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

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 during the past two decades in this area. This research can roughly be grouped into three categories: *widget-based* (e.g. [8–14]), *web-based* (e.g. [15–21]), and *function-based* (e.g. [22–25]).

Amongst this multitude of approaches we also find the Task Oriented Programming (TOP) paradigm [26, 27]. This approach has emerged during the past decade. During its evolution, it has taken its course via the above three categories and has developed and picked up core ideas in these areas. The first idea is that *types should drive the generation of GUIs*, which was first tested on widget based GUIs (*graphical editor components*) [28]. Second, *functions should define the relation between interactive components*, which was tested on web based programming (*iData*) [29]. Third, these functions should be *combinators to specify the flow of control and data between tasks* (*iTask*) [30]. Tasks embody work that has to be performed in a system, either by humans or computers, and during their execution they can be continuously observed by other tasks to see whether this should influence their own behavior.

In this paper we present the TOP approach to programming GUI applications. We illustrate this with a small case study for the well-known game of *tic-tac-toe*. The simplicity of this game allows us to concentrate on how to program its GUI. We compare the TOP specification with one written in *Object I/O* [31] and a *Clean* version of *Racket*'s `big-bang` approach [25, 7]. Object I/O is selected because it is representative for the class of widget-based I/O approaches. Racket `big-bang` is selected because it takes a more abstract, and simplified, approach.

The remainder of this paper is organized as follows. In Sect. 2 we identify the data structures and functions that are used by all versions to implement the game logic. The versions are presented in the following order: Object I/O, Racket

big-bang, and TOP iTask (Sections 3, 4, 5). We compare the versions in Sect. 6. Related work is discussed in Sect. 7 and we conclude in Sect. 8. 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
import 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
initGame         :: Players        -> Game
emptyBoard       :: TicTacToe
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 that is needed to guide a game of tic-tac-toe is collected in the `Game` record: it contains the current board, the names of the two players, and their turn. An initial game situation for two players `names` is created by `initGame names`. A tic-tac-toe board is represented as a 3×3 matrix of tiles. An initial board, in which all tiles are `Clear`, is defined by `emptyBoard`. Tiles can be updated by the players in turns. The player taking the even turns fills it with `Tic`s and the other player with `Tac`s. The coordinates to indicate individual cells are zero-based and run from left-to-right and top-to-bottom, so `{col=0, row=0}` is at the left-top, and `{col=2, row=2}` is at the right-bottom. When player `t` updates

a cell 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 or 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 Object I/O version of tic-tac-toe. Object I/O [10, 12, 31] is a widget-based GUI library. It offers features that are quite common with these kinds of libraries: a programmer has to concern herself with managing widgets, callback functions, and programming their mutual effects via shared state. In Object I/O this state is passed explicitly and in a hierarchical manner: within an interactive process, the top-level GUI elements (windows, dialogs, menus, timers, and receivers) share a common state, and every GUI element has the opportunity to define state locally that is accessible only by its components. Hence in Object I/O, a programmer has to decide how to use the state model. In this case we use the `Game` defined in Sect. 2 as well as a dedicated record to keep track of the identification values of the GUI components:

```
:: 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
             , nameIds :: (Id,Id)    // the tic / tac player text fields
             }
```

The `GameIds` are created with the function `openGameIds`, the implementation of which does not concern us right now. The initial state is given at the start:

```
Start :: *World -> *World
Start world
# (ids,world) = openGameIds world
= startIO SDI {ids = ids, game = initGame {tic="Mr. Tic",tac="Mrs. Tac"}}
              (startTicTacToe o getPlayerNames)
              [ProcessClose closeProcess]
              world
```

The Object I/O wrapper function `startIO` takes a number of arguments to get started with a GUI application. The first argument, `SDI`, lets it create GUI infrastructure for a program that uses at most one window (we use the same option to create the Racket `big-bang` function). The second argument identifies the shared state of this application. The third argument initializes the GUI of the application. In this case, its first task is to ask the user to enter more appropriate

names than the ones initially suggested, and then start the game. The fourth argument is an attribute of the interactive process that is created and contains the callback function that must be evaluated whenever the user wishes to close the interactive process. In this case the application terminates the entire interactive process, and therefore also the evaluation of `startIO`.

