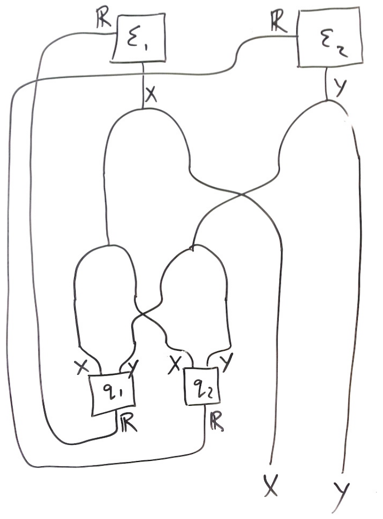
String diagrams for game theory: a (very) preliminary report

Jules Hedges
Aided and abetted by
Viktor Winschel   Philipp Zahn

February 17, 2015

1 Introduction

We introduce a string language for game theory. The string language is close to the one for non-compact categories (ie. causality usually flows one way), plus a feedback operation (that seems to be new) to deal with the forward-reasoning of players. For example here is a picture of a general bimatrix game:



Even though the string language is simple and intuitive, the semantics is subtle and heavily uses continuation passing style. (I've been working for quite some time on connections between CPS and game theory.) This note consists of a few pages of theory, then examples, and in the last section a Haskell implementation including implementations of each example. Most of the proofs don't exist yet, my evidence consists mostly of the fact that the implementation gives the right answers in a wide variety of examples.

2 Background definitions

Fix a cartesian closed category $\mathcal{C}$ that contains everything convenient, eg. finite sets of moves, real numbers. Later we also fix a strong monad on $\mathcal{C}$ to represent side effects in the game, eg. probability distributions of finite support, probabilistic state transformers.

For the theoretical sections we work only with pure strategies, for simplicity. To see the definitions for games with side effects, look at the code.

3 Pregames

Definition 1. *A pregame $\mathcal{G} = (A, R, \mathbb{P}, \mathbb{E}) : X \to Y$ consists of:*

- *Types X, Y of histories, plays*
- *Types A, R of strategies, outcomes*
- *A play function $\mathbb{P} : A \to X \to Y$*
- *An equilibrium relation $\mathbb{E} : A \times ((X \to Y) \to R) \to \mathbb{B}$*

A *game* $X \to Y$ is a pregame $(A, 1, \mathbb{P}, \mathbb{E}) : X \to Y$, ie. the outcomes have been 'closed off'. (So a pregame is a sort of 'open game' as in open/closed formulas in formal languages). In this case we have $((X \to Y) \to R) \cong 1$ and so the equilibrium relation $\mathbb{E}$ is a unary relation on strategies. A *computation* is a game $(1, 1, \mathbb{P}, *) : X \to Y$, ie. it is a game with no strategic content, in which additionally the unique strategy $* : 1$ is an equilibrium. We can lift any function $f : X \to Y$ to a computation $\overline{f} : X \to Y$ by setting $\mathbb{P} * = f$.

The deeply magical part here is the type $(X \to Y) \to R$, which is a sort of continuation. This is what makes the whole thing work.

In string diagrams we draw a pregame as a box with the wires X and Y at the top and bottom (so time flows downwards), and R going sideways. The R is what makes this interestingly different to usual string diagrams.

4 Decision problems

Other than computations, our basic pregames are decision problems. These are defined by relations $\varepsilon : (X \to Y) \times ((X \to Y) \to R) \to \mathbb{B}$ which specify when a player is happy with a strategy $X \to Y$ (mapping observations to choices) in a strategic context that maps every strategy to an outcome. For example, when $(R, \leq)$ is totally ordered, we can set

$$f \varepsilon k \iff kf = \max_{f'} kf'$$

(These relations started life as selection functions $\varepsilon : (Y \to R) \to Y$, and have been generalised beyond all recognition. First they became multivalued $(Y \to R) \to Y \to \mathbb{B}$, and then they became relativised to the history X.)

