Formalizing constructive game semantics for affine logic: A strategy for automated proof search

Harshikaa Agrawal

Delhi Public School R. K. Puram

Abstract

One of our next technological frontiers is the automation of proof search. Proof assistants like Coq enable this vision by checking the work of proof search algorithms to guarantee correctness. However, currently proof search algorithms rely upon statistics and do not use mathematical abstractions to reduce computational load.

There is existing research to show isomorphisms between theorems and games, and proofs and winning strategies. This abstraction could significantly aid the automation of proof search, especially given the corpus of work in the automation of finding winning strategies to games.

We build a novel proven-correct library that allows us to use the abstraction of games for proof search. We provide the mathematics to map sequents in affine logic to two player games inside Coq.

In the paper, we present a language to express affine logic to Coq and show that the language is consistent, as in Kurt Gödel or model theoretic consistency. We formalized games, and make design choices so that the mapping of game semantics to affine logic is possible in the constraints of type theory and functional programming; so that the mapping is computable and not just provable, which is important because otherwise, we can't use strategies to *find* proofs; and so that the structure is highly reusable in proving high powered results that utilize games for automated proof search.

We also present, representative examples of formalizations of high powered theorems using the library, and demonstrate the formalization of games in the library, using the example of tic-tac-toe.

Contents

1 Introduction

1.1 Opportunity

Proof Assistants, which are also known as Interactive Theorem Provers, promise to automate mathematics and unleash an era of meteoric progress in the field.

Proof assistants are used to guarantee that proof ideas for a theorem are correct. When Vladimir Voevodsky's Fields Medal winning work was invalidated by a counterexample, Voevodsky came to the view that proof assistants are necessary for verifying the correctness of intricate and nuanced proof ideas. Even after spending several years carefully studying his original proof with the counterexample in hand, he was not able to identify exactly where his error was, and instead just worked around the counterexample by adjusting his theorem statements.

One of the largest successful verification efforts is the formalization of the Feit–Thompson Theorem [Gon+13] which classifies all finite simple groups of odd order. The paper proof is about 250 pages long, but the formalization effort took a group of 15 researchers from Microsoft Research and Inria 6 years and 170,000 lines of code comprising 15,000 definitions and 4,300 theorems [Kni12]. So guaranteeing the correctness of proof ideas is an enormous undertaking!

The grand dream is to automate proof search by training algorithms that can find novel proofs of theorems to glean more insight from the theorem, and to search for proofs of conjectures, thus finding entirely new theorems. However, pages and pages of computer generated code is not easy to read, and thus there is no way of knowing if our algorithms are giving us beautiful insights that are beyond human creativity or just giving us incomprehensible nonsense. But, if the proof search algorithms work inside a proof assistant, we know that whatever theorem statements are stated to have been proven have actually been proven.

Needless to say, training a proof search algorithm is quite ambitious. Much less one that is complex enough to have required 90 years of researcher time. But, with the aid of proof assistants, there is a path to doing so! In the field of automation through software, there is a new Herculean task that promises a glorious bounty of knowledge.

1.2 Our Idea

Current work is geared at using deep learning models to write lots of lines of code inside a proof assistant, and relying on the proof assistant to reject incorrect code, thus creating a training signal as in Loos et al. [Loo+17] and Kaliszyk et al. [Kal+18]. This methodology is very computationally intensive and inefficient. Moreover, it is enormously challenging to reduce computational load because there is no space to inject insights about mathematical abstraction, and a complete reliance on statistical patterns.

We propose a novel approach to automating proof search by injecting insights of mathematical abstraction in the form of games. Blass [Bla92] developed a mapping from affine logic formulas to games, and an isomorphism between proofs and winning strategies. Every affine logic formula is a theorem statement. Thus, the theorems represented by affine logic formulas can be mapped to a game, and finding a winning strategy for the game is isomorphic to finding a proof for the theorem.

This specific proposal leverages the enormous progress made in machine learning for finding winning strategies in games [Sil+16; Vin+19]. Coming up with new winning strategies and new proofs are both extremely difficult tasks, but we have a lot of progress in one that we can use for progress in the other.

1.3 Our contribution

We have built a library in the proof assistant Coq [Coq21] that provides the mathematical infrastructure required to translate affine logic sequents, which represent theorems, into games. Our library can be used to formalize high powered mathematical results to utilize winning strategies in two-player games as proofs. The library can be found at `https://github.com/harshikaaagrawal/game-semantics-for-affine-logic`.

In this paper, we present our formalization of affine logic and game semantics; explain the design choices we make so that the formalization can be utilized for our purpose of automated proof search; the seminal definition of our library that translates affine logic sequents into games; representative examples of theorems using our library to connect winning strategies to proofs of sequents; and a representative example of translating popular games for usability in our library.

This project is in the new tradition of libraries at the intersection of programming languages verification and mathematics, which include formalizations of the Feit–Thomson Theorem [Gon+13], Category Theory [GCS14], Homotopy Type Theory [Bau+17] The Four–Colour Theorem [Gon05; Gon08], and the proof of the Kepler Conjecture [Hal+15].

2 Project Background

In this section, we aim to describe in broad strokes our project specification and project constraints. The concepts underpinning our work are extensive fields, so we limit ourselves to providing only relevant context to understand our contribution.

2.1 Proof Assistants

Type theory based proof assistants, like Coq, work by allowing the user to code functional programs and validating that the programs type check. By virtue of the Curry- Howard Isomorphism, validating that the correctness of the types of a program is equivalent to checking that the proof represented by that program is correct.

computation	set theory	logic
type	set	proposition
term / program	element of a set	proof
eliminator / recursion		case analysis / induction
type of pairs	Cartesian product ($\times$)	conjunction ($\wedge$)
sum type ($+$)	disjoint union ($\sqcup$)	disjunction ($\vee$)
function type	set of functions	implication ($\to$)
unit type	singleton set	trivial truth
empty type	empty set ($\emptyset$)	falsehood
dependent function type (Π)		universal quantification ($\forall$)
dependent pair type (Σ)		existential quantification ($\exists$)

Table 1: The Curry–Howard Isomorphism

The Curry–Howard Isomorphism states that types are in isomorphic correspondence with theorem statements, and functional programs having those types are in isomorphic correspondence with the proofs of the theorems corresponding to those types. Hence, we are able to write code in functional programming languages in order to prove theorems.

Note that this means we are not able to use the bounty of techniques of object-oriented or imperative-style programming. This precludes us from using loops and mutations, which are replaced by recursion and higher-order functions.

2.2 Affine Logic

Affine logic is a substructural logic that extends the usability of classical logic. For instance, in classical logic we use $\wedge$ to connect multiple resources. Since the rules of classical logic say that T gives us $T \wedge T$ which gives us $T \wedge T \wedge T \wedge \ldots$ we are limited to using resources that are inexhaustible and don't need to be tracked. For example, if we reason about money with classical logic, then from a penny we can get penny $\wedge$ penny continuing indefinitely to obtain infinite money. Anyone who has ever had to manage money knows that this does not represent reality!

