A Foundational Framework for the Specification and Verification of Mechanism Design

PIERRE JOUVELOT, Mines Paris, PSL University, France
EMILIO J. GALLEGO ARIAS, Inria, France

May, 12, 2022

We introduce `mech.v`, a new framework for the specification and verification of mechanisms, implemented using the Coq interactive theorem prover and the Mathematical Components library.

We provide a general definition of mechanism, a subclass capturing auctions, and a few examples of auctions including three notions of Vickrey-Clarke-Groves (VCG), proving their main incentive properties such as truthfulness with reasonable verification effort.

We define and prove some more lemmas and definitions usually found in the mechanism design (MD) literature such as Pareto-optimality, rationality, dominance and Nash equilibria.

We also define a mechanism refinement system, which we use to relate the implementation of an efficient version of the VCG mechanism for an online advertisement auction to the general VCG specification, allowing us to transfer the truthfulness property from the generic proof.

We hope that our library can be useful as a formally verified, unambiguous reference with applications ranging from verification of deployed mechanism to education, and that this is a first step to gather interesting results and definitions from the MD literature to eventually provide a comprehensive mechanically formalized reference.

1 INTRODUCTION

We live in an era where the complexity and size of mathematics and computer programs have well gone beyond what humans can understand and process. State-of-the-art mathematical papers routinely take hundreds of pages these days, and can only be read by a very reduced number of experts. Checking, and understanding them can take months, or even years, and the process remains brittle, as it is hard for human readers to reach full confidence in such results. Moreover, in addition to their academic impact, mistakes in mathematical theorems and software can be extremely costly, and create mistrust among rational players, either due to risk, to complexity, or to both accounts. Fortunately, solutions exist to help address those issues.

Formal Verification. The development of modern mathematical logic in the 20th century and, in particular, the advent of mechanized *type theory*[1] in the second half of it have provided mathematicians and computer scientists the means to check the truth of mathematical statements beyond any reasonable doubt.

Building on the development of this type-theoretical approach to logic, state-of-the-art computer programs called "proof assistants" such as Coq or Lean combine excellent expressivity with very strong guarantees of correctness, by means of reducing the proof-checking problem to a computation using a minimal, trusted kernel. Significant milestones have been achieved in the last years, including fully verified proofs of the 4-color theorem, the Kepler conjecture, the Feit-Thompson theorem for the classification of finite groups, and the construction of the Compcert verified C compiler.

Such pioneering work has enabled mathematicians and computer scientists to push the boundaries on the complexity of the reasoning they are willing to handle, as they can now be assisted by machines. This also helps saving reviewer time, which is a scarcer resource every day; see, for example, the effort involved in trying to review Shinichi Mochizuki's proof of the ABC conjecture.

[1] Caveat: this notion is totally different from the one used in mechanism design (See Section 3.1).

Moreover, these proof assistants, also called "interactive theorem provers" (ITP), also allow the verification of not only mathematics, but of algorithms that can be proved correct, and then used as part of the proof itself.

Very recently, a considerable part of the mathematics community have started to embrace ITPs in their day-to-day work, motivated by the complexity our their own proofs becoming challenging. For example, Peter Scholze used the Lean theorem prover to help him understand the correctness of an ongoing proof in analytic geometry [2], with excellent results, quoting M. Scholze:

> "Theorem 9.4 is an extremely technical statement, whose proof is however the heart of the challenge, and is the only result I was worried about. So with its formal verification, I have no remaining doubts about the correctness of the main proof. Thus, to me the experiment is already successful."

Not only do ITPs allow to ensure the correctness of a mathematical proof beyond any reasonable doubt, they also provide a standard, non-ambiguous mathematical language for formal specifications, which is a net improvement over the status-quo of communicating and stating properties using ink on paper or,, at best, LaTeXfiles. The construction of *formally-verified* libraries that serve as a repository of mathematical definitions and facts is well underway, with the a few popular examples being the AFP, based on Isabelle, mathlib, based on Lean, and math-comp, based on Coq. Research to have these libraries communicate among them is also underway.

Mechanism design. At the same time, as algorithmic governance advances steadily in its application domain, mechanism design has become very important in the world and takes a more central role in everyone's life. Be it in blockchains, distributed energy grids, advertisement auctions, or car routing, strategic agent planning plays a central role in today's economy. However, such pervasiveness comes with its own challenges due to complexity, opacity, and risk management due to mistakes and adversarial scenarios. Our motivation for the research reported in this paper arose exactly when we found ourselves facing that particular last point. When trying to implement the Vickrey-Clarke-Groves (VCG) auction as a library for use in a so-called "smart contract" in blockchain technology, we faced several challenges: we needed to define VCG formally, check that the implementation was bullet-proof, and moreover, state VCG's properties in a formal language that could be understood by users and verification-tool clients of our code. Our attempt was a success, in the sense that it took us just a few lines of Coq code to rigorously define VCG and prove its core property, truthfulness. This is when we decided to test the idea of building a small foundational library of mechanism-design implementations and concepts, with both provides canonical, machine-level definitions of mechanism and checks the correctness of definitions and implementations.

Contributions. Our paper contributes to what we consider as three key issues that need to be addressed before enabling the emergence of formally verified mechanisms in the field, namely reference, assurance and extension, detailed below.

Reference The first contribution is `mech.v`, an open-source library of formally defined core mechanisms and concepts of mechanism design. The library is implemented using the widely-used Coq proof assistant, since we believe the Coq notation system and language makes the definitions accessible to non-experts, just after a minimal amount of training.
We provide a generic definition of mechanism (using the class-based methodology of [16], an auction sub-class, and generic properties including *strategy-dominance* and *truthfulness*, which apply to all instances of the mechanism class. We also provide a few instances of the mechanism and auction classes; in particular we define Fixed Price, First Price, Second Price,

[2] https://xenaproject.wordpress.com/2021/06/05/half-a-year-of-the-liquid-tensor-experiment-amazing-developments/

Generalized Second Price, General VCG, and a refinement of VCG from [26] apt for online ad auctions. We also prove some key properties, such as the truthfulness of each of those mechanisms (or the lack of it).

This library can be seen as having both educational and scientific value, and allows us to ask for feedback from the community as to understand what further concepts and mechanisms could be useful to be added to it.

Assurance Similarly to traditional applications of theorem proving in mathematics and computer science, we check that the mechanisms defined in the libraries meet their intended specification, benefiting from the additional assurance that a mechanized proof brings as compared with a pen-and-paper proof outline.

