A Comparison of Task Oriented Programming with GUIs in Functional Languages

Peter Achten, Pieter Koopman, Steffen Michels, Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{p.achten,pieter,s.michels,rinus}@cs.ru.nl

research paper

Abstract. In this paper we compare the expressiveness of the Task Oriented Programming *iTask* approach of specifying interactive GUI applications with *ObjectIO* and *Racket big-bang*. *ObjectIO* is representative for the large class of traditional desktop widget based toolkits aiming to provide the programmer with full access the underlying GUI toolkit in a functional style. In contrast, the *Racket big-bang* approach offers the much more restricted setting of a single window and canvas to which the programmer adds callback and image rendering functions in a pure functional style. We demonstrate that both the *Racket big-bang* and *iTask* approaches result in significantly smaller GUI specifications by means of a small case study of the game of *tic-tac-toe*.

1 Introduction

Functional programming languages are known for allowing the concise specification of advanced data structures and algorithms. Several excellent books explain and illustrate this, consider for instance [1–7]. Although approaches differ, they express clear ideas on how to go about programming the "functional way". However, with respect to answering the question how to create GUI programs only a few of these books (e.g. [5, 7]) have clear answers. This is in contrast with the abundance of research that has been conducted on this subject during the past two decades (e.g. [8–25]). Although each of these approaches provide an answer how to program GUI applications in a functional language, they often do not result in concise solutions. In most cases, this is caused by the fact that they are restricted to providing an interface to the underlying GUI technology (desktop widget library or web-based), so the programmer is still exposed to the myriad of details of that technology.

In this paper we present the Task Oriented Programming (TOP) paradigm [26, 27] as an answer how to program GUI applications in a concise and functional style. In contrast to other approaches, TOP puts forward *observable tasks* and *task combinators* as the key ingredients to construct GUI applications. The *Clean iTask* library [28] is an implementation of this paradigm. This library is not a typical GUI library because it has been designed to specify the coordination the cooperation between human workers and computer systems using contemporary web browser technology for rendering purposes. Hence, by its very nature it is a multi-user system and not restricted to a particular underlying GUI technology. Nevertheless, GUI's can be specified in *iTask*. We perform a small case study of the well-known game of *tic-tac-toe*. The simplicity of this game allows us to concentrate on the GUI part. This is described in Sect. 5.

In order to substantiate the claim that the *Clean iTask* version is concise, we compare its solution with two other approaches. The first approach, described in Sect. 3, uses the *ObjectIO* library [11, 13] which is representative for a large class of approaches that have chosen to abstract over desktop widget-based GUI toolkits. Characteristic of these approaches is that the application logic is specified by means of *callback functions* and that the program manipulates *stateful widgets* to render the GUI. The second approach, described in Sect. 4, uses the *Racket big-bang* library. This approach was developed *"to reconciling I/O with purely functional programming, especially for a pedagogical setting"* [25]. In this solution the application logic is also specified by means of *callback functions*. However, to concentrate on the functional part of an application, its designers have rigorously eliminated everything that is concerned with manipulating stateful widgets and restricted the setting to a single window only. The GUI is rendered by means of a pure function that maps the program state to an image in a compositional way. For these reasons *Racket big-bang* applications are concise and can therefor be used to compare other approaches with.

In this paper we want to compare the approaches rather than the programming languages. For this reason all case studies are expressed in *Clean*. We have ported the relevant parts of the *Racket* libraries `universe` and `image` to *Clean ObjectIO*. In addition, to concentrate on the GUI specifications, we have moved the part of tic-tac-toe game that is concerned with the game logic to a separate `tictactoe` module that is described in Sect. 2.

In Sect. 6 we compare and discuss the three case studies. Sect. 7 presents related work, and Sect. 8 concludes. The versions are too long to be included completely in this paper. Their specifications are available at `https://svn.cs.ru.nl/repos/TicTacToeCaseStudies/`.

2 The Game Logic of Tic-tac-toe

In this section we present the `tictactoe` module that contains the data structures and computations that are used by all versions of the tic-tac-toe case study.

```
definition module tictactoe
import StdOverloaded, StdMaybe

:: Game        = { board :: TicTacToe      // the current board
                 , names :: Players        // the current two players
                 , turn  :: TicTac }       // the player at turn
:: Players     = { tic   :: Name           // the tic player takes even turns
                 , tac   :: Name   }       // the tac player takes odd turns
```

```
:: Name          :== String
:: TicTacToe     :== [[Tile]]
:: Tile            = Clear | Filled TicTac
:: TicTac          = Tic | Tac
:: Coordinate      = {col :: Int, row :: Int}
instance           ~ TicTac
instance          == TicTac, Tile, Coordinate

name_of          :: Players TicTac -> Name
init_game        :: Players        -> Game
game_over        ::                   TicTacToe -> Bool
it_is_a_draw     ::                   TicTacToe -> Bool
winner           ::                   TicTacToe -> Maybe TicTac
free_coordinates ::                   TicTacToe -> [Coordinate]
add_cell         :: Coordinate TicTac TicTacToe -> TicTacToe
tiles            ::                   TicTacToe -> [(Coordinate,Tile)]
```

