Weighted Programming

A Programming Paradigm for Specifying Mathematical Models

KEVIN BATZ, RWTH Aachen University, Germany
ADRIAN GALLUS, RWTH Aachen University, Germany
BENJAMIN LUCIEN KAMINSKI, Saarland University, Saarland Informatics Campus, Germany and University College London, United Kingdom
JOOST-PIETER KATOEN, RWTH Aachen University, Germany
TOBIAS WINKLER, RWTH Aachen University, Germany

We study weighted programming, a programming paradigm for specifying mathematical models. More specifically, the weighted programs we investigate are like usual imperative programs with two additional features: (1) nondeterministic *branching* and (2) *weighting* execution traces. Weights can be numbers but also other objects like words from an alphabet, polynomials, formal power series, or cardinal numbers. We argue that weighted programming as a paradigm can be used to specify mathematical models beyond probability distributions (as is done in probabilistic programming).

We develop weakest-precondition- and weakest-liberal-precondition-style calculi à la Dijkstra for reasoning about mathematical models specified by weighted programs. We present several case studies. For instance, we use weighted programming to model the *ski rental problem* — an *optimization problem*. We model not only the optimization problem itself, but also the best deterministic online algorithm for solving this problem as weighted programs. By means of weakest-precondition-style reasoning, we can determine the *competitive ratio* of the online algorithm *on source code level*.

CCS Concepts: • **Theory of computation → Models of computation**; **Programming logic**; **Denotational semantics**; Invariants; Pre- and post-conditions; **Program semantics**.

Additional Key Words and Phrases: weighted programming, denotational semantics, weakest preconditions

ACM Reference Format:
Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022. Weighted Programming: A Programming Paradigm for Specifying Mathematical Models. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 66 (April 2022), 30 pages. https://doi.org/10.1145/3527310

1 INTRODUCTION AND OVERVIEW

Weighted programs are usual programs with two distinct features: (1) nondeterministic *branching* and (2) the ability to *weight* the current execution trace. A prime and very well-studied example of weighted programs are *probabilistic programs* which can branch their execution depending on the outcome of a random coin flip. For instance, the program $\{ C_1 \} \left[\frac{1}{3}\right] \{ C_2 \}$ weights the trace that executes C_1 with probability $1/3$ and the trace executing C_2 with $1 - 1/3 = 2/3$. The weighted outcomes of the two branches are then — simply put — summed together.

Authors' addresses: Kevin Batz, kevin.batz@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Adrian Gallus, adrian.gallus@rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Benjamin Lucien Kaminski, b.kaminski@ucl.ac.uk, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany and University College London, London, United Kingdom; Joost-Pieter Katoen, katoen@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Tobias Winkler, tobias.winkler@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany.

2475-1421/2022/4-ART66
https://doi.org/10.1145/3527310

Besides applications as *randomized algorithms* for speed-up in solving computationally intractable problems, probabilistic programming has over the past decade gained rapidly increasing attention in machine learning. There, probabilistic programs serve as *intuitive algorithmic descriptions* of complicated probability distributions. As Gordon et al. [2014] put it:

> *"The goal of probabilistic programming is to enable probabilistic modeling* [...] *to be accessible to the working programmer, who has sufficient domain expertise, but perhaps not enough expertise in probability theory* [...]*."*

In this paper, we consider more general weights than probabilities — in fact: more general than numbers. We should stress that *we are not the first* to consider weighted programs (see e.g. [Aguirre and Katsumata 2020; Brunel et al. 2014; Gaboardi et al. 2021] and see Section 7 for detailed comparisons).

Our goal was, however, *not* to merely go from probabilistic to weighted programming, just for the sake of generalization. Instead, we advocate *weighted programming* as a *programming paradigm for specifying mathematical models*. In particular, our prime goal is to take a step towards making mathematical modeling more accessible to people with a programming background. In a nutshell:

> **Render mathematical modeling accessible to the working programmer**,
> who has sufficient domain expertise, but perhaps not enough expertise
> in the respective mathematical theory.

Towards that goal, let us have a look at how such modeling could work in practice.

Weighted Programming as a Paradigm for Specifying Mathematical Models

As a motivating example, we consider the classical *Ski Rental Problem* [Komm 2016], a classical *optimization problem*, studied also in the context of online algorithms and competitive analysis [Borodin and El-Yaniv 1998]. A precise *textual* description of the problem is as follows:

> ***The Scenario:*** *A person does not own a pair of skis but is going on a skiing trip for n days. At the beginning of each day, the person can chose between two options: Either* rent *a pair of skis, costing* 1€ *for that day; or* buy *a pair of skis, costing y€ (and then go skiing for all subsequent days free of charge).*
>
> ***The Question:*** *What is the optimal (i.e. minimal) amount of money that the person has to spend for a pair of skis for the entire length of the trip?*

Using weighted programming, we can *model the scenario of this optimization problem* in a quite natural, simple, and intuitive way by the weighted program opt on the right. Intuitively, this weighted program tests each day whether the vacation is already over (Line 1). If not, it does the following (Lines 2–5): First, it decrements the vacation length by 1 day (Line 2). It then models the two options that the person has for each day by a *nondeterministic branching* (Line 4). In the left branch, it realizes the first option: Paying 1€ (Line 3). In the right branch, it realizes the second option: Paying y€ and then setting the remaining vacation length to 0 (Line 5), because with respect to having to pay for a pair of skis (not with respect to the joy of skiing) the vacation has effectively ended.

The Scenario:

```
1:  while ( $n > 0$ ) {
2:      $n \coloneqq n - 1$ ⨾
        {
3:          $\odot\, 1$
4:      } $\oplus$ {
5:          $\odot\, y$ ⨾ $n \coloneqq 0$
        }
    }
```

The Question:

$$\mathrm{wp}\,[\![\mathrm{opt}]\!]\,(\mathbf{1}) \;=\; ?$$

If we now want to answer *the question* of the optimization problem, we first chose a suitable *semiring*. In this setting of optimizing (i.e. minimizing) incurred cost, the *tropical* semiring $\mathcal{T} = (\mathbb{N}^{+\infty},\, \min,\, +,\, \infty,\, 0)$ comes to mind. The carrier set of this semiring are the extended natural numbers. The addition ($\oplus$) in this semiring is taking the minimum of two numbers. In the program, this is reflected by the fact that in Line 4 we would like to make whatever choice is cheaper for us. The multiplication ($\odot$) is the standard *addition* of numbers. In the program, this is reflected, for example, in Line 3, where we add a 1 to the current execution trace.

For actually *answering the question* of the optimization problem, we determine a weakest precondition of sorts, but interpreted here in a more general "quantitative" setting, with respect to post"condition" $\mathbf{1}$ — the multiplicative identity of the semiring; in this case, $\mathbf{1}$ is the natural number 0. Intuitively, this will for each path multiply together the weights along the path (recall: semiring multiplication is natural number addition). Then, we sum over the weights of all paths (recall: semiring summation is natural number minimization), thus yielding the accumulated costs along the least expensive path. As a result, our weakest-precondition-style calculus will yield

$$\mathsf{wp}\ [\![\mathtt{opt}]\!]\ (\mathbf{1}) \;=\; n \oplus y\,,$$

i.e. the minimum of the numbers n and y. This is precisely the solution to our optimization problem: If the trip length n is larger than the cost of buying skis, we should buy skis which will cost us y€. If n is smaller than y, we should instead rent each day (at cost 1€/day) which will cost us n€.

Toward competitive analysis, we can now model the cost of a deterministic online algorithm `onl` that solves the ski rental problem and determine $\mathsf{wp}\ [\![\mathtt{onl}]\!]\ (\mathbf{1})$. Then, we can compute the ratio $\mathsf{wp}\ [\![\mathtt{onl}]\!]\ (\mathbf{1}) \,/\, \mathsf{wp}\ [\![\mathtt{opt}]\!]\ (\mathbf{1})$ to determine the *competitive ratio* of the online algorithm `onl`.

We stress that program `opt` from above is *not* strictly speaking executable. For that, one would need some sort of scheduler who determinizes the nondeterministic choices. It is also not immediately clear what weighting the individual execution traces on a physical computer would mean. Instead, the above weighted program *encodes a mathematical model*, namely an optimization problem, by means of an *algorithmic representation* — much in the spirit of a probabilistic program that is also not necessarily meant to be executed but instead models a probability distribution.

Lastly, we would like to note that determining weakest preconditions is related to *inference* in probabilistic programming [Gordon et al. 2014], where one is concerned with, e.g., determining the probability that the probabilistic program establishes some postcondition.

Contributions. Our main technical contribution is the — to the best of our knowledge — first weakest precondition-style reasoning framework for weighted programs, which conservatively extends both Dijkstra's classical weakest preconditions and weakest *liberal* weakest preconditions. Our weakest pre calculi capture the semantics of *unbounded and potentially nonterminating loops* effortlessly, while other works explicitly avoid partiality (see Section 7). Our weakest liberal preweightings even give a nuanced semantics to nonterminating runs in order to reason about such traces as well. We demonstrate the applicability of our framework by several examples.

To achieve a high degree of generality and applicability, our framework is parameterized by a monoid of *weights* for weighting computations of programs and so-called *weightings* that take over the role of "quantitative assertions". We prove well-definedness and healthiness conditions of our calculi, provide formal connections to an operational semantics, and develop easy-to-apply invariant-based reasoning techniques.

Outline. Section 2 provides preliminaries on monoids and semirings. Section 3 introduces the syntax and operational semantics of weighted programs. We introduce our weakest (liberal) preweighting calculi for reasoning about weighted programs in Section 4. Invariant-style reasoning for loops is presented in Section 5. In Section 6, we demonstrate the efficacy of our framework by means of several examples. In Section 7, we give an overview of and a comparison to other works that study weighted computations. We conclude in Section 8. Proofs and supplementary material can be found in Batz et al. [2022].

2 MONOIDS AND SEMIRINGS

The weights occurring in our programs are elements from a monoid. Intuitively, this is because we would like to "multiply" the weights on a program's computation trace together in order to obtain the total weight of that trace. In particular, this multiplication should be associative and allow for neutral, i.e. effectless weighting. In Section 4.2 further below, we introduce *monoid modules* that are another important ingredient for our theory.

Definition 2.1 (Monoids). A *monoid* $\mathcal{W} = (W, \odot, \mathbf{1})$ consists of a *carrier set* W, an *operation* $\odot\colon W \times W \longrightarrow W$, and an *identity* $\mathbf{1} \in W$, such that for all $a, b, c \in W$,

(1) the operation $\odot$ is associative, i.e. $a \odot (b \odot c) \;=\; (a \odot b) \odot c$, and
(2) $\mathbf{1}$ is an identity with respect to $\odot$, i.e. $a \odot \mathbf{1} \;=\; \mathbf{1} \odot a \;=\; a$.

The monoid $\mathcal{W}$ is called *commutative* if moreoever $a \odot b = b \odot a$ holds. △

Important examples of monoids are the *words monoid* $(\Gamma^*, \;\cdot\;, \epsilon)$ over alphabet $\Gamma \neq \emptyset$ and the *probability monoid* $([0,1], \;\cdot\;, 1)$ (the latter is commutative). Another algebraic structure that plays a key role in this paper are *semirings*. Even though they are not strictly required for our theory, they render the application of our framework easier and more intuitive. This is because every semiring is a monoid module over itself, which we explain in more detail in Section 4.2. Our definition of semirings is stated below; for an in-depth introduction, we refer to [Droste et al. 2009, Ch. 1, 2]. As usual, multiplication $\odot$ binds stronger than addition $\oplus$ and we omit parentheses accordingly.

Definition 2.2 (Semirings). A *semiring* $\mathcal{S} = (S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ consists of a *carrier set* S, an *addition* $\oplus\colon S \times S \longrightarrow S$, a *multiplication* $\odot\colon S \times S \longrightarrow S$, a *zero* $\mathbf{0} \in S$, and a *one* $\mathbf{1} \in S$, such that

(1) $(S, \oplus, \mathbf{0})$ forms a *commutative monoid*;
(2) $(S, \odot, \mathbf{1})$ forms a (possibly non-commutative) *monoid*;
(3) multiplication *distributes* over addition, i.e. for all $a, b, c \in S$,

$$a \;\odot\; (b \oplus c) \;=\; a \odot b \;\oplus\; a \odot c \quad \text{and} \quad (a \oplus b) \;\odot\; c \;=\; a \odot c \;\oplus\; b \odot c\,; \qquad \text{and}$$

(4) multiplication by zero *annihilates* S, i.e. $\mathbf{0} \odot a \;=\; a \odot \mathbf{0} \;=\; \mathbf{0}\,.$ △

Our "cheat sheet" in Table 1 lists various example semirings along with possible applications in a weighted programming context. Further well-known semirings not considered specifically in this paper include (i) the *Łukasiewicz* semiring [Gerla 2003; Nola and Gerla 2005] motivated by multivalued logics and related to tropical geometry [Gavalec et al. 2015], (ii) the *resolution* semiring [Bagnol 2014] from proof theory, (iii) the *categorial* and *lexicographic* semirings [Sproat et al. 2014] used in natural language processing, (iv) the *thermodynamic* semirings [Marcolli and Thorngren 2011, 2014] employed in information theory, and (v) the *confidence-probability* semiring [Wirsching et al. 2010]. We leave the study of applications of weighted programs over these semirings for future work. Finally, we mention that more complicated semirings can be created from existing ones through algebraic constructions like matrices, tensors, polynomials, or formal power series.

3 WEIGHTED PROGRAMS

For a monoid $\mathcal{W} = (W, \odot, \mathbf{1})$ of weights, we study the $\mathcal{W}$-weighted guarded command language $\mathcal{W}$-wGCL featuring — in addition to standard control-flow instructions — *branching* and *weighting*. If the monoid $\mathcal{W}$ is evident from the context, we omit the symbol and write just wGCL.

[1]Also known as *max-min-semiring*. The name *Bottleneck semiring* is taken from [Pouly 2010]. Quantitative verification using this semiring is further studied in [Zhang and Kaminski 2022a,b].

Table 1. Weighted programming cheat sheet.