In particular, we borrow a common technique from the programming literature, *proof by refinement*, to prove that the implementation of VCG for online ads auctions is a refinement of the general VCG version, thus transferring the truthfulness property for (almost) free. We introduce a *relational refinement mechanism* written in Coq that allows to reduce the proof of truthfulness of a mechanism `m1` to that of proving it is a refinement of an existing truthful mechanism `m2`, which is usually a much more compact and organized proof when stated mechanically than direct proofs.

While we don't think correctness is a concern in the current economic and mechanism design literature (modulo errors such as the ones in Reinhart/Rogoff, which we don't directly address[3]), we think this work is a step towards handling mechanically more complex proofs (such as the ones for probabilistic mechanisms), which are tedious to review or trust, and a step towards more reproducible research.

Extension A third important contribution of this paper appears by virtue of the foundational characteristic of our verification tool of choice: Coq proofs are simple combinatorial syntactic objects that can be manipulated and checked by minimalistic proof checkers. This way, the agents participating in a mechanism don't even have to trust the provided proof. They can opt to trust a different reference proof checking implementation to check for themselves that a particular strategy is, indeed, dominant, increasing their trust in the mechanism while relieving from possibly heavy cognitive load. In turn, this may allow the mechanism designer to use even more complex design methods that remain truthful while providing better welfare for all.

This virtuous circle may have several beneficial effects, for example in improving agents' rationality and participation. All an agent needs to understand to fully trust the mechanism is the statement of the mechanism's properties, which are, with the currently Coq technology, expressed in a manner close enough to human (mathematical) language.

Structure of the paper. After this introduction (Section 1), Section 2 surveys the related work for formal verification and design. Building upon Section 3, which presents the basis of the type-theoretical approach to theorem proving, Section 4 introduces the main concepts of the `mech.v` library for mechanism design, while some simple applications to basic auctions are sketched in Section 5. We detail our relational framework for mechanisms in Section 6, using VCG auctions, with its General and ad-oriented versions, as a use case. We conclude and discuss possible future work in Section 7.

2 RELATED WORK ON FORMAL VERIFICATION AND MECHANISM DESIGN

A first related work is [3], where relational logics [1, 2] and relational properties [10] capture the reasoning used in mechanism design quite effectively. The approach of [3] relied on a custom type

[3]See, for instance, https://www.bbc.com/news/magazine-22223190

system for a toy functional programming language, such that the mechanisms themselves and their utility could be specified as functions in this language where types (in the sense of C or Java types) can capture properties about the inputs and outputs of two different runs of the program. This is expressive enough to capture truthfulness:

$$\mathsf{U}_i : \{x : \mathsf{t} \mid x_1 = v_i\} \to \{r : \mathsf{real} \mid r_1 \geq r_2\}.$$

We can read the type for U as "the utility function U takes as an input a value x of type t for agent i, and outputs the utility r as a real. If the action on the first run, denoted by the subscript 1 is equal to the true belief of the agent, then the utility on the first run will be greater than equal to the one in the second run (subscript 2)". A custom type checker, in combination with SMT[4] solvers[5], was able to check several interesting examples for truthfulness and other properties, including probabilistic ones. This approach was extended in [4] in order to handle more complex properties such Bayesian Incentive Compatibility, and in particular the verification of the Replica-Surrogate-Matching of [19]; a detailed argument was made in this paper on the role that proof certificates can have on improving participation, as agents don't need to understand why the stated properties of a mechanism are obvious, but can rely on the verification tools (see [24]).

The approach followed in [4] is, however, not fully satisfactory for a few reasons.

- The approach used there uses custom-made tools, which, while providing a very good automation, are hard to trust themselves, and to maintain. Moreover, the tools used (including SMT solvers) are not foundational in nature, that is to say, contrary to Coq or Lean, they don't produce certificates that can be checked by a minimalistic, trusted kernel.
- Moreover, and related, the number of axioms relied upon can get large, as these tools cannot usually handle the full mathematics involved in proofs. For example in [4], the existence of a VCG mechanism is assumed, and parts of probability theory too. This can quickly turn into a problem as it is easy to create an incompatible set of axioms.
- And last, the expressivity of the custom tools can become a problem when one is interested in more general or complex theorems needing advanced mathematical notions. It is well understood that stating non-trivial mathematical theorems in a readable way inside interactive proof assistants requires complex notation and class systems (see for example [17]).

In this work, we use a foundational setting, thus improving or solving all these concerns; the definitions and proofs in this paper use standard, state-of-the-art math formalization technology and display, rely on the trusted Coq kernel, and produce certificates in the kernel language.

Closely related to the spirit of our work is [20], where the authors formalize a non-trivial amount of voting theory and definitions. The Lean system is very close to our use of Coq plus the Mathematical Components library, and indeed, technology to reuse proofs from Lean in Coq has been already prototyped so it is not hard to imagine a future where both libraries on mechanism design and voting could be combined. The most significant difference with our work — apart from the fact that we target a very different class of utility-based mechanisms instead of order-based ones — is that we address the issue of implementation of mechanisms, and we provide a program refinement framework that allows to relate a significantly different implementation to its specification in a systematic way.

Closely related too, but with marked differences, is the work of [9, 21]. In their work, they use the Isabelle proof assistant to prove some properties of the combinatorial variant of VCG. Their formulation of VCG is different and less general than ours, and in our setting, thanks to a more functional specification, several of their properties hold by construction. Moreover, they neither consider incentive properties, nor address the questions of providing efficient implementations of

[4]Satisfiability Modulo Theories.

[5]Domain-specific solvers that can decide a class of formulas automatically.

the auction by using a different interface as we do for VCG for Search ad-based auction. Finally, Isabelle doesn't provide proof certificates. A survey on mechanism design and verification by some of the same authors can be found at [22].

In [7], the authors use SMT solvers to prove an incompatibility between mechanism efficiency and manipulation avoidance in randomized aggregation mechanism. This is interesting work, and complementary to ours, as the integration of SMT solvers with interactive proof assistants has advanced a lot in recent times [6, 14]. Our approach, however, is not only limited to arbitrary properties, but also can scale to proofs of very large mathematical complexity, which is out of reach for current SMT solvers.

The work of [8] reduces the truthfulness of particular mechanisms to linear problems that can be checked efficiently; it would be interesting to actually use our framework to understand that particular class of mechanisms and provide a certified decision procedure. Testing for truthfulness is considered in [25]. The automated generation of truthful mechanisms with certain optimal properties is introduced in [11, 28].

