Verified Learning Without Regret

From Algorithmic Game Theory to Distributed Systems with Mechanized Complexity Guarantees

Samuel Merten(✉), Alexander Bagnall, and Gordon Stewart

Ohio University, Athens, OH, USA
{sm137907,ab667712,gstewart}@ohio.edu

Abstract. Multiplicative Weights (MW) is a simple yet powerful algorithm for learning linear classifiers, for ensemble learning à la boosting, for approximately solving linear and semidefinite systems, for computing approximate solutions to multicommodity flow problems, and for online convex optimization, among other applications. Recent work in algorithmic game theory, which applies a computational perspective to the design and analysis of systems with mutually competitive actors, has shown that no-regret algorithms like MW naturally drive games toward approximate Coarse Correlated Equilibria (CCEs), and that for certain games, approximate CCEs have bounded cost with respect to the optimal states of such systems.

In this paper, we put such results to practice by building distributed systems such as routers and load balancers with performance and convergence guarantees mechanically verified in Coq. The main contributions on which our results rest are (1) the first mechanically verified implementation of Multiplicative Weights (specifically, we show that our MW is no regret) and (2) a language-based formulation, in the form of a DSL, of the class of games satisfying Roughgarden smoothness, a broad characterization of those games whose approximate CCEs have cost bounded with respect to optimal. Composing (1) with (2) within Coq yields a new strategy for building distributed systems with mechanically verified complexity guarantees on the time to convergence to near-optimal system configurations.

Keywords: Multiplicative weights · Algorithmic game theory
Smooth games · Interactive theorem proving · Coq

1 Introduction

The Multiplicative Weights algorithm (MW, [1,25]) solves the general problem of "combining expert advice", in which an agent repeatedly chooses which action, or "expert", to play against an adaptive environment. The agent, after playing an action, learns from the environment both the cost of that action and of other actions it could have played in that round. The environment, in turn, may adapt

© The Author(s) 2018
A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 561–588, 2018.
https://doi.org/10.1007/978-3-319-89884-1_20

in order to minimize environment costs. MW works by maintaining a weighted distribution over the action space, in which each action initially has equal weight, and by updating weights with a linear or exponential loss function to penalize poorly performing actions.

MW is a *no-regret* algorithm: its expected cost approaches that of the best fixed action the agent could have chosen in hindsight (i.e., external regret tends to zero) as time $t \to \infty$. Moreover, this simple algorithm performs remarkably well: in number of rounds logarithmic in the size of the action space, MW's expected regret can be bounded by a small constant ϵ (MW has bounded external regret). In [1], Arora, Hazan, and Kale showed that MW has wide-ranging connections to numerous problems in computer science, including optimization, linear and semidefinite programming, and machine learning (cf. boosting [14]).

Our work targets another important application of MW: the approximate solution of multi-agent games, especially as such games relate to the construction of distributed systems. It is well known (cf. [30, Chapter 4]) that no-regret algorithms such as MW converge, when played by multiple independent agents, to a large equilibrium class known as Coarse Correlated Equilibria (CCEs). CCEs may not be socially optimal, but for some games, such as Roughgarden's smooth games [35], the social cost of such equilibrium states can be bounded by a constant factor of the optimal cost of the game (the game has bounded Price of Anarchy, or POA). Therefore, to drive the social cost of a smooth game to near optimal, it suffices simply to let each agent play a no-regret algorithm such as MW.

Moreover, a number of distributed systems can be encoded as games, especially when the task being distributed is viewed as an optimization problem. Consider, for example, distributed balancing of network flows over a set of web servers, an application we return to in Sect. 3. Assuming the set of flows is fixed, and that the cost of (or latency incurred by) assigning a flow to a particular web server increases as a function of the number of flows already assigned to that server (the traffic), then the load balancing application is encodable as a game in which each flow is a "player" attempting to optimize its cost (latency). An optimal solution of this game minimizes the total latency across all flows. Since the game is Roughgarden smooth (assuming affine cost functions), the social cost of its CCEs as induced by letting each player independently run MW is bounded with respect to that of an optimal solution.

1.1 Contributions

In this paper, we put such results to work by building the first verified implementation of the MW algorithm – which we use to drive all games to approximate CCEs – and by defining a language-based characterization of a subclass of games called Roughgarden smooth games that have robust Price of Anarchy guarantees extending even to approximate CCEs. Combining our verified MW with smooth games, we construct distributed systems for applications such as routing and load balancing that have verified convergence and correctness guarantees.

Specifically, our main contributions are:

- a new architecture, as embodied in the Cage system (https://github.com/gstew5/cage), for the construction of distributed systems with verified complexity guarantees, by composition of verified Multiplicative Weights (MW) with robust Price of Anarchy bounds via Roughgarden smoothness;
- the first formally verified implementation of the MW algorithm;
- a language-based characterization of Roughgarden smooth games, in the form of a mechanized DSL for the construction of such games together with smoothness preservation theorems showing that each combinator in the language preserves smoothness;
- the application of the resulting system to distributed routing and load balancing.

By *verified*, we mean our MW implementation has mechanically checked convergence bounds and proof of correctness within an interactive theorem prover (specifically, Ssreflect [16], an extension of the Coq [5] system). By *convergence* and *correctness*, we mean that we prove both that MW produces the right answer (functional correctness with respect to a high-level functional specification), but also that it does so with external regret[1] bounded by a function of the number of iterations of the protocol (convergence). Convergence of MW in turn implies convergence to an approximate CCE. By composing this second convergence property with Roughgarden smoothness, we bound the social, or total, cost of the resulting system state with respect to the optimal.

As we've mentioned, MW has broad application across a number of subdisciplines of computer science, including linear programming, optimization, and machine learning. Although our focus in this paper is the use of MW to implement no-regret dynamics, a general strategy for computing the CCEs of multiagent games, our implementation of MW (Sect. 5.3) could be used to build, e.g., a verified LP solver or verified implementation of boosting as well.

Limitations. The approach we outline above does not apply to all distributed systems, nor even to all distributed systems encodable as games. In particular, in order to prove POA guarantees in our approach, the game encoding a particular distributed system must first be shown Roughgarden smooth, a condition which does not always apply (e.g., to network formation games [35, Section 2]). More positively, the Smooth Games DSL we present in Sects. 3 and 4 provides one method by which to explore the combinatorial nature of Roughgarden smoothness, as we demonstrate with some examples in Sect. 3.

Relationship to Prior Work. Some of the ideas we present in this paper previously appeared in summary form in a 3-page brief announcement at PODC 2017 [4]. The current paper significantly expands on the architecture of the Cage system, our verified implementation of Multiplicative Weights, the definition of the Smooth Games DSL, and the composition theorems of Sect. 6 proving that the pieces fit together to imply system-wide convergence and quality bounds.

[1] The expected (per-step) cost of the algorithm minus that of the best fixed action.

1.2 Organization

The following section provides background on games, algorithmic game theory, and smoothness. Section 3 presents an overview of the main components of the CAGE approach, via application to examples. Section 4 provides more detail on the combinators of our Smooth Games DSL. Section 5 presents our verified implementation of MW. Section 6 describes the composition theorems proving that multi-agent MW converges to near-optimal ϵ-CCEs. Sections 7 and 8 present related work and conclude.

2 Background

2.1 Games

Von Neumann, Morgenstern, and Nash [28,29] (in the US) and Bachelier, Borel, and Zermelo [3,8,43] (in Europe) were the first to study the mathematical theory of strategic interaction, modern game theory. Nash's famous result [27] showed that in all finite games, mixed-strategy equilibria (those in which players are allowed to randomize) always exist. Since the 1950s, game theory has had huge influence in numerous fields, especially economics.

In our context, a game is a tuple of a finite type A (the strategy space) and a cost function C_i mapping tuples of strategies of type $A_1 \times A_2 \times \ldots \times A_N$ to values of type $\mathbb{R}$, the cost to player i of state $(a_1, \ldots, a_i, \ldots, a_N)$. For readers interested in formalization-related aspects, Listing 1 provides additional details.

Listing 1: Games in Ssreflect-Coq

In SSREFLECT-COQ, an extension of the standard Coq system, a finite type A : finType pairs the type A with an enumerator enum : list A such that for all $a : A$, count a enum $= 1$ (every element is included exactly once). To define games, we use operational type classes [38], which facilitate parameter sharing:

```
Class game (A : finType) (N : nat) (ℝ : realFieldType)
  `(costClass : CostClass N ℝ A) : Type ≜ {}.
```

costClass declares the cost function C_i, and N is the number of players.

A state $s : A_1 \times A_2 \times \ldots \times A_N$ is a *Pure Nash Equilibrium (PNE)* when no player $i \in [1, N]$ has incentive to change its strategy: $\forall s'_i.\ C_i(s) \leq C_i(s'_i, s_{-i})$. Here s'_i is an arbitrary strategy. Strategy s_i is player i's move in state s. By s'_i, s_{-i} we denote the state in which player i's strategy is s'_i and all other players play s. In other words, no player can decrease its cost by unilateral deviation.

Pure-strategy Nash equilibria do not always exist. Mixed Nash Equilibria (MNE), which *do* exist in all finite games, permit players to randomize over the strategy space, by playing a distribution σ_i over A. The overall state is the product distribution over the player distributions. Every PNE is trivially an MNE, by letting players choose deterministic distributions σ_i.