The minimum information required to guide a game of tic-tac-toe is collected in the `Game` record, containing the current board, the names of the two players, and their turn. An initial game situation for two players is created by `init_game`. A tic-tac-toe board is represented as a 3×3 matrix of tiles. In an initial board, all tiles are `Clear`. Tile coordinates are zero-based and run from left-to-right and top-to-bottom. When player `t` updates a tile at coordinate `c` in board `b`, then this results in a new board `add_cell c t b`. The coordinates of all `Clear` cells in a board `b` are returned by `free_coordinates b`, and the current status of all tiles is returned by `tiles b`. The first player who succeeds in filling a horizontal, vertical, or diagonal line of exclusively `Tic`s or `Tac`s wins, and is determined by the function `winner b`. When the board is entirely filled but does not produce a winner, then the game has come to a draw, which is computed by `it_is_a_draw b`. The function `game_over` computes whether a game is over because somebody has won or the game has come to a draw.

3 Tic-tac-toe in Object I/O

In this section we present the *ObjectIO* version of tic-tac-toe. Because of the size of the specification, 117 lines of type and function definitions, we highlight only the key parts. In *ObjectIO*, the GUI is rendered by means of stateful widgets (windows, controls, canvas, and so on). The behavior of these elements is controlled by means of callback functions that manipulate both the state of the widgets as well as the shared logical state of the program. It is the task of each and every callback function to keep the state of the widgets 'in sync' with the logical state of the program. The logical state, `GameSt`, is the `Game` defined in Sect. 2 as well as a record of the identification values of the stateful widgets:

```
:: GameSt  = { game    :: Game      // the current game
             , ids     :: GameIds } // the GUI identification values
:: GameIds = { nameId  :: Id        // the name tag of the current player
             , turnId  :: Id        // the turn-indicator
```

```
            , tileIds :: [Id] }        // the tic-tac-toe tiles
```

Fresh `GameIds` are created with `open_GameIds`. GUI programs start with `startIO`:

```
1 Start :: *World -> *World
2 Start world
3 # (ids,world) = open_GameIds world
4 = startIO SDI {game = init_game names, ids = ids}
5               (start_tictactoe o get_player_names names)
6               [ProcessClose closeProcess] world
7 where names = {tic="Mr.␣Tic",tac="Mrs.␣Tac"}
```

The key parameters are the initial value of the shared logical state (the second argument on line 4) and the GUI initialization function (line 5) which first asks the user to enter more appropriate names than the suggested ones (`get_player_names`) after which it can actually create the GUI (`start_tictactoe`).

Getting the player names is not a difficult task (Fig. 1 (left)), yet the specification is verbose. A modal dialog and its controls have to be created. The button callback function needs to read the text input elements and close the dialog and its controls. The key aspect of `get_player_names` is its signature:

```
get_player_names :: Players (PSt .ls) -> (Players,PSt .ls)
```

The polymorphic type tells us that this task works for any interactive program. More importantly, the new player names are not stored as a side effect in the process state but returned as an ordinary function result. This is only possible because this function creates and fully handles a modal dialog.

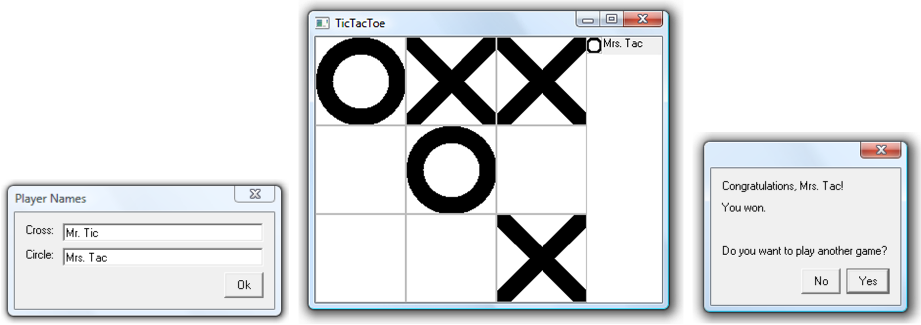

Fig. 1. The main screens in *ObjectIO*.