Optimization via the **Tropical semiring** $(\mathbb{N}^{+\infty}, \min, +, +\infty, 0)$

Weighting $\odot$ a: Accumulate cost a	**Branching $\oplus$:** Choose branch that will accumulate *minimal* cost	
Postweighting f: Cost that is accumulated after program termination; typically choose $f = 0$	**wp $[\![C]\!]$ (0):** Minimal accumulated cost amongst all terminating executions of C	**wlp $[\![C]\!]$ (f):** Minimum of wp $[\![C]\!]$ (f) and the minimal accumulated cost amongst all nonterminating executions of C

Optimization via the **Arctic semiring** $(\mathbb{N}^{+\infty}_{-\infty}, \max, +, -\infty, 0)$

Weighting $\odot$ a: Accumulate cost a	**Branching $\oplus$:** Choose branch that will accumulate *maximal* cost	
Postweighting f: Cost that is accumulated after program termination; typically choose $f = 0$	**wp $[\![C]\!]$ (0):** Maximal accumulated cost amongst all terminating executions of C	**wlp $[\![C]\!]$ (f):** Same as wp $[\![C]\!]$ (f) if all executions of C terminate, else $+\infty$

Optimization via the **Bottleneck semiring**[1] $(\mathbb{R}^{+\infty}_{-\infty}, \max, \min, -\infty, +\infty)$

Weighting $\odot$ a: Restrict capacity of current branch to a	**Branching $\oplus$:** Choose branch with accumulate *maximal* capacity	
Postweighting f: Upper bound after program termination; typically choose $f = +\infty$	**wp $[\![C]\!]$ (f):** Maximum bottleneck amongst all *terminating* executions of C	**wlp $[\![C]\!]$ (f):** Maximum bottleneck amongst all executions of C

Model Checking via the **Formal languages semiring** $(2^{\Gamma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$

Weighting $\odot$ a: Append symbol a to current trace	**Branching $\oplus$:** Account for/aggregate behavior of both branches	
Postweighting f: Language that is appended to each terminated trace; typically choose $f = \{\epsilon\}$	**wp $[\![C]\!]$ $(\{\varepsilon\})$:** Language of all terminating traces of C	**wlp $[\![C]\!]$ $(\{\varepsilon\})$:** Language of all terminating and nonterminating traces of C

Combinatorics via the (extended) **Natural Numbers semiring** $(\mathbb{N}^{+\infty}, +, \cdot, 0, 1)$

Weighting $\odot$ a: Make a copies of current path/trace	**Branching $\oplus$:** Sum up number of paths/traces of both branches	
Postweighting f: Number of copies that is made of each terminated path/trace; typically choose $f = 1$	**wp $[\![C]\!]$ (1):** Number of all terminating paths/traces of C	**wlp $[\![C]\!]$ (f):** Same as wp $[\![C]\!]$ (f) if all executions of C terminate, else $+\infty$

Hidden Markov Models via the **Viterbi semiring** $([0,1], \max, \cdot, 0, 1)$

Weighting $\odot$ a: Let what follows happen with probability a	**Branching $\oplus$:** Choose branch of maximal probability	
Postweighting f: Probability additionally multiplied to each terminated trace; typically choose $f = [\varphi]$, i.e. the indicator function of some event φ	**wp $[\![C]\!]$ $([\varphi])$:** Maximal probability of a terminating execution establishing φ	**wlp $[\![C]\!]$ $([\varphi])$:** Maximal probability of a non-terminating execution, or a terminating execution establishing φ

Verification/Debugging via the **Boolean semiring** $(\{0,1\}, \vee, \wedge, 0, 1)$

Weighting $\odot$ a: Assert predicate a	**Branching $\oplus$:** Angelic choice: choose "most true" branch	
Postweighting f: Postcondition (a predicate) that should be established after program termination	**wp $[\![C]\!]$ (f):** Weakest precondition of f, the weakest predicate g so that starting in g, program C *can* terminate in state $\tau \models f$	**wlp $[\![C]\!]$ (f):** Weakest liberal precondition of f, the weakest predicate g so that starting in g, program C *can* either diverge or terminate in state $\tau \models f$

Feature Selection via the **Why semiring** (propositional *positive* DNF, $\vee, \wedge, 0, 1$) over a finite set of variables $X_1, \ldots, X_n$ (cf. [Dannert et al. 2019])

Weighting $\odot$ a: Use resource a	**Branching $\oplus$:** Alternatives: use resources via C_1 or via C_2	
Postweighting f: Combinations of resources used after termination; typically choose $f = [\varphi]$	**wp $[\![C]\!]$ $([\varphi])$:** Possible alternatives (disjunction) of resource sets (conjunction) to reach event φ	**wlp $[\![C]\!]$ $([\varphi])$:** Possible alternatives (disjunction) of resource sets (conjunction) to either not terminate or reach event φ

3.1 Syntax

wGCL programs C adhere to the grammar

$$\begin{array}{llllr} C & \longrightarrow & \{\, C_1 \,\} \oplus \{\, C_2 \,\} & \mid \; \odot\, a & \text{(branching | weighting)} \\ & \mid & x := E & \mid \; \texttt{if}\;(\,\varphi\,)\;\{\, C_1 \,\}\;\texttt{else}\;\{\, C_2 \,\} & \text{(assignment | conditional choice)} \\ & \mid & C_1 \,⨟\, C_2 & \mid \; \texttt{while}\,(\,\varphi\,)\,\{\, C_1 \,\} & \text{(sequential composition | loop)} \\ & \mid & \texttt{skip} \;\equiv\; \odot\, \mathbb{1} & \mid \; \{\, C_1 \,\}\; {}_{a}\!\oplus_{b}\; \{\, C_2 \,\} \;\equiv\; \{\, \odot\, a \,⨟\, C_1 \,\} \oplus \{\, \odot\, b \,⨟\, C_2 \,\} & \text{(syntactic sugar)} \end{array}$$

where x is a program variable from a countable set Vars, E is an *arithmetic expression* over Vars, φ is a Boolean expression (also called *guard*), and a is a *weight* from the monoid's carrier W.

Our programs feature *branching* "$\{\, C_1 \,\} \oplus \{\, C_2 \,\}$" and *weighting* "$\odot\, a$" of the current computation path where $a \in W$ represents some weight. For example, we can express a probabilistic choice "execute C_1 with probability $1/3$ and C_2 otherwise" as $\{\, C_1 \,\}\; {}_{1/3}\!\oplus_{2/3}\; \{\, C_2 \,\}$ over the monoid $([0,1],\ \cdot,\ 1)$. We allow for syntactic sugar in weightings, e.g. $\odot\, a^x$ for some $a \in W$, $x \in \text{Vars}$.[2] The symbol $\odot$ used in the weight-statement is reminiscent of the corresponding monoid operation in $\mathcal{W}$. The symbol $\oplus$ we use for the branching-statement will become evident in Section 4.

3.2 Program States

A program state σ maps each variable in Vars to its value in $\mathbb{N}$. To ensure that the set of program states is countable,[3] we restrict to states in which at most finitely many variables have a non-zero value. Intuitively, those that appear in a given program are possibly assigned a non-zero value. Formally, the set Σ of program states is given by

$$\Sigma \quad := \quad \{\, \sigma\colon \mathsf{Vars} \to \mathbb{N} \;\mid\; \{\, x \in \mathsf{Vars} \mid \sigma(x) \neq 0 \,\} \;\text{ is finite} \,\}\,.$$

We overload notation and denote by $\sigma(\xi)$ the evaluation of the (arithmetic, Boolean, or weight) expression ξ in σ, i.e. the value obtained from evaluating ξ after replacing every variable x in ξ by $\sigma(x)$. We denote by $\sigma\,[x \mapsto v]$ the *update* of variable x by value v in state σ. Formally:

$$\sigma\,[x \mapsto v] \quad := \quad \lambda\, y.\, \begin{cases} v & \text{if } y = x, \\ \sigma(y) & \text{otherwise.} \end{cases}$$

3.3 Operational Semantics

To formalize our notion of *weighted computation paths*, we define small-step operational semantics [Plotkin 2004] in terms of a weighted computation graph, or rather a weighted *computation forest*.[4] Apart from our special weight operation $\odot\, a$, the operational semantics is standard but we include the details for the sake of completeness. Intuitively, the computation forest of $\mathcal{W}$-wGCL contains one tree $T_{C,\sigma}$ per program C and initial state σ representing the computation of C on initial state σ.

Definition 3.1 (Computation Forest of $\mathcal{W}$-wGCL). For the monoid $\mathcal{W} = (W,\ \odot,\ \mathbb{1})$, the *computation forest* of $\mathcal{W}$-wGCL is the (countably infinite) directed weighted graph $\mathcal{G} = (Q,\ \Delta)$, where

- $Q = (\text{wGCL} \cup \{\, \downarrow \,\}) \times \Sigma \times \mathbb{N} \times \{\, L, R \,\}^*$ is the set of vertices (called *configurations*);
- $\Delta \subseteq Q \times W \times Q$ is the set of directed weighted edges (called *transitions*) which is defined as the smallest set satisfying the SOS-rules in Fig. 1. △

[2]The corresponding program $i := x \,⨟\, \texttt{while}\,(\, i > 0 \,)\,\{\, \odot\, a \,⨟\, i := i - 1 \,\}$ requires introducing a fresh loop-variable $i \in \text{Vars}$.
[3]We restrict Σ a priori to avoid technical issues; even wGCL programs over uncountable Σ reach just countably many states.
[4]A path-based semantics in terms of trees is convenient for technical reasons; it allows to distinguish programs like `skip` from $\{\, \texttt{skip} \,\} \oplus \{\, \texttt{skip} \,\}$. The latter has two terminating computation paths with weight $\mathbb{1}$ and the former has only one.

$$\frac{\sigma' = \sigma\,[x \mapsto [\![E]\!](\sigma)\,]}{\langle x \coloneqq E,\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle {\downarrow},\ \sigma',\ n{+}1,\ \beta\rangle}\ \text{(assign)} \qquad \frac{}{\langle \odot\, a,\ \sigma,\ n,\ \beta\rangle \vdash_{a} \langle {\downarrow},\ \sigma,\ n{+}1,\ \beta\rangle}\ \text{(weight)}$$

$$\frac{\langle C_1,\ \sigma,\ n,\ \beta\rangle \vdash_{a} \langle {\downarrow},\ \sigma',\ n{+}1,\ \beta\rangle}{\langle C_1 \fatsemi C_2,\ \sigma,\ n,\ \beta\rangle \vdash_{a} \langle C_2,\ \sigma',\ n{+}1,\ \beta\rangle}\ \text{(seq. 1)} \qquad \frac{\langle C_1,\ \sigma,\ n,\ \beta\rangle \vdash_{a} \langle C_1',\ \sigma',\ n{+}1,\ \beta\rangle \quad \text{and} \quad C_1' \neq {\downarrow}}{\langle C_1 \fatsemi C_2,\ \sigma,\ n,\ \beta\rangle \vdash_{a} \langle C_1' \fatsemi C_2,\ \sigma',\ n{+}1,\ \beta\rangle}\ \text{(seq. 2)}$$

$$\frac{\sigma \models \varphi}{\langle \mathtt{if}\ (\ \varphi\ )\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle C_1,\ \sigma,\ n{+}1,\ \beta\rangle}\ \text{(if)} \qquad \frac{\sigma \models \neg\varphi}{\langle \mathtt{if}\ (\ \varphi\ )\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle C_2,\ \sigma,\ n{+}1,\ \beta\rangle}\ \text{(else)}$$

$$\frac{}{\langle \{\, C_1 \,\} \oplus \{\, C_2 \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle C_1,\ \sigma,\ n{+}1,\ \beta L\rangle}\ \text{(l. branch)} \qquad \frac{}{\langle \{\, C_1 \,\} \oplus \{\, C_2 \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle C_2,\ \sigma,\ n{+}1,\ \beta R\rangle}\ \text{(r. branch)}$$

$$\frac{\sigma \models \varphi}{\langle \mathtt{while}\,(\ \varphi\ )\,\{\, C \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle C \fatsemi \mathtt{while}\,(\ \varphi\ )\,\{\, C \,\},\ \sigma,\ n{+}1,\ \beta\rangle}\ \text{(while)} \qquad \frac{\sigma \models \neg\varphi}{\langle \mathtt{while}\,(\ \varphi\ )\,\{\, C \,\},\ \sigma,\ n,\ \beta\rangle \vdash_{\mathbb{1}} \langle {\downarrow},\ \sigma,\ n{+}1,\ \beta\rangle}\ \text{(break)}$$

Fig. 1. Structural operational semantics of wGCL-programs.

We use the notation $\kappa_1 \vdash_a \kappa_2$ instead of $(\kappa_1, a, \kappa_2) \in \Delta$. Intuitively, for a configuration $\langle C,\ \sigma,\ n,\ \beta\rangle$, the component C represents the program that still needs to be executed (thus playing the role of a "program counter") and $C = {\downarrow}$ indicates termination; σ is the current program state (variable valuation); n is the number of computation steps that have been executed so far and β is the history of left and right branches that have been taken. Remembering the number of computation steps and the history of left and right branches ensures that $\mathcal{G}$ is indeed a forest. Moreover, it is easy to check that $\mathcal{G}$ has no multi-edges, i.e. there is at most one weighted edge between any two configurations.

Note that $\mathcal{G}$ is finitely branching and that the only rules that alter the branching history β are (l. branch) and (r. branch). In particular, (if) and (else) do not change β because they are not *truly* branching: Indeed, all configurations of the form $\langle \mathtt{if}\ (\ \varphi\ )\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\},\ \sigma,\ n{+}1,\ \beta\rangle$ have a *unique* successor configuration which depends on whether or not σ satisfies φ.

An *initial* configuration is of the form $\kappa = \langle C,\ \sigma,\ 0,\ \epsilon\rangle$ for arbitrary C and σ. For initial configurations we also write $\langle C,\ \sigma\rangle$ instead of $\langle C,\ \sigma,\ 0,\ \epsilon\rangle$. By definition of Δ, initial configurations have no incoming transitions; they are thus the roots of the trees in the computation forest. Similarly, a configuration $\langle {\downarrow},\ \sigma,\ n,\ \beta\rangle$ is called *final*. Final configurations are the leaves of the forest.

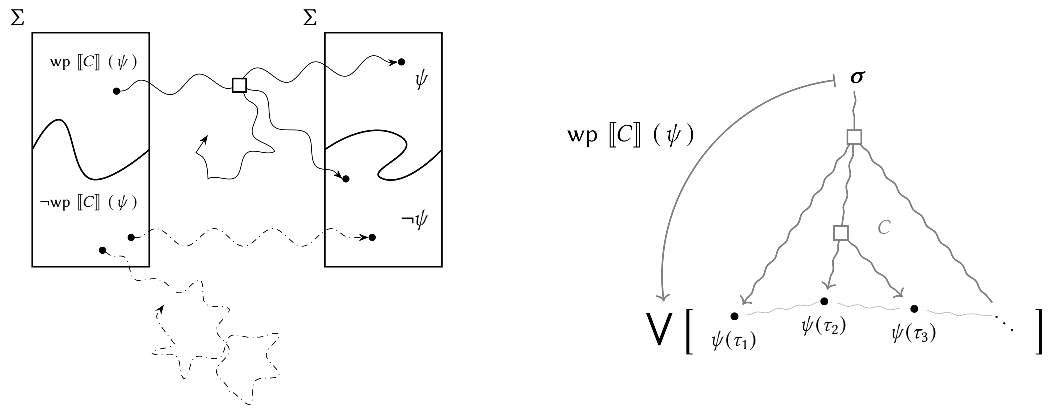
We can now define computation paths. For $\kappa \in Q$, let $\mathsf{succ}(\kappa) \coloneqq \{\, \kappa' \in Q \mid \exists a \in W \colon \kappa \vdash_a \kappa' \,\}$ be the set of all possible successor configurations. A *computation path* of length $n \in \mathbb{N}$ is a finite path $\pi = \kappa_0\kappa_1 \ldots \kappa_n$ in $\mathcal{G}$, i.e. $\kappa_{i+1} \in \mathsf{succ}(\kappa_i)$ for all $i = 0, \ldots, n-1$, such that κ_0 is initial. The set of all such paths is denoted $\mathsf{Paths}^n_{\kappa_0}$. Since $\mathcal{G}$ has no multi-edges, for two configurations κ and $\kappa' \in \mathsf{succ}(\kappa)$ we denote the unique weight $a \in W$ such that $\kappa \vdash_a \kappa'$ by $\mathsf{wgt}(\kappa\,\kappa')$. The *weight* of a computation path $\kappa_0\kappa_1 \ldots \kappa_n$ is then defined as

$$\mathsf{wgt}(\kappa_0\kappa_1 \ldots \kappa_n) \ \coloneqq\ \bigodot_{i=0}^{n-1} \mathsf{wgt}(\kappa_i\,\kappa_{i+1}) \ =\ \mathsf{wgt}(\kappa_0\,\kappa_1) \odot \mathsf{wgt}(\kappa_1\,\kappa_2) \odot \cdots \odot \mathsf{wgt}(\kappa_{n-1}\,\kappa_n)\ .$$

Recall that our monoids are not commutative in general, and thus the order of the above product matters. The *last state* of computation path $\pi = \kappa_0\kappa_1 \ldots \kappa_n$ is the program state at configuration κ_n and is denoted $\mathsf{last}(\pi) \in \Sigma$. The computation path π is called *terminal* if κ_n is final. Given an initial configuration κ_0, we define the set of *terminating computation paths* starting in κ_0 as

$$\mathsf{TPaths}_{\kappa_0} \quad = \quad \bigcup_{n \in \mathbb{N}} \left\{\, \pi \in \mathsf{Paths}^n_{\kappa_0} \;\middle|\; \pi \text{ is terminal} \,\right\}\ .$$

Note that $\mathsf{TPaths}_{\kappa_0}$ is a countable set for each κ_0. An *infinite* computation path is an infinite sequence $\kappa_0\kappa_1 \ldots$ such that $\kappa_0 \ldots \kappa_i$ is a computation path for all $i \geq 0$.