Correlated Equilibria (CEs) generalize MNEs to situations in which players coordinate via a trusted third party. In what follows, we'll mostly be interested in a generalization of CEs, called *Coarse Correlated Equilibria (CCEs)*, and their approximate relaxations. Specifically, a distribution σ over A^N (Listing 2) is a CCE when $\forall i \forall s'_i.\ \mathbb{E}_{s \sim \sigma}[C_i(s)] \leq \mathbb{E}_{s \sim \sigma}[C_i(s'_i, s_{-i})]$. $\mathbb{E}_{s \sim \sigma}[C_i(s)]$ is the expected cost to player i in distribution σ. The CCE condition states that there is no s'_i that could decrease player i's expected cost. CCEs are essentially a relaxation of MNEs which do not require σ to be a product distribution (i.e., the players' strategies may be correlated). CEs are a subclass of CCEs in which $\mathbb{E}_{s \sim \sigma}[C_i(s'_i, s_{-i})]$ may be conditioned on s_i.

A distribution σ over states may only be *approximately* a CCE. Define as ϵ-approximate those CCEs σ for which $\forall i \forall s'.\ \mathbb{E}_{s \sim \sigma}[C_i(s)] \leq \mathbb{E}_{s \sim \sigma}[C_i(s'_i, s_{-i})] + \epsilon$. Moving to s'_i can decrease player i's expected cost, but only by at most ϵ.

Listing 2: Discrete Distributions in Ssreflect-Coq

Since our games A are finite, discrete distributions suffice to formalize MNEs, CEs, and CCEs. We model such distributions as finite functions (those with finite domain) from the strategy space A to $\mathbb{R}$:

```
Record dist (A : finType) : Type ≜
  mkDist { pmf :> {ffun A → ℝ}; dist_ax : dist_axiom pmf }.
```

Here {ffun $A \to \mathbb{R}$} is Ssreflect syntax for the type of finite functions from A to $\mathbb{R}$. The second projection of the record, dist_ax, asserts that pmf represents a valid distribution: pmf is positive and $\sum_{a:A}$ pmf $a = 1$.
The Coq predicate eCCE:

```
Definition eCCE (ϵ : ℝ) (σ : dist A^N) : Prop ≜
  ∀(i : [0..N − 1]) (s′ : A),
  expectedCost i σ ≤ (expectedUnilateralCost i σ s′) + ϵ.
```

states that distribution σ (over N-tuples of strategies A, one per player) is an ϵ-approximate CCE, or ϵ-CCE.

2.2 Algorithmic Game Theory

Equilibria are only useful if we're able to quantify, with respect to the game being analyzed:

1. How good equilibrium states are with respect to the optimal configurations of a game. By optimal, we usually mean states s^* that optimize the social cost: $\forall s.\ \sum_i C_i(s^*) \leq \sum_i C_i(s)$.
2. How "easy" (read computationally tractable) it is to drive competing players of the game toward an equilibrium state.

Algorithmic game theory and the related fields of mechanism design and distributed optimization provide excellent tools here.

Good Equilibria. The *Price of Anarchy*, or POA, of game (A, C) quantifies the cost of equilibrium states of (A, C) with respect to optimal configurations. Precisely, define POA as the ratio of the social cost of the worst equilibrium s to the social cost of an optimal state s^*. POA near 1 indicates high-quality equilibria: finding an equilibrium in such a game leads to overall social cost close to optimal. Prior work in algorithmic game theory has established nontrivial POA bounds for a number of game classes: on various classes of congestion and routing games [2,6,10], on facility location games [40], and others [11,32].

In the system of Sect. 3, we use the related concept of *Roughgarden smooth games* [35], or simply *smooth games*, which define a subclass of games with canonical POA proofs. To each smooth game are associated two constants, λ and μ. The precise definition of the smoothness condition is less relevant here than its consequences: if a cost-minimization game is (λ, μ)-smooth, then it has POA $\lambda/(1-\mu)$. Not all games are smooth, but for those that are, the POA bound above extends even to CCEs and their approximations, a particularly large (and therefore tractable) class of equilibria [35, Sects. 3 and 4].

Tractable Dynamics. Good equilibrium bounds are most useful when we know how quickly a particular game converges to equilibrium [7,9,12,13,17]. Certain classes of games, e.g. potential games [26], reach equilibria under a simple model of dynamics called best response. As we've mentioned, we use a different distributed learning algorithm in this work, variously called Multiplicative Weights (MW) [1] or sometimes Randomized Weighted Majority [25], which drives *all* games to CCEs, a larger class of equilibrium states than those achieved by potential games under best response.

3 Cage by Example

No-regret algorithms such as MW can be used to drive multi-agent systems toward the ϵ-CCEs of arbitrary games. Although the CCEs of general games may have high social cost, those of *smooth* games, as identified by Roughgarden [35], have robust Price of Anarchy (POA) bounds that extend even to ϵ-CCEs. Figure 1 depicts how these pieces fit together in the high-level architecture of our Cage system, which formalizes the results of Sect. 2 in Coq. Shaded boxes are program-related components while white boxes are proof related.

3.1 Overview

At the top, we have a domain-specific language in Coq (DSL, box 1) that generates games with automatically verified POA bounds. To execute such games, we have verified (also in Coq) an implementation of the Multiplicative Weights algorithm (MW, 2). Correctness of MW implies convergence bounds on the games it executes: $O((ln\ |A|)/\epsilon^2)$ iterations suffice to drive the game to an ϵ-CCE (here, $|A|$ is the size of the action space, or game type, A).

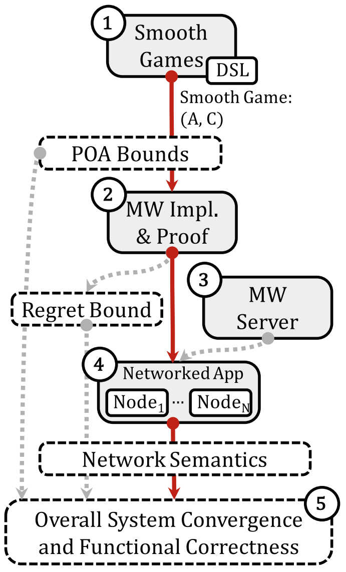


Fig. 1. System architecture

We compose N instances of multiplicative weights (4), one per agent, with a server (3) that facilitates communication, implemented in OCaml and modeled by an operational semantics in Coq. To actually execute games, we use Coq's code extraction mechanism to generate OCaml code that runs clients against the server, using an unverified OCaml shim to send and receive messages. We prove performance guarantees in Coq from POA bounds on the game and from the regret bound on MW.

3.2 Smooth Games DSL

The combinators exposed by the Smooth Games DSL operate over game types A, cost functions C, and smoothness parameters λ and μ. Basic combinators in this language include (i) Resource and (ii) Unit games, the first for coordinating access to shared resources under congestion and the second with fixed cost 0. Combinators that take other games as arguments include:

– the bias combinator $\mathsf{Bias}(A, b)$, which adds the fixed value b to the cost function associated with game A;
– the scalar combinator $\mathsf{Scalar}(A, m)$, which multiplies the output of the cost function C associated with game A by a fixed value m;
– the product combinator $A \times B$, corresponding to the parallel composition of two games A and B with cost equal to the sum of the costs in the two games;
– the subtype game $\{x : A,\ P(x)\}$, which constructs a new game over the dependent sum type $\Sigma x : A.P(x)$ (values x satisfying the predicate P);

– the singleton game $\mathsf{Singleton}(A)$, which has cost 1 if if player i "uses" the underlying resource ($\mathbb{B}_{\mathsf{Resource}}(f\ \ i) = \mathsf{true}$), and 0 otherwise. The function $\mathbb{B}_{-}(-)$ generalizes the notion of resource usage beyond the primitive Resource game. For example, $\mathbb{B}_{\mathsf{Scalar}(A,m)}(x) = \mathbb{B}_A(x)$: usage in a game built from the scalar combinator reduces to usage in the underlying game.

3.3 Example: Distributed Routing

We illustrate the Smooth Games DSL with an example: distributed routing over networks with affine latency functions (Fig. 2). This game is known to have POA 5/2 [35].

In a simple version of the game, N routing agents each choose a path from a global source vertex s to a global sink vertex t. Latency over edge e, modeled by an affine cost function $c_e(x) = a_e x + b_e$, scales in the amount of traffic x over that edge. An optimal solution minimizes the total cost to all agents.

We model each link in the network as a Resource game, which in its most basic form is defined by the following inductive datatype:

```
Inductive Resource : Type ≜
  | RYes : Resource
  | RNo : Resource.
```

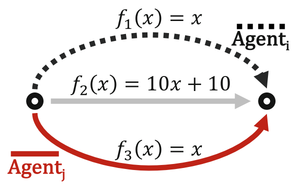


Fig. 2. Routing game with affine edge costs

RYes indicates the agent chose to use the resource (a particular edge) and RNo otherwise. The cost function for Resource is defined by:

```
Definition ResourceCostFun (i : [0..N − 1]) (s : [0..N − 1] →fin Resource) : ℝ ≜
  if s_i is RYes then traffic s else 0.
```

in which s is a map from agent labels to resource strategies and traffic s is the total number of agents that chose to use resource s. An agent pays traffic s if it uses the resource, otherwise 0. We implement Resource as a distinct inductive type, even though it's isomorphic to bool, to ensure that types in the Smooth Games DSL have unique game instances. To give each resource the more interesting cost function $c_e(x) = a_e x + b_e$, we compose Resource with a second combinator, $\mathsf{Affine}(a_e, b_e, \mathsf{Resource})$, which has cost 0 if an agent does not use the resource, and cost $a_e{*}(\mathsf{traffic}\ s) + b_e$ otherwise. This combinator preserves (λ, μ)-smoothness assuming $\lambda + \mu \geq 1$, a side condition which holds for Resource games.