The `start_tictactoe` function receives the new player names and creates the main interface: a single window that displays the current board as well as an indication which player is currently playing (Fig. 1 (middle)). Despite its verbosity we show its complete specification.

```
start_tictactoe :: (Players,PSt GameSt) -> PSt GameSt
```

```
2  start_tictactoe (names,pSt=:{ls=gameSt=:{ids}})
3  # gameSt = {gameSt & game = init_game names}
4  # window = Window "TicTacToe"
5            (   LayoutControl
6            (   ListLS
7                [ ListLS
8                  [ let tileId = ids.tileIds !! (row*3+col)
9                        coord  = {col=col,row=row}
10                   in CustomButtonControl {w=(wsize.w-textw)/3,h=wsize.h/3}
11                                          (tile_look Clear)
12                      [ ControlId         tileId
13                      , ControlFunction   (noLS (tile_pressed tileId coord))
14                      , ControlResize     resize_a_third
15                      , ControlPos        (if (col == 0) Left RightToPrev,zero) ]
16                 \\ col <- [0..2] ]
17               \\ row <- [0..2] ]
18           )   [ ControlResize   resize_proportional
19               , ControlViewSize {wsize & w=wsize.w-textw} ]
20           :+: CustomControl {w=boxw,h=boxw} (tile_look (Filled gameSt.game.turn))
21                                    [ ControlId ids.turnId ]
22           :+: TextControl names.tic [ ControlId ids.nameId
23                                     , ControlWidth (PixelWidth (textw-boxw)) ]
24           )
25           [ WindowClose    (noLS closeProcess)
26           , WindowViewSize wsize ]
27 = snd (openWindow Void window {pSt & ls = gameSt})
28 where wsize = {w=600, h=600}
29       boxw  = 18
30       textw = 80
```

Instead of rendering the board as a single image we prefer to create it as a composition of nine interactive elements (lines 4-17) that control one tile each. This simplifies rendering (`tile_look`) and handling user interaction (`tile_pressed`). In addition, `tile_look` is reused to indicate the current user glyph (lines 20-21).

The tile rendering function manipulates a canvas of abstract type `*Picture`. We include its specification to illustrate the details one is concerned with despite the fact that only very basic graphics need to be produced.

```
tile_look :: Tile SelectState UpdateState *Picture -> *Picture
tile_look tile selectSt {newFrame} picture
                = frame (cell (unfill newFrame picture))
where {x,y}     = newFrame.corner1
      {w,h}     = rectangleSize newFrame
      linewidth = min (w/5) (h/5)
      (mw,mh)   =     (w/2,  h/2)
      cell      = case tile of
                    Clear    = id
                    Filled t = appPicture  ( if (t == Tic) cross nought
                                           o setPenSize linewidth )
      frame     = appPicture (draw newFrame o setPenColour LightGrey)
```

```
nought   = drawAt {x=x+mw,y=y+mh} {oval_rx=mw,oval_ry=mh}
cross    =   drawLine {x=x,  y=y} {x=x+w,y=y+h}
           o drawLine {x=x+w,y=y} {x=x,y=y+h}
```

Whenever the player presses the customized tile button control the callback function `tile_pressed` is evaluated:

```
1 tile_pressed :: Id Coordinate (PSt GameSt) -> PSt GameSt
2 tile_pressed tid c pSt=:{ls=gameSt=:{game=game=:{board,names,turn},ids},io}
3 # io     = disableControl tid io
4 # io     = setControlLook tid True (True,tile_look (Filled turn)) io
5 # io     = setControlText ids.nameId (name_of names (~turn)) io
6 # io     = setControlLook ids.turnId True (True,tile_look (Filled (~turn))) io
7 # gameSt = {gameSt & game = {game & board = add_cell c turn board, turn = ~turn}}
8 # pSt    = {pSt & ls = gameSt, io = io}
9 = check_game_over pSt
```

The key aspect to observe is that this function has an effect on the stateful widgets (lines 3-6) that have to be kept 'in sync' with the logical state (line 7). Both states are passed to the function that checks whether the game is over.

```
check_game_over :: (PSt GameSt) -> PSt GameSt
check_game_over pSt=:{ls={game={board,names,turn}}}
| game_over board = openNotice notice pSt
| otherwise = pSt
where
  won      = winner board
  accolade = if (isNothing won)
               ["It␣is␣a␣draw."]
               ["Congratulations,␣" +++ name_of names (~turn) +++ "!","You␣won."]
  notice   = Notice (accolade ++ ["","Do␣you␣want␣to␣play␣another␣game?"])
               (NoticeButton "Yes" (noLS (new_game o get_player_names names)))
               [NoticeButton "No"  (noLS closeProcess)]
```

Whenever the game happens to be over, a notice (Fig. 1 (right)) is opened to congratulate the player. If the player presses the "No" button, then the entire program is stopped. If she decides to play a new game, she can enter new player names and have the `new_game` function take care of the remaining details:

```
new_game :: (Players,PSt GameSt) -> PSt GameSt
new_game (names,pSt=:{ls,io})
= {pSt & ls = gameSt
       , io = setControlText  gameSt.ids.nameId gameSt.game.names.tic
             (setControlLook  gameSt.ids.turnId True
                                (True,tile_look (Filled gameSt.game.turn))
             (setControlLooks [ (tid,True,(True,tile_look Clear))
                              \\ tid <- gameSt.ids.tileIds ]
             (enableControls  gameSt.ids.tileIds io)))
  }
where gameSt = {ls & game = init_game names}
```

All tiles are cleared and made available for playing, the first player's glyph and name is set, and the process state is 'synced'.

