Teaching Functional Programming with Soccer-Fun

Peter Achten

Model Based System Development, Radboud University Nijmegen, The Netherlands

P.Achten@cs.ru.nl

Abstract

In this paper we report on our experience with the functional framework Soccer-Fun, which is a domain specific language for simulating football. It has been developed for an introductory course in functional programming at the Radboud University Nijmegen, The Netherlands. We have used Soccer-Fun in teaching during the past four years. We have also experience in using Soccer-Fun for pupils in secondary education. Soccer-Fun is stimulating because it is about a well known problem domain. It engages students to problem solving with functional programming because it allows them to compete at several disciplines: the best performing football team can become champion of a tournament; the best written code can be awarded with a prize; students can be judged on the algorithms used. This enables every student to participate and perform at her favorite skill. Soccer-Fun is implemented in Clean and uses its GUI toolkit Object I/O for rendering. It can be implemented in any functional programming language that supports some kind of windowing toolkit.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.2 [*Design Tools and Techniques*]: Software libraries; D.3.2 [*Language Classifications*]: Applicative (functional) languages; I.6.3 [*Simulation and Modeling*]: Applications; I.6.m [*Simulation and Modeling*]: Miscellaneous; K.3.1 [*Computers and Education*]: Computer Uses in Education

General Terms Algorithms, Design, Experimentation

Keywords Clean, education, simulation, soccer

1. Introduction

The bachelor computer science curriculum at the Radboud University Nijmegen, The Netherlands, provides a compulsory introductory course in functional programming, called “Abstraction and Composition in Programming”. This second year course has now been taught for the past four years. As with any introductory course in any programming language, we first need to teach students the basic concepts of the programming language. For this course this is “classic” functional language material and covers topics ranging from simple basic types to recursive algebraic types; (higher order) functions; overloading; recursion and induction; correctness proofs, and more. The course is given in the functional programming language Clean [6, 12] and covers also Clean specific topics such as uniqueness types for programming with effects, applying strictness annotations, term graph rewriting, and dynamics. The exercises tend to favor abstract topics such as list processing tasks to exercise recursion, (syntax) tree operations to exercise algebraic data types and more recursion, interpreters over these data structures to exercise abstraction and even more recursion, and equational reasoning to stimulate thinking about software and, yes, recursion. Every year it turns out that there is a group of students who have a hard time understanding functional programming due to its abstract nature. Of course there is also a group of students who appreciate this style of programming.

Because the course is mandatory, and we think functional programming should be fun for everybody, we set out to find and create a stimulating range of exercises that would engage all students. An old statement by *Johan Cruijff*, a well-known Dutch football player, in an interview for a paper called *De Tijd* turned out to be very inspiring:

> *“If I play the ball and want to pass it to someone, then I need to consider my guardian, the wind, the grass, and the velocity with which players are moving. We compute the force with which to kick and its direction within a tenth of a second. It takes the computer two minutes to do the same!”* (De Tijd, 2 mei 1987)

Two aspects are intriguing about this statement:

1. Any programmer will take up the gauntlet to create a program that computes the force and direction well within a tenth of a second. This is no longer an interesting challenge, as Johan Cruijff made this statement in 1987, and computing power has increased enormously since then. Nevertheless, this timing aspect plays a role in the requirements of the main exercise later on (Sect. 7).
2. The most intriguing aspect is that he actually says that every football player computes a *function*: given some parameters (guardian, wind, grass, velocity of all players) compute a pair of two values (force and direction). Hence, the brain of a football player can be modeled as a function:

   $$guardian \times wind \times grass \times players \rightarrow (force, direction).$$

   Note that in Soccer-Fun, we are going to use a different function (Sect. 3).

Having found this great source of inspiration, the challenge for us was to create an environment that can be used for teaching functional programming and design exercises for students. Right from the start, we decided that the environment had to be graphical, because seeing is believing. It should have a competitive element to stimulate students to create better solutions than their competitors, and, of course, it had to promote functional programming. This has resulted in Soccer-Fun.

The first three versions were created by the author. We have used the experience of students with the framework to alter its

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FDPE'08, September 21, 2008, Victoria, BC, Canada.

Copyright © 2008 ACM 978-1-60558-068-5/08/09. . . $5.00

design, implementation, and exercises. The fourth version contains contributions by Wanja Krah [11]. The version that is described in this paper is the current development version.

Soccer-Fun is implemented in Clean using its GUI library Object I/O [4, 5]. Where necessary we explain the syntax of Clean. We refer programmers who are familiar with Haskell [10] to [3] which is a short list of the most striking syntactical differences between Clean and Haskell.

Soccer-Fun offers a GUI (see Fig. 1) whose main screen displays a football field. Teams and referees can be selected via the menu as well as other options. During a match the football players are constantly evaluating their brain function to decide what action to perform. The results of these actions are rendered, allowing the student to see immediately the consequences of her design and implementation choices. In Fig. 1 a match is played. On the left hand (*west*) side the team called *Wanja1* is playing against the *MiniEffiesE* team (on the *east* side). The first team has been developed by Wanja Krah as a sparring-partner for student teams. The second team implements a very simple set of rules: they all run after the ball and try to kick it in the goal of the opponent. The main challenge that was set for students is to create the best footballer brain and football team and to become champion of a tournament organized at the end of the teaching term.

The remainder of this paper is organized as follows. We start with a very brief explanation of football in Sect. 2. In Sect. 3 we introduce the domain specific language (DSL) of Soccer-Fun. This is the material that the students must work with. The DSL utilizes many type features of the host language: algebraic data types, record types, function types, and existentially quantified types. Note that uniqueness types, a distinguishing feature of Clean, plays no role in programming the brain, because the framework factors out programming with effects. The brain should be a pure function. (Analogously, to name just two other functional languages, a Haskell DSL shouldn't require the `IO` monad, and an ML [13] approach shouldn't require `ref` values.) Given the DSL, we can concentrate on its semantics. This is done in Sect. 4. We explain that footballer brains make up actions, which lead to effects. These are observed by a referee who can interfere in the game in various ways. Next, we introduce a series of exercises that the students can do to learn the Soccer-Fun DSL in Sect. 5. The role of the referee is discussed in more detail in Sect. 6. It turns out that he can be used for exercises as well. In Sect. 7 we describe the main exercise that was assigned to students. We report on our experience in using Soccer-Fun in the course, but also as a vehicle for exposing pupils in secondary eduction with concepts of functional programming in Sect. 8. In Sect. 9 we discuss a number of exercises that can be given that modify Soccer-Fun, rather than programming brains. These ideas are discussed separately because we have not used them in class. Related work is described in Sect. 10, and we come to conclusions and future work in Sect. 11.

2. Rules of the Game

This section is a brief introduction to football. If you know football, then you may wish to skip to Sect. 2.1 in which the simplifications are described that we apply. For full details about the rules of the game, we refer to the official laws of the game [7] that can be found on the site of the FIFA at `www.fifa.com`.

Football is a ball game played by two teams each consisting of at least seven and at most eleven players on a field that is between 100 - 110 metres long and 64 - 75 metres wide. At the two far sides of the field a goal is placed. The duration of a match is split into two equal halves. At each half, the game is started by means of a kick off at the centre of the football field. If a team kicks off in the first half, then the opponent team kicks off in the second half. Also, at half time, teams change playing sides to neutralize possible external effects such as sunlight and weather conditions. During a match the players attempt to kick the ball in the goal of the opponent, thereby scoring a goal. If one team has scored a goal, then the game is paused and starts again with a kick off by the opponents. The winner of the game is the team that has produced most goals. If an equal number of goals is scored then the match comes to a draw.

The goalkeeper guards the goal of his team and attempts to prevent the opponent from scoring goals. To ease his task, he is the only player who is allowed to use his arms and hands. However, he can only do this legally within the penalty-area.

Football is played with one football that is supposed to be on the football field. If the ball exits the field, the game is paused and the ball must be put back in play. If a player causes the football to exit the field along one of the long edges, then an opponent can throw in the ball from the position where the ball left the field (this is the only time a field player is allowed to play the ball with his hands). If a player causes the football to exit on the short edge of his opponents goal, then the game is resumed by means of a goal kick. Finally, if a player caused the ball to exit on the short edge of his own goal, then the opponent team receives a corner kick.