3 A QUICK PRIMER ON TYPE THEORY AS A MATHEMATICAL LANGUAGE

Interest in the foundations of modern mathematics gained traction on the second half of the XIX$^{\text{th}}$ century, and achieved an impressive development in the XX$^{\text{th}}$ one. In recent times, the task of determining whether a mathematical statement holds has gained central importance in many areas, as the complexity and size of proofs has grown beyond what human reviewers can address effectively.

There are several different approaches to provide assistance to mathematicians and computer scientists. A particular successful approach to the mechanization of mathematics is *type theory*, which was developed with contributions from B. Russell, A. Church, Martin-Löf, J.Y. Girard, T. Coquand, and many others.

3.1 Types as mathematical objects

The core idea of type theory is to classify objects by their *type*, which can be understood informally as them belonging to a class or set of objects. This has deep connections with programming.

Informally, we can see *type theory* as a kind of *set theory*. Objects x are given a type T, which we usually write $x : T$, which can informally be read as $x \in T$.

Often, T is a logical formula, and p is a program. For example, let T be "$4 = 2 + 2$". In this case, $p : 4 = 2 + 2$ can be read as "p is a proof for $4 = 2 + 2$". This key correspondence between programs and proofs is usually referred to as the Curry-Howard-Kolmogorov correspondence.

A proof checker for type theory uses rules to ensure that $x : T$ is a valid *type judgment*. Thus, for the above example, we will give the input $p : 4 = 2 + 2$ to our proof checker, and it will answer yes or no depending on whether the proof is correct. In the systems we use in practice, this process is decidable, but can take a very long time in some cases, thus proof engineers must use sometimes strategies to refactor proofs so they check in reasonable time.

The second key component type theory provides is *abstraction*, which allows us to reason under hypothesis. A common notation for abstraction is $\forall (x : T), P\ x$, which should be read as "for all objects x of type T, the property P holds for that x. This allows us to instantiate such properties easily. How proofs are built on type theory is beyond the scope of this short overview, but the most common approach is to use a computational interpretation, where a proof p of $p : \forall (x : T), P\ x$ is just a computable function that given as input x will return a proof of $P\ x$. Note how, in this setting, objects can also appear in types, a property of *dependent type theories*, a common terminology used for systems that allow this dependency to happen.

Modern type theories provide several facilities to work with top-level definitions, datatypes, records, and many other convenience utilities, but their core remains quite minimal.

An example of a non trivial theorem stated in type theory is the infinitude of primes:

```
Lemma prime_above : forall m, exists p, m < p ∧ prime p.
```

The above lemma reads "for any natural number m, there exists a natural number p, such that a) it is greater than m, and b), it is prime", where `prime` is defined in the mathematical components library, in this case as an algorithm of type `prime : nat → bool`. Note also how Coq can infer that the types of `m` and `p` are natural numbers, thus saving the programmer from verbose proof writing.

Note how close this formulation is to standard mathematical language. Finding a proof of this theorem would involves finding a program with the right type. This can get complex quickly, so in modern practice systems provide *tactics* closer to standard mathematical writing all will produce the right proofs. Note that tactics don't need to be trusted themselves, because at the end the system will check the generated proof term.

3.2 Coq overview

Coq [29] is mature interactive theorem prover, based on the Calculus of Inductive Constructions [12, 13, 27]. A recipient of the ACM Software System Award, Coq has been used to reach several first verifiation milestones such as the formal proof of the four colors theorem [18], the Feit-Thompson theorem [17], the CompCert verified C-compiler [23], and to this day plays a central role in the programming languages community.

The core of Coq is the Gallina Programming Language, which is higher-order dependently-typed λ-calculus in which proofs, types, and definitions are written. The main syntactic particularity of Gallina, as with other functional programming languages is that function application is denoted by space. Thus, the math-based term $f(x, y)$ is written `f x y` in Gallina.

Coq provides many facilities to help users to write formal proofs, including all kind of automatic and interactive tactics, a type inference engine, a proof search engine and programmable tactic language, a plugin system, an advanced parser and notation engine, an "extraction" system to generate programs, and a library and module organization system.

3.3 Trust Setting

A core feature of the type-theory setting for mechanized proof checking is that theorems and definitions are elaborated to a quite minimal core language, which can be then checked for consistency using a fairly small trusted "kernel". This is fairly remarkable, as, while current kernels are not designed to be adversarial-resistant, they have been in production for many decades and it is extremely unlikely for a user to find a bug in them. Thus, we can assert that verification using type theory provides a high degree of trust and guarantees; moreover, proof terms can be checked independently on different hardware / software platforms, thus providing assurance against possible Trojan horses by a malicious player. In the case of Coq, the kernel is around 10,000 lines of code, written in the OCaml functional programming language, and is not adversarial resistant, though significant progress has been made to develop a smaller and bootstrapped verified kernel so far.

3.4 Mathematical Structures

A common way to structure knowledge in interactive proof assistants is by defining a *class hierarchy*. Similar to object-oriented programming, users can define *class interfaces*, *subclasses*, and *implementations*. A typical example comes from universal algebra. We will say that a type T is a group, if there exists operations $\odot$ and $(\cdot)^{-1}$, of respective types $T \to T \to T$ and $T \to T$, such that the usual group laws hold. Usually, these operations and laws are *packed* using a *record*, which is

similar to a structure in most programming languages. We can now define the sub-class of groups that are commutative, by extending the record to include a new law $comm : \forall xy, x \odot y = y \odot x$. Note that all commutative groups are also groups.

The problem of defining *usable* mathematical structures is quite interesting and only partially solved, as the relations between structures can be quite complex and ambiguous. In this paper, we will follow the approach of [16] and use records to pack a particular carrier type and its properties.

3.5 The Mathematical Components Library

The Mathematical Components Library is the result of 20 years of work on the formalization of mathematics, and includes hundredths (if not thousand) of tested mathematical objects ready for use, including finite types, finite sets, matrices, groups, big operators, a full abstract algebraic and numerical hierarchy, linear algebra, etc…

For this work, finite types, tuples (sequences of fixed length), and big operators [5] are the more relevant parts of the library.

4 MECH.V, A LIBRARY FOR MECHANISMS

We describe in this section the main features of the new Coq `mech.v`[6] library dedicated to the specification of pure, i.e., deterministic, mechanisms.

4.1 Basic Definitions

All the major notions related to mechanisms are specified as Coq modules, which pack together definitions and lemmas under a unique naming space. The simplest one is the notion of *agent*.

```
Module agent.
Section Agent.
Variable n : nat.     (* number of agents in the mechanism *)
Definition type := 'I_n.
End Agent.
End agent.
```

An agent is represented as a value of an "ordinal" type, here the type `'I_n` of natural numbers strictly bounded by n, which is a parameter that has to be provided when one wants to create a new agent. For example, an agent `a` can be declared as `Variable (a : agent.type 10).` , where the name of the agent module, `agent`, is used to access names defined in it, such as `type`.