4 Tic-tac-toe in Racket big-bang

In this section we present tic-tac-toe using the *Racket big-bang* approach. It consists of 63 lines of type and function specifications. The conciseness is mainly due to the absence of stateful widgets. Instead, the programmer designs a logical state model that reflects the stages an interactive program passes through. Rendering is handled by a pure function that maps this logical state to an image. In this way the rendering of the GUI is automatically kept 'in sync' with the logical state. Similarly, only one keyboard and mouse callback function need to be specified. They are only concerned with the logical state. Termination is handled via a predicate that tests the current state value, again a pure function. From this account it follows that the logical state should be designed first:

```
:: GameSt = EnterNames Players TicTac | Play Game | Accolades Game | Stop

initGameSt :: Players -> GameSt
initGameSt players = EnterNames players Tic
```

`GameSt` describes the separate stages of tic-tac-toe (entering names, playing the game, receiving accolades, and termination).

An interactive application is started using the `big_bang` function.

```
1 Start :: *World -> (GameSt,*World)
2 Start world = big_bang (initGameSt {tic="Mr.␣Tic",tac="Mrs.␣Tac"})
3                     [ Name      "Tic␣Tac␣Toe"
4                     , To_draw   (render wsize, Just (wsize.w,wsize.h))
5                     , On_key    keys
6                     , On_mouse  mice
7                     , Stop_when (\game -> case game of Stop -> True; _ -> False)
8                     ] world
9 where wsize = {w=450,h=375}
```

Besides the initial state (line 2), `big_bang` needs to know the state transition functions. These are `keys` and `mice` that handle keyboard and mouse input respectively. The termination predicate `gameOver :: GameSt -> Bool` returns `True` only for the `Stop` game state. The only mandatory clause of `big_bang` is the rendering function, `render`, which is parameterized with the canvas size. This overloaded operation renders the various components (`GameSt`, `TicTacToe`, `Tile`, and `TicTac`).

Racket has a comprehensive drawing package, `2htdp/image`, in which images are specified in a compositional style rather than canvas-modifying operations. For the sake of this case study we have implemented only a small part of the package: basic images (`empty_image`, `text`, `circle`, `rectangle`, and `square`) and image combinators (`add_line`, `overlay(_align)`, `beside(_align)`, and `above(_align)`). Rendering the `GameSt` (see Fig. 2) covers a case for each stage of the game state:

```
class render a :: Size a -> Image
instance render GameSt where
  render size (EnterNames players turn)
    = overlay [ above [ text ("Player␣" <+ if (turn==Tic) 1 2) 24 Black
                      , overlay [ text (name_of players turn)  24 Black
```

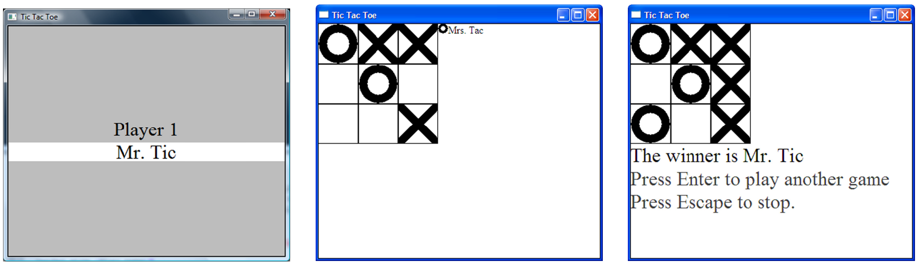

Fig. 2. The main screens in *Racket big-bang*.

```
6                                 , rectangle size.w 30 Solid White ] ]
7                  , rectangle size.w size.h Solid LightGrey ]
8  render size (Play {board,turn,names})
9    = beside_align TopY [ render size board
10                       , render {w=16,h=16} turn
11                       , text (name_of names turn) 12 Black ]
12 render size (Accolades {board,turn,names})
13   = above_align LeftX [ render size board
14                       , case winner board of
15                           Just t  = text ("The winner is " <+ name_of names t)
16                                                       24 Black
17                           nothing = text "It's a draw..."      24 Black
18                       , text "Press Enter to play another game" 24 DarkGrey
19                       , text "Press Escape to stop."            24 DarkGrey ]
20 render size Stop
21   = empty_image
```

For entering the player names, we have chosen to design a simple screen in which the name of one player is entered (lines 3-7). When playing the game, the tic-tac-toe board, the player glyph and name are displayed beside each other (lines 8-11). When receiving the accolades, the final board is shown above the name of the winner, if any, and information how to proceed (lines 12-19). The final state is rendered as the empty image (lines 20-21).