In order to control whether all players adhere to the rules of the game, a referee and two assistant referees are present. The referee keeps track of the playing time, the scored goals, and can decide to pause the game because he has detected a violation of the rules. The referee can caution an offending player with a yellow or red card. Receiving two yellow cards equals receiving one red card. Receiving a red card means a direct exit from the game of the player. Depending on the offence, usually the opponent team is awarded with an (in)direct free kick or a penalty kick. A free kick is direct if a goal can be scored out of it; it is indirect if at least one other player must be involved.

In football there is a subtle offence called *offside*. Being in offside position is not a violation, and is easily defined: a player is in offside position when he is at the opponents' half of the field and *"he is nearer to his opponents' goal line than both the ball and the second last opponent"* [7]. The offence is created when:

> *"A player in an offside position is only penalised if, at the moment the ball touches or is played by one of his team, he is, in the opinion of the referee, involved in active play by:*
>
> - *interfering with play or*
> - *interfering with an opponent or*
> - *gaining an advantage by being in that position"* [7]

The referee can punish the team by rewarding the opponent team with an indirect free kick. Being in active play is a rather subjective decision by the referee that must be taken within a split second, so this decision often creates many comments from the punished team, spectators, and reporters.

2.1 Simplifications in Soccer-Fun

In Soccer-Fun, a number of simplifications have been incorporated. There is no real concept of throw in: whenever the ball exits the field along one of the long edges, it is simply placed at the position where it left the field, and the opponent team is supposed to kick the ball first (this is checked by the referee). There are no penalties or indirect free kicks. Assistant referees are absent, and the referee even lacks a body. We ignore the weather and the condition of the football field. Although players can get injured, there are no substitutes for them. All players and referee are panoptic.

3. The Soccer-Fun DSL

In this section we introduce the Soccer-Fun DSL. It consists mostly of data types describing the entities of football as explained in

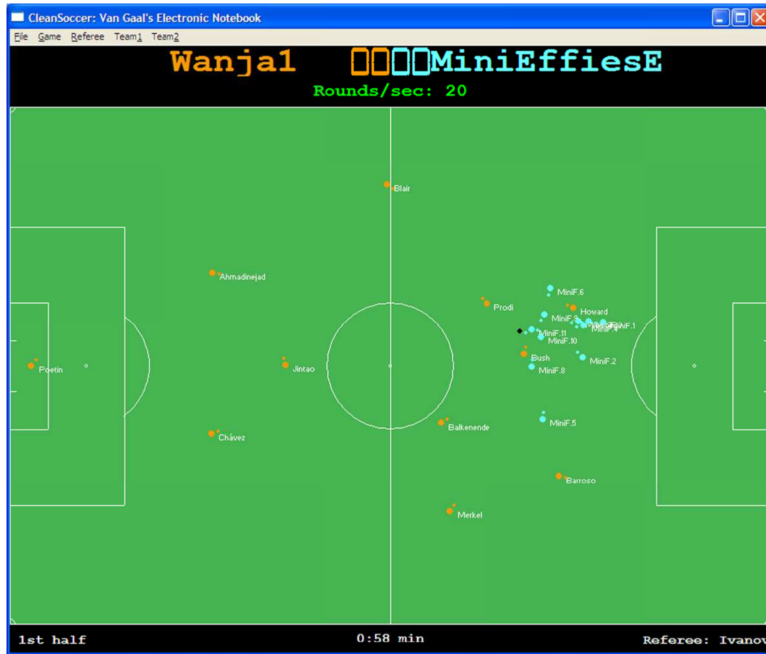

Figure 1. The Soccer-Fun framework in action.

Sect. 2 and has a very small set of basic functions for building brain functions. We start our discussion by first introducing the function that plays the pivotal role in this paper: our version of the footballer's brain. We stick to the idea of Johan Cruijff, hence it must be a function type. The full type is:

```
:: FootballerAI memory :==[1] FootballField
                              (Maybe Football)
                              Home
                              Half
                              Team
                              Team
                              TeamFootballer
                              memory
                           → (FootballerAction, memory)
```

The function type of a football player's brain is parameterized with the type of his memory. It takes eight arguments, one of which is the memory of the player, and computes two values: a single action that the player wishes to perform, and an updated memory value (for very simple brains the `:: Void = Void` type can be used). Apart from the memory, the arguments can be divided into three categories: metrics (Sect. 3.1), the whereabouts of the football (Sect. 3.2), and the description of all players (Sect. 3.3). The actions that a player can initiate are described in Sect. 3.4. A few of the most frequently used functions of Soccer-Fun are described in Sect. 3.5.

We should emphasize that we made the design decision that it is impossible for footballer brains to *communicate*, or perform any I/O operation (via files or messages or whatever means). The "intelligence" of a team is the sum of the intelligence of its players.

3.1 Metrics

In Soccer-Fun, all distances are given in metres. Because not all football fields have the same size, it is provided as a separate argument to the brain:

```
:: FootballField[2]
   = { fwidth  :: !FieldWidth       // 64m ≤ width ≤ 75m
     , flength :: !FieldLength      // 100m ≤ length ≤ 110m
     }
:: FieldWidth  :== Metre
:: FieldLength :== Metre
:: Metre       :== Real
```

The football field defines coordinates in a way that is standard for computer graphics: x-coordinates increase in value from left to right; y-coordinates from top to bottom. We distinguish between positions *on* the football field (`Position`) and positions *above* the football field (`Position3D`).

```
:: Position   = { px :: !XPos,       py :: !YPos }
:: Position3D = { pxy:: !Position,   pz :: !ZPos }
:: XPos       :== Metre
:: YPos       :== Metre
:: ZPos       :== Metre
```

The size of the goal and certain areas of a football field are provided as constants:

```
goal_width           :== 7.32
goal_height          :== 2.44
goal_area_depth      :== 5.50
penalty_area_depth   :== 16.50
penalty_spot_depth   :== 11.00
radius_penalty_area  :== 9.15
```

Players need to know at what side of the field their goal is. Their third brain argument (`:: Home = West | East`) tells this. `West` is the left-hand side of the football field, `East` is the right-hand side.

Angles are given in radians. Due to the flipped orientation of y-coordinates, angles are also flipped: the angle 0π points straight east, $\frac{1}{2}\pi$ south, π west, and $\frac{3}{2}\pi$ north.

```
:: Angle  :== Radian
```

[1] Type declarations start with `::`. `:==` Defines a type synonym. Function types have arity. Arguments are separated by whitespace.

[2] Record types are delimited by { and } and contain a non-empty list of *field* `::` *type* pairs. `!` annotates strictness.

```
:: Radian :== Real
```

Although the framework and the referee keep track of time, football players do not. They are only informed whether they are playing in the first half or second half of a match in their fourth brain argument (`:: Half = FirstHalf | SecondHalf`). The duration of a match is currently a constant value in Soccer-Fun. The user should be able to set it via the GUI in a future version.

Players and the ball move at a certain speed. As with positions, we find it useful to distinguish between speed *along* the surface of the football field (`Speed`) and *above* the football field (`Speed3D`). The speed along the surface is given by a direction in radians and a velocity in metres per second. The speed above the surface includes a velocity along the z-axis.

```
:: Speed     = { direction :: !Angle, velocity :: !Velocity }
:: Speed3D   = { speed2D   :: !Speed, speed3D  :: !Velocity }
:: Velocity :== Real
```

In Soccer-Fun, players stick to the ground and hence always have a `Speed` value, and only the football possesses a `Speed3D` value.

3.2 The Whereabouts of the Football

The second argument of a footballer's brain tells him where the football is. Because the football can be in the air, its position is given by a `Position3D` value, and its speed by a `Speed3D` value:

```
:: Football = {ballPos :: !Position3D, ballSpeed :: !Speed3D}
```

We can model the football with this value, and provide it to each brain. This is what we did in the first versions of Soccer-Fun. It turned out that this was not at all convenient for programming brains. In real football, a player can be in *possession of the ball*. In this situation a player manipulates the ball in such a way that it seems as if the ball is glued to his legs and feet. Students noticed that programming similar ball control was remarkably hard, so we decided to model this concept directly within the framework. Players can attempt to *gain* the ball. At most one player can succeed, and from that moment on he is in possession of the ball. The ball is no longer freely available in the game, and this is actually what is told by the second parameter of each football player's brain: it is a value of type (`Maybe Football`). This value is `Nothing` if some player possesses the ball, and is (`Just ball`) if the ball is freely available.

