ones. The RSL analogue to $\exists x\colon \varphi$ is thus an *infimum*, denoted by $\mathcal{C}\, x\colon f$, which picks a value for x to minimize runtime f. The RSL analogue to $\forall x\colon \varphi$ is a *supremum*, denoted by $\mathcal{D}\, x\colon f$, which picks a for x to maximize runtime f. To formally define our runtime quantifiers $\mathcal{C}$ and $\mathcal{D}$, we denote by

$$f\,[x/e] \quad = \quad \lambda(\mathfrak{s}, \mathfrak{h}).\; f(\mathfrak{s}\,[x \leftarrow \mathfrak{s}(e)]\,, \mathfrak{h})$$

the "syntactic" replacement of every "occurrence" of variable x in f by expression e. We then define

$$\begin{aligned} \mathcal{C}\, x\colon f \quad &= \quad \lambda(\mathfrak{s}, \mathfrak{h}).\; \inf\{\, f\,[x/v]\,(\mathfrak{s}, \mathfrak{h}) \mid v \in \mathsf{Vals}\,\} \qquad \text{and} \\ \mathcal{D}\, x\colon f \quad &= \quad \lambda(\mathfrak{s}, \mathfrak{h}).\; \sup\{\, f\,[x/v]\,(\mathfrak{s}, \mathfrak{h}) \mid v \in \mathsf{Vals}\,\}\,. \end{aligned}$$

Further details on these quantifiers are found in [Batz et al. 2021b, 2019]. To specify runtimes over data structures of arbitrary sizes, we also define runtime variants of *iterating separating conjunctions* and *inductive predicate definitions* (cf. [Reynolds 2002]).

Separating sums. To specify runtimes evaluated in variably-sized contiguous memory blocks, we use the iterative separating addition, called *separating sum* for short, given by

$$\bigoplus_{i=e}^{e'} f \quad = \quad \lambda(\mathfrak{s}, \mathfrak{h}).\; \begin{cases} \left(f\,[i/\mathfrak{s}(e)] \oplus \bigoplus_{i=e+1}^{e'} f\right)(\mathfrak{s}, \mathfrak{h}) & \text{if } \mathfrak{s}(e) \leq \mathfrak{s}(e') \\ [\![\mathbf{emp}]\!](\mathfrak{s}, \mathfrak{h}) & \text{if } \mathfrak{s}(e) > \mathfrak{s}(e')\,, \end{cases}$$

where $f \in \mathbb{T}$ and e, e' are arithmetic expressions evaluating to values in $\mathbb{N}$.

Table 2. Rules for the ert–transformer. Here v is a fresh variable not occuring in e or f.

C	$\mathsf{ert}\,[\![C]\!]\,(f)$
$\mathtt{tick}\,(e)$	$e + f$ or equivalently $\underline{e} \oplus f$
$x := e$	$f\,[x/e]$
$x := \mathtt{alloc}\,(e)$	$\mathfrak{Q}\, v\colon \left(\bigoplus_{i=1}^{e} \wr v+i-1 \mapsto 0\wr\right) \multimap f\,[x/v]$
$x := \langle e \rangle$	$\mathfrak{L}\, v\colon \wr e \mapsto v\wr \oplus \left(\wr e \mapsto v\wr \multimap f\,[x/v]\right)$
$\langle e \rangle := e'$	$\wr e \mapsto\ -\wr \oplus \left(\wr e \mapsto e'\wr \multimap f\right)$
$\mathtt{free}(e)$	$\wr e \mapsto\ -\wr \oplus f$
$\{\,C_1\,\}\ [p]\ \{\,C_2\,\}$	$p \cdot \mathsf{ert}\,[\![C_1]\!]\,(f)\ +\ (1-p)\cdot \mathsf{ert}\,[\![C_2]\!]\,(f)$
$\mathtt{if}\ (\varphi)\ \{\,C_1\,\}\ \mathtt{else}\ \{\,C_2\,\}$	$[\varphi]\cdot \mathsf{ert}\,[\![C_1]\!]\,(f)\ +\ [\neg\varphi]\cdot \mathsf{ert}\,[\![C_2]\!]\,(f)$
$C_1 \fatsemi C_2$	$\mathsf{ert}\,[\![C_1]\!]\,(\mathsf{ert}\,[\![C_2]\!]\,(f))$
$\mathtt{while}\ (\varphi)\,\{\,C'\,\}$	$\mathsf{lfp}\ g.\ \ [\neg\varphi]\cdot f\ +\ [\varphi]\cdot \mathsf{ert}\,[\![C']\!]\,(g)$

Example 3.7. Consider the following runtime f over the variables x, y, and i:

$$f \quad = \quad \bigoplus_{i=1}^{y} \Big(\mathfrak{L}\, z\colon \wr x+i-1 \mapsto z\wr \ \oplus\ \underline{(z\cdot z)} \Big)\ .$$

f *specifies* — read: *is not* ∞ *iff* — that the heap is an array of length y and evaluates to the sum of squares of the values stored in the array. The $\wr x+i-1 \mapsto z\wr$'s ensure the array structure and we use RSL's $\mathfrak{L}$ quantifier to ("existentially") refer to each location $x+i-1$'s content z. △

Coinductive runtime definitions. We specify runtimes of linked data structures using coinductive definitions, i.e. *greatest* fixed points gfp Ψ of recursive runtime equations of the form

$$f \ = \ \Psi(f) \qquad \text{where} \quad \Psi\colon \mathbb{T} \to \mathbb{T} \text{ monotone}\ .$$

Given a coinductive definition $f = \Psi(f)$, we just write f to refer to its solution gfp Ψ. For example, a runtime list $(e,\, e')$ specifiying that the heap is a singly-linked list segment from e to e' is given by

$$\mathsf{list}\,(e,\, e') \quad = \quad \underbrace{\wr e = e'\wr}_{\text{empty list}} \quad \underbrace{\sqcap}_{\substack{\text{"or"}\\ \text{cf. Section 3.5.3}}} \quad \underbrace{\mathfrak{L}\, z\colon \wr e \mapsto z\wr \oplus \mathsf{list}\,(z,\, e')}_{\text{lists of length} \geq 1}\ .$$

In words, a list specifies either the empty list such that $e = e'$ *or* a non-empty list in which e points to some location z that is the head of a list segment to e'. We can easily extend this definition to obtain the *size* of the list from e to e', or ∞ if the heap is not such a list by

$$\mathsf{listsize}\,(e,\, e') \quad = \quad \underline{\wr e = e'\wr} \quad \sqcap \quad (\underline{1} \oplus \mathfrak{L}\, z\colon \wr e \mapsto z\wr \oplus \mathsf{listsize}\,(z,\, e'))\ .$$

4 THE EXPECTED RUNTIME CALCULUS FOR hpGCL

We now extend the expected runtime calculus of Kaminski et al. [2018] by RSL, thus enabling capabilities for local reasoning about expected runtimes of programs that access and mutate dynamic memory. This is inspired by the quantitative separation logic of Batz et al. [2019].

The backward-moving *expected runtime transformer*

$$\mathsf{ert}\colon \mathsf{hpGCL} \to (\mathbb{T} \to \mathbb{T})$$

is defined by induction on hpGCL in Table 2. The transformer is defined in such a way that

$$\mathsf{ert}\ [\![C]\!]\ (0)\ (\mathfrak{s},\mathfrak{h}) \;=\; \begin{cases} \text{“expected number of ticks} \\ \quad \text{when executing } C \text{ on } (\mathfrak{s},\mathfrak{h})\text{”}, & \text{if } C \text{ memory-safe on } (\mathfrak{s},\mathfrak{h}) \\ \infty, & \text{if } C \text{ not memory-safe on } (\mathfrak{s},\mathfrak{h}). \end{cases}$$

— a fact we will make formally precise w.r.t. our operational MDP semantics in Section 4.4. More generally, to achieve compositionality, $\mathsf{ert}\ [\![C]\!]\ (f)\ (\mathfrak{s},\mathfrak{h})$ is a runtime that gives us the expected number of ticks it takes to first execute the program C on $(\mathfrak{s},\mathfrak{h})$ and then let time f pass, or ∞ in case C is not memory safe on $(\mathfrak{s},\mathfrak{h})$. We refer to f as the *postruntime*. Let us go over the rules.

Time consumption. How long does it take to execute $\texttt{tick}\,(e)$ and then let time f pass? $e + f$. Since e is always pure, this is equivalent to $\underline{e} \oplus f$, which can be more handy for local reasoning.

Assignment, Sequential composition, Conditional and probabilistic choice, While loop. All these cases have been treated in detail in [Kaminski 2019, Section 7.3, p.163–166]. The only difference is that in [Kaminski 2019] each basic instruction consumes 1 unit of time whereas we have here (by means of `tick`) a more fine-grained runtime model.

In order to be somewhat self-contained, however, let us quickly go over the case for assignments and probabilistic choice: How long does it take to execute $x := e$ (on initial state $(\mathfrak{s},\mathfrak{h})$) and then let time f pass? Executing $x := e$ itself takes no time. But we need to evaluate f in the state that is reached *after* the assignment, i.e. the final state $(\mathfrak{s}\,[x \leftarrow \mathfrak{s}(e)]\,,\mathfrak{h})$. This is precisely $f\,[x/e]$ but evaluated in the initial state $(\mathfrak{s},\mathfrak{h})$. For the probabilistic choice $\{\,C_1\,\}\ [p]\ \{\,C_2\,\}$, we simply take the weighted average of the expected time it takes to either execute C_1 or C_2 and then let time f pass.