We encode m affine resources by applying Affine to Resource m times, then folding under product:

```
T ≜ Affine(a_1,b_1,Resource)
  × Affine(a_2,b_2,Resource)
  × ...
  × Affine(a_m,b_m,Resource)
```

The associated cost function is the sum of the individual resource cost functions.

Values of type T may assign RYes to a subset of resources that doesn't correspond to a valid path in a graph $G = (V, E)$. To prevent this behavior, we apply to T the subtype combinator Σ, specialized to a predicate isValidPath(G, s, t) enforcing that strategies $(r_1, r_2, \ldots, r_{|E|})$ correspond to valid paths from s to t: T' $\triangleq$ $\Sigma_{\mathsf{isValidPath}(G,s,t)}$(T). The game T' is (5/3, 1/3)-smooth, just like the underlying Resource game, which implies POA of $(5/3)/(1 - 1/3) = 5/2$.

3.4 Example: Load Balancing

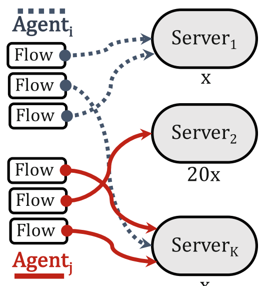


Fig. 3. Load balancing game

As a second example, consider the load balancing game depicted in Fig. 3, in which a number of network flows are distributed over several servers with affine cost functions. In general, N load balancing agents are responsible for distributing M flows over K servers. The cost of allocating a flow to a server is modeled by an affine cost function which scales in the total load (number of flows) on that server. Like routing, the load balancing game has POA 5/2. This is no coincidence; both are special cases of "finite congestion games", a class of games which have POA 5/2 when costs are linear [10]. The connection between them can be seen more concretely by observing that they are built up from the same primitive Resource game.

We model the system as an NM-player K-resource game in which each player corresponds to a single network flow. Each load balancing agent poses as multiple players (MW instances) in the game, one per flow, and composes the actions chosen by these players to form its overall strategy. The result of running the game is an approximate CCE with respect to the distribution of flows over servers.

Each server is defined as a Resource with an affine cost function, using the same data type and cost function as in the routing example. Instead of isValidPath, we use a new predicate exactlyOne to ensure that each network flow is assigned to exactly one server.

4 Smooth Games

Roughgarden smoothness [35] characterizes a subclass of games with canonical Price of Anarchy (POA) proofs. In [35], Roughgarden showed that smooth games have canonical POA bounds not only with respect to pure Nash equilibria but also with respect to mixed Nash equilibria, correlated equilibra, CCEs, and their approximate relaxations. In the context of Cage, we use smoothness to bound the social cost of games executed by multiple clients each running MW. We show how the technical pieces fit together, in the form of bounds on an operational semantics of the entire Cage system, in Sect. 6. This section introduces the technical definition of smoothness and the language of combinators,

Syntax

$$\begin{aligned}&\textit{Scalars } m, b; \quad \textit{Predicates } P\\ &\textit{Game types } A, B ::= \mathsf{Resource} \mid \mathsf{Unit} \mid \mathsf{Bias}(A, b) \mid \mathsf{Scalar}(A, m)\\ &\qquad\qquad \mid A \times B \mid \{x : A,\ P(x)\} \mid \mathsf{Singleton}(A)\end{aligned}$$

Judgment $\boxed{\vdash_{(\lambda,\mu)} (A, C)}$ read "Game (A, C) is (λ, μ)-smooth."

$$\frac{}{\vdash_{(\frac{5}{3},\frac{1}{3})} (\mathsf{Resource}, \mathsf{ResourceCostFun})}\ \mathsf{ResourceSmooth}$$

$$\frac{}{\vdash_{(1,0)} (\mathsf{Unit}, \mathsf{fun}\ i\ f.\ 0)}\ \mathsf{UnitSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C)}{\vdash_{(1,0)} (\mathsf{Singleton}(A), \mathsf{fun}\ i\ f.\ \mathsf{if}\ \mathbb{B}_A(f\ i)\ \mathsf{then}\ 1\ \mathsf{else}\ 0)}\ \mathsf{SingletonSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C)}{\vdash_{(\lambda,\mu)} (\{x : A,\ P(x)\}, \mathsf{fun}\ i\ f.\ C_i\ (\mathsf{fun}\ j.\ (f\ j).1))}\ \mathsf{SigmaSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C) \quad 1 \le \lambda + \mu \quad 0 \le b}{\vdash_{(\lambda,\mu)} (\mathsf{Bias}(A, b), \mathsf{fun}\ i\ f.\ C_i\ f + b)}\ \mathsf{BiasSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C) \quad 0 \le m}{\vdash_{(\lambda,\mu)} (\mathsf{Scalar}(A, m), \mathsf{fun}\ i\ f.\ m * C_i\ f)}\ \mathsf{ScalarSmooth}$$

$$\frac{\vdash_{(\lambda_A,\mu_A)} (A, C^A) \qquad \vdash_{(\lambda_B,\mu_B)} (B, C^B)}{\vdash_{(\max(\lambda_A,\lambda_B),\max(\mu_A,\mu_B))} (A \times B, \mathsf{fun}\ i\ f.\ C_i^A\ f + C_i^B\ f)}\ \mathsf{ProductSmooth}$$

Fig. 4. Smooth games DSL

or Smooth Games DSL of Sect. 3, that we use to build games that are smooth by construction.

Definition 1 (Smoothness). *A game (A, C) is (λ, μ)-smooth if for any two states $s, s^* : A^N$, the following inequality holds:*

$$\sum_{i=1}^{k} C_i(s_i^*, s_{-i}) \le \lambda \cdot C(s^*) + \mu \cdot C(s).$$

Here, $C_i(s_i^*, s_{-i})$ denotes the individual cost to player i in the mixed state where all other players follow their strategies from s, while player i follows the corresponding strategy from s^*. Smooth games bound the individual cost of players' unilateral deviations from state s to s^* by the weighted social costs of s and s^*. In essence, when λ and μ are small, the effect of any single player's deviation from a given state has minimal effect.

The smoothness inequality leads to natural proofs of POA for a variety of equilibrium classes. As an example, consider the following bound on the expected cost of ϵ-CCEs of (λ, μ)-smooth games:

Lemma smooth_eCCE $(d$: dist (state N T)) $(s'$: state N T) $(\epsilon : \mathbb{R})$:
eCCE ϵ $d \rightarrow$ optimal $s' \rightarrow$
ExpectedCost $d \leq \lambda*$(Cost s') + $\mu*$(ExpectedCost d) + $N*\epsilon$.

ExpectedCost d is the sum for all players i of the expected cost to player i of distribution d. N is the number of players in the game.

The smooth_eCCE bound implies the following Price of Anarchy bound on the expected cost, summed across all players, of distribution d:

Lemma smooth_POA ϵ $(d$: dist (state N T)) s' :
eCCE ϵ $d \rightarrow$ optimal $s' \rightarrow$
ExpectedCost $d \leq \lambda/(1-\mu)*$(Cost s') + $(N*\epsilon)/(1-\mu)$.

If d is an ϵ-CCE, then its cost is no more than $\lambda/(1$ - $\mu)$ times the optimal cost of s', plus an additional term that scales in the number of players N. For example, for concrete values $\lambda = 5/3$, $\mu = 1/3$, $\epsilon = 0.0375$, and $N = 5$, we get multiplicative approximation factor $\lambda/(1-\mu) = 5/2$ and additive factor 0.28. A value of $\epsilon = 0.0375$ is reasonable; as Sect. 5 will show, it takes fewer than 20,000 iterations of the Multiplicative Weights algorithm, in a game with strategy space of size 1000, to produce $\epsilon \leq 0.0375$.

4.1 Combinators

Figure 4 lists the syntax and combinators of the Smooth Games DSL we used in Sect. 3 to build smooth routing and load balancing games.

