Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.) in Informatik

Expected Runtimes of Probabilistic Pointer Programs

Betreuer: Prof. Dr. Thomas Noll

vorgelegt an der

Rheinisch-Westfälischen Technischen Hochschule Aachen

- Fakultät für Mathematik, Informatik und Naturwissenschaften -

von

Caroline Jabs

Matrikelnummer: 368103

Aachen, den 12.09.2019

Abstract

This thesis presents a programming language for probabilistic pointer programs and a formal method for reasoning about the expected runtimes of those programs: The expected runtime calculus. Similar to Dijkstra's weakest precondition calculus the main part of this calculus is an expected runtime transformer ert, which reasons backwards. It can compute (in)finite runtimes for probabilistic pointer programs with respect to a post-runtime.
We prove that this calculus is complete and sound with respect to an operational model. After introducing invariant-based proof rules for loops, we apply the calculus to several examples including the partitioning algorithm used in Hoare's quicksort algorithm and a random walk on a list.

Contents

1 Introduction **1**

2 Preliminaries **3**

3 A probabilistic Programming Language $\mathcal{P}$ **7**

4 Predicates **9**
4.1 Connectives between Runtimes . . . 10
4.1.1 Separating Conjunction . . . 10
4.1.2 Separating Implication . . . 10

5 Operational MRMs **11**

6 Expected Runtime Calculus **15**
6.1 Definition of ert . . . 15
6.2 Soundness of ert . . . 17
6.3 Properties of ert . . . 25
6.4 Loops . . . 25
6.5 Examples . . . 26
6.5.1 Switch . . . 26
6.5.2 Partitioning . . . 27
6.5.3 Dice . . . 32

7 Operational MDPs **35**

8 Expected Runtime Calculus with (De)Allocation **37**
8.1 Definition of ert with (de)allocation . . . 37
8.2 Soundness with (de)allocation . . . 38
8.3 More Examples . . . 40
8.3.1 List Generation . . . 41
8.3.2 List Deletion . . . 43
8.3.3 List Generation;List Deletion . . . 45
8.3.4 Comp . . . 48
8.3.5 Alloc . . . 54
8.3.6 Random List Walk . . . 56

9 Conclusion **65**

10 Appendix **69**
10.1 Random List Walk . . . 71

Chapter 1

Introduction

Probabilistic programs are programs which can contain probabilistic elements. A probabilistic statement in a program can be for instance the probabilistic choice between two subprograms, so that one is executed with probability p and the other one with probability $1 - p$. An example is shown in Listing 1.1, in which the program simulates a coin toss. 1 ("heads") and 0 ("tails") are both assigned to c with probability $\frac{1}{2}$.

```
1  {c := 1} [1/2] {c := 0}
```

Listing 1.1: Coin toss

Apart from simulating stochastic processes probabilistic programs can also be used to simulate unreliable hardware like shown in Listing 1.2. In this program we simulate the situation that the allocation of new memory fails in 5% of the cases. A failure is simulated by the empty and effectless program.

```
1  {x := new(0)} [0.95] {empty}
```

Listing 1.2: Faulty Allocation

But the main application for probabilistic programs is implementing probabilistic or randomized algorithms, which are used in machine learning, artificial intelligence, quantum computing and many other fields [5, 11, 12]. Randomized algorithms are for instance used because they can often achieve better expected runtimes than traditional algorithms without probabilistic elements. Hoare's quicksort algorithm with randomized pivot element is probably one of the most popular examples. While quicksort has a worst case runtime of $\mathcal{O}(n^2)$, the randomized version can achieve an expected runtime of $\mathcal{O}(n \cdot log(n))$ for every input with n being the length of the array we want to sort [6]. Due to the probabilistic statements in probabilistic programs a single run of a probabilistic program and its runtime cannot be predicted with certainty. We can only make statements about the expected runtime of those programs.
In this paper we are going to present a formal method for reasoning about expected runtimes of probabilistic pointer programs: The expected runtime calculus. Probabilistic pointer programs can contain probabilistic elements as well as (de)allocation of memory and pointer operations. The central part of the expected runtime calculus is an expected runtime transformer ert: ert[C](f) returns the expected runtime of program C with respect to post-runtime or continuation f, which captures the runtime of the code following C. Our expected runtime transformer is a marriage between the ert presented in [10], which is defined for pointerless probabilistic programs, and the Quantitative Separation Logic

for probabilistic pointer programs presented in [3].

Outline At first we will be introducing a few theoretical concepts including Markov Chains and Markov Decision Processes in Chapter 2, which we will employ as operational models for our programs. In Chapter 3 we are going to introduce the probabilistic programming language we will be using throughout this thesis. Predicates for reasoning about the stack and the heap are presented in Chapter 4. Then we will at first consider the programs without (de)allocation in Chapter 5 and 6 and then all programs in Chapter 7 and 8. In Chapter 5 respectively 7 we will present operational models for our programs. In Chapter 6 and 8 we will define our expected runtime transformer, prove its soundness and also apply it to several examples before we will reach our conclusion in Chapter 9.

Chapter 2

Preliminaries

Before we can present our expected runtime calculus, we need to introduce a few necessary theoretical concepts.

Definition 2.1 (Partial order [7])**.** *$(D, \preceq)$ is a partial order, if $\preceq \subseteq D \times D$ is a binary relation fulfilling*

$$\begin{aligned} x \preceq x &\text{ (Reflexivity)}, \\ x \preceq y \text{ and } y \preceq x \text{ implies } x = y &\text{ (Anti-symmetry) and} \\ x \preceq y \text{ and } y \preceq z \text{ implies } x \preceq z &\text{ (Transitivity)} \end{aligned}$$

for all $x, y, z \in D$.

Example 2.2. *$(\mathbb{R}^{\infty}_{\geq 0}, \leq)$ is a partial order with $r \leq \infty$ for all $r \in \mathbb{R}_{\geq 0}$.*

Definition 2.3 (ω-chain [8])**.** *For a partial order $(D, \preceq)$, a countable subset $\{s_i \in D \mid i \in \omega\} \subseteq D$ is called an ω-chain, if $s_i \preceq s_{i+1}$ for all $i \in \omega$.*

Definition 2.4 (Supremum of sets)**.** *If $(D, \preceq)$ is a partial order, then an upper bound of a subset $X \subseteq D$ is an element $d \in D$ with $x \preceq d$ for all $x \in X$. The supremum $\sup X$ is the least upper bound, which is an upper bound $d \in D$, such that for all other upper bounds $d' \in D$ holds $d \preceq d'$.*

Definition 2.5 (Infimum of sets)**.** *If $(D, \preceq)$ is a partial order, then a lower bound of a subset $X \subseteq D$ is an element $d \in D$ with $d \preceq x$ for all $x \in X$. The infimum $\inf X$ is the greatest lower bound, which is an lower bound $d \in D$, such that for all other lower bounds $d' \in D$ holds $d' \preceq d$.*

Definition 2.6 (ω-complete partial order (ω-cpo) [13])**.** *A set D with a partial order $\preceq$ and a least element $\perp$ build an ω-complete partial order $(D, \preceq, \perp)$, if every ω-chain $x_0 \preceq x_1 \preceq x_2 \preceq \ldots$ in D has a supremum in D.*

Definition 2.7 (Complete lattice [4])**.** *A partial order $(D, \preceq)$ is a complete lattice, if each subset $X \subseteq D$ has a supremum and an infimum.*

Example 2.8. *Due to the completeness of the real numbers, $(\mathbb{R}^{\infty}_{\geq 0}, \leq)$ is a complete lattice with $\inf R = \infty$, iff $R = \{\infty\} \subseteq \mathbb{R}^{\infty}_{\geq 0}$ and $\infty \in R \subseteq \mathbb{R}^{\infty}_{\geq 0}$ implies $\sup R = \infty$.*

Remark 2.9. *Each complete lattice is also an ω-complete partial order, since every ω-chain is a subset.*

Definition 2.10 (Monotony [13]). *Let D, E be two ω-cpos and $F : D \to E$. F is monotone, if*

$$x \preceq y \Rightarrow F(x) \preceq F(y)$$

for all $x, y \in D$.

Definition 2.11 (Continuity [13]). *Let D, E be two ω-cpos and $F : D \to E$. F is ω-continuous, if F is monotone and for every ω-chain $x_0 \preceq x_1 \preceq \dots$ in D we have*

$$F(\sup_n x_i) = \sup_n F(x_i).$$

Theorem 2.12 (Kleene's Fixedpoint Theorem [13]). *Let $(D, \preceq, \bot)$ be an ω-cpo and let $F : D \to D$ be ω-continuous. Then F has a least fixed point in D, which means there exists an $x \in D$ with $F(x) = x$ and $F(y) = y$ implies $y \preceq x$ for all $y \in D$.*

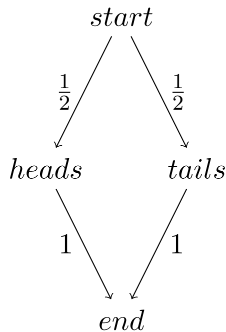
Definition 2.13. *We will use $\lambda X_\bullet e$ to denote a function f with $f(a) = e[x \backslash a]$ where $e[x \backslash a]$ is obtained by replacing all occurrences of variable x in expression e by argument a.*

As mentioned before, Markov chains will serve as models for probabilistic programs. They are very similar to transition systems. The only difference between them is the fact, that the successor configurations are chosen probabilistically instead of nondeterministically [1].

Definition 2.14 (**Markov Chain**). *A Markov chain (MC) is a 3-tuple $\mathcal{M} = (\mathcal{K}, \mathcal{P}, s_0)$, where $\mathcal{K}$ is a countable set of configurations, $\mathcal{P} : \mathcal{K} \times \mathcal{K} \to \mathbb{R}_{\geq 0}$ with $\sum_{\kappa' \in \mathcal{K}} \mathcal{P}(\kappa, \kappa') = 1$ for all $\kappa \in \mathcal{K}$ is a probability function and $\kappa_0 \in \mathcal{K}$ is a starting configuration. $\mathcal{P}(\kappa, \kappa')$ is the probability, that the system turns from configuration κ into configuration κ'. Every Markov chain induces an underlying digraph in which nodes represent configurations and edges exist between vertices κ and κ' exactly when $\mathcal{P}(\kappa, \kappa') > 0$.*

$\Pi^{\mathcal{M}}(s, t)$ is the set of all finite loop-free paths between the nodes s and t in the graph induced by Markov chain $\mathcal{M}$. If $\mathcal{M}$ is given within context, we will also use $\Pi(s, t)$ instead of $\Pi^{\mathcal{M}}(s, t)$. The Probability of a finite path $\pi = \pi_0 \dots \pi_n \in \Pi^{\mathcal{M}}(\pi_0, \pi_n)$ is computed as $\Pr^{\mathcal{M}}(\pi) = \prod_{i=0}^{n-1} \mathcal{P}(\pi_i, \pi_{i+1})$. The probability that we reach configuration t from configuration s is $\Pr^{\mathcal{M}}(s \vDash t) = \sum_{\pi \in \Pi^{\mathcal{M}}(s,t)} \Pr^{\mathcal{M}}(\pi)$, which is the sum over the probabilities of all possible paths from s to t. $s \vDash t$ denotes the event, that configuration t is reachable from configuration s.

Example 2.15. *We can model a coin toss as a Markov chain:*
$\mathcal{M}_{coin} = (\mathcal{K}, \mathcal{P}, s_0)$ with $\mathcal{K} = \{start, heads, tails, end\}$, $\kappa_0 = start$ and $\mathcal{P}$ as shown in the induced digraph:

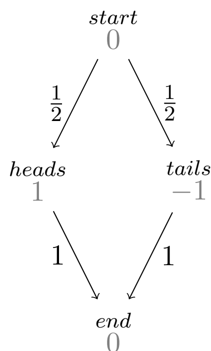


Now we have introduced a basic concept for modeling a probabilistic process. Since we are interested in the runtimes of programs, we need to keep track of the consumed time in our model, too. We will use Markov reward models for this by defining the reward as the consumed time.

Definition 2.16 (**Markov Reward Model** [1])**.** *A Markov reward model (MRM) $\widetilde{\mathcal{M}}$ is a tuple $(\mathcal{M}, rew)$ consisting of a Markov chain and a reward function $rew : \mathcal{K} \to \mathbb{R}$ mapping each configuration to a real number.*

For a finite path $\pi = \pi_0 \dots \pi_n \in \Pi^{\mathcal{M}}(\pi_0, \pi_n)$ the reward of the path is defined as $rew(\pi) = \sum_{i=0}^{n-1} rew(\pi_i)$ [10].

Example 2.17. *Anna and Bob are tossing a coin. If the coin shows heads, Anna gets 1€ from Bob. If it shows tails she has to give Bob 1€. We can model this game as a Markov reward model, where the reward is the money Anna wins respectively loses. For this Markov reward model ($\mathcal{M}_{coin}$, rew) we use the Markov chain presented in example 2.15 and define the reward function as shown in the following graph. The reward of a configuration is noted under the name of the configuration. As we can see the expected profit, which will be defined in Definition 2.18, is 0€.*



Now we can model a probabilistic system with a reward. Additionally, we need to be able to compute the expected reward until reaching a goal configuration from an initial configuration, because this will represent the expected runtime later.

Definition 2.18 (**Expected Reward of an MRM**)**.** *Let $\widetilde{\mathcal{M}}$ = ($\mathcal{M}$,rew) be a Markov reward model and $s, t \in \mathcal{K}$ two configurations. If there is a positive possibility of not reaching the goal configuration t from the starting configuration s, then this means, that there is at least one path γ from s that will never reach t and has a positive probability. In the worst case scenario γ could be a path on which the program never terminates and the runtime is infinite. For this reason we define the following: If $\Pr^{\mathcal{M}}(s \vDash t) < 1$ then the expected reward until reaching t from s is $\text{ExpRew}(s \vDash t) = \infty$.*
Otherwise $\text{ExpRew}^{\widetilde{\mathcal{M}}}(s \vDash t) = \sum_{\pi \in \Pi^{\mathcal{M}}(s,t)} \Pr^{\mathcal{M}}(\pi) \cdot rew(\pi)$.

Example 2.19. *Let us consider $\widetilde{\mathcal{M}} = (\mathcal{M}_{coin}, rew)$ from example 2.17.*
Then $\text{ExpRew}^{\mathcal{M}}(start \vDash end) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (-1) = 0$.

In a Markov chain or a Markov Reward Model there are no nondeterministic choices. Since allocation of memory is a nondeterministic process, we will need a model, which can realize probabilistic *and* nondeterministic choices. For this purpose we can use Markov decision processes.

Definition 2.20 (Markov Decision Process). *A Markov decision process (MDP) is a 5-tuple $\mathfrak{M} = (\mathcal{K}, Act, \mathcal{P}, s_0, rew)$, where $\mathcal{K}$ is the set of configurations, Act the set of possible actions, κ_0 the starting configuration and $rew : \mathcal{K} \to \mathbb{R}$ a reward function. $\mathcal{P} : \mathcal{K} \times Act \times \mathcal{K} \to \mathbb{R}_{\geq 0}$ is a probability function which fulfills $\sum_{\kappa' \in \mathcal{K}} \mathcal{P}(\kappa, \alpha, \kappa') \in \{0, 1\}$ for all $\kappa \in \mathcal{K}$ and all $\alpha \in Act$. $\sum_{\kappa' \in \mathcal{K}} \mathcal{P}(\kappa, \alpha, \kappa')$ can be 0 for some $\kappa \in \mathcal{K}$ and $\alpha \in Act$ because not all actions are possible in all configurations. If $\sum_{\kappa' \in \mathcal{K}} \mathcal{P}(\kappa, \alpha, \kappa') = 1$ for a configuration κ and a action α, then we say the action α is enabled in configuration κ. $\mathcal{E}(\kappa) \subseteq Act$ is the set of all enabled actions in configuration κ.*

Remark 2.21. *A Markov decision process $\mathfrak{M}$ already is a Markov chain if no nondeterministic choice is possible. This is for example the case, if $|\mathcal{E}(\kappa)| = 1$ for all $\kappa \in \mathcal{K}$.*

If we want to examine the expected reward of a MDP, we have to resolve the nondeterminism with a scheduler at first.

Definition 2.22 (Scheduler for a MDP). *A scheduler for a MDP $\mathfrak{M}$ is a mapping $\mathfrak{S} : \mathcal{K}^+ \to Act$ from all possible finite sequences of configurations to enabled actions of the last configuration in the sequence.*

A scheduler $\mathfrak{S}$ for $\mathfrak{M}$ induces a Markov Reward Model $\widetilde{\mathfrak{M}}_{\mathfrak{S}} = (\mathfrak{M}_{\mathfrak{S}}, rew_{\mathfrak{S}})$. For the expected runtime of a MDP we assume a demonic scheduler, which makes all choices in a way, that maximizes the runtime. Therefore we choose the supremum of all expected runtimes:

Definition 2.23 (Expected Reward of a MDP). *The expected reward for a MDP $\mathfrak{M}$ until reaching configuration t from configuration s is defined as follows:*
If

$$\inf_{\mathfrak{S}} \Pr^{\mathfrak{M}_{\mathfrak{S}}}\{s \models t\} = \inf_{\mathfrak{S}} \sum_{\pi \in \Pi(s,t)} \Pr^{\mathfrak{M}_{\mathfrak{S}}}\{\pi\} < 1,$$

then

$$\mathrm{ExpRew}^{\mathfrak{M}}(s \models t) = \infty.$$

Otherwise

$$\mathrm{ExpRew}^{\mathfrak{M}}(s \models t) = \sup_{\mathfrak{S}} \sum_{\pi \in \Pi(s,t)} \Pr^{\mathfrak{M}_{\mathfrak{S}}}\{\pi\} \cdot rew(\pi).$$

Chapter 3

A probabilistic Programming Language $\mathcal{P}$

We will work with a simple imperative programming language, which is inspired by the *heap-manipulating probabilistic guarded command language* in [3] and extended by a Uniform distribution command. The set of all *p*robabilisitic *p*ointer *p*rograms written in our program language $\mathcal{P}$ is called $\mathcal{P}^3$ and is given by the following grammar:

c	$\rightarrow$	**empty**	(empty program)
		$\mid x := e$	(assignment)
		$\mid x := Uniform(u, v)$	(Uniform assignment)
		$\mid c; c$	(sequential composition)
		$\mid$ **if** $(b)\ \{c\}$ **else** $\{c\}$	(conditional choice)
		$\mid$ **while** $(b)\ \{c\}$	(while-loop)
		$\mid \{c\}\ [p]\ \{c\}$	(probabilistic choice)
		$\mid < e > := e'$	(heap manipulation)
		$\mid x := < e >$	(heap lookup)
		$\mid x := new(\vec{e})$	(allocation)
		$\mid free(e)$	(deallocation)

In this grammar x is a variable in Var, where Var is a possibly infinite set of variables. e, e' are arithmetic expressions which can only consist of variables, numbers and mathematical operators. $\vec{e}$ is a vector of arithmetic expressions, u,$v \in \mathbb{Z}$ with $u \leq v$ are integers and $p \in \mathbb{Q} \cap (0, 1)$ a probability. b is a predicate, which can only compare arithmetic expressions. This way heap-data can neither be directly used in an arithmetic expression nor in a predicate and no memory-faults are possible during evaluation of an arithmetic expression or a predicate.
A **stack** keeps track of the values assigned to all the variables and can be represented by a mapping from variables to integers. The set of all stacks can be formally written as

$$S = \{s \mid s : Var \rightarrow \mathbb{Z}\}.$$

For a given stack s the value of an expression e is denoted $s(e)$.
A **heap** stores the values of a finite amount of allocated memory addresses as a finite mapping from addresses to integers. The set of all possible heaps is denoted

$$H = \{h \,|\, h : N \to \mathbb{Z}, N \subset \mathbb{N}_{>0}, |N| < \infty\}.$$

For a given heap $h : N \to \mathbb{Z}$ the domain $dom(h)$ is N. Two heaps h_1, h_2 are disjoint ($h_1 \perp h_2$), if their domains are disjoint ($dom(h_1) \cap dom(h_2) = \emptyset$).

$$h_1 \bigstar h_2 : dom(h_1) \cup dom(h_2) \to \mathbb{Z}, (h_1 \bigstar h_2)(n) = \begin{cases} h_1(n) & \text{, if } n \in dom(h_1) \\ h_2(n) & \text{, if } n \in dom(h_2) \end{cases}$$

is called the disjoint union of the heaps h_1 and h_2, if these are disjoint heaps.
The tuple (s, h) of the current stack and heap is called current **program state** and

$$\Sigma = \{(s, h) \,|\, s \in S, h \in H\}$$

is the set of all program states.
To note that variable x is set to value $v \in \mathbb{Z}$, we write $s[x \backslash v]$ for stack s, which is defined as follows:

$$s[x \backslash v] = \lambda\, y_\bullet \begin{cases} v & \text{, if } y = x \\ s(y) & \text{, if } y \neq x. \end{cases}$$

For heap h $h[u \backslash v]$ is defined similar.
Since the runtime of a program generally depends on the program state before execution, we represent runtimes as mappings from program states to non-negative real numbers or infinity. The set of all runtimes can then be formally described as

$$\mathbb{T} = \{f \,|\, f : \Sigma \mapsto \mathbb{R}_{\geq 0}^{\infty}\} \text{ [10].}$$

Lemma 3.1. $(\mathbb{T}, \preceq)$ *is a complete lattice with* $f \preceq g \Leftrightarrow f(\sigma) \leq g(\sigma)\ \forall \sigma \in \Sigma$ *and* $\perp = 0 : \sigma \mapsto 0$ *[10].*

Proof. It is already proven in [10], that $(\mathbb{T}, \preceq)$ is a partial order. The supremum $\sup T$ and infimum $\inf T$ of a subset $T \subseteq \mathbb{T}$ is defined pointwise:

$$\sup T = \lambda\sigma_\bullet \sup\{f(\sigma) \,|\, f \in T)\}$$

$$\inf T = \lambda\sigma_\bullet \inf\{f(\sigma) \,|\, f \in T)\}$$

The infimum and supremum of a subset $T \subseteq \mathbb{T}$ always exist, since $\mathbb{R}_{\geq 0}^{\infty}$ is a complete lattice.

□

Chapter 4

Predicates

For reasoning about the state of a program we will need the concept of predicates.

Definition 4.1 (**Predicates**). *A predicate is a function mapping each state to either* 0 *or* 1. *The set of predicates is given by* $\mathbb{P} = \{p \,|\, p : \Sigma \to \{0, 1\}\}$*, which is a subset of* $\mathbb{T} = \{f \mid f : \Sigma \to \mathbb{R}_{\geq 0}\}$.

A predicate can be seen as a function that checks if a concrete property holds for a program state or not and accordingly returns 1 or 0. We will need the following three predicates for defining our ert. The interpretation predicate can check boolean expressions over variables and thus just considers the program stack.

Definition 4.2 (**Interpretation predicate**). *The interpretation predicate* $[\![\,]\!]$ *interprets boolean expressions. For a boolean expression* b *and a program state* (s, h) *the interpretation predicate is defined as*

$$[\![b : true]\!](s, h) = \begin{cases} 1, \text{ if } s(b) = true \\ 0, \text{ if } s(b) = false \end{cases}$$

$$[\![b : false]\!](s, h) = \begin{cases} 0, \text{ if } s(b) = true \\ 1, \text{ if } s(b) = false. \end{cases}$$

The next two predicates just consider the program heap during evaluation: The points-to-predicate checks wether a certain heap cell is allocated and contains a given value. This predicate is important when considering a heap lookup or heap manipulation.

Definition 4.3 (**Points-to-predicate** [3]). *The points-to-predicate evaluates to 1 if and only if the given heap contains exactly one address* $s(e)$ *storing* $s(e')$. *For two arithmetic expressions* e, e' *and a program state* $(s, h) \in \Sigma$ *it is defined as*

$$[e \mapsto e'] = \lambda\,(s, h)\bullet \begin{cases} 1 & \text{, if } dom(h) = \{s(e)\} \text{ and } h(s(e)) = s(e') \\ 0 & \text{, otherwise .} \end{cases}$$

Similar to the points-to-predicate the allocated pointer predicate checks wether a given heap cell is allocated but the stored value is irrelevant. Thus this predicate can help checking wether a memory fault occurs.

Definition 4.4 (**Allocated Pointer Predicate** [3]). *The allocated pointer predicate* $[e \mapsto _]$ *evaluates to 1 if and only if the heap consists exactly of the address* $s(e)$. *The value*

stored in this address does not influence the evaluation of the allocated pointer predicate. For an arithmetic expression e and a program state $(s, h) \in \Sigma$ it is defined as

$$[e \mapsto _] = \lambda\,(s,h)\bullet \begin{cases} 1 & \text{, if } dom(h) = \{s(e)\} \\ 0 & \text{, otherwise .} \end{cases}$$

4.1 Connectives between Runtimes

Our expected runtime calculus is based on separation logic (SL), which uses the separating conjunction ($\bigstar$) and the separating implication ($\multimap\!\!\bigstar$). Both are defined for predicates in SL, so we have to redefine them for our purposes.