Agents participate to *mechanisms*, which are values of the type `mech.type` (for space reasons, we leave out the `Module mech` and `Section Mech` commands, similar to the ones above; we will do so in the rest of this paper).

```
Record type {A : Type} (n : nat) :=
    new {
        O : Type;              (* domain of outcomes *)
        M : n.-tuple A → O;    (* social outcome for the agents' actions *)
      }.
```

A mechanism is abstracted over two values: `A` is a value of type `Type`, which is intended here to be the one of the actions that agents can perform in the mechanism, and `n`, the number of agents involved. Given these two parameter-like values, mechanisms of type `mech.type A n` are then specified as records, packing both the type `O` of the mechanism's outcomes and `M`, the mechanism's core, which is a function mapping n actions (in `A` packed together in a tuple to one outcome, the social outcome

[6]. `v` is the file extension for Coq files

defined by the mechanism. Note how Coq allows a particular field, here `0`, to be both used as a field of the record, and also a type element of another field.

Finally, agents have behaviors, which are specified as *preferences*' values of the `prefs.type` type. In the code below[7], the variable `mech` is defined as the mechanism type we are dealing with here, while the `agent` variable does something similar[8] for the agent type Note that we introduce here the notion of a `strategy` which we develop in Section 4.3.

```
Variable (A : Type) (n : nat).

Definition mech : Type := @mech.type A n.

Notation agent := (agent.type n).
Notation strategy := (agent → A).

Record type (m : mech) :=
  new {
      v : strategy;                (* true value strategy *)
      U : agent → mech.O m → nat;  (* utility *)
      s : strategy                 (* strategy used in [m] *)
    }.
```

Presently, a value of type `prefs.type` packs, for each agent, three value together: (1) the "true value" strategy of the agent, (2) his or her notion of utility, mapping[9] a value of the outcome domain `mech.O m` of the mechanism `m` passed as an argument to the `prefs.type` construct to a numerical utility, here encoded as an integer, in `nat`, and (3) the current strategy of the agent.

4.2 Hierarchy

If the definitions above can be used to define very general mechanisms, as could, for example, be found in game theory, it is interesting to provide more specialised versions of those. We illustrate here how auctions and truthful mechanisms can be seen as specialized mechanisms.

4.2.1 Auction. An *auction* is a particular class of mechanism, dedicated to allocating resources in a priced market. In this particular case, actions, in `A`, are bids, and thus we restrict here actions to be values of some ordinal type `' I_m`.

```
Notation A := (' I_m).       (* actions are bids *)

Record type :=
    new {
        b :> @mech.type A n;                  (* base mechanism *)
        p : mech.O b → agent → option nat   (* price to pay *)
      }.
```

[7]The `@` sign is a Coq convention to indicate to the system that all parameters will be provided and, thus, that they don't have to be automatically inferred. The special notation `_` can be used as a placeholder for a parameter that Coq is expected to infer.

[8]The Coq `Notation` construct specifies a syntactic shortcut to the right-hand side of the assignment.

[9]In Coq, record fields are seen as functions; so, in the module `mech` defined for mechanisms, the function `O` applied to a mechanism `m` yields the social outcome computed by `m`. As mentioned before, outside of the module, `O` is not visible and, to be accessed, must have its name fully specified, here as `mech.O`.

An auction, of type `auction.type`, builds upon two fields. The first one is `b`, which is a value of its base-mechanism type[10] `@mech.type A n`. The second one, `p`, specifies the pricing convention used in the auction: given an outcome provided by the base mechanism `b`, it yields, for each agent, the price s/he has to pay in the given outcome. This price is given as an `option` type: it can be either `Some` k, if the agent has won in the bidding and has to pay k, or `None`, if the agent has lost.

In addition to providing a clearer definition of what auctions are as mechanisms, using this hierarchical approach is interesting in that it enables the description of general functions and properties that apply to this particular subset of mechanisms. For instance, we provide a default utility function, `auction.U`, that is the usual difference between an agent `i`' value (`v i`) and the price to pay, `p'`, if any.

```
Definition U (a : type) (v : strategy) :=
  fun i (o : mech.O a) ⇒ match p o i with | Some p' ⇒ v i − p' | _ ⇒ 0 end.
```

4.2.2 Truthful mechanism. In the auction case, we extended the notion of a mechanism with a pricing function. Coq enables records to include not only typical values, but also properties. A *truthful mechanism*, a value of the `truthfulMech.type` type, is a mechanism that satisfies the `truthful` property (see Section 4.4). Since the truthfulness property depends on the utility function of agents, a `prefs.type` value needs to be provided as well.

```
Record type :=
    new {
        b :> @mech.type A n;            (* base mechanism *)
        p : prefs.type b;               (* preferences *)
        _ : truthful p                  (* truthful property *)
      }.
```

Note that the third field doesn't have a name, as indicated by the _ sign. The Coq proof mechanism provides ways to access it, however, when need be.

In addition to adding general definitions to modules defined in a hierarchy, as we did in the auction case, general lemmas can be provided as well. In this particular case, we express, and prove formally[11], that all truthful mechanisms are weakly dominant (see Section 4.3 for the Coq definition of this notion) with respect to the "true value" strategy, as specified in the preferences (`p m`) for any truthful mechanisms `m`.

```
Definition true_value_strategy := prefs.v.

Lemma truthful_implies_weakly_dominant : forall (A : eqType) n (m : truthfulMech.type A n),
  weakly_dominant (p m) (true_value_strategy (p m)).
```

4.3 Properties

Mechanism results depend upon agents' strategies, i.e., which actions they take given a mechanism outcome. Here, a *strategy* is a function from agents to actions, for any given mechanism.

```
Variable (m : @mech.type A n).
Variable (p : @prefs.type A n m).

Definition strategy := agent → A.
Definition true_value_strategy := prefs.v p.
```

[10]The :> sign indicates that, wherever a mechanism value is needed, an auction can be seen as such, via its `b` field.
[11]Except in one case, proofs are omitted in this paper.

An important strategy is the "true value" strategy in which an agent performs the action s/he privately thinks the most appropriate; it is one of the elements of a `prefs` value such as `p`, as seen above. Relating one strategy to another is a key element of mechanism design: `mech.v` provides the most important of these relations, including theorems that can be derived from them (see Section 4.3. We discuss here dominance, Nash equilibrium and Pareto optimality.