Since there are systems where we need to track resources, we chose to base our library in affine logic so that we may prove theorems about such systems. The following are the rules of affine logic. For a given rule, the assumptions are above the line and the conclusions are below the line, the context is to the left of the turnstile ($\vdash$) and the result is to the right of the turnstile. Latin variables such as A, B, and C stand in for *formulas*. The variables Δ and Δ' are multisets of formulas and form the resource context, while Γ is the set of formulas forming the logical context. A sequent is an expression such as $\Delta \vdash A$, read "Delta proves A".

$$\frac{}{A \vdash A}\ (\mathrm{id}_A)$$

$$\frac{\Delta \vdash A \qquad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ (\text{cut})$$

$$\frac{\Delta \vdash A}{\Delta, \Delta' \vdash A}\ (\text{weakening})$$

$$\frac{\Gamma, A; \Delta, A \vdash C}{\Gamma, A; \Delta \vdash C}\ (\text{logical context})$$

Name	Symbol	Left Rule	Right Rule
Tensor	$\otimes$	$\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ (\otimes_L)$	$\frac{\Delta \vdash A \qquad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B}\ (\otimes_R)$
With	&	$\frac{\Delta, A \vdash C}{\Delta, A \& B \vdash C}\ (\&_{L_1})$ $\frac{\Delta, B \vdash C}{\Delta, A \& B \vdash C}\ (\&_{L_2})$	$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \& B}\ (\&_R)$
One	1	$\frac{\Delta \vdash C}{\Delta, 1 \vdash C}\ (1_L)$	$\frac{}{\bullet \vdash 1}\ (1_R)$
Zero	0	$\frac{}{\Delta, 0 \vdash C}\ (0_L)$	N/A
Implication "lolli"	$\multimap$	$\frac{\Delta \vdash A \qquad \Delta', B \vdash C}{\Delta, \Delta', (A \multimap B) \vdash C}\ (\multimap_L)$	$\frac{\Delta, A \vdash B}{\Delta \vdash (A \multimap B)}\ (\multimap_R)$
"bang" "of course"	!	$\frac{\Gamma, A; \Delta \vdash C}{\Gamma; \Delta, !A \vdash C}\ (!_L)$	$\frac{\Gamma; \bullet \vdash A}{\Gamma; \bullet \vdash !A}\ (!_R)$

As we see in Table 1, the Curry–Howard Isomorphism maps to structural logic. Given that affine logic is a substructural logic, we cannot make use of the Curry–Howard Isomorphism to build proof assistants with native support for affine logic.

2.3 Modelling proofs through game semantics

The idea behind modelling proofs through game semantics is that we can model proof search as an adversarial process between a prover (player 1) and a disprover (player 2). A winning strategy for player 1 generates a proof for the theorem, symmetrically a winning strategy for player 2 generates a counterexample for the theorem, thus resulting in a proof for the negation of the theorem.

Since our hope is to use the library for leveraging machine learning algorithms for proof search, it behooves us to ensure that the winning strategies are computable. This requires that we avoid classical axioms, i.e. those axioms without any computable interpretation, when formulating winning strategies.

2.4 Summary

The following is a list of project specifications followed by the constraints they pose.

Using functional programming in the proof assistant gives us the power of the Curry–Howard Isomorphism, and restricts our usage of agile programming techniques like loops and mutations.

Using affine logic allows us to extend the scope of our project to systems using resources that need to be tracked, however, there cannot be native support for substructural logics in any type theory based proof assistant.

Modelling proofs through games allows us to use algorithms that find winning strategies in adversarial games from proof search, however since we want to use the computation of the winning strategy we cannot use classical axioms of mathematics to support our work.

3 Library Design

In this section, we present the following: our formalization of affine logic; proof that there exists a model theoretic model of affine logic, namely the booleans; our formalization of the definition of game from Blass [Bla92] with design changes to meet the constraints of type theory and functional programming; the construction of our seminal definition that maps logical sequents to games while ensuring computability of strategy and reusability of the definition.

Note that in proof assistants "definitions" and "theorems" are roughly the same. Any construction where we later need to know the details of how we performed the construction, we call a definition; whereas any construction whose details are irrelevant, we call a theorem. The specifics of these constructions are quite important, as they lend themselves to each other. We share specific code to provide an overview of the structure of our proofs.

The following are identifiers that appear in the Coq standard library that we shall use, in order of appearance:

- `seq X` is the type of finite sequences whose elements are of type `X`
- `map` applies a function to every element in a list and is defined as

```
map f [] := []
map f (x :: xs) := f x :: map f xs
```

- `foldr` applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, and is defined as

  ```
  foldr f z [x1, x2, ..., xn] == x1 ‘f‘ (x2 ‘f‘ ... (xn ‘f‘ z)...)
  ```

  or

  ```
  foldr f z [] := z
  foldr f z (x :: xs) := f x (foldr f z xs)
  ```

- `andb` is boolean conjunction
- `Fixpoint` operates on recursive data types and inductive data types to define functions by recursion with base cases and recursive cases.
- `match` “matches” input data type to the unique case of each constructor which allows defining a function similar to proof by case analysis.

3.1 Affine Logic

We want our library to be usable in two ways: first, for users to be able to write theorem statements and have our library output a game corresponding to the theorem that ML algorithms can try to find a winning strategy for; second, to take the strategy that the ML algorithm finds and turn this into a proof of the original theorem statement and validate that the proof is correct. We chose affine logic to express theorem statements.

In order to achieve this, we need to construct languages so that we can tell the proof assistant what the theorem statement is, and so that we talk about what it means to have a valid proof of a given theorem statement. Since our theorems are expressed in affine logic, we construct a language to express affine logic to the proof assistant, with the criteria that the only valid sentences in the language are the ones that correspond to valid proofs.

This work has to be done from scratch, because Coq does not and cannot have native support for affine logic, as detailed in the previous section. There are two things to define in affine logic: first, we define the syntax of affine logic formulas; second we define the proof rules for what it means for a sequent to be provable in affine logic.

3.1.1 Method

In the literature, previously, the standard was to build a language by writing Gödel codes where a sentence is encoded by indexing the n symbols. The formed sentence is written above the string of prime numbers in ascending order, and every prime number is raised to the exponent i which is the index of the symbol above it. Then these powers of primes are multiplied. In this way an isomorphism is established between the positive integers and sentences. The validity of the sentence is then defined via factorization into powers of primes. This method requires enormous precision in holding several details that do not pertain to application the language is being built for.

We instead build our language by describing the datatype of valid syntax trees. This method is more agile, simple, easy to use, easy to compute, and rests on just as firm mathematical foundations via the Curry–Howard Isomorphism. We use inductive definitions to define the language of valid syntax trees in Coq. An inductive definition is a collection of constructors that specify how to build the corresponding datatype recursively. We define our language as follows:

Definition 1 (`syntax`)**.** The inductive `syntax` defines affine logic connectors in Coq.