4.1.1 Separating Conjunction

For our expected runtime calculus we will use a maximum of a product of runtimes as it is suggested in [3] for quantitative separation logic:

Definition 4.1.1 (**Runtime Separation Conjunction** [3])**.** *The runtime separation conjunction $f \bigstar g$ of two runtimes $f, g \in \mathbb{T}$ is defined as follows:*

$$f \bigstar g = \lambda\,(s,h)\bullet \max_{h_1,h_2}\{f(s,h_1)\cdot g(s,h_2) \mid h = h_1 \bigstar h_2\}$$

$f \bigstar g$ basically partitions the heap into two heaps h_1 and h_2 in a way that maximizes $f(s,h_1)\cdot g(s,h_2)$.

Remark 4.1.2. *$[u \mapsto (e_1,\ldots,e_n)]$ is a short notation for $[u \mapsto e_1] \bigstar \ldots \bigstar [u \mapsto e_n]$, which we can use to check multiple addresses at the same time.*

4.1.2 Separating Implication

Again, our definition for runtime separating implication is inspired by the following definition for the quantitative separating implication presented in [3]:

$$p \multimap\!\!\bigstar\; f = \lambda\,(s,h)\bullet \inf_{h'}\{f(s,h' \bigstar h) \mid h' \perp h \text{ and } p(s,h') = 1\}$$

for a runtime $f \in \mathbb{T}$ and a predicate $p \in \mathbb{P}$. Since we are more interested in worst case runtimes than in best case runtimes, we are using the supremum instead of the infimum:

Definition 4.1.3 (**Runtime Separating Implication**)**.** *The runtime separating implication of a predicate $p \in \mathcal{P}$ and a runtime $f \in \mathbb{T}$ is defined as*

$$p \multimap\!\!\bigstar\; f = \lambda\,(s,h)\bullet \sup_{h'}\{f(s,h' \bigstar h) \mid h' \perp h \text{ and } p(s,h') = 1\}$$

$p \multimap\!\!\bigstar\; f$ basically adds a heap h' that fulfills p and maximizes $f(s, h \bigstar h')$

Chapter 5

Operational MRMs

As mentioned before, we will work with models of probabilistic pointer programs to prove the soundness of our ert. At first we will only consider programs $\mathcal{P}^3_* \subseteq \mathcal{P}^3$ in which allocation and deallocation of memory does not appear. Since those programs do not contain any nondeterministic choices, we can model a program $C \in \mathcal{P}^3_*$ as a Markov chain $\mathcal{M}^f_{\sigma_0}[\![C]\!] = (\mathcal{K}, \mathcal{P}, \kappa_0)$, where $\mathcal{K} = ((\mathcal{P}^3 \cup \{\downarrow; C \mid C \in \mathcal{P}^3_*\} \cup \{\downarrow\}) \times \Sigma) \cup \{\langle sink \rangle\} \cup \{\langle \lightning \rangle\}$ is the set of configurations, $\mathcal{P}$ is given by the rules shown in Figure 5.1 and $\kappa_0 = \langle C, \sigma_0 \rangle$. We can use a reward function to compute the expected runtime of a program. For this we define rew as given in Table 5.1. Then $\widetilde{\mathcal{M}^f_{\sigma_0}}[\![C]\!] = (\mathcal{M}^f_{\sigma_0}[\![C]\!], rew)$ is a Markov reward model and $\text{ExpRew}^{\widetilde{\mathcal{M}^f_{\sigma_0}}[\![C]\!]}(\langle C, \sigma_0 \rangle \vDash \langle sink \rangle)$ is the expected runtime of program C starting in state σ_0 with respect to continuation $f \in \mathbb{T}$. In the following we will write $\text{ExpRew}^{\widetilde{\mathcal{M}^f_{\sigma_0}}[\![C]\!]}(\langle sink \rangle)$ instead of $\text{ExpRew}^{\widetilde{\mathcal{M}^f_{\sigma_0}}[\![C]\!]}(\langle C, \sigma_0 \rangle \vDash \langle sink \rangle)$.

$\kappa \in \mathcal{K}$	$\mathbf{rew}(\kappa)$
$\langle \downarrow, \sigma \rangle$	$f(\sigma)$
$\langle x := e, \sigma \rangle$, $\langle x :=< e >, \sigma \rangle$, $< x := Uniform(u, v), \sigma >$, $\langle < e > := e, \sigma \rangle$, $\langle \textbf{if}\,(b)\,\{c_1\}\,\textbf{else}\,\{c_2\}, \sigma \rangle$, $\langle \{c_1\}\,[p]\,\{c_2\}, \sigma \rangle$	1
$\langle sink \rangle, \langle \lightning \rangle, \langle \downarrow; c_2, \sigma \rangle$, $\langle \textbf{while}\,(b)\,\{c\}, \sigma \rangle$, $\langle \textbf{empty}, \sigma \rangle$	0

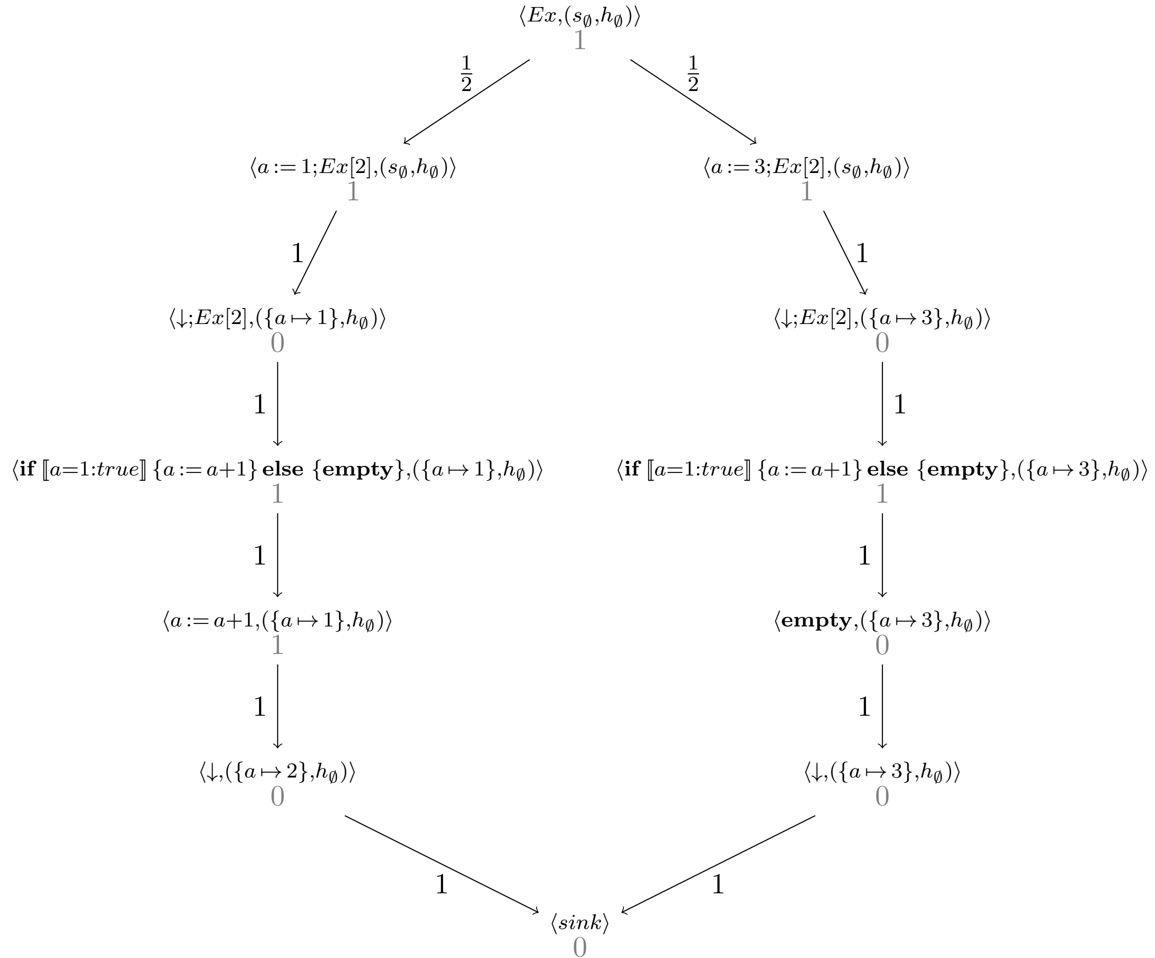
Table 5.1: Definition of reward function

Example 5.0.1. *The following program assigns 1 or 3 to variable a. Then it checks if variable a has value 1 and increments a if so.*

```
1  {a := 1} [1/2] {a := 3};
2  if [[a = 1 : true]] {a := a + 1} else {empty}
```

Listing 5.1: *Ex*

We are going to denote the i-th line as $Ex[i]$*. For an empty starting heap* $h_\emptyset$*, an empty starting stack* $s_\emptyset$ *and continuation* $f = 0$ *the MRM* $\widetilde{\mathcal{M}^0_{(s_\emptyset, h_\emptyset)}}[\![Ex]\!]$ *induces the following graph:*



The expected runtime of program Ex starting in state $(s_\emptyset, h_\emptyset)$ *with respect to continuation* 0 *can then be computed as*

$$
\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}^0_{(s_\emptyset,h_\emptyset)}}\llbracket Ex \rrbracket}(\langle sink \rangle) \\
&= \sum_{\pi \in \Pi(\langle Ex,(s_\emptyset,h_\emptyset)\rangle, \langle sink \rangle)} \mathrm{Pr}^{\mathcal{M}^0_{(s_\emptyset,h_\emptyset)}}(\pi) \cdot rew(\pi) \\
&= \frac{1}{2} \cdot 4 + \frac{1}{2} \cdot 3 \\
&= 3,5
\end{aligned}
$$

$$\frac{}{\langle empty, \sigma\rangle \xrightarrow{1} \langle \downarrow, \sigma\rangle}[\text{empty}] \qquad \frac{}{\langle \downarrow, \sigma\rangle \xrightarrow{1} \langle sink\rangle}[\text{term}]$$

$$\frac{}{\langle sink\rangle \xrightarrow{1} \langle sink\rangle}[\text{sink}] \qquad \frac{}{\langle \lightning\rangle \xrightarrow{1} \langle sink\rangle}[\text{fault}]$$

$$\frac{\langle c_1, \sigma\rangle \xrightarrow{p} \langle c_1', \sigma'\rangle,\ 0 < p \leq 1}{\langle c_1; c_2, \sigma\rangle \xrightarrow{p} \langle c_1'; c_2, \sigma'\rangle}[\text{seq}_1] \qquad \frac{}{\langle \downarrow; c, \sigma\rangle \xrightarrow{1} \langle c, \sigma\rangle}[\text{seq}_2]$$

$$\frac{}{\langle x := e, (s,h)\rangle \xrightarrow{1} \langle \downarrow, (s[x \backslash s(e)], h)\rangle}[\text{assgn}]$$

$$\frac{u \leq i \leq v}{\langle x := Uniform(u,v), (s,h)\rangle \xrightarrow{\frac{1}{v-u+1}} \langle \downarrow, (s[x \backslash i], h)\rangle}[\text{pr-assgn}]$$

$$\frac{[\![b : true]\!](\sigma) = 1}{\langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}, \sigma\rangle \xrightarrow{1} \langle c_1, \sigma\rangle}[\text{if-true}] \qquad \frac{[\![b : false]\!](\sigma) = 1}{\langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}, \sigma\rangle \xrightarrow{1} \langle c_2, \sigma\rangle}[\text{if-false}]$$

$$\frac{}{\langle \{c_1\}\ [p]\ \{c_2\}, \sigma\rangle \xrightarrow{p} \langle c_1, \sigma\rangle}[\text{prob}_1] \qquad \frac{}{\langle \{c_1\}\ [p]\ \{c_2\}, \sigma\rangle \xrightarrow{1-p} \langle c_2, \sigma\rangle}[\text{prob}_2]$$

$$\frac{[e \mapsto _]\bigstar 1(s,h) = 1}{\langle < e > := e', (s,h)\rangle \xrightarrow{1} \langle \downarrow, (s, h[s(e) \backslash s(e')])\rangle}[\text{heap-manip}]$$

$$\frac{1 - [e \mapsto _]\bigstar 1(s,h) = 1}{\langle < e > := e', (s,h)\rangle \xrightarrow{1} \langle \lightning\rangle}[\text{heap-manip-f}]$$

$$\frac{[e \mapsto _]\bigstar 1(s,h) = 1}{\langle x := < e >, (s,h)\rangle \xrightarrow{1} \langle \downarrow, (s[x \backslash h(s(e))], h)\rangle}[\text{heap-lookup}]$$

$$\frac{1 - [e \mapsto _]\bigstar 1(s,h) = 1}{\langle x := < e >, (s,h)\rangle \xrightarrow{1} \langle \lightning\rangle}[\text{heap-lookup-f}]$$

$$\frac{}{\langle \textbf{while}\ (b)\ \{c\}, \sigma\rangle \xrightarrow{1} \langle \textbf{if}\ (b)\ \{c; \textbf{while}\ (b)\ \{c\}\}\ \textbf{else}\ \{\textbf{empty}\}, \sigma\rangle}[\text{while}]$$

Figure 5.1: Rules for transition probability of operational MRMs

Chapter 6

Expected Runtime Calculus

In this thesis we present a sound and complete runtime calculus for the programming language $\mathcal{P}$, which is based on the calculi presented in [3] and [10]. This calculus computes expected runtimes of programs in dependency of their continuation, which is the expected runtime of the following programs. It does so by taking a program as an argument and returning a function. This function itself maps continuations to expected runtimes of the program C.

$$\mathrm{ert} : \mathcal{P}^3 \to (\mathbb{T} \to \mathbb{T})$$

The expected runtime of a program $C \in \mathcal{P}^3$ with continuation $f \in \mathbb{T}$ is written as $\mathrm{ert}[C](f)$. If we want to compute the expected runtime for a program $C \in \mathcal{P}^3$ without any following programs, we write $\mathrm{ert}[C](0)$.

6.1 Definition of ert

The expected runtime calculus ert is defined inductively over the structure of the programming language $\mathcal{P}$ like shown in Table 6.1. At first we will not consider allocation and deallocation of memory and add those rules in Chapter 8.
The two rules for heap manipulation and heap lookup are based on the rules of the *weakest preexpectation calculus* presented in [3]. The rules for the empty program, the sequential composition, the conditional choice and the while loop are identical to the rules presented in [10].

Empty program: Since the execution of an empty program does not consume any time and does not change the program state, $\mathrm{ert}[c](f)$ returns f.
Assignment: An assignment consumes one unit of time and the variable x is assigned the value of expression e. So the stack needs to be updated before continuation f is evaluated. We can express this by using the lambda notation for modifying the arguments of f.
Uniform assignment: A Uniform assignment is a probabilistic assignment. Each possible outcome has the same probability, which means we have to compute the average runtime of the different assignments to get the expected runtime.
Sequential composition: When applying the ert calculus, we are using backwards reasoning, which means we start analyzing a program at the end and work our way up to the beginning. Before we can compute the expected runtime for one program part, we always need to compute the expected runtime for the following program part first to get the continuation. If we want to compute $\mathrm{ert}[c_1; c_2](f)$, we need to compute $\mathrm{ert}[c_2](f)$

first, since this is the continuation of c_1.
Conditional choice: The execution of an if-else statement consumes 1 unit of time for evaluating the guard b. Depending on the evaluation of b either c_1 or c_2 is executed. In the ert calculus we design this by weighting the branches with $[\![b : true]\!]$ and $[\![b : false]\!]$, of which always one evaluates to 1 and the other to 0.
while-loop: As presented in [9], we can deduce the rule for while-loops by expressing them with the conditional choice:

$$\begin{aligned}
&\mathrm{ert}[\mathbf{while}\,(b)\,\{c\}](f)\\
&= \mathrm{ert}[\mathbf{if}\,(b)\,\{\mathbf{while}\,(b)\,\{c\}\}\,\mathbf{else}\,\{\mathbf{empty}\}](f)\\
&= 1 + [\![b : true]\!] \cdot \mathrm{ert}[c; \mathbf{while}\,(b)\,\{c\}](f) + [\![b : false]\!] \cdot \mathrm{ert}[\mathbf{empty}](f)\\
&= 1 + [\![b : true]\!] \cdot \mathrm{ert}[c](\mathrm{ert}[\mathbf{while}\,(b)\,\{c\}](f)) + [\![b : false]\!] \cdot f
\end{aligned}$$

Every result for $\mathrm{ert}[\mathbf{while}(b)\{c\}](f)$ thus has to be a fixedpoint of

$$\Phi : \mathbb{T} \to \mathbb{T}, X \mapsto 1 + [\![b : true]\!] \cdot \mathrm{ert}[c](X) + [\![b : false]\!] \cdot f.$$

We are interested in the least fixedpoint, which is always defined according to Theorem 6.1.1.

Theorem 6.1.1. *Every characteristic function*

$$\Phi : \mathbb{T} \to \mathbb{T}, X \mapsto 1 + [\![b : true]\!] \cdot \mathrm{ert}[c](X) + [\![b : false]\!] \cdot f$$

has a least fixedpoint in $\mathbb{T}$.

Proof. $(\mathbb{T}, \succeq, 0)$ is an ω-cpo (Lemma 3.1) and Φ is ω-continuous, because *ert* is ω-continuous (Lemma 6.3.1). Then the theorem follows with Kleene's Fixedpoint Theorem (Theorem 2.12). □

Probabilistic choice: The decision, which branch will be executed next, consumes one unit of time. Since c_1 is executed with probability p and c_2 with probability $1 - p$, we weight the branches with those probabilities to compute the expected runtime.
Heap Manipulation: Manipulating a heap cell consumes one unit of time. At first we separate the heap cell with address $s(e)$ from the rest of the heap with the help of the allocated pointer predicate and the runtime separating conjunction. Then we add a heap that fulfills the points-to-predicate $[e \mapsto e']$, which means, that it consists of one heap address $s(e)$ that stores value $s(e')$. The continuation f is evaluated on this updated heap.
Heap Lookup: Looking up a value that is stored in a heap cell consumes one unit of time. Since the predicate $[e \mapsto v]$ only resolves to 1, if $h(s(e)) = v$, we use this predicate and the runtime separating conjunction for finding the value that is stored in address $s(e)$. Then we have to add this memory cell to the heap again with the runtime separating implication. The continuation f is evaluated on this updated but unmodified heap and on an updated stack, in which the variable x is set to the value v we looked up.

As we defined ert inductively over the structure of the programming language $\mathcal{P}$ and there always exists a least fixedpoint for the characteristic function of a loop, ert is obviously well-defined and complete.

$\mathbf{C} \in \mathcal{P}^3$	$\mathbf{ert}[\mathbf{C}](f)$ for $f \in \mathbb{T}$
empty	f
$x := e$	$1 + \lambda\,(s,h)_\bullet\, f\,(s\,[x \backslash s(e)]\,,\, h)$
$x := Uniform(u,v)$	$1 + \frac{1}{v-u+1} \cdot \sum_{i=u}^{v} \lambda(s,h)_\bullet f(s[x\backslash i], h)$
$c_1; c_2$	$ert\,[c_1]\,(ert\,[c_2]\,(f))$
if $(b)\,\{c_1\}$ **else** $\{c_2\}$	$1 + [\![b : true]\!] \cdot \text{ert}[c_1](f) + [\![b : false]\!] \cdot \text{ert}[c_2](f)$
while $(b)\,\{c\}$	$\text{lfp X}_\bullet\, 1 + [\![b : false]\!] \cdot f + [\![b : true]\!] \cdot \text{ert}[c](X)$
$\{c_1\}\,[p]\,\{c_2\}$	$1 + p \cdot \text{ert}[c_1](f) + (1-p) \cdot \text{ert}[c_2](f)$
$< e > := e'$	$1 + [e \mapsto _] \bigstar ([e \mapsto e'] \mathrel{-\!\!\bigstar} f)$
$x := < e >$	$1 + \lambda(s,h)_\bullet \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \mathrel{-\!\!\bigstar} f(s[x\backslash v], h))$

Table 6.1: Definition of ert

6.2 Soundness of ert

To prove the soundness of our ert calculus, we will show that

$$\text{ExpRew}^{\widetilde{\mathcal{M}}^f_\sigma[\![C]\!]}(\langle sink \rangle) = \text{ert}[C](f)(\sigma)$$

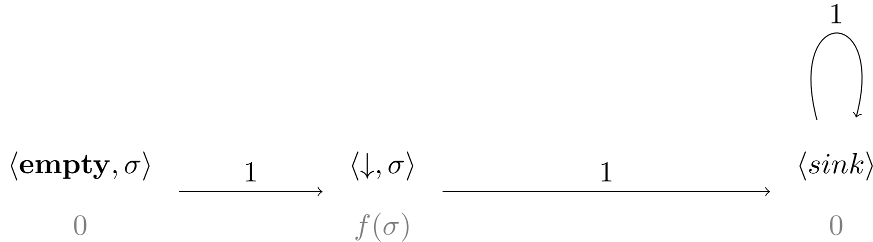
holds for each $C \in \mathcal{P}^3_*$, $\sigma \in \Sigma$ and $f \in \mathbb{T}$. We will prove this inductively over the structure of $C \in \mathcal{P}^3_*$. For this we will give the MRM for each case by showing the induced graph.

Base Case:

For the base case we have to examine the empty program, the assignment, the probabilistic assignment, heap manipulation and heap lookup.

