

Why Functional Programming Matters To Me

Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`p.achten@cs.ru.nl`

Abstract. Functional programming advocates a style of programming in which the programmer seeks to find a sufficiently small, yet powerful, set of abstractions that capture an entire class of problems, and use these abstractions to solve a concrete problem. I illustrate this by means of a case study in which I implement the game *Trax*TM. In this turn-based game two players attempt to create either a closed loop of a line of their own color, or make the line connect opposite ends of a tile set of some prescribed minimal dimensions. *Trax*TM is an attractive case because it has interesting computational problems, for which I use classical functional techniques, but also because it is a distributed multi-user application, for which I use the more recently developed *iTask* formalism.

1 Introduction

During my computer science studies at the University of Nijmegen my first exposure to functional programming was in 1987 – 1988 in a series of courses taught by Rinus Plasmeijer and Marko van Eekelen. Two appealing aspects of these courses were that we were asked to develop programs in David Turner’s programming language *Miranda*¹ [1, 2] and we were taught how functional programs can be compiled to efficient code using the intermediate language *Clean* (version 0.6 at that time). I learned to appreciate the beauty of functional programming (languages) and the semantic beauty of *term graph rewriting* [3, 4] of which *Clean* was and still is an implementation.

In this essay I explain why ever since my first exposure to it, functional programming matters to me. As a starting point, I refer to John Hughes’ seminal 1984 paper [5] in which he argues that functional programming matters because it offers *glue* with which to structure programs in an improved modular and reusable way, through the use of *higher order functions* and *lazy evaluation*. He sets the stage for how to go about solving a problem: “*It is also the goal for which functional programmers must strive - smaller and simpler and more general modules, glued together with the new glues we shall describe.*” ([5], pg.4).

When solving a computational problem, higher order functions improve the level of abstraction of a solution because instead of solving one particular problem a solution for an entire class of problems is developed. Lazy evaluation is a

¹ *Miranda* is a trademark of Research Software Ltd.

consequence of the fundamental *referential transparency* property of pure functional programming languages. No matter in what order a computing device goes about determining the final result of my program, it is guaranteed to be uniquely defined if it exists. Hence, during problem solving I can concentrate on whether my program *has* a solution and worry much less *how* this solution is going to be computed. Developing a functional program feels a lot like playing a game in which I know that the programming language and compiler stick to the same rules as myself.

I demonstrate the style of functional programming by showing how to implement the game *Trax*TM, which was brought to my attention by Rinus a couple of years ago. It is a 2-person turn-based tile game in which the players attempt to create either a closed loop of a line of their color (white or red) or make the line connect the far ends of the board that must have some minimal dimensions. Fig. 1 gives two examples of these winning states.

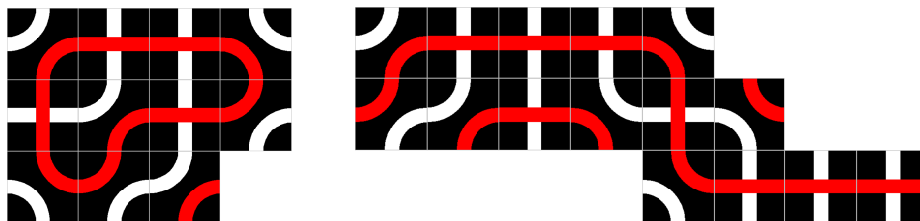


Fig. 1. Winning states for red: a closed loop (left) or a winning line (right).

Although *Trax*TM is a small and elegant game, it contains a number of sufficiently challenging problems, such as determining what tiles a player is allowed to place at which locations, determining if a configuration of tiles contains closed loops or winning lines, and how to prescribe and control the player actions. The specification of any game consists of two major parts: one part that introduces its *concepts and operations* – *what* the game is about – and one part that specifies what the *valid actions* are for each player – *how* to play the game –. The first part, described in Sect. 2, uses classical functional programming language features such as the above mentioned *higher order functions* and *lazy evaluation*, but also *algebraic and record data types* to model the domain of discourse accurately, *generic functions* [6, 7] to avoid boiler plate specifications, and *list comprehensions* to deal elegantly with collections, finite maps, streams, and operations on them. The second part, described in Sect. 3, uses the *iTask* formalism [8, 9]. Following John Hughes’ adage, the peculiarities of *Trax*TM are abstracted from first, after which it is easier to actually implement the game. It turns out that this abstraction has striking similarities with the Racket **big-bang** approach [10, 11]. I discuss this in more detail in Sect. 4. Finally, in Sect. 5, I hope to have explained to you why functional programming matters to me.