These constructors correspond to the ability to use variables (`Var`), and to the affine logic connectives 0, 1, $\otimes$, &, $\multimap$, !. The following is the list of constructors with their arguments.

```
Inductive syntax {var : Type} :=
| Var (v : var)
| Zero (* 0 *)
| One  (* 1 *)
| Tensor (Left : syntax) (Right : syntax) (* ⊗ *)
| With   (Left : syntax) (Right : syntax) (* & *)
| Implication (Left : syntax) (Right : syntax) (* ⊸ *)
| Bang (Right : syntax) (* ! *)
.

Notation "A * B" := (Tensor A B) : syntax_scope.
Notation "A && B" := (With A B) : syntax_scope.
Notation "A '-o' B" := (Implication A B) (at level 99) : syntax_scope.
Notation "! A" := (Bang A) (at level 9, format "! A") : syntax_scope.
Notation "0" := Zero : syntax_scope.
Notation "1" := One : syntax_scope.
```

Note that comments in the Coq language occur between `(*` and `*)`.

Definition 2 (`provable`)**.** The inductive `provable` contains the respective left and right rules of affine logic connectors as mentioned in the previous section. For a given rule, the assumptions are above the line and the conclusions are below the line, the context is to the left of the turnstile and the result is to the right of the turnstile. Provable is an inductive type family with two arguments: context of type seq syntax and result of type syntax. Here, type `seq X` is the type of finite sequences whose elements are of type `X`.

The following is the list of proof rules in Coq with their corresponding symbolic representation above each proof rule.

```
Inductive provable : seq syntax -> syntax -> Type :=
| permute_context Ctx1 Ctx2 P

(a : perm_eq Ctx1 Ctx2)     (b : Ctx1 ||- P)
(*-------------------------------------------*)
:                  Ctx2 ||- P

(*
```

$$\frac{}{A \vdash A}\ (\mathrm{id}_A)$$

```
*)

| idA A

(*-----------------------------*)
:          [:: A] ||- A
(*
```

$$\frac{\Delta \vdash A}{\Delta, \Delta' \vdash A}\ (\text{weakening})$$

```
*)

| weakening Ctx1 Ctx2 A

          (a : Ctx1 ||- A)
(*-----------------------------*)
:          Ctx1 ++ Ctx2 ||- A
```

```
(*
```

$$\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ (\otimes_L)$$

```
*)

| tensor_left Ctx A B P

    (a : A :: B :: Ctx ||- P)
(*------------------------------*)
:     A * B :: Ctx ||- P
(*
```

$$\frac{\Delta \vdash A \qquad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B}\ (\otimes_R)$$

```
*)

| tensor_right Ctx1 Ctx2 A B

 (a : Ctx1 ||- A)    (b : Ctx2 ||- B)
(*------------------------------------*)
:     Ctx1 ++ Ctx2 ||- A * B
(*
```

$$\frac{\Delta, A \vdash C}{\Delta, A \& B \vdash C}\ (\&_{L_1})$$

```
*)

| with_left_1 Ctx A B P

      (a : A :: Ctx ||- P)
(*------------------------------*)
:      A && B :: Ctx ||- P
```

```
(*
```

$$\frac{\Delta, B \vdash C}{\Delta, A\&B \vdash C}\ (\&_{L_2})$$

```
*)

| with_left_2 Ctx A B P

      (a : B :: Ctx ||- P)
(*------------------------------*)
:      A && B :: Ctx ||- P
(*
```

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A\&B}\ (\&_R)$$

```
*)

| with_right Ctx A B

(a : Ctx ||- A)    (b : Ctx ||- B)
(*------------------------------*)
:       Ctx ||- A && B
(*
```

$$\frac{\Delta \vdash C}{\Delta, 1 \vdash C}\ (1_L)$$

```
*)

| one_left Ctx P

          (a : Ctx ||- P)
(*------------------------------*)
:         1 :: Ctx ||- P
```

```
(*
```

$$\frac{}{\bullet \vdash 1}\ (1_R)$$

```
*)

| one_right

(*------------------------------*)
:           [::] ||- 1
(*
```

$$\frac{}{\Delta, 0 \vdash C}\ (0_L)$$

```
*)

| zero_left Ctx P


(*------------------------------*)
:     0 :: Ctx ||- P
(*
```

$$\frac{\Delta \vdash A \qquad \Delta', B \vdash C}{\Delta, \Delta', (A \multimap B) \vdash C}\ (\multimap_L)$$

```
*)

| implication_left Ctx1 Ctx2 A B P

      (a : Ctx1 ||- A)    (b : B :: Ctx2 ||- P)
(*--------------------------------------------------*)
:         (A -o B) :: Ctx1 ++ Ctx2 ||- P
```

```
(*
```

$$\frac{\Delta, A \vdash B}{\Delta \vdash (A \multimap B)}\ (\multimap_R)$$

```
*)

| implication_right Ctx A B

     (a : A :: Ctx ||- B)
(*------------------------------*)
:      Ctx ||- (A -o B)

where "Ctx ||- A" := (provable Ctx%syntax A).
```

Design Choice. In the symbolic version, contexts are implicitly lists modulo ordering of elements. Coq does not have lists modulo ordering nor multisets. To get around this we can either symbolically adjust all proof rules so that they mean the same thing when using ordered lists instead of multisets, or we can add another proof rule that allows us to freely change the order of the context. We do the latter, and call it `permute_context`. So instead of proving it as a theorem, we build `permute_context` into our datatype as a constructor.

3.2 Model of affine logic

In subsubsection 3.1.1 we defined syntax and provable to formalize affine logic. In this section, we will show that our formalization is correct, i.e. prove consistency of the formalization.

Model Theory is the study of relationships between different formal theories, where a formal theory is the set of valid sentences in a defined language that expresses a mathematical structure. Note that these relationships are not necessarily isomorphisms. A formal theory, like our formalization of affine logic above, can be validated as consistent by finding a model for it.

Theorem 1 (Consistency in English)**.** *From a proof of 0, we can derive absurdity.*

Proof. For every proof of 0 within our formalization, if we can show that this corresponds to an element of the empty set in the model, i.e. an object that does not exist, we will have shown that there are no proofs of absurdity in our formal theory because there are no elements of the empty set. □

This theorem relies on the existence of a model for the formalization, so we prove the theorem that the Booleans are a model of the formalization of affine logic.

Theorem 2 (Model Existence in English)**.** *The Booleans are a model of affine logic.*

Proof. Define a mapping from affine logic formulas to the booleans by sending 1 to True, 0 to False, the tensor of two formulas to the boolean conjunction, the with of two formulas to the boolean conjunction, the implication of two formulas $A \multimap B$ to True whenever B is mapped to True or A is mapped to False, and the Bang of a formula to the mapping of that formula. We denote the mapping of the formula f as $[\![f]\!]$.

Define a mapping from contexts to booleans by sending the list of formulas to the boolean conjunction of the boolean corresponding to each formula. The empty context is mapped to True. We denote the mapping of the context Δ by $[\![\Delta]\!]$.

By induction on the proof rules, we have that if $\Delta \vdash P$ and $[\![\Delta]\!]$ is True, then $[\![P]\!]$ is also True. □

We formalize the results in the proof assistant as follows.