The smoothness proof accompanying the judgment of Resource games is the least intuitive, and provides some insight into the behavior of smooth games. The structure of our proof borrows from a stronger result given by Roughgarden [35]: smoothness for resource games with affine cost functions and multiple resources. The key step is the following inequality first noted by Christodoulou and Koutsoupias [10]:

$$y(z+1) \leq \frac{5}{3}y^2 + \frac{1}{3}z^2$$

for non-negative integers y and z. We derive $(\frac{5}{3}, \frac{1}{3})$-smoothness of Resource games from the following inequalities:

$$\sum_{i=0}^{N-1} C_i(s_i^*, s_{-i}) \leq (\text{traffic } s^*) \cdot (\text{traffic } s + 1) \quad (1)$$

$$(\text{traffic } s^*) \cdot (\text{traffic } s + 1) \leq \frac{5}{3} \cdot (\text{traffic } s^*)^2 + \frac{1}{3} \cdot (\text{traffic } s)^2 \quad (2)$$

$$(\text{traffic } s^*) \cdot (\text{traffic } s + 1) \leq \frac{5}{3} \cdot C(s^*) + \frac{1}{3} \cdot C(s) \quad (3)$$

$$\sum_{i=0}^{N-1} C_i(s_i^*, s_{-i}) \leq \frac{5}{3} \cdot C(s^*) + \frac{1}{3}\mu \cdot C(s) \quad (4)$$

The inequality in step 1 is due to the fact that the cost per player in state s^* is at most $\mathsf{traffic}\ s + 1$, and there are exactly $\mathsf{traffic}\ s^*$ players incurring such cost. I.e., $(\mathsf{traffic}\ s^*) \cdot (\mathsf{traffic}\ s + 1)$ is the number of nonzero terms times the upper bound on each term. The substitution in step 3 comes from the fact that in any state s, $C(s) = (\mathsf{traffic}\ s)^2$; each of the m players using the resource incur cost m.

The proofs of smoothness for other combinators are straightforward. For example, since Unit games always have cost 0, all values of λ and μ satisfy the smoothness inequality: $0 \leq \lambda \cdot 0 + \mu \cdot 0$. We restrict the range of the cost function in SingletonSmooth games to $\{0, 1\}$ by applying the function $\mathbb{B}_A(\cdot)$, which generalizes the notion of "using a resource" to all the game types of Fig. 4. Smoothness of the Singleton game follows by case analysis on the results of $\mathbb{B}_A(\cdot)$ in the states s and s^* of the smoothness inequality. The games produced by the SigmaSmooth combinator have costs equal to those of the underlying games but restrict the domain to those states satisfying a predicate P. Since smoothness of the underlying bound holds for all states in A, the same bound holds of the restricted domain of states $a \in A$ drawn from P. Smoothness of product games relies on the fact that smoothness still holds if λ and μ are replaced with larger values. Thus, each of the argument games to ProductSmooth is $(\mathsf{max}(\lambda_A, \lambda_B), \mathsf{max}(\mu_A, \mu_B))$-smooth. The overall product game, which sums the costs of its argument games, is $(\mathsf{max}(\lambda_A, \lambda_B), \mathsf{max}(\mu_A, \mu_B))$-smooth as well.

It's possible to derive combinators from those defined in Fig. 4. For example, define as $\mathsf{Affine}(m, b, A)$ the game with cost function $mx + b$. We implement this game as $\{p : \mathsf{Scalar}(m, A) \times \mathsf{Scalar}(b, \mathsf{Singleton}(A)),\ p.1 = p.2\}$, or the subset of product games over the scalar game $\mathsf{Scalar}(m, A)$ and the $\{0, 1\}$ scalar game over b such that the first and second projections of each strategy p are equal.

5 Multiplicative Weights (MW)

At the heart of the CAGE architecture of Sect. 3 lies our verified implementation of the Multiplicative Weights algorithm. In this section, we present the details of the algorithm and sketch its convergence proof. Section 5.3 presents our verified MW implementation and mechanized proof of convergence.

```
For all $a \in A$, client initializes $w_1(a) = 1$.
For time $t \in [1 \dots T]$ :
    Client                                                     | Environment
    Let $\Gamma_t \triangleq \sum_{a \in A} w_t(a)$.           |
    Play strategy $p_t(a) = w_t(a)/\Gamma_t$.                  |
                                                               | Choose cost vector $c_t$.
    Update weights $w_{t+1}(a) \triangleq w_t(a) * (1 - \eta * c_t(a))$ |
```

Fig. 5. Multiplicative Weights (MW)

5.1 The Algorithm

The MW algorithm (Fig. 5) pits a client, or agent, against an adaptive environment. The agent maintains a weight distribution w over the action space, initialized to give each action equal weight. At each time step $t \in [1 \dots T]$, the agent commits to the distribution $w_t / \sum_{a \in A} w_t(a)$, communicating this mixed strategy to the environment. After receiving a cost vector c_t from the environment, the agent updates its weights w_{t+1} to penalize high-cost actions, at a rate determined by a learning constant $\eta \in (0, 1/2]$. High η close to 1/2 leads to higher penalties, and thus relatively less exploration of the action space.

The environment is typically adaptive, and may be implemented by a number of other agents also running instances of MW. The algorithm proceeds for a fixed number of epochs T, or until some bound on expected external regret (expected cost minus the cost of the best fixed action) is achieved. In what follows, we always assume that costs lie in the range $[-1, 1]$. Costs in an arbitrary but bounded range are also possible (with a concomitant relaxation of the algorithm's regret bounds), as are variations of MW to solve payoff maximization instead of cost minimization.

5.2 MW Is No Regret

The MW algorithm converges reasonably quickly: To achieve expected regret at most ϵ, it's sufficient to run the algorithm $O((ln\ |A|)/\epsilon^2)$ iterations, where $|A|$ is the size of the action space [36, Chapter 17]. Regret can be driven arbitrarily small as the number of iterations approaches infinity. Bounded regret suffices to prove convergence to an approximate CCE, as [36] also shows.

In this section, we present a high-level sketch of the proof that MW is no regret. We follow [36, Chapter 17], which has additional details. At the level of the mathematics, our formal proof makes no significant departures from Roughgarden.

Definition 2 (Per-Step External Regret). *Let a^* be the best fixed action in hindsight (i.e., the action with minimum cost given the cost vectors received from the environment) and let $OPT \triangleq \sum_{t=1}^{T} c_t(a^*)$. The expected per-step external regret of MW is*

$$\left(\sum_{t=1}^{T} \zeta_t - OPT \right) \ / \ T.$$

The summed term defines the cumulative expected cost of the algorithm for time $t \in [1 \dots T]$, where by ζ_t we denote the expected cost at time t:

$$\zeta_t = \sum_{a \in A} p_t(a) \cdot c_t(a) = \sum_{a \in A} \frac{w_t(a)}{\Gamma_t} \cdot c_t(a)$$

To get per-step expected regret, we subtract the cumulative cost of a^* and divide by the number of time steps T.

Theorem 1 (MW Has Bounded Regret). *The algorithm of Fig. 5 has expected per-step external regret at most $\eta + ln\ |A|\ /\ \eta T$.*

Proof Sketch. The proof of Theorem 1 uses a potential-function argument, with potential Φ_t equal the sum of the weights $\Gamma_t = \sum_{a \in A} w_t(a)$ at time t. It proceeds by relating the cumulative expected cost $\sum_t \zeta_t$ of the algorithm to OPT, the cost of the best fixed action, through the intermediate quantity Γ_{T+1}.

The proof additionally relies on the following two facts derived from the Taylor expansion $ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots$:

$$\begin{aligned} ln(1-x) &\leq -x, & x &< 1 \\ -x - x^2 &\leq ln(1-x), & x &\leq 1/2 \end{aligned}$$

□

By letting $\eta = \sqrt{ln\ |A|\ /\ T}$ (cf. [36, Chapter 17]), it's possible to restate the regret bound of Theorem 1 to the following arguably nicer bound:

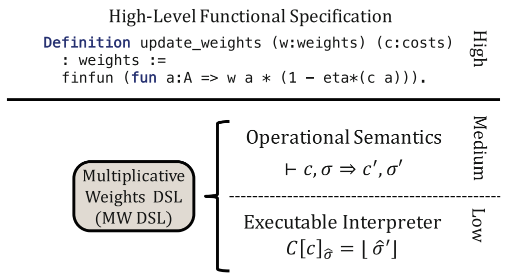
Corollary 1 (MW Is No Regret)

$$\left(\sum_{t=1}^{T} \zeta_t - OPT \right) /\ T \leq 2\sqrt{ln\ |A|\ /\ T}$$

Here, the number of iterations T must be large enough to ensure that $\eta = \sqrt{ln\ |A|\ /\ T} \leq 1/2$, thus ensuring that $\eta \in (0, 1/2]$.

5.3 MW Architecture

Our implementation and proof of MW (Fig. 6) were designed to be extensible. At a high level, the proof structure follows the program refinement methodology, in which a high-level mathematical but inefficient specification of MW (High-Level Functional Specification) is gradually made more efficient by a series of refinements to various features of the program (for example, by replacing an inefficient implementation of a key-value map with a more efficient balanced binary tree).


High-Level Functional Specification
High
Operational Semantics
$\vdash c, \sigma \Rightarrow c', \sigma'$
Medium
Multiplicative Weights DSL (MW DSL)
Executable Interpreter
$C[c]_{\hat{\sigma}} = \lfloor \hat{\sigma}' \rfloor$
Low

```
Definition update_weights (w:weights) (c:costs)
  : weights :=
  finfun (fun a:A => w a * (1 - eta*(c a))).
```

Fig. 6. MW architecture

For each such refinement, we prove that every behavior of the lower-level program is a possible behavior of the higher-level program it refines. Thus specifications proved for all behaviors of the high-level program also apply to each behavior at the low level. By behavior here, we mean the trace of action distributions output by MW as it interacts with, and receives cost vectors from, the environment.

