*JFP* **21** (1): 1–19, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000055 First published online 19 March 2010


# EDUCATIONAL PEARL

## *The Soccer-Fun project*

PETER ACHTEN
*Model Based System Development, Radboud University Nijmegen, The Netherlands*
(*e-mail:* P.Achten@cs.ru.nl)

**Abstract**

In the Soccer-Fun project, students program the brains of football players in a functional language. Soccer-Fun 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 becomes the champion of a tournament; the best written code is awarded with a prize; students are 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.

### 1 Introduction

This paper is an adapted version of Achten (2008), and reflects the current status of the Soccer-Fun project. Not all details of the framework can be presented here. For the missing parts, we refer to the above-mentioned paper and the documentation in the distribution.

The bachelor computer science curriculum at the Radboud University Nijmegen, The Netherlands, provides a compulsory introductory course "Functional Programming", which is also taken by artificial intelligence students. This course was previously called "Abstraction and Composition in Programming", which has been taught for the past four years, and was taken by computer science students exclusively. Both groups of students have had training in imperative programming (C) and object orientation (Java), and neither group has been exposed before to functional programming.

Learning a new programming paradigm always covers two main ingredients: students need to learn a new *programming language*, which is Clean (Brus *et al.* 1987; Plasmeijer & van Eekelen 2001) in our case, and new ways of *problem solving*. In the course we cover "classic" topics such as recursive algebraic types; (higher order) functions; overloading; recursion and induction; and correctness proofs. Clean-specific

topics are uniqueness types for programming with effects, applying strictness annotations, term graph rewriting, and dynamics. The supporting 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 of such 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 engages all students. A statement by Johan Cruijff, a well-known Dutch football player, 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, May 2, 1987)

The most intriguing aspect about this statement is that Cruijff actually says that every football player computes a *pure function*: given the necessary 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 pure function:

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

Note that we are going to use a different function (Section 2) because football players also move around according to some strategy and need to listen to the referee.

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 (see Figure 1 for an impression), because seeing is believing. It should have a competitive element to stimulate students to create better solutions than their fellow students. Last but not the least, it had to promote functional programming. This has resulted in Soccer-Fun. We emphasize that Soccer-Fun and its exercises are not meant to replace all of the well-known "classic" exercises, but serve as a supplement to the repertoire of functional programming exercises. Although the domain of Soccer-Fun is football, most of the elements that need to be created by the students are elementary (distribution of players on the field, being in position, strategies), and occur in other team sports such as hockey, basketball, handball, and rugby. Hence, Soccer-Fun is also attractive to students who have no interest in football.

Soccer-Fun is implemented in Clean using its GUI library Object I/O (Achten & Plasmeijer 1998; Achten & Wierich 2000). Where necessary, we explain the syntax of Clean. Haskell (Hudak 1992) programmers may wish to consult a short list of differences (Achten 2007). The graphics are deliberately simple. Soccer-Fun

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

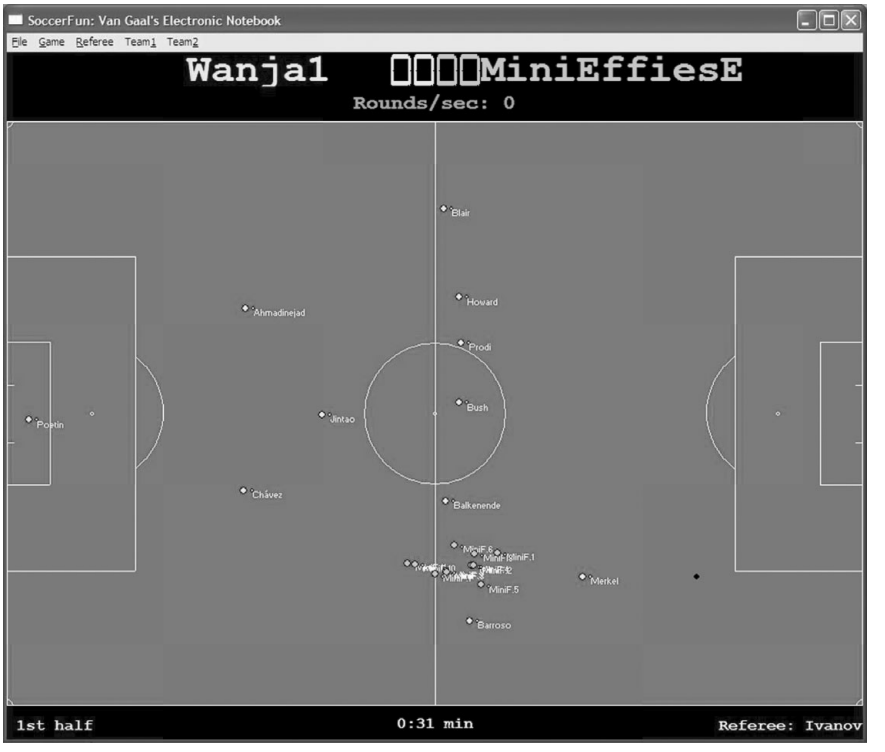


Fig. 1. The Soccer-Fun framework in action.

can be implemented in any functional programming language that supports GUI programming.