Empty program: C = **empty** [10]

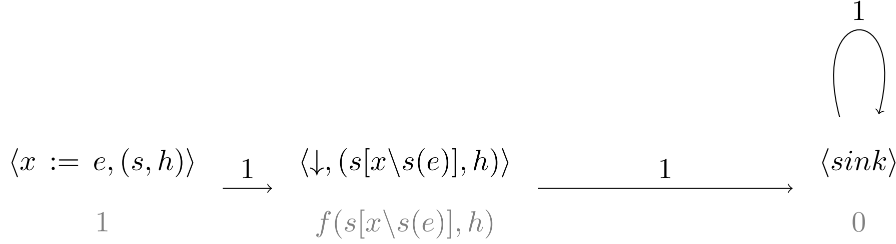
$\widetilde{\mathcal{M}}^f_\sigma[\![\mathbf{empty}]\!]$ contains exactly one infinite path which looks like the following:



Furthermore, $\Pi(\langle \mathbf{empty}, \sigma \rangle, \langle sink \rangle) = \{\langle \mathbf{empty}, \sigma \rangle \langle \downarrow, \sigma \rangle \langle sink \rangle\}$. Therefore we have

$$
\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}^f_\sigma[\![\mathbf{empty}]\!]}(\langle sink\rangle) \\
&= \sum_{\pi\in\Pi(\langle \mathbf{empty},\sigma\rangle,\langle sink\rangle)} Pr^{\widetilde{\mathcal{M}}^f_\sigma[\![\mathbf{empty}]\!]}\{\pi\}\cdot rew(\pi) \\
&= 1\cdot(0+f(\sigma)) \\
&= f(\sigma) \\
&= \mathrm{ert}[\mathbf{empty}](f)(\sigma).
\end{aligned}
$$

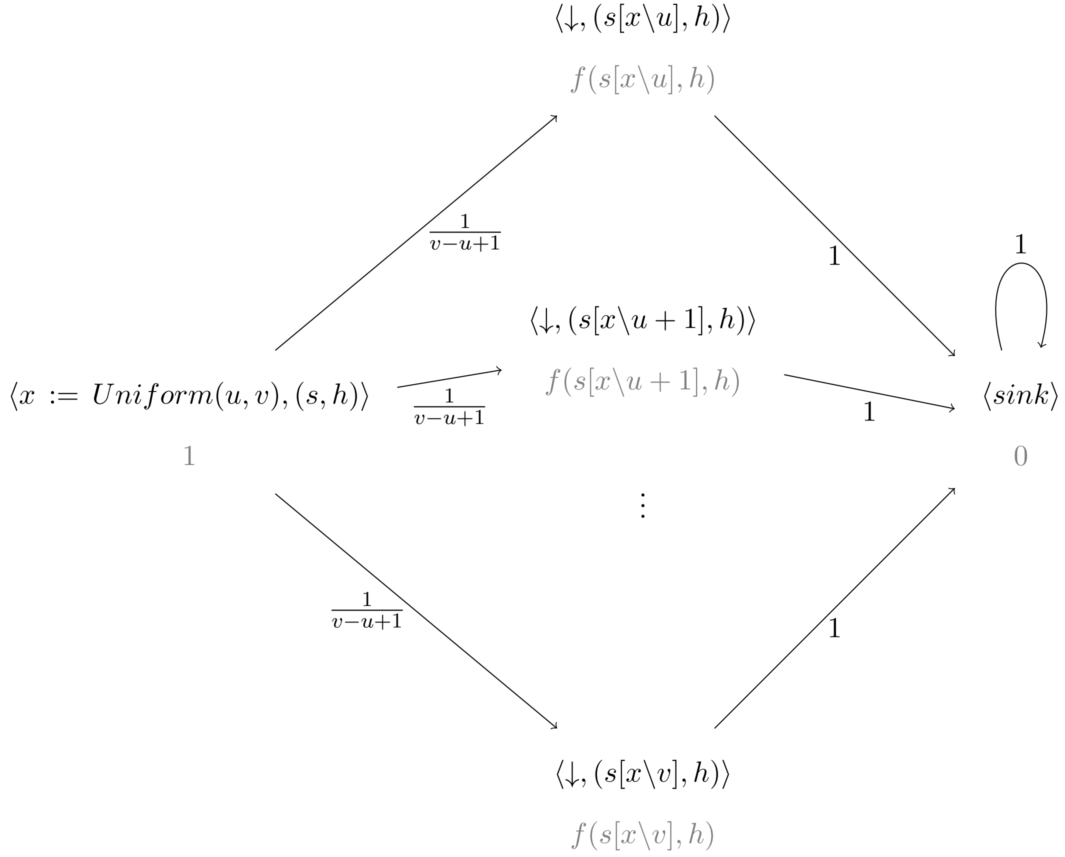
Assignment: $C = x := e$
$\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := e]\!]$ contains exactly one infinite path which looks like the following:



Furthermore, $\Pi(\langle x := e, (s,h)\rangle, \langle sink\rangle) = \{\langle x := e, (s,h)\rangle\langle\downarrow, (s[x\backslash s(e)], h)\rangle\langle sink\rangle\}$. Therefore we have

$$
\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}^f_{(s,h)}[\![x:=e]\!]}(\langle sink\rangle) \\
&= \sum_{\pi\in\Pi(\langle x:=e,(s,h)\rangle,\langle sink\rangle)} Pr^{\widetilde{\mathcal{M}}^f_{(s,h)}[\![x:=e]\!]}\{\pi\}\cdot rew(\pi) \\
&= 1\cdot(1+f(s[x\backslash s(e)],h)) \\
&= 1+f(s[x\backslash s(e)],h) \\
&= 1+\lambda\,(s,h)_\bullet\, f\,(s\,[x\backslash s(e)]\,,\,h)(s,h) \\
&= \mathrm{ert}[x := e](f)(s,h).
\end{aligned}
$$

Probabilistic assignment: $C = x := Uniform(u,v)$
$\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := Uniform(u,v)]\!]$ contains exactly $v-u+1$ infinite paths which look like the following:



Furthermore, $\Pi(\langle x := Uniform(u, v), (s, h)\rangle, \langle sink\rangle)$
$= \{\langle x := Uniform(u, v), (s, h)\rangle\langle\downarrow, (s[x\backslash u], h)\rangle\langle sink\rangle, \ldots,$
$\langle x := Uniform(u, v), (s, h)\rangle\langle\downarrow, (s[x\backslash v], h)\rangle\langle sink\rangle\}$.
Therefore we have

$$\begin{aligned}
&\text{ExpRew}^{\widetilde{\mathcal{M}^f_{(s,h)}}[\![x := Uniform(u,v)]\!]}(\langle sink\rangle)\\
&= \sum_{\pi\in\Pi(\langle x := Uniform(u,v),(s,h)\rangle,\langle sink\rangle)} Pr^{\mathcal{M}^f_{(s,h)}[\![x := Uniform(u,v)]\!]}\{\pi\}\cdot rew(\pi)\\
&= \frac{1}{v-u+1}\cdot\sum_{i=u}^{v} f(s[x\backslash i], h)\\
&= \frac{1}{v-u+1}\cdot\sum_{i=u}^{v} \lambda(s, h)_\bullet f(s[x\backslash i], h)(s, h)\\
&= \text{ert}[x := Uniform(u, v)](f)(s, h).
\end{aligned}$$

Heap manipulation: $C = < e > := e'$
For heap manipulation we need to distinguish between two cases: Either a memory fault occurs or not. Therefore we need two Markov reward models $\widetilde{\mathcal{M}}^f_{(s,h)}[\![< e > := e']\!]_1$ and $\widetilde{\mathcal{M}}^f_{(s,h)}[\![< e > := e']\!]_2$ which both contain one infinite path:

$\widetilde{\mathcal{M}}^f_{(s,h)}[\![< e > := e']\!]_1$:

$$\longrightarrow \underset{1}{\langle < e > := e', (s,h)\rangle} \xrightarrow{1} \underset{f(s,h[s(e)\backslash s(e')])}{\langle \downarrow, (s,h[s(e)\backslash s(e')])\rangle} \xrightarrow{1} \underset{0}{\langle sink \rangle} \circlearrowleft 1$$

$\widetilde{\mathcal{M}}^f_{(s,h)}[\![< e > := e']\!]_2$:

$$\longrightarrow \underset{1}{\langle < e > := e', (s,h)\rangle} \xrightarrow[1]{} \underset{0}{\langle \lightning \rangle} \xrightarrow[1]{} \underset{0}{\langle sink \rangle} \circlearrowleft 1$$

Furthermore,
$\Pi_1(\langle < e > := e', (s,h)\rangle, \langle sink\rangle) = \{\langle < e > := e', (s,h)\rangle \langle \downarrow, (s,h[s(e)\backslash s(e')])\rangle \langle sink\rangle\}$ and
$\Pi_2(\langle < e > := e', (s,h)\rangle, \langle sink\rangle) = \{\langle < e > := e', (s,h)\rangle \langle \lightning \rangle \langle sink\rangle\}$. Therefore we have

$$
\begin{aligned}
&\mathrm{ert}[< e > := e'](f)(s,h) \\
&= 1 + [e \mapsto _] \bigstar ([e \mapsto e'] \mathrel{-\!\!\bigstar} f)(s,h) \\
&= 1 + \max_{h_1,h_2} \{ \underbrace{[e \mapsto _](s,h_1)}_{=0, \text{ if } dom(h_1) \neq \{s(e)\}} \cdot ([e \mapsto e'] \mathrel{-\!\!\bigstar} f)(s,h_2) \mid h_1 \bigstar h_2 = h\} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ [e \mapsto _](s, \{s(e) \mapsto h(s(e))\}) \\ \qquad \cdot ([e \mapsto e'] \mathrel{-\!\!\bigstar} f)(s, h \backslash \{s(e)\}) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ 1 \cdot ([e \mapsto e'] \mathrel{-\!\!\bigstar} f)(s, h\backslash\{s(e)\}) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \sup_{h'} \{ f(s, h\backslash\{s(e)\} \bigstar h') \mid h\backslash\{s(e)\} \perp h' \wedge [e \mapsto e'](s,h') = 1\} & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \sup_{h'} \{ f(s, h\backslash\{s(e)\} \bigstar h') \mid h' = \{s(e) \mapsto s(e')\}\} & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ f(s, h\backslash\{s(e)\} \bigstar \{s(e) \mapsto s(e')\}) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ f(s, h[s(e)\backslash s(e')]) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= \begin{cases} 1 & \text{, if } s(e) \notin dom(h) \\ 1 + f(s, h[s(e)\backslash s(e')]) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= \begin{cases} 1 \cdot 1 & \text{, if } s(e) \notin dom(h) \\ 1 \cdot (1 + f(s, h[s(e)\backslash s(e')])) & \text{, if } s(e) \in dom(h) \end{cases}
\end{aligned}
$$

$$= \begin{cases} \sum_{\pi \in \Pi_2(\langle <e> := e', (s,h)\rangle, \langle sink\rangle)} Pr^{\mathcal{M}^f_{(s,h)}[\![<e> := e']\!]_2}\{\pi\} \cdot rew_2(\pi) & \text{, if } s(e) \in dom(h) \\ \sum_{\pi \in \Pi_1(\langle <e> := e', (s,h)\rangle, \langle sink\rangle)} Pr^{\mathcal{M}^f_{(s,h)}[\![<e> := e']\!]_1}\{\pi\} \cdot rew_1(\pi) & \text{, if } s(e) \notin dom(h) \end{cases}$$

$$= \begin{cases} \text{ExpRew}^{\widetilde{\mathcal{M}}^f_{(s,h)}[\![<e> := e']\!]_2}(\langle sink\rangle) & \text{, if } s(e) \in dom(h) \\ \text{ExpRew}^{\widetilde{\mathcal{M}}^f_{(s,h)}[\![<e> := e']\!]_1}(\langle sink\rangle) & \text{, if } s(e) \notin dom(h) \end{cases}$$

Heap lookup: $C = x := < e >$

Again, we have to distinguish between two cases: The one without any memory faults and the case of a memory fault. Therefore we need the two Markov reward models $\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := < e >]\!]_1$ and $\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := < e >]\!]_2$, which both contain exactly one infinite path.

$\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := < e >]\!]_1$:

$$\longrightarrow \underset{1}{\langle x := < e >, (s,h)\rangle} \xrightarrow{1} \underset{f(s[x\backslash h(s(e))], h)}{\langle \downarrow, (s[x\backslash h(s(e))], h)\rangle} \xrightarrow{1} \underset{0}{\langle sink\rangle} \circlearrowleft 1$$

$\widetilde{\mathcal{M}}^f_{(s,h)}[\![x := < e >]\!]_2$:

$$\longrightarrow \underset{1}{\langle x := < e >, (s,h)\rangle} \xrightarrow{1} \underset{0}{\langle \lightning \rangle} \xrightarrow{1} \underset{0}{\langle sink\rangle} \circlearrowleft 1$$

Furthermore,
$\Pi_1(\langle x := < e >, (s,h)\rangle, \langle sink\rangle) = \{\langle x := < e >, (s,h)\rangle\langle \downarrow, (s[x\backslash h(s(e))], h])\rangle\langle sink\rangle\}$ and $\Pi_2(\langle x := < e >, (s,h)\rangle, \langle sink\rangle) = \{\langle x := < e >, (s,h)\rangle\langle \lightning \rangle\langle sink\rangle\}$. Therefore we have

$$\begin{aligned}
&\text{ert}[x := < e >](f)(s,h) \\
&= 1 + \lambda(s,h)\bullet \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \multimap\!\!\bigstar\ f(s[x\backslash v], h)(s,h) \\
&= 1 + \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \multimap\!\!\bigstar\ f(s[x\backslash v], h) \\
&= 1 + \sup_{v \in \mathbb{Z}} \max_{h_1, h_2} \{ \underbrace{[e \mapsto v](s, h_1)}_{=0, \text{ if } dom(h_1) \neq \{s(e)\}} \cdot ([e \mapsto v] \multimap\!\!\bigstar\ f(s[x\backslash v], h))(s, h_2) | h_1 \bigstar h_2 = h\} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \sup_{v \in \mathbb{Z}} \overbrace{[e \mapsto v](s, \{s(e) \mapsto h(s(e))\})}^{\begin{cases} 1, \text{ if } v = h(s(e)) \\ 0, \text{ otherwise} \end{cases}} \\ \quad \cdot ([e \mapsto v] \multimap\!\!\bigstar\ f(s[x\backslash v], h))(s, h\backslash\{s(e)\}) & \text{, if } s(e) \in dom(h) \end{cases} \\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \max\{0, 1 \cdot ([e \mapsto h(s(e))] \\ \quad \multimap\!\!\bigstar\ f(s[x\backslash h(s(e))], h))(s, h\backslash\{s(e)\})\} & \text{, if } s(e) \in dom(h) \end{cases}
\end{aligned}$$

$$= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \max\{0, \sup_{h'}\{f(s[x \backslash h(s(e))], h \backslash \{s(e)\} \bigstar h') \mid \\ \quad h \backslash \{s(e)\} \perp h' \wedge [e \mapsto h(s(e))](s, h') = 1\}\} & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \max\{0, \sup_{h'}\{f(s[x \backslash h(s(e))], h \backslash \{s(e)\} \bigstar h') \mid \\ \quad h' = \{s(e) \mapsto h(s(e))\}\}\} & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \max\{0, \sup_{h'}\{f(s[x \backslash h(s(e))], h \backslash \{s(e)\} \bigstar \{s(e) \mapsto h(s(e))\})\}\} & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \max\{0, f(s[x \backslash h(s(e))], h)\} & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= \begin{cases} 1 & \text{, if } s(e) \notin dom(h) \\ 1 + f(s[x \backslash h(s(e))], h) & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= \begin{cases} 1 \cdot 1 & \text{, if } s(e) \notin dom(h) \\ 1 \cdot (1 + f(s[x \backslash h(s(e))], h)) & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= \begin{cases} \sum\limits_{\pi \in \Pi_2(\langle x := <e>, (s,h) \rangle, \langle sink \rangle)} Pr^{\mathcal{M}^f_{(s,h)} [\![x := <e>]\!]_2}\{\pi\} \cdot rew(\pi) & \text{, if } s(e) \notin dom(h) \\ \sum\limits_{\pi \in \Pi_1(\langle x := <e>, (s,h) \rangle, \langle sink \rangle)} Pr^{\mathcal{M}^f_{(s,h)} [\![x := <e>]\!]_1}\{\pi\} \cdot rew(\pi) & \text{, if } s(e) \in dom(h) \end{cases}$$

$$= \begin{cases} \text{ExpRew}^{\widetilde{\mathcal{M}}^f_{(s,h)} [\![x := <e>]\!]_2}(\langle sink \rangle) & \text{, if } s(e) \notin dom(h) \\ \text{ExpRew}^{\widetilde{\mathcal{M}}^f_{(s,h)} [\![x := <e>]\!]_1}(\langle sink \rangle) & \text{, if } s(e) \in dom(h) \end{cases}$$

Induction Hypothesis:
All subprograms $C' \in \mathcal{P}^3_*$ of C fulfill

$$\text{ExpRew}^{\widetilde{\mathcal{M}}^f_{\sigma_0} [\![C']\!]}(\langle sink \rangle) = \text{ert}[C'](f)(\sigma_0)$$

for all $\sigma \in \Sigma$ and all $f \in \mathbb{T}$.
Inductive Step:
For the inductive step we need to examine sequential composition, conditional choice, probabilistic choice and loops.
Sequential Composition: $C = c_1; c_2$ [10]
For this proof, we will use the fact, that

$$\text{ExpRew}^{\widetilde{\mathcal{M}}^f_{\sigma} [\![c_1; c_2]\!]}(\langle sink \rangle) = \text{ExpRew}^{\widetilde{\mathcal{M}}^{g(c_2,f)}_{\sigma} [\![c_1]\!]}(\langle sink \rangle) \tag{6.1}$$

where

$$g(c_2, f) = \text{ExpRew}^{\lambda \rho \bullet \widetilde{\mathcal{M}}^f_{\rho} [\![c_2]\!]}(\langle sink \rangle).$$

This statement is proven for MDPs in [10] on page 42. Since MRMs are special MDPs this statement holds automatically. Now we can show the following equality presented

in [10]:

$$
\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_1;c_2]\!]}(\langle sink\rangle) && \\
=&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}_{\sigma}^{\mathrm{ExpRew}^{\lambda\rho\bullet\widetilde{\mathcal{M}}^{f}_{\rho}[\![c_2]\!]}(\langle sink\rangle)}[\![c_1]\!]}(\langle sink\rangle) && (1)\\
=&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}_{\sigma}^{\lambda\rho\bullet\mathrm{ert}[c_2](f)(\rho)}[\![c_1]\!]}(\langle sink\rangle) && (\text{I.H. on } c_2)\\
=&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}_{\sigma}^{\mathrm{ert}[c_2](f)}[\![c_1]\!]}(\langle sink\rangle) && \\
=&\mathrm{ert}[c_1](\mathrm{ert}(c_2)(f))(\sigma) && (\text{I.H. on } c_1)\\
=&\mathrm{ert}[c_1;c_2](f)(\sigma). &&
\end{aligned}
$$

Conditional Choice: $C = \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}$ [10]
The induced graph of the MRM $\widetilde{\mathcal{M}}^{f}_{\sigma}[\![\textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}]\!]$ looks like this:

$\longrightarrow \langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}, \sigma\rangle$ (1)
$\xrightarrow{[\![b : true]\!]}$ $\langle c_1, \sigma\rangle$ $rew(\langle c_1, \sigma\rangle)$ $\rightsquigarrow$
$\xrightarrow{[\![b : false]\!]}$ $\langle c_2, \sigma\rangle$ $rew(\langle c_2, \sigma\rangle)$ $\rightsquigarrow$

Since $\langle c_1, \sigma\rangle$ and $\langle c_2, \sigma\rangle$ are the initial configurations for the MRMs $\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_1]\!]$ and $\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_2]\!]$, we can conclude

$$
\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}^{f}_{\sigma}[\![\textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}]\!]}(\langle sink\rangle) &&\\
=&1 + [\![b : true]\!](\sigma)\cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_1]\!]}(\langle sink\rangle) + [\![b : false]\!](\sigma)\cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_2]\!]}(\langle sink\rangle) &&\\
=&1 + [\![b : true]\!](\sigma)\cdot \mathrm{ert}[c_1](f)(\sigma) + [\![b : false]\!](\sigma)\cdot \mathrm{ert}[c_2](f)(\sigma) && (\text{I.H.})\\
=&\mathrm{ert}[\textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}](f)(\sigma). &&
\end{aligned}
$$

Probabilistic choice: $C = \{c_1\}\ [p]\ \{c_2\}$
$\widetilde{\mathcal{M}}^{f}_{(s,h)}[\![\{c_1\}\ [p]\ \{c_2\}]\!]$ contains at least 2 infinite paths which look like the following:

$\longrightarrow \langle \{c_1\}\ [p]\ \{c_2\}, \sigma\rangle$ (1)
$\xrightarrow{p}$ $\langle c_1, \sigma\rangle$ $rew(\langle c_1, \sigma\rangle)$ $\rightsquigarrow$
$\xrightarrow{1\text{-}p}$ $\langle c_2, \sigma\rangle$ $rew(\langle c_2, \sigma\rangle)$ $\rightsquigarrow$

$\langle c_1, \sigma\rangle$ and $\langle c_2, \sigma\rangle$ are the initial configurations for the MRMs $\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_1]\!]$ and $\widetilde{\mathcal{M}}^{f}_{\sigma}[\![c_2]\!]$. There-

fore we can conclude

$$\begin{aligned}
&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![\{c_1\}\,[p]\,\{c_2\}]\!]}(\langle sink\rangle)\\
&= 1 + p \cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![c_1]\!]}(\langle sink\rangle) + (1-p)\cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![c_2]\!]}(\langle sink\rangle)\\
&= 1 + p \cdot \mathrm{ert}[c_1](f)(\sigma) + (1-p)\cdot \mathrm{ert}[c_2](f)(\sigma) && \text{(I.H.)}\\
&= \mathrm{ert}[\{c_1\}\,[p]\,\{c_2\}](f)(\sigma).
\end{aligned}$$

Loop: $C = \textbf{while}\,(b)\,\{c'\}$ [10]

Before we can prove the soundness of the rule for loops, we have to make a few statements about loops. For this we introduce bounded loops:

Definition 6.2.1 (Bounded loops [10]). *If b is a boolean expression, $C \in \mathcal{P}^3$, $f \in \mathbb{T}$ and $k \in \mathbb{N}$, then*

$$\begin{aligned}
\textbf{while}^{<0}\,(b)\,\{C\} &= \textbf{empty} \text{ and}\\
\textbf{while}^{<k+1}\,(b)\,\{C\} &= \textbf{if}(b)\{C;\textbf{while}^{<k}(b)\{C\}\}\,\textbf{else}\,\{\textbf{empty}\}.
\end{aligned}$$

For a boolean expression b, a program $C \in \mathcal{P}^3$, a continuation $f \in \mathbb{T}$ and a program state $\sigma \in \Sigma$ the following equations hold:

$$\sup_{k\in\mathbb{N}} \mathrm{ert}[\textbf{while}^{<k}\,(b)\,\{C\}](f) = \mathrm{ert}[\textbf{while}\,(b)\,\{C\}](f) \tag{6.2}$$

and

$$\sup_{k\in\mathbb{N}} \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![\textbf{while}^{<k}\,(b)\,\{C\}]\!]}(\langle sink\rangle) = \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![\textbf{while}\,(b)\,\{C\}]\!]}(\langle sink\rangle). \tag{6.3}$$

The proof for (6.2) can be found on page 41 of [10] and the proof for (6.3) on page 43-44 of [10]. Both of the proofs work with MDPs. Since MRMs are special MDPs, those statements also hold for MRMs. For an arbitrary $k \in \mathbb{N}_{\geq 1}$ and $\sigma \in \Sigma$ we can additionally show this statement from [10]:

$$\begin{aligned}
&\mathrm{ert}[\textbf{while}^{<k}\,(b)\,\{c'\}]\\
=&\mathrm{ert}[\textbf{if}\,(b)\,\{c';\textbf{while}^{<k-1}\,(b)\,\{c'\}\}\,\textbf{else}\,\{\textbf{empty}\}](f)(\sigma)\\
=&1 + [\![b:true]\!](\sigma)\cdot \mathrm{ert}[c';\textbf{while}^{<k-1}\,(b)\,\{c'\}](f)(\sigma) + [\![b:false]\!](\sigma)\cdot \mathrm{ert}[\textbf{empty}](f)(\sigma)\\
=&1 + [\![b:true]\!](\sigma)\cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![c';\textbf{while}^{<k-1}\,(b)\,\{c'\}]\!]}(\langle sink\rangle) && \text{(I.H.)}\\
&+ [\![b:false]\!](\sigma)\cdot \mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![\textbf{empty}]\!]}(\langle sink\rangle)\\
=&\mathrm{ExpRew}^{\widetilde{\mathcal{M}}_\sigma^f[\![\textbf{while}^{<k}\,(b)\,\{c'\}]\!]}(\langle sink\rangle)
\end{aligned}$$

This also holds for $k = 0$, since

$$\begin{aligned}
&\mathrm{ert}[\mathrm{while}^{<0}\,(b)\,\{c'\}]\\
=&\mathrm{ert}[\textbf{empty}]\\
=&\mathrm{ExpRew}^{\mathcal{M}_\sigma^f[\![\textbf{empty}]\!]}(\langle sink\rangle)\\
=&\mathrm{ExpRew}^{\mathcal{M}_\sigma^f[\![\textbf{while}^{<0}\,(b)\,\{c'\}]\!]}(\langle sink\rangle)
\end{aligned}$$

Finally, we can prove the soundness of the loop rule:

$$
\begin{aligned}
&\mathrm{ert}[\mathrm{while}\,(b)\,\{c'\}](f)(\sigma) \\
=&\sup_{k\in\mathbb{N}} \mathrm{ert}[\mathrm{while}^{<k}\,(b)\,\{c'\}](f)(\sigma) && (6.2)\\
=&\sup_{k\in\mathbb{N}} \mathrm{ExpRew}^{\mathcal{M}_\sigma^f[\![\mathbf{while}^{<k}\,(b)\,\{c'\}]\!]}(\langle sink\rangle) \\
=&\mathrm{ExpRew}^{\mathcal{M}_\sigma^f[\![\mathbf{while}\,(b)\,\{c'\}]\!]}(\langle sink\rangle) && (6.3)
\end{aligned}
$$

6.3 Properties of ert

Lemma 6.3.1. *ert is ω-continuous.*

Proof. We proof this by induction over the structure of program $C \in \mathcal{P}^3$. The details can be found in 10.0.1 . □

Lemma 6.3.2. *ert is monotone.*

Proof. Since ert is ω-continuous, it is by definition also monotone. □

6.4 Loops

The hardest task while analyzing the expected runtime of a program is most of the time analyzing the loops in a program. Finding the needed fixedpoint can be very complicated and the exact fixedpoint can be a very complex formula. Due to this difficulty we will use two proof rules presented in [9], that help us at least approximating the needed fixedpoint. We are searching for a least fixedpoint of

$$\Phi : \mathbb{T} \to \mathbb{T}, X \mapsto 1 + [\![b : \mathrm{true}]\!] \cdot \mathrm{ert}[c](X) + [\![b : \mathrm{false}]\!] \cdot f,$$

which is the characteristic function of $\mathbf{while}\,(b)\,\{c\}$. The two following rules work with fixed point iteration, where we under- respectively over-approximate each step of the fixed point iteration with parametrized invariants. Let $I_n \in \mathbb{T}$ be parameterized in a natural number n.