Theorem 1 (Consistency in Coq)**.** `([::] ||- 0) -> False`

```
Proof. intro H; pose proof (model H); simpl in *; congruence. Qed.
```

Theorem 2 (Model Existence in Coq)**.**

```
Fixpoint truth_value (x : syntax) : bool
  := match x with
     | One  => true
     | Zero => false
     | Tensor l r => truth_value l && truth_value r
     | With l r   => truth_value l && truth_value r
     | Implication l r => if truth_value l then truth_value r else true
     | Bang r => truth_value r
     end.

Definition truth_value_of_context (ctx : seq syntax) : bool
  := foldr andb true (map truth_value ctx).
```

```
Theorem model : forall {Ctx P},
  (Ctx ||- P) -> truth_value_of_context Ctx = true -> truth_value P = true.

Proof.
  intros Ctx P H T.
  induction H;
  rewrite -> ?truth_value_of_context_cat,
    -> ?truth_value_of_context_cons,
    -> ?truth_value_of_context_nil,
    -> ?Bool.andb_true_r in *.
  all: simpl in *; auto.
  all: rewrite -> ?truth_value_of_context_cat,
         -> ?truth_value_of_context_cons,
         -> ?truth_value_of_context_nil,
         -> ?Bool.andb_true_r in *.
  all: simpl in *; auto.
  {
    apply truth_value_of_context_perm in a.
    rewrite -a // in T.
  }
  {
    rewrite -> ?Bool.andb_assoc in *.
    assert (X : truth_value A = true) by auto.
    rewrite -> X in *.
    auto.
  }
  {
    rewrite T in IHprovable.
    Search (_ && true).
    rewrite -> Bool.andb_true_r in *.
    destruct (truth_value A); auto.
  }
Qed.
```

Notice that the theorem statement already requires definitions, since the construction of the theorem statement is used in the proof.

3.3 Formal games

In this section we formalize games as in Blass [Bla92] in the proof assistant Coq. Our work is in finding elegant abstractions upon the original work that make it computable in a functional landscape. We provide a summary of key design choices that express the complexity and reasoning for this task.

Blass [Bla92, p. 187] says:

> A (strict) *game* (or debate or dialogue) between two players, P and O, consists of the following data: a set M of possible moves, a specification of P or O as the player who moves first, and a set G of infinite sequences of members of M, namely the plays won by P. Thus, a game is an ordered triple (M, s, G), where $s \in \{P, O\}$ and $G \subseteq {}^{\omega}M$, but we often write simply G. We write $\bar{s}$ for the nonstarting player, the member of $\{P, O\}$ other than s.

In Coq, we formalize this as

```
Inductive player := player_O | player_P.


Record game
:= { possible_move : Type
   ; first_player : player
   ; play_won_by_P : Stream possible_move -> Prop
   ; play_won_by_O : Stream possible_move -> Prop
   ; no_duplicate_winner : forall all_moves,
     play_won_by_P all_moves -> play_won_by_O all_moves -> False
   }.
```

Design Choice (Basic Design Choices)**.** `Type` is the rough analogue of the collection of sets, a `Record` is how we construct labelled tuples, `Stream` is Coq standard library version of infinite sequences.

Design Choice. Use of `Prop` instead of `bool` is introduced for specifying the subset of plays that are won by P. This gives us the power to depend on arbitrarily many moves in order to determine which player wins. If this were not made possible then games like chess would not be possible to formalize, given that we don't know ahead of time the longest possible game of chess. Furthermore, some of the infinite games that Blass [Bla92] takes advantage of to encode constructions in affine logic would be inaccessible to us.

Design Choice. Even though proofs are finite objects, we allow infinite games using `Stream` instead of `seq`, and handle infinite sequences as they come up. The problem with finite sequences is that some sequences don't represent complete plays of a game, and some games by design prevent the computation of a winner. For example in the game where the only moves are to "continue play" or "to withdraw", the strategy of always "continue play" is not a guaranteed winning strategy even though it is the only strategy that could yield a win.

Design Choice. We separate the specification on plays won by P from the specification of plays won by O. This allows us to avoid the usage of classical axioms, and it makes the double negation of a game "judgmentally equal" to the game, so that making use of the fact that they are the same does not require additional proof. To achieve this, we add the field `no_duplicate_winner`.

Blass [Bla92, p. 187] says:

> A *position* p in a game is a finite sequence of moves. It is a position with the starting player $s \in \{P, O\}$ to move if its length is even; otherwise, it is a position with the other player $\bar{s}$ to move. A *strategy* for either player is a function σ into M from the set of all positions where that player is to move. A player follows strategy σ in a play $x \in {}^{\omega}M$ if, for every position p in x (i.e., finite initial segment p of x) where that player is to move, the next term after p in x is $\sigma(p)$. A strategy σ for P (respectively, O) is a winning strategy if all (respectively, none) of the plays in which P (respectively, O) follows σ are in G.

In Coq, we formalize this as

```
Definition position (g : game) : Type
  := seq (possible_move g).

Definition play (g : game) : Type
  := Stream (possible_move g).

Definition next_player {g} (p : position g) : player
  := if Nat.even (List.length p)
     then first_player g
     else other_player (first_player g).

Definition strategy g (p : player) : Type
  := forall (pos : position g), possible_move g.
```

```
Definition player_follows_strategy
 {g} (p : player) (s : strategy g p) (all_moves : play g) : Prop
  := forall n : nat, let initial_segment := Streams.firstn all_moves n in
     forall h : next_player initial_segment == p,
     Streams.nth all_moves (n.+1) = s initial_segment.

Definition play_won_by {g} (p : player) (all_moves : play g) : Prop
  := match p with
     | player_P => play_won_by_P g all_moves
     | player_O => play_won_by_O g all_moves
     end.

Definition winning_strategy {g} {p : player} (s : strategy g p) : Prop
  := forall all_moves : play g,
     player_follows_strategy p s all_moves -> play_won_by p all_moves.
```

Design Choice. We choose to define strategy for player p by mapping from the set of all positions rather than from the set where p is the next player to move. This allows us to avoid needlessly passing around and manipulating proof objects when we are not planning on changing the meaning of any theorems.

Design Choice. In defining what it means for a player to follow a strategy, we quantify over the length of the segment instead of the initial segment itself. This allows for cleaner code design and higher usability.

3.4 Seminal definition

In this section we tie together our formalizations of affine logic and formal games to prove a mapping from affine logic sequents to game semantics. This mapping is the seminal definition of our library.

Theorem 3 (Seminal Definition in English)**.** *For every affine logic sequent, given a mapping of free variables in the sequent to games, we can map the entire sequent to a game.*

Proof. Map every connective symbol to its corresponding game, negate all games that came from formulas in the context, and join all resulting games with the game connective corresponding to $\bar{\otimes}$ (also denoted $⅋$, pronounced "par"). $\square$

Theorem 3 (Seminal Definition in Coq)**.**

```
forall {var : Type} (var_to_game : var -> strict.game)
       (context : seq (@affine.syntax var))
       (result : @affine.syntax var),
  strict.game
```

Proof.

```
Context {var : Type} (var_to_game : var -> strict.game).
Fixpoint syntax_to_game (s : @affine.syntax var) : strict.game :=
match s with
 | affine.Var v => var_to_game v
 | affine.Zero => strict.bottom
 | affine.One => strict.top
 | affine.Tensor Left Right => syntax_to_game Left ⊗ syntax_to_game Right
end.

Definition sequent_to_game (context : seq (@affine.syntax var))
  (result : @affine.syntax var) : strict.game :=
  foldl (fun g1 g2 => (g1) ~* (g2))
        (syntax_to_game result)
        (map (fun g => ~syntax_to_game g) context).
```

□

In order to construct the above proof, we constructed the top game, the bottom game, the negation game, the par game, and the tensor game. To ground this construction, we constructed a translation from relaxed games to strict games. This enterprise spanned approximately 700 lines of code and 26 definitions (which are theorems) and lemmas. Much of the effort was devoted towards proving that these games have no duplicate winner, i.e. proving equivalent characterizations of the opponent winning the game. Since this construction is too extensive to be pasted straight, we will now explain the key insights and design choices.