Allocation. Again, $x := \texttt{alloc}\,(e)$ itself takes no time, but we need to measure f in a state where the heap has been extended by e contiguous memory locations, all initialized to store value 0. $\bigoplus_{i=1}^{e} [\![v+i-1 \mapsto 0]\!]$ describes precisely such an extension. By $\multimap$ we impose this extension on f. What is left is to handle the nondeterminism arising from the memory allocator's choice of the first new location v. As we do upper-bound (worst-case) reasoning, we resolve this nondeterminism via a maximizing $\mathfrak{D}$ and measure f in the extended heap and in a stack where x has been updated to v.

Lookup. For $x := \langle e \rangle$, we first ensure via $[\![e \mapsto v]\!] \oplus \_\_$ that e is indeed allocated. (If not, the whole term becomes ∞, indicating a memory fault.) In connection with the $\mathfrak{L}$ quantifier, we moreover select the value v that e points to. The $\oplus$ has now carved the memory location $\{e \mapsto v\}$ out from the heap. Since we did not want to manipulate the heap, we reinsert $\{e \mapsto v\}$ via $[\![e \mapsto v]\!] \multimap \_\_$. What is left is to measure f but in a stack where x has been updated to v.

Mutation. For $\langle e \rangle := e'$, we first ensure via $[\![e \mapsto -]\!] \oplus \_\_$ that e is actually allocated, but care not about its stored value since we are about to overwrite it. The $\oplus$ has now carved location e out from the heap. In order to overwrite the value stored at e with e', we insert $\{e \mapsto e'\}$ into the heap via $[\![e \mapsto e']\!] \multimap \_\_$. This insertion of location e cannot fail (become ∞) because we have previously carved out precisely location e (unless e was not allocated in the first place, in which case the whole term becomes ∞ anyway). What is left is to measure f in the so-manipulated heap.

Deallocation. For $\texttt{free}(e)$, we need to measure f in a heap where the location e has been carved out. As demonstrated numerous times previously, such carving out is achieved by $[\![e \mapsto -]\!] \multimap \_\_$.

Theorem 4.1 (Healthiness Properties of ert). *Let C be a program; $F = \{f_1 \preceq f_2 \preceq \ldots\}$ be an ω-chain of runtimes; f, g be runtimes; and $u \in \mathbb{T}$ be a constant runtime. Then the following hold:*

(1) ω-continuity: $\quad \mathsf{ert}\,[\![C]\!]\,(\sup F) \;=\; \sup \mathsf{ert}\,[\![C]\!]\,(F)$

(2) Monotonicity: $\quad f \preceq g \quad \text{implies} \quad \mathsf{ert}\,[\![C]\!]\,(f) \preceq \mathsf{ert}\,[\![C]\!]\,(g)$

(3) Sub-additivity: $\quad \mathsf{ert}\,[\![C]\!]\,(f+g) \preceq \mathsf{ert}\,[\![C]\!]\,(f) + \mathsf{ert}\,[\![C]\!]\,(g)$

(4) Constant propagation: $\quad \mathsf{ert}\,[\![C]\!]\,(u+f) \preceq u + \mathsf{ert}\,[\![C]\!]\,(f)$

Remark 4.2 ((Non-)ω-continuity of ert*).* ω-continuity is actually somewhat *unexpected*. The weakest preexpectation transformer of Batz et al. [2019] who deals with determining (minimal) expected values of essentially the same hpGCL is indeed *not* ω-continuous due to the unbounded nondeterminism arising from memory allocation. Probably guided by this result, Haslbeck [2021, Section 4.2, p.46] *claims* that the ert transformer is also *not* ω-continuous. However, since ert is a *maximizer* and suprema commute, ert *does* enjoy the beneficial property of ω-continuity.

4.1 Local Reasoning for Expected Runtimes

To enable local reasoning, the ert calculus features a *frame rule for establishing upper bounds*:

Theorem 4.3 (Frame Rule for RSL Haslbeck [2021]). *For every $C \in$ hpGCL and runtimes f, g,*

$$\mathsf{Mod}(C) \cap \mathsf{Vars}(g) \;=\; \emptyset \qquad \text{implies} \qquad \mathsf{ert}\,[\![C]\!]\,(f \oplus g) \;\preceq\; \mathsf{ert}\,[\![C]\!]\,(f) \;\oplus\; g\,.$$

We call g in the above theorem the *frame*. Combining the above frame rule with the RSL analogues of Reynolds [2002]'s local rules for heap mutation, lookup, memory allocation, and auxiliary variable elimination enables SL-style source-code level proofs for upper-bounding expected runtimes:

Theorem 4.4 (Local Rules for RSL following Reynolds [2002]). *Let $C \in$ hpGCL. Then:*

(1) (mut): $\quad \mathsf{ert}\,[\![\langle e\rangle := e']\!]\,\big(\llbracket e \mapsto e' \rrbracket\big) \;\preceq\; \llbracket e \mapsto - \rrbracket$

(2) (lkp): $\quad \mathsf{ert}\,[\![x := \langle e\rangle]\!]\,\Big(\underline{\llbracket x = z \rrbracket} \oplus \llbracket e\,[x/y] \mapsto z \rrbracket\Big) \;\preceq\; \underline{\llbracket x = y \rrbracket} \oplus \llbracket e \mapsto z \rrbracket$,

(3) (alc): *if x does not occur in e, then*

$$\mathsf{ert}\,[\![x := \mathtt{alloc}(e)]\!]\left(\bigoplus_{i=1}^{e} \llbracket x + i - 1 \mapsto 0 \rrbracket\right) \;\preceq\; \llbracket \mathbf{emp} \rrbracket\,.$$

(4) (aux): *For all $f, g \in \mathbb{T}$ and all $y \in \mathsf{Vars}$ not occurring in C,*

$$\mathsf{ert}\,[\![C]\!]\,(f) \;\preceq\; g \qquad \text{implies} \qquad \mathsf{ert}\,[\![C]\!]\,\big(\lightning y\colon f\big) \;\preceq\; \lightning y\colon g\,.$$

4.2 Invariant-Based Reasoning for Loops

Recall that $\mathsf{ert}\,[\![\mathit{loop}]\!]\,(f)$ for $\mathit{loop} = \mathtt{while}\ (\varphi)\ \{\ \mathit{body}\ \}$ is defined as

$$\mathsf{lfp}\ g.\ \underbrace{[\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{ert}\,[\![\mathit{body}]\!]\,(g)}_{:=\Phi_f(g)}\,,$$

where we call Φ_f the ert-*characteristic functional* of *loop* w.r.t. postruntime f. For upper-bounding expected runtimes of loops — given as least fixed points — we have an invariant-based proof rule:

Theorem 4.5 (Park Induction for RSL). *Let $\mathit{loop} = \mathtt{while}\ (\varphi)\ \{\ \mathit{body}\ \}$ and $I, f \in \mathbb{T}$. Then*

$$\Phi_f(I) \;\preceq\; I \quad \text{implies} \quad \mathsf{ert}\,[\![\mathit{loop}]\!]\,(f) \;\preceq\; I\,.$$

We call such I an ert-*invariant*. Park induction can simplify the verification of loop runtimes significantly: to obtain an upper bound on the expected runtime of the *entire* loop, it essentially suffices to upper-bound $\mathsf{ert}\,[\![\mathit{body}]\!]\,(I)$, i.e. the expected runtime of *one* (arbitrary) loop *iteration*. Notice that the above proof rule is *complete* since $\mathsf{lfp}\ g.\ \Phi_f(g)$ is necessarily an ert-invariant.

4.3 Example: The Lagging List Traversal

We demonstrate the applicability of the ert calculus. Consider the program C:

$$\texttt{while}\ (y \neq 0)\ \{\ \texttt{tick}\,(1)\ \mathbin{⨟}\ \{\ y := \langle y \rangle\ \}\ [1/2]\ \{\ \texttt{skip}\ \}\ \}$$

C traverses a null-terminated list segment beginning at y. Every iteration costs 1 unit of time. The program then either traverses the next edge of the list (left branch), or forgets to do so (right branch), each with probability $1/2$. Using the frame rule, the local rules for lookup and auxiliary variable elimination, and monotonicity of ert we prove that

$$I \;\coloneqq\; 2 \cdot \mathcal{L}\, H\colon \mathsf{list}\,(H,\, y) \oplus \mathsf{listsize}\,(y,\, 0)$$

is an ert-invariant of C w.r.t. postruntime 0, which, by Theorem 4.5, implies that I upper-bounds the expected runtime of C, where $\mathsf{list}\,(H,\, y)$ is only needed for strengthening the loop invariant. Hence, when executed on a heap consisting of a null-terminated list containing the element y, the program C is *memory-safe* and takes, in expectation, a runtime of at most 2 times the size of the list segment beginning at y.

4.4 Soundness of the ert Calculus

We will now show that the ert calculus is sound in that it characterizes a program's expected execution time defined in terms of expected rewards of operational MDPs in Section 2.3. Formally, we show that, for every hpGCL program C and initial state $(\mathfrak{s}, \mathfrak{h})$, we have

$$\mathsf{ert}\ [\![C]\!]\ (0)\ (\mathfrak{s}, \mathfrak{h}) \quad = \quad \mathsf{ExpRew}\left(\mathcal{M}[\![C, \mathfrak{s}, \mathfrak{h}]\!]\right)\ ,$$

where the post-runtime 0 indicates that no time is consumed after termination of C.