Definition 6.4.1 (Lower ω-invariant [9]). *I_n is a lower ω-invariant of* $\boldsymbol{while}\,(b)\,\{c\}$ *exactly when*

$$I_1 \le \Phi(0) \text{ and } \forall\, n \ge 1 : I_{n+1} \le \Phi(I_n)$$

Upper ω-invariants are defined analogously:

Definition 6.4.2 (Upper ω-invariant [9]). *I_n is a upper ω-invariant of* $\boldsymbol{while}\,(b)\,\{c\}$ *exactly when*

$$I_1 \ge \Phi(0) \text{ and } \forall\, n \ge 1 : I_{n+1} \ge \Phi(I_n)$$

Theorem 6.4.3. *If I_n is a lower ω-invariant of* $\boldsymbol{while}\,(b)\,\{c\}$ *with respect to continuation $f \in \mathbb{T}$ and the limit of I_n exists then*

$$\lim_{n\to\infty} I_n \le \mathrm{ert}[\mathbf{while}\,(b)\,\{c\}](f).$$

*If I_n is an upper ω-invariant of **while** $(b)\,\{c\}$ with respect to continuation $f \in \mathbb{T}$ and the limit of I_n exists then*

$$\lim_{n\to\infty} I_n \geq \text{ert}[\textbf{while}\,(b)\,\{c\}](f).$$

Proof. The proof can be found in [10]. □

6.5 Examples

Now that we have proven the soundness of ert, we will apply the calculus to a few examples. At first we will take a look at an array, which starts at address a and has length n. This means that the array entries are at the addresses a, a+1, a+2, ... , a+n-1.

6.5.1 Switch

The first program *Switch* will exchange the content of the two addresses a+i and a+j in the array a.

```
1  x :=< a + i >;
2  y :=< a + j >;
3  < a + i >:= y;
4  < a + j >:= x
```

Listing 6.1: *Switch*

Since we are interested in the expected runtime of *Switch* without any following code, the post-runtime is 0 and we have to compute $ert[Switch](0)$. In order to improve readability we will refer to the i-th line of *Switch* as $Switch[i]$.

$$\begin{aligned}
&\text{ert}[Switch](0)\\
&= \text{ert}[Switch[1-3](\text{ert}[< a+j >:= x](0))\\
&= \text{ert}[Switch[1-3](1 + [a+j \mapsto _] \bigstar ([a+j \mapsto x] \mathrel{-\!\!\bigstar} 0))\\
&= \text{ert}[Switch[1-3](1 + [a+j \mapsto _] \bigstar 0)\\
&= \text{ert}[Switch[1-3](1+0)\\
&= \text{ert}[Switch[1-2](\text{ert}[< a+i >:= y](1))\\
&= \text{ert}[Switch[1-2](1 + [a+i \mapsto _] \bigstar ([a+i \mapsto y] \mathrel{-\!\!\bigstar} 1))\\
&= \text{ert}[Switch[1]](\text{ert}[y :=< a+j >](1 + [a+i \mapsto _] \bigstar ([a+i \mapsto y] \mathrel{-\!\!\bigstar} 1)))\\
&= \text{ert}[x :=< a+i >](1 + \sup_{v\in\mathbb{Z}} [a+j \mapsto v] \bigstar\\
&\qquad ([a+j \mapsto v] \mathrel{-\!\!\bigstar} (1 + [a+i \mapsto _] \bigstar ([a+i \mapsto v] \mathrel{-\!\!\bigstar} 1))))\\
&= 1 + \sup_{w\in\mathbb{Z}} [a+i \mapsto w] \bigstar ([a+i \mapsto w] \mathrel{-\!\!\bigstar} (1 + \sup_{v\in\mathbb{Z}} [a+j \mapsto v] \bigstar\\
&\qquad ([a+j \mapsto v] \mathrel{-\!\!\bigstar} (1 + [a+i \mapsto _] \bigstar ([a+i \mapsto v] \mathrel{-\!\!\bigstar} 1)))))\\
&= \begin{cases} 1 & \text{, if } a+i \notin dom(h)\\ 2 & \text{, if } a+i \in dom(h) \text{ and } a+j \notin dom(h)\\ 4 & \text{, otherwise}\end{cases}\\
&\leq 4
\end{aligned}$$

4 units of time is an upper bound for the expected runtime of *Switch*.
Because this notation of the analysis can look quite confusing even for short and simple programs, will we use a more intuitive notation from [9].

$\backslash\!\!\backslash \mathrm{ert}[C](f)$
C
$\backslash\!\!\backslash f$

The continuation is written below the program code. The lines of code are written on gray background and the expected runtime with respect to the continuation is written above the line of code. This will be the continuation for the next line of code. This notation shows that we are reasoning backwards. If two lines of analysis are next to each other without a line of code inbetween, we simplified the formula:

$\backslash\!\!\backslash 2$
$\backslash\!\!\backslash 1 + 1.$

If we use \\ instead of \\ we estimated an upper bound for the expected runtime:

$\backslash\!\!\backslash 3$
$\backslash\!\!\backslash [\![b : true]\!] \cdot 2 + 1.$

We are going to analyze the program *Switch* again with the new notation. This time we will estimate an upper bound and not calculate the exact expected runtime to maintain readability

	$\backslash\!\!\backslash 4$ $\backslash\!\!\backslash 1 + \sup_{v \in \mathbb{Z}} [a+i \mapsto v] \bigstar ([a+i \mapsto v] \mathrel{-\!\!\bigstar} 3)$
1	$x := < a + i >;$
	$\backslash\!\!\backslash 3$ $\backslash\!\!\backslash 1 + \sup_{v \in \mathbb{Z}} [a+j \mapsto v] \bigstar ([a+j \mapsto v] \mathrel{-\!\!\bigstar} 2)$
2	$y := < a + j >;$
	$\backslash\!\!\backslash 2$ $\backslash\!\!\backslash 1 + [a+i \mapsto _] \bigstar ([a+i \mapsto y] \mathrel{-\!\!\bigstar} 1)$
3	$< a + i > := y;$
	$\backslash\!\!\backslash 1$ $\backslash\!\!\backslash 1 + [a+j \mapsto _] \bigstar ([a+j \mapsto x] \mathrel{-\!\!\bigstar} 0)$
4	$< a + j > := x$
	$\backslash\!\!\backslash 0$

Listing 6.2: Analysis of *Switch*

6.5.2 Partitioning

The Program *Part* also works with an array. It places all elements smaller or equal to a pivot element, which is given by index i, left of the pivot element and the others on the right side of the array. This procedure is used in the quicksort algorithm in every step of the recursion.

```
1  l := a;
2  r := a + n − 1;
3  p := < i >;
4  temp_1 := < r >;
5  < i > := temp_1;
6  < r > := p;
7  i := l − 1;
8  j := l;
9  while(j < r){
10     x := < j >;
11     if(x ≤ p){
12         i := i + 1;
13         temp_2 := < i >;
14         < i > := x;
15         < j > := temp_2
16     }
17     else{
18         empty
19     };
20     j := j + 1;
21 }
22 temp_3 := < i + 1 >;
23 < i + 1 > := p;
24 < r > := temp_3
```

Listing 6.3: *Part*

Since computing the exact expected runtime is quite costly even for small examples, we will now compute an upper bound of the expected runtime of *Part*. Therefore we assume, that all cells in the array a are allocated. If they were not, the program would stop earlier with a fault and the runtime would be shorter. Additionally we assume, that a has a length of $n \geq 1$.
Since we are analyzing the program backwards, we are starting with the last three lines:

```
   \\3
   \\1 + sup_{v∈ℤ} [i + 1 ↦ v]★([i + 1 ↦ v] −★ 2)
22 temp_3 := < i + 1 >;
   \\2
   \\1 + [i + 1 ↦ _]★([i + 1 ↦ p] −★ 1)
23 < i + 1 > := p;
   \\1
   \\1 + [r ↦ _]★([r ↦ temp_3] −★ 0)
24 < r > := temp_3
   \\0
```

Listing 6.4: Analyzing $Part[22 - 24]$

To analyze the while loop of *Part* we need to find a least fixed point for the characteristic function

$$\Phi : \mathbb{T} \to \mathbb{T}, X \mapsto 1 + \llbracket j < r : true \rrbracket \cdot \mathrm{ert}[Part[10{-}20]](X) + \llbracket j < r : false \rrbracket \cdot \mathrm{ert}[Part[22{-}24]](0).$$

Before searching the fixed point we need to analyze $Part[10-20]$ for an arbitrary continuation X:

$$\backslash\!\!\backslash 1 + \sup_{v \in \mathbb{Z}} [j \mapsto v] \bigstar ([j \mapsto v] \mathbin{-\!\!\bigstar} (1 + \llbracket v \leq p : true \rrbracket \cdot (2 + \sup_{w \in \mathbb{Z}} [i+1 \mapsto w] \bigstar ([i+1 \mapsto w]$$
$$\mathbin{-\!\!\bigstar} (1 + [i+1 \mapsto _] \bigstar ([i+1 \mapsto x] \mathbin{-\!\!\bigstar} (1 + [j \mapsto _] \bigstar ([j \mapsto w]$$
$$\mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1, temp_2 \backslash w, i \backslash i+1, x \backslash v], h))))))))$$
$$+ \llbracket v \leq p : false \rrbracket \cdot (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1, x \backslash v], h))))$$

10 $x :=\ < j >;$

$$\backslash\!\!\backslash 1 + \llbracket x \leq p : true \rrbracket \cdot (2 + \sup_{w \in \mathbb{Z}} [i+1 \mapsto w] \bigstar ([i+1 \mapsto w] \mathbin{-\!\!\bigstar} (1 + [i+1 \mapsto _] \bigstar ([i+1 \mapsto x]$$
$$\mathbin{-\!\!\bigstar} (1 + [j \mapsto _] \bigstar ([j \mapsto w] \mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1, temp_2 \backslash w, i \backslash i+1], h))))))))$$
$$+ \llbracket x \leq p : false \rrbracket \cdot (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h))$$

11 **if** $(x \leq p)\{$

$$\backslash\!\!\backslash 2 + \sup_{w \in \mathbb{Z}} [i+1 \mapsto w] \bigstar ([i+1 \mapsto w] \mathbin{-\!\!\bigstar} (1 + [i+1 \mapsto _] \bigstar ([i+1 \mapsto x] \mathbin{-\!\!\bigstar}$$
$$(1 + [j \mapsto _] \bigstar ([j \mapsto w] \mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1, temp_2 \backslash w, i \backslash i+1], h)))))))$$

12 $i := i + 1;$

$$\backslash\!\!\backslash 1 + \sup_{w \in \mathbb{Z}} [i \mapsto w] \bigstar ([i \mapsto w] \mathbin{-\!\!\bigstar} (1 + [i \mapsto _] \bigstar ([i \mapsto x] \mathbin{-\!\!\bigstar}$$
$$(1 + [j \mapsto _] \bigstar ([j \mapsto w] \mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1, temp_2 \backslash w], h)))))))$$

13 $temp_2 :=\ < i >;$

$$\backslash\!\!\backslash 1 + [i \mapsto _] \bigstar ([i \mapsto x] \mathbin{-\!\!\bigstar} (1 + [j \mapsto _] \bigstar ([j \mapsto temp_2]$$
$$\mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)))))$$

14 $< i > := x;$

$$\backslash\!\!\backslash 1 + [j \mapsto _] \bigstar ([j \mapsto temp_2] \mathbin{-\!\!\bigstar} (1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)))$$

15 $< j > := temp_2$

$$\backslash\!\!\backslash 1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)$$

16 $\}$

17 **else** $\{$

$$\backslash\!\!\backslash 1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)$$

18 **empty**

$$\backslash\!\!\backslash 1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)$$

19 $\};$

$$\backslash\!\!\backslash 1 + \lambda(s,h)_{\bullet} X(s[j \backslash j+1], h)$$

20 $j := j + 1;$

$$\backslash\!\!\backslash X$$

Listing 6.5: Analyzing loop body of *Part*

We suggest

$$I_n = 4 + \sum_{k=0}^{n-1} 8 \cdot [\![j + k < r : true]\!]$$

as an upper ω-invariant for the given while-loop, because $f = ert[Part[22-24]](0) \leq 3$ and one execution of the loop body consumes maximal 7 units of time. By adding one unit of time for the evaluation of $j < r$ we get an amount of 4 units of time, if $j < r$ is true and 8 units of time otherwise. By using $[\![j + k < r : true]\!]$ as a factor in the sum, we take in consideration that j is incremented in every execution of the while-loop.

Proof.

$$\begin{aligned}
\Phi(0) &= 1 + [\![j < r : true]\!] \cdot ert[Part[10-20]](0) + [\![j < r : false]\!] \cdot \mathrm{ert}[Part[22-24]](0) \\
&\leq 1 + [\![j < r : true]\!] \cdot (1 + \sup_{v \in \mathbb{Z}} [j \mapsto v] \bigstar ([j \mapsto v] \mathrel{-\!\!\bigstar} (1 + [\![v \leq p : true]\!] \cdot \\
&\quad (2 + \sup_{w \in \mathbb{Z}} [i+1 \mapsto w] \bigstar ([i+1 \mapsto w] \mathrel{-\!\!\bigstar} (1 + [i+1 \mapsto _] \bigstar ([i+1 \mapsto x] \\
&\quad \mathrel{-\!\!\bigstar} (1 + [j \mapsto _] \bigstar ([j \mapsto w] \mathrel{-\!\!\bigstar} (1 + 0))))))) + [\![v \leq p : false]\!] \cdot (1 + 0)))) \\
&\quad + [\![j < r : false]\!] \cdot 3 \\
&\leq 1 + [\![j < r : true]\!] \cdot (1 + \sup_{v \in \mathbb{Z}} [j \mapsto v] \bigstar ([j \mapsto v] \mathrel{-\!\!\bigstar} \\
&\quad (1 + [\![v \leq p : true]\!] \cdot 5 + [\![v \leq p : false]\!] \cdot 1))) + [\![j < r : false]\!] \cdot 3 \\
&\leq \begin{cases} 8 \text{ , if } j < r \\ 4 \text{ , if } j \geq r \end{cases} \\
&\leq 4 + 8 \cdot [\![j < r : true]\!] \\
&= I_1
\end{aligned}$$

$$\Rightarrow \Phi(0) \leq I_1$$

$$\begin{aligned}
\Phi(I_n) &= 1 + [\![j < r : true]\!] \cdot ert[Part[10-20]](4 + 8n) + [\![j < r : false]\!] \cdot Part[22-24] \\
&\leq 1 + [\![j < r : true]\!] \cdot (1 + \sup_{v \in \mathbb{Z}} [j \mapsto v] \bigstar ([j \mapsto v] \mathrel{-\!\!\bigstar} \quad (1 + [\![v \leq p : true]\!] \cdot \\
&\quad (5 + 4 + \sum_{k=0}^{n-1} 8 \cdot [\![j + 1 + k < r : true]\!]) + [\![v \leq p : false]\!] \cdot \\
&\quad (1 + 4 + \sum_{k=0}^{n-1} 8 \cdot [\![j + 1 + k < r : true]\!])))) + [\![j < r : false]\!] \cdot 3 \\
&\leq \begin{cases} 12 + \sum_{k=0}^{n-1} 8 \cdot [\![j + 1 + k < r : true]\!] & \text{, if } j < r \\ 4 & \text{, if } j \geq r \end{cases} \\
&= \begin{cases} 12 + \sum_{k=1}^{n} 8 \cdot [\![j + k < r : true]\!] & \text{, if } j < r \\ 4 & \text{, if } j \geq r \end{cases}
\end{aligned}$$

$$= \begin{cases} 4 + \sum_{k=0}^{n} 8 \cdot [\![j+k < r : true]\!] & \text{, if } j < r \\ 4 & \text{, if } j \geq r \end{cases}$$

$$= I_{n+1}$$

$$\Rightarrow \Phi(I_n) \leq I_{n+1}$$

As $\Phi(0) \leq I_1$ and $\Phi(I_n) \leq I_{n+1}$ holds for all $n \geq 1$, I_n is an upper ω-invariant for the while loop by definition 6.4.2. □

As I_n is an upperω-invariant of the while-loop, according to Lemma 6.4.2 we can conclude

$$\begin{aligned}
&\text{ert}[\textbf{while}(j < r)\{Part[10-20]\}](Part[22-24](0)) \\
&\leq \lim_{n\to\infty} I_n \\
&= 4 + \sum_{k=0}^{\infty} 8 \cdot [\![j+k < r : true]\!] \\
&= 4 + \sum_{k=0}^{r-j-1} 8 \cdot [\![j+k < r : true]\!] \\
&= 4 + 8 \cdot (r - j).
\end{aligned}$$

Now we are able to find an upper bound for $\text{ert}[Part](0)$:

$\backslash\backslash 4 + 8 \cdot n$
$\backslash\backslash 12 + 8 \cdot (n-1)$
$\backslash\backslash 12 + 8 \cdot (a + n - 1 - a)$

```
1  l := a;
```

$\backslash\backslash 11 + 8 \cdot (a + n - 1 - l)$

```
2  r := a + n − 1;
```

$\backslash\backslash 10 + 8 \cdot (r - l)$
$\backslash\backslash 1 + \sup_{v\in\mathbb{Z}} [i \mapsto v] \bigstar ([i \mapsto v] -\!\!\bigstar\ (9 + 8 \cdot (r - l)))$

```
3  p :=< i >;
```

$\backslash\backslash 9 + 8 \cdot (r - l)$
$\backslash\backslash 1 + \sup_{v\in\mathbb{Z}} [r \mapsto v] \bigstar ([r \mapsto v] -\!\!\bigstar\ (8 + 8 \cdot (r - l)))$

```
4  temp1 :=< r >;
```

$\backslash\backslash 8 + 8 \cdot (r - l)$
$\backslash\backslash 1 + [i \mapsto _] \bigstar ([i \mapsto temp_1] -\!\!\bigstar\ (7 + 8 \cdot (r - l)))$

```
5  < i >:= temp1;
```

$\backslash\backslash 7 + 8 \cdot (r - l)$
$\backslash\backslash 1 + [r \mapsto _] \bigstar ([r \mapsto p] -\!\!\bigstar\ (6 + 8 \cdot (r - l)))$

```
6  < r >:= p;
```

$\backslash\backslash 6 + 8 \cdot (r - l)$

```
   \\6 + 8 · (r − l)
7  i := l − 1;
   \\5 + 8 · (r − l)
8  j := l;
   \\4 + 8 · (r − j)
9  while (j < r) {
10     x :=< j >;
11     if (x ≤ p) {
12         i := i + 1;
13         temp2 :=< i >;
14         < i >:= x;
15         < j >:= temp2
16     }
17     else {
18         empty
19     };
20     j := j + 1;
21 }
   \\3
22 temp3 :=< i + 1 >;
23 < i + 1 >:= p;
24 < r >:= temp3
   \\0
```

Listing 6.6: Computation of an upper bound for *Part*

As we can see, the expected runtime of the Partitioning program depends linearly on the length n of the given array.

6.5.3 Dice

The next program *Dice* simulates a game of dice with two players. In each round both players roll their dice. Player 1 uses a dice with 6 sides, numbered with the numbers from 1 to 6. Player 2 uses a dice with 4 sides and the numbers from 1 to 4 on it. They stop, when player 2 does not throw a smaller number than player 1 .

```
1 p1 := 1;
2 p2 := 0;
3 while (p2 < p1) {
4     p1 := Uniform(1, 6);
5     p2 := Uniform(1, 4)
6 }
```

Listing 6.7: *Dice*

At first we take a look at the loop body:

	$\backslash\!\!\backslash 1 + \frac{1}{6} \cdot \sum_{j=1}^{6} (1 + \frac{1}{4} \cdot \sum_{i=1}^{4} \lambda(s,h)_\bullet X(s[p2\backslash i, p1\backslash j], h))$
4	$p1 := Uniform(1,6);$
	$\backslash\!\!\backslash 1 + \frac{1}{4} \cdot \sum_{i=1}^{4} \lambda(s,h)_\bullet X(s[p2\backslash i], h)$
5	$p2 := Uniform(1,4)$
	$\backslash\!\!\backslash X$

Listing 6.8: Analyzing the loop body of *Dice*

We need to find a least fixed point for the characteristic function

$$\Phi : X \mapsto 1 + [\![p2 < p1 : true]\!] \cdot (1 + \frac{1}{6} \cdot \sum_{j=1}^{6} (1 + \frac{1}{4} \cdot \sum_{i=1}^{4} \lambda(s,h)_\bullet X(s[p2\backslash i, p1\backslash j], h))).$$

As an upper ω-invariant we suggest $I_n = 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot \sum_{k=0}^{n-1} (\frac{14}{24})^k$. The probability that Player 2 throws a smaller number than Player 1 is $\frac{14}{24}$. The probability that this happens k times in a row is $(\frac{14}{24})^k$ and each iteration of the while-loop consumes 3 units of time. If $p2 \geq p1$ holds in the beginning, then the loop body will not be executed and just one unit of time is consumed for the evaluation of $p2 < p1$.

Proof.

$$\begin{aligned}\Phi(0) &= 1 + [\![p2 < p1 : true]\!] \cdot 2 \\ &\leq 1 + [\![p2 < p1 : true]\!] \cdot 3 \\ &= I_1\end{aligned}$$

$$\begin{aligned}\Phi(I_n) &= 1 + [\![p2 < p1 : true]\!] \cdot (1 + \frac{1}{6} \cdot \sum_{j=1}^{6} (1 + \frac{1}{4} \cdot \sum_{i=1}^{4} (1 + [\![i < j : true]\!] \cdot 3 \cdot \sum_{k=0}^{n-1} \Big(\frac{14}{24}\Big)^k))) \\ &= 1 + [\![p2 < p1 : true]\!] \cdot (3 + \frac{1}{6} \cdot \sum_{j=1}^{6} (\frac{1}{4} \cdot \sum_{i=1}^{4} ([\![i < j : true]\!] \cdot 3 \cdot \sum_{k=0}^{n-1} \Big(\frac{14}{24}\Big)^k))) \\ &= 1 + [\![p2 < p1 : true]\!] \cdot (3 + \frac{1}{24} \cdot 14 \cdot 3 \cdot \sum_{k=0}^{n-1} \Big(\frac{14}{24}\Big)^k) \\ &= 1 + [\![p2 < p1 : true]\!] \cdot (3 + 3 \cdot \sum_{k=1}^{n} \Big(\frac{14}{24}\Big)^k) \\ &= 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot \sum_{k=0}^{n} \Big(\frac{14}{24}\Big)^k \\ &= I_{n+1}\end{aligned}$$

As $\Phi(0) \leq I_1$ and $\Phi(I_n) = I_{n+1}$ holds, I_n is an upper ω-invariant for the while loop. □

Since I_n is an upper and lower ω-invariant, we can conclude:

$$
\begin{aligned}
&\mathrm{ert}[\textbf{while}(p2 < p1)\{p1 := Uniform(1,4); p2 := Uniform(1,3)\}](0) \\
&\leq \lim_{n\to\infty} I_n \\
&= \lim_{n\to\infty} 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot \sum_{k=0}^{n-1} \left(\frac{14}{24}\right)^k \\
&= 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot \sum_{k=0}^{\infty} \left(\frac{14}{24}\right)^k \\
&= 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot \frac{1}{1 - \frac{14}{24}} \\
&= 1 + [\![p2 < p1 : true]\!] \cdot 3 \cdot 2{,}4 \\
&= 1 + [\![p2 < p1 : true]\!] \cdot 8{,}2
\end{aligned}
$$

Now we can analyze the whole program:

```
   \\11,2
   \\3 + 8,2
   \\3 + [[0 < 1 : true]] · 8,2
1  p1 := 1;
   \\2 + [[0 < p1 : true]] · 8,2
2  p2 := 0;
   \\1 + [[p2 < p1 : true]] · 8,2
3  while (p2 < p1){
4        p1 := Uniform(1,6);
5        p2 := Uniform(1,4)
6  }
   \\0
```

Listing 6.9: Analyzing *Dice*

This means one round in our game of dice lasts on average 11,2 units of time.

Chapter 7

Operational MDPs

In the next chapter we will introduce a complete expected runtime transformer. Therefore we need an operational model for all programs $C \in \mathcal{P}^3$. Since allocation of heap cells is nondeterministic, we cannot use Markov reward models as program models anymore. We will use Markov decision processes to model programs with memory allocation and deallocation. A MDP for a program $C \in \mathcal{P}^3$ with continuation $f \in \mathbb{T}$ and initial state $\sigma_0 \in \Sigma$ is given by $\mathfrak{M}^f_{\sigma_0} = (\mathcal{K}, Act, \mathcal{P}, \kappa_0, rew)$, where $\mathcal{K} = ((\mathcal{P}^3 \cup \{\downarrow; C \mid C \in \mathcal{P}^3\} \cup \{\downarrow\}) \times \Sigma) \cup \{\langle sink \rangle\} \cup \{\langle \lightning \rangle\}$ is the set of configurations, $Act = \mathbb{N}$ the set of possible actions, $\mathcal{P} : \mathcal{K} \times Act \times \mathcal{K} \to [0,1]$ a probability function as defined in Figure 7.1, $\kappa_0 = (C, \sigma_0)$ the initial configuration and $rew : \mathcal{K} \to \mathbb{R}_{\geq 0}$ a reward function as defined in Table 7.1.