We factor the lower implementation layers (Medium and Low) into an interpreter and operational semantics over a domain-specific language specialized to MW-style algorithms (MW DSL). The DSL defines commands for maintaining and updating weights tables as well as commands for interacting with the environment. We prove, for any DSL program c, that the interpretation of that program refines its behavior with respect to the small-step operational semantics (Medium). Our overall proof specializes this general refinement to an implementation of MW as a command in the DSL, in order to relate that command's interpreted behavior to the high-level functional specification.

5.4 MW DSL

The syntax and semantics of the MW DSL are given in Fig. 7. The small-step operational semantics ($\vdash c, \sigma \Rightarrow c', \sigma'$) is parameterized by an environment oracle that defines functions for sending action distributions to the environment (oracle_send) and for receiving the resulting cost vectors (oracle_recv). The oracle will in general be implemented by other clients also running MW (Sect. 6) but is left abstract here to facilitate abstraction and reuse. The oracle is stateful (the type T, of oracle states, may be updated both by oracle_send and oracle_recv).

Most of the operational semantics rules are straightforward. In the MW-Step-Weights rule for updating the state's weights table, we make use of an auxiliary expression evaluation function $E_{-}[-]$ (standard and therefore not shown in Fig. 7). The only other interesting rules are those for send and recv, which call oracle_send and oracle_recv respectively. In the relation oracle_recv, the first two arguments are treated as inputs (the input oracle state of type T and the channel) while the second two are treated as outputs (the cost vector of type $A \rightarrow \mathbb{Q}$ and the output oracle state). In the relation oracle_send, the first three arguments are inputs while only the last (the output oracle state) is an output.

Multiplicative Weights. As an example of an MW DSL program, consider our implementation (Listing 1.1) of the high-level MW of Fig. 5. To the right of each program line, we give comments describing the effect of each command. The program is itself divided into three functions:mult_weights_init, which initializes the weights table to assign weight 1 to each action a in the action space A; mult_weights_body, which defines the body of the main loop of MW; and mult_weights, which simply composes mult_weights_init with mult_weights_body.

Listing 1.1. MW DSL Implementation of Multiplicative Weights

```
Definition mult_weights_init (A : Type) ≜
  update (λ a : A ⇒ 1); (∗ For all a ∈ A, initialize w₁(a) = 1. ∗)
  send. (∗ Commit to the uniform distribution over actions. ∗)

Definition mult_weights_body (A : Type) ≜
  recv; (∗ Block until agent receives cost vector cₜ from environment. ∗)
  update (λ a : A ⇒ weight a ∗ (1 − η ∗ cost a)); (∗ Update weights. ∗)
  send. (∗ Commit to distribution wₜ/Γₜ. ∗)

Definition mult_weights (A : Type) (n : N.t) ≜
  mult_weights_init A; (∗ Initialize weights and commit to initial mixed strategy. ∗)
  iter n (mult_weights_body A). (∗ Do n iterations of the MW main loop. ∗)
```

The MW DSL contains commands and expressions that are specialized to MW-style applications. Consider the function mult_weights_body (line 5). It first receives a cost vector from the environment using the specialized recv command. At the level of the MW DSL, recv is somewhat abstract. The program does not specify, e.g., which network socket to use. Implementation details such as these are resolved by the MW interpreter, which we discuss below in Sect. 5.5.

After recv, mult_weights_body implements an update to its weights table as defined by the command: update $(\lambda a : A \Rightarrow \mathsf{weight}\ a * (1 - \eta * \mathsf{cost}\ a))$. As an argument to the update, we embed a function from actions $a \in A$ to expressions that defines how the weight of each action a should change at this step (time $t+1$). The expressions weight a and cost a refer to the weight and cost, respectively, of action a at time t. The anonymous function term is defined in Ssreflect-Coq, the metalanguage in which the MW DSL is defined.

5.5 Interpreter

To run MW DSL programs, we wrote an executable interpreter in Coq with type:

interp $(c : \mathsf{com}\ A)$ $(s : \mathsf{cstate})$: option cstate.

The type cstate defines the state of the interpreter after each step, and in general maps quite closely to the type of states σ used in the MW DSL operational semantics. It is given by the record:

Syntax

$$\begin{aligned}
\textit{Binary operators } \oplus &::= + \mid - \mid * \\
\textit{Expressions } e &::= d \mid -e \mid \mathsf{weight}\ a \mid \mathsf{cost}\ a \mid \eta \mid e_1 \oplus e_2 \\
\textit{Commands } c &::= \mathsf{skip} \mid \mathsf{update}\ (\lambda a : A \Rightarrow e) \mid c_1; c_2 \mid \mathsf{iter}\ n\ c \mid \mathsf{recv} \mid \mathsf{send}
\end{aligned}$$

Environment Oracle

$$\begin{aligned}
\mathsf{oracle\_recv} &: T \to \mathsf{oracle\_chanty} \to (A \to \mathbb{Q}) \to T \to \mathsf{Prop} \\
\mathsf{oracle\_send} &: T \to \mathsf{dist}\ A \to \mathsf{oracle\_chanty} \to T \to \mathsf{Prop}
\end{aligned}$$

States $\sigma \triangleq$

$\{\ \mathsf{SCosts} : A \to \mathbb{Q};\ \mathsf{SCostsOk} : \forall a.\ \lvert\mathsf{SCosts}\ a\rvert \le 1$	Current cost vector
$;\ \mathsf{SPrevCosts} : \mathsf{seq}\ \{c : A \to \mathbb{Q} \mid \forall a.\ \lvert c\ a\rvert \le 1\}$	Previous cost vectors
$;\ \mathsf{SWeights} : A \to \mathbb{Q}$	Weights table
$;\ \mathsf{SWeightsOk} : \forall a.\ 0 < \mathsf{SWeights}\ a$	
$;\ \mathsf{SEta} : \mathbb{Q};\ \mathsf{SEtaOk} : 0 < \mathsf{SEta} \le 1/2$	The η parameter
$;\ \mathsf{SOutputs} : \mathsf{seq}\ (\mathsf{dist}\ A)$	Committed distributions
$;\ \mathsf{SChan} : \mathsf{oracle\_chanty}$	I/O channel
$;\ \mathsf{SOracleSt} : T\ \}.$	Environment/oracle state

Operational Semantics

$$\frac{\sigma' = \sigma\{\mathsf{SWeights} \triangleq \lambda a : A \Rightarrow E_\sigma[e[x \leftarrow a]]\}}{\vdash \mathsf{update}\ (\lambda x : A \Rightarrow e), \sigma \Rightarrow \mathsf{skip}, \sigma'}\ \textsc{MW-Step-Weights}$$

$$\frac{}{\vdash \mathsf{skip}; c_2, \sigma \Rightarrow c_2, \sigma} \qquad \frac{\vdash c_1, \sigma \Rightarrow c_1', \sigma'}{\vdash c_1; c_2, \sigma \Rightarrow c_1'; c_2, \sigma'}$$

$$\frac{}{\vdash \mathsf{iter}\ 1\ c, \sigma \Rightarrow c, \sigma} \qquad \frac{1 < n}{\vdash \mathsf{iter}\ n\ c, \sigma \Rightarrow c; \mathsf{iter}\ (n-1)\ c, \sigma}$$

$$\frac{\mathsf{oracle\_recv}\ (\mathsf{SOracleSt}\ \sigma)\ (\mathsf{SChan}\ \sigma)\ c\ t}{\vdash \mathsf{recv}, \sigma \Rightarrow \mathsf{skip}, \sigma\{\mathsf{SCosts} \triangleq c;\ \mathsf{SPrevCosts} \triangleq \mathsf{SCosts}\ \sigma :: \mathsf{SPrevCosts}\ \sigma;\ \mathsf{SOracleSt} \triangleq t\}}$$

$$\frac{\mathsf{oracle\_send}\ (\mathsf{SOracleSt}\ \sigma)\ d\ ch\ t}{\vdash \mathsf{send}, \sigma \Rightarrow \mathsf{skip}, \sigma\{\mathsf{SOutputs} \triangleq d :: \mathsf{SOutputs}\ \sigma;\ \mathsf{SChan} \triangleq ch;\ \mathsf{SOracleSt} \triangleq t\}}$$

Fig. 7. MW DSL syntax and operational semantics, parameterized by an environment oracle defining the type T of environment states and the functions oracle_recv and oracle_send for interacting with the environment. The type A is that of states in the underlying game.

Record cstate : Type $\triangleq$	
{ SCosts : M.t $\mathbb{Q}$	Current cost vector
; SPrevCosts : list (M.t $\mathbb{Q}$)	Previous cost vectors
; SWeights : M.t $\mathbb{Q}$	Weights table
; SEta : $\mathbb{Q}$	The η parameter
; SOutputs : list ($A \to \mathbb{Q}$)	Committed distributions
; SChan : oracle_chanty	I/O channel
; SOracleSt : T }.	Environment/oracle state

At the level of cstates, we use efficient purely functional data structures such as AVL trees. For example, the type M.t $\mathbb{Q}$ denotes an AVL-tree map from actions A to rational numbers $\mathbb{Q}$. In the small-step semantics state, by contrast, we model the weights table not as a balanced binary tree but as a Ssreflect-Coq finite function, of type $\{\mathsf{ffun}\ A \to \mathbb{Q}\}$, which directly maps actions of type A to values of type $\mathbb{Q}$.