(a) **The set perspective:** Starting in wp $[\![C]\!]\ (\psi)$, C can terminate in ψ. Starting in $\neg$wp $[\![C]\!]\ (\psi)$, C either diverges or terminates in $\neg\psi$, but it cannot terminate in ψ.

(b) **The map perspective:** Given initial state σ, wp $[\![C]\!]\ (\psi)$ determines all final states τ_i reachable from executing C on σ, evaluates ψ in each τ_i, and returns the disjunction over these truth values.

Fig. 2. Two perspectives on the angelic weakest precondition of program C with respect to postcondition ψ.

4 WEIGHTING TRANSFORMER SEMANTICS

Throughout this section, we develop a weakest-precondition-style calculus à la Dijkstra [1975] for reasoning about weighted programs on source code level. We start with a recap on Dijkstra's weakest preconditions. We then gradually lift weakest preconditions to weakest pre*weightings*.

4.1 Weakest Preconditions

Dijkstra's weakest precondition calculus is based on *predicate transformers*

$$\mathrm{wp}\,[\![C]\!]\colon\quad \mathbb{B} \longrightarrow \mathbb{B}\,, \qquad \text{where} \quad \mathbb{B} = \{0,\,1\}^{\Sigma} \text{ is the set of } \textit{predicates} \text{ over } \Sigma,$$

which associate to each nondeterministic program C a mapping from predicates to predicates. Somewhat less common, we consider here an *angelic* setting, where the nondeterminism is resolved to our advantage. Specifically, the *angelic* weakest precondition transformer wp $[\![C]\!]$ maps a *post*-condition ψ over final states to a *pre*condition wp $[\![C]\!]\ (\psi)$ over initial states, such that executing the program C on an initial state satisfying wp $[\![C]\!]\ (\psi)$ guarantees that C *can*[5] terminate in a final state satisfying ψ, see also Figure 2a. More symbolically, if $[\![C]\!]_\sigma$ denotes the set of all final states reachable from executing C on σ, then

$$\sigma \models \mathrm{wp}\,[\![C]\!]\,(\psi) \qquad \text{implies} \qquad \exists\, \tau \in [\![C]\!]_\sigma\colon \quad \tau \models \psi\,.$$

While the above is a *set perspective* on wp, a different, but equivalent, perspective on wp is the *map perspective*, see Figure 2b. From this perspective, the postcondition is a function $\psi\colon \Sigma \longrightarrow \{0,\,1\}$ mapping program states to truth values. The predicate wp $[\![C]\!]\ (\psi)$ is then a function that takes as input an initial state σ, determines for each reachable final state $\tau \in [\![C]\!]_\sigma$ the (truth) value $\psi(\tau)$, and finally returns the disjunction over all these truth values. More symbolically,

$$\mathrm{wp}\,[\![C]\!]\,(\psi)\,(\sigma) \qquad = \qquad \bigvee_{\tau \in [\![C]\!]_\sigma} \psi(\tau)\,.$$

[5]Recall that C is a *nondeterministic* program.

It is this map perspective which we will now gradually lift to a weighted setting. For that, we first need to leave the realm of Boolean values in which the predicates live. Instead of acting on Boolean-valued predicates, our calculus will instead act on more general objects called *weightings*.

4.2 Weightings and Modules

For probabilistic programs, Kozen [1985] and later Morgan et al. [1996] have generalized predicates to real-valued functions $f\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$ (called *expectations* [McIver and Morgan 2005]) associating a quantity to every program state. With *weightings*, we generalize further by associating a more general "quantity" to every program state. Our wp-style calculus acts on these weightings instead of Boolean-valued predicates. Weightings form — just like first-order logic for weakest preconditions — the *assertion "language" of weakest preweighting reasoning*.

Let us fix a monoid $\mathcal{W}$ of weights. As with predicates and expectations, we need notions of addition *and* multiplication operations for our weightings. The monoid $\mathcal{W}$ constituting our programs' weights, however, only provides a multiplication $\odot$. We hence require that our weightings form a *$\mathcal{W}$-module $\mathcal{M}$* which *does* provide an addition. This is inspired by the probabilistic setting where the program weights — the probabilities — are taken from the interval $[0,1]$[6]. Expectations, however, map program states to arbitrary extended reals in $\mathbb{R}_{\geq 0}^{\infty}$ to reason about, e.g. expected values of program variables. Another advantage of distinguishing between $\mathcal{W}$ and $\mathcal{M}$ in general is explained in Example 4.4 further below.

We now define modules formally. (Monoid)-modules are similar to vector spaces over fields in that they also have a well-behaved *scalar multiplication*.

Definition 4.1 (Monoid-Modules). Let $\mathcal{W} = (W,\, \odot,\, \mathbb{1})$ be a monoid. A *(left) $\mathcal{W}$-module* $\mathcal{M} = (M,\, \oplus,\, \mathbb{0},\, \otimes)$ is a *commutative monoid* $(M,\, \oplus,\, \mathbb{0})$ equipped with a (left) action called *scalar multiplication* $\otimes\colon W \times M \to M$, such that for all monoid elements $v, w \in W$ and module elements $a, b \in M$,

(1) the scalar multiplication $\otimes$ is *associative*, i.e. $(v \odot w) \otimes a \;=\; v \otimes (w \otimes a)$,
(2) the scalar multiplication $\otimes$ is *distributive*, i.e. $v \otimes (a \oplus b) \;=\; (v \otimes a) \oplus (v \otimes b)$,
(3) $\mathbb{1}$ is *neutral* w.r.t. $\otimes$ and $\mathbb{0}$ *annihilates*, i.e. $\mathbb{1} \otimes a = a$ and $v \otimes \mathbb{0} = \mathbb{0}$. △

We are now in a position to define *weightings*:

Definition 4.2 (Weightings). Given a $\mathcal{W}$-module $\mathcal{M}$, a function $f\colon \Sigma \to M$ associating a weight from $\mathcal{M}$ to each program state is called *weighting*. We denote the set of all weightings by $\mathbb{W}$. Elements of $\mathbb{W}$ are denoted by $f, g, h, \ldots$ and variations thereof. △

The structure $(\mathbb{W},\, \oplus,\, \mathbb{0},\, \otimes)$, where $\oplus$, $\mathbb{0}$, and $\otimes$ are lifted pointwise, also forms a $\mathcal{W}$-module. We refer to $\mathbb{W}$ as the *module of weightings* over $\mathcal{M}$. We emphasize that all the results developed in this paper apply to the important — and simpler — special case where the monoid and the module together form a *semiring*: The multiplication $\odot$ of a semiring $\mathcal{S} = (S,\, \oplus,\, \odot,\, \mathbb{0},\, \mathbb{1})$ is then the left-action $\otimes$ of the multiplicative monoid of $(S,\, \odot,\, \mathbb{1})$ to the additive monoid $(S,\, \oplus,\, \mathbb{0})$. For this reason, we write $\odot$ instead of $\otimes$ (as both are associative and it should be clear from the rightmost multiplicant's type) and adopt the following convention:

Convention. *In all examples in this paper, unless stated otherwise, both the monoid $\mathcal{W}$ and the $\mathcal{W}$-module $\mathcal{M}$ are given in terms of a semiring $\mathcal{S}$ that will be clear from the context.*

[6]Note that probabilities form a monoid under multiplication.

Towards our goal of defining a weakest-precondition-style calculus for weighted programs, we restrict to *naturally ordered*[7] ω*-bicontinuous* modules:

Definition 4.3 (Natural Order). Given a module $\mathcal{M}$, the binary relation $\preceq\, \subseteq M \times M$ given by

$$a \preceq b \qquad \text{iff} \qquad \exists\, c \in M\colon \quad a \oplus c = b$$

is called the *natural order* on $\mathcal{M}$. If $\preceq$ is a partial order, we call $\mathcal{M}$ *naturally ordered.* △

The unique *least element* of a naturally ordered module is $\mathbb{0}$. We say that $\mathcal{M}$ is ω*-bicontinuous* if (1) both the natural order $\preceq$ and the reversed natural order $\succeq$ are (pointed[8]) ω-cpos [Abramsky 1994, Sec. 2.2.4], and (2) the operations $\oplus$ and $\odot$ are ω-continuous [Abramsky 1994, Sec. 2.2.4] functions (w.r.t. both $\preceq$ and $\succeq$). In particular, for $\mathcal{M}$ to be ω-bicontinuous, we require the natural order $\preceq$ to possess a *greatest* element $\top$. The definition of naturally ordered ω-bicontinuous *semirings* is completely analogous. All these properties translate to the aforementioned module of weightings: $\mathbb{W}$ is naturally ordered[9] if the underlying module $\mathcal{M}$ is naturally ordered and joins/meets can be defined pointwise. For example, the Boolean semiring $\mathcal{B}$ and the tropical semiring $\mathcal{T}$ described in Section 2 are ω-continuous. Please confer [Batz et al. 2022, App. A] for details on the above terms.

Example 4.4 (Modules). Let Γ be a non-empty alphabet. The structure $\mathcal{L}_\Gamma^\infty := (2^{\Gamma^\infty}, \cup, \emptyset, \cdot)$ forms a module over the word monoid Γ^*. Here, $\Gamma^\infty := \Gamma^* \cup \Gamma^\omega$ is the set of all finite and ω-words over Γ, and the subsets of Γ^∞ are subsequently called ω*-potent* languages over Γ. Our interest in $\mathcal{L}_\Gamma^\infty$ stems from the fact that we want to study infinite program runs as in Section 4.4. We stress that this cannot be achieved by simply defining a *semiring* on 2^{Γ^∞}. In fact, even though such a semiring can be defined, its multiplication would *not be* ω*-cocontinuous* (a counterexample is given in [Batz et al. 2022, App. B.3]). On the other hand, $\mathcal{L}_\Gamma^\infty$ *does* form an ω-*bi*continuous module (cf. [Batz et al. 2022, App. B.4]). △

4.3 Weakest Preweightings

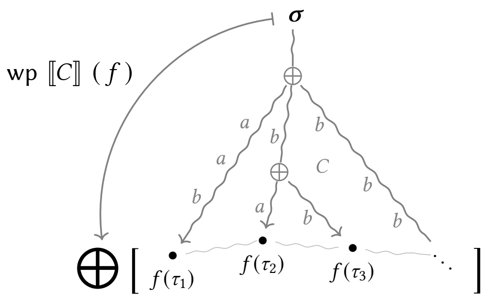
We now define a calculus for formal reasoning about weighted programs à la Dijkstra. In reference to Dijkstra's *weakest precondition calculus* and McIver & Morgan's *weakest preexpectation calculus*, we name our verification system *weakest preweighting calculus.*

First, we notice that predicates just form a specific semiring, namely $(\mathbb{B}, \vee, \wedge, 0, 1)$ and thus they are in particular *modules* over their underlying "Boolean monoid" $(\{0, 1\}, \wedge, 1)$. We refer to this as the *module of predicates.* With that in mind, we can now generalize the *map perspective* of weakest preconditions to weakest preweightings, see Figure 3a. Instead of a postcondition, we now have a post*weighting* $f\colon \Sigma \to M$ mapping program states to elements from our $\mathcal{W}$-module $\mathcal{M}$. The weakest preweighting $\mathsf{wp}\, [\![C]\!]\, (f)$ is then a function that takes as input an initial state σ, determines the weight w of each path starting in σ and terminating in some final state τ, scalar-multiplies the path's weight w to the corresponding postweight $f(\tau)$ from the module $\mathcal{M}$, and finally returns the module sum over all these so-determined weights, see Figure 3a.

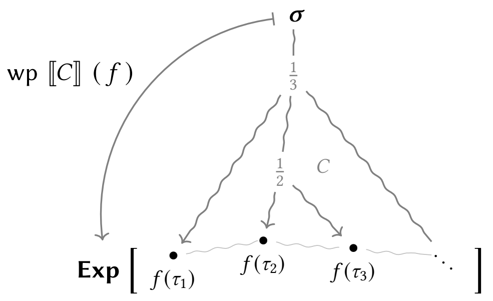
Figure 3b depicts how the general weighted setting is instantiated to a probabilistic setting: the postweightings become real-valued functions (expectations), the path weights become the paths' probabilities, and the summation remains a summation, thus obtaining an expected value.

One of the main advantages of Dijkstra's calculus is that the weakest preconditions can be defined by induction on the program structure, thus allowing for *compositional reasoning*. Indeed, the same

[7]More generally, partially ordered modules (where the partial order is compatible with the algebraic structure, e. g. addition and left-action are monotone) also work. However, the natural order is the least (w.r.t. $\subseteq$) such partial order. We employ the natural order for simplicity.
[8]We additionally require existence of a *least* element $\perp$.
[9]The natural order on $\mathbb{W}$ is the point-wise lifted natural order on $\mathcal{M}$.



(a) **Weakest preweightings:** Given initial state σ, wp $[\![C]\!]$ (f) (σ) determines the weight $w = a_1 a_2 \cdots$ of each path starting in σ and terminating in some final state τ_i, scalar-multiplies w to the corresponding postweight $f(\tau)$, and returns the module sum over all so-determined weights.



(b) **Weakest preexpectations:** Given initial state σ, wp $[\![C]\!]$ (f) (σ) determines the expected value (with respect to the probability distribution generated by executing C on σ) of f evaluated in the final states reached after executing C on σ.

Fig. 3. The meaning of weakest preweightings generally and weakest preexpectations specifically.

applies to our weighted setting. In the following, we fix an ambient monoid $\mathcal{W}$ of programs weights and an ω-bicontinuous $\mathcal{W}$-module $\mathcal{M}$ that constitutes the habitat of our weightings $\mathbb{W}$. We now go over each construct of wGCL and see how a weakest preweighting semantics can be developed and understood analogously to Dijkstra's weakest preconditions.

Assignment. The weakest precondition of an assignment is given by

$$\mathsf{wp}\ [\![x \coloneqq E]\!]\ (\psi) \ = \ \psi\,[x/E]\ ,$$

where $\psi\,[x/E]$ is the replacement of every occurrence of variable x in the postcondition ψ by the expression E. For weakest preweightings, we proceed analogously. That is, we "replace" every "occurrence" of x in f by E. Since f is actually not a syntactic object, we more formally define

$$\mathsf{wp}\ [\![x \coloneqq E]\!]\ (f) \ = \ f\,[x/E] \ \coloneqq \ \lambda\,\sigma.\, f\Big(\sigma\,[x \mapsto \sigma(E)]\Big)\ .$$

So the weighting f of the final state reached after executing the assignment $x \coloneqq E$ is precisely f evaluated at the state $\sigma\,[x \mapsto \sigma(E)]$ — the state obtained from σ by updating variable x to $\sigma(E)$.

Weighting. Consider the classical statement `assert` φ. Operationally, when executing `assert` φ on some initial state σ, we check whether σ satisfies the predicate φ. If $\sigma \models \varphi$, the execution trace "passes through" the assertion and potentially proceeds with whatever program comes after the assertion. If, however, $\sigma \not\models \varphi$, then the execution trace at hand is so-to-speak "annihilated". Intuitively, these two cases can be thought of as multiplying (or weighting) the execution trace either by a multiplicative identity *one* or by an annihilating *zero*, respectively.

Denotationally, the weakest precondition of `assert` φ is given by

$$\mathsf{wp}\ [\![\mathtt{assert}\,\varphi]\!]\ (\psi) \ = \ \varphi \wedge \psi\ .$$

Indeed, whenever an initial state σ satisfies the precondition $\varphi \wedge \psi$, then (a) executing `assert` φ will pass through asserting φ and moreover — since the assertion itself does not alter the current program state — (b) it terminates in state σ which also satisfies the postcondition ψ. Dually, if σ does not satisfy $\varphi \wedge \psi$, then either (a) executing `assert` φ does not pass through asserting φ or (b) it does pass through the assertion but σ does not satisfy the postcondition ψ.