$\kappa \in \mathcal{K}$	$\mathbf{rew}(\kappa)$
$\langle \downarrow, \sigma \rangle$	$f(\sigma)$
$\langle x := e, \sigma \rangle$, $\langle x := Uniform(a,b), \sigma \rangle$, $\langle x := < e >, \sigma \rangle$, $\langle < e > := e, \sigma \rangle$, $\langle \mathbf{if}\,(b)\,\{c_1\}\,\mathbf{else}\,\{c_2\}, \sigma \rangle$, $\langle \{c_1\}\,[p]\,\{c_2\}, \sigma \rangle$, $\langle free(e), \sigma \rangle$, $\langle x := \mathbf{new}(\vec{e}), \sigma \rangle$	1
$\langle sink \rangle$, $\langle \lightning \rangle$, $\langle \downarrow; c_2, \sigma \rangle$, $\langle \mathbf{while}\,(b)\,\{c\}, \sigma \rangle$, $\langle \mathbf{empty}, \sigma \rangle$	0

Table 7.1: Definition of reward function

$$\frac{}{\langle empty, \sigma\rangle \xrightarrow{0,1} \langle \downarrow, \sigma\rangle}[\text{empty}] \qquad \frac{}{\langle \downarrow, \sigma\rangle \xrightarrow{0,1} \langle sink\rangle}[\text{term}]$$

$$\frac{}{\langle sink\rangle \xrightarrow{0,1} \langle sink\rangle}[\text{sink}] \qquad \frac{}{\langle \lightning\rangle \xrightarrow{0,1} \langle sink\rangle}[\text{fault}]$$

$$\frac{\langle c_1, \sigma\rangle \xrightarrow{0,p} \langle c_1', \sigma'\rangle,\ 0 < p \leq 1}{\langle c_1; c_2, \sigma\rangle \xrightarrow{0,p} \langle c_1'; c_2, \sigma'\rangle}[\text{seq}_1] \qquad \frac{}{\langle \downarrow; c, \sigma\rangle \xrightarrow{0,1} \langle c, \sigma\rangle}[\text{seq}_2]$$

$$\frac{}{\langle x := e, (s,h)\rangle \xrightarrow{0,1} \langle \downarrow, (s[x \backslash s(e)], h)\rangle}[\text{assgn}]$$

$$\frac{u \leq i \leq v}{\langle x := Uniform(u,v), (s,h)\rangle \xrightarrow{0,\frac{1}{v-u+1}} \langle \downarrow, (s[x \backslash i], h)\rangle}[\text{pr-assgn}]$$

$$\frac{[\![b : true]\!](\sigma) = 1}{\langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}, \sigma\rangle \xrightarrow{0,1} \langle c_1, \sigma\rangle}[\text{if-true}] \qquad \frac{[\![b : false]\!](\sigma) = 1}{\langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\}, \sigma\rangle \xrightarrow{0,1} \langle c_2, \sigma\rangle}[\text{if-false}]$$

$$\frac{}{\langle \{c_1\}\ [p]\ \{c_2\}, \sigma\rangle \xrightarrow{0,p} \langle c_1, \sigma\rangle}[\text{prob}_1] \qquad \frac{}{\langle \{c_1\}\ [p]\ \{c_2\}, \sigma\rangle \xrightarrow{0,1-p} \langle c_2, \sigma\rangle}[\text{prob}_2]$$

$$\frac{[e \mapsto _]\bigstar 1(s,h) = 1}{\langle < e > := e', (s,h)\rangle \xrightarrow{0,1} \langle \downarrow, (s, h[s(e) \backslash s(e')])\rangle}[\text{heap-manip}]$$

$$\frac{1 - [e \mapsto _]\bigstar 1(s,h) = 1}{\langle < e > := e', (s,h)\rangle \xrightarrow{0,1} \langle \lightning\rangle}[\text{heap-manip-f}]$$

$$\frac{[e \mapsto _]\bigstar 1(s,h) = 1}{\langle x := < e >, (s,h)\rangle \xrightarrow{0,1} \langle \downarrow, (s[x \backslash h(s(e))], h)\rangle}[\text{heap-lookup}]$$

$$\frac{1 - [e \mapsto _]\bigstar 1(s,h) = 1}{\langle x := < e >, (s,h)\rangle \xrightarrow{0,1} \langle \lightning\rangle}[\text{heap-lookup-f}]$$

$$\frac{}{\langle \textbf{while}\ (b)\ \{c\}, \sigma\rangle \xrightarrow{0,1} \langle \textbf{if}\ (b)\ \{c; \textbf{while}\ (b)\{c\}\}\ \textbf{else}\ \{\textbf{empty}\}, \sigma\rangle}[\text{while}]$$

$$\frac{[e \mapsto _]\bigstar 1(s,h) = 1}{\langle free(e), (s,h)\rangle \xrightarrow{0,1} \langle \downarrow, (s, h \backslash \{s(e)\})\rangle}[\text{dealloc}]$$

$$\frac{1 - [e \mapsto _]\bigstar 1(s,h) = 1}{\langle free(e), (s,h)\rangle \xrightarrow{0,1} \langle \lightning\rangle}[\text{dealloc-f}]$$

$$\frac{dim(\vec{e}) = n\ \wedge\ \{u, \ldots, u+n-1\} \cap dom(h) = \emptyset}{\langle x := new(\vec{e}), (s,h)\rangle \xrightarrow{u,1} \langle \downarrow, (s[x \backslash u], h \bigstar \{u \mapsto \vec{e}\})\rangle}[\text{alloc}]$$

Figure 7.1: Rules for transition probability of operational MDPs

Chapter 8

Expected Runtime Calculus with (De)Allocation

8.1 Definition of ert with (de)allocation

Now we are going to expand the ert calculus, so that we can analyze allocation and deallocation of heap cells. This expanded version of our expected runtime transformer is defined for all possible programs $C \in \mathcal{P}^3$. It is complete, because we define it inductively over the structure of our programming language $\mathcal{P}$.

$\mathbf{C} \in \mathcal{P}^{\mathbf{3}}$	$\mathbf{ert}[\mathbf{C}](f)$ for $f \in \mathbb{T}$
empty	f
$x := e$	$1 + \lambda\,(s,h)_{\bullet}\, f\,(s\,[x \backslash s(e)]\,,\, h)$
$x := Uniform(u,v)$	$1 + \frac{1}{v-u+1} \cdot \sum_{i=u}^{v} \lambda(s,h)_{\bullet} f(s[x\backslash i], h)$
$c_1 \,;\, c_2$	$ert\,[c_1]\,(ert\,[c_2]\,(f))$
if $(b)\,\{c_1\}$ **else** $\{c_2\}$	$1 + [\![b : \text{true}]\!] \cdot \text{ert}[c_1](f) + [\![b : \text{false}]\!] \cdot \text{ert}[c_2](f)$
while $(b)\,\{c\}$	$\text{lfp X}_{\bullet}\, 1 + [\![b : \text{false}]\!] \cdot f + [\![b : \text{true}]\!] \cdot \text{ert}[c](X)$
$\{c_1\}\,[p]\,\{c_2\}$	$1 + p \cdot \text{ert}[c_1](f) + (1-p) \cdot \text{ert}[c_2](f)$
$< e > := e'$	$1 + [e \mapsto _] \bigstar ([e \mapsto e'] \text{-}\!\!\bigstar\, f)$
$x := < e >$	$1 + \lambda(s,h)_{\bullet} \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \text{-}\!\!\bigstar\, f(s[x\backslash v], h))$
$free(e)$	$1 + [e \mapsto _] \bigstar f$
$x := new(\vec{e})$	$1 + \lambda(s,h)_{\bullet} \sup_{u \in \mathbb{N}_{>0}} [u \mapsto \vec{e}] \text{-}\!\!\bigstar\, f(s[x\backslash u], h)$

Deallocation: Deallocation of a memory cell consumes one unit of time. Since a cell is removed from the heap, the continuation f is evaluated on the reduced heap. If the given address does not exists in the heap, the continuation will not be evaluated, because the program stops due to a memory fault. This is realized by the allocated pointer predicate

and the runtime separation conjunction:

$$\begin{aligned}
&[e \mapsto _] \bigstar f(s,h) \\
&= \sup_{h_1,h_2} \{[e \mapsto _](s,h_1) \cdot f(s,h_2) \mid h_1 \bigstar h_2 = h\} \\
&= \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ \sup\limits_{h_2}\{1 \cdot f(s,h_2) \mid h_1 = \{e \mapsto h(e)\} \wedge h_1 \bigstar h_2 = h\} & \text{, if } e \in dom(h) \end{cases} \\
&= \begin{cases} 0 & \text{, if } s(e) \notin dom(h) \\ f(s, h \backslash \{e \mapsto h(e)\}) & \text{, if } e \in dom(h) \end{cases}
\end{aligned}$$

Allocation: Allocation of memory cells consumes one unit of time, too. The addresses of the new cells are chosen nondeterministically and the continuation is evaluated on the extended heap and the updated stack. We assume the chosen addresses maximize the continuation, because we are more interested in the worst than in the best possible expected runtime. Those new addresses do not overlap with the old heap. The runtime separation implication takes care of this automatically by its definition. We assume that our memory resources are unlimited and the allocation of memory thus never fails.

8.2 Soundness with (de)allocation

To prove the soundness of our ert calculus we have to show, that

$$\text{ExpRew}^{\mathfrak{M}^f_\sigma[\![C]\!]}(\langle sink \rangle) = \text{ert}[C](f)(\sigma)$$

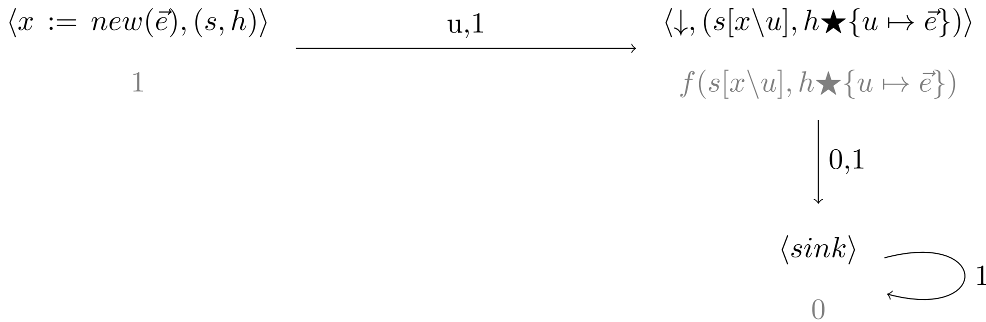
holds for all $C \in \mathcal{P}^3$, $\sigma \in \Sigma$ and $f \in \mathbb{T}$. We will prove this by induction over the structure of $C \in \mathcal{P}^3$. If we take a look at all the rules in Figure 7.1, we can see that apart from allocation no nondeterministic choice is possible, because there is just one possible action: 0.

<u>**Base Case:**</u>

For the empty program, the assignment and the probabilistic assignment $|\mathcal{E}(\kappa)| = |\{0\}| = 1$ for all configurations $\kappa \in \mathcal{K}$ that appear in their MDPs. This means the operational MDPs are Markov chains in those cases and the expected reward is computed exactly like in the proof in Section 6.2. Therefore we do not have to prove these cases again.

Allocation: $C := x := \textbf{new}(\vec{e})$

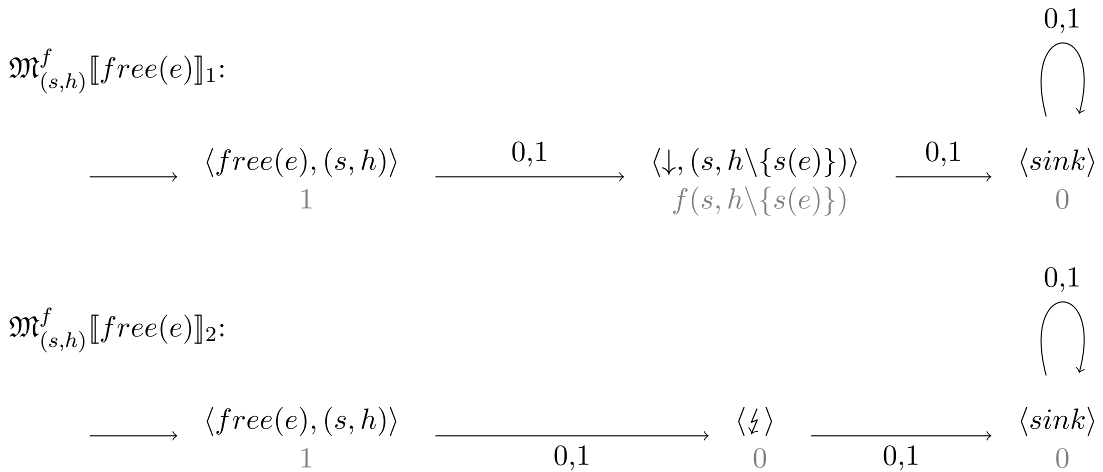
Let $dim(\vec{e})$ be n. The MDP $\mathfrak{M}^f_{(s,h)}[\![x := \textbf{new}(\vec{e})]\!]$ has infinitely many states following directly after the initial state which then lead immediately to the sink state. All those states are reachable with different actions $u \in Act$. In the following graph we just drew one of those states as an example.



Furthermore,
$\Pi(\langle x := \mathbf{new}(\vec{e}), (s,h)\rangle, \langle sink\rangle) = \{\langle x := \mathbf{new}(\vec{e}), \sigma\rangle\langle\downarrow, (s[x\backslash u], h\bigstar\{u \mapsto \vec{e}\})\rangle\langle sink\rangle, \mid$
$\forall u \in \mathbb{N}_{>0}$ with $\{u, \ldots, u+n-1\} \cap dom(h) = \emptyset\}$. Therefore we have

$$
\begin{aligned}
&\text{ert}[x := new(\vec{e})](f)(s,h)\\
&= 1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto \vec{e}] \text{---}\!\!\bigstar\ f(s[x\backslash u], h)\\
&= 1 + \sup_{u \in \mathbb{N}_{>0}} \sup_{h'} \{f(s[x\backslash u], h\bigstar h') \mid h' \perp h \wedge [u \mapsto \vec{e}](s,h') = 1\}\\
&= 1 + \sup_{u \in \mathbb{N}_{>0}} \sup_{h'} \{f(s[x\backslash u], h\bigstar h') \mid h' \perp h \wedge h' = \{u \mapsto \vec{e}\}\}\\
&= 1 + \sup_{u \in \mathbb{N}_{>0}} \sup_{h'} \{f(s[x\backslash u], h\bigstar h') \mid \{u, \ldots, u+n-1\} \cap dom(h) = \emptyset \wedge h' = \{u \mapsto \vec{e}\}\}\\
&= 1 + \sup_{u \in \mathbb{N}_{>0}} \{f(s[x\backslash u], h\bigstar\{u \mapsto \vec{e}\}) \mid \{u, \ldots, u+n-1\} \cap dom(h) = \emptyset\}\\
&= 1 + \sup_{(u,\ldots,u+n-1) \in (\mathbb{N}_{>0} \backslash dom(h))^n} f(s[x\backslash u], h\bigstar\{u \mapsto \vec{e}\})\\
&= \sup_{(u,\ldots,u+n-1) \in (\mathbb{N}_{>0} \backslash dom(h))^n} 1 \cdot (1 + f(s[x\backslash u], h\bigstar\{u \mapsto \vec{e}\}))\\
&= \sup_{\mathfrak{G}} \sum_{\pi \in \Pi(\langle\langle x := \mathbf{new}(\vec{e}), (s,h)\rangle, \langle sink\rangle))} Pr^{\mathfrak{M}^f_{(s,h)}\mathfrak{G}}(\pi) \cdot rew(\pi)\\
&= \text{ExpRew}^{\mathfrak{M}^f_{(s,h)}[\![x := \mathbf{new}(\vec{e})]\!]}(f).
\end{aligned}
$$

Deallocation: $C := free(e)$
For heap manipulation we need to distinguish between two cases: Either a memory fault occurs or not. Therefore we need two Markov reward models $\mathfrak{M}^f_{(s,h)}[\![free(e)]\!]_1$ and $\mathfrak{M}^f_{(s,h)}[\![free(e)]\!]_2$ which both contain one infinite path:



Furthermore, $\Pi_1(\langle free(e), (s,h)\rangle, \langle sink\rangle) = \{\langle free(e), (s,h)\rangle\langle\downarrow, (s, h\backslash\{s(e)\})\rangle\langle sink\rangle\}$ and $\Pi_2(\langle free(e), (s,h)\rangle, \langle sink\rangle) = \{\langle free(e), (s,h)\rangle\langle \lightning \rangle\langle sink\rangle\}$. Therefore we have

$$
\begin{aligned}
&\mathrm{ert}[\mathit{free}(e)](f)(s,h)\\
&= 1 + ([e \mapsto _] \bigstar f)(s,h)\\
&= 1 + \max_{h_1,h_2}\{\ \underbrace{[e \mapsto _](s,h_1)}_{=0,\text{ if } dom(h_1)\neq\{s(e)\}}\ \cdot f(s,h_2) \mid h_1 \bigstar h_2 = h\}\\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h)\\ [e \mapsto _](s,\{s(e) \mapsto h(s(e))\}) \cdot f(s, h\backslash\{s(e)\}) & \text{, if } s(e) \in dom(h)\end{cases}\\
&= 1 + \begin{cases} 0 & \text{, if } s(e) \notin dom(h)\\ 1 \cdot f(s, h\backslash\{s(e)\}) & \text{, if } s(e) \in dom(h)\end{cases}\\
&= \begin{cases} 1 & \text{, if } s(e) \notin dom(h)\\ 1 + f(s, h\backslash\{s(e)\}) & \text{, if } s(e) \in dom(h)\end{cases}\\
&= \begin{cases} 1 \cdot 1 & \text{, if } s(e) \notin dom(h)\\ 1 \cdot (1 + f(s, h\backslash\{s(e)\})) & \text{, if } s(e) \in dom(h)\end{cases}\\
&= \begin{cases} \sup\limits_{\mathfrak{G}} \sum\limits_{\pi \in \Pi_2(\langle \mathit{free}(e),(s,h)\rangle,\langle \mathit{sink}\rangle)} Pr^{\mathfrak{M}^f_{(s,h)}\mathfrak{G}}[\![\mathit{free}(e)]\!]_2\{\pi\} \cdot rew_{\mathfrak{G}}(\pi) & \text{, if } s(e) \notin dom(h)\\ \sup\limits_{\mathfrak{G}} \sum\limits_{\pi \in \Pi_1(\langle \mathit{free}(e),(s,h)\rangle,\langle \mathit{sink}\rangle)} Pr^{\mathfrak{M}^f_{(s,h)}\mathfrak{G}}[\![\mathit{free}(e)]\!]_1\{\pi\} \cdot rew_{\mathfrak{G}}(\pi) & \text{, if } s(e) \in dom(h)\end{cases}\\
&= \begin{cases} \mathrm{ExpRew}^{\mathfrak{M}^f_{(s,h)}[\![\mathit{free}(e)]\!]_2}(\langle \mathit{sink}\rangle) & \text{, if } s(e) \notin dom(h)\\ \mathrm{ExpRew}^{\mathfrak{M}^f_{(s,h)}[\![\mathit{free}(e)]\!]_1}(\langle \mathit{sink}\rangle) & \text{, if } s(e) \in dom(h)\end{cases}
\end{aligned}
$$

Induction Hypothesis:
All subprograms $C' \in \mathcal{P}^3$ of C fulfill

$$\mathrm{ExpRew}^{\mathfrak{M}^f_{\sigma_0}[\![C']\!]}(\langle \mathit{sink}\rangle) = \mathrm{ert}[C'](f)(\sigma_0)$$

for all $\sigma \in \Sigma$ and all $f \in \mathbb{T}$.

Inductive Step:
For the inductive step we need to examine sequential composition, conditional choice, probabilistic choice and loops. The proofs of all of those cases are analogous to the ones in chapter 6.2. You just need to exchange $\widetilde{\mathcal{M}}$ by $\mathfrak{M}$ and rew by $rew_{\mathfrak{G}}$. All the statements we made in Chapter 6.2 also hold for MDPs.

8.3 More Examples

Now that we have proven the soundness of our whole calculus we are going to analyze more examples. The first three examples will consider a single linked list. The first program *List Generation* flips a coin and either adds an element to a list or stops. The second program *List Deletion* will delete a list element by element. As a third example we will take a look at the runtime of *List Generation*; *List Deletion*, which is a concatenation of those two programs.

8.3.1 List Generation

```
1  i := 1;
2  x := 0;
3  len := 0;
4  while(i := 1){
5      x := new(x);
6      i := Uniform(0,1);
7      len := len + 1
8  }
```

Listing 8.1: *List Generation*

At first, we will search an upper ω-invariant for the while loop.
For this we need to compute $\text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len + 1](X)$ first.

$\Lsh 1 + \sup_{a \in \mathbb{N}_{>0}} [a \mapsto x] \mathrel{-\!\!\bigstar} (2 + \frac{1}{2} \cdot \sum_{j=0}^{1} \lambda(s,h)_{\bullet} X(s[len \backslash len+1, i \backslash j, x \backslash a], h))$

5 $x := \textbf{new}(x);$

$\Lsh 2 + \frac{1}{2} \cdot \sum_{j=0}^{1} \lambda(s,h)_{\bullet} X(s[len \backslash len+1, i \backslash j], h)$

$\Lsh 1 + \frac{1}{2} \cdot \sum_{j=0}^{1} (1 + \lambda(s,h)_{\bullet} X(s[len \backslash len+1, i \backslash j], h))$

6 $i := Uniform(0,1)$

$\Lsh 1 + \lambda(s,h)_{\bullet} X(s[len \backslash len+1], h)$

7 $len := len + 1$

$\Lsh X$

Listing 8.2: Analyzing loop body of *List Generation*

The while loop determines the characteristic function

$$\Phi : X \mapsto 1 + [\![i = 1 : true]\!] \cdot \text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len + 1](X)$$

As an upper ω-invariant we suggest $I_n = 1 + [\![i = 1 : true]\!] \cdot \sum_{k=0}^{n-1} 4 \cdot \left(\frac{1}{2}\right)^k$.

Proof.

$$\begin{aligned}\Phi(0) &= 1 + [\![i = 1 : true]\!] \cdot \text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len + 1](0) \\ &= 1 + [\![i = 1 : true]\!] \cdot 3 \\ &\leq 1 + [\![i = 1 : true]\!] \cdot 4 \\ &= I_1\end{aligned}$$

$$\begin{aligned}
\Phi(I_n) &= 1 + [\![i = 1 : true]\!] \cdot \mathrm{ert}[x := \mathbf{new}(x); i := Uniform(0,1); len := len + 1](I_n) \\
&= 1 + [\![i = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot (1 + \sum_{k=0}^{n-1} 4 \cdot \left(\frac{1}{2}\right)^k) + \frac{1}{2} \cdot 1) \\
&= \begin{cases} 1 & , \text{if } i \neq 1 \\ 5 + \frac{1}{2} \sum_{k=0}^{n-1} 4 \cdot \left(\frac{1}{2}\right)^k & , \text{if } i = 1 \end{cases} \\
&= \begin{cases} 1 & , \text{if } i \neq 1 \\ 5 + \sum_{k=1}^{n} 4 \cdot \left(\frac{1}{2}\right)^k & , \text{if } i = 1 \end{cases} \\
&= \begin{cases} 1 & , \text{if } i \neq 1 \\ 1 + \sum_{k=0}^{n} 4 \cdot \left(\frac{1}{2}\right)^k & , \text{if } i = 1 \end{cases} \\
&= 1 + [\![i = 1 : true]\!] \cdot \sum_{k=0}^{n} 4 \cdot \left(\frac{1}{2}\right)^k \\
&= I_{n+1}
\end{aligned}$$

□

Since we have proven that $I_n = 1 + [\![i = 1 : true]\!] \cdot \sum_{k=0}^{n-1} 4 \cdot \left(\frac{1}{2}\right)^k$ is an upper ω-invariant for the while loop, we can conclude the following:

$$\begin{aligned}
&\mathrm{ert}[\mathbf{while}(i = 1)\{x := \mathbf{new}(x); i := Uniform(0,1); len := len + 1\}](0) \\
&\leq \lim_{n \to \infty} I_n \\
&= \lim_{n \to \infty} 1 + [\![i = 1 : true]\!] \cdot \sum_{k=0}^{n-1} 4 \cdot \left(\frac{1}{2}\right)^k \\
&= 1 + [\![i = 1 : true]\!] \cdot \sum_{k=0}^{\infty} 4 \cdot \left(\frac{1}{2}\right)^k \\
&= 1 + [\![i = 1 : true]\!] \cdot \frac{4}{1 - \frac{1}{2}} \\
&= 1 + [\![i = 1 : true]\!] \cdot 8
\end{aligned}$$

Now we can analyze the whole program *List Generation*:

$\Large\setminus\!\!\setminus 12$
$\Large\setminus\!\!\setminus 4 + 8$
$\Large\setminus\!\!\setminus 4 + [\![1 = 1 : true]\!] \cdot 8$

1 $i := 1;$

$\Large\setminus\!\!\setminus 3 + [\![i = 1 : true]\!] \cdot 8$

2 $x := 0;$

$\Large\setminus\!\!\setminus 2 + [\![i = 1 : true]\!] \cdot 8$

$\backslash\!\!\backslash\, 2 + [\![i = 1 : true]\!] \cdot 8$

3 $len := 0;$

$\backslash\!\!\backslash\, 1 + [\![i = 1 : true]\!] \cdot 8$

4 $\mathbf{while}(i = 1)\{$
5 $\quad x := \mathbf{new}(x);$
6 $\quad i := Uniform(0,1);$
7 $\quad len := len + 1$
8 $\}$

$\backslash\!\!\backslash\, 0$

Listing 8.3: Analysis of *List Generation*

This means, the program *List Generation* has an expected runtime of 12 units of time or less.