A strategy `s` for a mechanism `m` is *dominant* for an agent `i` with preferences `p` if, whatever strategy `s'` envisioned for `i` and the other agents, the utility `U` of the action `(s i)` is always greater[12]. Here, actions are gathered into n-tuples, which can be updated with the `set_in_actions` function.

```
Definition U := prefs.U p.

Definition actions (s : strategy) := [tuple s j | j < n].
Definition set_in_actions (s s' : strategy) i := [tuple if j == i then s' j else s j | j < n].

Definition dominates (g : rel nat) (s s' : strategy) (i : agent) :=
  g (U i (m (set_in_actions s' s i))) (U i (m (actions s'))).
```

The `dominates` boolean function is abstracted over the comparison relation `g` (for "greater than" or "greater than or equal", as we see below for the two dominance relations) and compares the utilities of `i` depending on whether it uses the (supposedly) dominating strategy `s` or any other `s'` in the mechanism `m`. Using "standard" mechanism design notation, this is equivalent to $U_i\ (m\ (s'_{-i}, s\ i)) >_g U_i\ (m\ s')$. Notice how the `actions` tuple (`actions s'`) is passed to the `m` mechanism (actually the `M` field of the `m` record, via a coercion not detailed here) to yield the corresponding outcome, passed then as a second argument to `U`.

```
Variable (s : strategy).

Definition weakly_dominant := forall i s', dominates geq s s' i.
Definition strictly_dominant := forall i s', ¬ (s =1 s') → dominates gtn s s' i.
```

The definitions of Nash equilibria and Pareto optimality are provided in a similar fashion: a strategy `s` is a Nash equilibrium if, for any agent `i`, using another strategy `s'` would lead to a lower utility, and it is Pareto-optimal if using a different strategy `s'` that increases her utility would, on the other hand, lower the one of another agent `i'`.

```
Definition Nash_equilibrium (s : strategy) :=                (* ∀ i s', U_i (m s) ≥ U_i (m (s_-i, s' i)) *)
  forall (i : agent) (s' : strategy),
    U i (m (actions s)) >= U i (m (set_in_actions s s' i)).

Definition Pareto_optimal (s : strategy) :=
                          (* ∀ i s', U_i (m (s_-i, s' i) > U_i (m s) ⟹ ∃i', U_i (m (s_-i, s' i) < U_i (m s) *)
  forall (s' : strategy) (i: agent),
    let aa := actions s in
    let a' a := set_in_actions s s' i in
    U i (m a' a) > U i (m aa) → (exists i', U i' (m a' a) < U i' (m aa)).
```

4.4 Theorems

Given the formal definitions of the key properties that strategies can have, one can use those in at least two ways: the first is to prove general theorems about mechanisms, while the second is

[12] We use here the "greater than" (resp., "greater than or equal") function `gtn` (resp. `geq`) on natural numbers.

to specialize those definitions to handle specific mechanisms, for which we describe a powerful relational framework to map one mechanism to another one. We focus here on the first application, and address the second one in Sections 5 and 6.

We have specified and formally proven some key fundamental theorems about mechanisms. For instance, the lemma `dominant_is_Nash` states that any dominant strategy `s` is a Nash equilibrium (proof omitted), while `Nash_uniq` ensures that, if it is strictly dominant, then the Nash equilibrium is unique and equal to `s`:

```
Variable (A : eqType) (n : nat) (m : mech.type n) (p : prefs.type m) (s : strategy A n).

Lemma dominant_is_Nash : weakly_dominant p s → Nash_equilibrium p s.  (* Proof omitted *)

Lemma Nash_uniq :
  strictly_dominant p s → forall s' : strategy A n, Nash_equilibrium p s' → s' =1 s.
Proof.
rewrite /strictly_dominant /dominates /Nash_equilibrium /= => d s' N i.
move: (d i s').
have [] := boolP ([ forall i, s i == s' i]) => [/eqfunP → //|/not_eqfunP ness' /(_ ness')].
by rewrite ltnNge N.
Qed.
```

The proof for the lemma `Nash_uniq` is given here just to give a feel of how such proofs work. The proof begins with the `Proof` keyword; the first `rewrite` command unfolds definitions; `move:` adds the proposition `(d i s')` , which is the application of the property of strict dominance `d` to an agent `i` and strategy `s'` , to the current list of hypothesis; `have` introduces here a case analysis based on whether `s` and `s'` are equal or not (the two branches are treated in the `[.|.]` construct, the first one being trivial); and the final `rewrite` command applies two lemmas (`ltnNge`, provided by the mathcomp library `nat`, states that "less than" is the opposite of "greater or equal", while `N` is the name given to the Nash equilibrium hypothesis present in the lemma specification). The final `Qed` command asks the Coq proof assistant to validate the previous proof.

The hierarchy of mechanisms enables the elegant formulation of other interesting and general theorems about more specific cases. For instance, one can state, and prove, that, for all truthful mechanisms, the true value strategy is weakly dominant.

```
Lemma truthful_implies_weakly_dominant :
  forall (A : eqType) (n : nat) (m : truthfulMech.type A n),
    weakly_dominant (truthfulMech.p m) (true_value_strategy (truthfulMech.p m))
```

Note how the truthfulness property needed here is, in fact, hidden behind the definition of the `truthfulMech.type`, and doesn't need to be stated again explicitly. This approach builds upon the modularity and abstraction features of the Coq mathcomp framework.

5 BASIC AUCTIONS

The `mech.v` library provides the specifications and some key properties for the most common auction mechanisms, namely Fixed Price, First Price, Second Price, Generalized Second Price, General VCG, Combinatorial VCG and VCG for Search; the last three are detailed in Section 6. Truthfulness are derived either by instantiation or use of our relational framework. For non-truthful mechanisms (First Price, Generalized Second Price), we provide explicit counter examples, which are used by the `not_truthful_inv` theorem provided in the library. As a use case for `mech.v`, we focus here on Second Price, for single-item auctions.