An invariant of Soccer-Fun is that the football is *either* freely available *or* it is possessed by *one* player. Hence, in order to know the whereabouts of the football each brain function needs to inspect the above parameter as well as every football player to determine where the football is. For this purpose a utility function is provided, `getFootball`, that finds the football. Its exact type is discussed below.

3.3 The Football Players

The fifth, sixth (both of type `Team`) and seventh (of type `TeamFootballer`) brain arguments tell the football player where everybody is, including himself. As said in Sect. 2.1, all players are panoptic. (It would be interesting to make the framework more realistic and constrain this knowledge to the viewing range of a player; but this has not been implemented.) The first `Team` argument contains all team members of the player, excluding himself, and the second `Team` argument contains all opponents. A team consists of one goalkeeper and ten field players and belongs to a club:

```
:: Team      = { clubName :: !ClubName
               , keeper   :: !Maybe TaggedFootballer
               , fielders :: ![TaggedFootballer]
               }
:: ClubName :== String
```

Note that the goalkeeper is a `Maybe` value because the referee can hand out red cards to any player, including the goalkeeper. Because Soccer-Fun does not include substitutes, this means that a team no longer has a goalkeeper.

A `TaggedFootballer` is a football player with a player's number. By convention, the goalkeeper has number 1, and the player's numbers of all players should be different. A `TeamFootballer` adds the clubname to such a football player. For conversion two straightforward type classes are available.

```
:: TaggedFootballer = { nr            :: !PlayersNumber
                      , player        :: !Footballer
                      }
:: PlayersNumber   :== Int
:: TeamFootballer   = { club          :: !ClubName
                      , tagged_player :: !TaggedFootballer
                      }

class[3]  toFootballer     a :: !a → Footballer
instance toFootballer     TeamFootballer
instance toFootballer     TaggedFootballer
instance toFootballer     Footballer

class    toTeamFootballer a b :: !a !b → TeamFootballer
instance toTeamFootballer (ClubName,PlayersNumber) Footballer
instance toTeamFootballer  ClubName          TaggedFootballer
```

Football players are defined with a rather extensive set of attributes.

```
:: Footballer = ∃ memory:
               { name    :: !String
               , length  :: !Length
               , nose    :: !Angle
               , pos     :: !Position
               , speed   :: !Speed
               , skills  :: !MajorSkills
               , ball    :: !Maybe Football
               , effect  :: ![FootballerEffect]
               , fatigue :: !Fatigue
               , health  :: !Health
               , events  :: !Events
               , brain   :: !Brain (FootballerAI memory) memory
               }
:: Brain ai memory = { memory :: memory, ai :: ai }
```

A football player has a `name` (need not be unique), and a `length` (in metres). He looks in the direction of his `nose`, is at a `position`, and moves with a certain `speed`. The types of the latter two fields indicate that a football player is always on the ground.

As explained earlier in Sect. 3.2, a football player can be in possession of the ball. If he is, and has position p and speed s, then `ball = Just {ballPos=`p`, ballSpeed=`s`}`; otherwise `ball = Nothing`. The function `getFootball` that was discussed in Sect. 3.2 has type:

```
getFootball :: !(Maybe Football) ![a] → Football | toFootballer a
```

It is expected to be applied to the second argument of the brain function and the list of all football players (which are (`Team`/`Tagged`)-`Footballer` values). The function `fromTeam :: Team → [TeamFootballer]` is useful to obtain all players from a team.

Soccer-Fun includes a `fatigue` / `health` model. Both are `Real` values between 0.0 (exhausted / in bad health) and 1.0 (fit / in excellent health). Actions decrease the player's `fatigue` if they are above some threshold value, and increase `fatigue` if below that threshold value. For instance, a dash decreases fatigue, while walking increases fatigue. Players can get hurt due to actions of their own or of other players, which decreases their `health` value. The yield of every footballer action is affected negatively by both values.

A football player can select three skills as `MajorSkills` value. The `Skill` type enumerates skills:

[3] Multi-parameter type constructor classes are introduced by **class**.

```
:: MajorSkills :== (!Skill,!Skill,!Skill)
:: Skill = Running | Dribbling | Rotating | Gaining  | Kicking
         | Heading | Feinting  | Jumping  | Catching | Tackling
         | Schwalbing | PlayingTheater
```

When he performs an action that is governed by a major skill, his yield will be better than average. In this way, the student can create a variety of football players easily. A number of skill-dependent functions are available in Soccer-Fun to determine whether it makes sense to perform an action. As an example, trying to gain the ball does not make sense when the ball is out of reach. The distance depends on your major skills and length:

```
maxGainReach :: !MajorSkills !Length → Metre
```

Every footballer action has an `effect`: this is what the footballer has actually done. Even though his brain may want him to run at 20m/s, his body won't be capable of doing this. Soccer-Fun takes every action into account and computes a realistic effect. This value is passed to the football player next time.

Every action has an effect, but there are also `events` that are triggered by these actions. This is the case when the player has fallen, received a reprimand from the referee, or is affected by a referee action (these are discussed in detail in Sect. 6).

```
:: Events    = { fallenDown     :: !Bool
               , reprimands     :: ![Reprimand]
               , refereeActions :: ![RefereeAction]
               }
:: Reprimand = Warning | YellowCard | RedCard
```

Finally, the most important attribute of a football player is his `brain`. A (`Brain memory`) value is a pair of his "intelligence", the (`FootballerAI memory`) function, and a matching `memory` value. The memory is encapsulated with an existential quantifier. This prohibits players to "read the mind" of other players.

3.4 Actions and Effects

The goal of the brain function is to compute an appropriate `FootballerAction` value, and perhaps update his memory. It should be noted that an action expresses only the *intention* to perform that action; as said earlier, Soccer-Fun computes the actual effect of each and every action. Soccer-Fun provides the following actions:

```
:: FootballerAction = Move Speed Angle | Feint FeintDirection
                    | GainBall         | Catch
                    | KickBall Speed3D | HeadBall Speed3D
                    | Schwalbe         | PlayTheater
                    | Tackle TaggedFootballer Velocity
:: FeindDirection   = FeintLeft | FeintRight
```

The first two actions cause a player to move: (`Move` s a) lets him move at speed s, after rotating his `nose` (and therefor his body) over angle a. Moving is most effective in the same direction as his `nose`, and least effective in direction `nose`$+\pi$. (`Feint` d) causes a player to make a feint manoeuvre either to the left or the right. This is useful for a striker when trying to sidestep a defender.

Any player can gain possession of the ball with `GainBall`. Within his penalty area, the goalkeeper can `Catch` it. The ball remains with the player until he either plays it or it is gained by another player. Note that when the player is in possession of the ball, his movements are slower.

The ball can be played via kicking (`KickBall` s) or heading (`HeadBall` s). In both cases, s is the intended new speed of the ball, which becomes freely available in the match.

The final three actions are concerned with unclean play. Performing a `Schwalbe` causes the football player to fall to the ground, which is usually followed by `PlayTheater`, hoping to convince the referee that an opponent has attacked the player. (Note that for players who perform these actions, Soccer-Fun decreases their `health` value for inspection by the referee.) Performing a (`Tackle` p v) is an attempt to bring player p to an abrupt halt. Depending on the velocity v with which this action is intended to be performed, this may cause damage to p's `health` value. Of course, all of these actions can cause the referee to reprimand the unfair player, who runs the risk of receiving a yellow or red card.