When viewing the above through our monoid and module glasses, $\wedge$ is just the scalar-multiplication in the module of predicates. So in other words, $\mathsf{wp}\,[\![\texttt{assert}\ \varphi]\!]\,(\psi)$ weights (multiplies) ψ with φ. Therefore, we generalize from *conjunction with a predicate* to (scalar-)*multiplication with a monoid element* a from $\mathcal{W}$ and introduce the statement $\odot\, a$ into the programming language. Operationally, our execution traces are weighted and the $\odot\, a$ statement scalar-multiplies the current execution trace's weight by an a. Denotationally, the weakest preweighting of $\odot\, a$ is given by

$$\mathsf{wp}\,[\![\odot\, a]\!]\,(f) \;=\; a \odot f\,.$$

Remark 4.5 (On Non-commutativity and Notation). Recall that multiplication of weights is generally not commutative — think, for example, about the word monoid Γ^*. In the light of potential non-commutativity, the flipping of the sides, i.e. $\odot\, a$ in the program syntax *versus* $a \odot \underline{\hspace{1em}}$ in the denotational weighting transformer semantics, is on purpose: Programs are usually read (and executed) in a forward manner. Assuming that the weights along an execution trace are collected from left to right, from initial to final state, weighting by a is a *right-multiplication*, appending at the end of the current execution trace the weight a.

Weakest preweightings, on the other hand, are backward-moving: The f is a *post*weighting that potentially abstracts or summarizes the effects of subsequent computations. Whenever we encounter on our way from the back to the front of a program a weighting by a, we thus have to *pre*pend a to the current postweighting f, yielding a *left-multiplication* $a \odot f$ in the denotations. $\triangle$

***Branching*.** We now consider the classical angelic nondeterministic choice $\{\, C_1 \,\} \,\square\, \{\, C_2 \,\}$. Operationally, when "executing" this choice on some initial state σ, *either* the program C_1 *or* the program C_2 will be executed, chosen nondeterministically. Hence, the execution will reach either a final state in which executing C_1 on σ terminates or a final state in which executing C_2 on σ terminates (or no final state if both computations diverge).

Denotationally, the *angelic* weakest precondition of $\{\, C_1 \,\} \,\square\, \{\, C_2 \,\}$ is given by

$$\mathsf{wp}\,[\![\{\, C_1 \,\} \,\square\, \{\, C_2 \,\}]\!]\,(\psi) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(\psi) \;\vee\; \mathsf{wp}\,[\![C_2]\!]\,(\psi)\,.$$

Indeed, whenever an initial state σ satisfies the precondition $\mathsf{wp}\,[\![C_1]\!]\,(\psi) \vee \mathsf{wp}\,[\![C_2]\!]\,(\psi)$ then executing C_1 or executing C_2 will terminate in some final state satisfying the postcondition ψ.

Again viewed through our module glasses, $\vee$ is just the addition of the module of predicates. So in other words, $\mathsf{wp}\,[\![\{\, C_1 \,\} \,\square\, \{\, C_2 \,\}]\!]\,(\psi)$ unions (adds) $\mathsf{wp}\,[\![C_1]\!]\,(\psi)$ and $\mathsf{wp}\,[\![C_2]\!]\,(\psi)$. We thus generalize from *disjunction of two predicates* to *addition of two module elements* and introduce the statement $\{\, C_1 \,\} \oplus \{\, C_2 \,\}$ into the programming language. Operationally, we have the same interpretation as in the classical case: Either the program C_1 can be executed or the program C_2. Denotationally, the weakest preweighting of $\{\, C_1 \,\} \oplus \{\, C_2 \,\}$ is given by

$$\mathsf{wp}\,[\![\{\, C_1 \,\} \oplus \{\, C_2 \,\}]\!]\,(f) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(f) \;\oplus\; \mathsf{wp}\,[\![C_2]\!]\,(f)\,.$$

$\mathsf{wp}\,[\![C_i]\!]\,(f)$ tells us what element we obtain if C_i is executed, and the module addition $\oplus$ tells us how to account for the fact that either C_1 or C_2 could have been executed.

***Conditional Choice*.** We now consider the classical conditional choice $\texttt{if}\ (\,\varphi\,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}$. Operationally, when executing $\texttt{if}\ (\,\varphi\,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}$ on some initial state σ, we check whether σ satisfies the predicate φ. If $\sigma \models \varphi$, the program C_1 is executed; otherwise the program C_2.

Denotationally, the weakest precondition of $\texttt{if}\ (\,\varphi\,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}$ (and in fact also the weakest precondition of $\{\, \texttt{assert}\ \varphi \fatsemi C_1 \,\} \,\square\, \{\, \texttt{assert}\ \neg\varphi \fatsemi C_2 \,\}$) is given by

$$\mathsf{wp}\,[\![\texttt{if}\ (\,\varphi\,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}]\!]\,(\psi) \;=\; \varphi \wedge \mathsf{wp}\,[\![C_1]\!]\,(\psi) \;\vee\; \neg\varphi \wedge \mathsf{wp}\,[\![C_2]\!]\,(\psi)\,.$$

Indeed, whenever an initial state σ satisfies the above precondition then either $\sigma \models \varphi$ and then — since then σ must also satisfy $\mathsf{wp}\ [\![C_1]\!]\ (\psi)$ — executing C_1 will terminate in a final state satisfying φ, or $\sigma \not\models \varphi$ and — since then σ must also satisfy $\mathsf{wp}\ [\![C_2]\!]\ (\psi)$ — executing C_2 will terminate in a final state satisfying φ.

In terms of monoids and modules, $\varphi \wedge \ldots$ *could* be viewed as a scalar-multiplication by either $\mathbb{1}$ (leaving the right operand unaltered) or by $\mathbb{0}$ (annihilating the right operand). However, general monoids do not posses an annihilating $\mathbb{0}$. In order to reenact the desired behavior, we introduce the *Iverson bracket* $[\varphi]$ of a predicate φ, which for a weighting $f \in \mathbb{W}$ defines the weighting

$$([\phi]\, f)(\sigma) \quad = \quad \begin{cases} f(\sigma) & \text{if } \sigma \models \varphi\,, \\ \mathbb{0} & \text{otherwise}\,. \end{cases}$$

With this notation at hand, we define the weakest preweighting of $\texttt{if}\ (\varphi)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ by

$$\mathsf{wp}\ [\![\texttt{if}\ (\varphi)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!]\ (f) \;=\; [\varphi]\,\mathsf{wp}\ [\![C_1]\!]\ (f) \;\oplus\; [\neg\varphi]\,\mathsf{wp}\ [\![C_2]\!]\ (f)\ .$$

By convention, $[\phi]$ binds stronger than $\oplus$. Depending on the truth value of φ, the above weakest preweighting thus *selects* either the preweighting $\mathsf{wp}\ [\![C_1]\!]\ (f)$ or the preweighting $\mathsf{wp}\ [\![C_2]\!]\ (f)$.

Sequential Composition. Our composite statement $C_1 \fatsemi C_2$ is standard. Operationally, C_1 is executed first and then — provided that C_1 terminates — C_2 is executed. A distinguishing feature of the classical weakest precondition transformer is that it moves *backwards* through the program, and the same applies to our weighting transformer, i.e.

$$\mathsf{wp}\ [\![C_1 \fatsemi C_2]\!]\ (f) \;=\; \mathsf{wp}\ [\![C_1]\!]\ (\mathsf{wp}\ [\![C_2]\!]\ (f))\ .$$

Indeed, to compute the weakest preweighting of the composition $C_1 \fatsemi C_2$ w.r.t. to some $f \in \mathbb{W}$, we first compute an intermediate weighting $\mathsf{wp}\ [\![C_2]\!]\ (f)$, which we then feed into $\mathsf{wp}\ [\![C_1]\!]$.

Looping. Operationally, a loop $\texttt{while}\,(\varphi)\,\{C\}$ is equivalent to the infinite nested conditional

$$\texttt{if}\ (\varphi)\ \{C \fatsemi \texttt{if}\ (\varphi)\ \{C \fatsemi \texttt{if}\ (\varphi)\ \{C \fatsemi \ldots\}\ \texttt{else}\ \{\texttt{skip}\}\}\ \texttt{else}\ \{\texttt{skip}\}\}\ \texttt{else}\ \{\texttt{skip}\}\ ,$$

which is the same as saying that $\texttt{while}\,(\varphi)\,\{C\} \equiv \texttt{if}\ (\varphi)\ \{C \fatsemi \texttt{while}\,(\varphi)\,\{C\}\}\ \texttt{else}\ \{\texttt{skip}\}$. With the rules for conditional choice and composition as explained above, it is thus reasonable to require that the preweighting $\mathsf{wp}\ [\![\texttt{while}\,(\varphi)\,\{C\}]\!]\ (f)$ should be a *fixed point* of the function

$$X \;\mapsto\; [\varphi]\,\mathsf{wp}\ [\![C]\!]\ (X) \;\oplus\; [\neg\varphi]\,f$$

which is indeed just the wp-characteristic function defined above. For both the classical weakest precondition transformer as well as for our weighted wp, we choose the semantics to be the *least* fixed point, which exists uniquely if the ambient module $\mathcal{M}$ *is ω-continuous* (see Theorem 4.7 below). In the classical Boolean setting, this corresponds to choosing the *strongest* (least) possible predicate that satisfies the fixed point equation. This ensures that the weakest precondition contains only those initial states where the loop can actually *terminate* in a state satifying the postcondition — but no such states for which the loop cannot terminate at all. Taking the least fixed point in the weighted setting generalizes this intuition as we will show in Theorem 4.15 below.

***Properties of* wp.** Based on the above discussion, we now define wp formally, state healthiness and soundness properties, and provide several examples.

Definition 4.6 (Weakest Preweighting Transformer). The transformer $\mathsf{wp}\colon \mathsf{wGCL} \to (\mathbb{W} \to \mathbb{W})$ is defined by induction on the structure of wGCL according to the rules in Table 2. The function

$$\Phi_f\colon \quad \mathbb{W} \to \mathbb{W}, \quad X \;\mapsto\; [\neg\varphi]\,f \;\oplus\; [\varphi]\,\mathsf{wp}\ [\![C']\!]\ (X)$$

Table 2. Rules defining the weakest preweighting $\mathsf{wp}\,[\![C]\!]\,(f)$ of program C w.r.t. postweighting f.

C	$\mathsf{wp}\,[\![C]\!]\,(f)$
$x \coloneqq E$	$f\,[x/E]$
$C_1 \fatsemi C_2$	$\mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(f))$
$\mathtt{if}\,(\varphi)\,\{C_1\}\,\mathtt{else}\,\{C_2\}$	$[\varphi]\,\mathsf{wp}\,[\![C_1]\!]\,(f) \;\oplus\; [\neg\varphi]\,\mathsf{wp}\,[\![C_2]\!]\,(f)$
$\{C_1\} \oplus \{C_2\}$	$\mathsf{wp}\,[\![C_1]\!]\,(f) \;\oplus\; \mathsf{wp}\,[\![C_2]\!]\,(f)$
$\odot\, a$	$a \odot f$
$\mathtt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\, X.\; [\neg\varphi]\,f \;\oplus\; [\varphi]\,\mathsf{wp}\,[\![C']\!]\,(X)$

whose least fixed point defines the weakest preweighting of $\mathtt{while}\,(\varphi)\,\{C\}$ is called the wp-*characteristic function* of $\mathtt{while}\,(\varphi)\,\{C\}$ with respect to postweighting f. △

Theorem 4.7 (Well-Definedness of wp). *Let the monoid module $\mathcal{M}$ over $\mathcal{W}$ be ω-continuous. For all $\mathcal{W}$-wGCL programs C, the weighting transformer* $\mathsf{wp}\,[\![C]\!]$ *is a well-defined ω-continuous endofunction on the module of weightings over $\mathcal{M}$. In particular, if Φ_f is the* wp*-characteristic function of* $\mathtt{while}\,(\varphi)\,\{C\}$ *with respect to postweighting f, then*

$$\mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) \quad = \quad \bigsqcup_{i\in\mathbb{N}} \Phi_f^i(\mathbf{0}) .$$

Our wp satisfies the following so-called *healthiness criteria* (see e.g. [Hino et al. 2016; Hoare 1978; Keimel 2015; McIver and Morgan 2005]) or homomorphism properties [Back and von Wright 1998]:

Theorem 4.8 (Healthiness). *Let the monoid module $\mathcal{M}$ over $\mathcal{W}$ be ω-continuous. For all $\mathcal{W}$-wGCL programs C, the* wp *transformer is*

(1) monotone, *i.e. for all $f, g \in \mathbb{W}$,* $\quad f \preceq g \quad$ implies $\quad \mathsf{wp}\,[\![C]\!]\,(f) \preceq \mathsf{wp}\,[\![C]\!]\,(g)$,
(2) strict, *i.e.* $\quad \mathsf{wp}\,[\![C]\!]\,(\mathbf{0}) = \mathbf{0}$,
(3) additive, *i.e. for all $f, g \in \mathbb{W}$,* $\quad \mathsf{wp}\,[\![C]\!]\,(f \oplus g) = \mathsf{wp}\,[\![C]\!]\,(f) \oplus \mathsf{wp}\,[\![C]\!]\,(g)$.
(4) Moreover, if $\mathcal{W}$ is commutative, then wp *is* homogeneous, *i.e. for all $a \in W$ and $f \in \mathbb{W}$,*

$$\mathsf{wp}\,[\![C]\!]\,(a \odot f) \;=\; a \odot \mathsf{wp}\,[\![C]\!]\,(f) ,$$

and together with (3), wp *then becomes* linear.

Homogeneity does not hold in general: Consider the formal languages semiring $\mathcal{L}_{\{a,b\}}$ and the program $C = \odot\,\{a\}$ with the constant postweighting $f = \mathbf{1} = \{\epsilon\}$. Then

$$\mathsf{wp}\,[\![C]\!]\,(\{b\}\cdot f) \;=\; \{a\}\cdot\{b\} \;=\; \{ab\} \;\neq\; \{ba\} \;=\; \{b\}\cdot\{a\} \;=\; \{b\}\cdot \mathsf{wp}\,[\![C]\!]\,(f) .$$

The next theorem states that wp indeed generalizes the *map perspective* on classical weakest preconditions as anticipated at the beginning of Section 4.3. The operational semantics as well as $\mathsf{TPaths}_{\langle C,\sigma\rangle}$, wgt, and last are defined in Section 3.3.

Theorem 4.9 (Soundness of wp). *Let the monoid module $\mathcal{M}$ over $\mathcal{W}$ be ω-continuous. For all $C \in$ wGCL, $\sigma \in \Sigma$ and $f \in \mathbb{W}$,*

$$\mathsf{wp}\,[\![C]\!]\,(f)\,(\sigma) \quad = \quad \bigoplus_{\pi\in\mathsf{TPaths}_{\langle C,\sigma\rangle}} \mathsf{wgt}(\pi) \odot f(\mathsf{last}(\pi)) . \qquad (1)$$

$$\begin{array}{lr}
{}^{\bowtie}/\!\!/\!\!/\; g' & (\text{meaning } g \bowtie g') \\
{}^{\mathrm{wp}}/\!\!/\!\!/\; g & (\text{meaning } g = \mathrm{wp}\, [\![C]\!]\, (f)) \\
\quad C & \\
\;/\!\!/\!\!/\; f & (\text{postweighting is } f)
\end{array}
\qquad
\begin{array}{l}
{}^{=}/\!\!/\!\!/\; 2 \quad = g' \\
{}^{\mathrm{wp}}/\!\!/\!\!/\; [x > 0]\,(1+1) \;\oplus\; [x = 0]\,(2 \min 3) \quad = g \\
\quad \texttt{if}\;(x > 0)\;\{\odot 1 ⨾ \odot 1\}\;\texttt{else}\;\{\{\odot 2\} \oplus \{\odot 3\}\} \\
/\!\!/\!\!/\; 0 \quad = f
\end{array}$$

(a) Style for wp annotations. $\bowtie \in \{\preceq, =, \succeq\}$. (b) Annotations for Example 4.10.