In order to make the rendering complete, the instances for the remaining components need to be defined. They are self-explanatory:

```
instance render TicTacToe where
  render size board          = above [ beside [  render {w=64,h=64} cell
                                              \\ cell <- row ]
                                     \\ row <- board ]
instance render Tile where
  render {w,h} Clear         = square (min w h) Outline Black
  render {w,h} (Filled turn) = overlay [ square (min w h) Outline Black
                                       , render {w=w-2,h=h-2} turn ]
instance render TicTac where
  render {w,h} Tic           = add_line (add_line empty_image
```

```
                                                 0 0 w h Black ((min w h)/5))
                                                 w 0 0 h Black ((min w h)/5)
  render {w,h} Tac           = overlay [ circle (3*(min w h)/10) Solid White
                                       , circle (  (min w h)/2 ) Solid Black ]
```

The keyboard plays a role in the `EnterNames` and `Accolades` stage of tic-tac-toe.

```
1  keys :: GameSt KeyEvent -> GameSt
2  keys (EnterNames players turn) key
3  | key == "\\r"        = if (turn==Tic) (EnterNames players Tac)
4                                         (Play (initGame players))
5  | key == "\\b"        = EnterNames (alterName initStr players turn) turn
6  | key == "␣" || alpha = EnterNames (alterName (flip (+++) key) players turn) turn
7  | otherwise           = EnterNames players turn
8  where alpha           = isAlpha (hd (fromString key))
9  keys (Accolades game) key
10 | key == "\\r"        = EnterNames game.names Tic
11 | key == "escape"     = Stop
12 keys state _          = state
13
14 alterName :: (String -> String) Players TicTac -> Players
15 alterName f players Tic  = {players & tic = f players.tic}
16 alterName f players tac  = {players & tac = f players.tac}
```

For entering the player names a simplified text-input element is created (lines 2-8). When the user enters the return key, the next name should be entered (line 3) or the game commences (line 4). The only 'edit' key that is handled is the backspace key (line 5). The name gets extended with the current key if it is a letter or a space (line 6). All other keys do not alter the state (line 7). The only role of the keys handler in case of the accolades (lines 9-11) is to allow the players to choose to either play again (line 10) or stop the program (line 11).

The mouse only plays a role in the `Play` stage of tic-tac-toe.

```
1 mice :: GameSt Int Int MouseEvent -> GameSt
2 mice (Play game) x y mouse
3 | mouse == "button-down" && isMember c (free_coordinates game.board)
4   # game    = {game & board = add_cell c game.turn game.board, turn = ~game.turn}
5   = if (game_over game.board) Accolades Play game
6 where c     = {col = x / 64, row = y / 64}
7 mice state _ _ _
8   = state
```

Only if the mouse is pressed in a free tile (line 3), the board gets updated and the next player proceeds (line 4). If the game happens to be over, then the program moves on to the `Accolades` stage, otherwise it remains in the `Play` stage (line 5).

5 Tic-Tac-Toe in Task Oriented Programming

In this section we present the TOP *iTask* approach to specify tic-tac-toe. This specification consists of 40 lines of type and function definitions. One key contributing factor to its brevity is that the application's task structure can be expressed in a direct manner with tasks and combinators instead of indirectly via callback functions that inspect the shared program state. In addition, a large part of the GUI is derived generically from the data models that are used in the specification. Only the rendering of the board requires separate attention. The main screens of this version are shown in Fig. 3.

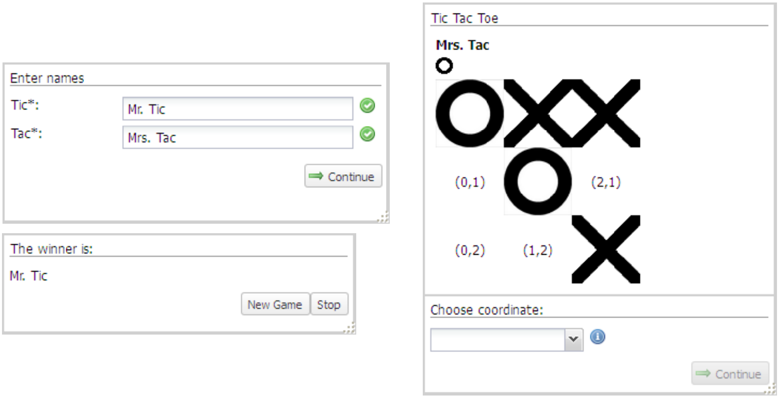

Fig. 3. The main screens in *iTask*.

Analogous to the other versions, the game starts with an initial suggestion for the player names:

```
1 play_tictactoe :: Task Void
2 play_tictactoe = game {tic = "Mr.␣Tic", tac = "Mrs.␣Tac"}
3
4 game :: Players -> Task Void
5 game players   =                      new_names players
6                  >>= \next_players -> play (init_game next_players)
7                  >>= \winner       -> accolades next_players winner
```

A game consists of three consecutive tasks: ask the user to enter new names (line 5), then playing the game until the end (line 6), and presenting the accolades to the winner, if any (line 7).