For each *action* an *effect* is defined, plus a few more:

```
:: FootballerEffect
   = Moved       Speed Angle      | Feinted    FeintDirection
   | GainedBall Success           | CaughtBall Success
   | KickedBall (Maybe Speed3D)   | HeadedBall (Maybe Speed3D)
   | Schwalbed                    | PlayedTheater
   | Tackled     TaggedFootballer Velocity Success
   | OnTheGround FramesToGo
   | Reprimanded Reprimand
   | ScoredGoal  Home
:: FramesToGo :== Int
:: Reprimand   = Warning | YellowCard | RedCard
```

The new effects are (`OnTheGround` n) which indicates that the player has fallen to the ground, and will remain lying there for n frames; when a player receives a reprimand r, then this is reported as (`Reprimanded` r); (`ScoredGoal` h) when a goal has been scored for the indicated home side h.

3.5 Other Soccer-Fun functions

In addition to the functions that have been described, Soccer-Fun has a small set of utility functions. The most frequently used are:

```
angleWithObject :: !Position !Position             → Angle
inPenaltyArea   :: !FootballField !Home !Position → Bool
goal_poles      :: !FootballField                  → (!Metre,!Metre)
```

(`angleWithObject` p_1 p_2) returns the angle between two lines that intersect at p_1, and where the first line has angle 0π, and the second line goes through p_2. The result angle can be used for rotating towards a point p_2, or for playing the football to p_2. (`inPenaltyArea` f h p) holds if position p is within the penalty area of side h of a football field f. (`goal_poles` f) yields the north and south y-coordinates of the two goal poles on a football field f, which is useful for kicking or heading the ball in that goal.

Finally, for inclusion in Soccer-Fun, a student needs to build a team as a function of type `:: Home FootballField → Team`. The two arguments specify at what side of the football field he will start playing, and what the dimensions of the football field are. These are necessary for the line-up of the players.

4. Semantics of Soccer-Fun

The state of a match is a value of type `Match` that consists of two teams, a football, a football field, a referee, the current playing half, playing time, time unit, score, random stream, and a flag that eliminates randomness (the latter two are discussed in Sect. 4.1).

```
:: Match = { team1        :: !Team
           , team2        :: !Team
           , theBall      :: !Maybe Football
           , theField     :: !FootballField
           , referee      :: !Referee
           , playingHalf  :: !Half
           , playingTime  :: !PlayingTime
           , unittime     :: !Seconds
           , score        :: !Score
           , probs        :: !RandomStream P
           , not4real     :: !Bool
           }
:: PlayingTime :== Minutes
:: TimeUnit    :== Seconds
:: Minutes     :== Real
:: Seconds     :== Real
```

```
:: Score        :== (!NrOfGoals,!NrOfGoals)
:: NrOfGoals    :== Int
:: P            :== Real  // 0.0 ≤ p ≤ 1.0
```

The meaning of a match is just an iteration of a single-step function, `stepMatch :: !Match → (![RefereeEvent],!Match)`, or, monadically, `stepMatch :: St Match [RefereeEvent]` (with `:: St s a :== s → (a,s)`) until the referee has decided that the game is over.

Note that `stepMatch` does not perform any I/O. In Soccer-Fun, this is done by a separate function that renders a `Match` value. This offers the possibility to completely compute the outcome of a match without providing visual feedback. This is particularly useful when computing the outcome of a tournament.

The definition of `stepMatch` is complicated because actions of football players can interact (for instance, two players who want to kick the ball) or contradictory (two players both trying to gain the ball); physical constraints need to be considered (a player cannot rotate over π radians when running at maximum speed, the movement of a freely available ball must be computed); the fatigue/health model must be applied; the referee interferes during the match by pausing and restarting the game, replacing and even expelling players; and so on. This rules out a simple semantics in which every player computes his action and performs it.

We have chosen the following scheme:

1. The brain of every football player computes an action. This is an uncomplicated function that needs to pass the proper arguments to each brain function, and update the player because his memory may have changed.

   ```
   playersTurn :: St Match [FootballerWithAction]

   :: FootballerWithAction
        :== (FootballerAction,(ClubName,PlayersNumber))
   ```

   We need to know who (identified by clubname and player's number) wants to perform what action, hence the result type.

2. Filter actions: *at most* one of all {`GainBall`, `KickBall`, `HeadBall`, `Catch`} actions is selected, using the random stream; only feasible `Tackle` actions are selected; actions of tackled players are neutralized.

   ```
   selectActions :: ![FootballerWithAction]
        → St Match ([FootballerWithAction],[FootballerWithAction])
   ```

   `selectActions` yields the theater playing actions and the remaining successful actions.

3. The brain of the referee computes his decisions based on the actions that he "sees". Only for the referee, the health for football players who are playing theater (determined by the first argument) is decreased, depending on their `PlayingTheater` skill.

   ```
   refereeTurn :: [FootballerWithAction] [FootballerWithAction]
        → St Match [RefereeAction]
   ```

   The referee (actions) is discussed in more detail in Sect. 6.

4. Compute the effect of all selected player and referee actions and update players and football accordingly. The function receives both the intended actions and the successful actions to inform the player that an intended action may have failed.

   ```
   performActions :: [FootballerWithAction]
                     [FootballerWithAction] [RefereeAction]
                  → St Match Void
   ```

   This is the "heart" of the `Match` transition function.

5. Advance playing time, which is a trivial function.

   ```
   advanceTime :: St Match Void
   ```

   It is the task of the referee to decide whether a match has ended. This implies that the playing time can become negative, but this is what happens in real football matches as well.

Combining the above functions gives the following top-level definition of `stepMatch`[4]:

```
stepMatch :: St Match [RefereeAction]
stepMatch = do actions              ← playersTurn                        1.
               (theater,pActions) ← selectActions actions                2.
               rActions             ← refereeTurn theater pActions       3.
               performActions actions pActions rActions                  4.
               advanceTime                                               5.
               return rActions
```

The `RefereeAction`s are returned for rendering.

`performActions` is the heart of the Soccer-Fun semantics. This function also has a heart which is the following transition function:

```
performAction :: !FootballerWithAction → IdFun (!RandomStream P
                                                ,!Maybe Football
                                                ,![TeamFootballer]
                                                ,![TeamFootballer]
                                                )
:: IdFun st :== st → st
```

which computes the effect of each action and applies it to the football and all players of both teams. After these actions have been computed, and the ball has become freely available, its next position and speed is computed. Finally, all referee events are applied to the new state of the match.

As an illustration of `performAction`, we show the `GainBall` rule.

```
performAction (GainBall,player_id) (probs,mball,team1,team2)          1.
♯ (team1,team2)= splitAt (length team1)                                2.
                   (unbreak ( map loseBall uneq1                       3.
                            , new_self                                 4.
                            , map loseBall uneq2                       5.
                   )        )                                          6.
= (probs,Nothing,team1,team2)                                          7.
where                                                                  8.
  players      = team1 ++ team2                                        9.
  ball         = getFootball mball players                            10.
  (uneq1,self,uneq2)                                                  11.
               = break (identify_player player_id) players            12.
  me           = toFootballer self                                    13.
  new_ball     = {ball & ballPos   = {zero & pxy=me.pos}              14.
                       , ballSpeed = {zero & speed2D=me.speed}}       15.
  new_me       = {me   & effect    = [GainedBall Success]             16.
                       , ball      = Just new_ball}                   17.
  new_self     = toTeamFootballer player_id new_me                    18.
```

The definition uses two very useful helper functions:

```
break   :: !(a→Bool) ![a] → ([a],a,[a])
unbreak ::               !(![a],a,[a]) → [a]
```

(`break` $p\ xs$) breaks xs into (as, b, cs), such that $\forall a \in as.\neg(p\ a)$, $(p\ a)$, and $\neg(p\ (\mathtt{hd}\ cs))$ if cs is not empty. (`unbreak` (as, b, cs)) reconstructs xs.

The rule states clearly that the player who gains the ball really has it in his possession (line 17) and that this effect is reported (line 16). A ball in possession adapts position and speed of its owner (lines 14-15). No other player has the ball (line 3 and 5).

4.1 Predictable Games

Soccer-Fun uses a stream of pseudo-random values for several purposes. In the above section we have seen that they are used to select at most one of simultaneous `FootballerActions` {`GainBall`, `KickBall`, `HeadBall`, `Catch`}. They are also used in `performAction` to

[4] Although Clean does not support **do** notation, we use it for readability.

increase realism. The effect of an action can deviate from what was intended. The amount of deviation depends on the skill of the player, so this should be taken into account.