The Soccer-Fun project is open to participation for anybody who likes to contribute, either by means of improving the implementation, port it to another functional language, develop exercises, documentation, or example teams. The current implementation and documentation can be found at

http://www.cs.ru.nl/P.Achten/SoccerFun/SoccerFun.html.

The remainder of this paper is organized as follows. In Section 2 we introduce the Soccer-Fun API, which is an instructive example to show students how to give structure to a problem domain with a programming language. We have experimented with a variety of exercises, and briefly discuss them in Section 3. We highlight the exercises that are concerned with the main task of creating brain functions in Section 4. In Section 5 we report on our experience. Related work is described in Section 6. We come to conclusions in Section 7.

## 2 The Soccer-Fun API

In this section we introduce the Soccer-Fun API, which utilizes "classic" type features of the host language: algebraic data types, record types, function types, and existentially quantified types. Note that uniqueness types, a distinguishing feature

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

of Clean, play no role here. The framework factors out programming side effects. To emphasize the functional nature of Cruijff's statement, the brain can and should be a pure function. Analogously, to name just two other functional languages, a Haskell API should not require the `IO` monad, and an ML (Ullman 1998) approach should not require `ref` values. This design choice does not rule out an implementation of Soccer-Fun in a language that supports side effects. However, the API should be free of side-effects as it is not needed.

### 2.1 The brain

We start our discussion by first introducing the function that plays the pivotal role in this paper: our version of the football player's brain. We stick to the idea of Cruijff, hence the brain must have a function type:

```
:: FootballerAI memory :== (BrainInput, memory) -> (BrainOutput, memory)
```

`::` $t$ `:==` $u$ introduces the synonym type $t$ with definition $u$. Here, $t$ is the type constructor `FootballerAI` that is parameterized with the type variable `memory`.

The situation described by Cruijff only applies when a footballer wants to play the ball. When playing a match, the brain needs more information. This information is passed to the brain as a record of type `BrainInput` and contains the actions of the referee, the whereabouts of the football, the status of the other players, and his own status. Based on this information and his memory value of type `memory`, the brain computes the output of type `BrainOutput`, which is a single action of type `FootballerAction` that the player wishes to perform (explained in Section 2.5), and an updated memory value.

```
:: BrainInput    = { referee   :: [RefereeAction]
                   , football  :: FootballState
                   , others    :: [Footballer]
                   , me        :: Footballer
                   }
:: BrainOutput :== FootballerAction
```

A record of type $t$ with fields $f_i$ of type $t_i$ is introduced by `::` $t$ `= {`$f_1$ `::` $t_1$, ..., $f_n$ `::` $t_n$`}` ($n > 0$). We have observed that it is more convenient to use a record instead of separate function arguments because this allows us to refer to these arguments by their field names.

### 2.2 Metrics

Before we discuss the brain input in detail, we first settle on the metrics that we are going to use. In Soccer-Fun, all distances are given in metres, for which we use the built-in floating points of type `Real`. The dimensions of a football field are provided as a structured value:

```
:: FootballField    =   { fwidth :: FieldWidth, flength :: FieldLength }
:: FieldWidth       :== Metre
:: FieldLength      :== Metre
:: Metre            :== Real
```

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

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.[1] We distinguish between positions *on* and *above* the football field (`Position(3D)`).

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

Players need to know their goal's side on the field. This is defined by their *home*:

```
:: Home            =   West | East
instance other Home
```

`::` $t$ `=` $alt_1$ `|` ... `|` $alt_n$ introduces an algebraic data type $t$ with $n > 0$ data constructors, each having an arbitrary number of (recursive) arguments. `West` is the left-hand side of the football field, `East` is the right-hand side. The overloaded function `other` flips the value to the "other" value in a two-value domain.

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

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

Players and ball move at a certain speed. 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 meters per second. The speed above the surface is defined by a velocity along the $z$-axis.

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

Players stick to the ground and always have a `Speed`. Only the football has a `Speed3D` value.