We can consider a decision problem $\varepsilon : (X \to Y) \times ((X \to Y) \to R) \to \mathbb{B}$ as a pregame $\overline{\varepsilon} : X \to Y$ by defining

- $A_{\overline{\varepsilon}} = X \to Y$

- $R_{\overline{\varepsilon}} = R$
- $\mathbb{P}_{\overline{\varepsilon}} f x = f x$
- $\mathbb{E}_{\overline{\varepsilon}} = \varepsilon$

5 Category structure

Given $\mathcal{G} : X \to Y$ and $\mathcal{H} : Y \to Z$ we define $\mathcal{H} \circ \mathcal{G} : X \to Z$ such that

- $A_{\mathcal{H} \circ \mathcal{G}} = A_{\mathcal{G}} \times A_{\mathcal{H}}$
- $R_{\mathcal{H} \circ \mathcal{G}} = R_{\mathcal{G}} \times R_{\mathcal{H}}$
- $\mathbb{P}_{\mathcal{H} \circ \mathcal{G}}(a, b) = \mathbb{P}_{\mathcal{H}} b \circ \mathbb{P}_{\mathcal{G}} a$
- $(a, b)\mathbb{E}_{\mathcal{H} \circ \mathcal{G}} k \iff a\mathbb{E}_{\mathcal{G}} k_1 \wedge b\mathbb{E}_{\mathcal{H}} k_2$ where
$$k_1 = \lambda(f : X \to Y).\pi_1(k(\mathbb{P}_{\mathcal{H}} b \circ f))$$
$$k_2 = \lambda(f : Y \to Z).\pi_2(k(f \circ \mathbb{P}_{\mathcal{G}} a))$$
(as a hint to mental type checking, we have $k : (X \to Z) \to R \times S$).

Theorem 1. *There is a category $\mathbb{C}$ in which objects are types, and morphisms are pregames. The composition is $\circ$ and the identity is $\overline{\mathrm{id}}$.*

Proof. Left unit: Take $\mathcal{G} : X \to Y$ and consider $\mathcal{G} \circ \overline{\mathrm{id}_X} : X \to Y$. Up to natural isomorphism we have

- $A_{\mathcal{G} \circ \overline{\mathrm{id}_X}} = A_{\mathcal{G}} \times 1 = A_{\mathcal{G}}$
- Similarly $R_{\mathcal{G} \circ \overline{\mathrm{id}_X}} = R_{\mathcal{G}}$
- $\mathbb{P}_{\mathcal{G} \circ \overline{\mathrm{id}_X}} a x = \mathbb{P}_{\mathcal{G}} a(\mathbb{P}_{\overline{\mathrm{id}_X}} * x) = \mathbb{P}_{\mathcal{G}} a(\mathrm{id}_X\, x) = \mathbb{P}_{\mathcal{G}} a x$
- $a\mathbb{E}_{\mathcal{G} \circ \overline{\mathrm{id}_X}} k \iff a\mathbb{E}_{\mathcal{G}} k_2$ where
$$k_2 = \lambda(f : X \to Y).k(f \circ \mathbb{P}_{\overline{\mathrm{id}_X}} *) = k$$

Right unit: symmetric.

Associativity: Take $\mathcal{G} : X \to Y$, $\mathcal{H} : Y \to Z$ and $\mathcal{I} : Z \to W$. The only condition that really needs checking is the one for $\mathbb{E}$. Up to natural transformation both ways of bracketing give

$$(a, b, c)\mathbb{E}_{\mathcal{I} \circ \mathcal{H} \circ \mathcal{G}} k \iff a\mathbb{E}_{\mathcal{G}} k_1 \wedge b\mathbb{E}_{\mathcal{H}} k_2 \wedge c\mathbb{E}_{\mathcal{I}} k_3$$

(where $k : (X \to W) \to R \times S \times T$) where

$$k_1 = \lambda(f : X \to Y).\pi_1(k(\mathbb{P}_{\mathcal{I}} c \circ \mathbb{P}_{\mathcal{H}} b \circ f))$$
$$k_2 = \lambda(f : Y \to Z).\pi_2(k(\mathbb{P}_{\mathcal{I}} c \circ f \circ \mathbb{P}_{\mathcal{G}} a))$$
$$k_3 = \lambda(f : Z \to W).\pi_3(k(f \circ \mathbb{P}_{\mathcal{H}} b \circ \mathbb{P}_{\mathcal{G}} a))$$

□

Game-theoretically this categorical composition is a strange thing, like a sequential composition in which the first player has been 'forgotten' (or 'cut' in logic...). We can get back a game-theoretically sensible sequential composition using categorical composition together with copying (the comonoid structure in the next section). (See example 5.) This is 'splitting the atom of game theory', decomposing sequential composition into something more primitive.

6 Tensor and comonoid structure

Just as categorical composition gives a primitive version of sequential games, the monoidal structure gives a primitive version of simultaneous games.