2 What Trax is About

This part of the *Trax*TM specification deals with the elements of the game. I proceed bottom-up and start with basic elements (Sect. 2.1), show how to match two tiles (Sect. 2.2), create only correct tile configurations (Sect. 2.3), define the concept of mandatory moves (Sect. 2.4), and finally compute the sets of closed loops and winning lines (Sect. 2.5).

2.1 Tiles, Lines, Coordinates

In this section all basic elements are defined that are needed in the *Trax*TM specification. This amounts to modeling the entities as well as operations on these entities by means of data types and access functions.

A *Trax*TM tile has two sides, each displaying a white line and a red line. On the one side the lines cross each other, and on the other side they evade each other. These sides are placed in six different configurations (Fig. 2). A tile



Fig. 2. The six possible tile configurations.

configuration is modeled by defining which edges are connected by the red line:

```
:: Tile = { end1 :: !Edge, end2 :: !Edge }
:: Edge = North | East | South | West
```

The names `vertical`, `horizontal`, and so on are each of type `Tile` and identify the tiles as depicted in Fig. 2. For instance:

```
vertical = { end1 = North, end2 = South }
```

When modeling entities with data types it is a good habit to think right away which basic operations (such as comparison, arithmetic, printing, parsing) are sensible because this unlocks useful general purpose functions (such as sorting, searching, printing, storage via (de)serialization). These functions exist as higher-order polymorphic functions and overloaded functions. The basic operation is typically an argument of these general purpose functions (explicit in case of higher-order functions and implicit in case of overloaded functions). Many basic operations can be expressed by *induction on the structure of types* for which purpose generic functions can be deployed. The concise declarations:

```
derive gEq      Edge
derive gLexOrd  Edge
```

In this case study it is useful to be able to enumerate all elements of a finite (and small) domain. This is a typical example of a custom generic function:

A detailed explanation of this function is out of the scope of this essay. In a nutshell, the last six lines define the induction on the structure of types, and the first two lines define the meaning for the basic types `Bool` and `Char`. Enumeration of all `Edge` values can be derived from this recipe:

For `Tile` values the generic scheme generates too many values, so the generic scheme must be overruled:

Other sensible basic operations on **Tile** and **Edge** values are comparison (`==` and `<`), printing (`toString`), and taking the opposite value (`~`). Their definitions are straightforward:

```
instance == Tile  
where    == {end1=a1,end2=a2} {end1=b1,end2=b2}  
        = (a1,a2) == (b1,b2) || (a2,a1) == (b1,b2)  
  
instance toString Tile  
where    toString tile = lookup1 tile [(horizontal,"horizontal")  
                                       ,(vertical, "vertical" )  
                                       ,(northwest, "northwest" )  
                                       ,(northeast, "northeast"  
                                       ),(southeast, "southeast" )  
                                       ,(southwest, "southwest" )  
                                       ]  
  
instance ~ Tile  
where    ~ tile      = lookup1 tile [(horizontal, vertical )  
                                       ,(vertical, horizontal )  
                                       ,(northwest, southeast )  
                                       ,(northeast, southwest )  
                                       ,(southwest, northeast )  
                                       ,(southeast, northwest )  
                                       ]  
  
gEq      {|Tile|} t1 t2 = t1 == t2
```

```

instance ==      Edge where == e1 e2 =  e1 == e2
instance <      Edge where <  e1 e2 = (e1 == e2) == LT
instance ~      Edge where ~  e      = case e of
                                North = South
                                South = North
                                West  = East
                                East  = West

```

On many occasions, it is necessary to find a value v in a list of key-value pairs (k, v) using a key k . The `lookup` and `lookup1` search functions capture this pattern (they are similar to the Haskell Prelude `lookup` function):

```

lookup          :: !k ![(k,v)] -> [v] | Eq k
lookup key table = [v | (k,v) <- table | k == key]

lookup1         :: !k ![(k,v)] -> v | Eq k
lookup1 key table = hd (lookup key table)

```

The *line color* entity is defined in the same spirit as *edges*:

```

:: LineColor = RedLine | WhiteLine
derive gFDomain LineColor
derive gEq    LineColor
instance ==    LineColor where == c1 c2    = c1 == c2
instance ~    LineColor where ~ RedLine    = WhiteLine
                                ~ WhiteLine = RedLine

```

A configuration of tiles such as those depicted in Fig. 1 is called a *trax*. A simple way to model a trax is by listing the tiles and their coordinates:

```

:: Trax      = { tiles :: ![(Coordinate,Tile)] }
:: Coordinate = { col :: !Int, row :: !Int }

derive gEq    Coordinate
derive gLexOrd Coordinate
derive gPrint  Coordinate
instance ==    Coordinate where == c1 c2    = c1 == c2
instance <    Coordinate where <  c1 c2    = (c1 == c2) == LT
instance toString Coordinate where toString c = printToString c
instance zero  Coordinate where zero        = {col=zero, row=zero}
instance zero  Trax      where zero        = { tiles = [] }
instance ==    Trax      where == t1 t2    = sortBy fst_smaller t1.tiles
                                ==
                                sortBy fst_smaller t2.tiles
gEq[|Trax|] t1 t2              = t1 == t2

col {col}      = col
row {row}      = row
fst_smaller (a,_) (b,_) = a < b