Soccer-Fun provides the usual overloaded operators for adding, subtracting, comparing, converting, and printing purposes. Another particularly useful function is

```
angleWithObject    :: Position Position -> Angle
```

which, when applied to positions $p_1$ and $p_2$, returns the angle between two lines that intersect at $p_1$, and where the first line has angle $0\pi$, and the second line also goes through $p_2$. The result angle can be used for rotating toward $p_2$, or for playing the football to $p_2$.

[1] Some students do not get used to this convention. We plan to convert to standard mathematical convention in a next version.

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

### 2.3 The whereabouts of the football

The `football` field of a footballer's brain input tells him where the football is. The football is *either* freely available on or above the football field *or* it is *possessed* by a football player. In that situation a player manipulates the ball in such a way that it seems as if the ball is glued to his legs and feet. 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. When gained by a footballer, the ball inherits his current speed and position. Because the football can also be in the air, its position is given by a `Position3D` value, and its speed by a `Speed3D` value:

```
:: FootballState    = Free Football | GainedBy FootballerID
:: Football         = { ballPos :: Position3D, ballSpeed :: Speed3D }
```

Given the current brain input, the student can retrieve the current `Football` with

```
getBall             :: BrainInput -> Football
```

### 2.4 The football players

The brain input fields `others` and `me` inform the footballer about all players, including himself. Football players are defined with a rather extensive set of attributes. We discuss only the most interesting ones and refer to Achten (2008) for further details.

```
:: Footballer       = E. memory:
                      { playerID :: FootballerID, name     :: String
                      , length   :: Length,       nose     :: Angle
                      , pos      :: Position,     speed    :: Speed
                      , stamina  :: Stamina,      health   :: Health
                      , skills   :: MajorSkills,  effect   :: Maybe FootballerEffect
                      , brain    :: Brain (FootballerAI memory) memory      }
:: FootballerID     = { clubName :: ClubName,     playerNr :: PlayersNumber }
:: Brain ai memory  = { ai       :: ai,           m        :: memory        }
```

The keyword `E.` in the `Footballer` type is an *existential quantifier* to hide the actual memory value of a player. In real life, people cannot read each other's minds, and Soccer-Fun football players are no exception to this rule. Data hiding by means of existential encapsulation resembles making class data `private` in Java, and poses no problems to our students.

A footballer is uniquely identified with a `FootballerID`, which defines the club for which he is playing (`:: ClubName :== String`), and his player's number (`:: PlayersNumber :== Int`).

A football player can select three major skills:

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

The yield of actions governed by a major skill is better than average. In this way, the student can create a variety of football players easily. Soccer-Fun uses and provides

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

skill-dependent functions to determine if actions are successful. In this paper we use one such function to determine if the ball is within the kicking reach of the football player:

```
maxKickReach        :: Footballer -> Metre
```

The final, and the most important attribute, of a football player is his `brain`, which is a pair of his "intelligence", the (`FootballerAI memory`) function, and a matching `memory` value.