Given games $\mathcal{G} : X_1 \to Y_1$ and $\mathcal{H} : X_2 \to Y_2$ we define $\mathcal{G} \otimes \mathcal{H} : X_1 \times X_2 \to Y_1 \times Y_2$ by

- $A_{\mathcal{G} \otimes \mathcal{H}} = A_\mathcal{G} \times A_\mathcal{H}$
- $R_{\mathcal{G} \otimes \mathcal{H}} = R_\mathcal{G} \times R_\mathcal{H}$
- $\mathbb{P}_{\mathcal{G} \otimes \mathcal{H}}(a, b)(x_1, x_2) = (\mathbb{P}_\mathcal{G} a x_1, \mathbb{P}_\mathcal{H} b x_2)$
- $(a, b)\mathbb{E}_{\mathcal{G} \otimes \mathcal{H}} k \iff a\mathbb{E}_\mathcal{G} k_1 \wedge b\mathbb{E}_\mathcal{H} k_2$ (where $k : (X_1 \times X_2 \to Y_1 \times Y_2) \to R \times S$) where

$$k_1 = \lambda(f : X_1 \to Y_1).\pi_1(k\lambda((x_1, x_2) : X_1 \times X_2).(f x_1, \mathbb{P}_\mathcal{H} b x_2))$$

$$k_2 = \lambda(f : X_2 \to Y_2).\pi_2(k\lambda((x_1, x_2) : X_1 \times X_2).(\mathbb{P}_\mathcal{G} a x_1, f x_2))$$

Theorem 2 (:: Maybe Theorem)**.** $(\mathbb{C}, \otimes, 1)$ *is a symmetric monoidal closed category. Or maybe a closed Freyd category. The important question is, what are the valid operations on string diagrams? We probably get different answers depending on whether side-effects are coming from a commutative or noncommutative monad.*

For every type X we also have copying and deleting computations $\Delta : X \to X \times X$, $! : X \to 1$ by lifting them from $\mathcal{C}$ (using the overline operation from section 3). This makes every object of $(\mathbb{C}, \otimes, 1)$ into a commutative comonoid.

N.B. $\otimes$ is not a cartesian product, for what looks like similar reasons to in a Freyd category. Making a decision and copying the result is not the same thing as making the decision twice. The object 1 is not even terminal, because like in a Freyd category we can delete information in nontrivial ways.

7 Strange trace-like thing

Suppose $\mathcal{G}$ is a pregame of the form $\mathcal{G} : 1 \to R_1 \times Y$ with $R_\mathcal{G} = R_1 \times R_2$. We define a game $\Box\mathcal{G} : 1 \to Y$ with

- $A_{\Box\mathcal{G}} = A_\mathcal{G}$
- $R_{\Box\mathcal{G}} = R_2$
- $\mathbb{P}_{\Box\mathcal{G}} a = \pi_2 \circ \mathbb{P}_\mathcal{G}$
- $a\mathbb{E}_{\Box\mathcal{G}} k \iff a\mathbb{E}_\mathcal{G} k'$ where

$$k' = \lambda((r, y) : R_1 \times Y).(r, ky)$$

or to be more precise,

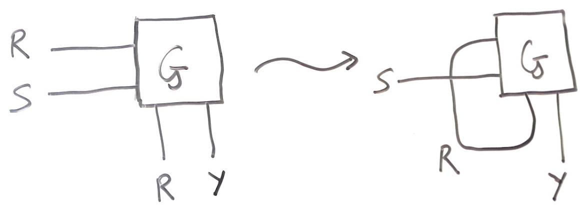
$$k' = \lambda(f : 1 \to R_1 \times Y).(\pi_1(f*), k\lambda(* : 1).\pi_2(f*))$$

where $* : 1$.

There's a fairly close analogy to logic here. $\mathcal{G}$ is like a formula with free variables of type R_1 and R_2 (and we can think of R_2 as a tuple of all the remaining free variables). Then we close off one variable with a quantifier, and the other variables remain open.

This is the least obvious part from the point of view of category theory. The biggest restriction is that this only applies to morphisms from 1 (points?), which looks ugly and more importantly makes it unclear how to do repeated games (in which there is utility feeding back between stages that are defined relative to a history). If this condition is relaxed then a strategy only determines an outcome relative to X. The feedback operation can then be applied if $R_{\mathcal{G}} = (X \to MR_1) \times R_2$. The presence of the monad M in that type seems to make things really difficult.

In any case we draw this operation in string diagrams like this:

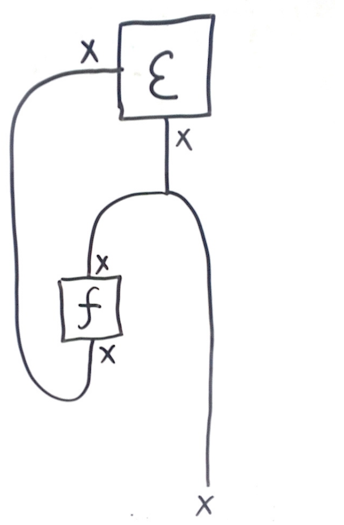


I have 2 different ways of thinking about this. One is that it's capturing just the right amount of backward-causality that you get in game theory caused by players reasoning about future events. Another is that it's related to delimited continuations eg. with shift/reset, where decisions correspond to 'shift' (they capture a continuation) and the twist corresponds to 'reset' (it delimits a continuation).