```

For navigation, we introduce functions to compute next coordinates:

```

north c = {c & row = c.row-1}
south c = {c & row = c.row+1}
west  c = {c & col = c.col-1}
east  c = {c & col = c.col+1}

```

```

go North = north
go South = south
go West  = west
go East  = east

```

Finally, of a Trax we need to know its current number of tiles (`nr_of_tiles`), the minimum and maximum values of the coordinates (`bounds`), the number of columns and rows that a trax occupies (`dimension`), and which tile, if any, can be found at a coordinate (`tile_at`). We wind up this section with their definitions:

```

nr_of_tiles :: !Trax -> Int
nr_of_tiles trax = length trax.tiles

bounds :: !Trax -> (!Int,!Int), (!Int,!Int))
bounds trax
| nr_of_tiles trax > 0 = ((minList cols,maxList cols), (minList rows,maxList rows))
| otherwise           = abort "bounds_applied_to_empty_set_of_tiles.\n"
where coords         = map fst trax.tiles
      cols           = map col coords
      rows           = map row coords

dimension :: !Trax -> (!Int,!Int)
dimension trax
| nr_of_tiles trax > 0 = (maxx - minx + 1, maxy - miny + 1)
| otherwise           = abort "dimension_applied_to_empty_set_of_tiles.\n"
where ((minx,maxx),(miny,maxy)) = bounds trax

tile_at :: !Trax !Coordinate -> Maybe Tile
tile_at trax c = case lookup c trax.tiles of
                  [tile : _] = Just tile
                  none_found = Nothing

```

2.2 Matching of Tiles

A new tile can only be added to the current trax at a specific location if the colors of the lines at its edges *match* with those of the currently present neighbouring tiles. For this purpose, I need to know the line colors of an empty location and the line colors of a tile. If some edge at a coordinate is not next to a tile, there is no color (`Nothing`), otherwise the color is associated with the edge (`Just color`).

```

:: LineColors := [(Edge,Maybe LineColor)]

```

The core function to determine line colors of empty locations and tiles is `tile-colors` which inspects a tile and returns all edge-color pairs. The derived function `color_at_tile` tells what color the tile has at some edge.

```

tilecolors :: !Tile -> LineColors
tilecolors tile = [(North,Just n),(East,Just e),(South,Just s),(West,Just w)]
where (n,e,s,w) = lookup1 tile [(horizontal,(WhiteLine,RedLine,WhiteLine,RedLine))
                                ,(vertical, (RedLine,WhiteLine,RedLine,WhiteLine))
                                ,(northwest, (RedLine,WhiteLine,WhiteLine,RedLine))
                                ,(northeast, (RedLine,RedLine,WhiteLine,WhiteLine))
                                ,(southwest, (WhiteLine,WhiteLine,RedLine,RedLine))
                                ,(southeast, (WhiteLine,RedLine,RedLine,WhiteLine))
                                ]

```

```

color_at_tile :: !Edge !Tile -> LineColor
color_at_tile edge tile = fromJust (lookup1 edge (tilecolors tile))

```

The line colors of an empty location are assembled by looking at the line color of the opposite edge of a neighbour tile at each of its edges. In this definition, `gFDomain{!|}` enumerates all Edge values, and `gMap{!*->*|}` is the functor that applies its first function argument to `Just` a value if there is one.

```

linecolors :: !Trax !Coordinate -> LineColors
linecolors trax c
  = [ (edge,gMap{!*->*|} (color_at_tile (~edge)) (tile_at trax (go edge c)))
    \ \ edge <- gFDomain{!|}
    ]

```

Two such line colors match if at each edge they either have the same color or if either one has no color:

```

linecolors_match :: !LineColors !LineColors -> Bool
linecolors_match a b
  = and [match c1 c2 \ \ (_,c1) <- sortBy fst_smaller a
        & (_,c2) <- sortBy fst_smaller b
        ]
where
  match (Just c1) (Just c2) = c1 == c2
  match _ _ = True

```

With the matching function, the collection of tiles that match particular line colors can be determined:

```

possible_tiles :: !LineColors -> [Tile]
possible_tiles colors
  = [tile \ \ tile <- gFDomain{!|} | linecolors_match colors (tilecolors tile)]