To speed up computation on rationals, we use a dyadic representation $q = \frac{n}{2^d}$, which facilitates fast multiplication. We do exact arithmetic on dyadic $\mathbb{Q}$ instead of floating point arithmetic to avoid floating-point precision error. Verification of floating-point error bounds is an interesting but orthogonal problem (cf. [31,34]).

The field SOutputs in the cstate record, a list of functions mapping actions $a \in A$ to their probabilities, stores the history of weights distributions generated by the interpreter as send commands are executed. To implement commands such as send and recv, we parameterize our MW interpreter by an environment oracle, just as we did the operational semantics. The operations implemented by the interpreter environment oracle are functional versions of the operational semantics oracle_send and oracle_recv:

$$\mathsf{oracle\_send}' : \forall A{:}\mathsf{Type},\ T \to A \to \mathsf{oracle\_chanty} * T$$
$$\mathsf{oracle\_recv}' : \forall A{:}\mathsf{Type},\ T \to \mathsf{oracle\_chanty} \to \mathsf{list}\ (A{*}\mathbb{Q}) * T$$

The oracle state type T is provided by the implementation of the oracle, as in the operational semantics. The command oracle_send′ takes a state of type T and a value of type A as arguments and returns a pair of a channel of type oracle_chanty (on which to listen for a response from the environment) and a new oracle state of type T. The command oracle_recv′ takes as arguments the oracle state and channel and returns a list of (a, q) pairs, representing a cost vector over actions, along with the new oracle state.

5.6 Proof

The top-level theorem proved of our high-level functional specification of MW is:

```
Theorem perstep_weights_noregret :
 (expCostsR − OPTR)/T ≤ η + ln size_A / (η ∗T).
```

The expression expCostsR is the cumulative expected cost of MW on a sequence of cost vectors, or the sum, for each time t, of the expected cost of the MW algorithm at time t. OPTR is the cumulative cost over T rounds of the best fixed action. The number η (a dyadic rational required to lie in range $(0, 1/2]$) is the learning parameter provided to MW and ln size_A is the natural log of the size of the action space A. T is the number of time steps. In contrast to the interpreter and semantics of Sect. 5.3 (where we do exact arithmetic on dyadics), for reasoning and specification at the level of the proof we use Coq's real number library and real-valued functions such as square root and log.

By choosing η to equal $\sqrt{ln\ \mathsf{size\_A}\ /\ T}$, Corollary 1 showed that it's possible to restate the right-hand side of the inequality in perstep_weights_noregret to

$2 *$ sqrt (ln size_A / T), thus giving an arguably nicer bound. Since in our implementation of MW we require that η be a dyadic rational, we cannot implement $\eta = \sqrt{ln \text{ size\_A} / T}$ directly (ln size_A is irrational). We do, however, prove the following tight approximation for all values of η approaching $\sqrt{ln \text{ size\_A} / T}$:

```
Lemma perstep_weights_noregret' :
  ∀r : ℝ. r ≠ −1 → η = (1+r)∗(sqrt (ln size_A / T)) →
  (expCostsR − OPTR)/T ≤
  (1+r)∗(sqrt (ln size_A / T)) + (sqrt (ln size_A / T))/(1+r).
```

In the statement of this lemma, the r term quantifies the error (how far η is from its optimal value sqrt (ln size_A / T). We require that $r \neq -1$ to ensure that division by $1 + r$ is well-defined. The resulting bound approaches $2 *$ sqrt (ln size_A / T) as r approaches 0.

High-Level Functional Specification. Our high-level functional specification of MW closely models the mathematical specification of MW given in Fig. 5. For example, the following four definitions:

```
Definition weights : Type ≜ {ffun A → ℚ}.
Definition costs : Type ≜ {ffun A → ℚ}.
Definition init_weights : weights ≜ λ(_ : A) ⇒ 1.
Definition update_weights (w:weights) (c:costs) : weights ≜
  λa : A ⇒ w a ∗ (1 − η ∗ c a).
```

construct the types of weight (weights) and cost vectors (costs), represented as finite functions from A to $\mathbb{Q}$; define the initial weight vector (init_weights), which maps all actions to cost 1; and define the MW weight update rule (update_weights). The recursive function:

```
Fixpoint weights_of (cs : seq costs) (w : weights) : weights ≜
  if cs is c :: cs′ then update_weights (weights_of cs′ w) c else w.
```

defines the vector that results from using update_weights to repeatedly update w with respect to cost vectors cs.

Adaptive Vs. Oblivious Adversaries. In our high-level specification of MW, we parameterize functions like weights_of by a fixed sequence of cost vectors cs rather than model interaction with the environment, as is done in Fig. 5. An execution of our low-level interpreted MW, even against an adaptive adversary, is always simulatable by the high-level functional specification by recording in the low-level execution the cost vectors produced by the adversary, as is done by the SPrevCosts field (Sect. 5.5), and then passing this sequence to weights_of. This strategy is quite similar to using backward induction to solve the MW game for an oblivious adversary.

Connecting the Dots. To connect the MW interpreter to the high-level specification, we prove a series of refinement theorems (technically, backward simulations). As example, consider:

```
Lemma interp_step_plus :
  ∀(a₀ : A) (s : state A) (t t' : cstate) (c : com A),
  interp c t = Some t' →
  match_states s t →
  ∃c' s', final_com c' ∧
    ((c =CSkip ∧ s = s') ∨ step_plus a₀ c s c' s') ∧
    match_states s' t'.
```

which relates the behavior of the interpreter (interp c t) when run on an arbitrary command c in cstate t to our model of MW DSL commands as specified by the operational semantics.

To prove that the operational semantics correctly refines our high-level functional specification of MW (and therefore satisfies the regret bounds given at the start of Sect. 5.6), we prove a similar series of refinements. Since backward simulations compose transitively, we prove regret bounds on our interpreted MW just by composing the refinements in series. The bounds we prove in this way are parametric in the environment oracle with which MW is instantiated. When the oracle state types differ from source to target in a particular simulation, as is the case in our proof that the MW DSL interpreter refines the operational semantics, we require that the oracles simulate as well.

6 Coordinated MW

A system of multiple agents each running MW yields an ϵ-CCE of the underlying game. If the game being played is smooth – for example, it was built using the combinators of the Smooth Games DSL of Sect. 4 – then the resulting ϵ-CCE has bounded social cost with respect to a globally optimal strategy. In this section, we put these results together by (1) defining an operational semantics of distributed interaction among multiple clients each running MW, and (2) proving that distributed executions of this semantics yield near-optimal solutions, as long as the underlying game being played is smooth.

6.1 Machine Semantics

We model the evolution of the distributed machine by the operational semantics in Fig. 8. Client states (client_state) bundle commands from the MW DSL (Sect. 5) with MW states parameterized by the ClientPkg oracle. The client oracle send and receive functions model single-element (pin) queues, represented as values of type option (dist A), storing values sent by an MW node, and of type option ($A \to Q$), storing values received by an MW node.

States of the coordinated machine (type machine_state N A) map client indices in range $[0..N-1]$ to client states (type client_state A). Machine states also record, at each iteration of the distributed MW protocol, the history of distributions received from the clients in that round (type seq ($[0..N-1] \to$ dist A)), which will be used to prove Price of Anarchy bounds in the next section (Sect. 6.2). We say that all_clients_have_sent in a particular machine state m,

Client Oracle

$$\begin{array}{l}
\mathsf{ClientPkg} \triangleq \\
\quad \{\ \mathsf{sent} : \mathsf{option}\ (\mathsf{dist}\ A); \\
\qquad \mathsf{received} : \mathsf{option}\ (A \to \mathbb{Q}); \\
\qquad \mathsf{received\_ok} : \forall v.\ \mathsf{received} = \mathsf{Some}\ v \to \forall a.\ 0 \leq v_a \leq 1\ \} \\
\mathsf{client\_oracle\_recv}\ A\ (p : \mathsf{ClientPkg})\ (- : \mathsf{unit})\ (v : A \to \mathbb{Q})\ (p' : \mathsf{ClientPkg}) \triangleq \\
\quad p.\mathsf{received} = \mathsf{Some}\ v \wedge p'.\mathsf{received} = \mathsf{None} \wedge p'.\mathsf{sent} = p.\mathsf{sent} \\
\mathsf{client\_oracle\_send}\ A\ (p : \mathsf{ClientPkg})\ (d : \mathsf{dist}\ A)\ (- : \mathsf{unit})\ (p' : \mathsf{ClientPkg}) \triangleq \\
\quad p.\mathsf{sent} = \mathsf{None} \wedge p'.\mathsf{sent} = \mathsf{Some}\ d \wedge p'.\mathsf{received} = p.\mathsf{received}
\end{array}$$

Machine States

$$\begin{array}{l}
\mathsf{client\_state}\ A \ni \sigma \triangleq (\mathsf{com}\ A * \mathsf{state}\ A\ \mathsf{ClientPkg}\ \mathsf{unit}) \\
\mathsf{machine\_state}\ N\ A \ni m \triangleq \\
\quad \{\ \mathsf{clients} : [0..N-1] \to \mathsf{client\_state}\ A; \\
\qquad \mathsf{hist} : \mathsf{seq}\ ([0..N-1] \to \mathsf{dist}\ A)\ \} \\
\mathsf{all\_clients\_have\_sent}\ A\ (m : \mathsf{machine\_state})\ (f : [0..N-1] \to \mathsf{dist}\ A) \triangleq \\
\quad \forall i : [0..N-1].\ \mathbf{let}\ (-, \sigma) \triangleq \mathsf{m.clients}\ i\ \mathbf{in} \\
\quad (\mathsf{SOracleSt}\ \sigma).\mathsf{received} = \mathsf{None} \wedge (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{Some}\ f_i.
\end{array}$$