8 Implementation

See the last section for a complete Haskell implementation of the definitions above, and the examples below. As the examples get more complicated I get less precise with the mathematics, because you can check the details in the code (if you read Haskell).

9 Example 1: Decision problem



We model the decision problem of a player who makes a choice $x : X$ which results in an outcome $fx : X$, and the player would like the choice to be the same as the outcome. For example in a Keynes beauty contest, X is the set of candidates and the agent would like to vote for the winner. This decision is modelled by the selection function

$$\varepsilon : X \times (X \to X) \to \mathbb{B}$$

(the type is really $(1 \to X) \times ((1 \to X) \to X) \to \mathbb{B})$ given by

$$x\varepsilon f \iff x = fx$$

We build up to the point before we bend the arrow around:

$$\mathcal{G} = (f \otimes \mathrm{id}) \circ \Delta \circ \varepsilon : 1 \to X \times X$$

(where Δ is copying) (Here we consider f as a computation, and start abusing notation by not overlining things with no strategic content.) The types here (up to natural isomorphism) are $A_{\mathcal{G}} = X$ (ie. the player's strategy is just a choice of move) and $R_{\mathcal{G}} = (1 \to X) \times 1$. $\mathcal{G}$ is given by

$$\mathbb{P}_{\mathcal{G}} x * = (fx, x)$$

$$x\mathbb{E}_{\mathcal{G}} k \iff x\varepsilon k'$$

(where $k : (1 \to X \times X) \to (1 \to X) \times 1)$ where

$$k' = \lambda(x : X).\pi_1(k\lambda(* : 1).(fx, x))*$$

Now we make $\square\mathcal{G} : 1 \to X$ with $R_{\square\mathcal{G}} = 1$ (ie. $\square\mathcal{G}$ is a game, although a 'degenerate' 1-player game). Now we get

$$\mathbb{P}_{\square\mathcal{G}} x * = x$$

$$x \mathbb{E}_{\square \mathcal{G}} * \iff x \mathbb{E}_{\mathcal{G}} k \iff x \varepsilon k'$$

where

$$k = \lambda(g : 1 \to X \times X).\pi_1 \circ g$$

$$k' = \lambda(x : X).\pi_1(k\lambda(* : 1).(fx, x))* = f$$

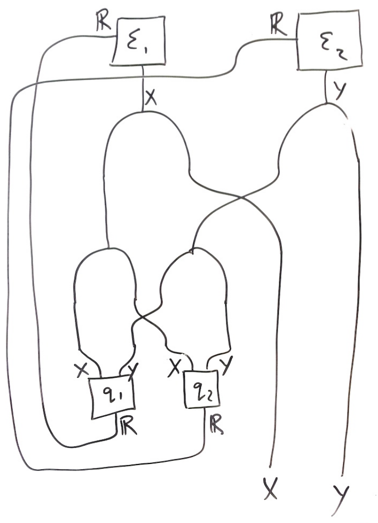
So x is an equilibrium of $\square \mathcal{G}$ iff it is a fixpoint of f.

Obvious observation: Even for the simplest possible example this is nearly impossible to keep track of all the type isomorphisms by hand. Computer support is vital!

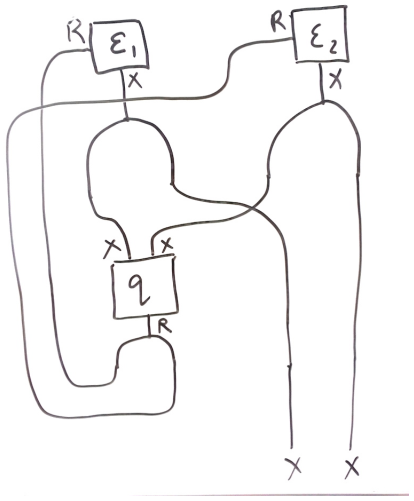
Each example will work in the same way: first build a pregame by directly translating the string diagram into combinators, then apply the contravariant action with a type isomorphism to get R into the right form, and finally apply feedback. Even in the simplest example it is totally impractical to track by hand the type isomorphisms needed for R. One of two things are needed: either a type system that makes unifications like $X \sim 1 \times X$ and $X \sim 1 \to X$, or a layer of code generation between the user and Haskell that calculates the isomorphisms manually.