```

2.3 Correct Configurations by Construction

In a trax a new tile must be placed at one of the free edges. The collection of free coordinates is the union of all free neighbours of all tiles.

```

free_coordinates :: !Trax -> [Coordinate]
free_coordinates trax = removeDupSortedList
  (sort (flatten (map (free_neighbours trax)
                     (map fst trax.tiles))))

```

```

free_neighbours :: !Trax !Coordinate -> [Coordinate]
free_neighbours trax c = [ c' \\ c' <- neighbours c | isNothing (tile_at trax c') ]

neighbours :: !Coordinate -> [Coordinate]
neighbours c      = map (flip go c) gFDomain{! *}

```

Using the matching function and knowing valid free locations, it is now possible to safely add a tile to a trax:

```

add_tile :: !Coordinate !Tile !Trax -> Trax
add_tile c tile trax
| nr_of_tiles trax == 0 || isMember c (free_coordinates trax)
    &&
    linecolors_match (linecolors trax c) (tilecolors tile)
    = {trax & tiles = [(c,tile) : trax.tiles]}
| otherwise = trax

```

Starting with the `zero` instance of `Trax` and using only `add_tile` it is guaranteed that the trax is always in a valid configuration.

2.4 Mandatory Moves

After a player has added a tile to a trax all free locations that have no freedom as to what tile can be placed must be filled with their only tile candidate. These are the *mandatory moves*. It is sufficient to examine only the free neighbours of the placed tile. Those at which red or white occurs more than once belong to the collection of *mandatory tiles*.

```

mandatory_tiles :: !Trax !Coordinate -> [Coordinate]
mandatory_tiles trax c
= case tile_at trax c of
    Nothing = []
    -       = [free \\ free <- free_neighbours trax c
                | hasDup (filter isJust (map snd (linecolors trax free)))]

```

The mandatory moves need to be performed until there are no more mandatory tiles. Each move adds one tile to a trax. Hence, the structure of this algorithm is similar to the classic *fold* functions, except that each move may append extra list elements to be folded. Let's introduce *queued fold* functions that have an extra first function that determines which elements are to be appended:

```

qfoldl :: (a -> b -> [b]) (a -> b -> a) a ![b] -> a
qfoldl _ _ a [] = a
qfoldl f g a [b:bs] = let a' = g a b in qfoldl f g a' (bs ++ f a' b)

qfoldr :: (a -> b -> [b]) (b -> a -> a) a ![b] -> a
qfoldr _ _ a [] = a
qfoldr f g a [b:bs] = let a' = g b a in qfoldr f g a' (bs ++ f a' b)

```


The computation `mandatory_tiles` determines which free locations need to be filled, and is the first argument of the queued fold function. The function `move` updates the trax by adding the only possible tile at a given location.

```
mandatory_moves :: !Trax !Coordinate -> Trax
mandatory_moves trax c
| isNothing (tile_at trax c)
  = abort ("mandatory_moves:_no_tile_at_" <+++ c <+++ "\n")
| otherwise = qfoldl mandatory_tiles move trax (mandatory_tiles trax c)
where move trax filler
      = add_tile filler (hd (possible_tiles (linecolors trax filler))) trax
```

2.5 Closed Loops and Winning Lines

As illustrated in Fig. 1, a game of *Trax*TM ends as soon as a player constructs a *closed loop* or a *winning line*. In either case, it is necessary to extract a line of some given color from a trax. This results in a core function, `track`:

```
:: Line == [Coordinate]

track :: !Trax !LineColor !Edge !Coordinate -> Line
track trax color edge c
  = case tile_at trax c of
    Nothing      = []
    Just tile    = let edge' = other_edge (perspective color tile) edge
                  in [c : track trax color (~edge') (go edge' c)]

perspective :: !LineColor !Tile -> Tile
perspective colour tile = if (colour == RedLine) tile (~tile)

other_edge :: !Tile !Edge -> Edge
other_edge tile edge    = if (edge == tile.end1) tile.end2 tile.end1
```

As explained at the start of Sect. 2.1, tiles are defined from the point of view of the red line. The function `perspective` gives the proper representation of a tile from the given perspective. The line is constructed by ‘following’ tiles at some edge and determining at which next edge to proceed. A line either terminates at an empty location or it does not terminate, in which case it is a closed loop. In the latter case, the `track` algorithm computes an infinitely long line, but thanks to lazy evaluation this is not a problem. Despite their infinite nature, closed loops can be detected and made finite:

```
is_loop :: !Line -> Bool
is_loop [c:cs] = isMember c cs
is_loop empty  = False

cut_loop :: !Line -> Line
cut_loop [c:cs] = [c : takeWhile ((<>) c) cs]
```