We will prove a more general claim for arbitrary post-runtimes $f \in \mathbb{T}$ instead of the fixed post-runtime 0. To account for f in our operational semantics, we first extend the reward function *rew* of our operational MDPs $\mathcal{M}[\![C, \mathfrak{s}, \mathfrak{h}]\!]$ such that we collect a reward of $f(\mathfrak{s}', \mathfrak{h}')$ whenever C successfully terminates as indicated by an execution step from configuration $(\mathit{term},\, \mathfrak{s}',\, \mathfrak{h}')$ to sink:

$$\mathit{rew}\colon \quad \mathsf{Conf} \to \mathbb{R}^{\infty}_{\geq 0}, \qquad c \mapsto \begin{cases} \mathfrak{s}(e), & \text{if } c = (\texttt{tick}\,(e)\,,\ \mathfrak{s},\ \mathfrak{h}) \\ f(\mathfrak{s}, \mathfrak{h}), & \text{if } c = (\mathit{term},\ \mathfrak{s},\ \mathfrak{h}) \\ \infty, & \text{if } c = (\mathit{fault},\ \mathfrak{s},\ \mathfrak{h}) \\ 0, & \text{otherwise.} \end{cases}$$

We denote by $\mathcal{M}[\![C, f, \mathfrak{s}, \mathfrak{h}]\!]$ the operational MDP introduced in Section 2.3 but with the above reward function. Our expected runtime calculus is then sound in the following sense:

Theorem 4.6 (Soundness of ert). *For all $C \in$ hpGCL, runtimes $f \in \mathbb{T}$, and program states $(\mathfrak{s}, \mathfrak{h})$,*

$$\mathsf{ert}\ [\![C]\!]\ (f)\ (\mathfrak{s}, \mathfrak{h}) \quad = \quad \mathit{ExpRew}\left(\mathcal{M}[\![C, f, \mathfrak{s}, \mathfrak{h}]\!]\right)\ .$$

Our soundness theorem clarifies what the expected runtime calculus actually computes: as long as *no* program execution of C on initial state $(\mathfrak{s}, \mathfrak{h})$ leads to a memory fault, $\mathsf{ert}\ [\![C]\!]\ (0)\ (\mathfrak{s}, \mathfrak{h})$ is the expected execution time measured in units of time consumed by $\texttt{tick}\,(e)$ statements; if *some* program execution of C on initial state $(\mathfrak{s}, \mathfrak{h})$ *does* lead to a memory fault, we cannot give a finite bound on the expected runtime and have to conclude $\mathsf{ert}\ [\![C]\!]\ (0)\ (\mathfrak{s}, \mathfrak{h}) = \infty$.

Towards a proof of our soundness theorem, a *runtime transformer* is a function of the form

$$\mathsf{rt}\colon \mathsf{hpGCL} \cup \{\mathit{term}, \mathit{fault}\} \to (\mathbb{T} \to \mathbb{T})$$

that maps the terminated programs to the postruntime, i.e. $\mathsf{rt}[\![\mathit{term}]\!](f) = f$, and memory faults to an infinite runtime, i.e. $\mathsf{rt}[\![\mathit{fault}]\!](f) = \infty$.

We then lift the partial ordering $\preceq$ on runtimes in $\mathbb{T}$ to an order on runtime transformers, i.e.

$$\mathsf{rt} \preceq \mathsf{rt}' \quad \text{iff} \quad \forall C \in \mathsf{hpGCL} \cup \{\mathit{term}, \mathit{fault}\}\ \forall f \in \mathbb{T}\colon \quad \mathsf{rt}[\![C]\!](f) \preceq \mathsf{rt}'[\![C]\!](f)\ .$$

Clearly, we can naturally extend the expected runtime calculus $\mathsf{ert}\colon \mathsf{hpGCL} \to (\mathbb{T} \to \mathbb{T})$ to a runtime transformer. We denote the resulting *extended expected runtime calculus* by eert, i.e.

$$\mathsf{eert} \quad = \quad \lambda C\ \lambda f.\ \begin{cases} f, & \text{if } C = \mathit{term} \\ \infty, & \text{if } C = \mathit{fault} \\ \mathsf{ert}\ [\![C]\!]\ (f), & \text{otherwise.} \end{cases}$$

Analogously, our operational semantics induces a runtime transformer oprt that maps every program, runtime, and state to the expected reward of the corresponding operational MDP, i.e.

$$\mathsf{oprt} \quad = \quad \lambda C\ \lambda f\ \ \lambda(\mathfrak{s}, \mathfrak{h}).\ \mathsf{ExpRew}\left(\mathcal{M}[\![C, f, \mathfrak{s}, \mathfrak{h}]\!]\right)\ .$$

To prove our soundness theorem, it then suffices to show eert $\preceq$ oprt and oprt $\preceq$ eert.

We will leverage that our runtime transformers eert and oprt satisfy the well-established optimality equations for MDPs, also known as Bellman equations (cf. [Puterman 2005]). Formally, a runtime transformer rt is *Bellman compliant* if and only if for all $C \in \mathsf{hpGCL}$, $f \in \mathbb{T}$, and $(\mathfrak{s}, \mathfrak{h})$,[8]

$$\mathsf{rt}[\![C]\!](f)(\mathfrak{s}, \mathfrak{h}) \quad = \quad \mathit{rew}(C, \mathfrak{s}, \mathfrak{h}) + \sup_{a \in \mathsf{Act}} \sum_{(C,\mathfrak{s},\mathfrak{h}) \xrightarrow[p]{a} (C',\mathfrak{s}',\mathfrak{h}')} p \cdot \mathsf{rt}[\![C']\!](f)(\mathfrak{s}', \mathfrak{h}')\ .$$

In words, the runtime computed by a Bellman compliant runtime transformer is the reward collected for leaving the current configuration $(C, \mathfrak{s}, \mathfrak{h})$ plus the runtimes of all direct successor configurations $(C', \mathfrak{s}', \mathfrak{h}')$ weighted by the probability moving to configuration $(C', \mathfrak{s}', \mathfrak{h}')$; if there is a nondeterministic choice between different actions (i.e. we can choose a distribution over successor configurations), then we take one that maximizes the overall runtime.

A long established result on MDPs (even those with countable state spaces and actions) is that their expected rewards satisfy the Bellman equations [Puterman 2005, Theorem 7.1.3]. Hence:

Lemma 4.7. oprt *is Bellman compliant.*

In fact, the same holds for the (extended) expected runtime calculus:

Lemma 4.8. eert *is Bellman compliant.*

Proof. By structural induction on the rules of our execution relation (Figure 1). □

Equipped with two Bellman compliant runtime transformers, we now prove that eert $\preceq$ oprt and oprt $\preceq$ eert. The first inequality can be proven directly by structural induction:

Lemma 4.9. eert $\preceq$ oprt.

Proof. By construction, we have

$$\mathsf{eert}[\![\mathit{term}]\!](f) = f \preceq \mathsf{oprt}[\![\mathit{term}]\!](f) \quad \text{and} \quad \mathsf{eert}[\![\mathit{fault}]\!](f) = \infty \preceq \mathsf{oprt}[\![\mathit{fault}]\!](f)\ .$$

It then suffices to show that by induction on the structure of hpGCL programs that, for all $C \in \mathsf{hpGCL}$ and all $f \in \mathbb{T}$, we have $\mathsf{eert}[\![C]\!](f) \preceq \mathsf{oprt}[\![C]\!](f)$. □

We do not directly show the converse direction, i.e. oprt $\preceq$ eert, since we can invoke a more general result for MDPs that goes back to Blackwell [1967]:

[8] Textbooks typically restrict the supremum to the set of actions that are *enabled* in the given configuration. To simplify notation, we take the supremum over all actions Act = Vals and agree on the convention that $\sum_{\emptyset} \ldots = 0$.

Lemma 4.10. *For every Bellman compliant runtime transformer* rt*, we have* rt $\geq$ oprt.

Proof. Since all runtimes in $\mathbb{T}$ are non-negative and our execution relation never gets stuck, the MDPs induced by our operational semantics are positive models (cf. beginning of [Puterman 2005, Chapter 7.2]). The claim is then a special case of [Blackwell 1967, Theorem 2]. □

Notice that a variant of Blackwell's theorem is also found in the textbook of Puterman [2005, Theorem 7.2.2]. However, Puterman considers only positive *bounded* models, even though the proof of his theorem does not seem to rely on having a bounded model. Finally, we conclude:

Proof of Theorem 4.6. By Lemma 4.8, eert is a Bellman compliant runtime transformer. By Lemma 4.10, this implies oprt $\leq$ eert. Moreover, by Lemma 4.9, we have eert $\leq$ oprt. Hence, eert = oprt. Now, for any hpGCL program C, runtime $f \in \mathbb{T}$, and state $(\mathfrak{s}, \mathfrak{h})$, we have

$$\begin{aligned} \mathsf{ert}\ [\![C]\!]\ (f)\,(\mathfrak{s},\mathfrak{h}) &= \mathsf{eert}[\![C]\!](f)(\mathfrak{s},\mathfrak{h}) && (C \in \mathsf{hpGCL}\text{, by construction of eert}) \\ &= \mathsf{oprt}[\![C]\!](f)(\mathfrak{s},\mathfrak{h}) && (\text{eert} = \text{oprt as shown above}) \\ &= \mathsf{ExpRew}\left(\mathcal{M}[\![C,f,\mathfrak{s},\mathfrak{h}]\!]\right). && (\text{definition of oprt}) \quad \square \end{aligned}$$