10 Example 2: bimatrix game with pure strategies

Here is a picture of a general bimatrix game:



We will consider an example, Meeting In New York (a coordination game) in which the two players receive the same payoff. Since both players receive the same utility (and have the same preferences) we can use a simpler representation:



(Corollary: not all bimatrix games are created equal - qualitative features of bimatrix games which are not revealed by the usual formulation can be seen graphically here.)

Here R is the 2-element poset $\{\text{lose} < \text{win}\}$, and both players are maximising. The computation $q : X \times X \to R$ returns win if the inputs are equal, and lose if they are not equal. Strategies are pairs $X \times X$.

The two selection functions are the same (because the players have the same preferences) and have type $\varepsilon : X \times (X \to R) \to \mathbb{B}$. The definition is that

$$x \varepsilon k \iff kx = \max_{x'} kx'$$

The type of strategies for the whole thing is $X \times X$, and $\mathbb{E}$ correctly picks out the coordinating strategies (A, A) and (B, B).

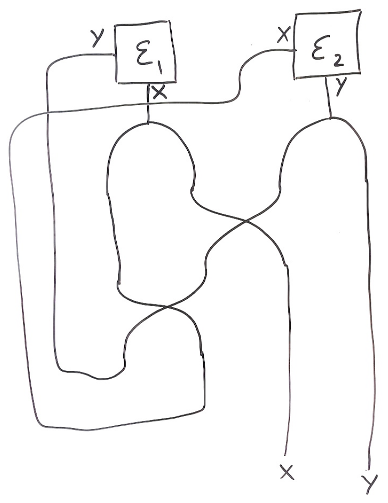
11 Example 3: bimatrix game with mixed strategies

Take the same example, but change the monad from identity to the Giry monad Δ (where ΔX is probability distributions on X of finite support). We need to use the expectation algebra $E : \Delta\mathbb{R} \to \mathbb{R}$, so outcomes will change from discrete win/lose to numerical utilities $\mathbb{R}$. The selection functions of the players simply change to

$$x \varepsilon k \iff E(kx) = \max_{x'} E(kx')$$

Technically here x' should vary over all mixed strategies, but in the code we take a shortcut - since everything in this example is multilinear, k can only be maximised at a vertex of the simplex, so it suffices to check only the 2 pure strategies.

12 Example 4: bimatrix game with nonclassical preferences



We can model the same game again in a better way, by throwing away utilities and taking selection functions seriously. The idea is that in a coordination game players don't have preferences over numbers or any abstract 'outcomes', they have preferences directly over each other's choice. The picture labels the two types of choices X and Y for clarity, but actually $X = Y$. Both players want to coordinate, which is modelled by a fixpoint selection function with type

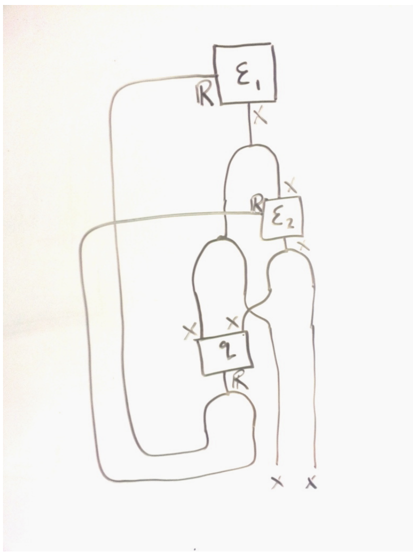
$$\varepsilon : \Delta X \times (X \to \Delta X) \to \mathbb{B}$$

given by

$$a \varepsilon k \iff a = a \gg= k$$

We can see by experimenting that this model does the same as the previous one, even though it's implemented in a totally different way - the previous one doesn't have any fixpoints appearing explicitly anywhere.

13 Example 5: sequential game



The next game is a sequential version of the coordination game, in which the second player can observe the move of the first player. The selection function for the first player is the same as the one from example 3, and the one for the second player has type

$$\varepsilon_2 : (X \to \Delta X) \times ((X \to \Delta X) \to \Delta\mathbb{R}) \to \mathbb{B}$$

This again is defined by

$$a\varepsilon_2 k \iff E(ka) = \max_{a'} E(ka')$$

but this time a' is varying over strategies contingent on the observed move. Again we take a shortcut, and just check the 4 pure strategies.

This example confirms the intuition that this game has equilibria in which the first player is mixing with arbitrary probability, and the second player plays a copycat strategy.