Let’s first find all closed loops in a trax and do this separately for the two colors:

```

loops :: !Trax -> [(LineColor,Line)]
loops trax = [(RedLine, loop) \\ loop <- color_loops trax trax.tiles RedLine]
++
[(WhiteLine,loop) \\ loop <- color_loops trax trax.tiles WhiteLine]

```

The basic idea is to inspect each tile in a trax, use `track` to follow it, and collect the found line if it is a loop. Before proceeding with another tile, the tiles from the line can be removed from the trax because they cannot be part of another line of the same color:

```

color_loops :: !Trax ![(Coordinate,Tile)] !LineColor -> [Line]
color_loops trax [(c,tile):tiles] color
| is_loop line      = [line : loops]
| otherwise         = loops
where line          = track trax color (start_edge tile color) c
      tiles'        = removeMembersBy (\(c,t) c' -> c == c') tiles (cut_loop line)
      loops         = color_loops trax tiles' color
color_loops _ [] _ = []

```

```

start_edge :: !Tile !LineColor -> Edge
start_edge tile color = choose (lookup1 tile [(horizontal,(West, North))
, (vertical, (North,West ))
, (northwest, (North,South))
, (northeast, (North,South))
, (southeast, (South,North))
, (southwest, (South,North))
])
where choose          = if (color == RedLine) fst snd

```

Determining all winning lines in a trax is a matter of finding lines that connect either the far west with the far east or the far north with the far south. Obviously, an empty trax cannot contain a winning line:

```

winning_lines :: !Trax -> [(LineColor,Line)]
winning_lines trax
| nr_of_tiles trax == 0 = []
| otherwise             = winning_lines_at trax West ++ winning_lines_at trax North

```

The set of winning lines can be specified with a single list comprehension:

```

winning_lines_at :: !Trax !Edge -> [(LineColor,Line)]
winning_lines_at trax edge
| max - min + 1 < minimum_winning_line_length = []
| otherwise
= [ (color,line)
  \\ (c,tile) <- trax.tiles | min == coord c
  , color <- [color_at_tile edge tile]
  , line <- [track trax color edge c] | not (is_loop line)
  , end <- [last line] | max == coord end
  , Just tile <- [tile_at trax end] | color_at_tile (~edge) tile == color
]
where ((minx,maxx),(miny,maxy)) = bounds trax

```

```

(min,max,coord)      = lookup1 edge [ (West, (minx,maxx,col))
                                       , (East, (maxx,minx,col))
                                       , (North,(miny,maxy,row))
                                       , (South,(maxy,miny,row))
                                       ]

```

If the trax is not big enough, then a winning line is not found. The first tile of a winning line must start at the given edge of the trax. Following its track must not result in a closed loop. Moreover, its last tile must be at the far other end of the trax, and also its line color must be at the opposite edge.

3 How to Play Trax

This part of the specification of *Trax*TM is concerned with coordinating and visualizing the actions of the two players. The *iTask* formalism is used to model this behavior. This is done in three stages: first, the concept of *turns* is formalized (Sect. 3.1); second, the peculiarities of *Trax*TM are abstracted away to create a general specification of *n*-player turn-based games (Sect. 3.2); third, the abstraction is used to implement a two player *Trax*TM game (Sect. 3.3).

3.1 Turns

A Turn is specified as follows:

```

:: Turn = { bound :: !Int, current :: !Int }
derive class iTask Turn
instance ==    Turn where == t1 t2    = t1 == t2
instance toInt Turn where toInt turn = turn.current

new bound | bound > 0      = {bound = bound, current = 0}
next turn={current,bound} = {turn & current = (current + 1)      rem bound}
prev turn={current,bound} = {turn & current = (current - 1 + bound) rem bound}
match nr turn              = nr == turn.current

```

Thus, in a game with a bounded number of players, each player is identified with a unique number. The functions `next` and `prev` identify the next and previous player, and with the function `match` a player can check whether her number matches with the current turn.

3.2 *n*-Person Turn-Based Games

To abstract away from the details of a specific *n*-person turn-based game, a collection of characteristics functions that operate on a state `st` is introduced:

```

:: Game st = { game    :: String
               , state  :: [User]    -> st
               , over   :: (Turn,st) -> Bool
               , winner :: (Turn,st) -> Task Turn

```

```

    , move    :: (Turn,st) -> Task st
    , board   :: (Turn,st) -> [HtmlTag]
  }

```

The `game` field identifies the game. The `state` function makes the players known to the game state. The zero-based index position i in this list of users matches with a player's turn in the game, so `match i t` is `True` only if it is player i 's turn. When the game is `over`, the `winner` task declares which player has won. The `move` task prescribes a single move by the current player. Finally, the state is rendered by means of the `board` function.