5 THE AMORTIZED EXPECTED RUNTIME CALCULUS

In *amortized analysis*, instead of analyzing the worst-case runtime of C, we average the runtime of C over a whole sequence $C^n = C_1 \fatsemi \ldots \fatsemi C_n$ of n consecutive executions of C. One technique for amortized analysis is the *potential method*, whose core idea is to *make frequently occurring low "normal-case" runtimes of C mildly larger* and in return be able to *make the seldomly occurring worst-case runtimes a lot smaller*, thus smoothing out seldomly occurring runtime-peaks in the sequence C^n. Key ingredient to achieve such smoothing is a potential function:

Definition 5.1 (Potential Functions [Sleator and Tarjan 1985; Tarjan 1985]*).* A *potential function* is a function π of type States $\to \mathbb{R}_{\geq 0}$. Note that $\pi \in \mathbb{T}$ with $\pi \prec \infty$. △

The potential function needs to be chosen so that each time C has small runtime, the potential is mildly increased. Each time C has large runtime, on the other hand, the potential should be drastically decreased. The *amortized runtime* of C is then C's actual runtime plus *the change in potential*. Indeed, then the amortized runtime of C in the cheap case is C's small runtime plus a mildly positive change in potential — overall still a small number. The amortized runtime of the expensive case, on the other hand, is C's large runtime plus a large *negative* change in potential — overall, again (hopefully), a small number. Why did the potential do the trick?

Let us denote the runtime of executing C_i by rt_i and the potential attained afterwards by π_i. The amortized runtime of executing sequence element C_{i+1} is then $rt_{i+1} + \pi_{i+1} - \pi_i$. Summing the amortized runtimes over the whole sequence gives

$$\sum_{i=0}^{n}\left(rt_{i+1} + \pi_{i+1} - \pi_i\right) \quad = \quad \underbrace{\sum_{i=0}^{n}\left(rt_{i+1}\right)}_{\text{actual runtime of the sequence}} + \overbrace{\pi_n}^{\text{non-negative}} - \underbrace{\pi_0}_{\text{assumed to be } 0}. \qquad (\dagger)$$

It is now easy to see that if we start with initial potential $\pi_0 = 0$, then the amortized runtime of the whole sequence *overapproximates* the actual runtime of the sequence. Indeed, we could recover the actual runtime by subtracting from the amortized runtime the (non-negative) final potential π_n.

As we are concerned with *expected* (amortized) runtimes of randomized algorithms, we would need to take expected changes in potential into account and telescoping is not as obvious anymore. Moreover, changes in potential may become negative. In fact: they should! Otherwise, we have no

Table 3. Rules for the aert–transformer w.r.t. potential π.

C	$\mathbf{aert}_\pi\,[\![C]\!]\,(X)$
$\texttt{tick}\,(e)$	$e + X$
C is atomic and not $\texttt{tick}$	$\mathsf{ert}\,[\![C]\!]\,(X+\pi) - \pi$
$C_1 \fatsemi C_2$	$\mathsf{aert}_\pi\,[\![C_1]\!]\,(\mathsf{aert}_\pi\,[\![C_2]\!]\,(X))$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{aert}_\pi\,[\![C_1]\!]\,(X) \;+\; [\neg\varphi]\cdot \mathsf{aert}_\pi\,[\![C_2]\!]\,(X)$
$\{C_1\}\,[p]\,\{C_2\}$	$p\cdot \mathsf{aert}_\pi\,[\![C_1]\!]\,(X) \;+\; (1-p)\cdot \mathsf{aert}_\pi\,[\![C_2]\!]\,(X)$
$\texttt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\,Y.\;\; [\neg\varphi]\cdot X \;+\; [\varphi]\cdot \mathsf{aert}_\pi\,[\![C']\!]\,(Y)$

chance of compensating for expensive operations. In the following, we present an ert-style calculus that can capture *amortized expected runtimes* and we prove that it essentially satisfies the above telescoping property, so that it also over-approximates true *expected* runtimes.

5.1 The Calculus

Let us fix a potential function π and define a set of *amortized runtimes* relative to π.

Definition 5.2 (Amortized Runtimes). The set $\mathbb{A}_\pi$ of amortized runtimes with respect to potential function π, π-runtimes for short, is defined as

$$\mathbb{A}_\pi \;=\; \{\, X\colon \mathsf{States} \to \mathbb{R}\cup\{\infty\} \mid \forall(\mathfrak{s},\mathfrak{h}) \in \mathsf{States}\colon -\pi(\mathfrak{s},\mathfrak{h}) \le X(\mathfrak{s},\mathfrak{h}) \,\}\;.$$

We denote amortized runtimes by X, Y, Z and variations. We extend $\preceq$ from $\mathbb{E}$ to $\mathbb{A}_\pi$ naturally by

$$X \;\preceq\; Y \qquad \text{iff} \qquad \text{for all } (\mathfrak{s},\mathfrak{h}) \in \mathsf{States}\colon \quad X(\mathfrak{s},\mathfrak{h}) \;\le\; Y(\mathfrak{s},\mathfrak{h})\;.$$

$(\mathbb{A}_\pi, \preceq)$ forms a complete lattice with least element $-\pi$. △

The backward-moving *amortized expected runtime transformer*

$$\mathsf{aert}_\pi\colon \mathsf{hpGCL} \to (\mathbb{A}_\pi \to \mathbb{A}_\pi)$$

is defined by induction on hpGCL in Table 3 and manipulates π-runtimes, which can in principle become negative (as negative as $-\pi$), instead of ordinary runtimes. Like ert, the aert transformer is defined in such a way that

$$\mathsf{aert}_\pi\,[\![C]\!]\,(0)\,(\mathfrak{s},\mathfrak{h}) \;=\; \infty \quad \text{if} \quad C \text{ is not memory-safe on } (\mathfrak{s},\mathfrak{h})\;.$$

Let us briefly go over the rules defining aert.

Time consumption. Executing $\texttt{tick}\,(e)$ and then letting (amortized) time X pass takes $e + X$ units of time and cannot change the potential. We cannot go for $\underline{e} \oplus X$ because $\oplus$ is undefined on $\mathbb{A}_\pi$.

All other atoms. We have $\mathsf{aert}_\pi\,[\![\mathit{atom}]\!]\,(X) = \mathsf{ert}\,[\![\mathit{atom}]\!]\,(X+\pi) - \pi$. E. g., for $x := e$, this gives $X\,[x/e] + \pi\,[x/e] - \pi$. Here, we can see how the change in potential is propagated through the program at the level of atoms.

Composite constructs. Defined like ert, but we need to compose the terms of aert components.

Theorem 5.3. aert_π *is ω-continuous, i.e. for all $C \in$ hpGCL and ω-chains $F = \{X_1 \preceq X_2 \preceq \ldots\}$,*

$$\mathsf{aert}_\pi \llbracket C \rrbracket (\sup F) \;=\; \sup \mathsf{aert}_\pi \llbracket C \rrbracket (F) \;.$$

Continuity of the aert transformer follows from continuity of the ert transformer and the following central theorem, which formalizes the telescoping principle of Equation (†) for *expected* runtimes:

Theorem 5.4 (Telescoping for aert). *For all $C \in$ hpGCL and $X \in \mathbb{A}_\pi$, we have*

$$\mathsf{aert}_\pi \llbracket C \rrbracket (X) \;=\; \mathsf{ert} \llbracket C \rrbracket (X + \pi) - \pi \;.$$

In some sense, the above theorem tells us that the following *almost* (modulo some technical particularities of the `alloc` statement which we omit here) holds:

$$\mathsf{aert}_\pi \llbracket C \rrbracket (0) \;\approx\; \mathsf{ert} \llbracket C \rrbracket (0) \;+\; \text{“expected potential after executing } C\text{”} - \pi \qquad (‡)$$

The reason is that $\mathsf{ert} \llbracket C \rrbracket (\pi)$ can (again: almost) be decomposed into $\mathsf{ert} \llbracket C \rrbracket (0)$ plus the expected value of π after executing C.

If C is an entire sequence of operations, we can now relate the right-hand-sides of (†) and (‡): the “actual runtime” corresponds to $\mathsf{ert} \llbracket C \rrbracket (0)$, the expected potential after executing C corresponds to π_n, and the initial potential π_0 corresponds to π. Again, this explanation breaks slightly for programs featuring dynamic memory allocation, but Theorem 5.4 holds also for those programs and allows us to prove soundness of the aert transformer in the following *overapproximating* sense:

Theorem 5.5 (Soundness of aert). *For every $C \in$ hpGCL, we have*

$$\mathsf{ert} \llbracket C \rrbracket (0) \;\preceq\; \mathsf{aert}_\pi \llbracket C \rrbracket (0) + \pi \;.$$

Proof. $\mathsf{ert} \llbracket C \rrbracket (\pi) - \pi \;=\; \mathsf{aert}_\pi \llbracket C \rrbracket (0)$ (by Theorem 5.4 where $X = 0$)
implies $\mathsf{ert} \llbracket C \rrbracket (0) - \pi \;\preceq\; \mathsf{aert}_\pi \llbracket C \rrbracket (0)$ ($\mathsf{ert} \llbracket C \rrbracket (0) \preceq \mathsf{ert} \llbracket C \rrbracket (\pi)$ by monotonicity of ert)
implies $\mathsf{ert} \llbracket C \rrbracket (0) \;\preceq\; \mathsf{aert}_\pi \llbracket C \rrbracket (0) + \pi$ □

A further handy decomposition of aert is so-called *constant propagation* also known from the non-RSL ert calculus of Kaminski et al. [2018]:

Theorem 5.6 (Constant Propagation for aert). *For all $C \in$ hpGCL and all constant $const \in \mathbb{T}$,*

$$\mathsf{aert}_\pi \llbracket C \rrbracket (X + const) \;\preceq\; \mathsf{aert}_\pi \llbracket C \rrbracket (X) + const \;.$$

*Example 5.7 (*aert *Reasoning).* Consider $Op = \{\, \mathtt{tick}\,(x) \,\}\ [1/2]\ \{\, x := 0 \,\} \fatsemi x := x + 1$. This operation either consumes x units of time or resets x to 0, each with probability $1/2$. Furthermore, each invocation of Op increases x. Let us now perform both aert as well as ert analyses.