It is convenient for students to switch off this added bit of realism when they are testing the brains that they have implemented. In the Soccer-Fun GUI this is done by selecting the *Predictable* command, which toggles the `not4real` field of a `Match`. In such a predictable setting no deviations are computed, and randomness is only used to select actions that are mutually exclusive.

It should be noted that the outcome of a game is fully determined by the initial random seed value (which is derived from the time), and the two initial `Teams`. For replaying an entire game we only need these values. This characteristic is used within Soccer-Fun in several ways: the student can *step* through an entire game without affecting the outcome; halting and continuing a game also does not affect the result. The rendering time does not change the outcome as well, hence the result of a game can be computed without any visual feedback. In earlier versions of Soccer-Fun, when the referee dialog popped up to inform of a decision, the user could ask for a replay including a future computation. This is particularly helpful in case of determining whether a player in offside position is in active play. The current version of Soccer-Fun does not have this feature, but this is planned to be re-integrated.

5. Train the Brain

Soccer-Fun is well suited to set up a range of exercises that lead in a natural way to the final task of creating a successful team. In order to familiarize the students with Soccer-Fun, it is useful the start with a number of small basic tasks (Sect. 5.1). Following this we can concentrate on developing strategies that determine the behavior of each team player (Sect. 5.2). For the finishing touch, all teams should obey the rules of the game and respond correctly to the decisions of the referee. We defer this subject until Sect. 6 in which the referee and his decisions are presented in detail.

In our experience, it is very helpful to have students first make an informal description of the brain function that needs to be created, and then to create the concrete Soccer-Fun code. The informal description is expressed in structured natural language, as a collection of guarded equations (**if** *cond* ⇒ *action*). If the brain function does not require a memory, then *action* is just a footballer action, otherwise it is a pair *(action,memory)*. The Soccer-Fun code closely follows the structure of the informal description, but evidently needs to fill in all the (computational) details. In this section we adopt this approach.

5.1 Basic exercises

The exercises that we discuss here are small, and can be done in class together with the students.

5.1.1 Run a (number of) lap(s)

Let the footballer start at the north west corner of the football field. Make him run laps in clockwise direction within the boundaries of the football field.

One might think of the following naïve solution:

if I am close to a corner ⇒ rotate $\frac{1}{2}\pi$ clockwise
otherwise ⇒ run ahead

There are two problems with this scheme: first, as soon as a corner is reached, the first condition remains true, and second, it assumes that the player is able to perform a perfect rotation over $\frac{1}{2}\pi$ radians. A slightly more complicated scheme must be implemented which memorizes the corner to which the player needs to run:

if I am close to a corner
  if nose in right direction ⇒ (run ahead,nextcorner)
  otherwise ⇒ (rotate, corner)
otherwise ⇒ (run ahead,corner)

This simple brain needs to memorize the desired corner:

```
:: Corner = NorthWest | NorthEast | SouthEast | SouthWest

nextCorner :: Corner → Corner
dirCorner  :: Corner → Angle
```

`nextCorner` computes the next corner the player has to go to. `dirCorner` yields the direction the player's nose has to face to reach that point. With these functions we can define the footballer's brain:

```
laps :: FootballerAI Corner
laps {flength = fl,fwidth = fw} _ _ _ _ _ me corner
| close_to_corner
    | nose_ok       = (run,nextCorner corner)
    | otherwise     = (rotate,        corner)
| otherwise         = (run,           corner)
where
    {pos,nose}      = toFootballer me
    {px,py}         = pos
    diff_corner     = dirCorner (nextCorner corner) - nose
    close_to_corner = case corner of
                        SouthEast = px ≥ fl - 5.0 && py ≥ fw - 5.0
                        NorthEast = px ≥ fl - 5.0 && py ≤ 5.0
                        SouthWest = px ≤ 5.0      && py ≥ fw - 5.0
                        NorthWest = px ≤ 5.0      && py ≤ 5.0
    nose_ok         = abs diff_corner ≤ 0.01
    rotate          = Move zero diff_corner
    run             = Move {zero & velocity = 5.0} zero
```

With this brain, a footballer starts rotating as soon as he reaches the desired corner, and is not facing the right direction. When he faces the right direction, or is not close to a corner, he runs ahead.

Variations Parameterize `laps` in such a way that the player runs in either clockwise or counter-clockwise direction; limit the number of laps to a given value; generalize the brain in such a way that he can start at any location on the football field and facing any direction.

5.1.2 Run to a location

A useful functionality of any player is to run to a certain location (for instance his default position, or to gain a freely available ball, or to try to gain the ball from an opponent). The task is to have the footballer run to a specified location which can be stored in his memory or passed as an additional argument. For this particular exercise the latter approach is preferred because it expresses more clearly the constant nature of the position. Storing the location in memory has as advantage that it can be changed according to circumstance: in that case the player automatically runs to the new location. Let's assume the value is available as a separate argument. Now a naïve definition is sufficient:

if close to position ⇒ stand still
if nose in right direction ⇒ run ahead
otherwise ⇒ rotate to position

To implement this brain, the students get to know the useful function `angleWithObject` (Sect. 3.5).

```
fix :: Position → FootballerAI m
fix point _ _ _ _ _ _ me memory
| close_to_point    = (stand_still,memory)
| nose_ok           = (run_ahead,  memory)
| otherwise         = (rotate,     memory)
where
    {pos,nose}       = toFootballer me
    angle_with_point = angleWithObject pos point
    diff_nose        = angle_with_point - nose
```

```
close_to_point   = dist pos point < 3.0
nose_ok          = abs diff_nose  < 0.1*pi
stand_still      = Move zero zero
run_ahead        = Move {zero & velocity = 10.0} zero
rotate           = Move {zero & velocity = 10.0} diff_nose
```

In order to reach a position at a given `point`, the brain tells the player to first rotate towards that point, and when looking in that direction to run to the point. It is satisfied as soon as the player is within three metres distance of the given point.

Variations Same as above, but place position in memory; instead of a single position in memory, use an infinite list of positions that need to be visited in sequence; use the latter version to implement the `laps` from Sect. 5.1.1.

5.1.3 Passing the ball

In football, players need to pass the ball to other players, hence it makes sense to implement a brain that passes the ball correctly to another player. The task is to implement $1 < n < 11$ football players who are standing on the football field who need to pass the ball to each other. A player to whom the ball is passed must first gain the ball before passing it to the next player. For this exercise the student must switch off the computation of deviations of footballer actions as described in Sect. 4.1 because otherwise his players will need to go look for the ball in case it has gone wide.

The footballer's brain receives as parameter the position of the next player to whom the ball should be passed after he has gained possession of the ball. To increase effectiveness, the player rotates towards the next player. The brain does not require a memory, and has a concise specification:

if I possess the ball
    if I look at target ⇒ pass the ball
    otherwise ⇒ rotate
if I can gain the ball ⇒ gain
otherwise ⇒ stand still

This brain needs the (`Maybe Football`) argument and both `Team` values to get the whereabouts of the `Football`. Its new actions are `GainBall` and `KickBall`. To determine whether he possesses the ball, he only needs to inspect his `ball` field. To know whether the ball is within gaining reach, he can use the `maxGainReach` function (Sect. 3.3). The interesting question is how hard to kick the ball in order to allow the next player to gain it. When playing the ball over the field there is a considerable friction. A simple rule of thumb is to kick the ball with a velocity that is equal to the distance to the target multiplied by a factor five. All in all, we get the following brain:

```
kick :: Position → FootballerAI Void
kick target _ freeball _ _ team opponents me memory
| possess_ball
    | look_at_target  = (pass_ball,   memory)
    | otherwise       = (rotate,      memory)
| close_to_ball       = (gain,        memory)
| otherwise           = (stand_still,memory)
where
    {pos,nose,length,ball,skills}
                      = toFootballer me
    possess_ball      = isJust ball
    v                 = 5.0*(dist pos target)
    {ballPos}         = getFootball freeball
                            [me:fromTeam team ++ fromTeam opponents]
    close_to_ball     = dist pos ballPos ≤ maxGainReach skills length
    angle_with_target = angleWithObject pos target
    diff_target       = angle_with_target - nose
    look_at_target    = abs diff_target < 0.1*pi
    gain        = GainBall
    stand_still = Move zero zero
    rotate      = Move zero diff_target
    pass_ball   = KickBall {zero & speed2D =
                           {direction=angle_with_target,velocity=v}}
```

Variations Instead of playing the ball over the field, play it through the air; in that case, players should pass the ball via heading.