### 2.5 Actions and effects

The brain function computes an appropriate `FootballerAction` value, and perhaps updates his memory. It should be noted that an action expresses only the *intention* to perform that action. Even though his brain may want him to run at 20 m/s, his body will not be capable of doing this. Soccer-Fun takes every action into account and computes a realistic `FootballerEffect`, which is an algebraic data type that provides a data constructor for each `FootballerAction` in past tense with basically the same arguments. One extra effect is included: (`OnTheGround` $n$), which tells the footballer that he is lying on the ground for the next $n$ time frames. Soccer-Fun provides the following actions:

```
:: FootballerAction =   Move     Speed Angle | Feint    FeintDirection
                    |   GainBall             | CatchBall
                    |   KickBall Speed3D     | HeadBall Speed3D
                    |   Schwalbe             | PlayTheater
                    |   Tackle   FootballerID Velocity
:: FeintDirection   =   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 attempt to gain possession of the ball with `GainBall`. Only within his penalty area, the goalkeeper can legally use `CatchBall`. The ball remains with the player until he either plays it or it is gained by another player. Note that a player is slowed down when in possession of the ball.

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`[2] 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.

[2] *Schwalbe* is a german word. It means the *swallow* bird. During flight, swallows sometimes let themselves suddenly drop, and continue flying. This term is adopted in football because it resembles the above-mentioned behavior of unfair players.

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

Performing a (`Tackle` *id v*) is an attempt to bring the player with `FootballerID` *id* to an abrupt halt. Depending on the velocity *v* with which this action is intended to be performed, this may cause damage to that player'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.

### 2.6 *The referee actions*

The referee inspects all effects of the football players and takes appropriate actions. This is a list of referee actions. For instance, when a player violates the rules, this is signalled as a referee action, but also the reprimand (a warning, yellow or red card) and penalty (free kick, penalty kick, corner kick) are returned. The referee can also *repel* players from the ball in case they do not obey the rules of the game. For instance, when one team has been granted a goal kick, but the other team is still playing the ball.

### 2.7 *Team building*

To add a team to Soccer-Fun, a function of type `::` `Home FootballField -> Team`, where `Team :==` `[Footballer]` needs to be created by the student. The arguments tell the team at which side of the football field they start playing and its dimensions. These are necessary for the line up of the players. In a team, footballers play for the same club. Only the keeper has number 1, and no two fielders of the same club should have the same number.

### 2.8 *The* Soccer-Fun *framework*

When two teams are selected, the student uses Soccer-Fun to start a match. What basically happens is that at a regular time interval, the brain function of the referee and all players are evaluated with the proper arguments. This yields a list of referee actions and one intended action for each player. The framework computes an effect for each player. Pseudo-random numbers are used to resolve conflicts, such as when several players attempt to gain the ball at the same time. When the effects have been computed, they are applied to all players and football, which results in a new state of the match. This new state is rendered, and evaluation continues.

The computation of effects in Soccer-Fun takes care that neither the players nor the football can exit the football field. Brains are pure functions, as is the computation of the pseudo-random numbers, which takes a time-stamp to generate an initial random seed. Hence, a match is completely determined by the two teams at start of the match and the time stamp. To increase realism, pseudo-random numbers can also be used to add deviations in the computations of effects. In that case, the player skills control the range of deviation. This can be switched on and off in the Soccer-Fun GUI.

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

## 3 Exercises

In Soccer-Fun students develop a team that is equipped with the best brain functions. As the Soccer-Fun API uses many language features, they cannot do this right away. Here are suggestions for exercises that can be given before developing brain functions.

- Soccer-Fun provides a collection of basic geometric data types and functions (Section 2.2). One can devise exercises to develop data types to represent positions, speeds, velocities, angles, vectors, and so on. This trains the student in designing types with *algebraic types* and *record types*. Clearly, such values need to be compared, converted, and computed with. These are typical examples of *overloading*.
- In the start of the course students write recursive functions over lists to learn functional style recursion and lists. However, once they have learned this, we encourage students to use *list comprehensions* and the *standard list functions*, such as `map`, `filter`, `fold` instead. In Soccer-Fun this comes quite naturally with queries on the footballers. Examples are as follows: who are your team members, who is nearest, who is in offside position, who is in free position.
- Soccer-Fun handles all physics. One can devise exercises to compute the trajectory of the football and enhance it with air friction, influence of wind, rotation, and so on. The ball may bounce against players or goal poles, which can be included as well. At the current stage, Soccer-Fun implements a simplistic physical model.
- Soccer-Fun handles all rendering. This rendering uses basic elements that are present within any graphical toolkit: lines, curves, rectangles, and text. One can devise exercises to render Soccer-Fun elements. The rendering primitives are imperative. It is more interesting to teach a functional style of rendering. In particular, *higher order functions* can be illustrated well in this domain using combinator functions, continuations, state monad, and so on.

## 4 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 our experience, it is helpful to have students first make an informal description of the brain function that needs to be created, and only then to write code. Such a *brain sketch* is expressed in structured natural language, as a collection of (nested) guarded equations of the form **if** *cond* ⇒ *action*. The brain sketch must have a name and mention all the necessary arguments, which is at least the brain input. Because simple brains do not require a memory, this argument is optional. The general format is

| *brain input* |
|---|
| **if** *cond* ⇒ *action* |
| ⋮ |
| **otherwise** ⇒ *action* |

| *brain (input, memory)* |
|---|
| **if** *cond* ⇒ *(action, memory)* |
| ⋮ |
| **otherwise** ⇒ *(action, memory)* |

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

The Soccer-Fun code closely follows the structure of the brain sketch, but evidently needs to fill in all the (computational) details.

### 4.1 Basic exercises

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

#### 4.1.1 A few trivial brains

To stimulate functional problem solving, we start with a few very basic brains, and develop more complex brains from simpler ones as we move along. These simple functions do not need a memory, and to emphasize this, we introduce another type synonym:

```
:: FootballerAI' :== BrainInput -> BrainOutput
```

Brain functions of this type are identified with suffix '. For succinctness, we adopt the convention that brain input fields that are used are pattern matched at the function argument:

```
returnAI'          :: FootballerAction -> FootballerAI'
returnAI' action   = const action

move'              :: Speed Angle -> FootballerAI'
move' speed angle = returnAI' (Move speed angle)

halt'              :: FootballerAI'
halt'              = move' zero zero

rotate'            :: Angle -> FootballerAI'
rotate' angle      = move' zero angle

ahead'             :: Velocity -> FootballerAI'
ahead' v           = \input=:{me} -> move' {direction=me.nose,velocity=v} zero input
```

(`returnAI' action`) is the brain that always performs `action`, (`move' s a`) is just the (`Move s a`) action, `halt'` makes the footballer stand still, (`rotate' a`) makes the footballer rotate his nose over angle `a`, and (`ahead' v`) makes the footballer follow his nose with velocity `v`.

It is useful to have similar basic brain functions such as `move'` for each of the footballer actions (`feint'`, `gainBall'`, and so on). We assume that they are available with their name and arguments derived from the data constructor analogously to `move'`. Finally, players that suffer from amnesia can upgrade their brains to ordinary brain functions with:

```
amnesia            :: FootballerAI' -> FootballerAI m
amnesia f          = \(input, m) -> (f input, m)
```

In the remainder we assume that we also have the brain functions without amnesia. They have the same name, with the ' suffix removed (`returnAI`, `move`, and so on).

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

### 4.1.2 Run a (number of) lap(s)

Let the footballer start at a corner of the football field. Make him run laps in clockwise direction within the boundaries of the field. The dimension of the field is relevant:

*laps' field input*

| | | |
|---|---|---|
| **if** I am close to a corner | | |
| **if** nose in right direction | ⇒ *ahead'* $5\frac{m}{s}$ | *input* |
| **otherwise** | ⇒ *rotate'* to next corner | *input* |
| **otherwise** | ⇒ *ahead'* $5\frac{m}{s}$ | *input* |

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 just runs ahead. The corner data type and its access functions are easy:

```
:: Corner          =  NorthWest | NorthEast | SouthEast | SouthWest
next               :: Corner -> Corner
dir                :: Corner -> Angle
cornerOf           :: FootballField Position -> Maybe Corner
```

The function `next` computes the next corner the player has to go to, `dir` yields the direction the player has to face to reach that point, and `cornerOf` returns (`Just corner`) in case the player is close to `corner`, and `Nothing` otherwise. We now implement the brain function:

```
laps' :: FootballField -> FootballerAI'
laps' field input=:{me}
| close_to_corner
    | nose_ok       = ahead'  v      input
    | otherwise     = rotate' angle input
| otherwise         = ahead'  v      input
where
  close_to_corner   = isJust   (cornerOf field me.pos)
  corner            = fromJust (cornerOf field me.pos)
  angle             = dir (next corner) - me.nose
  nose_ok           = abs angle <= 0.01
  v                 = 5.0
```

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

### 4.1.3 Run to a location

A useful functionality is to run to a certain location within a range (for instance, his default position, or to gain a freely available ball, or to try to gain the ball from an opponent).

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

| *fix' point precision input* |
|---|

| | |
|---|---|
| **if** I am close enough to *point* | ⇒ *halt' input* |
| **otherwise** | ⇒ *move'* and rotate to *point input* |

To implement this brain, the student learns to use `angleWithObject` (Section 2.2).

```
fix'              :: Position Metre -> FootballerAI'
fix' point eps input=:{me}
| distance <= eps = halt' input
| otherwise       = move' {direction=angle,velocity=v} (angle - me.nose) input
where
  distance        = dist            me.pos point
  angle           = angleWithObject me.pos point
  v               = max 6.0 distance
```

In order to reach a given `point`, the brain tells the player to rotate and run toward that point. It is satisfied as soon as the player is within the given distance of that point.

*Variations*: Same as above, but place the position in memory; instead of a single position in memory, use a(n in)finite list of positions that need to be visited in sequence; use the latter version to implement the `laps'` from Section 4.1.2.

### *4.1.4 Passing the ball*

Players need to pass the ball to other players or attempt to score a goal. Hence, it makes sense to implement a brain that passes the ball correctly to an interesting position. We provide the brain function with the desired new position of the ball. The football player can only kick the ball if it is within kicking range, otherwise he can do nothing.

| *kick' target input* |
|---|

| | | |
|---|---|---|
| **if** I can kick the ball | ⇒ *kickBall'* to *target* | *input* |
| **otherwise** | ⇒ *halt'* | *input* |

This brain needs to know where the football is, and for that purpose it can use the `getBall` function (Section 2.3). To know whether the ball is within kicking reach, the brain function requires the `maxKickReach` function (Section 2.4). The interesting question is how hard the footballer should kick the ball in order to make sure that the ball arrives at the desired location. 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'
kick' point input=:{me}
| dist me.pos ball.ballPos <= maxKickReach me
                  = kickBall' {vxy = {direction=angle,velocity=v},vz = 0.0} input
| otherwise       = halt' input
```

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

```
where
    ball        = getBall input
    angle       = angleWithObject me.pos point
    v           = 5.0*dist me.pos point
```

*Variations*: Instead of playing the ball over the field, play it through the air; create a kick function that accepts a `Position3D` location.