8.3.2 List Deletion

This program deletes a single linked list, if x is the address of the last element and len is the length of the list.

1 $\mathbf{while}(len > 0)\{$
2 $\quad y :=< x >;$
3 $\quad free(x);$
4 $\quad x := y;$
5 $\quad len := len - 1$
6 $\}$

Listing 8.4: *List Deletion*

At first we will take a look at the loop body.

$\backslash\!\!\backslash\, 1 + \sup_{v \in \mathbb{Z}} [x \mapsto v] \bigstar ([x \mapsto v] \mathrel{-\!\!\bigstar} (1 + [x \mapsto _] \bigstar$
$\quad (2 + \lambda(s,h)_{\bullet} X(s[len \backslash len - 1, x \backslash v, y \backslash v], h))))$

2 $y :=< x >;$

$\backslash\!\!\backslash\, 1 + [x \mapsto _] \bigstar (2 + \lambda(s,h)_{\bullet} X(s[len \backslash len - 1, x \backslash y], h))$

3 $free(x);$

$\backslash\!\!\backslash\, 2 + \lambda(s,h)_{\bullet} X(s[len \backslash len - 1, x \backslash y], h)$

4 $x := y;$

$\backslash\!\!\backslash\, 1 + \lambda(s,h)_{\bullet} X(s[len \backslash len - 1], h)$

5 $len := len - 1$

$\backslash\!\!\backslash\, X$

Listing 8.5: Analyzing the loop body of *List Deletion*

The while loop defines the characteristic function

$$\Phi : X \mapsto 1 + [\![len > 0 : true]\!] \cdot \mathrm{ert}[y :=< x >; free(x); x := y; len := len - 1](X).$$

We suggest $I_n = 1 + [\![len - (n-1) > 0 : true]\!] \cdot 5 \cdot n + [\![len - (n-1) > 0 : false]\!] \cdot 5 \cdot len$ as an upper ω-invariant of the while loop. We assume that the variable len is not negative.

Proof.

$$\begin{aligned}
\Phi(0) &= 1 + [\![len > 0 : true]\!] \cdot \mathrm{ert}[y :=< x >; free(x); x := y; len := len - 1](0) \\
&= 1 + [\![len > 0 : true]\!] \cdot (1 + \sup_{v \in \mathbb{Z}} [x \mapsto v] \bigstar ([x \mapsto v] \mathrel{-\!\!\bigstar} (1 + [x \mapsto _] \bigstar (2 + 0)))) \\
&\leq 1 + [\![len > 0 : true]\!] \cdot 4 \\
&= 1 + [\![len > 0 : true]\!] \cdot 4 + [\![len > 0 : false]\!] \cdot 0 \\
&\leq 1 + [\![len > 0 : true]\!] \cdot 5 \cdot 1 + [\![len > 0 : false]\!] \cdot 0 \\
&= 1 + [\![len > 0 : true]\!] \cdot 5 \cdot 1 + [\![len > 0 : false]\!] \cdot 5 \cdot len \\
&= I_1
\end{aligned}$$

$$\begin{aligned}
\Phi(I_n) &= 1 + [\![len > 0 : true]\!] \cdot \mathrm{ert}[y :=< x >; free(x); x := y; len := len - 1](I_n) \\
&= 1 + [\![len > 0 : true]\!] \cdot (1 + \sup_{v \in \mathbb{Z}} [x \mapsto v] \bigstar ([x \mapsto v] \mathrel{-\!\!\bigstar} (1 + [x \mapsto _] \bigstar \\
&\quad (2 + 1 + [\![len - n > 0 : true]\!] \cdot 5 \cdot n + [\![len - n > 0 : false]\!] \cdot 5 \cdot (len - 1))))) \\
&\leq 1 + [\![len > 0 : true]\!] \cdot (5 + [\![len - n > 0 : true]\!] \cdot 5 \cdot n + [\![len - n > 0 : false]\!] \cdot \\
&\quad 5 \cdot (len - 1)) \\
&= \begin{cases} 1 & \text{, if } len \not> 0 \\ 1 + 5 + 5 \cdot n & \text{, if } len > 0 \text{ and } len - n > 0 \\ 1 + 5 + 5 \cdot (len - 1) & \text{, if } len > 0 \text{ and } len - n \not> 0 \end{cases} \\
&= \begin{cases} 1 & \text{, if } len \not> 0 \\ 1 + 5 \cdot (n + 1) & \text{, if } len > 0 \text{ and } len - n > 0 \\ 1 + 5 \cdot len & \text{, if } len > 0 \text{ and } len - n \not> 0 \end{cases} \\
&= 1 + [\![len - n > 0 : true]\!] \cdot [\![len > 0 : true]\!] \cdot 5 \cdot (n + 1) \\
&\quad + [\![len - n > 0 : false]\!] \cdot [\![len > 0 : true]\!] \cdot 5 \cdot len
\end{aligned}$$

Since we have assumed, that len is not negative:

$$\begin{aligned}
&= 1 + [\![len - n > 0 : true]\!] \cdot 5 \cdot (n + 1) \\
&\quad + [\![len - n > 0 : false]\!] \cdot [\![len \neq 0 : true]\!] \cdot 5 \cdot len \\
&= 1 + [\![len - n > 0 : true]\!] \cdot 5 \cdot (n + 1) + [\![len - n > 0 : false]\!] \cdot 5 \cdot len \\
&= I_{n+1}
\end{aligned}$$

□

With $I_n = 1 + [\![len - (n-1) > 0 : true]\!] \cdot 5 \cdot n + [\![len - (n-1) > 0 : false]\!] \cdot 5 \cdot len$ as an upper ω-invariant for the while loop we can conclude that the expected runtime depends

linearly on the length len of the list:

$$\begin{aligned}
&\mathrm{ert}[\mathit{List\ Deletion}](0)\\
&= \mathrm{ert}[\mathbf{while}(len > 0)\{y :=< x >; free(x); x := y; len := len - 1\}](0)\\
&\leq \lim_{n\to\infty} I_n\\
&= \lim_{n\to\infty} 1 + [\![len - (n-1) > 0 : true]\!] \cdot 5 \cdot n + [\![len - (n-1) > 0 : false]\!] \cdot 5 \cdot len\\
&= 1 + 5 \cdot len
\end{aligned}$$

```
   \\1 + 5 · len
1  while(len > 0){
2        y :=< x >;
3        free(x);
4        x := y;
5        len := len − 1
6  }
   \\0
```

Listing 8.6: Analysis of *List Deletion*

8.3.3 List Generation;List Deletion

As a third program we will examine how the expected runtime behaves if we execute the programs *List Generation* and *List Deletion* in sequence. The lines 9 to 14 have already been analyzed in the analysis of *List Deletion*.

```
1  i := 1;
2  x := 0;
3  len := 0;
4  while(i = 1){
5        x :=new(x) ;
6        i := Uniform(0, 1) ;
7        len := len + 1
8  }
   \\1 + 5 · len
9  while(len > 0){
10       y :=< x > ;
11       free(x) ;
12       x := y;
13       len := len − 1
14 }
   \\0
```

Listing 8.7: *List Generation*; *List Deletion*

Now we take a look at the characteristic function of the first while loop in List;List Deletion:

$$\begin{aligned}
\Phi : X \mapsto\ & 1 + [\![i = 1 : true]\!] \cdot \mathrm{ert}[x := \mathbf{new}(x); i := Uniform(0,1); len := len + 1](X)\\
& + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len)
\end{aligned}$$

We have already computed $\text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len+1](X)$ in section 8.3.1. As an upper ω-invariant we suggest

$$I_n = 1 + [\![i = 1 : true]\!] \cdot (4 + \sum_{k=1}^{n-1} \left(\frac{1}{2}\right)^k \cdot (5 + 5(len + k)))) + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len).$$

Proof.

$$\begin{aligned}
\Phi(0) &= 1 + [\![i = 1 : true]\!] \cdot \text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len+1](0) \\
&\quad + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len) \\
&\le 1 + [\![i = 1 : true]\!] \cdot 3 + [\![i = 1 : false]\!](1 + 5 \cdot len) \\
&\le 1 + [\![i = 1 : true]\!] \cdot 4 + [\![i = 1 : false]\!](1 + 5 \cdot len) \\
&= I_1
\end{aligned}$$

$$\begin{aligned}
\Phi(I_n) &= 1 + [\![i = 1 : true]\!] \cdot \text{ert}[x := \textbf{new}(x); i := Uniform(0,1); len := len+1](I_n) \\
&\quad + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len) \\
&\le 1 + [\![i = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot (2 + 5 \cdot (len + 1) \\
&\qquad + 1 + 4 + \sum_{k=1}^{n-1} (\left(\frac{1}{2}\right)^k \cdot (5 + 5(len + k + 1))))) + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len) \\
&= \begin{cases} 1 + 3 + \frac{1}{2} \cdot (2 + 5 \cdot (len+1) + 1 + 4 \\ \qquad\qquad + \sum_{k=1}^{n-1} (\frac{1}{2})^k \cdot (5 + 5(len + k + 1))) & \text{, if } i = 1 \\ 2 + 5 \cdot len & \text{, if } i \neq 1 \end{cases} \\
&= \begin{cases} 4 + 1 + 2,5 \cdot (len+1) + 0,5 + 2 + \sum_{k=2}^{n} (\frac{1}{2})^k \cdot (5 + 5(len + k)) & \text{, if } i = 1 \\ 2 + 5 \cdot len & \text{, if } i \neq 1 \end{cases} \\
&= \begin{cases} 10 + 2,5 \cdot len + \sum_{k=2}^{n} (\frac{1}{2})^k \cdot (5 + 5(len + k)) & \text{, if } i = 1 \\ 2 + 5 \cdot len & \text{, if } i \neq 1 \end{cases} \\
&= \begin{cases} 5 + \sum_{k=1}^{n} (\frac{1}{2})^k \cdot (5 + 5(len + k)) & \text{, if } i = 1 \\ 2 + 5 \cdot len & \text{, if } i \neq 1 \end{cases} \\
&= 1 + [\![i = 1 : true]\!] \cdot (4 + \sum_{k=1}^{n} \left(\frac{1}{2}\right)^k \cdot (5 + 5(len + k)))) + [\![i = 1 : false]\!] \cdot (1 + 5 \cdot len) \\
&= I_{n+1}
\end{aligned}$$

□

We use the geometric series to calculate the limit of I_n:

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$$

Then we derive...

$$\frac{1}{(1-x)^2} = \sum_{k=1}^{\infty} k \cdot x^{k-1}$$

and multiply by x :

$$\frac{x}{(1-x)^2} = \sum_{k=1}^{\infty} k \cdot x^{k}$$

As we started with the geometric series, this holds for an arbitrary $0 < x < 1$. We set $x = \frac{1}{2}$.

$$2 = \sum_{k=1}^{\infty} \frac{k}{2^k}$$

Then we can conclude for the runtime of the while loop, that

$$\begin{aligned}
&\text{ert}[\mathbf{while}(i=1)\{x := \mathbf{new}(x); i := Uniform(0,1); len := len+1\}](1+5\cdot len)\\
&\leq \lim_{n\to\infty} I_n\\
&= 1 + [\![i=1:true]\!]\cdot(4+\sum_{k=1}^{\infty}(\left(\frac{1}{2}\right)^k\cdot(5+5(len+k)))) + [\![i=1:false]\!]\cdot(1+5\cdot len)\\
&= \begin{cases} 1+4+\sum\limits_{k=1}^{\infty}((\frac{1}{2})^k\cdot(5+5(len+k))) & \text{, if } i=1\\ 2+5\cdot len & \text{, if } i\neq 1\end{cases}\\
&= \begin{cases} 5+\sum\limits_{k=1}^{\infty}((\frac{1}{2})^k\cdot(5+5\cdot len)) + 5\cdot\sum\limits_{k=1}^{\infty}\frac{k}{2^k} & \text{, if } i=1\\ 2+5\cdot len & \text{, if } i\neq 1\end{cases}\\
&= \begin{cases} 5+\frac{(5+5\cdot len)}{1-\frac{1}{2}}+5\cdot 2 & \text{, if } i=1\\ 2+5\cdot len & \text{, if } i\neq 1\end{cases}\\
&= \begin{cases} 5+10+10\cdot len+10 & \text{, if } i=1\\ 2+5\cdot len & \text{, if } i\neq 1\end{cases}\\
&= \begin{cases} 25+10\cdot len & \text{, if } i=1\\ 2+5\cdot len & \text{, if } i\neq 1\end{cases}\\
&= [\![i=1:true]\!]\cdot(25+10\cdot len) + [\![i=1:false]\!]\cdot(2+5\cdot len).
\end{aligned}$$

Now we can analyze the whole program:

```
    \\28
    \\3 + 1 · 25 + 0 · 2
    \\3 + [[1 = 1 : true]] · 25 + [[1 = 1 : false]] · 2
1   i := 1;
    \\2 + [[i = 1 : true]] · 25 + [[i = 1 : false]] · 2
2   x := 0;
    \\1 + [[i = 1 : true]] · 25 + [[i = 1 : false]] · 2
3   len := 0;
    \\[[i = 1 : true]] · (25 + 10 · len) + [[i = 1 : false]] · (2 + 5 · len)
4   while(i = 1){
5       x := new(x);
6       i := Uniform(0, 1);
7       len := len + 1
8   }
    \\1 + 5 · len
9   while(len > 0){
10      y :=< x >;
11      free(x);
12      x := y;
13      len := len − 1
14  }
    \\0
```

Listing 8.8: Analysis of *List Generation*; *List Deletion*

28 units of time is an upper bound for the expected runtime of *List Generation*; *List Deletion*.

8.3.4 Comp

As we can see in the following example *Comp*, our calculus is not compositional.
This means, that two programs with finite expected runtime do not necessarily have a finite expected runtime when executed in sequence. For showing this fact we choose two programs $Comp_1$ and $Comp_2$, that have a finite expected runtime when executed alone. But when they are executed successively their expected runtime is infinite. For proving an infinite expected runtime we show that the minimal expected runtime is infinte by using lower ω-invariants instead of upper ω-invariants. This example is inspired by an example in [10].

```
1   b := 1;
2   x := 1;
3   while(b = 1){
4       x := x ∗ x ∗ 2;
5       {b := 0} [1/2] {b := 1}
6   }
```

Listing 8.9: $Comp_1$

$Comp_1$ flips a coin in each iteration of it's while loop and either stops or continues to compute $x := 2 \cdot x^2$.

```
1 while (x > 0) {
2     x := x − 1
3 }
```

Listing 8.10: $Comp_2$

$Comp_2$ decrements x until it is not positive anymore.
At first we want to prove that $Comp_1$ and $Comp_2$ indeed have finite expected runtimes.

Comp$_1$

To compute ert[$Comp_1$](0) we have to analyze the loop body at first:

$\sslash 3 + \frac{1}{2} \cdot \sum_{j=0}^{1} (\lambda(s,h)_\bullet X(s[b\backslash j, x\backslash 2 \cdot x^2], h))$

$\sslash 2 + \frac{1}{2} \cdot \sum_{j=0}^{1} (1 + \lambda(s,h)_\bullet X(s[b\backslash j, x\backslash 2 \cdot x^2], h))$

4 $x := x * x * 2;$

$\sslash 1 + \frac{1}{2} \cdot \sum_{j=0}^{1} (1 + \lambda(s,h)_\bullet X(s[b\backslash j], h))$

5 $\{b := 0\}\,[\frac{1}{2}]\,\{b := 1\}$

$\sslash X$

Listing 8.11: Analyzing loop body of $Comp_1$

$\Phi_1 : X \mapsto 1 + [\![b = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot \sum_{j=0}^{1} \lambda(s,h)_\bullet X(s[b\backslash j, x\backslash 2 \cdot x^2], h))$ is the characteristic function of the loop in $Comp_1$.
As an upper ω-invariant for the while loop we suggest: $I_n = 1 + [\![b = 1 : true]\!] \cdot \sum_{k=0}^{n} (\frac{1}{2})^k \cdot 4$.

Proof.

$$\begin{aligned}
\Phi_1(0) &= 1 + [\![b = 1 : true]\!] \cdot (3 + 0) \\
&= 1 + [\![b = 1 : true]\!] \cdot (2 + 1) \\
&= 1 + [\![b = 1 : true]\!] \cdot \sum_{k=0}^{1} \left(\frac{1}{2}\right)^k \cdot 2 \\
&\leq 1 + [\![b = 1 : true]\!] \cdot \sum_{k=0}^{1} \left(\frac{1}{2}\right)^k \cdot 4 \\
&= I_1
\end{aligned}$$

$$
\begin{aligned}
\Phi_1(I_n) &= 1 + [\![b = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot \sum_{j=0}^{1} \lambda(s,h)_{\bullet} I_n(s[b \backslash j], h)) \\
&= 1 + [\![b = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot (1 + 1 + \sum_{k=0}^{n} \left(\frac{1}{2}\right)^k \cdot 4)) \\
&= 1 + [\![b = 1 : true]\!] \cdot (4 + \frac{1}{2} \cdot (\sum_{k=0}^{n} \left(\frac{1}{2}\right)^k \cdot 4)) \\
&= 1 + [\![b = 1 : true]\!] \cdot (4 + \sum_{k=1}^{n+1} \left(\frac{1}{2}\right)^k \cdot 4) \\
&= 1 + [\![b = 1 : true]\!] \cdot \sum_{k=0}^{n+1} \left(\frac{1}{2}\right)^k \cdot 4 \\
&= I_{n+1}
\end{aligned}
$$

□

Since I_n is an upper ω-invariant, we can conclude

$$
\begin{aligned}
&\Rightarrow \mathrm{ert}[\mathbf{while}(b = 1)\{x := x * x * 2; \{b := 0\}[\frac{1}{2}]\{b := 1\}\}](0) \\
&\leq \lim_{n \to \infty} I_n \\
&= 1 + [\![b = 1 : true]\!] \cdot 4 \cdot \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k \\
&= 1 + [\![b = 1 : true]\!] \cdot 4 \cdot \frac{1}{1 - \frac{1}{2}} \\
&= 1 + [\![b = 1 : true]\!] \cdot 8.
\end{aligned}
$$

Now we can analyze $Comp_1$ completely:

```
  \\ 11
  \\ 3 + 1 · 8
  \\ 3 + ⟦1 = 1 : true⟧ · 8
1 b := 1;
  \\ 2 + ⟦b = 1 : true⟧ · 8
2 x := 1;
  \\ 1 + ⟦b = 1 : true⟧ · 8
3 while(b = 1){
4     x := x * x * 2;
5     {b := 0} [1/2] {b := 1}
6 }
  \\ 0
```

Listing 8.12: Analysis of $Comp_1$

As we can see, $Comp_1$ has a finite expected runtime of not more than 11 units of time.

Comp$_2$

It is easy to see, that $\Phi_2 = 1 + [\![x > 0 : true]\!] \cdot (1 + \lambda(s,h)_{\bullet}X(s[x \backslash x-1], h))$ is the characteristic function of the while loop in $Comp_2$. As an upper and lower ω-invariant for $Comp_2$ we suggest

$$I_n = 1 + [\![x > 0 : true]\!] \cdot ([\![x - (n-1) > 0 : true]\!] \cdot 1 + \sum_{i=0}^{n-2} 2 \cdot [\![x - i > 0 : true]\!]).$$

Proof.

$$\begin{aligned} \Phi_2(0) &= 1 + [\![x > 0 : true]\!] \cdot 1 \\ &= I_1 \end{aligned}$$

$$\begin{aligned} \Phi_2(I_n) = 1 + [\![x > 0 : true]\!] \cdot (1 + 1 + [\![x - 1 > 0 : true]\!] \cdot ([\![x - (n-1) - 1 > 0 : true]\!] \cdot 1 \\ + \sum_{i=0}^{n-2} 2 \cdot [\![x - i - 1 > 0 : true]\!])) \end{aligned}$$

$$\begin{aligned} &= 1 + [\![x > 0 : true]\!] \cdot \\ &\quad (2 + [\![x - 1 > 0 : true]\!] \cdot ([\![x - n > 0 : true]\!] \cdot 1 + \sum_{i=1}^{n-1} 2 \cdot [\![x - i > 0 : true]\!])) \\ &= 1 + [\![x > 0 : true]\!] \cdot (2 + [\![x - n > 0 : true]\!] \cdot 1 + \sum_{i=1}^{n-1} 2 \cdot [\![x - i > 0 : true]\!]) \\ &= 1 + [\![x > 0 : true]\!] \cdot ([\![x - n > 0 : true]\!] \cdot 1 + \sum_{i=0}^{n-1} 2 \cdot [\![x - i > 0 : true]\!]) \\ &= I_{n+1} \end{aligned}$$

□

Since I_n is an upper and lower ω-invariant, we can conclude

$$\begin{aligned} &\Rightarrow \text{ert}[\textbf{while}(x > 0)\{x := x - 1\}](0) \\ &= \lim_{n \to \infty} I_n \\ &= 1 + [\![x > 0 : true]\!] \cdot 2x. \end{aligned}$$

We can see that the expected runtime of $Comp_2$ is linear in the size of x and thus is definitely finite.

Comp$_1$; Comp$_2$

Now we are going to show, that $Comp_1;Comp_2$ has an infinite expected runtime: At first we analyze the loop body of $Comp_1$. As we have seen in section 8.10,

$$\begin{aligned}&\text{ert}[\textbf{while}(b=1)\{x:=x*x*2;\{b:=0\}[\frac{1}{2}]\{b:=1\}\}](X)\\&=3+\frac{1}{2}\cdot\sum_{j=0}^{1}(\lambda(s,h)_{\bullet}X(s[b\backslash j,x\backslash 2\cdot x^2],h)).\end{aligned}$$

Therefore we have

$$\begin{aligned}\Phi_1 : X \mapsto\ & 1+[\![b=1:true]\!]\cdot(3+\frac{1}{2}\cdot\sum_{j=0}^{1}(\lambda(s,h)_{\bullet}X(s[b\backslash j,x\backslash 2\cdot x^2],h)))\\&+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x).\end{aligned}$$

As a lower ω-invariant we suggest

$$\begin{aligned}I_n =\ & 1+[\![b=1:true]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x\cdot(n-1)+8-\frac{6}{2^{n-1}})\\&+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x)\end{aligned}$$

As suggested in [10], we suggest an upper ω-invariant with the help of the following template:
$I_n = 1+[\![b=1:true]\!]\cdot(a_n+b_n\cdot[\![x>0:true]\!]\cdot 2x)+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x)$.
Since we want to obtain
$\Phi(0) = 1+[\![b=1:true]\!]\cdot 3+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x)\geq I_1$,
we know that $a_1\leq 3$ and $b_1=0$.
$\Phi(I_n) = 1+[\![b=1:true]\!]\cdot(\frac{9}{2}+\frac{a_n}{2}+(b_n+1)\cdot[\![x>0]\!]\cdot 2x)+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x)\geq I_{n+1}$ implies that $a_{n+1}\leq\frac{9}{2}+\frac{a_n}{2}$ and $b_{n+1}\leq b_n+1$.
$b_n=n-1$ and $a_n=9-\frac{6}{2^{n-1}}$ meet these requirements.