The number of agents is abstracted over a variable `n''`, so that the total number, `n`, of agents, in the domain `A` is greater or equal to 2 (Second Price calls for at least that number of agents). A bid proposed by an agent is an ordinal, bounded by `p`; all the bids are gathered in n-tuples, of type `bids`. The Second Price algorithm is defined in its own Section: given an initial set of bids `bs0` and an agent `i`, the algorithm sorts the bids in a decreasing fashion and computes the index `i'` of `i` in the sorted tuple `bs`. If `i` has the highest bid, and thus its index `i'` is `ord0`, the ordinal of value 0, then `i` is the winner and the price s/he has to pay is `price`, which is the value of the bid of the next highest bidder (`agent.succ` is defined in the `agent` module).

```
Variable (n'' : nat).
Definition n' := n''.+1.
Definition n := n'.+1.

Definition A := agent.type n.
Notation agent_succ := (@agent.succ n').

Variable p' : nat.
Definition p := p'.+1.
Definition bid := 'I_p.
Definition bids := n.-tuple bid.

Section Algorithm.
Variable (bs0 : bids) (i : A).
Let bs := tsort bs0.
Let i' := idxa bs0 i.
Definition is_winner := (i' == ord0).
Definition price : nat := tnth bs (agent_succ i').          (* bs_{i'+1} *)
End Algorithm.
```

Now, by viewing this algorithm as an instance of `mech.type`, and more precisely of `auction.type`, we will be able to use the library's definition of `truthfulness` to help us specify this property for Second Price in a concise fashion. The definitions of the auction components are done in two steps: (1) `m` is a mechanism that returns as social outcome an n-tuple of pairs of boolean values, indicating for each agent whether s/he won or not, and the tuple of bids used; (2) `a` is an auction derived from `m` which, for each agent `i` and given outcome `o`, yields the price to pay: either `Some` price computed using the Second Price algorithm, or `None`, when losing.

```
Definition m := mech.new (fun bs ⇒ map_tuple (fun i ⇒ (is_winner bs i, bs)) (agent.agents n)).

Definition a :=
  @auction.new _ _ m (fun o i ⇒ if (tnth o i).1 then Some (price (tnth o i).2 i) else None).
```

Given any true value strategy `v`, we can then directly reuse the default `prefs` value provided in the `auction` module to express the truthfulness property that Second Price mechanism possesses.

```
Variable v : agent.type n → bid.

Definition prefs := auction.prefs a v.

Theorem truthful_SP : truthful prefs.   (* Proof omitted *)
```

Thanks to this instantiation into `auction.type`, we also able to prove that this algorithm is Pareto-optimal (the proof is omitted, here):

```
Lemma Pareto_optimal_SP : Pareto_optimal prefs (true_value_strategy prefs).
```

We proceeded in a similar fashion for all the basic auctions mentioned above. The case of VCG is a bit different, since we relied also on the relational framework introduced by `mech.v`; we detail this in Section 6.

6 MECH.V RELATIONAL FRAMEWORK: A VCG USE CASE

The `mech.v` library provides three versions of the famous Vickrey-Clarke-Groves mechanism: one is the General VCG mechanism, another is Combinatorial VCG, and the last one is VCG for Search, a mechanism for auctioning multiple items. We focus here on General VCG and VCG for Search, since these two specifications are used to illustrate the relational framework introduced by `mech.v` to help transfer the truthfulness property[13] from one mechanism to another one. In this particular case, the somewhat easy proof for General VCG truthfulness is transferred to VCG for Search; the relational framework avoids digging too much in the specifics of the VCG for Search mechanism (a direct proof turns out to be much more cumbersome). Note that this transfer is even easier for Combinatorial VCG, since this mechanism is expressed as a direct instantiation of General VCG, and the proof of truthfulness is thus just one line.

6.1 General VCG

General VCG, listed below, is abstracted over the type `O` of possible outcomes, a particular instance `o0` (to ensure non-emptiness of `O`) and an agent `i`. Here, any agent, among n, is defined by its `bidding`, a finite function that maps any possible outcome into the Coq domain `nat` of natural numbers. General VCG, given its last parameter, a tuple `bs` of `biddings`, must compute the outcome `oStar` that maximizes the total `bidSum o` of bids[14]. In a truthful mechanism, where the bids of agents and their "true values" coincide, this outcome maximizes the global good, or "welfare". For agent `i`, the `price` s/he accordingly has to pay to win whatever is in `oStar` for her is a penalty induced by the impact on the global good of her presence in the bidding process (`welfare_with_i`) compared to when she is not (`welfare_without_i`), which could have yielded a possibly different optimal outcome.

```
Variable (O : finType) (o0 : O) (i : A).

Definition bidding := {ffun O → nat}.
Definition biddings := n.-tuple bidding.
Variable (bs : biddings).
Local Notation "'bidding_ j" := (tnth bs j) (at level 10).

Implicit Types (o : O) (bs : biddings).

Definition bidSum o := \sum_(j < n) 'bidding_j o.                    (* ∑_{j=0}^{n-1} bs_j o *)
Definition bidSum_i o := \sum_(j < n | j != i) 'bidding_j o.          (* ∑_{j=0, j≠i}^{n-1} bs_j o *)

Definition oStar := [arg max_(o > o0) (bidSum o)].                   (* o* = arg max_o ∑_{j=0}^{n-1} bs_j o *)
```

[13] This approach could be extended to other properties.

[14] We assume, for now, that such optimal outcome is unique, a restriction that should be alleviated in future work. Note that, in itself, VCG is non-deterministic in its choice, there.

```
Definition welfare_with_i := bidSum_i oStar.
Definition welfare_without_i := \max_o bidSum_i o.

Definition price := welfare_without_i − welfare_with_i.
                                    (* max_{o∈O} Σ_{j=0,j≠i}^{n−1} bs_j o − Σ_{j=0,j≠i}^{n−1} bs_j o* *)
```

Seeing General VCG as a `mech.type` instance is easy (see definitions below); we define an outcome for `mech.type` has a pair made of the tuple of prices for all agents and the optimal outcome `oStar`. The utility function defined within `prefs` is the natural one, i.e., value minus price.

```
Definition m : mech.type n :=
  mech.new (fun bs ⇒ (map_tuple (fun a ⇒ price o0 a bs) (agent.agents n), oStar o0 bs)).

Variable v : A → bidding O.

Definition p : prefs.type m := prefs.new v (fun i (o : mech.O m) ⇒ v i o.2 − tnth (o.1) i) v.
```

The `mech.v` library provides a proof that General VCG is truthful, i.e., `truthful p`, and also that it has the "no transfer" property (the mechanism never has to pay agents) and is individually rational (all prices are less than the agents' true values).