### 4.2 Strategy exercises

In a team, players need to act according to a strategy in order to play a good match.

#### 4.2.1 Fix point variations

With the `fix‘` brain (Section 4.1.3) the student can readily implement several frequently occurring footballer behaviors. The first concerns the line-up of players, which is an important aspect of any team sport. Players should place themselves on the field in such a way that their team can control a significant part. One way to achieve this is by assigning a region of the football field that each player controls. 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‘` $p$). The second behavior moves a player to the ball:

```
to_ball‘ :: FootballerAI‘
to_ball‘ input = fix‘ (getBall input).ballPos.pxy zero input
```

In a team you need to agree 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 goes for it.

```
gain‘ :: FootballerAI‘
gain‘ input=:{others,me}
| closest       = to_ball‘ input
| otherwise     = halt‘    input
where
  {ballPos}     = getBall input
  closest       = dist me.pos ballPos <=
                  minList [ dist pos ballPos \\ {pos} <- others | sameClub me p ]
```

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*: Make a dynamic line-up, depending on whether your team is in possession of the ball. If it is, then the line-up should advance toward the goal of the opponent, otherwise the line-up should withdraw toward the home goal.

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

### 4.2.2 Intermezzo: listen to the referee

During a match, a footballer needs to know where the goals are. At the start of the match, he is told where his `home` side is (Section 2.7). He needs to listen to the referee to know when the second half starts. The referee action `EndHalf` (tested with the predicate `isEndHalf`) tells him this. We can educate footballers with a wrapper memory and wrapper brain function:

```
:: HomeM m              = { home :: Home, mem :: m }

educate                 :: (Home -> FootballerAI m) -> FootballerAI (HomeM m)
educate home_ai (input=:{referee}, memory=:{home})
                        = (action, {memory & home = new_home, mem = new_memory})
where
  new_home              = if (any isEndHalf referee) other id home
  (action, new_memory) = home_ai new_home (input, memory.mem)
```

It is easy to write a similar function for footballers with amnesia. In the exercises below, we assume that we deal with educated footballers.

### 4.2.3 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 us work out a brain that decides to pass the ball to a 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 too far away from the goal, he dribbles toward the goal, otherwise he tries to kick the ball in the goal:

*play' field home input*

| | |
|---|---|
| **if** I am in best position | |
| **if** I am close to goal | ⇒ *kick'* in center of goal *input* |
| **otherwise** | ⇒ *fix'* center of goal *input* |
| **otherwise** | ⇒ *kick'* to player in best position *input* |

```
play'               :: FootballField Home -> FootballerAI'
play' field=:{fwidth,flength} home input=:{others,me}
| in_best_position
  | near_goal        = kick' goal      input
  | otherwise        = fix'  goal zero input
| otherwise          = kick' best      input
where
  in_best_position = isEmpty better
  better           = [p \\ p <- others | sameClub me p && dist p.pos goal < d_goal]
  best             = (hd better).pos
  goal             = {py = fwidth / 2.0, px = if (home == West) flength zero}
  d_goal           = dist my.pos goal
  near_goal        = d_goal <= 20.0
```

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

*Variations*: Take intermediate opponents or the *offside rule* into account. 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.

#### 4.2.4 The referee as a coach

During a match, the referee constantly inspects all executed effects, and creates verdicts. Hence, a referee *monitors* the performance of brain functions. For instance, in case of the running laps exercise (Section 4.1.2), a referee can check whether the footballer is constantly close to an edge of the football field and running in the correct direction. If he is not, he issues a message. Currently, Soccer-Fun has four "coaching" referees. The first checks whether a footballer is running in a slalom fashion around a number of stationary opponents to the ball at the other side of the field. The second checks whether footballers pass the ball correctly in sequence to each other. The third checks whether one player passes the ball to another team player such that moving opponents can not intercept the ball. The fourth checks whether a goalkeeper is correctly defending the center of his goal. Students work on exercises until the referee is satisfied. They need less supervision, and examination time is reduced because the referee gives the verdict.

### 4.3 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 using a functional style in which brain functions are glued together to create more complex brain functions. With the brain functions `gain`‘ and `play`‘ basic footballers are created. This stimulates students to continue improving their teams and challenges them to invent a better set of rules for their players' brains. Brain sketches let the students think about the brain function without getting swamped early in programming details.