For aert, we choose as potential function $\pi = x$. We will then make program annotations as shown in Figure 2. The left one of these annotations are the aert annotations and they can be read (best from bottom to top) as follows: 0 is the postruntime. 1 is the result (after simplifications) of $\mathsf{aert}_\pi \llbracket x := x + 1 \rrbracket (0) = 0 + x\,[x/x+1] - x = 1$. The resulting 1 is also the postruntime to consider for both branches of the probabilistic choice. $1 - x$ is the result (again, after simplification) of $\mathsf{aert}_\pi \llbracket x := 0 \rrbracket (1)$. Likewise for the other branch of the probabilistic choice. Finally, at the very top, 1 is the result of combining (and simplifying) the two outcomes of the branches of the probabilistic choice according to the rule for aert — in this case: $1 = \frac{1}{2} \cdot (x + 1) + \frac{1}{2} \cdot (1 - x)$. The same annotation style applies to the ert annotations on the far right.

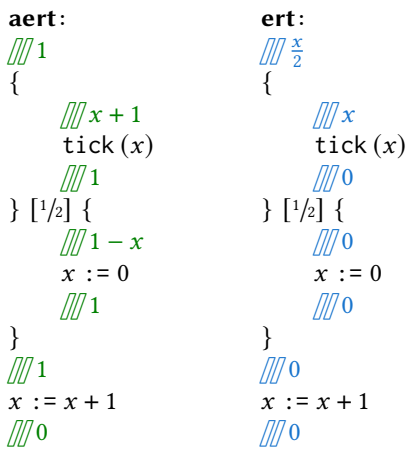

Fig. 2. aert reasoning (left) and ert reasoning (right).

We can read off that the amortized expected time to perform a single Op is 1. Moreover, Theorem 5.6 immediately yields $\mathsf{aert}_\pi\,[\![Op]\!]\,(const) = const + 1$ and from there it is easy to prove by induction that $\mathsf{aert}_\pi\,[\![Op^n]\!]\,(0) = n$. From there, in turn, we obtain by Theorem 5.5 that

$$\mathsf{ert}\,[\![Op^n]\!]\,(0) \;\preceq\; \mathsf{aert}_\pi\,[\![Op^n]\!]\,(0) + x \;=\; n + x\,.$$

This was relatively easy and gives us a very clear upper bound on the expected time it takes to execute a sequence of n Op's, namely $n + x$ which is n if potential x was 0 initially.

Obtaining the same insight solely via ert reasoning would have been harder. First, we can read off that $\mathsf{ert}\,[\![Op]\!]\,(0) = \frac{x}{2}$. But what about $\mathsf{ert}\,[\![Op^n]\!]\,(0)$? For $n = 2$, we get $\mathsf{ert}\,[\![Op\,\mathbin{;}\,Op]\!]\,(0) = \frac{3x}{4} + \frac{1}{4}$. For $n = 3$, we get $\frac{7x}{8} + \frac{5}{8}$. For $n = 4$, we get $\frac{15x}{16} + \frac{17}{16}$. Seeing a pattern here is less easy than it was for aert, especially for the constant part of the term.

5.2 Local Reasoning for Amortized Expected Runtimes

To enable local reasoning, the aert calculus also features a *frame rule for upper bounds*:

Theorem 5.8 (Frame Rule for aert). *Let $C \in \mathsf{hpGCL}$ and $f, g \in \mathbb{T}$. Then*

$$\mathsf{Mod}\,(C) \cap \mathsf{Vars}(g) = \emptyset \qquad \text{implies}$$
$$\mathsf{aert}_\pi\,[\![C]\!]\,((f \oplus g) - \pi) \;\preceq\; \big((\mathsf{aert}_\pi\,[\![C]\!]\,(f - \pi) + \pi) \oplus g\big) - \pi\,.$$

Proof.
$$\begin{aligned}
\mathsf{aert}_\pi\,[\![C]\!]\,((f \oplus g) - \pi) &= \mathsf{ert}\,[\![C]\!]\,(f \oplus g) - \pi && \text{(by Theorem 5.4)}\\
&\preceq (\mathsf{ert}\,[\![C]\!]\,(f) \oplus g) - \pi && \text{(by Theorem 4.3)}\\
&= \big((\mathsf{ert}\,[\![C]\!]\,(f) - \pi + \pi) \oplus g\big) - \pi\\
&= \big((\mathsf{aert}_\pi\,[\![C]\!]\,(f - \pi) + \pi) \oplus g\big) - \pi && \text{(by Theorem 5.4)}\quad \square
\end{aligned}$$

While this rule may look quite involved, it is still helpful: (1) postruntimes during reasoning are often of the form $\_\_\_ - \pi$ and (2) the heavy lifting, i.e. the aert-reasoning, is still done more locally, namely on $f - \pi$ instead of $(f \oplus g) - \pi$.

5.3 Compositional Reasoning about Nested Data Structures

The aert calculus is parameterized in a potential function π, which must be chosen carefully with respect to the data structure that is analyzed. On a first glance, having to fix a potential function for aert-based reasoning might hamper compositionality. For example, assume we have already analyzed the amortized expected runtime of a data structure, say D_1, using some potential function π_1. Furthermore, suppose a second data structure, say D_2, internally uses D_1 as a subcomponent, and its analysis requires a slightly different potential function, say $\pi_1 \oplus \pi_2$, to account for additional potential of other elements of D_2. Naively, we then have to analyze the aert of D_1 again, this time with respect to the extended potential $\pi_1 \oplus \pi_2$. However, this *need not be necessary*. Instead, we can re-use our existing analysis of D_1 with respect to potential π_1 to analyze the amortized expected runtime of D_2 with respect to the extended potential $\pi_1 \oplus \pi_2$.

More precisely, we can – under mild conditions for π_1 and π_2 – re-use an existing upper bound on $\mathsf{aert}_{\pi_1}[\![C]\!](X)$ to obtain an upper-bound on $\mathsf{aert}_{\pi_1 \oplus \pi_2}[\![C]\!](X)$. The following theorem leverages the frame rule for aert and monotonicity to enable such compositional reasoning:

Theorem 5.9. *Let π_1, π_2 be potentials with $\pi_1 \preceq \pi_1 \oplus \pi_2$. Then, for all $C \in \mathsf{hpGCL}$ and $X \in \mathbb{A}_{\pi_1}$,*

$$\mathsf{Mod}(C) \cap \mathsf{Vars}(\pi_2) = \emptyset \quad \textit{and} \quad X + (\pi_1 \oplus \pi_2) \preceq (X + \pi_1) \oplus \pi_2$$
$$\text{implies} \qquad \mathsf{aert}_{\pi_1 \oplus \pi_2}[\![C]\!](X) \preceq (\mathsf{aert}_{\pi_1}[\![C]\!](X) + \pi_1) \oplus \pi_2 \;-\; \pi_1 \oplus \pi_2 \,.$$

Proof. Notice that all of the above expressions are well-defined. Then, consider the following:

$$\begin{aligned}
&\mathsf{aert}_{\pi_1 \oplus \pi_2}[\![C]\!](X) \\
&= \mathsf{ert}[\![C]\!](X + (\pi_1 \oplus \pi_2)) \;-\; \pi_1 \oplus \pi_2 && \text{(by Theorem 5.4)} \\
&\preceq \mathsf{ert}[\![C]\!]((X + \pi_1) \oplus \pi_2) \;-\; \pi_1 \oplus \pi_2 && \text{(by monotonicity of ert (Theorem 4.1) and assumption)} \\
&\preceq \mathsf{ert}[\![C]\!](X + \pi_1) \oplus \pi_2 \;-\; \pi_1 \oplus \pi_2 && \text{(by frame rule for ert (Theorem 4.3))} \\
&= (\mathsf{aert}_{\pi_1}[\![C]\!](X) + \pi_1) \oplus \pi_2 \;-\; \pi_1 \oplus \pi_2 && \text{(by Theorem 5.4)} \quad \square
\end{aligned}$$

In the above proof, the assumptions $\mathsf{Mod}(C) \cap \mathsf{Vars}(\pi_2) = \emptyset$ and $X + (\pi_1 \oplus \pi_2) \preceq (X + \pi_1) \oplus \pi_2$ enable framing the potential π_2; we remark that the latter assumption immediately holds if both potentials do not depend on the heap. Moreover, the assumption $\pi_1 \preceq \pi_1 \oplus \pi_2$ ensures that $\mathsf{aert}_{\pi_1 \oplus \pi_2}[\![C]\!](X)$ is well-defined. We use the above compositionality theorem to analyze a load-balancing approach on top of an already analyzed amortized data structure in Section 5.5.2.