First, here are the snippets of code that illuminate the structure of what needed to be said about different games.

```
Definition top : game.
refine {| possible_move := unit
        ; first_player := player_P
        ; play_won_by_P all_moves := True
        ; play_won_by_O all_moves := False
        ; no_duplicate_winner all_moves P_wins O_wins := O_wins
|}.
Defined.

Definition bottom : game.
refine {| possible_move := unit
        ; first_player := player_O
        ; play_won_by_P all_moves := False
        ; play_won_by_O all_moves := True
        ; no_duplicate_winner all_moves P_wins O_wins := P_wins
|}.
Defined.

Definition negation (g : game) : game.
refine {| possible_move := possible_move g
        ; first_player := other_player (first_player g)
        ; play_won_by_P all_moves := play_won_by_O g all_moves
        ; play_won_by_O all_moves := play_won_by_P g all_moves
        ; no_duplicate_winner all_moves P_winner O_winner
          := no_duplicate_winner g all_moves O_winner P_winner |}.
Defined.
Notation "~ g" := (negation g) : game_scope.

Definition strict_par (g1 : strict.game) (g2 : strict.game) : strict.game
  := ~((~g1) ⊗ (~g2)).
Infix "~*" := strict_par (at level 40, left associativity) : game_scope.



Definition strict_tensor (g1 : strict.game) (g2 : strict.game) : strict.game
  := to_strict (tensor g1 g2).
Infix "⊗" := strict_tensor (at level 40, left associativity) : game_scope.
```

Second, we explain relaxed games which Blass [Bla92] used to define tensor games. The most complex formalization was of the relaxed tensor game which was used by us to define the strict tensor game.

Blass [Bla92, p. 187] says:

> It will often be convenient to describe games in a way that does not adhere to all the conventions built into our definition of strict

games. In a relaxed game, we do not assume that the players move alternately; rather, the rules specify, for each position, who is to move next. We require, for technical reasons, that a player has only finitely many consecutive moves. Clearly, any game of this sort can be regarded as a strict game by declaring a block of consecutive moves by the same player in the relaxed game to be a single move in the strict game. This will, of course, require that the set M of possible moves in the strict game is a bit more complex than in the relaxed game.

In Coq, we formalize this as:

```
Module relaxed.
Record game
:= { possible_move : Type
   ; first_player : player
   ; play_won_by_P : Stream possible_move -> Prop
   ; play_won_by_O : Stream possible_move -> Prop
   ; no_duplicate_winner : forall all_moves,
       play_won_by_P all_moves -> play_won_by_O all_moves -> False
   ; next_player : seq possible_move -> player
   ; next_move_is_valid : seq possible_move -> possible_move -> bool}.
```

We use relaxed games as a stepping stone for building tensor games so that we are able to factor out the conversion to strict games. Though Blass [Bla92] is able to do this in 1 sentence, Coq does not allow us to declare truth assertions as "clear" and leave the details as an exercise to the reader. So, we embark upon a journey spanning 200 lines of code. Here, we also elide the proofs and write the definitions and lemmas to illuminate the key ideas.

```
Definition out_of_turn_move {g}
  (moves : seq (possible_move g)) (actual_current_player : player) : bool
   := actual_current_player != next_player g moves.


Definition strict_moves_to_relaxed_moves_with_player {g : game}
  (moves : seq (possible_move g * seq (possible_move g)))
    : seq (possible_move g * player)
  := flatten (map (fun '(n, (s0, s)) =>
       let p := if Nat.even n
                then first_player g
                else other_player (first_player g) in
       map (fun m => (m, p)) (s0 :: s))
      (enumerate moves)).

Fixpoint first_invalid_move_of_relaxed_moves_with_player_helper {g : game}
  (moves_so_far : seq (possible_move g))
  (remaining_moves : seq (possible_move g * player))
   : option player :=
  match remaining_moves with
  | [::] => None
  | (move, p) :: moves =>
     if negb (out_of_turn_move moves_so_far p)
         && next_move_is_valid g moves_so_far move
     then first_invalid_move_of_relaxed_moves_with_player_helper
       (rcons moves_so_far move) moves
     else Some p
  end.
Definition first_invalid_move_of_relaxed_moves_with_player {g : game}
  (relaxed_moves_with_player : seq (possible_move g * player)) : option player
  := first_invalid_move_of_relaxed_moves_with_player_helper
       [::] relaxed_moves_with_player.

Definition first_invalid_move {g : game}
  (moves : seq (possible_move g * seq (possible_move g))) : option player
 := first_invalid_move_of_relaxed_moves_with_player
      (strict_moves_to_relaxed_moves_with_player moves).

Lemma first_invalid_move_firstn_monotone_lt {g : game} all_moves n m
 : (n < m)%coq_nat
    -> @first_invalid_move g (Streams.firstn all_moves n) <> None
    -> first_invalid_move (Streams.firstn all_moves m)
       = first_invalid_move (Streams.firstn all_moves n).

Definition to_strict (g : game) : strict.game.
refine {| strict.possible_move := possible_move g *
  seq (possible_move g);
    strict.first_player := first_player g
   ; strict.play_won_by_P all_moves
      := (exists n : nat,
            first_invalid_move (Streams.firstn all_moves n)
              == Some player_O)
   \/ (~(exists n : nat,
          first_invalid_move (Streams.firstn all_moves n)
            == Some player_P)
       /\ play_won_by_P g (Streams.flatten all_moves)
    )
   ; strict.play_won_by_O all_moves
      := (exists n : nat,
           first_invalid_move (Streams.firstn all_moves n)
             == Some player_P)
   \/ (~(exists n : nat,
           first_invalid_move (Streams.firstn all_moves n)
             == Some player_O)
       /\ play_won_by_O g (Streams.flatten all_moves)
    )
|}.
```

Design Choice. In the strict game, every time it is the turn of player p, p must make at least one move. We encode this by declaring that the set M of moves is `possible_move g * seq (possible_move g)`. If this wasn't the case then the formalization would break.

Design Choice. We define games won by P and O separately despite being the case that seemingly games not won by P are won by O. We do this as a consequence of separating the descriptions of the games won by P and the games won by O. The reason we don't define one as the negation of the other is that it serves as useful sanity check on our constructions to define them symmetrically.

Design Choice. To define games lost by invalid moves we need to be precise with how we handle infinite and finite sequences. We indirect through finite sequences to make the definitions more computational, however this required proving monotonicity conditions that say that once we've found an answer on some finite prefix, that answer does not change as a result of making more moves after the invalid move.