Machine Step $\boxed{\vdash m \Longrightarrow m'}$

$$\mathsf{cost\_vec}\ A\ i\ :\ A \to \mathbb{Q} \triangleq \lambda a.\ \sum_{(p:[0..N-1]\to A \mid p_i = a)} \prod_{(j \mid i \neq j)} f_j\ p_j * C_i\ p$$

$$\frac{\begin{array}{c} m.\mathsf{clients}\ i = (c, \sigma) \qquad m'.\mathsf{clients}\ i = (c, \sigma') \qquad \sigma\ \sim_{\mathcal{O}}\ \sigma' \\ (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{None} \qquad (\mathsf{SOracleSt}\ \sigma').\mathsf{received} = \mathsf{Some}\ (\mathsf{cost\_vec}\ f\ i) \end{array}}{\mathsf{server\_sent\_cost\_vector}\ i\ f\ m\ m'}$$

$$\frac{m.\mathsf{clients}\ i = (c, \sigma) \qquad (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{None} \qquad \vdash c, \sigma \Rightarrow c', \sigma'}{\vdash m \Longrightarrow m\{\ \mathsf{clients} \triangleq m.\mathsf{clients}[i \mapsto (c', \sigma')]\ \}}\ \mathsf{ClientStep}$$

$$\frac{\begin{array}{c} \mathsf{all\_clients\_have\_sent}\ m\ f \\ (\forall i.\ \mathsf{server\_sent\_cost\_vector}\ i\ f\ m\ m') \qquad m'.\mathsf{hist} = f :: m.\mathsf{hist} \end{array}}{\vdash m \Longrightarrow m'}\ \mathsf{ServerStep}$$

Fig. 8. Semantics of the distributed machine

committing to the set of distributions f, if each client's received buffer is empty and its sent buffer contains the distribution f_i, of type dist A.

The machine step relation models a server–client protocol, distinguishing server steps (ServerStep) from client steps (ClientStep). Client steps, which run commands in the language of Fig. 7, may interleave arbitrarily. Server steps are synchronized by the all_clients_have_sent relation to run only after all clients have completed the current round. The work done by the server is modeled by the auxiliary relation server_sent_cost_vector $i\ f\ m\ m'$, which constructs and sends to client i the cost vector derived from the set of client distributions f. The relation $\sigma \ \sim_{\mathcal{O}}\ \sigma'$ states that σ and σ' are equal up to their SOracleSt components.

In the distributed MW setting, the cost to player i of a particular action $a : A$ is defined as the expected value, over all N-player strategy vectors p in which player i chose action a ($p_i = a$), of the cost to player i of p, with the expectation over the $(N-1)$-size product distribution induced by the players $j \neq i$.

6.2 Convergence and Optimality

Our proof that MW is no regret (Sect. 5) extends to system-wide convergence and optimality guarantees, with respect to the distributed execution model of Fig. 8 in which each client runs our MW implementation. The proof has three major steps:

1. Show that no-regret clients implementing MW are still no regret when interleaved in the distributed semantics of Fig. 8.
2. Prove that per-client regret bounds – one for each client running MW – imply system-wide convergence to an ϵ-CCE.
3. Use POA results for smooth games from Sect. 4 to bound the cost, with respect to that of an optimal state, of all such ϵ-CCEs.

Composing 1, 2, and 3 proves that the distributed machine of Fig. 8 – when instantiated to clients running MW – converges to near-optimal solutions to smooth games. We briefly describe each part in turn.

Part 1 : No-regret clients are still no regret when interleaved. That MW no-regret bounds lift to an MW client running in the context of the distributed operational semantics of Fig. 8 follows from the oracular structure of our implementation of MW (Sect. 5) – clients interact with other clients and with the server only through the oracle.

In particular, for any execution $\vdash m \Longrightarrow^{+} m'$ of the machine of Fig. 8, and for any client i, there is a corresponding execution of client i with respect to a small nondeterministic oracle that simply "guesses" which cost vector to supply every time the MW client executes a recv operation. Because MW is no regret for all possible sequences of cost vectors, proving a refinement against the nondeterministic oracle implies a regret bound on client i's execution from state m_i to state m'_i.

We lift this argument to all the clients running in the Fig. 8 semantics by proving the following theorem:

Theorem all_clients_bounded_regret $A\ m\ m'$ T (ϵ : rat) :
  hist m = nil $\rightarrow 0 <$ size (hist m') $\rightarrow$ final_state m' $\rightarrow$
  $\vdash m \Longrightarrow^{+} m' \rightarrow$
  ($\forall i$, m.clients i = (mult_weights A T, init_state A η tt (init_ClientPkg A))) $\rightarrow$
  η + ln size_A/(η ∗T) $\leq \epsilon \rightarrow$
  machine_regret_eps m' ϵ.

The predicate $\mathsf{machine\_regret\_eps}$ holds in state s', against regret bound ϵ, if all clients have expected regret in state s' at most ϵ (with respect to the σ_T distribution we describe below), for any rational ϵ larger than $\eta + \mathsf{ln\ size\_A}/(\eta * \mathsf{T})$ (the regret bound we proved of MW in Sect. 5).

We assume that the history is empty in the initial state ($\mathsf{hist}\ m = \mathsf{nil}$), and that at least one round was completed ($0 < \mathsf{size}\ (\mathsf{hist}\ m')$). By $\mathsf{final\_state}\ m'$, we mean that all clients have synchronized with the server (by receiving a cost vector and sending a distribution) and then have terminated in CSkip. All clients in state m are initialized to execute T steps of MW over game A ($\mathsf{mult\_weights}\ A\ \mathsf{T}$), from an initial state and initial $\mathsf{ClientPkg}$.

Part 2: System-wide convergence to an ϵ-CCE. The machine semantics of Fig. 8 converges to an approximate Coarse Correlated Equilibrium (ϵ-CCE).

More formally, consider an execution $\vdash m \Longrightarrow^+ m'$ of the Fig. 8 semantics that results in a state m' for which $\mathsf{machine\_regret\_eps}\ m'\ \epsilon$ (all clients have regret at most ϵ, as established in Part I). The distribution σ_T, defined as the time-averaged history of the product of the distributions output by the MW clients at each round, is an ϵ-CCE:

$$\sigma_T \triangleq \lambda p.\ \frac{\sum_{i=1}^{T} \prod_{j=1}^{N} (\mathsf{hist}\ m')_i^j\ p_j}{T}$$

By $(\mathsf{hist}\ m')_i^j$ we mean the distribution associated to player j at time i, as recorded in the execution history stored in state m'. The value $((\mathsf{hist}\ m')_i^j\ p_j)$ is the probability that client j chose action p_j in round i.

We formalize this property in the following Coq theorem:

Theorem $\mathsf{machine\_regret\_eCCE}\ m'\ \epsilon$:
  $\mathsf{machine\_regret\_eps}\ m'\ \epsilon \rightarrow$
  $\mathsf{eCCE}\ \epsilon\ \sigma_T$.

which states that σ_T is an eCCE, with approximation factor ϵ, as long as each client's expected regret over σ_T is at most ϵ ($\mathsf{machine\_regret\_eps}\ m'\ \epsilon$) – exactly the property we proved in Part 1 above.

Part 3 System-wide regret bounds. The machine semantics of Fig. 8 converge to a state with expected cost bounded with respect to the optimal cost.

Consider an execution of the Fig. 8 semantics $\vdash m \Longrightarrow^+ m'$ and an ϵ satisfying the conditions of $\mathsf{all\_clients\_bounded\_regret}$. If the underlying game is smooth, the expected cost of the time-averaged distribution of the clients in m', σ_T, is bounded with respect to the cost of an optimal strategy profile s' by the following Coq theorem:

Theorem $\mathsf{systemwide\_POA\_bound}\ A\ m\ m'\ \mathsf{T}\ (\epsilon : \mathsf{rat})\ s'$:
  $\mathsf{hist}\ m = \mathsf{nil} \rightarrow \vdash m \Longrightarrow^+ m' \rightarrow 0 < \mathsf{size}\ (\mathsf{hist}\ m') \rightarrow \mathsf{final\_state}\ m' \rightarrow$
  $(\forall i,\ m.\mathsf{clients}\ i = (\mathsf{mult\_weights}\ A\ \mathsf{T},\ \mathsf{init\_state}\ A\ \eta\ \mathsf{tt}\ (\mathsf{init\_ClientPkg}\ A))) \rightarrow$
  $\eta + \mathsf{ln\ size\_A}/(\eta*\mathsf{T}) \leq \epsilon \rightarrow$
  $\mathsf{optimal}\ s' \rightarrow$
  $\mathsf{ExpectedCost}\ \sigma_T \leq \lambda/(1-\mu) * \mathsf{Cost}\ s' + (\mathsf{N}*\epsilon/(1-\mu))$

In the above theorem, λ and μ are the smoothness parameters of the game A while N is the number of players. $\mathsf{Cost}\ s'$ is the social (total) cost of the optimal state s'.