Given these characteristic functions, it is possible to define the general structure of n -person turn-based games:

```

play_for_N :: !Int !(Game st) -> Task Turn | iTask st
play_for_N n game
=
    get_players n
  >>= \all -> withShared (new n,game.Game.state all)
    (\sharedGameSt -> anyTask [ user @: play_for_1 game nr sharedGameSt
                               \\\ user <- all & nr <- [0..]
                               ])

```

A game is a task that returns a winner defined by the turn. First, n players are selected, using the `get_players` task that is described below. During the game, players can see the current state of the game at all times. Only one of them can actually change the state of the game. Hence, their task descriptions *share* the state, which is captured with the `withShared` task combinator. The player actions are controlled with the `play_for_1` task. The `anyTask` combinator evaluates all tasks in the list until one terminates.

The `get_players` task describes the selection of the participants, which is modeled as a multiple choice of all currently registered users:

```

get_players :: !Int -> Task [User]
get_players n
= enterSharedMultipleChoice ("Select_" <+++ max 0 n <+++ "_players") [] users
  >>* [ WhenValid (\selection -> length selection == max 0 n) return
      , Always    ActionCancel (throw "Selection_of_players_cancelled.")
      ]

```

Whenever the correct number of players are chosen, the list can be returned by the current user. It is also always possible to simply terminate this task, in which case the entire game terminates.

Each player basically does two things: gaze at the rendered game and make a move during their turn:

```

play_for_1 :: !(Game st) !Int !(Shared (Turn,st)) -> Task Turn | iTask st
play_for_1 game my_turn sharedGameSt
= gaze ||- play
where gaze = viewSharedInformation ("Play_with_" <+++ my_turn)
      [ ViewWith game.board ] sharedGameSt
      play = watch sharedGameSt
      >>* [ WhenValid game.over game.winner

```

```

    , WhenValid (\(turn,_) -> match my_turn turn)
      (\(turn,st) ->
        game.move (turn,st)
        >>= \st -> set (next turn,st) sharedGameSt
        >>| play
      )
  ]

```

Gazing at the game is realized with the `viewSharedInformation` task which uses the rendering function to display the current value of the shared game state. Not only the player gazes at the game, the task also monitors the current value of the shared game state, using the `watch` task function which merely echoes the current value of the shared game state. Whenever it is detected that the game is over, the winner is declared and the game terminates. At a player's turn, she performs the move task, the next player is chosen, and the game state is updated.

3.3 The Specialization of Trax

With the generalized framework for n -person turn-based games available, the specification of *Trax*TM amounts to deciding upon a suitable game state and characteristic functions. The game state needs to know the current trax and the persons who are playing the game.

```

:: TraxSt = { trax :: !Trax, names :: ![User] }
derive class iTask TraxSt

initial_state :: ![User] -> TraxSt
initial_state users = { trax = zero, names = users }

play_trax :: Task Turn
play_trax = play_for_N 2 { game = "Trax"
                        , state = initial_state
                        , over = game_over
                        , winner = declare_winner
                        , move = make_a_move
                        , board = show_board
                        }

```

The game is over as soon as a closed loop or winning line exists:

```

game_over :: !(Turn,TraxSt) -> Bool
game_over (_,traxSt)
  = not (isEmpty (loops traxSt.trax ++ winning_lines traxSt.trax))

```

If the previous player managed to create a closed loop or winning line, then that player has won the game, otherwise the current player has won:

```

declare_winner :: !(Turn,TraxSt) -> Task Turn
declare_winner (turn,traxSt={trax,names})
  = viewInformation "The_winner_is:" [ViewWith (toString o (player names))] winner
where winners = loops trax ++ winning_lines trax
      last_player = prev turn

```

```
winner      = if (isMember (toLineColor last_player) (map fst winners))
              last_player turn
```

```
toLineColor turn = if (match 0 turn) RedLine WhiteLine
player [a,b] turn = if (match 0 turn) a b
```

Performing a move in the game amounts to letting the player choose a free coordinate, and then select a matching tile. This tile is added to the current trax, and the mandatory moves are performed.

```
make_a_move :: !(Turn,TraxSt) -> Task TraxSt
make_a_move (turn,traxSt={trax})
    =          chooseCoordinate trax
    >>= \new -> chooseTile    new trax
    >>= \tile -> return {traxSt & trax = mandatory_moves
                        (add_tile new tile trax) new}
```

At the start of the game, only the zero coordinate is free. In any other case, the player can select one of the available free coordinates:

```
chooseCoordinate :: !Trax -> Task Coordinate
chooseCoordinate trax
| nr_of_tiles trax == 0 = return zero
| otherwise             = enterChoice "Choose_coordinate:"
                        [ChooseWith ChooseFromComboBox toString]
                        (free_coordinates trax)
```