Fig. 4. Annotations for weakest preweightings. It is more intuitive to read these from the bottom to top.

wp-*Annotations*. In the spirit of Hoare-style reasoning, we will annotate programs as is shown abstractly in Fig. 4a and concretely in Fig. 4b. Read the annotations *from bottom to top* as follows:

(1) $/\!\!/\!\!/\; f$ This first annotation states that we start our reasoning from postweighting $f \in \mathbb{W}$.
(2) ${}^{\mathrm{wp}}/\!\!/\!\!/\; g$ The superscript wp before the annotation indicates that this annotation is obtained from applying $\mathrm{wp}\,[\![\ldots]\!]\,(\ldots)$. The program passed into wp is the line immediately below this annotation — in this case C — and the continuation passed into wp is the annotation immediately below the program — in this case f. Hence, this annotation states $g = \mathrm{wp}\,[\![C]\!]\,(f)$.
(3) ${}^{\bowtie}/\!\!/\!\!/\; g'$ This last annotation states that $g \bowtie g'$, for $\bowtie \in \{\preceq, =, \succeq\}$. We thus allow rewriting, or (like the classical rule of *consequence* in Hoare logic) to perform a monotonic relaxation.

Let us illustrate wp by means of two examples. Recall our convention that the monoid $\mathcal{W}$ and the module $\mathcal{M}$ stem from a semiring $\mathcal{S}$ unless stated otherwise.

Example 4.10. Let $\mathcal{S} = \mathcal{T} = (\mathbb{N}^{+\infty}, \min, +, \infty, 0)$ be the *tropical* semiring and consider a wGCL-program C. Theorem 4.9 implies that for all states $\sigma \in \Sigma$,

$$\mathrm{wp}\,[\![C]\!]\,(0)(\sigma) \quad = \quad \text{“minimum weight of all terminating computation paths that start in } \sigma\text{”}$$

where the weight of a path is the *usual sum* of all weights along that path. Notice that the above “0” is the map to the *natural number* 0 and *not* the semiring $\mathbf{0} = \infty$. For instance, we can verify that

$$\mathrm{wp}\,[\![\;\texttt{if}\;(x > 0)\;\{\odot 1 ⨾ \odot 1\}\;\texttt{else}\;\{\{\odot 2\} \oplus \{\odot 3\}\}\;]\!]\,(0) \quad = \quad 2\,.$$

For that, consider the program annotations in Fig. 4b, which express that $2 = [x > 0]\,(1+1) \;\oplus\; [x = 0]\,(2 \min 3) = \mathrm{wp}\,[\![C]\!]\,(0)$ where $C = \texttt{if}\;(x > 0)\;\{\ldots\}$. This reflects that if $x > 0$ initially, then the only possible terminating path has weight $1 + 1 = 2$. Otherwise, i.e. if $x = 0$, then the minimum weight of the two possible paths is also 2. △

Example 4.11. Let $\mathcal{S} = \mathcal{L}_\Gamma = (2^{\Gamma^*}, \cup, \cdot, \emptyset, \{\epsilon\})$ be the semiring of formal languages over Γ. Similarly to Example 4.4, we now choose the monoid $\mathcal{W} = \Gamma^*$ of words and view $\mathcal{L}_\Gamma$ as a Γ^*-module, i.e. the weighting-statements are of the form $\odot\, w$ for some single *word* $w \in \Gamma^*$. The weightings $\mathbb{W}$, however, associate an entire *language* to each state. For all initial states $\sigma \in \Sigma$, we have

$$\mathrm{wp}\,[\![C]\!]\,(\{\epsilon\})(\sigma) \quad = \quad \text{“language of all terminating computation paths starting in } \sigma\text{”}\,,$$

where each terminating path contributes the single word obtained from concatenating all symbols occurring in the weight-statements along this path (this may also yield the empty word $\epsilon = \mathbf{1}$). △

4.4 Weakest Liberal Preweightings

The weakest preweighting calculus developed in the previous section assigns a weight to each initial state σ based on the *terminating* computation paths starting in σ and the postweighting f. In particular, wp ignores (more precisely: assigns weight $\mathbf{0}$ to) nonterminating behavior, i.e. the preweighting $\mathsf{wp}\ [\![C]\!]\ (f)$ is independend of the *infinite* computation paths of C. For instance, in the formal languages semiring $\mathcal{L}_{\{a,b\}}$ with monoid $\mathcal{W} = \{a, b\}^*$, we have for all $f \in \mathbb{W}$ that

$$\mathsf{wp}\ [\![\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, a\,\}]\!]\ (f) \quad = \quad \emptyset \quad = \quad \mathsf{wp}\ [\![\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, b\,\}]\!]\ (f)\ ,$$

even though the computation trees of the two programs are clearly distinguishable.

In this section, we define weakest *liberal* preweightings (wlp) as a means to reason about such infinite, i.e. nonterminating, program behaviors, thus generalizing Dijkstra's classical weakest liberal preconditions. Unlike Dijkstra's weakest liberal preconditions who just assign true (instead of false) to *any* nonterminating behavior, our weakest liberal preweightings can inspect nonterminating behavior more nuancedly. As a teaser: our weakest liberal preweightings *can* distinguish between $\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, a\,\}$ and $\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, b\,\}$ as we will demonstrate below.

Reconsidering the *map perspective* on weakest preconditions explained in Section 4.3, the weakest liberal precondition of program C with respect to postcondition ψ maps an initial state σ to true iff (i) C started on σ *can* terminate in a state τ satisfying ψ, or (ii) it is *possible* that C does not terminate at all, or both. In more symbolic terms,

$$\mathsf{wlp}\ [\![C]\!]\ (\,\psi\,)\,(\sigma) \quad = \quad \bigvee_{\tau \in [\![C]\!]_\sigma} \psi(\tau) \ \vee\ [\![C]\!]^{\Uparrow}_\sigma\,,$$

where $[\![C]\!]^{\Uparrow}_\sigma$ holds iff the nondeterministic program C *may not terminate* on σ.[10] We have

$$\mathsf{wlp}\ [\![C]\!]\ (\,\psi\,)\,(\sigma) \ = \ \mathsf{wp}\ [\![C]\!]\ (\,\psi\,)\,(\sigma)\ \vee\ [\![C]\!]^{\Uparrow}_\sigma \qquad \text{and} \qquad \mathsf{wlp}\ [\![C]\!]\ (\,\mathsf{false}\,)\,(\sigma) \ = \ [\![C]\!]^{\Uparrow}_\sigma\,, \quad (2)$$

implying that $\mathsf{wlp}\ [\![C]\!]\ (\,\mathsf{false}\,)$ captures precisely the nonterminating behavior of C, and hence $\mathsf{wlp}\ [\![C]\!]\ (\,\mathsf{false}\,)$ characterizes precisely the difference between $\mathsf{wp}\ [\![C]\!]\ (f)$ and $\mathsf{wlp}\ [\![C]\!]\ (f)$.

In the realm of monoids and modules, the predicate false is the zero $\mathbb{0}$ of the Boolean semiring. We now define a weakest liberal pre*weighting* calculus generalizing (2) by satisfying

$$\forall f \in \mathbb{W}: \qquad \mathsf{wlp}\ [\![C]\!]\ (f) \quad = \quad \mathsf{wp}\ [\![C]\!]\ (f)\ \oplus\ \mathsf{wlp}\ [\![C]\!]\ (\mathbb{0})\ .$$

Intuitively, $\mathsf{wlp}\ [\![C]\!]\ (\mathbb{0})$ captures the weights of the nonterminating paths in C: For the two example programs from the beginning of this subsection considered over the Γ^*-module $\mathcal{L}^\infty_{\{a,b\}}$ of ω-potent formal languages, we get for example

$$\mathsf{wlp}\ [\![\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, a\,\}]\!]\ (\mathbb{0}) \ = \ \{\,a^\omega\,\} \quad \text{and} \quad \mathsf{wlp}\ [\![\mathtt{while}\,(\,\mathsf{true}\,)\,\{\,\odot\, b\,\}]\!]\ (\mathbb{0}) \ = \ \{\,b^\omega\,\}\ .$$

Definition 4.12 (Weakest Liberal Preweighting Transformer). The transformer $\mathsf{wlp}\colon \mathsf{wGCL} \to (\mathbb{W} \to \mathbb{W})$ is inductively defined according to Table 2 with lfp replaced by gfp and with every occurrence of wp replaced by wlp. In particular, $\mathsf{wlp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C'\,\}]\!]\ (f)$ is defined as the *greatest* fixed point of the *characteristic function*

$$\Psi_f\colon \mathbb{W} \to \mathbb{W}, \quad X \quad \mapsto \quad [\neg\varphi]\,f\ \oplus\ [\varphi]\,\mathsf{wlp}\ [\![C']\!]\ (X)\ . \qquad \triangle$$

We obtain a well-definedness result analogous to Theorem 4.7:

Theorem 4.13 (Well-Definedness of wlp). *Let $\mathcal{M}$ be an ω-cocontinuous $\mathcal{W}$-module. For all $\mathcal{W}$-wGCL programs C, the transformer $\mathsf{wlp}\ [\![C]\!]$ is a well-defined ω-cocontinuous endofunction on the module of weightings over $\mathcal{M}$. In particular, if Ψ_f is the wlp-characteristic function of $\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}$ with respect to postweighting f, then*

$$\mathsf{wlp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (f) \quad = \quad \prod_{i \in \mathbb{N}} \Psi_f^i(\top)\ .$$

[10]Recall that we consider angelic nondeterminism.

As stated above, we furthermore get the following fundamental property:

Theorem 4.14 (Decomposition of wlp). *Let $\mathcal{M}$ be an ω-bicontinuous $\mathcal{W}$-module. Then for all programs C and postweightings f,*

$$\mathsf{wlp}\,[\![C]\!]\,(f) \quad = \quad \mathsf{wp}\,[\![C]\!]\,(f) \,\oplus\, \mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})\;.$$

Moreover, we get a statement relating (infinite) computation paths and $\mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})$:

Theorem 4.15 (Soundness of wlp). *Let the monoid module $\mathcal{M}$ over $\mathcal{W}$ be ω-bicontinuous*[11]. *Then for all programs C and initial states σ,*

$$\mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})\,(\sigma) \quad = \quad \bigsqcap_{n\in\mathbb{N}} \;\bigoplus_{\pi\in\mathsf{Paths}^n_{\langle C,\,\sigma\rangle}} \mathsf{wgt}(\pi)\odot\top\;. \tag{3}$$

Note that Theorem 4.15 is phrased in terms of the *finite* computation paths $\mathsf{Paths}^n_{\langle C,\,\sigma\rangle}$. This is because it is somewhat difficult to define a general *infinite product* in $\mathcal{W}$ that is compliant with the way our wlp assigns weights to infinite computation paths. Nevertheless, the right-hand side of (3) depends *only* on the infinite paths which can be seen intuitively as follows: For arbitrary $n \in \mathbb{N}$ consider the *finite* (not necessarily terminating) computation paths up to length n. Let $v(n)$ denote the sum their weights, where the weight of each path is additionally multiplied by $\top$, the top element of $\mathcal{M}$. Then $(v(n))_{n\in\mathbb{N}}$ is a *decreasing* chain in the module $\mathcal{M}$. In the limit (i.e. infimum), all *terminating* computation paths will be ruled out as each of them has some finite length. The limit/infimum of the $v(n)$ exists by our theory and is independent of the program's terminating paths. In fact, for programs that do not exhibit infinite paths, we can show that the limit is $\mathbf{0}$ using Kőnig's classic infinity lemma. We discuss the implications of this in Section 5.2.

Note that Theorem 4.15 indeed implies that wlp is backward compatible to classical weakest liberal preconditions: In the Boolean semiring, the right-hand side of (3) equals true iff there exists an infinite computation path starting in σ, and thus $\mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})\,(\sigma) \;\equiv\; [\![C]\!]^{\Uparrow}_{\sigma}$ holds as expected.

Example 4.16. Reconsider the tropical semiring $\mathcal{T} = (\mathbb{N}^{+\infty}, \min, +, \infty, 0)$ with $\top = 0$. The infimum $\bigsqcap$ in the natural order is the supremum in the standard order on $\mathbb{N}^{+\infty}$, and multiplication with the top element $\top = 0$ is effectless as $a \odot \top = a + 0 = a$. It follows from Theorem 4.15 that

$$\mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})\,(\sigma) \quad = \quad \text{“minimum weight of all \textit{infinite} computation paths starting in } \sigma\text{”}\,,$$

or ∞ — the tropical $\mathbf{0}$ — if no infinite path exists. Hence $\mathsf{wlp}\,[\![C]\!]\,(0)\,(\sigma)$ — where the natural number 0 is the tropical $\mathbf{1}$ — is the minimum path weight among all finite *and infinite* computation paths starting in σ. For example, for the program C given by

$$\mathtt{while}\,(x=2)\,\{\quad \{x \coloneqq 3\,⨟\,\odot 5\} \oplus \{\mathtt{skip}\}\quad\}$$

and initial state σ with $\sigma(x) = 2$, we have $\mathsf{wp}\,[\![C]\!]\,(0)\,(\sigma) = 5$ but $\mathsf{wlp}\,[\![C]\!]\,(0)\,(\sigma) = 0$ because there exists an infinite path (only performing skip) with weight $0 < 5$. △

Example 4.17. Let $\mathcal{M} = \mathcal{L}^{\infty}_{\Gamma}$ be the module of ω-potent formal languages over the monoid of words $\mathcal{W} = \Gamma^*$ (cf. Example 4.11). Thus, weightings $f \in \mathbb{W}$ associate states with languages that contain *both* finite *and ω-words*. Let C be a Γ^*-wGCL program. It follows from Theorem 4.15 that

$$\mathsf{wlp}\,[\![C]\!]\,(\mathbf{0})\,(\sigma) \quad = \quad \begin{matrix}\text{“language of all (finite and } \omega\text{-)words that have some} \\ \text{\textit{infinite} computation path starting in } \sigma \text{ as prefix”}\end{matrix}\;,$$

[11]In the statement, we assert ω-*bi*continuity: our proof makes heavy use of Theorem 4.14 and thus we need wp to be well-defined. However, it might be possible to prove a link between operational semantics and wlp assuming only ω-*co*continuity. But a proof seems much more convoluted than our current one.

where we have identified computation paths with the words they are labelled with. In particular, if all infinite paths of C are weighted with an ω-word, then $\mathsf{wlp}\ [\![C]\!]\ (\mathbf{0})\,(\sigma)$ is precisely the language consisting of all these words. For example, let $\Gamma = \{a, b\}$ and consider the following program C:

$$\texttt{while}\,(\,x = 1\,)\,\{\quad \{\,x \coloneqq 0\,⨾\,\odot\, a\,\} \oplus \{\,\odot\, b\,\}\quad \}$$

If initially $\sigma(x) = 1$, then $\mathsf{wp}\ [\![C]\!]\ (\mathbf{1})\,(\sigma) = \{\,a, ba, bba, \ldots\,\}$, where $\mathbf{1} = \{\,\epsilon\,\}$, but $\mathsf{wlp}\ [\![C]\!]\ (\mathbf{0})\,(\sigma) = \{b^{\omega}\}$ and hence

$$\mathsf{wlp}\ [\![C]\!]\ (\mathbf{1})\,(\sigma) \;=\; \mathsf{wp}\ [\![C]\!]\ (\{\,\epsilon\,\})\,(\sigma) \,\cup\, \mathsf{wlp}\ [\![C]\!]\ (\mathbf{0})\,(\sigma) \;=\; \{\,b^{\omega}, a, ba, bba, \ldots\,\}\ . \qquad \triangle$$

Remark 4.18 (Probabilistic Weakest Liberal Preexpectations). McIver and Morgan [2005] and Kozen [1985] define a probabilistic wlp-semantics where $\mathsf{wlp}\ [\![C]\!]\ (\,0\,)\,(\sigma)$ yields the probability that C diverges on input σ. Technically, probabilistic programs are wGCL-programs over the real semiring $(\mathbb{R}^{\infty}_{\geq 0}, +, \cdot, 0, 1)$ where branching and weighting is restricted to statements of the form $\{\,\ldots\,\}\ {}_{p}\!\oplus_{q}\ \{\,\ldots\,\}$, where $p + q = 1$. However, by Theorem 4.15, our wlp over $\mathbb{R}^{\infty}_{\geq 0}$ yields for *all loops* and *all states* $\mathsf{wlp}\ [\![\mathit{loop}]\!]\ (\,0\,)\,(\sigma) \in \{\mathbf{0} = 0, \top = \infty\}$ and is thus trivial. We cannot simply fix this by choosing probabilities in $[0, 1]$ as our module $\mathcal{M}$ since $[0, 1]$ is not closed under addition. Nonetheless, we can recover the wlp of Kozen [1985]; McIver and Morgan [2005] by considering the *greatest fixed point below or equal to* 1 instead of the true gfp in $\mathbb{R}^{\infty}_{\geq 0}$, i.e. we would consider a modified transformer $\mathsf{wlp}^{\leq 1}$. It is easy to show that this is well-defined and still satisfies Theorem 4.14 and Theorem 4.15 (with the multiplication $\odot\top$ on the right hand side of (3) omitted). $\triangle$

5 VERIFICATION OF LOOPS

For loop-free programs, weakest (liberal) preweightings can be obtained essentially by means of syntactic reasoning. For loops, however, this is not the case since we need to reason about fixed points. This section introduces easy-to-apply proof rules for bounding weakest (liberal) preweightings of loops, generalizing rules from the probabilistic setting [McIver and Morgan 2005].