6.2 VCG for Search

VCG for Search is an special case[15] of General VCG that applies to the auctioning of multiple items of different values. A typical application is the allocation of `k` dedicated slots for advertising present in search return pages, as provided, for instance, by Google; note, though, that Google has been recently said to be moving away from Generalized Second Price, which is somewhat similar to VCG, to adopt First Price[16]. Each slot value is characterized by a click-through rate ("ctr", defined in a way similar to bids, as an ordinal), which is the probability that a user will click on an ad when displayed in this particular slot (experimentally, this depends on the ad position in the page and its size). The definition of the VCG for Search algorithm is displayed below.

```
Variable (bs0 : bids) (i : A).
Let bs := tsort bs0.
Let i' := idxa bs0 i.

Definition is_winner := i' < k'.          (* k = k'.+1 *)

Lemma slot_won_is_slot : minn i' last_slot < k.
Proof. exact: geq_minr. Qed.
Definition slot_won : slot := Ordinal slot_won_is_slot.

Definition externality (bs : bids) s :=                    (* bs_s ∗ (ctr_{s−1} − ctr_s) *)
  let j := slot_as_agent s in tnth bs j ∗ (' ctr_(slot_pred s) − (' ctr_s)).

Definition price :=   (* Price per impression (divide by ctr_s, for price per click):
                         Σ_{s=i'+1}^{k−1} bs_s ∗ (ctr_{s−1} − ctr_s) *)
```

[15] The `mech.v` library provides a machine-verified proof of the "well-known" fact that the price formula for VCG for Search is derivable from the one for General VCG.

[16] https://www.neowin.net/news/google-is-doing-away-with-the-second-price-auctioning-mechanism-on-adsense.

```
if i' < k then \sum_(s < k | i'.+1 <= s) externality bs s else 0.
```

The bids `bs0` of all bidders are sorted in a bid-decreasing manner, and the first `k` agents win the slot `slot_won` corresponding to their own sorting index, `i'` (slots are assumed to also be down-sorted). Note how this ordinal, of value here `i'`, is created by passing to the `Ordinal` constructor the proof `slot_won_is_won` of a lemma that states that the index `i'` is indeed a slot, i.e., a member of the ordinals of value less than `k`. The price paid by agent `i` for getting slot `i'` is the sum of the externalities of all the agents bidding less than `i` did; they are computed using bid-weighted differences of ctrs between adjacent slots.

As before, VCG for Search can be framed as an instance of `mech.type`, where the outcome domain is an `n`-tuple of structures in `R`, each of which stating whether the agent won, and if so, at what price and what:

```
Structure R := Result {awins : bool; price : nat; what : slot}.
Definition O : Type := n.-tuple R.

Definition m :=
  mech.new (fun bs ⇒ map_tuple (fun a ⇒ Result (is_winner bs a) (price bs a) (slot_won bs a))
                               (agent.agents n)).

Definition A := bid.
Definition v : agent.type n → A := true_value.

Definition p : prefs.type m :=
  prefs.new v
            (fun i (o : mech.O m) ⇒ let r := tnth o i in
                                    if awins r then v i * 'ctr_(what r) - price r else 0)
            v.
```

6.3 Refinement Framework

The `mech.v` library provides an abstract relational framework to help transfer property proofs from one mechanism to another, as long as they are properly related to each other. This has been used to derive in a very elegant manner the truthfulness of VCG for Search from the one existing for General VCG. To relate mechanisms `m2` and `m1`, of respective types `@mech.type A2 n` and `@mech.type A1 n`, one needs first to provide the following key elements.

- `Ra`, a relation between the action domains, of type[17] `agent → A1 → A2 → Prop`;
- `Ro`, a relation between the outcome domains, of type `O1 → O2 → Prop`, where `Oi` is a shorthand for `mech.O mi`;
- a "refinement" lemma, `MR` that states that `m1` and `m2` send related inputs (i.e., actions) to related outputs (i.e., outcomes);
- a "compatibility" lemma, `RelFP`, for utilities, i.e., a proposition that expresses that, if two outcomes are related, then the corresponding utilities also are too (using, here, equality).

An important special case is detailed in `mech.v`, i.e., when one is able to derive the properties of `m2` from the ones of `m1`. In this case, the following additional elements are also required:

- a mapping function `fR`, to specify how to compute an action `a1` for `m1` given one, `a2`, for `m2`;

[17] `Prop` is the name of logical properties in Coq.

- given preferences `p1` and `p2` for, resp., `m1` and `m2`, in particular the true value strategies `v1` and `v2`, two lemmas, the first one, `fRP`, ensuring that `fR i a2` is related to `a2` by `Ra`, and the second, that `v2 i` is mapped to `v1 i` by `fR`.

Given all these elements, the proof of truthfulness implication, `truthful p1 → truthful p2`, is provided in `mech.v`.

6.4 VCG Use Case

The `mech.v` provides the definitions and lemmas for all the elements required by the refinement formalism described above. We give a feel for how this applies to VCG via a few examples of how these are stated; identifiers annotated with the suffix "1" correspond to General VCG (mechanism `m1`), while those without suffix relate to VCG for Search (mechanism `m2`).

We build the action domain `A1` for General VCG upon VCG for Search elements: `A1 = bidding O'`, where `O'` is defined (definition omitted) as the domain of `k`-tuples of winning agents selected by VCG for Search. Similarly, one needs to define `O1` out of `O' : O1 = (n.-tuple nat * O')%type`. An element of `O1` is a pair gathering the tuple of the agents' prices according to VCG for Search and the tuple of winning agents, in `O'`. A good example to illustrate how these elements interact can be seen in how the true value strategy `v1` for General VCG can be derived from VCG for Search's `v`:

```
Definition v1 : agent.type n → A1 := fun i ⇒ [ffun o' : O' ⇒ v i * 'ctr_(slot_of i o')].
```

Elements of `A1` are finite functions, as described in Section 6.1. The multiplication by the ctr of the slot of `i` in `o'` is needed, since VCG for Search prices have to be divided by ctrs, to get prices per click.

The relation `Ra` between inputs is specified as follows:

```
Definition Ra i (a1 : A1) (a : A) := forall o' : O', a1 o' = a * 'ctr_(slot_of i o').
```

For all VCG for Search outputs `o'`, the bid value `(a1 o')` according to General VCG has to be equal to the VCG for Search bid, i.e., `a` multiplied by the ctr of the slot of `i` in `o` as before.

A last example is the relation `Ro` between the mechanisms' outputs, in `O1` for General VCG and `O` for VCG for Search (see Section 6.2). It is specified as follows:

```
Definition Ro (o1 : O1) (o : O) :=
  forall i, let r := tnth o i in
            let s := slot_of i o1.2 in
            awins r = (s != last_slot) ∧ (awins r → what r = s ∧ price r = tnth o1.1 i).
```

`Ro` states how the outcomes relate for each agent `i`: `r` is a structure in `R`, and `Ro` checks that the fields of `r` are compatible with the results computed by General VCG, in `o1`; note that the `last_slot` is the slot allocated to each agent that doesn't win in the VCG for Search algorithm.