The tensor game $G_1 \otimes G_2$ comprises two game setups where P has to win both G_1, G_2 and O only has to win one of G_1, G_2 to win the tensor game. There are additional quirks like O gets to decide which game to make a move in, and P must make a move in the game that O just made a move in. The only time P gets the choice of which game to play is if the first turn in both games are the turns of P. The tensor game models the tensor product.

```
Definition tensor_next_player (p1 : player) (p2 : player) : player
  := match p1, p2 with
| player_P, player_P => player_P
| player_P, player_O => player_P
| player_O, player_P => player_P
| player_O, player_O => player_O
end.



Definition left_game_not_abandoned {g1 g2}
  (all_moves : Stream (strict.possible_move g1 + strict.possible_move g2))
    : Prop := forall n : nat, exists m : nat, m >= n
  /\ exists left_move, Streams.nth all_moves m = inl left_move.

Definition right_game_not_abandoned {g1 g2}
  (all_moves : Stream (strict.possible_move g1 + strict.possible_move g2))
    : Prop := forall n : nat, exists m : nat, m >= n
  /\ exists right_move, Streams.nth all_moves m = inr right_move.

Lemma left_game_not_abandoned_tl {g1 g2 all_moves}
: @left_game_not_abandoned g1 g2 all_moves
  -> @left_game_not_abandoned g1 g2 (Streams.tl all_moves).

Lemma right_game_not_abandoned_tl {g1 g2 all_moves}
: @right_game_not_abandoned g1 g2 all_moves
  -> @right_game_not_abandoned g1 g2 (Streams.tl all_moves).

CoInductive moves_compatible_with {g1 g2}
  (left_moves : strict.play g1) (right_moves : strict.play g2)
  (all_moves : Stream (strict.possible_move g1 + strict.possible_move g2))
```

```
    : Prop :=
| moves_compatible_with_left
 : Streams.hd all_moves = inl (Streams.hd left_moves)
   -> moves_compatible_with
       (Streams.tl left_moves) right_moves (Streams.tl all_moves)
   -> moves_compatible_with left_moves right_moves all_moves
| moves_compatible_with_right
 : Streams.hd all_moves = inr (Streams.hd right_moves)
   -> moves_compatible_with left_moves
       (Streams.tl right_moves) (Streams.tl all_moves)
   -> moves_compatible_with left_moves right_moves all_moves.

Lemma left_game_not_abandoned_compatible_with_same_game_ex
  {g1 g2} {left_moves left_moves' : strict.play g1}
: (exists (right_moves right_moves' : strict.play g2) all_moves,
   moves_compatible_with left_moves right_moves all_moves
   /\ moves_compatible_with left_moves' right_moves' all_moves
   /\ left_game_not_abandoned all_moves)
  -> Streams.EqSt left_moves left_moves'.

Lemma left_game_not_abandoned_compatible_with_same_game
  {g1 g2} {left_moves left_moves' : strict.play g1}
    {right_moves right_moves' : strict.play g2} {all_moves}
: moves_compatible_with left_moves right_moves all_moves
  -> moves_compatible_with left_moves' right_moves' all_moves
  -> left_game_not_abandoned all_moves
  -> Streams.EqSt left_moves left_moves'.

Lemma right_game_not_abandoned_compatible_with_same_game_ex
  {g1 g2} {right_moves right_moves' : strict.play g2}
: (exists (left_moves left_moves' : strict.play g1) all_moves,
   moves_compatible_with left_moves right_moves all_moves
   /\ moves_compatible_with left_moves' right_moves' all_moves
   /\ right_game_not_abandoned all_moves)
  -> Streams.EqSt right_moves right_moves'.
```

```
Lemma right_game_not_abandoned_compatible_with_same_game
  {g1 g2} {left_moves left_moves' : strict.play g1}
    {right_moves right_moves' : strict.play g2}
  {all_moves}
: moves_compatible_with left_moves right_moves all_moves
  -> moves_compatible_with left_moves' right_moves' all_moves
  -> right_game_not_abandoned all_moves
  -> Streams.EqSt right_moves right_moves'.

Definition tensor (g1 : strict.game) (g2 : strict.game) : game.
refine {|
   possible_move := strict.possible_move g1 + strict.possible_move g2
 ; first_player
   := tensor_next_player (strict.first_player g1) (strict.first_player g2)
 ; next_player moves_so_far
   := let (moves_so_far1, moves_so_far2)
          := partition_map (fun move => move) moves_so_far in
      tensor_next_player (strict.next_player moves_so_far1)
                         (strict.next_player moves_so_far2)
 ; next_move_is_valid moves_so_far next_move
   := let (moves_so_far1, moves_so_far2)
          := partition_map (fun move => move) moves_so_far in
      let next_player
          := tensor_next_player
               (strict.next_player moves_so_far1) (strict.next_player moves_so_far2)
      in
      match next_move with
      | inl next_move => next_player == strict.next_player moves_so_far1
      | inr next_move => next_player == strict.next_player moves_so_far2
      end
 ; play_won_by_P all_moves
   := exists right_moves left_moves,
     moves_compatible_with left_moves right_moves all_moves
     /\ ((~left_game_not_abandoned all_moves
          /\ strict.play_won_by_P g2 right_moves)
         \/ (~right_game_not_abandoned all_moves
             /\ strict.play_won_by_P g1 left_moves)
         \/ (left_game_not_abandoned all_moves
             /\ right_game_not_abandoned all_moves
             /\ strict.play_won_by_P g1 left_moves
             /\ strict.play_won_by_P g2 right_moves))
 ; play_won_by_O all_moves
   := exists right_moves left_moves,
     moves_compatible_with left_moves right_moves all_moves
     /\ ((right_game_not_abandoned all_moves
          /\ strict.play_won_by_O g2 right_moves)
         \/ (left_game_not_abandoned all_moves
             /\ strict.play_won_by_O g1 left_moves))
  |}.
```

Design Choice. Without classical axioms we cannot compute whether or not a game was abandoned, nor even how many moves it will take to get back to a game that has been abandoned yet. So, instead of starting from the plays in the combined tensor game as Blass [Bla92] does, we must instead start with plays in the left and right games and assert their compatibility with the play in the combined tensor game.

4 Examples

In the previous section, we have described the seminal definition and described the construction of its proof. Our library gives users a functional programming based framework to write theorems that helps utilize winning strategies for proof search. Furthermore users are able to reuse our work and import the design choices required in this domain, including the ones that enable computability.

In this section, we present a selection of high powered results that can aid the use of winning strategies to find proofs. These results were proven using the library.

We also present the formalization of tic-tac-toe in our library as a reference point for formalizing other games. We share this in the form of code, as Coq code is essentially mathematics, and the specific structure of the code is quite important.