At the start of the game, any tile can be selected. If the free coordinate is known, the player must select a tile that matches the line colors at that specific location.

```
chooseTile :: !Coordinate !Trax -> Task Tile
chooseTile c trax
    = enterChoice "Choose_tile:"
      [ChooseWith ChooseFromRadioButtons (TileTag (16,16))]
      (if (nr_of_tiles trax == 0) gFDomain{!|*|}
          (possible_tiles (linecolors trax c))
      )
```

All that is left to do is to define a rendering of the trax. To this end, it is useful to specify a few helper definitions to create this *html*-based rendering:

```
TileTag :: !(Int,Int) !Tile -> HtmlTag
TileTag (w,h) tile = ImgTag [ SrcAttr    ("/" <+++ toString tile <+++ ".png")
                             , WidthAttr  (toString w)
                             , HeightAttr (toString h)
                             ]
tr          = TrTag []
td          = TdTag []
h3  x      = H3Tag [] [text x]
text x     = TdTag [AlignAttr "center"] [Text (toString x)]
```

In the *iTask* architecture, a task can place additional resources in a folder named *Static*. For each of the possible tiles, it contains a *.png* file (in fact, the ones of

Fig. 2). The `TileTag` function generates an image tag that displays this file with suitable dimensions. With these helper definitions, the `trax` is rendered as a *html* table. A cell displays either a tile, or the coordinate of a free location, or nothing at all. In addition, the name of the current player is displayed.

```
show_board :: !(Turn,TraxSt) -> [HtmlTag]
show_board (turn,traxSt={trax,names})
| nr_of_tiles trax == 0 = [h3 ("Select_any_tile,_" <+++ current_player)]
| otherwise             = [h3 current_player, board]
where board              = TableTag [BorderAttr "0"]
                        [ tr [ cell {col=minx + x - 1,row=miny + y - 1}
                              \\ x <- [0 .. nrcol + 1]
                              ]
                          \\ y <- [0 .. nrow + 1]
                          ]
  cell c                  = case tile_at trax c of
    Nothing               = if (isMember c free) (text c) (text "")
    Just tile              = td [TileTag (42,42) tile]
  current_player          = player names turn
  free                   = free_coordinates trax
  (nrcol,nrow)           = dimension      trax
  ((minx,maxx),(miny,maxy)) = bounds      trax
```

4 Related Work

The *n*-person turn-based game abstraction that is described in Sect. 3.2 bears a striking similarity with the **big-bang** abstraction that is provided in the *Racket world approach* [10]. This approach is designed to lower the threshold for beginning programmers to create interactive applications [12, 11]. The key element of this abstraction is the **big-bang** expression:

(**big-bang** *state-expr clause*⁺)

in which *state-expr* represents the initial state value that is shared in the world program (similar to the **st** type parameter of the **Game st** record) and *clause* is a tagged list that specifies the attributes and event handlers of the world program:

```
clause = (on-tick    tick-expr)
        | (on-tick    tick-expr rate-expr)
        | (on-tick    tick-expr rate-expr limit-expr)
        | (on-key     key-expr)
        | (on-pad      pad-expr)
        | (on-release release-expr)
        | (on-mouse    mouse-expr)
        | (to-draw     draw-expr)
        | (to-draw     draw-expr width-expr height-expr)
        | (stop-when   stop-expr)
        | (stop-when   stop-expr last-scene-expr)
        | (check-with  world?-expr)
```

```

| (record?    r-expr)
| (state      boolean-expr)
| (on-receive rec-expr)
| (register    IP-expr)
| (name       name-expr)

```

The event handlers are only concerned with the logical state of the world program and can be expressed as pure functions. For instance, the *tick-expr* of the **on-tick** clause is a pure function that computes a new state from the current one. The clause list must contain at least one **to-draw** member: *draw-expr* is a function that computes an image from the current state. Each time a new state is computed, this function is evaluated to create a new rendering of the game. In the *n*-person turn-based **Game st** abstraction, this corresponds with the **board** function, except that the latter generates a *html* rendering. The *stop-expr* of the **stop-when** clause has a similar role as the **over** predicate of the game abstraction and the *name-expr* of the **name** clause corresponds with the **game** member.

Racket world programs can be part of a distributed application using the *universe* abstraction. In a nutshell, a world program **registers** itself on a *server* identified by *IP-expr*. Event handlers either compute only a new state, as described above, or a pair of a new state *and* a message of some type. In the latter case, the message is sent to the server. A world program can receive messages from the server via the **on-receive** *rec-expr* function. The final component to be defined is the *server* which keeps track of registered world programs and the messages that are sent. It can serve as a broadcasting unit, or inspect messages to decide to what other worlds these should be sent.