The `new_names` task uses an *update task* that provides the user with means to alter the provided information in a type-safe way (Fig. 3 (left-top)).

```
new_names :: Players -> Task Players
new_names players = updateInformation "Enter␣names" [] players
```

The `play` task uses the `Game` structure defined in Sect. 2 to keep track of the game progress (Fig. 3 (right)).

```
:: GameSt :== Game

play :: GameSt -> Task Name
```

```
4  play game=:{board,turn}
5  = viewInformation "Tic␣Tac␣Toe" [ViewWith show_board] game
6       ||-      enterChoice "Choose␣coordinate:"
7                            [ChooseWith ChooseFromComboBox toString]
8                            (free_coordinates board)
9    >>= \new ->
10   let game‘ = { game & board = add_cell new turn board, turn  = ~turn }
11    in if (not (game_over game‘.board)) (play game‘)
12      (if (it_is_a_draw game‘.board)    (return "nobody")
13          (return (name_of game‘.names (fromJust (winner game‘.board)))))
```

While displaying the state of the game (line 5), the `play` task asks the user to choose one of the available free tiles (lines 6-8). This is recorded in the game state (line 10). Finally, in case the game is over, the winner's name is returned, if any (line 11-13).

The `accolades` task is provided with the player and winner names:

```
1  accolades :: Players Name -> Task Void
2  accolades players winner
3      = viewInformation "The␣winner␣is:" [] winner
4        >>* [ Always (Action "New␣Game" []) (game players)
5            , Always (Action "Stop"     []) (return Void) ]
```

The user can choose to play another game (line 4) or stop altogether (line 5).

The final part of the specification defines the rendering of the tic-tac-toe board (Fig. 3 (right)). The function `show_board` transforms a game state to a rendering in *html*:

```
1  show_board :: GameSt -> [HtmlTag]
2  show_board {board,names,turn}
3    = [H3Tag [] [Text (name_of names turn)], TileTag (16,16) turn, tictactoe]
4  where
5    tictactoe = TableTag [BorderAttr "0"]
6                         [ TrTag [] [ cell {col=x,row=y} \\ x <- [0..2] ]
7                         \\ y <- [0..2] ]
8    cell c    = case lookup1 c (tiles board) of
9                  Filled t = TdTag [] [TileTag (64,64) t]
10                 clear    = TdTag [AlignAttr "center"] [Text (toString c)]
11
12 TileTag (w,h) t = ImgTag [ SrcAttr    ("/" <+++ t <+++ ".png")
13                          , WidthAttr  (toString w)
14                          , HeightAttr (toString h) ]
```

The rendering displays the current player, the glyph she is playing with, and the board (line 3). Image files (`"Tic.png"` and `"Tac.png"`) have to be used to render the cross and nought glyphs because in the version described in this paper, *iTask* has no canvas support.

6 Comparison

In this section we compare and discuss the versions of the tic-tac-toe case study.

The first aspect to compare concerns the code size of the GUI specifications: *ObjectIO*: 117 *loc* (100%), *Racket big-bang*: 63 *loc* (54%), *iTask*: 40 *loc* (34%). In the *ObjectIO* version, 60 *loc* are necessary to define the modal dialogs, the main window and its components. This explains why the *Racket big-bang* version is proportionally shorter: it already implements the infrastructure for a single-document application. Both in *ObjectIO* and *Racket big-bang* the three separate stages of the application (entering names, playing the game, giving the accolades) need to be rendered explicitly. This explains why the *iTask* version is proportionally shorter than the other two versions: all screens except the game playing screen are derived automatically from the model types.

In order to make a fair comparison, the three tic-tac-toe versions should provide identical user interfaces. This has not succeeded. The *Racket big-bang* version re-invents text-edit functionality in the entering names screen and refrains from re-inventing buttons in the accolades screen, and instead takes an escape route by using the keyboard to allow the players to choose whether or not to continue playing. At this stage, the *iTask* version has no stable support for defining customized tasks that use canvas style graphics in combination with event handlers. Instead we had to resort to providing the user with a choice between the available tiles which results in a less intuitive and somewhat disconnected user experience.

The *ObjectIO* and *Racket big-bang* versions use pure functions (`tile_look` and the `render` functions respectively) to render the game. In *ObjectIO* this is a `*Picture` transformer function, whereas in *Racket big-bang* it computes an `Image` in a compositional way. The required graphics for rendering a board are simple: a rectangle that is either empty or filled with a circle or two lines. Only the *Racket big-bang* version is proportional to this simple task: the `Tile` and `TicTac` instances of the `render` function formulate the above characterization of the graphics in a concise way. In the *iTask* version we were forced to 'cheat' by rendering these pictures by means of pre-rendered bitmaps. It should be mentioned that we could have done this in *Racket* as well because bitmaps are first-class `Image`s.