5.4 Reasoning about Loops

As aert for loops is also defined via a least fixed point of a function Ψ_X, similarly to ert, we obtain an invariant-based proof rule for upper-bounding amortized expected runtimes:

Theorem 5.10 (Park Induction for aert). *Let loop = $\texttt{while}\,(\varphi)\,\{\,body\,\}$ and $X, J \in \mathbb{A}_\pi$. Then*

$$\Psi_X(J) \preceq J \quad \textit{implies} \quad \mathsf{aert}_\pi[\![loop]\!](X) \preceq J \,.$$

Example 5.11. Consider the loop in Figure 3 with clearly non-constant expected runtime. For every $m \in \mathbb{N}$, let $\mathsf{nextpow2}(m)$ be the smallest power of 2 greater than- or equal to m, and let $\mathsf{pow2}(m)$ be the predicate that evaluates to true iff m is a power of 2. Using the potential function $\pi \coloneqq 2 \cdot x - \mathsf{nextpow2}(x)$ and the aert loop invariant $J = [c = 0] \cdot 2$, the amortized expected runtime of the loop is shown to be constant. Annotations are best read from bottom to top: $X \coloneqq 0$ is the postruntime. Somewhat differently from Example 5.7, 0 is not copied to the loop body, but instead, invariant J is employed and pushed through the loop body (possibly with simplifications and *overapproximations*), obtaining $[c = 0] + 1$. We have now overapproximated $\mathsf{aert}_\pi[\![body]\!](J)$

$\geq$ /// $[c = 0] \cdot 2$ ($\Psi_0(J) \preceq J$ hence $\text{aert}_\pi\,[\![loop]\!]\,(0) \preceq J$)
Ψ /// $[c \neq 0] \cdot 0 \;+\; [c = 0] \cdot ([c = 0] + 1)$
`while ( c = 0 ) {`
  /// $[c = 0] + 1$ (obtain $\text{aert}_\pi\,[\![body]\!]\,(J) \preceq [c = 0] + 1$)
  /// $1/2 \cdot (0 + [c = 0] \cdot 2 + 2)$
  {
    /// 0
    $c := 1$
    /// $[c = 0] \cdot 2$
  } [1/2] {
    /// $[c = 0] \cdot 2 + 2$
    `if ( pow2 (x) ) { tick (x) } else { skip }`;
    /// $[c = 0] \cdot 2 + 2 - [\text{pow2}\,(x)] \cdot x$
    $x := x + 1$
    /// $[c = 0] \cdot 2$
  } [] $[c = 0] \cdot 2$ (we employ invariant $J := [c = 0] \cdot 2$)
} /// 0 (0 is the post π-runtime)

Fig. 3. A probabilistic loop with constant amortized expected runtime. Here $\pi = 2 \cdot x - \text{nextpow2}\,(x)$.

by $[c = 0] + 1$. To resemble an overapproximation of the characteristic function Ψ_0 applied to J, we construct $Z = [\neg\varphi] \cdot 0 \;+\; [\varphi] \cdot ([c = 0] + 1)$. The final (topmost) annotation indicated that, indeed $J \geq Z$ which in total confirms $J \geq \Psi_0(J)$, thus confirming — by Theorem 5.10 — that J is an upper bound for the total amortized expected runtime of the loop with respect to postruntime 0.

5.5 Case Studies

5.5.1 The Randomized Dynamic Table. A dynamic table is a dictionary data structure for maintaining a table of elements in the heap. The data structure provides, amongst others, an operation *Insert(y)* for inserting a new element with content y at the end of the table. We first describe a well-known deterministic implementation of dynamic tables using fixed-size arrays that runs in *constant* amortized time. We then employ our aert calculus to prove that a *randomized* variant of this implementation runs in *constant expected* amortized time.

We can implement dynamic tables by means of fixed-size arrays: Maintain an array A of size $s \geq 1$ in the heap and keep track of the number of cells o currently occupied by some element. A call to *Insert(y)* then behaves as follows. If $o < s$, then store y at $A[o]$—the first (i.e., with smallest offset) cell that is not occupied by some element yet—and increase o by one. In this case, we assume a runtime of 1 for storing the value y. Otherwise, i.e, if $o = s$, we need to allocate a new array A' of size $s' > s$, copy all elements from A to A', and store the new element v at $A'[o]$. We then increase o by one, deallocate the old array A, and set A to A'. In this case, we assume a runtime of $s + 1$ for copying the elements from A to A' and for storing the new element y. A clear downsize of this implementation is that *Insert(y)* has a *non-constant* worst-case runtime of $s + 1$.

```
if (o = s) {
    {s' := s + 1} [1/s+1] {s' := 2 · s} ;
    A' := alloc(s') ;
    ArrayCopy(A, s, A') ;
    DeleteArray(A, s) ;
    A := A' ;
    s := s'
} ;
⟨A + o⟩ := y ;
o := o + 1 ;
tick(1)
```

Fig. 4. Randomized Dynamic Table Insert *RandInsert*(y).

However, by choosing the size s' of the new array A' carefully, we can do better in an *amortized* sense—a prime example of amortized analysis [Cormen et al. 2009, Chapter 17]. For $s' = 2 \cdot s$, i.e., by doubling the size of the array each time it is full, we achieve a *constant* amortized time for *Insert(y)*. We remark that, when increasing the list size by some constant by, e.g., choosing $s' = s + 1$, the amortized runtime of *Insert(y)* is *not* constant.

We now consider our randomized variant *RandInsert*(y) shown in Figure 4 on p. 24. Instead of deterministically choosing $s' = 2 \cdot s$, our randomized implementation chooses $s' = s + 1$ with probability $1/s+1$ and $s' = 2 \cdot s$ with the remaining probability $1 - 1/s+1$ in case $o = s$. Thus, for small array sizes s, our randomized variant behaves with high probability like the deterministic variant with non-constant amortized runtime, while *in the limit* behaving like the classical variant with constant amortized runtime. Program *ArrayCopy(A, s, A')* copies the array A of size s to A' and consumes s units of time. *DeleteArray(A, s)* deallocates the array A and consumes no time.

Using our aert calculus and the potential $\pi = 2 \cdot o \dot{-} s$, we prove in a fully calculational way that our randomized variant is *memory-safe* and runs in *constant amortized expected time*. We have

$$\mathsf{aert}_{\pi}\,[\![\mathit{RandInsert}\,(y)]\!]\,(0) \;\leq\; \underline{4} \oplus \underline{(\!|\, o \leq s \wedge s \geq 1 \,|\!)} \oplus \bigoplus_{i=1}^{s} (\!|\, A + i - 1 \mapsto -\,|\!)\,, \tag{1}$$

i.e., when invoked on an array with head A of size at least 1 and where the offset o of the last occupied cell is at most s, *RandInsert*(y) is *memory-safe* and runs in an *amortized expected time* of at most 4. We emphasize that local reasoning simplifies our amortized analysis significantly: The frame rule enables to specify memory-safety and expected runtimes of the sub-programs *ArrayCopy(A, s, A')* and *DeleteArray(A, s)* separately, and to employ these specifications in the broader context of *RandInsert*(y).

5.5.2 The Load-Balanced Randomized Dynamic Table. To demonstrate the aert calculus' capabilities for compositional reasoning about nested data structures (cf. Section 5.3), we consider a variant of the randomized dynamic table in Section 5.5.1 that internally uses *two* dynamic tables instead of one for load balancing reasons, e. g., to enable parallel operations on the smaller tables. In particular, our variant supports an operation *BalancedInsert*(y) for inserting a value y. To ensure that the load

of the two internal dynamic tables is balanced in expectation, we flip a fair coin to decide in which of the dynamic tables the value y is to be inserted.

To model this operation in hpGCL, we take two copies $\mathit{RandInsert}_1\,(y)$ and $\mathit{RandInsert}_2\,(y)$ of the program in Figure 4, where every variable x is replaced by a copy variable x_1 and x_2, respectively. The operation $\mathit{BalancedInsert}\,(y)$ is then given by the hpGCL program

$$\{\ \mathit{RandInsert}_1\,(y)\ \}\ [0.5]\ \{\ \mathit{RandInsert}_2\,(y)\ \}\ .$$