5.1 Invariant-Based Verification of Loops

Let us fix throughout the rest of the section an ambient monoid $\mathcal{W}$ of program weights and an ambient ω-bicontinuous $\mathcal{W}$-module $\mathcal{M}$. Since the wp- and wlp-characteristic functions of loops are ω-(co)continuous (see Theorem 4.7 and 4.13), we obtain proof rules for loops by Park induction [Batz et al. 2022, Thm. A.4]:

Theorem 5.1 (Induction Rules for Loops). *Let Φ_f and Ψ_f be the* wp- *and* wlp-*characteristic functionals of the loop* $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *with respect to postweighting f. Then for all $I \in \mathbb{W}$,*

$$\begin{array}{lll} \Phi_f(I) \,\preceq\, I & \text{implies} & \mathsf{wp}\ [\![\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,) \,\preceq\, I\,, \qquad \text{and} \\ I \,\preceq\, \Psi_f(I) & \text{implies} & I \,\preceq\, \mathsf{wlp}\ [\![\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,)\ . \end{array}$$

The weightings I are called wp-*superinvariants* and wlp-*subinvariants*, respectively (or just *invariants* if clear from context). In many cases — in particular for loop-free loop bodies — the above proof rules are easy to apply as they only require to apply the respective characteristic functional *once*. Example 5.5 demonstrates invariant-based reasoning and our annotation-style for loops.

What about the converse directions, i.e. lower bounds for wp and upper bounds for wlp? For that, the analogous formulations of the above proof rules do not hold in general [Kaminski 2019]. In the next subsection we show that in the case of *terminating* programs, these formulations *do* hold.

5.2 Terminating Programs and Unique Fixed Points

The notion of *universal certain termination* is central to the results of this section:

Definition 5.2 (Universal Certain Termination). A wGCL-program C *terminates certainly* on initial state $\sigma \in \Sigma$ if there does not exist an infinite computation path starting in $\langle C,\, \sigma \rangle$. Moreover, C is *universally certainly terminating* (UCT) if it terminates certainly on all $\sigma \in \Sigma$. △

Certain termination of a program $C \in \mathsf{wGCL}$ is also known as *demonic termination* of the program obtained from C by ignoring all weight-statements and interpreting branching as demonic nondeterminism. Note that all loop-free programs are trivially UCT. A well-established method for proving certain termination is by use of *ranking functions* [Dijkstra 1975]. An important consequence of UCT is that wp and wlp coincide. This is intuitively clear because if C is UCT then $\mathsf{wlp}\ [\![C]\!]$ has no additional nonterminating behavior to account for compared to $\mathsf{wp}\ [\![C]\!]$. Formally:

Theorem 5.3 (Unique Fixed Points by Universal Certain Termination). *Let* $\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *have a UCT loop body C and let Φ_f and Ψ_f be its* wp- *and* wlp-*characteristic functionals with respect to an arbitrary postweighting f. Then $\Phi_f = \Psi_f$.*
Furthermore, let $I, J \in \mathbb{W}$ be fixed points of Φ_f. Then

$$\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\} \text{ terminates certainly on } \sigma \qquad \text{implies} \qquad I(\sigma) \;=\; J(\sigma)\;.$$

Moreover, if $\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *is UCT, then Φ_f has a unique fixed point and*

$$\mathsf{wp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,) \;=\; \mathsf{wlp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,)\;.$$

Hence, the converse directions of the rules in Theorem 5.1 *do* hold for UCT loops with UCT loop-body. In particular, we can reason about *exact* weakest (liberal) preweightings of such loops.

Corollary 5.4. *If both* $\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *and C are UCT, then for all $f \in \mathbb{W}$ and all $I \in \mathbb{W}$,*

$$\begin{aligned} I \;\preceq\; \Phi_f(I) \qquad &\text{implies} \qquad I \;\preceq\; \mathsf{wp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,)\;, \textit{ and} \\ \Psi_f(I) \;\preceq\; I \qquad &\text{implies} \qquad \mathsf{wlp}\ [\![\mathtt{while}\,(\,\varphi\,)\,\{\,C\,\}]\!]\ (\,f\,) \;\preceq\; I\;. \end{aligned}$$

Let us now look at reasoning about loops in action. For this, we extend our annotation scheme to loops as shown in Fig. 5a. Again, read the annotations from bottom to top as follows and consider $\bowtie$ as $\preceq$ for simplicity:

(1) $/\!\!/\!\!/\ f$ We start our reasoning from postweighting $f \in \mathbb{W}$.
(2) $\|\ I$ We choose (*creatively*) an invariant I which we are going to push through the loop body.
(3) ${}^{\mathrm{wp}}/\!\!/\!\!/\ I'$ This annotation is obtained (*uncreatively*) from applying $\mathsf{wp}\ [\![\ldots]\!]\ (\ldots)$, just as in Fig. 4. The program passed into wp is the loop body C and the continuation is the invariant I. Hence, this annotation states $I' = \mathsf{wp}\ [\![C]\!]\ (\,I\,)$ and by that we have pushed I through the loop body.
(4) ${}^{\preceq}/\!\!/\!\!/\ I''$ This annotation states that $I' \preceq I''$, i.e. I'' overapproximates I', just as in Fig. 4.
(5) ${}^{\Phi}/\!\!/\!\!/\ g$ This annotation g is obtained from I'' — the result of pushing the invariant I through the loop body (and possibly overapproximating the result) — by constructing $g = [\neg\varphi]\, f \oplus [\varphi]\, I''$. This annotation states that $g \succeq \Phi_f(I)$.
(6) ${}^{\preceq}/\!\!/\!\!/\ I$ This annotation states that $g \;\preceq\; I$, just as in Fig. 4. Since $\Phi_f(I) \;\preceq\; g \;\preceq\; I$, this final annotation states by Theorem 5.1 that $\mathsf{wp}\ [\![\mathit{loop}]\!]\ (\,f\,) \preceq I$ and we could continue reasoning with I.

Example 5.5. Consider the *arctic* semiring $\mathcal{A} = (\mathbb{N}^{+\infty} \cup \{\,-\infty\,\},\ \max,\ +,\ -\infty,\ 0)$ and the program

$$C \quad = \quad \mathtt{while}\,(\,x > 0 \wedge y > 0\,)\,\{\quad \{\,x \coloneqq x - 1\,\fatsemi\, y \coloneqq y + 1\,\} \oplus \{\,y \coloneqq y - 1\,\}\,\fatsemi \quad \odot 1 \quad \}\;.$$

C is UCT, witnessed by the ranking function $r = 3x + 2y$: Both branches of the loop body strictly decrease the value of r. We verify that $I = [\neg\varphi]\, 0 \oplus [\varphi]\, (2(x-1) + y)$, where $\varphi = (x > 0 \wedge y > 0)$,

```
$^{\bowtie}/\!\!/\!\!/\ I$                                  (meaning $g \bowtie I$)
$^{\Phi}/\!\!/\!\!/\ g$                                     ($g = [\neg\varphi]\, f \oplus [\varphi]\, I''$)
while ( $\varphi$ ) {
    $^{\bowtie}/\!\!/\!\!/\ I''$                            (meaning $I' \bowtie I''$)
    $^{\mathrm{wp}}/\!\!/\!\!/\ I'$                         (meaning $g = \mathrm{wp}\ [\![C]\!]\ (f)$)
    $C$
    $|\!|\!|\ I$                                            (we employ invariant $I$)
}
$/\!\!/\!\!/\ f$                                            (postweighting is $f$)
```

(a) Annotation style for loops using invariants.

```
$^{=}/\!\!/\!\!/\ [x{=}0 \vee y{=}0]0 \oplus [x{=}0 \vee y{=}0](2x{+}y) = I$
$^{\Phi}/\!\!/\!\!/\ [x{=}0 \vee y{=}0]0 \oplus [x{>}0 \wedge y{>}0]I'' = g$
while ( $x > 0 \wedge y > 0$ ) {
    $^{=}/\!\!/\!\!/\ [(x{>}1) \vee (x{>}0 \wedge y{>}1)](2x{+}y) \oplus 1 \odot [x{\le}1 \vee y{\le}1]0 = I''$
    $^{\mathrm{wp}}/\!\!/\!\!/$ (expression ommited) $= I'$
      $\{ x \coloneqq x{-}1 \,⨟\, y \coloneqq y{+}1 \} \oplus \{ y \coloneqq y{-}1 \} \,⨟ \odot 1$
      $|\!|\!|\ [x{=}0 \vee y{=}0]0 \oplus [x{=}0 \vee y{=}0](2x{+}y) = I$
}
$/\!\!/\!\!/\ 0 = f$
```

(b) wp loop annotations for Example 5.5.

Fig. 5. Inside the loop, we push an invariant I (provided externally, denoted by $|\!|\!|\ I$) through the loop body, thus obtaining I'' which is (possibly an over- or underapproximation of) $I' = \mathrm{wp}\ [\![C']\!]\ (I)$. Above the loop head, we then annotate $g = [\neg\varphi]\, f \oplus [\varphi]\, I''$. In the first line, we establish $g \bowtie I$, for $\bowtie\ \in \{\preceq, =, \succeq\}$.

is a fixed point of Φ_0 in Fig. 5b. Hence, by Corollary 5.4, we get $\mathrm{wp}\ [\![C]\!]\ (0) = I$. By Theorem 4.9, $\mathrm{wp}\ [\![C]\!]\ (0)$ is the maximum weight among all terminating computations paths. In C, the weight of a path is the number of times it passes through the loop body. We thus conclude that the number of C's loop iterations is bounded by $2(x-1)+y$ if initially $x > 0 \wedge y > 0$ holds. This bound is sharp.

6 CASE STUDIES

6.1 Competitive Analysis of Online Algorithms by Weighted Programming

Field:	Competitive Analysis	**Model:**	Optimization Problem	**Techniques:**	wp
Problem:	Ski Rental Problem	**Semiring:**	Tropical Semiring		

We now demonstrate *how to model optimization problems by means of weighted programming* and how to reason about *competitive ratios of online algorithms* [Borodin and El-Yaniv 1998; Fiat and Woeginger 1998] *on source code level* by means of our wp calculus with the aid of invariants. In particular, we model both the *optimal solution to the Ski Rental Problem itself* as well as *the optimal deterministic online algorithm for the problem* as weighted programs. We argue that weighted programming provides a natural formalism for reasoning about the competitive ratio of online algorithms since weighted programs enable the succinct integration of *cost models*.

6.1.1 Online Algorithms and Competitive Analysis. Online algorithms perform their computation without knowing the entire input a priori. Rather, parts of the input are revealed to the online algorithm during the course of the computation. We consider here the well-known *Ski Rental Problem* [Komm 2016]: Suppose we go an a ski trip for an *a priori unknown* number of $n \geq 1$ days and we do *not* own a pair of skis. At the beginning of each day, we must choose between either renting skis for exactly one day (cost: 1 Euro) or to buy a pair of skis (cost: y euros).

The optimization goal is to minimize the total cost for the whole trip. If we knew the duration n of the trip a priori, the optimal solution would be rather obvious: If $n \geq y$, we *buy* the skis. Otherwise, we are cheaper off renting every day. This situation would correspond to an *offline* setting, with both n and y at hand, allowing for an optimal solution. Conversely, if the trip duration n is *unkown* and only the cost y of the skis is known, we are in an online setting of the Ski Rental Problem.

```
while ( $n > 0$ ) {
    $n := n - 1$ ⨟
    { (* rent *)
        $\odot\, 1$
    } $\oplus$ { (* buy *)
        $\odot\, y$ ⨟
        $n := 0$ (* terminate *)
    } }
```

(a) The program C_{opt}.

```
$c := 0$ ⨟
while ( $n > 0$ ) {
    $n := n - 1$ ⨟ $c := c + 1$ ⨟
    if ( $c < y$ ) {
        $\odot\, 1$
    } else {
        $\odot\, y$ ⨟ $n := 0$
    } }
```

(b) The program C_{onl}.

Fig. 6. The *optimal solution* to the Ski Rental Problem is modeled by C_{opt}. The program C_{onl} implements the optimal *deterministic online algorithm*.

Lacking knowledge about the entire input a priori often comes at the cost of *non-optimality*: An online algorithm typically performs worse than the optimal offline algorithm. *Competitive analysis* [Borodin and El-Yaniv 1998] is a technique for measuring the degree of optimality of an online algorithm. The central notion is the *competitive ratio* of an online algorithm. Given a problem instance ρ, denote by $\mathsf{ONL}(\rho)$ and $\mathsf{OPT}(\rho)$ the cost of an online algorithm ONL and the cost of its optimal offline counterpart OPT on ρ, respectively. The competitive ratio of ONL is defined as

$$\sup_{\rho} \frac{\mathsf{ONL}(\rho)}{\mathsf{OPT}(\rho)} ,$$

i.e. the smallest constant upper-bounding the ratio between the cost of ONL and OPT for all problem instances ρ. We determine such competitive ratios by wp-reasoning on weighted programs.

6.1.2 *Modeling Infinite-State Online Algorithms as Weighted Programs.*

Together with wp-reasoning, weighted programs over the tropical semiring $\mathcal{T}$ provide an appealing formalism for the competitive analysis of *infinite-state* online algorithms since (1) (nondeterministic) programs naturally describe algorithmic problems- and solutions, and (2) reasoning on *source code level* enables reasoning about *infinite-state* models. Modeling online algorithms as weighted programs is inspired by [Aminof et al. 2009, 2010], who employ finite-state weighted automata for the automated competitive analysis of *finite-state* online algorithms. We drop the restriction to finite-state algorithms which comes, however, at the cost of full automation of their verification.

Consider the nondeterministic weighted program C_{opt} on the left-hand side of Figure 6 (let us ignore the annotations for the moment). An initial program state $\sigma \in \Sigma$ fixes an instance of the Ski Rental Problem given by the duration $\sigma(n)$ of the trip and the cost $\sigma(y)$ of the skis. Every execution of C_{opt} on σ corresponds to one possible solution: Each iteration of the loop corresponds to one day of the ski trip. As long as the trip did not end ($n > 0$), we can either rent the skis (first branch) or buy the skis (second branch). If we buy the skis, there is no further choice to be taken, so the loop terminates. The cost of each choice is modeled by weighing the respective branches appropriately.