5.2 Strategy exercises

In a team, players need to act according to a strategy, i.e. a number of rules they adhere to in order to play good football. In this section we discuss a number of exercises that are related to strategy.

5.2.1 Active line-up

The line-up of players is an important aspect of football. Players should place themselves on the field in such a way that their team can control a significant part of the football field. This can be achieved by assigning to each player a region of the football field that he should control. His default strategy then is to move to the center p of this region when there is nothing else to do. This is just a matter of evaluating (`fix` p) that was presented in Sect. 5.1.2.

```
brain :: Position → FootballerAI Memory
brain p field freeball home half team opponents me m
...
| otherwise = fix p field freeball home half team opponents me m
```

Variations Make the line-up dynamic, depending on whether your team is in possession of the ball. If it is, then the line-up should advance towards the goal of the opponent. If the opponent team is in possession of the ball, then the line-up should withdraw towards their home goal.

5.2.2 Gaining the ball

In a team you need to have an agreement on who is going to try to gain the ball if it is not possessed by your team. A simple rule is that the player who is closest to the ball will attempt to gain the ball. First, he needs to get to the ball, and second, when sufficiently close to the ball, he should gain the ball.

```
gain :: FootballerAI m
gain field freeball home half team opponents me m
| close_to_ball  = (GainBall,m)
| closest        = fix ballPos field freeball home half
                           team opponents me m
where
    {pos,length,ball,skills}
                 = toFootballer me
    possess_ball = isJust ball
    {ballPos}    = getFootball freeball
                       [me : fromTeam team ++ fromTeam opponents]
    close_to_ball= dist pos ballPos ≤ maxGainReach skills length
    closest      = dist pos ballPos ≤
                     minList [  dist (getPosition fp) ballPos
                             \\ fp ← fromTeam team
                             ]
```

Note that the situation may arise that there are several candidates to go to the ball, because the comparison uses ≤. However, using < may result in a situation that nobody goes to the ball. This is clearly less desirable than having more players run to the ball.

Variations Increase the number of players who try to gain the ball. Anticipate earlier on gaining the ball: if the ball is moving towards you, move forward and gain it.

5.2.3 What to do with the ball

When a player is in possession of the ball, he must decide what to do: he can pass the ball to another player, he can dribble, or he can try to score a goal.

Let's work out a brain that decides to pass the ball to the first team player who is closer to the goal. If no such player is available, then the player himself is in the best position. If he is to far away from the goal, he dribbles towards the goal, otherwise he kicks the ball towards the goal. The informal scheme is:

if I am in best position
  if I am close to goal ⇒ kick ball in goal
  otherwise ⇒ run to goal
otherwise ⇒ pass the ball to player in best position

and the corresponding Soccer-Fun realization:

```
playball :: FootballerAI m
playball field freeball home half team opponents me m
= action field freeball home half team opponents me m
where
    action     = if in_best_position
                   (if near_goal (kick goal)
                                 (fix  goal)
                   )             (kick best)
    in_best_position
               = isEmpty better
    better     = [pos \\ {pos} ← players | dist pos goal < d_goal]
    best       = hd better
    goal_x     = if (home == East) zero field.flength
    goal       = {px = goal_x, py = 0.5*field.fwidth}
    d_goal     = dist (toFootballer me).pos goal
    near_goal  = d_goal ≤ 20.0
    players    = fromTeam team
```

Variations Take into account whether it is safe to pass the ball to a team player (consider number of opponents between you and team player and/or the number of defenders of that player). Alternatively, pass the ball through the air and make sure it ends exactly at the feet of the team player.

5.2.4 Offside

Despite its simplicity, the above strategy of passing the ball to players who are in better positions is very effective. However, it is also too simplistic because it overlooks the offside rule that was mentioned in Sect. 2. To repeat, a player is in offside position when he is at the opponents' half of the field and is closer to the goal line of his opponents than both the ball and the last two opponents. It is not hard to define a predicate that defines this:

```
offside :: Field (Maybe Football) Home Team Team Footballer → Bool
offside field freeball home team opponents me
              = home == West && px > maxList metrics ||
                home == East && px < minList metrics
where
    {px}      = (toFootballer me).pos
    pxs       = sort [  (toFootballer fp).pos.px
                     \\ fp ← fromTeam opponents
                     ]
    x_last_2  = if (home == West) ((reverse pxs)!!1) (pxs!!1)
    {ballPos} = getFootball freeball
                    [me : fromTeam team ++ fromTeam opponents]
    metrics   = [0.5*field.flength, ballPos.px, x_last_2]
```

The task is to adapt the brain of Sect. 5.2.3 in such a way that the ball is passed only to team players who are not in offside position.

Variations The *offside trap* is a well-known defensive strategy in which a team deliberately places strikers of the opposing team in offside position by moving forward just before he is given the ball by his team players. Prevent football players from running into the offside trap, and let defenders and goalkeeper open up the offside trap for strikers.

5.3 Discussion

In this section we have presented a range of exercises that can be used to create footballer brain functions. They illustrate that Soccer-Fun is suitable for an incremental approach. This stimulates students to continue improving their teams and challenges them to invent a better set of rules for their players' brains. The use of structured natural language expressions as "sketches" of the brain function is helpful to let the students think about the brain function without getting swamped in the details of programming.

The topics that are covered in the exercises cover mainly working with structured data types such as algebraic types, record types, and lists. One can use the framework to illustrate applications of more advanced list processing tasks as well as working with tree structures. In particular, the exercises that concern implementing strategies are suited for this purpose. We give two examples.

The first example teaches students point-free programming with lists. When a football player has a *plan*, he executes that plan until it is finished. When finished, he makes a new plan. A player can make a tactical decision. These need not be the same as the football actions, so they should be modeled by a new algebraic type. Each alternative represents a possible decision. Now the player needs to decide what to do. This can be expressed elegantly in a functional style: first, *concatenate* a list of all possible decisions; second, *map* a weight function that computes the *success rate* $0 \leq r \leq 1$ of a decision; third, *filter* out all decisions below a certain threshold value; fourth, *select* the maximum element; fifth, *map* the decision to a list of footballer actions.

The second example uses a version of *min-max trees* to make decisions. The student writes a weight function that computes the *desirability* of a complete match state. Next, he constructs a tree structure, whose nodes are complete match states, and that have an arbitrary number of child trees. The root of this tree is the current match state. Its children are computed by applying footballer actions to their parent state. The next level of children are computed by applying opponent actions to their states. This continues until some fixed depth. The best decision is the root decision that is on a path from the root to a state such that the desirability is maximal for the player's team, and minimal for the opponent team.

6. The Referee

In Sect. 4 we have explained where the referee comes in, but we have not yet properly introduced him. This is done in this section.

Just as a `Footballer`, a referee has a brain. In fact, besides a name, that is all he has:

```
:: Referee = ∃ memory:
             { name  :: !String
             , brain :: !Brain (RefereeAI memory) memory
             }
```

The brain function of the referee is similar to that of a footballer, and it receives almost the same list of arguments:

```
:: RefereeAI memory :== PlayingTime
                        TimeUnit
                        FootballField
                        (Maybe Football)
                        Half
                        Team
                        Team
                        [FootballerWithAction]
                        (memory,RandomStream P)
    → ([RefereeAction], (memory,RandomStream P))
```

Because the referee is in charge of the time he receives the intended playing time and also the time unit. The latter can be relevant in case the referee wants to replay part of the game, or even predict

the future. The third up to the seventh arguments are the same as for a footballer brain (the Home argument is missing because it does not make sense for a referee). The referee needs to judge the actions of the players, which are provided to him in the eighth argument. Finally, a referee uses and updates a memory, and consumes the random stream of the match.