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. As one example consider using lists: in the memory, store a list of actions that need to be performed until the list is empty, and compute a new list when done. The list can be created in a typical function compositional style: generate a list of possible decisions, map a weight function to compute success rates, select the decision with maximum success, and map the decision to footballer actions. Alternatively, one can use *game trees* (Bird & Wadler 1988), which is also an appealing technique for artificial intelligence students.

## 5 Experience

We have used Soccer-Fun in the compulsory second-year bachelor functional programming course for the past four years. Soccer-Fun succeeds in keeping

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

significantly more students motivated. Almost all students like the assignment. At the end of the course the students have created a complete team. We wrap up the course with a tournament together with all students. It always has the atmosphere of a real football tournament. The tournament yields a champion, who is awarded. Because the *code* of the champion team is not necessarily a champion as well, we also award a prize for the "most functional style" code.

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, in which they receive a crash course on functional programming based on Soccer-Fun. Most pupils have experience in mostly imperative languages. Functional programming is entirely new to them. We use the brain sketches to think about the brain's function and compare this with the Clean implementation. Pupils respond enthusiastically to Soccer-Fun. They appreciate the example implementations, and usually come up with improvements quickly. Because we have noticed that encoding the improvement gives them a hard time, we have provided all the functions that have been described in Section 4, and show them with small prescribed steps how they can enhance a very simple footballer that implements `to_ball`‘ (Section 4.2.1). We wish to experiment with constructing another API on top of Soccer-Fun to further close the gap with the brain sketches.

## 6 Related Work

Mathew Nelson's Robocode (http://robocode.sourceforge.net/) is an exemplary framework that is targeted at teaching object orientation and Java in particular, and with slightly less peaceful intentions than Soccer-Fun, in which you program a military tank that drives around on a square area, together with other military tanks. Each tank executes an algorithm, aiming to eliminate other tanks by firing grenades at each other, and hopefully survive longest and become champion. Robocode has been 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 promotional activities as described in Section 5. Such a framework can be an effective teaching tool, as well as stimulating and fun.

Another object-oriented framework is Alice (Conway 2000). With this framework novice programmers can create intricate 3D environments that are populated with objects. These objects react to their environment in the usual object-oriented sense by programming event handlers. One strong point of Alice is that users can create behaviors in a compositional way that strongly resembles a combinator style of programming. Another strong point of Alice is that the code is executed while developing to show the user the effect of her code. If we want to make Soccer-Fun more comprehensible for a secondary education audience, we should take the lessons learned by the Alice team into account.

In the NetLogo (Wilensky 1999) programmable modeling environment, complex dynamic systems are described with scripts that are applied to turtles. The environment allows users to interactively and incrementally experiment with dynamic

systems. The base language and concepts of NetLogo are imperative. An environment such as Soccer-Fun can certainly be modeled within NetLogo: the football players are turtles that follow their rules as dictated by their brain, the football field is modeled as a "non-wrapping" plane (otherwise football players can leave the field from one edge and appear at the opposite side), the referee is modeled as an observer, and the football can be a turtle as well. We distinguish Soccer-Fun from NetLogo by our focus on modeling the brain as a pure function, and using functional concepts only to implement brain functions.

The RoboCup project (http://www.robocup.org/) is related to Soccer-Fun, but is targeted at entirely different technology. It is the ultimate aim of this project that hardware robots compete with humans in a football match. In this project a simulator software, the RoboCup Soccer Simulator, is available in which you can create your own robots.

UNIVERSE (Felleisen *et al.* 2009) is a library that extends the DrScheme (Findler *et al.*2002) programming environment with functions to handle *distributed*, window-based I/O applications in a functional style. UNIVERSE has been designed to teach functional programming concepts to middle school, high school, and university students. This has been achieved by imposing a strict and clear separation of concerns when programming interactive applications: the students only write functions within the functional domain (describing numbers, text, and images), as well as callback functions that handle events (keyboard, mouse, and messages). The UNIVERSE library takes care of all low-level, boilerplate issues. This is very similar to Soccer-Fun, where the student only writes a brain function, and the framework takes care of the rest. UNIVERSE is obviously more general than Soccer-Fun. An interesting (range of) exercise(s) for UNIVERSE is to implement a Soccer-Fun-like application and the suggested exercises in this paper.

Yampa (Hudak *et al.* 2003) is a functional reactive programming language in which robots can be created. Functional reactive programming promotes high-level construction of time-based behaviors, called signals. These signals can be continuous (e.g., the motion of the robot) as well as discrete (e.g., collision detection). Dance (Huang & Hudak 2003) 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 can certainly imagine that this is possible. We speculate that in these approaches a specification of a footballer brain is more geared toward controlling physical movement. We have not explored this avenue.