We can also model imperfect information by inserting another computation box between the two players, which forgets some information carried in the wire. Since this box can also have side effects the games can be really dynamic with the game itself having effects interleaved with the players' decisions.

The solution concept defined by $\mathbb{E}$ corresponds to Nash equilibrium, but I think it extends quite easily to subgame-perfect equilibrium, which is a more appropriate solution concept for sequential games.

14 Example 6: starting to really show off

(This section contains no mathematics... I don't understand how my own code works, so just look in the appendix)

The point of this example is to show the power not of string diagrams, but of allowing side-effects from an arbitrary monad. Generally the theory of correlated equilibrium is quite different to ordinary Nash, but it falls out as a special case if we use a reader monad to allow the players' strategies to have read-only access to a shared source of randomness. This was originally intended as a half-way step to using state monads to model learning agents. For this we need selection functions to be 'relations with side effects' (which makes more sense as functional programming than as mathematics) because the equilibrium check also needs to have access to the randomising device. In fact the two monads are different: the players need to see the concrete signal (after the randomness has been resolved) but the check needs to have first-class access to the device itself. The code itself looks very dubious, but it is also doing the right thing on a more complex example, with totally read-only code using a very outdated version of the equilibrium checker (pre-dating string diagrams, ie. over 2 weeks old). My main point in showing this example is that there's serious expressive power to be gained by using different monads, and also a lot of theory to be figured out.