We analyze the amortized expected runtime $\mathsf{aert}_{\pi_1 \oplus \pi_2}\,[\![\mathit{BalancedInsert}\,(y)]\!]\,(0)$ using the *extended potential function* $\pi_1 \oplus \pi_2$, where, for $i \in \{1, 2\}$, the potential $\pi_i = 2 \cdot o_i \dot{-} s_i$ is a copy of the potential used for analyzing $\mathit{RandInsert}\,(y)$. We will apply Theorem 5.9 to reuse our existing analysis for $\mathit{RandInsert}\,(y)$ (c.f. Equation (1)) and the frame rule (Theorem 5.8) to account for the second dynamic table. To this end, we first calculate for $j = 1$ and $j' = 2$ (and analogously for $j = 2$ and $j' = 1$):

$$\begin{aligned}
&\mathsf{aert}_{\pi_j}\,[\![\mathit{RandInsert}_j\,(y)]\!]\,(0)\\
\leq\ &\mathsf{aert}_{\pi_j}\,[\![\mathit{RandInsert}_j\,(y)]\!]\left(\pi_j \oplus \bigoplus_{i=1}^{s_{j'}} \llparenthesis A_{j'} + i - 1 \mapsto\ -\rrparenthesis\ -\ \pi_j\right) && \text{(by monotonicity of aert)}\\
\leq\ &(\mathsf{aert}_{\pi_j}\,[\![\mathit{RandInsert}_j\,(y)]\!]\,(0) + \pi_j) \oplus \bigoplus_{i=1}^{s_{j'}} \llparenthesis A_{j'} + i - 1 \mapsto\ -\rrparenthesis\ -\ \pi_j && \text{(by Theorem 5.8)}\\
\leq\ &\Big(\underline{4} \oplus \underline{\llparenthesis o_j \leq s_j \wedge s_j \geq 1\rrparenthesis} \oplus \bigoplus_{i=1}^{s_j} \llparenthesis A_j + i - 1 \mapsto\ -\rrparenthesis + \pi_j\Big) \oplus \bigoplus_{i=1}^{s_{j'}} \llparenthesis A_{j'} + i - 1 \mapsto\ -\rrparenthesis\ -\ \pi_j\\
&&& \text{(Equation (1))}\\
\leq\ &\underbrace{\underline{4} \oplus \underline{\llparenthesis o_j \leq s_j \wedge s_j \geq 1\rrparenthesis} \oplus \Big(\bigoplus_{i=1}^{s_j} \llparenthesis A_j + i - 1 \mapsto\ -\rrparenthesis\Big) \oplus \Big(\bigoplus_{i=1}^{s_{j'}} \llparenthesis A_{j'} + i - 1 \mapsto\ -\rrparenthesis\Big)}_{=:X_j}\\
&&& (\pi_j \text{ does not depend upon the heap})
\end{aligned}$$

In other words, we can extend the bound for $\mathit{RandInsert}_j\,(y)$ by a specification involving the array $A_{j'}$ not occurring in $\mathit{RandInsert}_j\,(y)$. This gives us

$$\begin{aligned}
&\mathsf{aert}_{\pi_1 \oplus \pi_2}\,[\![\{\ \mathit{RandInsert}_1\,(y)\ \}\ [0.5]\ \{\ \mathit{RandInsert}_2\,(y)\ \}]\!]\,(0)\\
=\ &0.5 \cdot \mathsf{aert}_{\pi_1 \oplus \pi_2}\,[\![\mathit{RandInsert}_1\,(y)]\!]\,(0) + 0.5 \cdot \mathsf{aert}_{\pi_1 \oplus \pi_2}\,[\![\mathit{RandInsert}_2\,(y)]\!]\,(0) && \text{(by Table 3)}\\
\leq\ &0.5 \cdot \big((\mathsf{aert}_{\pi_1}\,[\![\mathit{RandInsert}_1\,(y)]\!]\,(0) + \pi_1) \oplus \pi_2\ -\ \pi_1 \oplus \pi_2\big)\\
&\quad + 0.5 \cdot \big((\mathsf{aert}_{\pi_2}\,[\![\mathit{RandInsert}_2\,(y)]\!]\,(0) + \pi_2) \oplus \pi_1\ -\ \pi_1 \oplus \pi_2\big) && \text{(apply Theorem 5.9 twice)}\\
\leq\ &0.5 \cdot \big((X_1 + \pi_1) \oplus \pi_2\ -\ \pi_1 \oplus \pi_2\big) + 0.5 \cdot \big((X_2 + \pi_2) \oplus \pi_1\ -\ \pi_1 \oplus \pi_2\big) && \text{(by above reasoning)}\\
\leq\ &0.5 \cdot X_1 + 0.5 \cdot X_2 && (\pi_1 \text{ and } \pi_2 \text{ do not depend upon the heap})\\
=\ &\underline{4} \oplus \underline{\llparenthesis o_1 \leq s \wedge s_1 \geq 1 \wedge o_2 \leq s \wedge s_2 \geq 1\rrparenthesis}\\
&\quad \oplus \Big(\bigoplus_{i=1}^{s_1} \llparenthesis A_1 + i - 1 \mapsto\ -\rrparenthesis\Big) \oplus \Big(\bigoplus_{i=1}^{s_2} \llparenthesis A_2 + i - 1 \mapsto\ -\rrparenthesis\Big)\ .
\end{aligned}$$

That is, if both of the arrays A_1 and A_2 satisfy their respective specifications, then $\mathit{BalancedInsert}\,(y)$ is *memory-safe* and runs in *constant* amortized expected time.

5.5.3 The Insert-Delete-FindAny Problem [Brodal et al. 1996]. The Insert-Delete-FindAny problem is to maintain a dictionary data structure storing numbers, which provides three operations:

```
remove(x) ⨟
if ( len = 0 ) {
    any := 0 ⨟ rank := 0
} else {
    if ( x = any ) {
        Sample ⨟ Rank
    } else {
        v_x := ⟨x + 2⟩ ⨟ v_any := ⟨any + 2⟩ ⨟
        tick(1) ⨟
        if ( v_x < v_any ) {rank := rank ∸ 1 }
    }
} ⨟ free(x, x + 1, x + 2)
```

(a) Program *delete*(*x*).

```
add(y) ⨟
{
    any := H ⨟ Rank
} [1/len+1] {
    if ( len ≥ 2 ) {
        v_any := ⟨any + 2⟩ ⨟
        tick(1) ⨟
        if ( y < v_any ) {rank := rank + 1 }
    } else {
        any := H ⨟ rank := 1
    }
}
```

(b) Program *insert*(*y*).

Fig. 5. Programs *delete*(*x*) and *insert*(*y*).

- *insert*(*y*) inserts a new element with content *y* into the dictionary.
- *delete*(*x*) gets a *pointer x* to some element in the dictionary and removes this element.
- *FindAny* returns an *arbitrary* element *any* from the dictionary together with its *rank*, which is defined as one plus the number of elements in the dictionary whose content is strictly smaller than the content of *any*, or returns 0 if the dictionary is empty.

We assume a runtime model that counts the number of comparisons of elements in the dictionary. Brodal et al. [1996] provide randomized algorithms of the above operations using doubly-linked lists that each run in *constant* amortized expected time. Remarkably, they prove that every *deterministic* implementation is less efficient in the sense that there is *no* deterministic implementation of the above operations that achieves a constant amortized runtime.

We encode the algorithms provided by Brodal et al. [1996] in hpGCL and use our aert calculus to prove on source-code level that these operations indeed run in constant amortized expected time. The operations *insert*(*y*) and *delete*(*x*) are depicted in Figure 5. *FindAny* is realized by maintaining the variables *any* and *rank* with the desired properties. We specify doubly-linked lists co-inductively:

$$\begin{aligned} &\mathsf{dll}\,(H,\ end,\ pre,\ len,\ succ)\\ :=\ &\underline{\llparenthesis H = succ \land pre = end \land len = 0 \rrparenthesis}\\ &\qquad \sqcap \big(\underline{\llparenthesis len \geq 1 \rrparenthesis} \oplus \lightning v\colon\ [H \mapsto v, pre, -] \oplus \mathsf{dll}\,(v,\ end,\ H,\ len - 1,\ succ)\big) \end{aligned}$$

In particular, $\lightning\, end\colon \mathsf{dll}\,(H,\ end,\ 0,\ len,\ 0)$ specifies that the heap consists of a null-terminated doubly-linked list (with head) H. Every element x of a doubly-linked list consists of three locations in the heap: location x stores the *successor* element (or 0 if x is the last element), location $x + 1$ stores the *predecessor* element (or 0 if x is the first element), and $x + 2$ stores the *content*.

Let us now consider *delete*(*x*). Assume that the heap consists of a doubly-linked list H of length *len* containing the element x. We first execute *remove*(*x*), which removes the element x from the list without deallocating the locations associated to x, and decreases *len* by one. Then, if the list becomes empty, we set *any* and *rank* to 0. Otherwise, i.e, if the list is not empty, there are two

possible cases: Either $x = any$, which means that we removed our current *any* element. We thus need to find a new *any* together with its rank. This is realized by the programs *Sample* and *Rank*. *Sample* first samples some element uniformly at random from the list with head H and stores the result in variable *any* (requires no comparisons). *Rank* then computes the rank of the new *any* and stores the result in variable *rank* (the number of comparisons required is *len*). In the other case, we have $x \neq any$. We check whether the rank of *any* needs to be updated by comparing the content of x to the content of *any*. Finally, we deallocate the pointers associated to the element x.

Let us now consider $insert\,(y)$. Assume that the heap consists of a doubly-linked list H of length *len*. We start by executing $add\,(y)$, which allocates a new element with content y, inserts this new element at the front of the doubly-linked list with head H—thus becoming the new head—, and increases *len* by one. We then proceed randomly: With probability $1/len+1$, we set *any* to the new element H and compute its rank (which again takes *len* comparisons). With the remaining probability $1 - 1/len+1$, we either keep the current *any* and check whether its rank needs to updated if the list was not empty before (i.e., if $len \geq 2$), or set *any* to H and *rank* to 1 if the list was empty before (i.e., if $len \leq 1$).