The referee monitors the game and interferes in case of fouls or other reasons. The interference is expressed as a value of type RefereeAction, which is a rather large collection of algebraic data constructors. Seven are concerned with detecting a foul by a specific TeamFootballer *tf*: Hands, OwnBallIllegally, DangerousPlay, Offside, and (Tackle/Schwalbe/Theater)Detected. Such a player receives a Reprimand *r* as (ReprimandPlayer *tf r*). Five actions are concerned with the duration of the match: (Pause/Continue)Game, EndHalf, GameOver, and (AddTime *t*), with *t* minutes of extra playing time. When the referee registers the scoring of a goal by a team, then this yields (Goal *t*) with *t* a value of type :: ATeam = Team1 | Team2. Here, Team1 is the team that started the match on the west side of the football field, and Team2 is the other team. After a goal has been scored by team *t*, a (CenterKick (other *t*)) is granted, where other toggles its argument value. The game can be resumed by a team *t* at a position *p* with a (DirectFreeKick *t p*) and (ThrowIn *t p*) or by a (GoalKick *t*) and a (Penalty *t*) (the latter is only registered, planned for future implementation). The last resumption is via a (Corner *t e*) where *e* is an :: Edge = North | South value. In case of resumptions, the referee usually drives the players away to ensure that one player is in possession of the ball, and all others stay at a reasonable distance. This updates the positions, directions, speeds, and ball possessions of teams t_1 and t_2 and is a (ReplaceTeams t_1 t_2) action. Not every foul causes game suspension if the infringed team *t* would suffer from pausing the game. In that case the referee decides to give that team (Advantage *t*). Finally, for training sessions (Sect. 6.1), the referee is used to give feedback to the performance of the student's implementation of a task. This is a text *t* in (TellMessage *t*).

In the exercises that were discussed in Sect. 5, we have neglected referee actions. If you want to create a proper football team, then you need to obey the actions of the referee. This can be done in a similar style: players should check whether they are entitled to play the ball (this is just an extra guard). This only has to be done for (Center/Goal/DirectFree)Kick and ThrowIn, Penalty, and Corner.

6.1 The referee as a coach

The referee constantly monitors a match. Such a concept is also very useful to provide the student feedback about the performance of his implementation of a task. For instance, in case of the running laps exercise (Sect. 5.1.1), a referee can check whether the footballer is constantly close to an edge of the football field. If he is not, he issues a message (using the TellMessage referee action).

We have used this idea to define a number of specialized referees. They can monitor the following exercises:

1. A number of dummy players are standing still on the field. Another player is standing near the west goal. He has to dribble in a slalom fashion towards the east goal and score a goal to end the exercise. The referee checks whether all dummies are overtaken correctly and if the ball has been kicked in the goal.
2. An exercise similar to 5.1.3: a number of players need to pass the ball to each other. The referee monitors whether the ball arrives correctly at each player.
3. A number of opponent players are running over the field in north-south direction and vice versa. On opposite sides there are two players. The west side player needs to kick the ball to the east side player over the football field in such a way that it is not gained by one of opponents.
4. A number of dummy team players are placed on the football field. Some of them are covered by opponents, some are not. It is the task of another team player to pass the ball to a team player who is not covered by an opponent.
5. This is a special exercise for training the brain of a goalkeeper. He is standing in the goal area, and is surrounded by a number of opponents who pass the ball to each other. It is the task of the goalkeeper to move in such a way that none of the opponents can kick the ball to the centre of the goal without touching the goalkeeper. If he fails, the referee detects this, and shows the error by allowing the opponent to kick the ball in the goal.

The concept of a monitoring referee allows students to work on exercises until the referee is satisfied. For such exercises, they need less supervision, and it also reduces the examination time of the lecturer. He can run the exercise with the desired referee, and only needs to check whether the exercise passes or fails. Note that in combination with the fact that the framework can run independently of rendering, this can be done very quickly.

7. The Main Exercise

The main exercise that was given to students was:

Design and implement your own football team. Do this within an implementation (*.icl*) module with your name, and that exports (*.dcl*) a function with the name Team*MyName* and following type:

```
definition module MyName

import Footballer

TeamMyName :: !Home !FootballField → Team
```

You can chose the initial line-up of your players. A team consists of one goalkeeper and ten field players. Incorporate your team in Soccer-Fun by adding the following lines in module Team.icl:

```
implementation module Team

import MyName                    // import your module

allAvailableTeams = [ Team_Wanja
                    , TeamMyName  // add your team to team list
                    ]
```

The footballers *must* comply to the following requirements:

Goalkeeper: He is not allowed to leave the *penalty area*. If the ball is reasonably within reach, then he must intercept it. "Reasonably within reach" means that if the goalkeeper can, in principle, reach the ball sooner than one of his opponents, he is obliged to play the ball. This depends on the strategy of his team players: the goalkeeper is not demanded to interfere with his team players.

Field players: The field players are not allowed to run after the ball in a group. Instead, they are required to spread over the football field in a reasonable way[5]. A line-up is reasonable when each player controls an area of the football field, and that the pairwise intersection between any two areas is nil or very small. They need to take these positions, which may depend on the state of the match (e.g. attacking and defending). Field players are not allowed to be in possession of the ball for a long period. They are obliged to pass the ball to other players if this is reasonable: if a team player is in a better position than the player himself, and is in position, then the ball must be passed to that player.

[5] This requirement is included to prevent students to submit a simple minded solution. A strategy in which all players run after the ball is inferior, but would be legal if not prevented in this way.

Efficiency: Every football brain must be sufficiently efficient, i.e.: if the created team is playing against itself, then the computation of all football actions of all players must be done within $\frac{1}{20}$ second. You can check this by means of the *frame-rate indicator*: this is the number that is displayed behind the text `Rounds/sec:` in the GUI of Soccer-Fun. This number is not allowed to decrease below 20 (except when the referee dialog pops up).

Functional style: The implementation is also judged on "functional style". This is done as follows: you receive 20 *bonus* points. For each piece of code in your implementation for which an alternative exists that uses functions and data structures from `StdEnv` you lose one bonus point.

After all assignments have been submitted and reviewed, there is a tournament in which all teams compete against one another. There are two champions: the winner of the competition according to customary tournament rules *and* the winner of the "functional style" prize, which is the contribution with the largest number of remaining bonus points. Both champions receive a suitable award.

8. Experience

We have used Soccer-Fun in two different contexts: first, in the compulsory, second year bachelor course "Abstraction and Composition in Programming" (Sect. 8.1), and second, for promotional activities for secondary education (Sect. 8.2). We also reflect on our other experiences (Sect. 8.3).

8.1 Academic education

The students are second year bachelor students in computer science. They have had training in imperative programming (C) and object orientation (Java). The course "Abstraction and Composition in Programming" is an introductory course in functional programming, using Clean. As is apparent from the DSL of Soccer-Fun as described in Sect. 3, one can not expose the students immediately to it because it uses many language features. However, many of the basic exercises that have been discussed in Sect. 5 can be introduced during the course after 5-6 lessons. The students need to learn a lot of new concepts (see the summary in Sect. 1). In our experience, the best approach to let the students work with Soccer-Fun is to gradually work out increasingly complicated exercises, interwoven with the more 'traditional' exercises. In this way students get to learn Soccer-Fun as well as the foundation that is required for functional programming. This is necessary, because Soccer-Fun does not contain many functions, and hence students need to work out most of the functionality themselves.

At the end of the course the students need to do the main exercise. The course is ended with a tournament in which all submitted teams that meet the requirements of the final exercise compete. This is done together with all students. This is a nice conclusion of the course. It always had the atmosphere of a real football tournament. The tournament finally yields a champion, and (s)he as well as the "functional style" champion are awarded.

8.2 Secondary education

We have used Soccer-Fun for promotional activities to interest pupils from secondary education to study computer science at the Radboud University Nijmegen. Pupils take part in a full day program. During this day they receive a crash course on functional programming. Most pupils have some experience in mostly imperative programming languages, but functional programming is entirely new to them. They also get a brief introduction to Soccer-Fun. We have developed a number of examples of fairly simple teams that are very similar to the exercises described in Sect. 5. In these examples there are many opportunities to improve their behavior. By paying a lot of attention to the examples using the guarded equations style, and point out the similarity with the Clean code, they should be able to make small improvements to one of the examples.