Now recall that in the tropical semiring $\mathcal{T}$ we have $\oplus = \min$, $\odot = +$, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$. Thus, the weight of a terminating computation path π is the sum of the weights along π, i.e. the cost of the solution given by π. This enables determining the *optimal cost* for every initial program state σ, i.e. every instance of the Ski Rental Problem, by wp-reasoning (cf. Example 4.11) since

$$\mathsf{wp}\, [\![C_{\text{opt}}]\!]\, (\mathbf{1})\, (\sigma) \;=\; \text{“minimum weight of all terminating computation paths starting in } \sigma\text{”} .$$

Program C_{onl} on the right-hand side in Figure 6 implements the optimal solution for the *online* version of the Ski Rental Problem. The decisions made by C_{onl} must therefore not depend on n. Let us compare the programs C_{opt} and C_{onl}. Program C_{onl} is obtained from C_{opt} by introducing a counter c keeping track of the elapsed time and by replacing the nondeterministic choice in C_{opt} by a *deterministic* one. As long as the current duration of the trip is smaller than the cost of the skis, we rent the skis. As soon as this duration is at least the cost of the skis, we buy the skis. Since C_{onl} is deterministic, the *cost* of C_{onl} on σ is given by $\mathsf{wp}\;[\![C_{\mathrm{onl}}]\!]\;(\mathbf{1})\,(\sigma)$.

6.1.3 Determining Competitive Ratios by wp*-Reasoning.* Due to the above reasoning,

$$\sup_{\sigma\in\Sigma} \frac{\mathsf{wp}\;[\![C_{\mathrm{onl}}]\!]\;(\mathbf{1})\,(\sigma)}{\mathsf{wp}\;[\![C_{\mathrm{opt}}]\!]\;(\mathbf{1})\,(\sigma)}\;.$$

is the competitive ratio of C_{onl}. Hence, we obtain the competitive ratio of C_{onl} by determining $\mathsf{wp}\;[\![C_{\mathrm{opt}}]\!]\;(\mathbf{1})$ and $\mathsf{wp}\;[\![C_{\mathrm{onl}}]\!]\;(\mathbf{1})$. This can be done in an *invariant-based* manner:

Theorem 6.1. *We have*

$$\mathsf{wp}\;[\![C_{opt}]\!]\;(\mathbf{1}) = n \oplus y\;,\textit{ and } \mathsf{wp}\;[\![C_{onl}]\!]\;(\mathbf{1}) = [n=0]\,0 \oplus [0<y]\,\big((2y-1)\oplus[n \le y-1]\,n\big)\;.$$

Proof. Since both C_{opt} and C_{onl} are UCT (witnessed by the ranking function n), it suffices to show that the above weightings are fixed points of the respective characteristic functional by Corollary 5.4. We proceed by annotating the programs. See [Batz et al. 2022, App. E.1] for details. □

The fact that $\mathsf{wp}\;[\![C_{\mathrm{opt}}]\!]\;(\mathbf{1}) = n \oplus y = \lambda\sigma.\;\sigma(n)\min\sigma(y)$ corresponds to our informal description from the beginning of this section: Depending on whether the duration of the trip n exceeds the cost y of the skis, it is optimal to either immediately buy the skis or to keep renting them every day. The cost $\mathsf{wp}\;[\![C_{\mathrm{onl}}]\!]\;(\mathbf{1})$ of C_{onl} is more involved. If $n=0$, the cost of C_{onl} is 0. Otherwise, i.e. if the trip lasts for at least one day, there are two cases. If n is strictly smaller than y, then the cost of C_{onl} is the minimum of $2y-1$ and n. Otherwise, i.e. if n is at least y, the cost of C_{onl} is $2y-1$.

We can now determine the competitive ratio of C_{onl}. Let, for simplicity, both $n>0$ and $y>0$ so that $n\oplus y>0$. This assumption is reasonable since the problem becomes trivial if the trip ends immediately or the skis are gratis. Given two weightings f, g with $g>0$, we define $\frac{f}{g} = \lambda\sigma.\frac{f(\sigma)}{g(\sigma)}$. We conclude that the competitive ratio of C_{onl} is 2 since 2 is the smallest constant upper bounding

$$\frac{\mathsf{wp}\;[\![C_{\mathrm{onl}}]\!]\;(\mathbf{1})}{\mathsf{wp}\;[\![C_{\mathrm{opt}}]\!]\;(\mathbf{1})} \;=\; \frac{(2y-1)\oplus[n\le y-1]\,n}{n\oplus y} \;=\; [n\ge y]\,\big(2-\frac{1}{y}\big)\oplus[n<y]\,1\;.$$

6.2 Mutual Exclusion

Field:	Formal Verification	**Model:**	Computation Traces	**Techniques:**	wlp
Problem:	Mutual Exclusion	**Module:**	ω-potent Formal Languages		

In this case study, we instantiate weighted programming with the module of ω-potent formal languages to reason about *infinite* behaviors of a semaphore-based mutual exclusion algorithm. This is done in an invariant-based manner enabled by wlp-reasoning.

6.2.1 A Mutual Exclusion Protocol. Consider the program C_{mut} shown in Fig. 7a and disregard the weightings for the moment. Program C_{mut} models N processes participating in a semaphore-based mutual exclusion protocol. On each iteration of the non-terminating while loop, a scheduler selects one of the N processes. The status $\ell[i]$ of the selected process i is either idle (n), waiting (w), or critical (c). If the process i idles, it enters the waiting state. If the process i is waiting, it checks whether the binary semaphore (modeled by the shared variable y) allows to enter the critical section

($y > 0$) and, if so, enters the critical section. Otherwise, i.e. if $y = 0$, the process must continue waiting. Finally, if the process i is in the critical section, it releases the critical section and updates the semaphore appropriately.

It can be shown by standard means that the protocol modeled by C_{mut} indeed ensures mutual exclusion. That is, whenever we start in a state where at most one process is in the critical section and $y = 0$, it will never be the case that more than one process is in the critical section. However, the protocol exhibits unfair behavior. Suppose the semaphore forbids some waiting process k to enter the critical section, i.e. $y = 0$ and $\ell[k] = w$. It is then possible that the scheduler behaves in an adversarial manner such that process k is going to *starve*, i.e. wait *forever.*

6.2.2 Reasoning about Infinite Behavior by wlp*-Reasoning.* We prove that the protocol exhibits unfair behavior by weighted programming and wlp-reasoning. To that end, we instantiate our framework with the Γ^*-module[12] $\mathcal{L}_\Gamma^\infty$ of ω-potent formal languages over Γ (cf. Example 4.17), where

$$\Gamma \;=\; \bigcup_{j \in \mathbb{N}\setminus\{0\}} \{R_j, W_j, C_j\}\ .$$

Recall that our module addition $\oplus$ is union $\cup$, the monoid and scalar-multiplications are concatenations and the zero element is $\mathbf{0} = \emptyset$. Intuitively, (finite or infinite) behaviors of C_{mut} correspond to (finite or infinite) words over Γ. For instance, the ω-word $C_1 W_2^\omega$ indicates that process 1 enters the critical section and that subsequently process 2 waits forever. This is realized by weighing the branches of the loop body in C_{mut} appropriately: If the process i enters the critical section, waits, or releases the critical section, we weight the corresponding branch by C_i, W_i, or R_i, respectively. This is similar to labeling the states of a transition system by atomic propositions to express properties of the system in, e.g. LTL [Baier and Katoen 2008]. Notice, however, that the transition system underlying C_{mut} is *infinite* so that standard finite-state model checking techniques do not apply. Now recall from Example 4.17 that the *language of ω-words* produced by the loop in C_{mut} on initial state σ is $\mathsf{wlp}\, [\![C_{\mathrm{mut}}]\!]\, (\mathbf{0})\, (\sigma)$. Since the natural ordering $\preceq$ on $\mathcal{L}_\Gamma^\infty$ is $\subseteq$, verifying that C_{mut} indeed exhibits the described unfair behavior boils down to proving that

$$[1 \le k \le N \wedge \ell[k] = w \wedge y = 0]\, W_k^\omega \;\preceq\; \mathsf{wlp}\, [\![C_{\mathrm{mut}}]\!]\, (\mathbf{0})\ ,$$

i.e. we are obliged to prove a *lower bound* on the weakest liberal preweighting of C_{mut} w.r.t. (irrelevant) postweighting $\mathbf{0}$, which is done in an invariant-based manner [Batz et al. 2022, App. E.2]. The above property indeed states that C_{mut} exhibits unfair behavior: Whenever some process k is waiting and the semaphore forbids entering the critical section ($y = 0$), the behavior W_k^ω is possible, i.e. process k might wait forever.

6.3 Proving a Combinatorial Identity by Program Analysis

Field:	Combinatorics	**Model:**	Combinatorial class	**Techniques:**	wp
Problem:	Counting bit patterns	**Semiring:**	Natural numbers		

We instantiate our framework with the semiring $(\mathbb{N}^{+\infty}, +, \cdot, 0, 1)$ to count the number of computation paths in our programs. If a program C does not contain weight-statements, it follows from Theorem 4.9 that the number of terminating computation paths starting in σ is given by $\mathsf{wp}\, [\![C]\!]\, (\,1\,)\, (\sigma)$. More generally, given a predicate φ over the program variables, the number of paths terminating in a state satisfying φ on initial state σ is given by $\mathsf{wp}\, [\![C]\!]\, (\,[\varphi]\, 1\,)\, (\sigma)$. Thus, counting computation paths reduces to weakest preweighting-reasoning as illustrated in the following example.

[12]This is the only example in this section where we do not pursue the *default* method of specifying both the monoid $\mathcal{W}$ and the module $\mathcal{M}$ at once by means of a single semiring $\mathcal{S}$.

```
while ( true ) {
    $\bigoplus_{j=1}^{N} \{ i \coloneqq j \}$ ⨾
    if ( $\ell[i] = n$ ) {
        $\ell[i] \coloneqq w$
    } else if ( $\ell[i] = w$ ) {
        if ( $y > 0$ ) { $\odot C_i$ ⨾ $y \coloneqq y - 1$ ⨾ $\ell[i] \coloneqq c$ }
        else { $\odot W_i$ }
    } else if ( $\ell[i] = c$ ) {
        $\odot R_i$ ⨾ $y \coloneqq y + 1$ ⨾ $\ell[i] \coloneqq n$
    }
}
```

(a) The program C_{mut}.

```
$m \coloneqq 0$ ⨾ $c \coloneqq 0$ ⨾ // $res \coloneqq$ [] ⨾
while ( $n > 0$ ) {
    $n \coloneqq n - 1$ ⨾ {
        $c \coloneqq 0$ // append ( $res$, 0 ) ⨾
    } ⊕ {
        $c \coloneqq c + 1$ ⨾ // append ( $res$, 1 ) ⨾
        $m \coloneqq \max(m, c)$
    }
}
```

(b) The program C_{count}.

Fig. 7. The program C_{mut} is a mutual exclusion protocol adapted from [Baier and Katoen 2008]. The program C_{count} generates n-bit strings and stores the maximum number of consecutive 1's in m.

Suppose we were to count the number of bit strings of length n that avoid the pattern "11". Program C_{count} in Fig. 7b non-deterministically "constructs" bit strings of length equal to the (input) variable n and simultaneously keeps track of the maximum amount of consecutive 1's that have occurred in variable m. Since we are interested in counting strings not containing "11", we have to determine $\text{wp}\,[\![C_{\text{count}}]\!]\,(\,[m \leq 1]\,1\,)$. To handle the loop in C_{count}, we employ the loop invariant

$$I \quad \coloneqq \quad [m \leq 1]\,([c = 0]\,\text{Fib}(n+2) \;\oplus\; [c > 0]\,\text{Fib}(n+1))$$

and verify that I is indeed a fixed point of the wp-characteristic functional of the loop [Batz et al. 2022, App. E.3]. Here, $\text{Fib}(n)$ is the n-th *Fibonacci number* defined recursively via $\text{Fib}(0) \coloneqq 0$, $\text{Fib}(1) \coloneqq 1$, and for all $n \geq 2$, $\text{Fib}(n) \coloneqq \text{Fib}(n-1) + \text{Fib}(n-2)$. Since C_{count} is obviously certainly terminating and I is a fixed point of the wp-characteristic function of the loop w.r.t. postweighting $[m \leq 1]\,1$, we have

$$\text{wp}\,[\![C_{\text{count}}]\!]\,(\,[m \leq 1]\,1\,) \quad = \quad I\,[c/0]\,[m/0] \quad = \quad \text{Fib}(n+2)\,,$$

by Theorem 5.3, i.e. the number of "11"-avoiding bit strings of length n is equal to $\text{Fib}(n+2)$.

7 RELATED WORK

We organize related works in three categories: (1) Other generalized predicate transformers, (2) semiring programming paradigms, (3) other approaches to modelling optimization problems.

7.1 Generalized Predicate Transformers and Hoare Logics

A well-known concrete instance of generalized, quantitative predicates are *potential functions* $\Phi\colon \Sigma \longrightarrow \mathbb{R}_{\geq 0}$. Such functions are used in amortized complexity analysis [Tarjan 1985] can be regarded an instance of the weightings introduced in this paper. Carbonneaux [2018]; Carbonneaux et al. [2015] present a resource bound verification system for a subset of C programs based on potential functions. A non-trivial subset of their verification rules can be recovered by instantiating our framework with the tropical semiring, and interpreting their resource consumption statement `tick(n)` as our weight primitive $\odot\, n$. More specifically, Carbonneaux et al. [2015] define a quantitative Hoare triple $\{\, \Phi \,\}\, P \,\{\, \Phi' \,\}$, where Φ, Φ' are potential functions, and P is a (deterministic) program. Such a triple is valid iff for all initial states $\sigma \in \Sigma$ such that P terminates in a final state σ' it holds that $\Phi(\sigma) \geq n + \Phi'(\sigma')$, where n is the resource consumption of P started on σ. It follows that $\{\, \mathrm{wp}\, [\![P]\!]\, (\, \Phi' \,) \,\}\, P \,\{\, \Phi' \,\}$ is always a valid triple; furthermore, $\mathrm{wp}\, [\![P]\!]\, (\, \Phi' \,)$ is the *least* potential X that validates the triple $\{\, X \,\}\, P \,\{\, \Phi' \,\}$. While the programming language from [Carbonneaux et al. 2015] has advanced features such as procedures and recursion, it lacks a non-deterministic choice as present in wGCL. A promising direction for future work is to investigate whether the automatic inference algorithm of Carbonneaux et al. [2015] can be extended to non-deterministic programs.

Very recent works have studied predicate transformers and Hoare-style logics from an abstract categorical perspective. A generic approach to define predicate transformers, like our wp and wlp, is given by Aguirre and Katsumata [2020], but *only for loop-free programs*. On the loop-free fragment of wGCL, our weakest preweighting transformer wp is an instance of their framework. They capture the computational side effects, like our weightings, in a *monad*. More precisely, our transformer is obtained from the composed monad `MSet (Writer w -)` of a multiset monad `MSet -` that distributes over a writer monad `Writer w -`. This specific instance, however, is not discussed explicitly by Aguirre and Katsumata [2020]. The writer monad corresponds to our weighting monoid $\mathcal{W}$, whereas the multiset monad captures the branching construct $\{\, C_1 \,\} \oplus \{\, C_2 \,\}$ that we treat via $\mathcal{W}$-modules. In contrast to their work, our wp is defined for loops. Moreover, we introduce *two* transformers, wp for finite computations and wlp that additionally accounts for *infinite computations*. Finally, the correspondence to an operational semantics is not established in [Aguirre and Katsumata 2020]. An interesting direction for future work is to explicitly construct a *strongest postcondition* transformer for weighted programming, which Aguirre and Katsumata [2020] define non-constructively as an adjoint to wp. Problems with defining strongest postexpectations for probabilistic programs, see [Jones 1990], demonstrate that giving a *concrete* strongest post semantics is far less easy, even if it can be defined abstractly as an adjoint.