The *ObjectIO* version 'switches off' tiles after being pressed, and uses this to its advantage because it does not have to check whether a clear tile has been selected. Obviously, the disadvantage is that when starting another game, you should not forget to 'switch on' all tiles. In the *Racket big-bang* version this test is required because the state transition must be defined for any possible mouse event. The TOP version mimics the behavior of the *ObjectIO* version by limiting the user's choice to the empty tiles of the board. In this way, it makes explicit what is implicit in the *ObjectIO* version, and what is tested afterwards in the *Racket big-bang* version.

The effort required to 'distill' the application behavior varies greatly for the three versions. In *ObjectIO* we must unravel the rendering and logical operations and keep track of their effects on both the logical state and the set of stateful widgets. The callback functions clearly illustrate that it is the responsibility of the programmer to keep the logical state 'in sync' with the set of stateful widgets. In *Racket big-bang* the situation is much easier because synchronization is dealt with by the system. To understand the program, it suffices to study the state and the mouse and keyboard handler state transitions. This amounts to discovering the underlying logical state machine and its transitions. Because *iTask* has been developed explicitly to deal with tasks and their evaluation order, understanding the behavior of the application requires the least effort.

7 Related Work

The *Racket* `2htp/image` approach of defining graphics in a compositional way fits in a long tradition that can be traced back to Peter Henderson's Functional Geometry [29]. In a follow-up paper [30] he comments: *"This idea is not new. It was published in 1982, but even then it was based on contemporary views of what was good practice in declarative systems."*. The GUI library Haggis [10] uses a similar compositional approach and extends it to build the entire GUI of an application.

As we have stated in the introduction, there is no lack of research on how to program GUI applications in functional languages. A large class of these solutions are more or less traditional, stateful, callback-oriented GUI libraries in a spirit similar to *ObjectIO* ([8, 12, 20, 22] and many more). We conjecture that in these approaches the case study will not differ too much in terms of *loc* and logical structure. In this paper we have not studied the brand of functional programming that is known as *reactive animation*. This paradigm was started by the seminal paper by Elliot and Hudak [31] and spawned a number of related approaches which are enumerated elsewhere [17]. These approaches take a radically different view on interactive programs, using time-dependent units as building blocks instead of stateful event handlers as in *ObjectIO* or *Racket big-bang* or observable tasks as in *iTask*.

8 Conclusions

In this paper we have presented a case study of a GUI application, tic-tac-toe, expressed in three different formalisms: *ObjectIO*, *Racket big-bang*, and TOP *iTask*. The purpose of this case study is to compare the different formalisms with respect to their ability to concisely and clearly specify a GUI application. All versions use the same `tictactoe` module for the game logic which consists of 61 *loc* of type and function definitions. None of the approaches result in large specifications: the largest, *ObjectIO*, is 117 *loc*. Their relative sizes vary greatly: in comparison with the *ObjectIO* version (100%) the size of the *Racket big-bang* version is 54% and the TOP *iTask* version is 34%. These numbers should not be interpreted in a very strict manner because the line count is very dependent on the layout of the code. We have attempted to define the versions in the style that is conventional for the approaches. Nevertheless, it gives an indication of the conciseness of the formalism.

In comparison with *ObjectIO*, the *Racket big-bang* version offers a similar user-experience with respect to playing the game. However, we are 'forced' to solve the task of entering player names and choosing how to continue after the end of a game in a somewhat ad hoc way. The TOP *iTask* version does not suffer from this issue but instead offers an awkward user experience in playing the game because the current version lacks facilities to define manipulatable graphics. This will be possible in a next version of the *iTask* system. It is interesting to investigate to what extent the compositional style of graphics specification of the *Racket* `image` library can be used.

Of the three versions, the application behavior is hardest to distill in the *ObjectIO* version because the callback functions need to concern themselves with the details of manipulating the set of stateful widgets as well as the logical state. This is less of an issue in the *Racket big-bang* version because the rendering of the GUI is synced automatically with the logical state. In this approach the application is modeled as a state machine. The transitions are defined by the event handlers. The advantage of this approach is that it is clear for the modeler where to define the transitions, and where to look for when uncovering the state machine. However, just as with *ObjectIO*, the flow of control is present only implicitly. The TOP *iTask* version makes the application flow of control explicit. The generic abstractions take care of the automatic synchronization of the application state with respect to its rendered GUI.

As a final remark we point out that in all three systems it is possible to turn the case study into a *distributed* version. In *ObjectIO* the programmer can use the standard *TCP* library. Consequently, this addition will have a relatively large impact on implementing a distributed version in *ObjectIO*. *Racket* provides similar direct support for *TCP*, but in addition, it extends the *big-bang* approach with means to communicate directly with other *big-bang* 'worlds', which together form the 'universe'. After registering, any callback function can send extra information to the server world with it has connected. The server world program forwards this message to the other registered worlds. To receiver these messages, a *big-bang* program must have created another event handler that is called whenever a message is received. Consequently, in *Racket big-bang* creating a distributed tic-tac-toe version proceeds in an analogous way and is proportional to the task. Finally, in *iTask* work distribution is integrated by design: any task (composition) t can be distributed to any worker u in the system by the task assignment operator, u `@:` t. Therefor, a distributed version of tic-tac-toe amounts to creating a task structure in which all workers have a view task on the current board, modeled as a shared state, and only one worker can update this shared state at a time.