7 Related Work

Reinforcement Learning, Bandits. There is extensive work on reinforcement learning [39], multi-agent reinforcement learning (MARL [19]), and multi-armed bandits (MAB, [15]), more than can be cited here. We note, however, that Q-learning [41], while similar in spirit to MW, addresses the more general scenario in which an agent's action space is modeled by an arbitrary Markov Decision Process (in MW, the action space is a single set A). Our verified MW implementation is most suitable, therefore, for use in the full-information analog of MAB problems, in which actions are associated with "arms" and each agent learns the cost of all arms – not just the one it pulled – at each time step. In this domain, MW has good convergence bounds, as we prove formally of our implementation in this paper. Relaxing our verified MW and formal proofs to the partial information Bandit setting is interesting future work.

Verified Distributed Systems. EventML [33] is a domain-specific language for specifying distributed algorithms in the Logic of Events, which can be mechanically verified within the Nuprl proof assistant. Work has been done to develop methods for formally verifying distributed systems in Isabelle [20]. Model checking has been used extensively (e.g., [21,24]) to test distributed systems for bugs.

Verdi [42] is a Coq framework for implementing verified distributed systems. A Verdi system is implemented as a collection of handler functions which exchange messages through the network or communicate with the "outside world" via input and output. Application-level safety properties of the system can be proved with respect to a simple, idealized network semantics. A verified system transformer (VST) can then be used to transform the executable system into one which is robust to network faults such as reordering, duplication, and dropping of packets. The safety properties of the system proved under the original network semantics are preserved under the new faulty semantics, with minimal additional proof effort required of the programmer.

The goals of Verdi are complementary to our own. We implement a verified no-regret MW algorithm, together with a language of Roughgarden smooth games, for constructing distributed systems with verified convergence and correctness guarantees. Verdi allows safety properties of a distributed system to be lifted to analogous systems which tolerate various network faults, and provides a robust runtime system for execution in a practical setting. It stands to reason, then, that Verdi (as well as follow-on related work such as [37]) may provide a natural avenue for building robust executable versions of our distributed applications. We leave this for future work.

Chapar [23] is a Coq framework for verifying causal consistency of distributed key-value stores as well as correctness of client programs with respect to causally consistent key-value stores. The implementation of a key-value store is proved correct with respect to a high-level specification using a program refinement method similar to ours. Although Chapar's goal isn't to verify robustness to network faults, node crashes and message losses are modeled by its abstract operational semantics.

IronFleet [18] is a framework and methodology for building verified distributed systems using a mix of TLA-style state machine refinement, Hoare logic, and automated theorem proving. An IronFleet system is comprised of three layers: a high-level state machine specification of the overall system, a more detailed distributed protocol layer which describes the behavior of each agent in the system as a state machine, and the implementation layer in which each agent is programmed using a variant of the Dafny [22] language extended with a trusted set of UDP networking operations. Correctness properties are proved with respect to the high-level specifications, and a series of refinements is used to prove that every behavior in the implementation layer is a refinement of some behavior in the high-level specification. IronFleet has been used to prove safety and liveness properties of IronRSL, a Paxos-based replicated state machine, as well as IronKV, a shared key-value store.

Alternative Proofs. Variant proofs of Theorem 1, such as the one via KL-divergence (cf. [1, Section 2.2]), could be formalized in our framework without modifying most parts of the MW implementation. In particular, because we have proved once and for all that our interpreted MW refines a high-level specification of MW, it would be sufficient to formalize the new proof just with respect to the high-level program of Sect. 5.6.

8 Conclusion

This paper reports on the first formally verified implementation of Multiplicative Weights (MW), a simple yet powerful algorithm for approximately solving Coarse Correlated Equilibria, among many other applications. We prove our MW implementation correct via a series of program refinements with respect to a high-level implementation of the algorithm. We present a DSL for building smooth games and show how to compose MW with smoothness to build distributed systems with verified Price of Anarchy bounds. Our implementation and proof are open source and available online.

Acknowledgments. This material is based on work supported by the National Science Foundation under Grant No. CCF-1657358. We thank the ESOP anonymous referees for their comments on an earlier version of this paper.

References

1. Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theor. Comput. **8**(1), 121–164 (2012)
2. Awerbuch, B., Azar, Y., Epstein, A.: The price of routing unsplittable flow. In: Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing, pp. 57–66. ACM (2005)
3. Bachelier, L.: Théorie mathématique du jeu. Annales Scientifiques de l'Ecole Normale Supérieure **18**, 143–209 (1901)
4. Bagnall, A., Merten, S., Stewart, G.: Brief announcement: certified multiplicative weights update: verified learning without regret. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017. ACM (2017)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2013)
6. Bhawalkar, K., Gairing, M., Roughgarden, T.: Weighted congestion games: price of anarchy, universal worst-case examples, and tightness. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 17–28. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15781-3_2
7. Blum, A., Monsour, Y.: Learning, regret minimization, and equilibria. In: Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V. (eds.) Algorithmic Game Theory. Cambridge University Press, Cambridge (2007). Chapter 4
8. Borel, E.: La théorie du jeu et les équations intégrales à noyau symétrique. Comptes rendus de l'Académie des Sci. **173**(1304–1308), 58 (1921)
9. Borowski, H., Marden, J.R., Shamma, J.: Learning efficient correlated equilibrium. Submitted for journal publication (2015)
10. Christodoulou, G., Koutsoupias, E.: The price of anarchy of finite congestion games. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 67–73. ACM (2005)
11. Demaine, E.D., Hajiaghayi, M., Mahini, H., Zadimoghaddam, M.: The price of anarchy in network creation games. In: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 292–298. ACM (2007)
12. Fanelli, A., Moscardelli, L., Skopalik, A.: On the impact of fair best response dynamics. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 360–371. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32589-2_33
13. Foster, D.P., Vohra, R.V.: Calibrated learning and correlated equilibrium. Games Econ. Behav. **21**(1), 40–55 (1997)
14. Freund, Y., Schapire, R.E.: A desicion-theoretic generalization of on-line learning and an application to boosting. In: Vitányi, P. (ed.) EuroCOLT 1995. LNCS, vol. 904, pp. 23–37. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59119-2_166
15. Gittins, J.C.: Bandit processes and dynamic allocation indices. J. R. Stat. Soc. Ser. B (Methodol.) **41**(2), 148–177 (1979)
16. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Technical report, INRIA (2015)
17. Hart, S., Mas-Colell, A.: A simple adaptive procedure leading to correlated equilibrium. Econometrica **68**(5), 1127–1150 (2000)

18. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17. ACM (2015)
19. Hu, J., Wellman, M.P., et al.: Multiagent reinforcement learning: theoretical framework and an algorithm. In: ICML, vol. 98, pp. 242–250. Citeseer (1998)
20. Küfner, P., Nestmann, U., Rickmann, C.: Formal verification of distributed algorithms. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 209–224. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33475-7_15
21. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
22. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
23. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: ACM SIGPLAN Notices, vol. 51. ACM (2016)
24. Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (2009)
25. Littlestone, N., Warmuth, M.K.: The weighted majority algorithm. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science. IEEE (1989)
26. Monderer, D., Shapley, L.S.: Potential games. Games Econ. Behav. **14**(1), 124–143 (1996)
27. Nash, J.: Non-cooperative games. Ann. Math. **54**(2), 286–295 (1951)
28. Nash, J.F.: Equilibrium in n-player games. Proc. Natl. Acad. Sci. (PNAS) **36**(1), 48–49 (1950)
29. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior, vol. 60. Princeton University Press, New Jersey (1944)
30. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory, vol. 1. Cambridge University Press, New York (2007)
31. Panchekha, P., et al.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Not. **50**(6), 1–11 (2015)
32. Perakis, G., Roels, G.: The price of anarchy in supply chains: quantifying the efficiency of price-only contracts. Manag. Sci. **53**(8), 1249–1268 (2007)
33. Rahli, V.: Interfacing with proof assistants for domain specific programming using EventML. In: Proceedings of the 10th International Workshop on User Interfaces for Theorem Provers, Bremen, Germany (2012)
34. Ramananandro, T., et al.: A unified Coq framework for verifying C programs with floating-point computations. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. ACM (2016)
35. Roughgarden, T.: Intrinsic robustness of the price of anarchy. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 513–522. ACM (2009)
36. Roughgarden, T.: Twenty Lectures on Algorithmic Game Theory. Cambridge University Press, New York (2016)
37. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. In: Proceedings of the ACM on Programming Languages, vol. 2 (POPL), Article 28 (2018)

38. Spitters, B., Van der Weegen, E.: Type classes for mathematics in type theory. Math. Struct. Comput. Sci. **21**(04), 795–825 (2011)
39. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, vol. 1. MIT press, Cambridge (1998)
40. Vetta, A.: Nash equilibria in competitive societies, with applications to facility location, traffic routing and auctions. In: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science, pp. 416–425. IEEE (2002)
41. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
42. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: ACM SIGPLAN Notices, vol. 50, pp. 357–368. ACM (2015)
43. Zermelo, E.: Über eine anwendung der mengenlehre auf die theorie des schachspiels. In: Proceedings of the Fifth International Congress of Mathematicians, vol. 2, pp. 501–504. II, Cambridge University Press, Cambridge (1913)