Given these elements and the additional required lemma, VCG for Search can be proven truthful, via General VCG's truthfulness theorem[18].

7 CONCLUSION

We introduce `mech.v`, a new framework for specifying and proving properties of mechanisms using the Coq and SSReflect/mathcomp proof assistant. With almost than 6000 lines of Coq code, for both specifications and proofs, `mech.v` is now more than just a proof-of-concept (pun intended) development for formally verified mechanisms. Introducing generic mechanism definitions, actual

[18]This proof relies on a couple of sort-related side hypotheses, which could be proved away if deemed necessary. Also, we assume that the VCG optimal outcome is unique, leaving for future work the issue of handling the case when this is actually not the case. Recall though that VCG is agnostic on which choice is taken here, since the global welfare remains the same in all cases.

mechanism algorithms and proofs of their key properties based on powerful techniques (relational or instantiation-based), it handles seven common use cases focused on utility-based auction mechanisms, among which three instances of the key VCG mechanism, providing machine-verified proofs (or disproofs) of truthfulness for all of them (plus some additional lemmas). We believe `mech.v` provides thus a strong infrastructure from which sound, formally machine-verified existing and new mechanisms can be developed, helping increase the confidence the services they offer are well-founded.

This new framework can be extended in a variety of ways. First, additional notions related to deterministic mechanisms can be added to `mech.v`, e.g., stable matching or other notions of equilibria, as found in textbooks. Second, an other path for development is to tackle ordering-based (voting) or probabilistic mechanisms, which require an extension of the current arithmetics-based specifications to handle orderings and real numbers. Third, we would like to properly document this library to be more amenable to use as an "executable" textbook for economy-education purposes, following the "literate programming" movement; for this, we intend to build upon the jsCoq interactive web-based environment [15], which provides a flexible and powerful Coq-compatible infrastructure for the kind of multimedia material we envision, including text, images, programs and more. And finally, we are looking forward to getting input and feedback from the economics community on what is the best way to proceed with this kind of formal approach, now that the development of `mech.v` presented here has established the validity of such an approach.

ACKNOWLEDGMENTS

We thank Pierre Fleckinger (Mines Paris) for his insights on mechanism design.

REFERENCES

[1] G. Barthe, C. Fournet, B. Grégoire, P P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages*.

[2] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, Washington, DC, USA, 411–424. https://doi.org/10.1109/CSF.2014.36

[3] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 55–68. https://doi.org/10.1145/2676726.2677000

[4] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2016. Computer-aided Verification in Mechanism Design. In *Web and Internet Economics: 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings*, Yang Cai and Adrian Vetta (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–293. https://doi.org/10.1007/978-3-662-54110-4_20

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

[6] Valentin Blot, Louise Dubois de Prisque, Chantal Keller, and Pierre Vial. 2021. General Automation in Coq through Modular Transformations. In *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021 (EPTCS, Vol. 336)*, Chantal Keller and Mathias Fleury (Eds.). 24–39. https://doi.org/10.4204/EPTCS.336.3

[7] Florian Brandl, Felix Brandt, Manuel Eberl, and Christian Geist. 2018. Proving the Incompatibility of Efficiency and Strategyproofness via SMT Solving. *J. ACM* 65, 2, Article 6 (jan 2018), 28 pages. https://doi.org/10.1145/3125642

[8] Simina Brânzei and Ariel D. Procaccia. 2015. Verifiably Truthful Mechanisms. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science* (Rehovot, Israel) *(ITCS '15)*. Association for Computing Machinery, New York, NY, USA, 297?306. https://doi.org/10.1145/2688073.2688098

[9] Marco B Caminati, Manfred Kerber, Christoph Lange, and Colin Rowat. 2015. Sound Auction Specification and Implementation. 547–564. https://doi.org/10.1145/2764468.2764511

[10] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (sep 2010), 1157–1210.

[11] Vincent Conitzer and Tuomas Sandholm. 2002. Complexity of mechanism design. Morgan Kaufmann Publishers Inc., 103–110.

[12] Th. Coquand and G. Huet. 1988. The Calculus of Constructions. *Information and Computation* 76 (1988).

[13] Thierry Coquand, Gérard Huet, and Christine Paulin-Mohring. 2015. Notes on the prehistory of Coq. https://github.com/coq/coq/blob/da4c6c4022625b113b7df4a61c93ec351a6d194b/dev/doc/README-V1-V5.asciidoc

[14] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

[15] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2017. jsCoq: Towards Hybrid Theorem Proving Interfaces. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016 (Electronic Proceedings in Theoretical Computer Science, Vol. 239)*, Serge Autexier and Pedro Quaresma (Eds.). Open Publishing Association, 15–27. https://doi.org/10.4204/EPTCS.239.2

[16] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[17] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving (ITP)*. 163–179. https://doi.org/10.1007/978-3-642-39634-2_14

[18] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2008. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. https://hal.inria.fr/inria-00258384

[19] Jason D Hartline, Robert Kleinberg, and Azarakhsh Malekian. 2011. Bayesian incentive compatibility via matchings. SIAM, 734–747.

[20] Wesley H. Holliday, Chase Norman, and Eric Pacuit. 2021. Voting Theory in the Lean Theorem Prover. In *Logic, Rationality, and Interaction*, Sujata Ghosh and Thomas Icard (Eds.). Springer International Publishing, Cham, 111–127.

[21] Florian Kammüller, Manfred Kerber, and Christian W. Probst. 2017. Insider Threats and Auctions: Formalization, Mechanized Proof, and Code Generation. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 8, 1 (2017), 44–78. https://doi.org/10.22667/JOWUA.2017.03.31.044

[22] Manfred Kerber, Christoph Lange, and Colin Rowat. 2016. An Introduction to Mechanized Reasoning. *CoRR* abs/1603.02478 (2016). http://arxiv.org/abs/1603.02478

[23] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[24] Shengwu Li. [n.d.]. Obviously Strategy-Proof Mechanisms. *SSRN Electronic Journal* ([n. d.]). https://doi.org/10.2139/ssrn.2560028

[25] Ahuva Mu'alem and Noam Nisan. 2008. Truthful approximation mechanisms for restricted combinatorial auctions. 64, 2 (2008), 612–631.

[26] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. 2007. *Algorithmic game theory*. Cambridge University Press.

[27] Christine Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 328–345.

[28] Tuomas Sandholm. 2003. Automated Mechanism Design: A New Application Area for Search Algorithms. In *Principles and Practice of Constraint Programming – CP 2003*, Francesca Rossi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–36.

[29] The Coq Development Team. 2022. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.5846982