15 Code dump

```
{-# LANGUAGE NoMonomorphismRestriction, GADTs #-}

import Control.Monad (liftM, liftM2)

--import for examples
import Data.Functor.Identity
import Numeric.Probability.Distribution hiding (map, check, lift)
import Data.Ratio
import Control.Monad.Trans.Reader
import Control.Monad.Trans.Class

type Computation m m' x y = Pregame m m' x y () ()
type Decision m m' x y r = Pregame m m' x y r (x -> m y)

data Pregame m m' x y r a where
  Arr :: (Monad m, Monad m') => (x -> y) -> Computation m m' x y
  Contra :: (Monad m, Monad m') => (r -> s) -> Pregame m m' x y s a ->
      Pregame m m' x y r a
  Decision :: (Monad m, Monad m') => ((x -> m y) -> ((x -> m y) -> m r)
      -> m' Bool) -> Decision m m' x y r
  Compose :: (Monad m, Monad m') => Pregame m m' x y r a -> Pregame m m'
      y z s b -> Pregame m m' x z (r, s) (a, b)
  Tensor :: (Monad m, Monad m') => Pregame m m' x y r a -> Pregame m m' x
      ' y' s b -> Pregame m m' (x, x') (y, y') (r, s) (a, b)
  Twist :: (Monad m, Monad m') => Pregame m m' () (r, y) (r, s) a ->
      Pregame m m' () y s a

play :: Pregame m m' x y r a -> a -> x -> m y
play (Arr f) () x = return (f x)
play (Contra _ g) a x = play g a x
play (Decision _) f x = f x
play (Compose g h) (a, a') x = play g a x >>= play h a'
play (Tensor g h) (a, a') (x, x') = liftM2 (,) (play g a x) (play h a' x
    ')
play (Twist g) a x = liftM snd (play g a x)
```

```
check :: Pregame m m' x y r a -> a -> ((x -> m y) -> m r) -> m' Bool
check (Arr _) _ _ = return True
check (Contra i g) a k = check g a $ \f -> k f >>= return . i
check (Decision e) a k = e a k
check (Compose g h) (a, b) k = let k1 f = liftM fst $ k $ \x -> f x >>=
    play h b
                               k2 f = liftM snd $ k $ \x -> play g a x
                                   >>= f
                            in liftM2 (&&) (check g a k1) (check h b k2)
check (Tensor g h) (a, b) k = let k1 f = liftM fst $ k $ \(x, x') ->
    liftM2 (,) (f x) (play h b x')
                              k2 f = liftM snd $ k $ \(x, x') -> liftM2
                                  (,) (play g a x) (f x')
                           in liftM2 (&&) (check g a k1) (check h b k2)
check (Twist g) a k = check g a $ \f -> do r <- liftM fst (f ())
                                           s <- k $ \() -> liftM snd (f ())
                                           return (r, s)


-- %%%%%%%%%%%%%%%%%%%%%

copy = Arr $ \x -> (x, x)
identity = Arr id

data X = A | B deriving (Show, Eq, Ord)
data R = Lose | Win deriving (Show, Eq, Ord)

-- Example 1: decision problem

player1 :: Decision Identity Identity () X X
player1 = Decision $ \a k -> Identity $ runIdentity (a ()) == (
    runIdentity $ k a)

g1 = player1 `Compose` copy `Compose` ((Arr $ const B) `Tensor` identity)
g1' = Contra (\(x, ()) -> ((x, ()), ((), ()))) g1
g1'' = Twist g1'

{-
*Main> :t check g1''
check g1''
  :: ((() -> Identity X, ()), ((), ()))
     -> ((() -> Identity X) -> Identity ()) -> Identity Bool
*Main> runIdentity $ check g1'' ((\() -> Identity A, ()), ((), ())) $
    const $ Identity ()
False
*Main> runIdentity $ check g1'' ((\() -> Identity B, ()), ((), ())) $
    const $ Identity ()
True
*Main>
-}

-- Example 2: bimatrix game, pure strategies
```

```
player2 :: Decision Identity Identity () X R
player2 = Decision $ \a k -> Identity $ runIdentity (k a) == (maximum $
    map (runIdentity . k) as)
  where as = [\() -> return A, \() -> return B]

q2 :: Computation Identity Identity (X, X) R
q2 = Arr $ \(a, b) -> if a == b then Win else Lose

g2 = copy ‘Compose‘ ((player2 ‘Compose‘ copy) ‘Tensor‘ (player2 ‘Compose‘
     copy)) ‘Compose‘ (Arr $ \((a,b),(c,d)) -> ((a,c),(b,d))) ‘Compose‘
    ((q2 ‘Compose‘ copy) ‘Tensor‘ identity)
g2’ = Contra (\((r,s),()) -> ((((), ((r, ()), (s, ()))), ()), (((), ()),
    ()))) g2
g2’’ = Twist g2’

{-
*Main> :t check g2’’
check g2’’
  :: ((((), ((() -> Identity X, ()), (() -> Identity X, ()))), ()),
     (((), ()), ()))
    -> ((() -> Identity (X, X)) -> Identity ()) -> Identity Bool
*Main> runIdentity $ check g2’’ ((((), ((\() -> Identity A, ()), (\() ->
    Identity A, ()))), ()), (((), ()), ())) $ const $ Identity ()
True
*Main> runIdentity $ check g2’’ ((((), ((\() -> Identity A, ()), (\() ->
    Identity B, ()))), ()), (((), ()), ())) $ const $ Identity ()
False
*Main> runIdentity $ check g2’’ ((((), ((\() -> Identity B, ()), (\() ->
    Identity A, ()))), ()), (((), ()), ())) $ const $ Identity ()
False
*Main> runIdentity $ check g2’’ ((((), ((\() -> Identity B, ()), (\() ->
    Identity B, ()))), ()), (((), ()), ())) $ const $ Identity ()
True
-}

-- Example 3: bimatrix game, mixed strategies

type Random = T Rational

-- Notice the common structure with player2: an algebra of the identity
    monad (runIdentity) got replaced with an algebra of the probability
    monad (expectation)

player3 :: Decision Random Identity () X Rational