## 7 Conclusions

The main goal of the Soccer-Fun project is to motivate students to functional programming. Soccer-Fun is stimulating because it covers a well-known problem domain, and offers a graphical user interface (GUI) that despite its simplicity, effectively shows what the brains are thinking. Students get visual feedback about the performance of their created brains. The competitive element makes students *want* to create a better team and therefore they are motivated and required to

https://doi.org/10.1017/S0956796810000055 Published online by Cambridge University Press

express their ideas in a functional language. Talented students can show how well they understand functional programming, and less skilled students can still achieve acceptable results, and even become tournament champion. A special feature of Soccer-Fun is the coaching referee that monitors the performance of student exercises. Although 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. Most importantly, the brain is naturally modeled by a pure function because it computes one action from input data. The type system restricts the set of admissible brain functions that a student might come up with. Function abstraction and composition is stimulated by creating bigger brains from smaller ones. We think Soccer-Fun is a welcome addition to the exercise repertoire for any introductory course in functional programming.

## Acknowledgments

First of all, the author would like to thank Rinus Plasmeijer, who is co-teacher of the functional programming courses, for his enthusiasm and inspiration. Wanja Krah has developed parts of Soccer-Fun during a master's thesis project of the computer science study of the Hogeschool Brabant, The Netherlands. The author is grateful for the comments and ideas of students and pupils who have used Soccer-Fun. The author thanks the referees and participants at FDPE'08 for their feedback as well as the referees of this manuscript. Thomas van Noort read the proofs of the final version of this paper.

## References

Achten, P. (2007) *Clean for Haskell98 programmers – A quick reference guide*. Available at: `http://www.st.cs.ru.nl/papers/2007/achp2007-CleanHaskellQuickGuide.pdf`

Achten, P. (2008) Teaching functional programming with Soccer-Fun. In *Proceedings of the 1st Workshop on Functional and Declarative Programming in Education, FDPE'08*. Huch, F. & Parkin, A. (eds). Victoria, BC, Canada: ACM Press, pp. 61–72.

Achten, P. & Plasmeijer, R. (1998) Interactive functional objects in Clean. *Selected papers of the 9th International Symposium on the Implementation of Functional Languages, IFL'97*. Clack, C., Hammond, K. & Davie, T. (eds), LNCS, vol. 1467. Springer, pp. 304–321.

Achten, P. & Wierich, M. (2000) *A Tutorial to the Clean Object I/O Library (version 1.2)*. Technical report CSI-R0003. Nijmegen, The Netherlands: Institute for Computing and Information Sciences, Radboud University, p. 294.

Bird, R. & Wadler, P. (1988) *Introduction to Functional Programming*. Prentice Hall.

Brus, T., van Eekelen, M., van Leer, M. & Plasmeijer, R. (1987) Clean: A language for functional graph rewriting. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture, FPCA'87*. Springer, pp. 364–384.

Conway, M., *et al.* (2000) Alice: Lessons learned from building a 3D system for novices. In *CHI'00: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Turner, T. & Szwillus, G. (eds), New York, NY: ACM, pp. 486–493.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2009) A Functional I/O system or, Fun for freshman kids. In *Proceedings of the 2009 International Conference on Functional Programming, ICFP'09*. Edinburgh, Scotland: ACM, pp. 47–58.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S. Steckler, P. & Felleisen, M. (2002) DrScheme: A programming environment for Scheme, *J. funct. program.*, 12 (2): 159–182. doi:10.1017/S0956796801004208.

Huang, L. & Hudak, P. (2003) *Dance: A Declarative Language for the Control of Humanoid Robots*. Tech. rept. YALEU/DCS/RR-1253. Yale University.

Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2003) Arrows, robots, and functional reactive programming. *Proceedings of the 4th International Summer School on Advanced Functional Programming, AFP'03*. Jeuring, J. & Peyton Jones, S. (eds), LNCS, vol. 2638. Oxford, UK: Springer-Verlag, pp. 159–187.

Hudak, P., *et al.* (1992) Report on the programming language Haskell, a non-strict, purely functional language, *Sigplan Notices*, 27 (5): 1–164.

Plasmeijer, R. & van Eekelen, M. (2001) *Concurrent Clean Language Report (version 2.0)*. Available at: `http://clean.cs.ru.nl/`

Ullman, J. (1998) *Elements of ML Programming (ML97 edition)*. Prentice Hall Inc.

Wilensky, U. (1999) *Netlogo*. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available at: `http://ccl.northwestern.edu/netlogo/`