Pupils respond enthusiastically to Soccer-Fun. They appreciate the example implementations, and usually come up with improvements quickly. It turns out that really creating these improvements, and make them work, gives them a hard time. We think we can improve on this by constructing another DSL on top of the current Soccer-Fun that is much closer to the guarded equations. Pupils need more basic functionality, and this should be provided as well.

8.3 General experience

Although a framework such as Soccer-Fun could have been realized as a teaching vehicle for imperative or object oriented programming, we think that it displays a number of interesting functional language aspects. Soccer-Fun makes good use of the type system to restrict the set of admissible brain functions that a student might come up with. The student can not create a brain function that uses effects, simply because its type does not allow it. We can use the programming tasks that were described in Sect. 5 also as a framework to teach more classic programming problems such as working with lists and manipulation of decision trees. Soccer-Fun itself is an illustrative example of a typical functional programming approach: use a domain specific language to formulate solutions, and define an interpreter to execute the solution.

Soccer-Fun improves the programming skill of most students because they *want* to create a better team and therefor they are motivated and required to express their ideas in a functional language. The final task does not prescribe how to do this, so students actively need to come up with solutions. Talented students can show how well they understand functional programming. Less skilled students can achieve acceptable results.

9. Alternative exercises

So far we have focussed on the original task: to create a brain for a footballer that can win a tournament. However, Soccer-Fun itself can also be used for alternative exercises. Here we make a few suggestions. These have not been tested in practice.

- In a similar way as a referee constantly monitors a football game, one can extend the framework with a *commentator* who provides sensible information about the match. The text that he outputs can be displayed in the Soccer-Fun GUI.
- One can collect and display match statistic such as the percentage of time that a team is in possession of the ball, number of goals, corners, penalties, fouls, and so on.
- The current rendering of Soccer-Fun is kept very simple: football players are depicted as filled circles with a small circle 'orbiting' around them to indicate the direction of their nose. Next to the player their name is displayed. The 'camera' is right above the field. If the ball is higher, then it is rendered larger than when it is on the field. A first improvement is to render the game in $2\frac{1}{2}$ dimensions. In that way, one can follow the movement of the ball through the air much better.

10. Related Work

Soccer-Fun is a framework in which strategies for footballers are programmed in the form of a function.

An exemplary framework, that is targeted at teaching object orientation and Java in particular, and with slightly less peaceful intentions, is Robocode [1] by Mathew Nelson in which you program a military tank that drives around on a square area, together with other military tanks. Each tank executes an algorithm which

goal it is to eliminate other tanks by firing granades at each other, and hopefully survive longest and become champion. Robocode is around for quite a while (since late 2000). It has very attractive graphics and sound effects. It effectively uses the OO paradigm to quickly get programmers up and running with their first tank. In the past, we have used Robocode ourselves for similar promotional activities as described above. From our experience in using Robocode, we know that such a framework can be a very effective teaching tool, as well as stimulating and fun for participants.

Related to Soccer-Fun, but with entirely different technology is RoboCup [2]. It is the aim of RoboCup that hardware robots compete with humans in a football match. In this project also simulator software, the RoboCup Soccer Simulator, is available in which you can create your own robots.

Going from simulated robots to software robots is a small step. Yampa [9] is a functional reactive programming language in which robots can be created. Connecting Yampa with hardware robots is the language Dance [8], which is a high level language based on Yampa and *Labanotation*, which is a formalism to denote humanoid movement. We are not aware of any project that uses Yampa or Dance to play football in either a competitive or educational setting, but one would imagine that this is possible. We speculate that a specification of a footballer is more geared towards physical movement, which is an avenue that we have not explored within Soccer-Fun.

11. Conclusions and Future Work

In this paper we have presented Soccer-Fun, which is a functional framework in which the brain of a footballer is expressed as a pure function. Soccer-Fun offers a GUI in which teams of football players are equipped with these brains to play football games. We have developed Soccer-Fun and used it in education for the past four years. Soccer-Fun is stimulating because it covers a well known problem domain, and offers a GUI that despite its simplicity, effectively shows what the result of executing the brain functions is. Students get direct feedback about the performance of the brains that they have created. A special feature of Soccer-Fun is that it's model of referees is suited to monitor the performance of student exercises, and provide feedback to the student. Based on our experience, we think Soccer-Fun is a welcome addition to the repertoire of functional programming exercises for any introductory course in functional programming.

We have used Soccer-Fun also for pupils from secondary education. Here, the use of natural language specifications in the form of guarded equations is very helpful: they capture the intuition of a possible implementation quickly, and can also serve as a basis for a real implementation. This only works when a sufficient number of example implementations are provided, that they can use for small extensions and changes. We have noticed that although pupils have no problem in improving examples of players' brains with the informal notation, they do have problems with the concrete realization. We want to eliminate this by providing a higher-level of abstraction over the current Soccer-Fun DSL.

Other future work covers creating exercises that are concerned with Soccer-Fun itself, rather than programming brains, as explained in Sect. 9. Not all standard football situations (penalty, indirect free kick, and so on) are currently handled by Soccer-Fun. This needs to be incorporated. We want to develop more referees that act as judges of exercises, as explained in Sect. 6.1. We wish to improve the rendering, in such a way that a user can choose between different kinds of rendering schemes. We wish to experiment with an alternative semantic model, in which the rather complicated computation of all effects of all player actions in one big step is simplified to a fair scheduling of single player actions. Another experiment is to drop panoptic players, and instead introduce viewing ranges for both players and referees. Note that this does not change the types, and is also easy to implement, but it does radically change the footballer brains: they no longer "see" all positions of all players, and it is likely that they need make a mental image of the possible locations of all players.

It is clear that the Soccer-Fun project has not reached its final form yet. We have many ideas for improvements and experiments and want to get more feedback from students. The ultimate goal, however, of Soccer-Fun is to let everybody experience that functional programming is fun.

Acknowledgments

The author would first all like to thank Rinus Plasmeijer, who is co-teacher of the course "Abstraction and Composition in Programming" for his enthusiasm and inspiration. Wanja Krah has developed parts of the fourth version of Soccer-Fun during this master thesis project of the computer science study of the Hogeschool Brabant, The Netherlands. We are grateful to the comments and ideas of students who have used Soccer-Fun. We thank the reviewers for their constructive comments.

References

[1] The Robocode site. http://robocode.sourceforge.net/.

[2] The RoboCup site. http://www.robocup.org/.

[3] P. Achten. Clean for Haskell98 Programmers – A Quick Reference Guide –. Available at: `http://www.st.cs.ru.nl/papers/-2007/CleanHaskellQuickGuide.pdf`, July 13 2007.

[4] P. Achten and R. Plasmeijer. Interactive Functional Objects in Clean. In C. Clack, K. Hammond, and T. Davie, editors, *Proc. of the 9th International Workshop on the Implementation of Functional Languages, IFL 1997, St.Andrews, UK, Selected Papers*, volume 1467 of *LNCS*, pages 304–321. Springer Verlag, Sep 1998.

[5] P. Achten and M. Wierich. A Tutorial to the Clean Object I/O Library - Version 1.2. Technical Report CSI-R0003, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands, Feb. 2000. 294 pages.

[6] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph writing. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 364–384, London, UK, 1987. Springer Verlag.

[7] Fédération Internationale de Football Association. *Laws of the Game 2007/2008*. FIFA-Strasse 20, 8044 Zürich, Switzerland, July 2007. http://www.fifa.com/.

[8] L. Huang and P. Hudak. Dance: A declarative language for the control of humanoid robots. Technical Report YALEU/DCS/RR-1253, Yale University, August 2003.

[9] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In J. Jeuring and S. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, Oxford*, volume 2638 of *LNCS*, pages 159–187. Springer Verlag, 2003.

[10] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[11] W. Krah. Eindverslag Project: CleanShooter. Afstudeerstage, Academie voor ICT en Media van de Avans Hogeschool te Breda at Radboud University Nijmegen, March 16 2007. In Dutch.

[12] R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.

[13] J. Ullman. *Elements of ML Programming – ML97 Edition*. Prentice Hall Inc., 1998.