player3 = Decision $ \a k -> Identity $ expected (k a) == (maximum $ map
    (expected . k) as)
  where as = [\() -> return A, \() -> return B]

q3 :: Computation Random Identity (X, X) Rational
q3 = Arr $ \(a, b) -> if a == b then 1 else 0

g3 = copy ‘Compose‘ ((player3 ‘Compose‘ copy) ‘Tensor‘ (player3 ‘Compose‘
     copy)) ‘Compose‘ (Arr $ \((a,b),(c,d)) -> ((a,c),(b,d))) ‘Compose‘
    ((q3 ‘Compose‘ copy) ‘Tensor‘ identity)
```

```
g3' = Contra (\((r,s),()) -> ((((), ((r, ()), (s, ()))), ()), (((), ()),
    ()))) g3
g3'' = Twist g3'

{-
*Main> runIdentity $ check g3'' ((((), ((\() -> certainly A, ()), (\() ->
     certainly A, ()))), ()), (((), ()), ())) $ const $ return ()
True
*Main> runIdentity $ check g3'' ((((), ((\() -> choose (1%2) A B, ()),
    (\() -> choose (1%2) A B, ()))), ()), (((), ()), ())) $ const $
    return ()
True
*Main> runIdentity $ check g3'' ((((), ((\() -> choose (1%2) A B, ()),
    (\() -> choose (1%3) A B, ()))), ()), (((), ()), ())) $ const $
    return ()
False
-}

-- Example 4: bimatrix game with nonclassical preferences

type Random' = T Double

player4 :: Decision Random' Identity () X X
player4 = Decision $ \a k -> Identity $ approx (a ()) (k a)

g4 = copy `Compose` ((player4 `Compose` copy) `Tensor` (player4 `Compose`
     copy)) `Compose` (Arr $ \((a,b),(c,d)) -> ((c,a),(b,d)))
g4' = Contra (\((r,s),()) -> (((), ((r, ()), (s, ()))), ())) g4
g4'' = Twist g4'

{-
*Main> runIdentity $ check g4'' (((), ((\() -> certainly A, ()), (\() ->
    certainly A, ()))), ()) $ const $ return ()
True
*Main> runIdentity $ check g4'' (((), ((\() -> choose 0.5 A B, ()), (\()
    -> choose 0.5 A B, ()))), ()) $ const $ return ()
True
*Main> runIdentity $ check g4'' (((), ((\() -> choose 0.5 A B, ()), (\()
    -> choose 0.4 A B, ()))), ()) $ const $ return ()
False
-}

-- Example 5: sequential game with mixed strategies

player5 = player3

player5' :: Decision Random Identity X X Rational
player5' = Decision $ \a k -> Identity $ expected (k a) == (maximum $ map
     (expected . k) as)
  where as = [const $ certainly A, const $ certainly B,
             certainly, \x -> certainly $ case x of {A -> B; B -> A}]

q5 = q3
```

```
g5 = player5 `Compose` copy `Compose` (copy `Tensor` (player5' `Compose`
    copy)) `Compose` (Arr $ \((a,b),(c,d)) -> ((a,c),(b,d))) `Compose`
    ((q5 `Compose` copy) `Tensor` identity)
g5' = Contra (\((r,s),()) -> ((((r, ()), ((), (s, ()))), ()), (((), ()),
    ()))) g5
g5'' = Twist g5'

{-
*Main> runIdentity $ check g5'' ((((\() -> choose (123%456) A B, ()), (()
    , (return, ()))), ()), (((), ()), ())) $ const $ return ()
True
*Main> runIdentity $ check g5'' ((((\() -> choose (123%456) A B, ()), (()
    , (const $ return A, ()))), ()), (((), ()), ())) $ const $ return ()
False
-}

-- Example 6: correlated equilibrium

data Signal = S1 | S2 deriving (Show, Eq, Ord)

-- No idea how this works... it was reverse engineered from read-only
    code that might have used Bayesian updating somehow... in any case
    it seems to do the right thing
player6 :: Decision (ReaderT Signal Random) (Reader (Random Signal)) () X
     Rational
player6 = Decision $ \a k ->
  do device <- ask
     let a1 () = runReaderT (a ()) S1
     let a2 () = runReaderT (a ()) S2
     let k1 a' = runReaderT (k $ \() -> lift $ a' ()) S1
     let k2 a' = runReaderT (k $ \() -> lift $ a' ()) S2
     return $ (runIdentity $ check player3 a1 k1) && (runIdentity $ check
          player3 a2 k2)

q6 :: Computation (ReaderT Signal Random) (Reader (Random Signal)) (X, X)
     Rational
q6 = Arr $ \(a,b) -> if a == b then 1 else 0

g6 = copy `Compose` ((player6 `Compose` copy) `Tensor` (player6 `Compose`
     copy)) `Compose` (Arr $ \((a,b),(c,d)) -> ((a,c),(b,d))) `Compose`
    ((q6 `Compose` copy) `Tensor` identity)
g6' = Contra (\((r,s),()) -> ((((), ((r, ()), (s, ()))), ()), (((), ()),
    ()))) g6
g6'' = Twist g6'

-- (s6,s6) is a correlated equilibrium. In (s6.s6') the players have
    tragically failed to agree on how to interpret the coordinating
    signal
s6, s6' :: ReaderT Signal Random X
s6 = do signal <- ask
       return $ case signal of {S1 -> A; S2 -> B}
s6' = do signal <- ask
        return $ case signal of {S1 -> B; S2 -> A}
```

```
device :: Random Signal
device = choose (1%2) S1 S2

{-
*Main> runIdentity $ runReaderT (check g6'' ((((), ((\() -> return A, ())
    , (\() -> return A, ()))), ()), ((((), ()), ())) $ const $ return ())
     device
True
*Main> runIdentity $ runReaderT (check g6'' ((((), ((\() -> return A, ())
    , (\() -> return B, ()))), ()), ((((), ()), ())) $ const $ return ())
     device
False
*Main> runIdentity $ runReaderT (check g6'' ((((), ((\() -> s6, ()), (\()
     -> s6, ()))), ()), ((((), ()), ())) $ const $ return ()) device
True
*Main> runIdentity $ runReaderT (check g6'' ((((), ((\() -> s6, ()), (\()
     -> s6', ()))), ()), ((((), ()), ())) $ const $ return ()) device
False
-}
```