In a similar spirit, Gaboardi et al. [2021] introduce the notion of *graded categories* to unify *graded monadic* and *graded comonadic* effects. The gradings are over partially ordered monoids (pomonoids) and can, for example, model probabilities or resources like our weightings. In the setting of *imperative* languages, they consider it "natural to have just the multiplicative structure of the semiring as a pomonoid" [Gaboardi et al. 2021, Sec. 6] because their programs only have one input and output. The additive structure of semiring gradings has been used to join multiple *inputs* for resource consumption in the λ-calculus with *comonadic contexts* [Brunel et al. 2014; Ghica and Smith 2014; Petricek et al. 2014]. In contrast, we use addition to join multiple *outputs* in *monadic computations*, e. g. *branching* in our examples (Fig. 6 and Fig. 7). Hence, it might be interesting future work to extend their categorical semantics with branching. They go on to construct a *Graded Hoare Logic (GHL)* with *judgments* $\vdash_w \{\phi\}\, C\, \{\psi\}$ corresponding to $[\phi]\, w \preceq \mathrm{wp}\, [\![C]\!]\, (\, [\psi]\, \mathbf{1}\,)$ given *(Boolean)* pre- and postconditions ϕ, ψ, program C, and a weight w from a semiring. Although *unbounded* loops have been studied in *concrete instances*, they restrict to *bounded* loops in the general setting: "This allows us to focus on the grading structures for total functions, leaving the study of the interaction between grading and partiality to future work." [Gaboardi et al. 2021, Sec. 2]. Our work does not impose such restrictions. Both of our verification calculi wp and wlp deal with possibly unbounded loops.

Swierstra and Baanen [2019] handle effects by monads, focussing on *functional* rather than *imperative* programming. They show how to *synthesize* programs from specifications using general results on predicate transformers. Combining these synthesis techniques with the above monad instance of [Aguirre and Katsumata 2020] for the synthesis of weighted programs is an interesting direction for future work.

7.2 Computing with Semirings

There exist a number of computation and programming paradigms in the literature that — similarly to our approach — are parameterized by a semiring. O'Conner [2012] and Dolan [2013] show that computational problems such as shortest paths, deriving the regular expression of a finite automaton, dataflow analysis, and others can be reduced to linear algebra over a suitable semiring. They also provide concise Haskell implementations solving the resulting matrix problems in a unified way. The heart of these techniques is to compute the so-called *star* or *closure* $x^* = 1 + x + x^2 + \ldots$ where x is a matrix over the semiring. The same x^* is also the least solution of the equation $x^* = 1 + x \cdot x^*$. This fixed point equation is closely related to the lfp occurring in our wp. In fact, it can be interpreted as an automata-theoretic explicit-state analog to our wp. Our framework, however, extends this to infinite state spaces and allows reasoning in a symbolic fashion. The above techniques, on the other hand, would require an infinite transition matrix x, and are therefore limited to *finite-state* problems, e. g. shortest paths in *finite* graphs.

Functional and declarative approaches for programming with semirings have also been explored. Laird et al. [2013] and Brunel et al. [2014] consider functional languages parameterized by a semiring and provide a categorical semantics. Their languages feature weighting computation steps similar to our language. Additionally, Brunel et al. [2014] provide static analysis techniques to obtain upper bounds on the weight of a computation. Indeed, with an appropriate semiring, the semantics defined in these works also allows reasoning about e. g. best/worst-case resource consumption, reachability probabilities, or expected values. In contrast to [Brunel et al. 2014; Laird et al. 2013], our programming language is imperative and our semantics generalizes weakest preconditions. Moreoever, while Laird et al. [2013] exemplify how their framework can be used to detect infinite reduction sequences, it does not provide a general way to assign a weight to diverging computation paths as our wlp does. Brunel et al. [2014] do not deal with infinite computations.

Belle and De Raedt [2020] pursue a declarative approach by computing the *weighted model count* of logical formulae in some theory where the literals are weighted in a semiring. Applications include matrix factorization, computing polyhedral volumes, or probabilistic inference. Furthermore, Balkir et al. [2020]; Cohen et al. [2008] study *weighted logic programs* with a focus on parsers. This declarative paradigm is, however, rather different from our weighted programs which allow specifying models in an algorithmic, imperative manner.

Kleene Algebras with Tests (KAT) [Kozen 1999, 2000] can model imperative programs in an abstract fashion by identifying them with the objects from a *Kleene algebra* with an embedded Boolean subalgebra. An important application of KAT is *equational reasoning*; and hence to e. g. derive the rules of Hoare logic by applying algebraic manipulations. Note that a Kleene algebra is itself an idempotent semiring whose purpose, however, is not to model weights of any kind but the programs themselves. Nonetheless, to reason about weighted computations similar to us, KAT was recently generalized to Graded KAT [Gomes et al. 2019] by replacing the Boolean subalgebra with a more general object that can be viewed as a semiring with additional operations and axioms. The elements of this semiring constitute the graded (or weighted) outcomes of the tests. However, (Graded) KAT are no concrete programming languages; their main purpose is to prove general results about imperative languages with loops and conditionals in an abstract fashion. Indeed, investigating which of our wp (wlp) and invariant-based proof rules can be derived in Graded KAT is an appealing direction for future work.

7.3 Optimization

There exists a large amount of work on modelling and solving optimization problems. A prominent example is constrained optimization (e. g. linear programming [Horen 1985; Schrijver 1999]) for which standardized- and domain-specific languages exist [Lofberg 2004; Nethercote et al. 2007]. Modelling and solving optimization problems with weighted programming differs mainly in two aspects from these techniques. (1) The way *how* optimization problems are modelled and (2) *what* is modelled and for what purpose. Regarding aspect (1), techniques like integer linear programming or languages like MiniZinc model optimization problems in a constraint-based manner. With weighted programs, we describe these problems instead in an algorithmic fashion. As an intuition, constrained optimization vs. weighted programming could be considered analogous to logic programming vs. imperative programming.

Regarding aspect (2), constraint-based techniques often model *one particular* problem instance for which an optimal solution is *computed*. Weighted programs, on the other hand, provide a means to model and reason about every (out of possibly infinitely many) problem instance at once. This comes, however, at the cost of computability. The case study on the ski rental problem exemplifies this: We verify the competitive ratio of the optimal online algorithm for *every* trip duration of the ski rental problem. Automating this verification process is an appealing direction for future work.

More closely related is the work by Bistarelli et al. [1997], who generalize Constraint Logic Programming (CLP) by parameterizing CLP with a semiring S. Elements and operations of S take over the role Boolean constants and connectives. This allows to, e. g. solve optimization problems by finding atom instantiations of minimal cost.

8 CONCLUSION

We have studied weighted programming as a programming paradigm for specifying mathematical models. We developed a weakest (liberal) precondition-style verification framework for reasoning about both finite and infinite computations of weighted programs and demonstrated the efficacy of our framework on several case studies. Future work includes automated reasoning about weighted programs using, e.g., generalizations of k-induction [Batz et al. 2021; Sheeran et al. 2000] and weighted program synthesis [Alur et al. 2015; Manna and Waldinger 1980]. Further directions are weighted separation logics [Batz et al. 2019; Ishtiaq and O'Hearn 2001; Reynolds 2002] as well as to investigate "sampling" algorithms for weighted programs. For instance, what would be an analogon to MCMC sampling in a weighted setting?

ACKNOWLEDGMENTS

This work was supported by the ERC AdG Frappant (787914) (https://cordis.europa.eu/project/id/787914) and RTG 2236 UnRAVeL (https://gepris.dfg.de/gepris/projekt/282652900) funded by the German Research Foundation. Part of this work was carried out at Schloss Dagstuhl – Leibniz Center for Informatics. We thank Lena Verscht and Linpeng Zhang for the fruitful discussions at Schloss Dagstuhl.

REFERENCES

Samson Abramsky. 1994. *Handbook of Logic in Computer Science*. Vol. 3. Clarendon Press, Chapter Domain Theory. http://www.cs.bham.ac.uk/~axj/papers.html

Alejandro Aguirre and Shin-ya Katsumata. 2020. Weakest Preconditions in Fibrations. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 352)*. Elsevier, 5–27. https://doi.org/10.1016/j.entcs.2020.09.002

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

Benjamin Aminof, Orna Kupferman, and Robby Lampert. 2009. Reasoning About Online Algorithms with Weighted Automata. In *SODA*. SIAM, 835–844. https://doi.org/10.1137/1.9781611973068.91

Benjamin Aminof, Orna Kupferman, and Robby Lampert. 2010. Reasoning about Online algorithms with Weighted Automata. *ACM Trans. Algorithms* 6, 2 (2010), 28:1–28:36. https://doi.org/10.1145/1721837.1721844

Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus - A Systematic Introduction*. Springer. https://doi.org/10.1007/978-1-4612-1674-2

Marc Bagnol. 2014. *On the Resolution Semiring. (Sur le Semi-anneau de Résolution)*. Ph. D. Dissertation. Aix-Marseille University, Aix-en-Provence, France.

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

Esma Balkir, Daniel Gildea, and Shay B. Cohen. 2020. Tensors over Semirings for Latent-Variable Weighted Logic Programs. In *IWPT 2020*. Association for Computational Linguistics, 73–90. https://doi.org/10.18653/v1/2020.iwpt-1.8

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV (2) (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 524–549. https://doi.org/10.1007/978-3-030-81688-9_25

Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022. Weighted Programming. *CoRR* abs/2202.07577 (2022). https://doi.org/10.48550/arXiv.2202.07577

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. https://doi.org/10.1145/3290347

Vaishak Belle and Luc De Raedt. 2020. Semiring Programming: A Semantic Framework for Generalized Sum Product problems. *Int. J. Approx. Reason.* 126 (2020), 181–201. https://doi.org/10.1016/j.ijar.2020.08.001

Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. 1997. Semiring-based Constraint Logic Programming. In *IJCAI (1)*. Morgan Kaufmann, 352–357.

Allan Borodin and Ran El-Yaniv. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *ESOP (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Quentin Carbonneaux. 2018. *Modular and certified resource-bound analyses*. Ph. D. Dissertation. Yale University.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 467–478. https://doi.org/10.1145/2737924.2737955

Shay B. Cohen, Robert J. Simmons, and Noah A. Smith. 2008. Dynamic Programming Algorithms as Products of Weighted Logic Programs. In *ICLP (Lecture Notes in Computer Science, Vol. 5366)*. Springer, 114–129. https://doi.org/10.1007/978-3-540-89982-2_18

Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. 2019. Generalized Absorptive Polynomials and Provenance Semantics for Fixed-Point Logic. *CoRR* abs/1910.07910 (2019). https://doi.org/10.48550/arXiv.1910.07910

Edsger Wybe Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Stephen Dolan. 2013. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *ICFP*. ACM, 101–110. https://doi.org/10.1145/2500365.2500613

Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata* (1st ed.). Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-3-642-01492-5

Amos Fiat and Gerhard J. Woeginger (Eds.). 1998. *Online Algorithms, The State of the Art*. Lecture Notes in Computer Science, Vol. 1442. Springer. https://doi.org/10.1007/BFb0029561 the book grow out of a Dagstuhl Seminar, June 1996.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, and Tetsuya Sato. 2021. Graded Hoare Logic and its Categorical Semantics. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 234–263. https://doi.org/10.1007/978-3-030-72019-3_9

Martin Gavalec, Zuzana Nemcova, and Sergei Sergeev. 2015. Tropical linear algebra with the Łukasiewicz T-norm. *Fuzzy Sets Syst.* 276 (2015), 131–148. https://doi.org/10.1016/j.fss.2014.11.008

Brunella Gerla. 2003. Many-valued logic and semirings. *Neural Network World* 13 (01 2003).

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Leandro Gomes, Alexandre Madeira, and Luís Soares Barbosa. 2019. Generalising KAT to Verify Weighted Computations. *Sci. Ann. Comput. Sci.* 29, 2 (2019), 141–184. https://doi.org/10.7561/sacs.2019.2.141

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *FOSE*. ACM, 167–181. https://doi.org/10.1145/2593882.2593900

Wataru Hino, Hiroki Kobayashi, Ichiro Hasuo, and Bart Jacobs. 2016. Healthiness from Duality. In *LICS*. ACM, 682–691. https://doi.org/10.1145/2933575.2935319

C. A. R. Hoare. 1978. Some Properties of Predicate Transformers. *J. ACM* 25, 3 (1978), 461–480. https://doi.org/10.1007/978-1-4612-3228-5_6

Jeff Horen. 1985. Linear Programming. *Networks* 15, 2 (1985), 273–274. https://doi.org/10.1002/net.3230150211

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. ACM, 14–26. https://doi.org/10.1145/360204.375719

Claire Jones. 1990. *Probabilistic Non-Determinism*. Ph. D. Dissertation. University of Edinburgh, UK.

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

Klaus Keimel. 2015. Healthiness Conditions for Predicate Transformers. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 319)*. Elsevier, 255–270. https://doi.org/10.1016/j.entcs.2015.12.016

Dennis Komm. 2016. *An Introduction to Online Computation - Determinism, Randomization, Advice*. Springer. https://doi.org/10.1007/978-3-319-42749-2

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

Dexter Kozen. 1999. On Hoare Logic and Kleene Algebra with Tests. In *LICS*. IEEE Computer Society, 167–172. https://doi.org/10.1109/lics.1999.782610

Dexter Kozen. 2000. On Hoare Logic and Kleene Algebra with Tests. *ACM Trans. Comput. Log.* 1, 1 (2000), 60–76. https://doi.org/10.1145/343369.343378

Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. 2013. Weighted Relational Models of Typed Lambda-Calculi. In *LICS*. IEEE Computer Society, 301–310. https://doi.org/10.1109/lics.2013.36

Johan Lofberg. 2004. YALMIP: A Toolbox for Modeling and Optimization in MATLAB. *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)* (2004), 284–289. https://doi.org/10.1109/cacsd.2004.1393890

Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. https://doi.org/10.1145/357084.357090

Matilde Marcolli and Ryan Thorngren. 2011. Thermodynamic Semirings. *CoRR* abs/1108.2874 (2011). https://doi.org/10.48550/arXiv.1108.2874

Matilde Marcolli and Ryan Thorngren. 2014. Thermodynamic semirings. *Journal of Noncommutative Geometry* 8, 2 (2014), 337–392. https://doi.org/10.4171/jncg/159

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/b138392

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

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *CP (Lecture Notes in Computer Science, Vol. 4741)*. Springer, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38

Antonio Nola and Brunella Gerla. 2005. Algebras of Lukasiewicz's logic and their semiring reducts. *Contemp. Math* 377 (01 2005). https://doi.org/10.1090/conm/377/06988

Russel O'Conner. 2012. A Very General Method of Computing Shortest Paths. Personal blog entry. http://r6.ca/blog/20110808T035622Z.html

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 123–135. https://doi.org/10.1145/2628136.2628160

Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001

Marc Pouly. 2010. Semirings for Breakfast. https://marcpouly.ch/pdf/internal_100712.pdf Visited on 2022-03-21..

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/lics.2002.1029817

Alexander Schrijver. 1999. *Theory of Linear and Integer Programming*. Wiley.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD (Lecture Notes in Computer Science, Vol. 1954)*. Springer, 108–125. https://doi.org/doi.org/10.1007/3-540-40922-x_8

Richard Sproat, Mahsa Yarmohammadi, Izhak Shafran, and Brian Roark. 2014. Applications of Lexicographic Semirings to Problems in Speech and Language Processing. *Comput. Linguistics* 40, 4 (2014), 733–761. https://doi.org/10.1162/coli_a_00198

Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 103:1–103:26. https://doi.org/10.1145/3341707

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318. https://doi.org/10.1137/0606031

Günther J. Wirsching, Markus Huber, and Christian Kölbl. 2010. *The confidence-probability semiring*. Technical Report 2010-04. Fakultät für Angewandte Informatik.

Linpeng Zhang and Benjamin Lucien Kaminski. 2022a. Quantitative Strongest Post. *CoRR* abs/2202.06765 (2022). https://doi.org/10.48550/arXiv.2202.06765

Linpeng Zhang and Benjamin Lucien Kaminski. 2022b. Quantitative Strongest Post. *PACMPL* (2022). Issue OOPSLA. https://doi.org/10.1145/3527331 To appear.