The *Racket* universe approach to create distributed applications differs from the *iTask* approach. The main difference is that in *iTask* task distribution is accomplished from within the task specification and that communication occurs via observing each other's task value and shared data. In the *Racket* universe, world applications are more or less independent applications that use explicit message passing and receiving for communication.

5 Why Functional Programming Matters To Me

The case study in sections 2 and 3 shows that the glue that was identified by John Hughes is put to good use: higher order functions and lazy evaluation are used throughout the specification. This is also the case for other functional language features: list comprehensions deal with sets, lists, and streams in a uniform manner, the type class system unlocks useful functions for dedicated model data types, and generic functions capture type-dependent functionality in a single definition. However, having these language features available in a functional language only partially answers the question why functional programming matters to me. The other part of the answer concerns their impact on the way they help me to *solve problems*. Regardless of the programming paradigm, when solving a programming problem, I need to answer questions about that problem.

I illustrate this in terms of the case study. The first kind of question is always about the entities of the problem domain:

1. What are the entities in a game of *Trax*TM?

The answer is a collection of data types and their basic operations (most of them were defined in Sect. 2.1). Except for the choice to use streams to represent closed loops (Sect. 2.5), the data types in the case study are likely to result in similar representations in other programming languages and paradigms.

The second kind of question investigates the relation between these entities:

2. Which are the line colors of a (possibly empty) location in a trax?
3. When do two sets of line colors match?
4. Which tiles can be placed correctly at a free locations in a trax?
5. What trax results from performing the mandatory moves in a trax?
6. Which closed loops and winning lines does a trax contain?

The key observation is that these questions want to discover the relation between the entities and a well-defined result: this is exactly what functions and functional programming are about. *Functions are computable answers.* (Sections 2.2 – 2.5.)

The third kind of question investigates the relation with the end-users:

7. What is the order in which players take turns?
8. What is a player allowed to do?
9. When is a game of *Trax*TM over and who is the winner?

They investigate the concerted action between computable functions and user actions: this is exactly what tasks and *iTask* is about. Therefore, by specifying the corresponding tasks, you make the entities and functions tangible for users in the form of an executable application (Sect. 3).

Functional programming matters to me because it is a way of thinking and speaking that helps me to give the right answers to the right questions when solving computational problems.

Acknowledgements

I thank Pieter Koopman and Bas Lijnse for their advice during this case study. The reviewer's constructive comments have helped to greatly improve this essay. Finally, I thank Rinus for sharing his passion for functional programming, teaching, research, music, comic books, and elegant games.

References

1. Turner, D.: Miranda: A non-strict functional language with polymorphic types. In: Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture. Volume 201 of Springer Lecture Notes in Computer Science., Nancy, France (September 1985) 1–16

2. Turner, D.: An overview of Miranda. *SIGPLAN Notices* **21(12)** (December 1986) 158–166
3. Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., Sleep, M.: Term graph rewriting. In Bakker, Nijman, Treleaven, eds.: *Proceedings of Parallel Architectures and Languages Europe, PARLE '87*. Volume 259 of LNCS., Springer-Verlag (1987) 141–158
4. Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., Sleep, M.: Towards an intermediate language based on graph rewriting. In Bakker, Nijman, Treleaven, eds.: *Proceedings of Parallel Architectures and Languages Europe, PARLE '87*. Volume 259 of LNCS., Springer-Verlag (1987) 159–175
5. Hughes, J.: Why functional programming matters. *Computer Journal* **32(2)** (1989) 98–107
6. Hinze, R.: A new approach to generic functional programming. In Reps, T., ed.: *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00*, Boston, MA, USA, ACM Press (2000) 119–132
7. Alimarine, A.: *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen (2005) ISBN 3-540-67658-9.
8. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In Hinze, R., Ramsey, N., eds.: *Proceedings of the International Conference on Functional Programming, ICFP '07*, Freiburg, Germany, ACM Press (2007) 141–152
9. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-Oriented Programming in a Pure Functional Language. In: *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, Leuven, Belgium, ACM (September 2012) 195–206
10. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: *How to Design Programs*, Second Edition. MIT Press (2012) <http://www.ccs.neu.edu/home/matthias/HtDP2e/>.
11. Morazán, M.: Functional Video Games in the CS1 Classroom. In Page, R., Horváth, Z., Zsók, V., eds.: *Proceedings of the 11th Symposium on Trends in Functional Programming, TFP '10*. Volume 6546 of LNCS. (2010) 166–183
12. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: A Functional I/O System * or, Fun for Freshman Kids. In: *Proceedings International Conference on Functional Programming, ICFP '09*, Edinburgh, Scotland, UK, ACM Press (2009)