Now define the potential $\pi := len \cdot (1 + [any = x])$. We prove

$$\mathsf{aert}_\pi\,[\![delete\,(x)]\!]\,(0) \;\leq\; 1 + \wr x \mapsto -,-,-\wr \oplus 0 + \wr any \mapsto -,-,-\wr \oplus 0 + \wr end\colon \mathsf{dll}\,(H,\, end,\, 0,\, len,\, 0)$$

i.e., if the heap consists of a doubly-linked list H of size *len* containing the (not necessarily distinct) elements *any* and x, then $delete\,(x)$ is memory-safe and runs in an amortized expected time of at most 1. Moreover, we show

$$\mathsf{aert}_\pi\,[\![insert\,(y)]\!]\,(0) \;\leq\; 4 + [len \geq 1] \cdot (\wr any \mapsto -,-,-\wr \oplus 0) + \wr end\colon \mathsf{dll}\,(H,\, end,\, 0,\, len,\, 0)$$

i.e., if the heap consists of a doubly-linked list H of size *len* containing the element *any* in case H is non-empty, then $insert\,(y)$ is memory-safe and runs in an amortized expected time of at most 4.

6 RELATED WORK

There is a plethora of research on the verification of runtime bounds. We focus on literature most closely related to our approach, specifically techniques for formal reasoning about (1) expected runtimes of probabilistic programs and (2) amortized runtimes of non-probabilistic programs.

Reasoning about expected runtimes. Our ert calculus combines two existing approaches to enable proving upper bounds on expected runtimes of randomized algorithms manipulating dynamic data structures: the original ert calculus of Kaminski et al. [2018] and quantitative separation logic (QSL) of Batz et al. [2019]. Developing a calculus based on a separating addition (our $\oplus$) was initially proposed by Matheja [2020, Chapter 9.1]. Haslbeck [2021, Chapter 4] formalized this idea and proved that one obtains a variant of QSL for reasoning about upper bounds. His calculus and its properties essentially coincide with our ert with two exceptions: (1) ert allows the allocation of arbitrarily large chunks of memory instead of fixed-sized ones; and (2) we prove soundness of ert with respect to an operational semantics based on MDPs; earlier attempts to prove soundness by Haslbeck [2021, p. 47] lead to technical issues which were not further pursued.

Ngo et al. [2018]; Wang et al. [2020] apply the potential method for automatic reasoning about expected runtimes. The soundness theorem in Ngo et al. [2018] relies on the soundness of the original ert calculus of Kaminski et al. [2018]. Our more general calculus for RSL provides foundations for proving their techniques sound when applied to probabilistic pointer programs. Wang et al. [2020] presents a type-based analysis for deriving over-approximations of expected runtime. Their upper bounds are proven sound w.r.t. a distribution-based operational cost semantics. Other approaches for analyzing expected runtimes of probabilistic programs, such as [Brázdil et al. 2015; Celiku and McIver 2005; Meyer et al. 2021; Monniaux 2001; Moosbrugger et al. 2021], neither support dynamic

data structures nor consider amortization. Recently, Leutgeb et al. [2022] defined a type-and-effect system for a functional programming language that can automatically infer logarithmic amortized bounds on randomized tree and heap structures. Our amortized calculus is a weakest-precondition-style framework whose soundness w.r.t. an operational semantics is shown using a novel technique, and which recovers a well-known interpretation of amortized expected runtime analysis at the level of program semantics for arbitrary sequences of data structure operations.

Verifying amortized runtimes for non-probabilistic programs. Amortized runtime analysis builds upon either the *potential method* (a.k.a. physicists view) or the *banker's view* as already proposed in Tarjan [1985], who already noted that these two views are equivalent. The paper Haslbeck and Nipkow [2018] surveys existing verification techniques for amortized runtimes. In particular, the potential method has been formalized and applied in an interactive theorem prover by Nipkow [2015]; Nipkow and Brinkop [2019]. Carbonneaux et al. [2014] developed a quantitative logic similar to our ert transformer (for deterministic programs) based on potentials. Potentials are also at the foundation of (automatic but not necessarily amortized) type-based runtime analyses, e. g., [Kahn and Hoffmann 2020; Rajani et al. 2021], pioneered by Hoffmann [2011]. A recent survey of type-based analysis is given in Hoffmann and Jost [2022].

The banker's view of amortized analysis has been integrated into separation logic by Atkey [2011]. He introduced time credits, a dedicated resource modeling the remaining amount of time a program may consume. With this view, one can naturally reason about time credits in the same way as for heap allocated memory, e. g. by storing time credits in individual elements of dynamic data structures. In contrast to many other runtime verification techniques, Atkey proves his approach sound w.r.t. a program semantics. The intricacies encountered when using time credits for reasoning about *asymptotic* (amortized) complexities are discussed by Guéneau et al. [2018]. Charguéraud and Pottier [2019] implemented time credits in a verification tool and verified the amortized complexity of the Union-Find data structure. A variant of time credits, called time receipts [Mével et al. 2019], enables reasoning about lower runtime bounds.

None of these works reason about amortized *expected* runtimes of randomized algorithms. To enable this, we chose to use the potential method to formalize aert, since potentials are closely related to expectations, which also map states to a quantity. Reasoning about potentials thus seems natural if one is used to working with expectations and quantitative invariants. Some of our proof rules exploit the above similarity to mix potentials and expectations, e.g. Theorems 5.8 and 5.9.

7 CONCLUSION

We have presented calculi featuring compositionality and local reasoning for the verification of (amortized) expected runtimes of probabilistic pointer programs. We have established soundness results w.r.t. an operational semantics and demonstrated the applicability of our techniques.

Future work includes the runtime verification of (randomized) splay-trees [Albers and Karpinski 2002; Fürer 1999; Sleator and Tarjan 1985] and skip lists [Pugh 1989], and mechanizing the aert-calculus building upon the work by Haslbeck [2021]. Further promising directions for automated aert reasoning include leveraging entailment checking techniques for quantitative separation logic [Batz et al. 2022a] and generalizations of k-induction for probabilistic programs [Batz et al. 2021a].

ACKNOWLEDGMENTS

We thank Gerhard Woeginger on the fruitful discussions about amortized analysis. Furthermore, we are grateful for the reviewers for their highly constructive feedback that, in particular, contributed to the development of Theorem 5.9.

REFERENCES

Susanne Albers and Marek Karpinski. 2002. Randomized Splay Trees: Theoretical and Experimental Results. *Inform. Process. Lett.* 81, 4 (2002), 213–221.

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Log. Methods Comput. Sci.* 7, 2 (2011).

Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *LICS*. IEEE, 1–13.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021a. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV (2) (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 524–549.

Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. 2022a. Foundations for Entailment Checking in Quantitative Separation Logic. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 57–84.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021b. Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic — A Logic for Reasoning about Probabilistic Programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2022b. A Calculus for Amortized Expected Runtimes - Extended Version. *CoRR* (2022). to appear.

David Blackwell. 1967. Positive dynamic programming. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1. University of California Press, 415–418.

Tomás Brázdil, Stefan Kiefer, Antonín Kucera, and Ivana Hutarová Vareková. 2015. Runtime Analysis of Probabilistic Programs with Unbounded Recursion. *J. Comput. System Sci.* 81, 1 (2015), 288–310.

Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. 1996. The Randomized Complexity of Maintaining the Minimum. *Nord. J. Comput.* 3, 4 (1996), 337–351.

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *PLDI*. ACM, 270–281.

Orieta Celiku and Annabelle McIver. 2005. Compositional Specification and Analysis of Cost–Based Properties in Probabilistic Programs. In *Proc. of the International Symposium on Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 3582)*. Springer, 107–122.

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365.

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

Martin Fürer. 1999. Randomized Splay Trees. In *SODA*. ACM/SIAM, 903–904.

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *ESOP (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 533–560.

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

Maximilian Paul Louis Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds - A Study in Meta Theory. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 155–171.

Jan Hoffmann. 2011. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. Ph. D. Dissertation. LMU Munich.

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* (2022).

Samin S. Ishtiaq and Peter William O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 14–26.

David M. Kahn and Jan Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 12077)*. Springer, 359–380.

Benjamin Lucien Kaminski. 2019. *Advanced Weakest Precondition Calculi for Probabilistic Programs*. Dissertation. RWTH Aachen University, Aachen. https://doi.org/10.18154/RWTH-2019-01829

Benjamin Lucien Kaminski and Joost-Pieter Katoen. 2017. A Weakest Pre–expectation Semantics for Mixed–sign Expectations. In *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1–12.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* (2018), 30.

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. System Sci.* 30, 2 (1985), 162–178.

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Proc. of the International Conference on Computer-Aided Verification (Lecture Notes inn Computer Science)*. (to appear).

Christoph Matheja. 2020. *Automated Reasoning and Randomization in Separation Logic*. Dissertation. RWTH Aachen University, Germany.

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 3–29.

Fabian Meyer, Marcel Hark, and Jürgen Giesl. 2021. Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 12651)*. Springer, 250–269.

David Monniaux. 2001. An Abstract Analysis of the Probabilistic Termination of Programs. In *Proc. of the Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2126)*. Springer, 111–126.

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021. Automated Termination Analysis of Polynomial Probabilistic Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 12648)*. Springer, 491–518.

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 496–512.

Tobias Nipkow. 2015. Amortized Complexity Verified. In *ITP (Lecture Notes in Computer Science, Vol. 9236)*. Springer, 310–324.

Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *J. Autom. Reason.* 62, 3 (2019), 367–391.

William W. Pugh. 1989. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *WADS (Lecture Notes in Computer Science, Vol. 382)*. Springer, 437–449.

Martin Lee Puterman. 2005. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.

John Charles Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of the Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 55–74.

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* (1985), 652–686.

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* (1985), 306–318.

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 110:1–110:31.

Received 2022-07-07; accepted 2022-11-07