4.1 Theorems

Theorem 4. *In game g, for player p, the negation of strategy s is the strategy s for player $\neg p$ in the game $\neg g$.*

```
Definition negation_strategy {g} {p} (s : strategy g p) : strategy (~g) (~p) := s.
Notation "~ s" := (negation_strategy s) : strategy_scope.
```

Theorem 5. *Player p follows s in g in the play of all moves iff $\neg p$ follows $\neg s$ in $\neg g$ in the play of all moves.*

```
Lemma negation_player_follows_strategy {g} {p} {s : strategy g p} {all_moves : play g}
: player_follows_strategy p s all_moves
   <-> player_follows_strategy (~p) (~s) all_moves.
Proof.
  unfold player_follows_strategy.
  apply forall_iff_compat.
  intro n.
  apply imp_iff_compat_r.
  unfold next_player.
  simpl.
  generalize (Nat.even (Datatypes.length (Streams.firstn all_moves n)));
    intro fp_move.
  generalize (first_player g); intro fp.
  Local Open Scope player_scope.
  destruct fp_move, fp, p.
  all: simpl.
  all: done.
Qed.
```

Theorem 6. *The play all moves is won by p in g iff the play of all moves is won by $\neg p$ in $\neg g$.*

```
Lemma negation_play_won_by {g} {p} {all_moves : play g}
 : play_won_by p all_moves <-> play_won_by (g:=~g) (~ p) all_moves.
Proof.
```

```
  unfold play_won_by in *.
  simpl.
  destruct p.
  { simpl.
    reflexivity.
  }
  { simpl.
    reflexivity.
  }
Qed.
```

Theorem 7. *Strategy s is a winning strategy for p in g iff $\neg s$ is a winning strategy for $\neg p$ in $\neg g$.*

```
Lemma negation_winning_strategy {g} {p} {s : strategy g p}
  : winning_strategy s <-> winning_strategy (~s).
Proof.
  unfold winning_strategy.
  unfold winning_strategy.
  apply forall_iff_compat; intro w.
  rewrite <- negation_player_follows_strategy.
  apply imp_iff_compat_l.
  apply negation_play_won_by.
Qed.
```

Theorem 8. *In the top game where the only make that either player can make is to continue the game, there is a strategy for either player which consists of always continuing the game. Furthermore, there is a play of the top game which consists of both players continuing the game forever.*

```
Definition trivial_strategy {p} : strategy top p := fun all_moves_so_far => tt.
CoFixpoint trivial_play : play top := Streams.Cons tt trivial_play.
```

Theorem 9. *For any strategy s for either player p, in any play of the top game p follows s in that play.*

```
Lemma top_player_follows_strategy {p} {s : strategy top p} {all_moves : play top}
: player_follows_strategy p s all_moves.
Proof.
  unfold player_follows_strategy.
  intros n Hp.
  destruct (Streams.nth all_moves n.+1).
  destruct (s (Streams.firstn all_moves n)).
  reflexivity.
Qed.
```

Theorem 10. *A play of the top game is won by a player iff that player is player P.*

```
Lemma top_play_won_by {p} {all_moves : play top}
 : play_won_by p all_moves <-> p = player_P.
Proof.
  unfold play_won_by, play_won_by_P, play_won_by_O, top.
  case: p => //.
Qed.
```

Theorem 11. *Any strategy in the top game for a player is a winning strategy iff the player is player P.*

```
Lemma top_winning_strategy {p} {s : strategy top p}
 : winning_strategy s <-> p = player_P.
Proof.
  unfold winning_strategy.
  setoid_rewrite top_play_won_by.
  firstorder.
  eapply H.
  apply top_player_follows_strategy.
  Unshelve.
  exact trivial_play.
Qed.
```

Theorem 12. *For any strategy s for either player p, in any play of the bottom game p follows s in that play.*

```
Lemma bottom_player_follows_strategy {p} {s : strategy bottom p} {all_moves : play bottom}
: player_follows_strategy p s all_moves.
Proof.
  unfold player_follows_strategy.
  intros n Hp.
  destruct (Streams.nth all_moves n.+1).
  destruct (s (Streams.firstn all_moves n)).
  reflexivity.
Qed.
```

Theorem 13. *A play of the bottom game is won by a player iff that player is player O.*

```
Lemma bottom_play_won_by {p} {all_moves : play bottom}
 : play_won_by p all_moves <-> p = player_O.
Proof.
  unfold play_won_by, play_won_by_P, play_won_by_O, top.
  case: p => //.
Qed.
```

Theorem 14. *Any strategy in the top game for a player is a winning strategy iff the player is player O.*

```
Lemma bottom_winning_strategy {p} {s : strategy bottom p}
 : winning_strategy s <-> p = player_O.
Proof.
  unfold winning_strategy.
  setoid_rewrite bottom_play_won_by.
  firstorder.
  eapply H.
  apply bottom_player_follows_strategy.
  Unshelve.
  exact trivial_play.
Qed.
```