Acknowledgements

The authors thank Marco T. Morazán for answering our questions of *Racket*, the `Universe`, and everything. We thank Bas Lijnse for his advice on *iTask*.

References

1. Bird, R., Wadler, P.: Introduction to functional programming. Prentice Hall (1988)

2. Okasaki, C.: Purely Functional Data Structures. Cambridge Univ. Press (1998)
3. Bird, R.: Introduction to functional programming using Haskell (second edition). Prentice Hall (1998)
4. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: How to Design Programs: An introduction to programming and computing. MIT Press (2001)
5. Hudak, P.: The Haskell school of expression: learning functional programming through multimedia. Cambridge University Press, New York, NY, USA (2000)
6. Hutton, G.: Programming in Haskell. Cambridge University Press (2007)
7. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: How to Design Programs, Second Edition. MIT Press (2012)
8. Dwelly, A.: Functions and dynamic user interfaces. In: Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA '89. (September 1989) 371–381
9. Carlsson, M., Hallgren, T.: Fudgets - a graphical user interface in a lazy functional language. In: Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture, FPCA '93, Kopenhagen, Denmark (1993)
10. Finne, S., Peyton Jones, S.: Composing Haggis. In: Eurographics Workshop on Programming Paradigms in Graphics, Maastricht, the Netherlands, Springer (1995) 85–101
11. Achten, P., Plasmeijer, R.: The ins and outs of Concurrent Clean I/O. Journal of Functional Programming **5**(1) (1995) 81–110
12. Claessen, K., Vullinghs, T., Meijer, E.: Structuring graphical paradigms in TkGofer. In: Proceedings of the 2nd International Conference on Functional Programming, ICFP '97. Volume 32(8)., Amsterdam, The Netherlands, ACM Press (9-11, June 1997) 251–262
13. Achten, P., Plasmeijer, R.: Interactive functional objects in Clean. In Clack, C., Hammond, K., Davie, T., eds.: Selected Papers of the 9th International Workshop on the Implementation of Functional Languages, IFL '97. Volume 1467 of LNCS., Springer-Verlag (September 1998) 304–321
14. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Proceedings of the 5th Haskell Workshop, Haskell '01. (September 2001)
15. Hanus, M.: High-level server side web scripting in Curry. In: Proceedings of the 3rd International Symposium on the Practical Aspects of Declarative Programming, PADL '01, Springer-Verlag (2001) 76–92
16. Achten, P., Peyton Jones, S.: Porting the Clean Object I/O library to Haskell. In Mohnen, M., Koopman, P., eds.: Selected Papers of the 12th International Workshop on the Implementation of Functional Languages, IFL '00. Volume 2011 of LNCS., Springer-Verlag (September 2001) 194–213
17. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Proceedings of the 4th International Summer School on Advanced Functional Programming, AFP '03, Oxford, UK
18. Graunke, P., Findler, R., Krishnamurthi, S., Felleisen, M.: Modeling web interactions. In Degano, P., ed.: Proceedings of the 12th European Symposium on Programming, ESOP '03. Volume 2618 of Lecture Notes in Computer Science.
19. Elsman, M., Hallenberg, N.: Web programming with SMLserver. In: Proceedings of the 5th International Symposium on the Practical Aspects of Declarative Programming, PADL '03, New Orleans, LA, USA, Springer-Verlag (January 2003)
20. Leijen, D.: wxHaskell: a portable and concise GUI library for Haskell. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, Snowbird, Utah, USA, ACM (2004) 57–68

21. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: web programming without tiers. In: Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06. Volume 4709., CWI, Amsterdam, The Netherlands, Springer-Verlag (7-10, November 2006)
22. Hanus, M.: Type-oriented construction of web user interfaces. In: Proceedings of the 8th International Conference on Principles and Practice of Declarative Programming, PPDP '06, ACM Press (2006) 27–38
23. Loitsch, F., Serrano, M.: Hop client-side compilation. In: Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '07, New York, NY, USA, Interact (2-4, April 2007) 141–158
24. Elliot, C.: Tangible functional programming. In: Proceedings of the 12th International Conference on Functional Programming, ICFP '07, Freiburg, Germany, ACM Press (1-3, October 2007) 59–70
25. 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)
26. 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
27. Lijnse, B.: TOP to the Rescue – Task-Oriented Programming for Incident Response Applications. PhD thesis, Radboud University Nijmegen (2013)
28. 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
29. Henderson, P.: Functional geometry. In Friedman, D., Wise, D., eds.: Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania, ACM Press (1982) 179–187
30. Henderson, P.: Functional geometry. Higher-Order and Symbolic Computation **15** (2002) 349–365
31. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the second ACM SIGPLAN international conference on Functional Programming, Amsterdam, The Netherlands, ACM (1997) 263–273