Proof.

$$\begin{aligned}\Phi_1(0) &= 1+[\![b=1:true]\!]\cdot 3+[\![b=1:false]\!]\cdot(1+[\![x>0:true]\!]\cdot 2x)\\&=I_1\end{aligned}$$

$$\begin{aligned}
\Phi_1(I_n) &= 1 + [\![b = 1 : true]\!] \cdot (3 + \frac{1}{2} \cdot (2 + [\![x^2 > 0 : true]\!] \cdot 4x^2) + \frac{1}{2} \cdot (2 + [\![x^2 > 0 : true]\!] \cdot \\
&\quad 4x^2 \cdot (n-1) + 8 - \frac{6}{2^{n-1}}) + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&= 1 + [\![b = 1 : true]\!] \cdot (5 + [\![x^2 > 0 : true]\!] \cdot 2x^2 + [\![x^2 > 0 : true]\!] \cdot 2x^2 \cdot (n-1) \\
&\quad + 4 - \frac{6}{2^n}) + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&= 1 + [\![b = 1 : true]\!] \cdot ([\![x^2 > 0 : true]\!] \cdot 2x^2 + [\![x^2 > 0 : true]\!] \cdot 2x^2 \cdot (n-1) \\
&\quad + 9 - \frac{6}{2^n}) + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&= 1 + [\![b = 1 : true]\!] \cdot (1 + [\![x^2 > 0 : true]\!] \cdot 2x^2 \cdot n + 8 - \frac{6}{2^n}) \\
&\quad + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&\geq 1 + [\![b = 1 : true]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x^2 \cdot n + 8 - \frac{6}{2^n}) \\
&\quad + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&\text{Since } x \in \mathbb{Z} \text{ we know that } x^2 \geq x: \\
&\geq 1 + [\![b = 1 : true]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x \cdot n + 8 - \frac{6}{2^n}) \\
&\quad + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&= I_{n+1}
\end{aligned}$$

□

Since I_n is an upper ω-invariant, we can conclude

$$\begin{aligned}
&\Rightarrow \mathrm{ert}[\textbf{while}(b = 1)\{x := x * x * 2; \{b := 1\}[\frac{1}{2}]\{b := 0\}\}](1 + [\![x > 0]\!] \cdot 2x) \\
&\geq \lim_{n\to\infty} I_n \\
&= \lim_{n\to\infty} 1 + [\![b = 1 : true]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x \cdot n + 8 - \frac{6}{2^{n-1}}) \\
&\quad + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x) \\
&= 1 + [\![b = 1 : true]\!] \cdot (9 + [\![x > 0 : true]\!] \cdot \infty) + [\![b = 1 : false]\!] \cdot (1 + [\![x > 0 : true]\!] \cdot 2x).
\end{aligned}$$

Now we can analyze the whole program $Comp_1; Comp_2$:

```
  \\∞
  \\ ≥ 3 + ∞
  \\ ≥ 3 + ⟦1 = 1 : true⟧ · ∞ + ⟦1 = 1 : false⟧ · 3
1 b := 1;
  \\ ≥ 2 + ⟦b = 1 : true⟧ · ∞ + ⟦b = 1 : false⟧ · 3
  \\ ≥ 2 + ⟦b = 1 : true⟧ · (9 + ⟦1 > 0 : true⟧) · ∞)
       +⟦b = 1 : false⟧ · (1 + ⟦1 > 0 : true⟧ · 2)
2 x := 1;
  \\ ≥ 1 + ⟦b = 1 : true⟧ · (9 + ⟦x > 0 : true⟧ · ∞)
       +⟦b = 1 : false⟧ · (1 + ⟦x > 0 : true⟧ · 2x)
3 while(b = 1){
4       x := x * x * 2;
5       {b := 0} [1/2] {b := 1}
6 }
  \\1 + ⟦x > 0 : true⟧ · 2x
7 while(x > 0){
8       x := x − 1
9 }
  \\0
```

Listing 8.13: Analysing $Comp_1; Comp_2$

As we can see, the expected runtime of $Comp_1; Comp_2$ is at least infinite, which means it is infinite.

8.3.5 Alloc

Our next example allocates a new heap cell and stores the address. Then it decrements the address until it is equal zero or stops immediately if the address is a negative number. As an address is an arbitrarily great natural number, each program run has to terminate after a finite amount of time. This means that the program terminates after a finite amount of time with probability one, but the expected runtime is infinite.

```
1 x :=new(0);
2 while(x > 0){
3       x := x − 1;
4 }
```

Listing 8.14: *Alloc*

$\Phi : X \mapsto 1 + [\![x > 0 : true]\!] \cdot (1 + \lambda(s,h)_{\bullet} X(s[x \backslash x-1], h))$ is the characteristic function of the while loop.
Since we want to show, that the expected runtime is infinite, we will use a lower ω-invariant and suggest $I_n = \sum_{i=0}^{n-1} [\![x - i > 0 : true]\!] \cdot 2$.

Proof.

$$\begin{aligned}\Phi(0) &= 1 + [\![x > 0 : true]\!] \cdot 1 \\ &\geq [\![x > 0 : true]\!] \cdot 2 \\ &= I_1\end{aligned}$$

$$\begin{aligned}\Phi(I_n) &= 1 + [\![x > 0 : true]\!] \cdot (1 + \sum_{i=0}^{n-1} [\![x - 1 - i > 0 : true]\!] \cdot 2) \\ &= 1 + [\![x > 0 : true]\!] \cdot (1 + \sum_{i=1}^{n} [\![x - i > 0 : true]\!] \cdot 2) \\ &\geq [\![x > 0 : true]\!] \cdot (2 + \sum_{i=1}^{n} [\![x - i > 0 : true]\!] \cdot 2) \\ &= [\![x > 0 : true]\!] \cdot 2 + \sum_{i=1}^{n} [\![x - i > 0 : true]\!] \cdot 2 \\ &= \sum_{i=0}^{n} [\![x - i > 0 : true]\!] \cdot 2 \\ &= I_{n+1}\end{aligned}$$

□

As I_n is a lower ω-invariant for the while-loop, we can conclude

$$\begin{aligned}&\Rightarrow \mathrm{ert}[\mathbf{while}(x > 0)\{x := x - 1\}](0) \\ &\quad\geq \lim_{n\to\infty} I_n \\ &\quad= [\![x > 0 : true]\!] \cdot 2x.\end{aligned}$$

```
   \\ ∞
   \\ ≥ ∞
   \\ ≥ 1 + ∞
   \\ ≥ 1 + sup_{u∈ℕ>0} [u ↦ 0] −★ 2u
   \\ ≥ 1 + sup_{u∈ℕ>0} [u ↦ 0] −★ ([[u > 0 : true]] · 2u)
1  x := new(0) ;
   \\ ≥ [[x > 0 : true]] · 2x
2  while(x > 0){
3      x := x − 1;
4  }
   \\ 0
```

Listing 8.15: Analysis of *Alloc*

8.3.6 Random List Walk

A popular example for infinite runtime is the one-dimensional random walk [10]: It is a program which chooses randomly if it increments or decrements a variable until said variable is not positive anymore. In this thesis we are going to examine a different random walk: The *Random List Walk.* The program builds a short double linked list just consisting of two elements, the head h and the current element x. Each element stores the address of the element before and of the next one. The tail always stores zero instead of the address of the next element. We walk either left or right and extend the list if we reach the tail t. The program terminates when we reach the head h.

```
1   h :=new(0,0);
2   x :=new (h,0);
3   < h+1 >:= x;
4   while(x ≠ h){
5       {temp :=< x+1 >} [1/2] {temp :=< x >};
6       if(temp = 0){
7           temp :=new(x,0);
8           < x+1 >:= temp
9       }
10      x := temp
11  }
```

Listing 8.16: *Random List Walk*

For analyzing this program we will introduce the following definition for *len*, the length of a double linked list:

Definition 8.3.1. *The length of a list from α to β is defined as*

$$len(\alpha, \beta) = [\![\alpha \neq \beta : true]\!] \cdot \sup_{\gamma \in \mathbb{Z}} [\gamma + 1 \mapsto \beta] \bigstar (1 + len(\alpha, \gamma)).$$

Lemma 8.3.2. *The length of a list can also be computed with the help of a longer list if it is defined:* $len(\alpha, \beta) = [\![\alpha \neq \beta : true]\!] \cdot \sup_{\gamma \in \mathbb{Z}} [\gamma \mapsto \beta] \bigstar (len(\alpha, \gamma) - 1)$.

At first we will analyze the loop body.

	$\backslash\!\backslash 1 + \frac{1}{2} \cdot (1 + \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \bigstar ([x+1 \mapsto v] \mathrel{-\!\!\bigstar} (1 + [\![v = 0 : true]\!] \cdot (1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)]$ $\mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u, temp\backslash v], h)))))$ $+ [\![v = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet X(s[x\backslash v, temp\backslash v], h))))) + \frac{1}{2} \cdot (1 + \sup_{w\in\mathbb{Z}} [x \mapsto w]$ $\bigstar([x \mapsto w] \mathrel{-\!\!\bigstar} (1 + [\![w = 0 : true]\!] \cdot (1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _]$ $\bigstar([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u, temp\backslash w], h)))))$ $+ [\![w = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet X(s[x\backslash w, temp\backslash w], h)))))$
5	$\{temp := < x+1 >\} \, [\frac{1}{2}] \, \{temp := < x >\};$
	$\backslash\!\backslash 1 + [\![temp = 0 : true]\!] \cdot (1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _]$ $\bigstar([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u], h)))))$ $+ [\![temp = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet X(s[x\backslash temp], h))$
6	**if** $(temp = 0)\,\{$
	$\backslash\!\backslash 1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar$ $([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u], h))))$
7	$temp :=$ **new** $(x, 0);$
	$\backslash\!\backslash 1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto temp] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash temp], h)))$
8	$< x+1 > := temp$
9	$\}$
	$\backslash\!\backslash 1 + \lambda(s,h)_\bullet X(s[x\backslash temp], h)$
10	$x := temp$
	$\backslash\!\backslash X$

Φ is the characteristic function of the while loop:

$$\begin{aligned}\Phi : X \mapsto & 1 + \frac{1}{2} \cdot (1 + \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \bigstar ([x+1 \mapsto v] \mathrel{-\!\!\bigstar} (1 + [\![v = 0 : true]\!] \cdot (1 \\ & + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 \\ & + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u, temp\backslash v], h))))) + [\![v = 0 : false]\!] \cdot (1 \\ & + \lambda(s,h)_\bullet X(s[x\backslash v, temp\backslash v], h))))) + \frac{1}{2} \cdot (1 + \sup_{w\in\mathbb{Z}} [x \mapsto w] \bigstar ([x \mapsto w] \\ & \mathrel{-\!\!\bigstar} (1 + [\![w = 0 : true]\!] \cdot (1 + \sup_{u \in \mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar \\ & ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet X(s[x\backslash u, temp\backslash u, temp\backslash w], h))))) \\ & + [\![w = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet X(s[x\backslash w, temp\backslash w], h)))))\end{aligned}$$

As a lower ω-invariant for the while loop we suggest :
$I_n = 1 + \sum_{k=0}^{n} [\![len(h,x) > k : true]\!] \cdot a_{n,k}$
We use the definition of $a_{n,k}$ presented in [10] in B.1:

$$a_{n,k} = \frac{1}{2^n}\left[-\binom{n}{\lfloor\frac{n-k}{2}\rfloor} + 2 \cdot \sum_{i=0}^{n-k} 2^i \binom{n-i}{\lfloor\frac{n-i-k}{2}\rfloor}\right]$$

Additionally we need the following properties of $a_{n,k}$, which are proven for all $n \in \mathbb{N}$ in [10]:

$$a_{0,0} = 1 \tag{8.1}$$

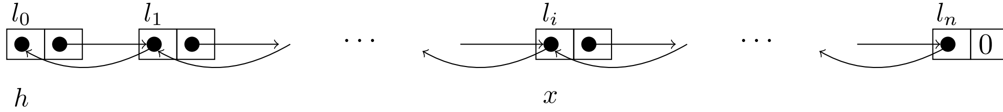
$$a_{n+1,0} = 2 + \frac{1}{2} \cdot (a_{n,0} + a_{n,1}) \tag{8.2}$$

$$a_{n+1,k} = \frac{1}{2} \cdot (a_{n,k-1} + a_{n,k+1}) \text{ for all } 1 \leq k \leq n+1 \tag{8.3}$$

$$a_{n,k} = 0 \text{ for all } k > n \tag{8.4}$$

$$\lim_{n\to\infty} a_{n,0} = \infty \tag{8.5}$$

For proving this ω-invariant, we need the following knowledge about the heap:
After execution of line 5 and before execution of line 6 the heap always looks like this:



The following equations hold at this point of the execution:

$$x = l_i \wedge 0 < i \leq n \tag{8.6}$$

$$a_j \neq 0 \;\; \forall\, 0 \leq j \leq i \tag{8.7}$$

$$a_j \neq h \;\; \forall\, i \leq j \leq n \tag{8.8}$$

This is shown in 10.1.

Proof.

$$\begin{aligned}
\Phi(0) =\, & 1 + \frac{1}{2} \cdot (1 + \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \bigstar ([x+1 \mapsto v] \mathrel{-\!\!\bigstar} (1 + [\![v = 0 : \mathit{true}]\!] \cdot \\
& (1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1+0)))) \\
& + [\![v = 0 : \mathit{false}]\!] \cdot (1+0)))) + \frac{1}{2} \cdot (1 + \sup_{w\in\mathbb{Z}} [x \mapsto w] \bigstar ([x \mapsto w] \\
& \mathrel{-\!\!\bigstar} (1 + [\![w = 0 : \mathit{true}]\!] \cdot (1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \\
& \mathrel{-\!\!\bigstar} (1+0)))) + [\![w = 0 : \mathit{false}]\!] \cdot (1+0)))) \\
& \geq 2 \\
& \overset{8.1}{=} 1 + a_{0,0} \\
& \geq 1 + \sum_{k=0}^{0} [\![len(h,x) > k : \mathit{true}]\!] \cdot a_{n,k} \\
& = I_0
\end{aligned}$$

$$\Phi(I_n) = 1 + \frac{1}{2} \cdot (1 + \sup_{v\in\mathbb{Z}}[x+1 \mapsto v] \bigstar ([x+1 \mapsto v] \mathrel{-\!\!\bigstar} (1 + [\![v = 0 : true]\!] \cdot$$
$$(1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u]$$
$$\mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet I_n(s[x\backslash u], h)))))) + [\![v = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet I_n(s[x\backslash v], h)))))$$
$$+ \frac{1}{2} \cdot (1 + \sup_{w\in\mathbb{Z}}[x \mapsto w] \bigstar ([x \mapsto w] \mathrel{-\!\!\bigstar} (1 + [\![w = 0 : true]\!] \cdot (1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)]$$
$$\mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet I_n(s[x\backslash u], h)))))$$
$$+ [\![w = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet I_n(s[x\backslash w], h)))))$$

With our knowledge of the heap we can follow:

$$= 1 + \frac{1}{2} \cdot (1 + 1 + [\![l_{i+1} = 0 : true]\!] \cdot (1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar$$
$$([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet I_n(s[x\backslash u], h))))) + [\![l_{i+1} = 0 : false]\!] \cdot (1$$
$$+ \lambda(s,h)_\bullet I_n(s[x\backslash l_{i+1}], h))) + \frac{1}{2} \cdot (1 + 1 + [\![l_{i-1} = 0 : true]\!] \cdot (1$$
$$+ \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1$$
$$+ \lambda(s,h)_\bullet I_n(s[x\backslash u], h))))) + [\![l_{i-1} = 0 : false]\!] \cdot (1 + \lambda(s,h)_\bullet I_n(s[x\backslash l_{i-1}], h)))$$
$$\overset{(8.7)}{=} 1 + \frac{1}{2} \cdot (2 + [\![l_{i+1} = 0 : true]\!] \cdot (1 + \sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar$$
$$([x+1 \mapsto u] \mathrel{-\!\!\bigstar} (1 + \lambda(s,h)_\bullet I_n(s[x\backslash u], h))))) + [\![l_{i+1} = 0 : false]\!] \cdot$$
$$(1 + \lambda(s,h)_\bullet I_n(s[x\backslash l_{i+1}], h))) + \frac{1}{2} \cdot (3 + \lambda(s,h)_\bullet I_n(s[x\backslash l_{i-1}], h))$$
$$= 4 + \frac{1}{2} \cdot ([\![l_{i+1} = 0 : true]\!] \cdot (\sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u]$$
$$\mathrel{-\!\!\bigstar} (2 + \sum_{k=0}^{n} [\![len(h,u) > k : true]\!] \cdot a_{n,k})))) + [\![l_{i+1} = 0 : false]\!] \cdot$$
$$(1 + \sum_{k=0}^{n} [\![len(h,l_{i+1}) > k : true]\!] \cdot a_{n,k}) + 1 + \sum_{k=0}^{n} [\![len(h,l_{i-1}) > k : true]\!] \cdot a_{n,k})$$
$$\overset{8.3.1}{=} 4 + \frac{1}{2} \cdot ([\![l_{i+1} = 0 : true]\!] \cdot (\sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u]$$
$$\mathrel{-\!\!\bigstar} (2 + \sum_{k=0}^{n} [\![[\![h \neq u : true]\!] \cdot \sup_{\gamma\in\mathbb{Z}} [\gamma+1 \mapsto u] \bigstar (1 + len(h,\gamma)) > k : true]\!] \cdot a_{n,k}))))$$
$$+ [\![l_{i+1} = 0 : false]\!] \cdot (1 + \sum_{k=0}^{n} [\![len(h,l_{i+1}) > k : true]\!] \cdot a_{n,k}) + 1$$
$$+ \sum_{k=0}^{n} [\![len(h,l_{i-1}) > k : true]\!] \cdot a_{n,k})$$
$$= 4 + \frac{1}{2} \cdot ([\![l_{i+1} = 0 : true]\!] \cdot (\sup_{u\in\mathbb{N}_{>0}} [u \mapsto (x,0)] \mathrel{-\!\!\bigstar} (1 + [x+1 \mapsto _] \bigstar ([x+1 \mapsto u] \mathrel{-\!\!\bigstar}$$
$$(2 + \sum_{k=0}^{n} [\![[x+1 \mapsto u] \bigstar (1 + len(h,x)) > k : true]\!] \cdot a_{n,k})))) + [\![l_{i+1} = 0 : false]\!] \cdot$$
$$(1 + \sum_{k=0}^{n} [\![len(h,l_{i+1}) > k : true]\!] \cdot a_{n,k}) + 1 + \sum_{k=0}^{n} [\![len(h,l_{i-1}) > k : true]\!] \cdot a_{n,k})$$

$$= 4 + \frac{1}{2} \cdot (\llbracket l_{i+1} = 0 : true \rrbracket \cdot (3 + \sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ \llbracket l_{i+1} = 0 : false \rrbracket \cdot (1 + \sum_{k=0}^{n} \llbracket len(h, l_{i+1}) > k : true \rrbracket \cdot a_{n,k}) + 1$$

$$+ \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$\overset{8.3.1}{=} 4 + \frac{1}{2} \cdot (\llbracket l_{i+1} = 0 : true \rrbracket \cdot (3 + \sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ \llbracket l_{i+1} = 0 : false \rrbracket \cdot (1 + \sum_{k=0}^{n} \llbracket \llbracket h \neq l_{i+1} : true \rrbracket \cdot \sup_{\gamma \in \mathbb{Z}} [\gamma + 1 \mapsto l_{i+1}]$$

$$\bigstar (1 + len(h, \gamma)) > k : true \rrbracket \cdot a_{n,k}) + 1 + \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$\overset{(8.8)}{=} 4 + \frac{1}{2} \cdot (\llbracket l_{i+1} = 0 : true \rrbracket \cdot (3 + \sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ \llbracket l_{i+1} = 0 : false \rrbracket \cdot (1 + \sum_{k=0}^{n} \llbracket \sup_{\gamma \in \mathbb{Z}} [\gamma + 1 \mapsto l_{i+1}] \bigstar (1 + len(h, \gamma)) > k : true \rrbracket \cdot a_{n,k})$$

$$+ 1 + \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$= 4 + \frac{1}{2} \cdot (\llbracket l_{i+1} = 0 : true \rrbracket \cdot (3 + \sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ \llbracket l_{i+1} = 0 : false \rrbracket \cdot (1 + \sum_{k=0}^{n} \llbracket [x + 1 \mapsto l_{i+1}] \bigstar (1 + len(h, x)) > k : true \rrbracket \cdot a_{n,k})$$

$$+ 1 + \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$= 4 + \frac{1}{2} \cdot (\llbracket l_{i+1} = 0 : true \rrbracket \cdot (3 + \sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ \llbracket l_{i+1} = 0 : false \rrbracket \cdot (1 + \sum_{k=0}^{n} \llbracket len(h, x)) > k - 1 : true \rrbracket \cdot a_{n,k})$$

$$+ 1 + \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$\geq 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k} + \sum_{k=0}^{n} \llbracket len(h, l_{i-1}) > k : true \rrbracket \cdot a_{n,k})$$

$$\overset{8.3.2}{=} 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} \llbracket len(h, x) > k - 1 : true \rrbracket \cdot a_{n,k} +$$

$$\sum_{k=0}^{n} \llbracket \llbracket h \neq l_{i-1} : true \rrbracket \cdot \sup_{\gamma \in \mathbb{Z}} [\gamma \mapsto l_{i-1}] \bigstar (len(h, \gamma) - 1) > k : true \rrbracket \cdot a_{n,k})$$

$$= 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} [\![len(h,x) > k-1 : true]\!] \cdot a_{n,k} + \sum_{k=0}^{n} [\![[\![h \neq l_{i-1} : true]\!] \cdot [x \mapsto l_{i-1}] \bigstar (len(h,x)-1) > k : true]\!] \cdot a_{n,k})$$

If $h = l_{i-1}$, then $i = 1$ and $x = l_1$. This means

$$\begin{aligned} len(h,x) &= [\![h \neq x : true]\!] \cdot \sup_{\gamma \in \mathbb{Z}} [\gamma + 1 \mapsto x] \bigstar (1 + len(h,\gamma)) \\ &= [\![h \neq x : true]\!] \cdot [h+1 \mapsto x] \bigstar (1 + len(h,h)) \\ &= 1. \end{aligned}$$

So $[\![len(h,x) - 1 > k]\!]$ is false for all $k \geq 0$. So we can use that

$$\begin{aligned} &\sum_{k=0}^{n} [\![([\![h \neq l_{i-1} : true]\!] + [\![h \neq l_{i-1} : false]\!]) \cdot [x \mapsto l_{i-1}] \bigstar (len(h,x)-1) > k : true]\!] \cdot a_{n,k}) \\ = &\sum_{k=0}^{n} [\![[\![h \neq l_{i-1} : true]\!] \cdot [x \mapsto l_{i-1}] \bigstar (len(h,x)-1) > k : true]\!] \cdot a_{n,k}). \end{aligned}$$

$$\begin{aligned} &5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} [\![len(h,x) > k-1 : true]\!] \cdot a_{n,k} + \sum_{k=0}^{n} [\![[\![h \neq l_{i-1} : true]\!] \cdot [x \mapsto l_{i-1}] \bigstar \\ &(len(h,x)-1) > k : true]\!] \cdot a_{n,k}) \\ &= 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} [\![len(h,x) > k-1 : true]\!] \cdot a_{n,k} + \sum_{k=0}^{n} [\![([\![h \neq l_{i-1} : true]\!] \\ &\quad + [\![h \neq l_{i-1} : false]\!]) \cdot [x \mapsto l_{i-1}] \bigstar (len(h,x)-1) > k : true]\!] \cdot a_{n,k}) \\ &= 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} [\![len(h,x) > k-1 : true]\!] \cdot a_{n,k} \\ &\quad + \sum_{k=0}^{n} [\![[x \mapsto l_{i-1}] \bigstar (len(h,x)-1) > k : true]\!] \cdot a_{n,k}) \\ &= 5 + \frac{1}{2} \cdot (\sum_{k=0}^{n} [\![len(h,x) > k-1 : true]\!] \cdot a_{n,k} + \sum_{k=0}^{n} [\![len(h,x) > k+1 : true]\!] \cdot a_{n,k}) \\ &= 5 + \frac{1}{2} \cdot (\sum_{k=-1}^{n-1} [\![len(h,x) > k : true]\!] \cdot a_{n,k+1} + \sum_{k=1}^{n+1} [\![len(h,x) > k : true]\!] \cdot a_{n,k-1}) \\ &= 5 + \frac{1}{2} \cdot (\sum_{k=1}^{n-1} [\![len(h,x) > k : true]\!] \cdot (a_{n,k+1} + a_{n,k-1}) \\ &\quad + [\![len(h,x) > -1 : true]\!] \cdot a_{n,0} + [\![len(h,x) > 0 : true]\!] \cdot a_{n,1} \\ &\quad + [\![len(h,x) > n : true]\!] \cdot a_{n,n-1} \\ &\quad + [\![len(h,x) > n+1 : true]\!] \cdot a_{n,n}) \end{aligned}$$

$$
\begin{aligned}
&\geq 5 + \frac{1}{2} \cdot (\sum_{k=1}^{n-1} [\![len(h,x) > k : true]\!] \cdot (a_{n,k+1} + a_{n,k-1}) \\
&\quad + [\![len(h,x) > 0 : true]\!] \cdot a_{n,0} + [\![len(h,x) > 0 : true]\!] \cdot a_{n,1} \\
&\quad + [\![len(h,x) > n : true]\!] \cdot (a_{n,n-1} + 0) \\
&\quad + [\![len(h,x) > n+1 : true]\!] \cdot (a_{n,n} + 0)) \\
&\overset{8.4}{=} 5 + \frac{1}{2} \cdot (\sum_{k=1}^{n-1} [\![len(h,x) > k : true]\!] \cdot (a_{n,k+1} + a_{n,k-1}) \\
&\quad + [\![len(h,x) > 0 : true]\!] \cdot (a_{n,0} + a_{n,1}) \\
&\quad + [\![len(h,x) > n : true]\!] \cdot (a_{n,n-1} + a_{n,n+1}) \\
&\quad + [\![len(h,x) > n+1 : true]\!] \cdot (a_{n,n} + a_{n,n+2})) \\
&\overset{8.2}{=} 3 + \frac{1}{2} \cdot \sum_{k=1}^{n+1} [\![len(h,x) > k : true]\!] \cdot (a_{n,k+1} + a_{n,k-1}) + [\![len(h,x) > 0 : true]\!] \cdot a_{n+1,0} \\
&\overset{8.3}{=} 3 + \sum_{k=1}^{n+1} [\![len(h,x) > k : true]\!] \cdot a_{n+1,k} + [\![len(h,x) > 0 : true]\!] \cdot a_{n+1,0} \\
&= 3 + \sum_{k=0}^{n+1} [\![len(h,x) > k : true]\!] \cdot a_{n+1,k} \\
&\geq 1 + \sum_{k=0}^{n+1} [\![len(h,x) > k : true]\!] \cdot a_{n+1,k} \\
&= I_{n+1}
\end{aligned}
$$