4.2 Games

```
Definition tic_tac_toe_game : relaxed.game.
refine {|
   relaxed.possible_move := nat * nat
 ; relaxed.first_player := player_P
 ; relaxed.next_player moves_so_far := if Nat.even (List.length moves_so_far)
                                       then player_P else player_O
 ; relaxed.play_won_by_P all_moves
   := exists n : nat,
     tic_tac_toe.game_outcome
       (Streams.firstn all_moves n) tic_tac_toe.initial_state
     == Some tic_tac_toe.player_1
 ; relaxed.play_won_by_O all_moves
   := exists n : nat,
     tic_tac_toe.game_outcome
       (Streams.firstn all_moves n) tic_tac_toe.initial_state
     == Some tic_tac_toe.player_2
 ; relaxed.next_move_is_valid := tic_tac_toe.next_move_is_valid_or_game_finished
  |}.

Module tic_tac_toe.
Notation new_line := (String "010" EmptyString) (only parsing).

Inductive player := player_1 | player_2.
Definition cell := option player.
Definition cell_to_string (c : cell) : string :=
  match c with
  |None => " "
  |Some player_1 => "X"
  |Some player_2 => "O"
  end.
Definition board := seq (seq cell).
Definition empty : cell := None.
Coercion some_player (p : player) : cell := Some p.
Definition set_cell (b : board) (r : nat) (c : nat) (p : player) : board
  := set_nth [::] b r (set_nth empty (nth [::] b r) c p).
```

```
Definition get_cell (b : board) (r : nat) (c : nat) : cell
  := nth empty (nth [::] b r ) c.
Definition get_column (b : board) (c : nat) : seq cell
  := map (fun r => get_cell b r c) [:: 0 ; 1 ; 2 ].
Definition rows (b: board) : seq (seq cell) := b.
Definition columns (b : board) : seq (seq cell) :=
  map (fun c => get_column b c) [:: 0 ; 1 ; 2 ].
Definition diagonals (b : board) : seq (seq cell)
  := [:: [:: get_cell b 0 0 ; get_cell b 1 1 ; get_cell b 2 2]
      ; [:: get_cell b 0 2 ; get_cell b 1 1 ; get_cell b 2 0] ].
Definition cells_same_non_empty (s : seq cell) : bool
  := foldr andb true (map (fun c => c == nth empty s 0) s)
     && (nth empty s 0 != empty).
Definition any_cells_same_non_empty (s : seq (seq cell)) : bool :=
  foldr orb false (map cells_same_non_empty s).
Definition any_row (b: board) : bool :=
  any_cells_same_non_empty (rows b).
Definition any_column (b : board) : bool :=
  any_cells_same_non_empty (columns b).
Definition any_diagonal (b : board) : bool
  := any_cells_same_non_empty (diagonals b).
Definition other_player (current_player : player) : player
  := match current_player with
     |player_1 => player_2
     |player_2 => player_1
     end.
Definition initial_board : board
  := nseq 3 (nseq 3 empty).
Definition output_row (s: seq cell) : string
  := foldr String.append ""%string (map cell_to_string s) ++ new_line.
Definition output_board (b : board) : string :=
  foldr String.append ""%string (map output_row b).
Definition game_result (state : board * player) : option player
  := let (b, current_player) := state in
     if any_row b || any_column b
        || any_diagonal b
     then Some (other_player current_player)
     else None.
Definition initial_state : board * player
  := (initial_board, player_1).
Definition game_result_string (p : option player) : string := later.
Definition game_intro : string := later.
Definition main_game (b : board) : unit := later.
Definition input_and_make_move (current_player : player) (b : board)
  : player * board := later.
Definition move_is_valid (b : board) (r : nat) (c : nat) : bool
  := (r <= 3) && (c <= 3) && (get_cell b r c == empty).
Definition make_move (b: board) (current_player : player)
           (r : nat) (c : nat) : board * bool
  := if move_is_valid b r c
     then (set_cell b r c current_player, true)
     else (b, false).
Definition make_single_move (state : board*player) (next_move : nat*nat)
  : (board*player)*bool
  := let (r, c) := next_move in
     let (b, p) := state in
     let (new_board, is_valid)
         := make_move b p r c in
     ((new_board, other_player p), is_valid).
Fixpoint make_moves (moves : seq (nat*nat)) (state : board*player) : board*player
  := match moves with
     |[::] => state
     |move :: moves =>
      let new_state := fst (make_single_move state move) in
      make_moves moves new_state
     end.

Definition next_move_is_valid
  (moves_so_far : seq (nat * nat)) (next_move : nat * nat) : bool
  := let (r, c) := next_move in
     let (b, _) := make_moves moves_so_far initial_state in
     move_is_valid b r c.
```

```
Fixpoint game_outcome (moves : seq (nat * nat)) (state : board*player) : option player
  := (*None means game is still in progress,
       some player means game is over and that player won*)
    match moves with
    | [::] => None
    | move :: moves =>
      let (new_state, is_valid) := make_single_move state move in
      if is_valid
      then match game_result new_state with
           | None => game_outcome moves new_state
           | Some winner => Some winner
           end
      else
        Some (other_player (snd state))
    end.

Compute output_board
  (fst (make_move initial_board player_1 1 2) ).
Compute output_board
  (fst (make_move (fst (make_move initial_board player_1 1 2)) player_2 1 1) ).
Compute output_board
  (fst (make_moves [:: (1, 1) ; (2, 2) ; (1, 0) ; (0,0) ; (1, 2)] initial_state)).
Compute game_result
  (make_moves [:: (1, 1) ; (2, 2) ; (1, 0) ; (0,0) ; (1, 2)] initial_state).

Definition next_move_is_valid_or_game_finished (moves_so_far : seq (nat * nat))
  (next_move : nat * nat) : bool
  := next_move_is_valid moves_so_far next_move
     || (game_outcome moves_so_far initial_state != None).

End tic_tac_toe.
```

Above, is the formalization of tic-tac-toe which corresponds to the structure of game semantics. Tic-tac-toe is one of the more complex games to formalize because of its grid-like structure, as opposed to a game like Nim.

5 Acknowledgments

The author acknowledges Jason Gross, PhD, MIT CSAIL for providing expertise on proof assistants and formal logic, and Rajashree Agrawal, Reed College for providing expertise in games. The author is grateful for their generous mentorship and unwavering support.

6 Bibliography

[Bla92] Andreas Blass. "A game semantics for linear logic". In: *Annals of Pure and Applied Logic* 56.1 (1992), pp. 183–220. ISSN: 0168-0072. DOI: `10.1016/0168-0072(92)90073-9`. URL: `https://www.sciencedirect.com/science/article/pii/0168007292900739`.

[Gon05] Georges Gonthier. *A computer-checked proof of the Four Colour Theorem*. 2005. URL: `https://audentia-gestion.fr/MICROSOFT/4colproof.pdf`.

[Gon08] Georges Gonthier. "Formal Proof–The Four-Color Theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393. URL: `https://www.ams.org/notices/200811/tx081101382p.pdf`.

[Kni12] Rob Knies. *Six-year journey leads to proof of Feit-Thompson Theorem*. Microsoft. Oct. 12, 2012. URL: `https://phys.org/news/2012-10-six-year-journey-proof-feit-thompson-theorem.html`.

[AJ13] Samson Abramsky and Radha Jagadeesan. "Games and Full Completeness for Multiplicative Linear Logic". In: *CoRR* abs/1311.6057 (2013). arXiv: `1311.6057`.

[Gon+13] 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. "A Machine-Checked Proof of the Odd Order Theorem". In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin, and David Pichardie. Vol. 7998. LNCS. Rennes, France: Springer, July 2013, pp. 163–179. DOI: `10.1007/978-3-642-39634-2\_14`. URL: `https://hal.inria.fr/hal-00816699`.

[YP13] Edward Z. Yang and Frank Pfenning. *Unofficial Lecture Notes on Linear Logic and Session-based Concurrency at OPLSS*. July 2013. URL: `https://www.cs.uoregon.edu/research/summerschool/summer13/lectures/pfenning-2.pdf`.

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. "Experience Implementing a Performant Category-Theory Library in Coq". In: *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP'14)*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, July 2014, pp. 275–291. ISBN: 978-3-319-08970-6. DOI: `10.1007/978-3-319-08970-6_18`. eprint: `1401.7694`. URL: `https://jasongross.github.io/papers/category-coq-experience-itp-submission-final.pdf`.

[Hal+15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. *A formal proof of the Kepler conjecture*. 2015. arXiv: `1501.02155 [math.MG]`.

[Sil+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: `10.1038/nature16961`.

[Bau+17] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. Paris, France: ACM, Jan. 2017, pp. 164–172. ISBN: 978-1-4503-4705-1. DOI: `10.1145/3018610.3018615`. eprint: `1610.04591`. URL: `https://jasongross.github.io/papers/2017-HoTT-formalization.pdf`.

[Loo+17] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. "Deep Network Guided Proof Search". In: *CoRR* abs/1701.06972 (2017). arXiv: `1701.06972`.

[Kal+18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek OlsÃ¡k. "Reinforcement Learning of Theorem Proving". In: *CoRR* abs/1805.07563 (2018). arXiv: `1805.07563`.

[Xav+18] Bruno Xavier, Carlos Olarte, Giselle Reis, and Vivek Nigam. "Mechanizing Focused Linear Logic in Coq". In: *Electronic Notes in Theoretical Computer Science* 338 (2018). The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017), pp. 219–236. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2018.10.014`. URL: `https://www.sciencedirect.com/science/article/pii/S157106611830080X`.

[Vin+19] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II.* 2019. URL: `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`.

[Coq21] The Coq Development Team. *The Coq Proof Assistant.* Version 8.13. Jan. 2021. DOI: `10.5281/zenodo.4501022`.

[Gro21] Jason S. Gross. "Performance Engineering of Proof-Based Software Systems at Scale". PhD Thesis. Massachusetts Institute of Technology, Feb. 2021. URL: `https://jasongross.github.io/papers/2021-JGross-PhD-EECS-Feb2021.pdf`.