□

Now we have proven, that I_n is indeed a lower ω-invariant for the while loop in *Random List Walk*, which we will call c_{loop}.

$$
\begin{aligned}
\Rightarrow \mathrm{ert}[c_{loop}](0) &\geq \lim_{n\to\infty} I_n \\
&= \lim_{n\to\infty} 1 + \sum_{k=0}^{n} [\![len(h,x) > k : true]\!] \cdot a_{n,k} \\
&\geq \lim_{n\to\infty} 1 + [\![len(h,x) > 0 : true]\!] \cdot a_{n,0} \\
&= 1 + [\![len(h,x) > 0 : true]\!] \cdot \lim_{n\to\infty} a_{n,0} \\
&\overset{8.5}{=} 1 + [\![len(h,x) > 0 : true]\!] \cdot \infty
\end{aligned}
$$

```
     \\ ∞
     \\ ≥ ∞
     \\ ≥ 4 + ∞
     \\ ≥ 1 + sup_{w∈N>0} [w ↦ (0,0)] -★ (1 + sup_{u∈N>0} [u ↦ (w,0)] -★ (1 + [w + 1 ↦ _] ★ ([w + 1 ↦ u]
        -★ (1 + [[1 > 0 : true]] · ∞))))
     \\ ≥ 1 + sup_{w∈N>0} [w ↦ (0,0)] -★ (1 + sup_{u∈N>0} [u ↦ (w,0)] -★ (1 + [w + 1 ↦ _] ★ ([w + 1 ↦ u]
        -★ (1 + [[[[w ≠ u : true]] · [w + 1 ↦ u]★(1 + len(w,w)) > 0 : true]] · ∞))))
     \\ ≥ 1 + sup_{w∈N>0} [w ↦ (0,0)] -★ (1 + sup_{u∈N>0} [u ↦ (w,0)] -★ (1 + [w + 1 ↦ _] ★ ([w + 1 ↦ u]
        -★ (1 + [[[[w ≠ u : true]] · sup_{γ∈Z}[γ + 1 ↦ u]★(1 + len(w,γ)) > 0 : true]] · ∞))))
     \\ ≥ 1 + sup_{w∈N>0} [w ↦ (0,0)] -★ (1 + sup_{u∈N>0} [u ↦ (w,0)] -★ (1 + [w + 1 ↦ _] ★ ([w + 1 ↦ u]
        -★ (1 + [[len(w,u) > 0 : true]] · ∞))))
 1   h := new(0,0);
     \\ ≥ 1 + sup_{u∈N>0} [u ↦ (h,0)] -★ (1 + [h + 1 ↦ _] ★ ([h + 1 ↦ u]
        -★ (1 + [[len(h,u) > 0 : true]] · ∞)))
 2   x := new(h,0);
     \\ ≥ 1 + [h + 1 ↦ _] ★ ([h + 1 ↦ x] -★ (1 + [[len(h,x) > 0 : true]] · ∞))
 3   < h + 1 > := x;
     \\ ≥ 1 + [[len(h,x) > 0 : true]] · ∞
 4   while(x ≠ h){
 5       {temp := < x + 1 >} [1/2] {temp := < x >};
 6       if(temp = 0){
 7           temp := new(x,0);
 8           < x + 1 > := temp
 9       }
10       x := temp
11   }
     \\ 0
```

Listing 8.17: Analyzing *Random List Walk*

Chapter 9

Conclusion

As we have seen, the presented calculus for expected runtimes is sound with respect to the operational model and complete by definition. To our knowledge it uses the first expected runtime transformer for probabilistic pointer programs. We have successfully used our calculus to analyze a variety of program examples. The calculus can not only compute finite but also infinite expected runtimes. Even though all those examples were small, it could still be observed that an exact computation of the expected runtime is not helpful. Upper bounds for expected runtimes can be computed less costly and are more meaningful, but for big real life examples it is not pleasant to develop those by hand. Therefore automatization of the analysis is necessary. Apart from the analysis of loops, all rules can be easily performed automatically. The analysis of loops on the other hand can be very involved as we have seen in section 8.3.6 while analyzing the random list walk. For analyzing a loop we always need to find or at least approximate the least fixedpoint of the characteristic function. Those fixedpoints are often easier to find, when we have an understanding of the program's purpose. Since the analysis of loops often requires creative thinking or knowledge about the program's purpose, it is not trivial to automatize this process. Although it is not easy to analyze an arbitrary while loop, certain groups of while loops can already be analyzed automatically like presented in [2].

Bibliography

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[2] K. Batz. Proof rules for expected run-times of probabilistic programs. Bachelor Thesis, 2017.

[3] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and T. Noll. Quantitative separation logic: A logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.*, 3(POPL):34:1–34:29, Jan. 2019.

[4] G. Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. American Mathematical Society, 1940.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. Oldenbourg Verlag, 2010.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[8] D. Draheim. *Semantics of the Probabilistic Typed Lambda Calculus*. Springer-Verlag, 2017.

[9] B. L. Kaminski, J.-P. Katoen, and C. Matheja. Analyzing expected runtimes by program verification. Unpublished manuscript.

[10] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. *arXiv e-prints*, page arXiv:1601.01001, Jan 2016.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[12] J. W. Sanders and P. Zuliani. Quantum programming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2000.

[13] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

Chapter 10

Appendix

Lemma 10.0.1. *ert is ω-continuous.*

Proof. We have to prove the following:

- $ert[C](\sup_n f_i) = \sup_n ert[C](f_i)$ for every ω-chain $\{f_i \mid i \in \omega\} \subseteq \mathbb{T}$ and every program $C \in \mathcal{P}^3$
- ert is monotone.

$\sup_n f_i$ and $\sup_n ert[C](f_i)$ always exist, since $(\mathbb{T}, \preceq)$ is a complete lattice.
We will prove the first property by induction over the structure of program $C \in \mathcal{P}^3$:
<u>**Base Case:**</u>
Empty program: $C = \textbf{empty}$
This case has already been shown in [10].

Assignment: $C = x := e$

$$
\begin{aligned}
&ert[x := e](\sup_n f_n) \\
&= 1 + \lambda\,(s,h)_\bullet \sup_n f_n\,(s\,[x \backslash s(e)]\,,\,h) \\
&= \sup_n 1 + \lambda\,(s,h)_\bullet\, f_n\,(s\,[x \backslash s(e)]\,,\,h) && \text{(1 is const. for n)} \\
&= \sup_n ert[x := e](f_n)
\end{aligned}
$$

Probabilistic assignment: $C = x := Uniform(u, v)$

$$
\begin{aligned}
&ert[x := Uniform(u,v)](\sup_n f_n) \\
&= 1 + \frac{1}{v-u+1} \cdot \sum_{i=u}^{v} \lambda(s,h)_\bullet \sup_n f_n\,(s[x \backslash i], h) \\
&= \sup_n 1 + \frac{1}{v-u+1} \cdot \sum_{i=u}^{v} \lambda(s,h)_\bullet\, f_n\,(s[x \backslash i], h) && \text{(1 is const. for n)} \\
&= \sup_n ert[x := Uniform(u,v)](f_n)
\end{aligned}
$$

Heap manipulation: $C = \,< e > := e'$

$ert[< e > := e'](\sup_n f_n)$

$= 1 + [e \mapsto _] \bigstar ([e \mapsto e'] \mathrel{-\!\!\bigstar} \sup_n f_n)$

$= \sup_n 1 + [e \mapsto _] \bigstar ([e \mapsto e'] \mathrel{-\!\!\bigstar} f_n)$ (1 and the predicates are const. for n)

$= \sup_n ert[< e > := e'](f_n)$

Heap lookup: $C = x := < e >$

$ert[x := < e >](\sup_n f_n)$

$= 1 + \lambda(s,h)\bullet \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \mathrel{-\!\!\bigstar} \sup_n f_n(s[x \backslash v], h))$

$= \sup_n 1 + \lambda(s,h)\bullet \sup_{v \in \mathbb{Z}} [e \mapsto v] \bigstar ([e \mapsto v] \mathrel{-\!\!\bigstar} f_n(s[x \backslash v], h))$

(1 and the predicates are const. for n)

$= \sup_n ert[x := < e >](f_n)$

Allocation: $C = x := new(\vec{e})$

$ert[x := new(\vec{e})](\sup_n f_n)$

$= 1 + \lambda(s,h)\bullet \sup_{u \in \mathbb{N}_{>0}} [u \mapsto \vec{e}] \mathrel{-\!\!\bigstar} \sup_n f_n(s[x \backslash u], h)$

$= \sup_n 1 + \lambda(s,h)\bullet \sup_{u \in \mathbb{N}_{>0}} [u \mapsto \vec{e}] \mathrel{-\!\!\bigstar} f_n(s[x \backslash u], h)$ (1 and the predicate are const. for n)

$= \sup_n ert[x := new(\vec{e})](f_n)$

Deallocation: $C = free(e)$

$ert[free(e)](\sup_n f_n)$

$= 1 + [e \mapsto _] \bigstar \sup_n f_n$

$= \sup_n 1 + [e \mapsto _] \bigstar f_n$ (1 and the predicate are const. for n)

$= \sup_n ert[free(e)](f_n)$

Induction Hypothesis:

All subprograms $C' \in \mathcal{P}^3$ of C fulfill

$$ert[C'](\sup_n f_i) = \sup_n ert[C'](f_i)$$

for every ω-chain $\{f_i \mid i \in \omega\} \subseteq \mathbb{T}$.

Inductive Step:
Sequential Composition: $C = c_1; c_2$
This case has already been shown in [10].

Conditional Choice: $C = \textbf{if}\,(b)\,\{c_1\}\,\textbf{else}\,\{c_2\}$
This case has already been shown in [10].

Probabilistic choice: $C = \{c_1\}\,[p]\,\{c_2\}$

$$
\begin{aligned}
& ert[\{c_1\}\,[p]\,\{c_2\}](\sup_n f_n) \\
& = 1 + p \cdot \mathrm{ert}[c_1](\sup_n f_n) + (1-p) \cdot \mathrm{ert}[c_2](\sup_n f_n) \\
& = 1 + p \cdot \sup_n \mathrm{ert}[c_1](f_n) + (1-p) \cdot \sup_n \mathrm{ert}[c_2](f_n) && \text{(I.H.)} \\
& = \sup_n 1 + p \cdot \mathrm{ert}[c_1](f_n) + (1-p) \cdot \mathrm{ert}[c_2](f_n) && (1, p \text{ and } 1-p \text{ are const. for n}) \\
& = \sup_n ert[\{c_1\}\,[p]\,\{c_2\}](f_n)
\end{aligned}
$$

Loop: $C = \textbf{while}\,(b)\,\{c'\}$
This case has already been shown in [10].

Now we have to prove the monotony of ert:
We consider $f_1, f_2 \in \mathbb{T}$ with $f_1 \preceq f_2$ and an arbitrary program $C \in \mathcal{P}^3$. We have to show, that $ert[C](f_1) \preceq ert[C](f_2)$ holds. $f_1 \preceq f_2$ builds an ω-chain and $\sup_{n\in\{1,2\}} f_n = f_2$. Then we can conclude:

$$
\begin{aligned}
& \sup_{n\in\{1,2\}} ert[C](f_n) \\
& = ert[C](\sup_{n\in\{1,2\}} f_n) && (\omega - continuity) \\
& = ert[C](f_2).
\end{aligned}
$$

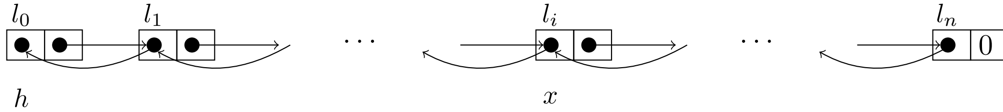
And with $ert[C](f_1) \preceq \sup_{n\in\{1,2\}} ert[C](f_n)$ we can conclude: $ert[C](f_1) \preceq ert[C](f_2)$. □

10.1 Random List Walk

```
1  h :=new(0,0);
2  x :=new (h,0);
3  < h+1 >:= x;
4  while(x ≠ h){
5        {temp :=< x+1 >} [1/2] {temp :=< x >};
6        if(temp = 0){
7              temp :=new(x,0);
8              < x+1 >:= temp
9        }
10       x := temp
11 }
```

Listing 10.1: *Random List Walk*

We will show that the heap looks like this every time before we are going to execute line 5:



The following equations hold at this point of the execution:

$$x = l_i \wedge 0 < i \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i$$
$$l_j \neq h \;\; \forall\, i \leq j \leq n$$

Proof. We will show this by induction:
Base case:
In the following proof we are going to do *forward* reasoning! This means we see a sketch of the heap, then a line of code and beneath that is a sketch of the updated heap. We will start by taking a look at the heap when we reach line 5 for the first time. Before the execution of line 1, our heap is empty.

1 $h := \textbf{new}(0, 0);$

0 0

h

$h \neq 0$

2 $x := \textbf{new}(h, 0);$

0 0 ● 0

h x

$h \neq 0$
$x \neq 0$
$x \neq h$

3 $< h + 1 > := x;$

0 ● ● 0

h x

$h \neq 0$
$x \neq 0$
$x \neq h$

4 $\textbf{while}(x \neq h)\{$

0 ● ● 0

h x

$h \neq 0$
$x \neq 0$
$x \neq h$

As we can see the heap fulfills all equations and has the required structure.

Inductive step:
Now we have to show that a heap, which has the required properties at line 5, still does so, if it reaches line 5 again after an execution of the while-loop. Therefore we have to distinguish between two cases, because the execution of line 5 is not deterministic.
Case 1:

l_0 l_1 … l_i … l_n 0

h x

$$x = l_i \wedge 0 < i \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i$$
$$l_j \neq h \;\; \forall\, i \leq j \leq n$$

5 $\quad temp := < x >;$

l_0 l_1 … l_{i-1} l_i … l_n 0

h $temp$ x

$$x = l_i \wedge 0 < i \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i$$
$$l_j \neq h \;\; \forall\, i \leq j \leq n$$
$$temp = l_{i-1}$$

$0 \leq i - 1 < i \Rightarrow l_{i-1} = temp \neq 0$

6 $\quad$ **if** $(temp = 0)\{$
7 $\quad\quad temp :=$ **new** $(x, 0);$
8 $\quad\quad < x + 1 > := temp$
9 $\quad \}$
10 $\quad x := temp$
11 $\}$

l_0 l_1 … l_{i-1} l_i … l_n 0

h x

$$x = l_{i-1} \wedge 0 < i \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i-1$$
$$l_j \neq h \;\; \forall\, i \leq j \leq n$$

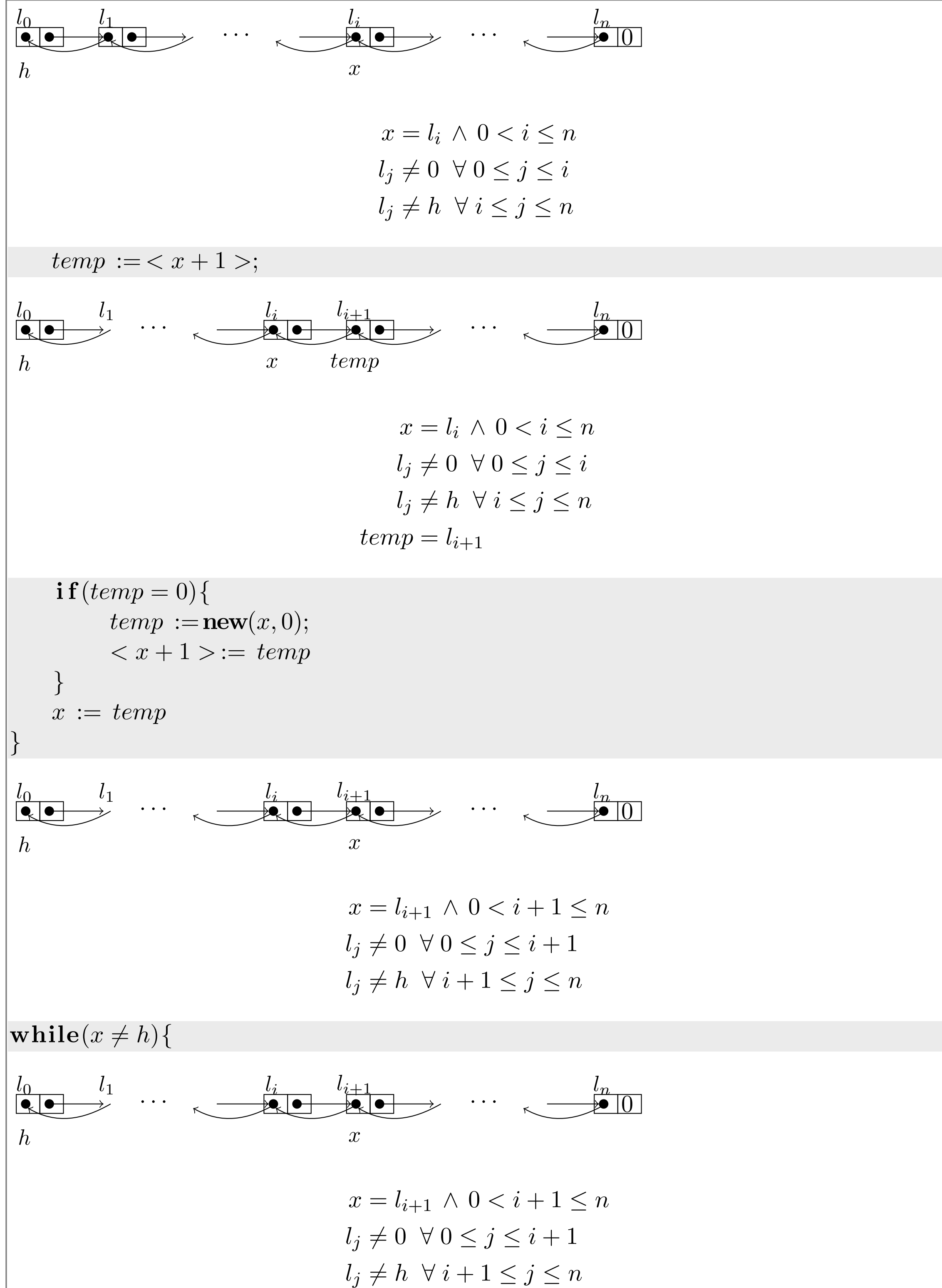
4 **while** $(x \neq h)\{$

l_0 l_1 … l_{i-1} l_i … l_n 0

h x

$$x = l_{i-1} \wedge 0 < i - 1 \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i-1$$
$$l_j \neq h \;\; \forall\, i-1 \leq j \leq n$$

Case 2:
This time also have to distinguish between two subcases, which depend on the value of *temp*.
Case 2a: $temp \neq 0$



$$x = l_i \wedge 0 < i \leq n$$
$$l_j \neq 0 \ \forall\, 0 \leq j \leq i$$
$$l_j \neq h \ \forall\, i \leq j \leq n$$

```
5       temp := < x + 1 >;
```

$$x = l_i \wedge 0 < i \leq n$$
$$l_j \neq 0 \ \forall\, 0 \leq j \leq i$$
$$l_j \neq h \ \forall\, i \leq j \leq n$$
$$temp = l_{i+1}$$

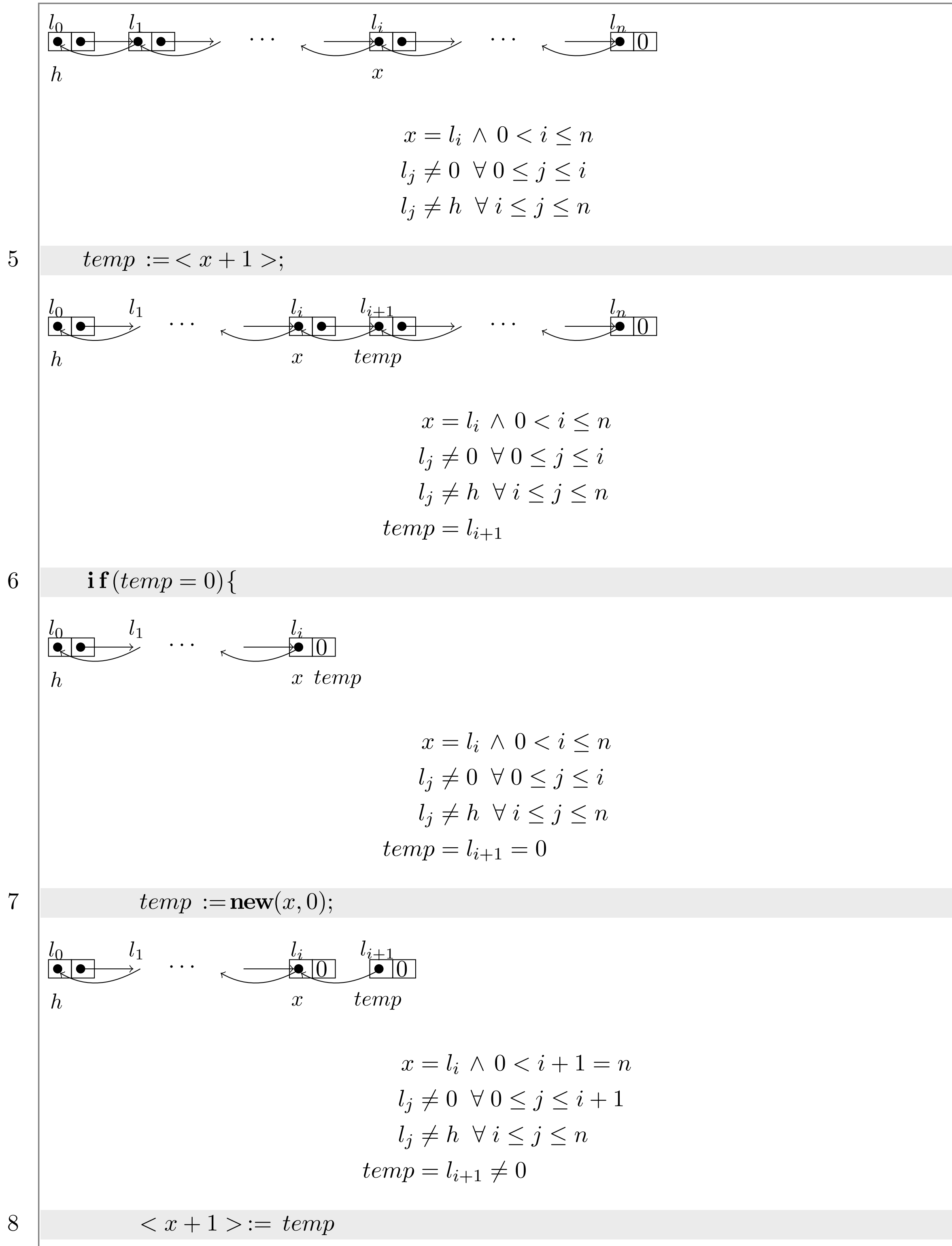
```
6       if(temp = 0){
7           temp := new(x, 0);
8           < x + 1 > := temp
9       }
10      x := temp
11  }
```

$$x = l_{i+1} \wedge 0 < i+1 \leq n$$
$$l_j \neq 0 \ \forall\, 0 \leq j \leq i+1$$
$$l_j \neq h \ \forall\, i+1 \leq j \leq n$$

```
4   while(x ≠ h){
```

$$x = l_{i+1} \wedge 0 < i+1 \leq n$$
$$l_j \neq 0 \ \forall\, 0 \leq j \leq i+1$$
$$l_j \neq h \ \forall\, i+1 \leq j \leq n$$

Case 2b: $temp = 0$



$$x = l_i \wedge 0 < i+1 = n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i+1$$
$$l_i \neq \;\; \forall\, i \leq j \leq n$$
$$temp = l_{i+1} \neq 0$$

```
9      }
10     x := temp
11 }
```


$$x = l_{i+1} \wedge 0 < i+1 = n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i+1$$
$$l_i \neq \;\; \forall\, i \leq j \leq n$$

```
4  while(x ≠ h){
```


$$x = l_{i+1} \wedge 0 < i+1 \leq n$$
$$l_j \neq 0 \;\; \forall\, 0 \leq j \leq i+1$$
$$l_j \neq h \;\; \forall\, i+1 \leq j \leq n$$

□