Getting the player names is not a difficult task (Fig. 1 (left-top)), yet its specification is very verbose. It takes 27 lines of code to specify the dialog, create it, and close it after reading the names that are entered by the user. When terminated, the `game.names` field contains the entered names.

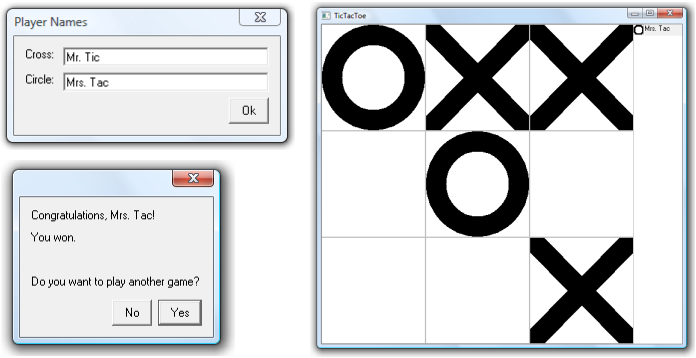

Fig. 1. Three main screens in tic-tac-toe: view and enter player names (left-top), playing the game (right), and congratulate winner (left-bottom).

The purpose of the `startTicTacToe` task is to create the main interface which consists of a single window in which the current board is displayed as well as an indication which player is currently playing (Fig. 1 (right)). This function is even more verbose and requires 39 lines. Instead of rendering the board as a single image, it is composed of 9 button controls with a customized look. With these tile controls the player updates a single tile in the board. Therefore, it is parameterized with the coordinate of that tile.

```
TileControl :: Id Coordinate Size [ControlAttribute (ls,PSt GameSt)]
   -> CustomButtonControl ls (PSt GameSt)
TileControl tid c size atts
    = CustomButtonControl size (tile_look Clear)
        [ ControlResize   (\_ _ {w,h} -> {w=w/3, h=h/3})
        , ControlFunction (noLS (tileF tid c))
        , ControlId       tid
        : atts
        ]
```

The customized look is defined by the function `tile_look`. It is parameterized with the value of the cell at coordinate `c` of the current `board`. The look function

performs rendering operations on an abstract canvas 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=rect}
  = seq [unfill rect, cell, setPenColour LightGrey, draw rect, setPenColour Black]
where
  {x,y}     = rect.corner1
  {w,h}     = rectangleSize rect
  linewidth = min (w/5) (h/5)
  (mw,mh)   = (w/2, h/2)
  cell      = case tile of
                Clear    = id
                Filled t = seq [ setPenSize linewidth
                               , if (t == Tic) cross nought
                               , setPenSize 1
                               ]
  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 thus customized tile button control its callback function tileF is evaluated:

```
tileF :: Id Coordinate (PSt GameSt) -> PSt GameSt
tileF tid c pSt=:{ls=gameSt=:{game=game=:{board,names,turn},ids},io}
# io     = disableControl tid io
# io     = setControlLook tid True (True,tile_look (Filled turn)) io
# io     = setControlText ids.nameId (name_of names (~turn)) io
# game   = {  game & board = add_cell c turn board, turn = ~turn}
# gameSt = {gameSt & game = game}
# io     = switch_turn gameSt io
# pSt    = {pSt & ls = gameSt, io = io}
= check_game_over pSt
```

This function is a typical callback function because it must account for all possible effects that result from the player pressing this button. These effects are to disable the control (line 3) in order to disallow the player to press it again; it alters the current look of the control itself (line 4) to show the new content of the cell; it alters the displayed name of the current player to the other player (line 5), and it stores the updated tic-tac-toe board and turn in the game state (line 6 and 7). Finally, the next player is told with which element she is playing (line 8). These values are stored in the state of the entire interactive process (line 9), and it is checked if the game is over (line 10).

The final part that is to be discussed concerns termination of the game. The function itself is straightforward:

```
check_game_over :: (PSt GameSt) -> PSt GameSt
check_game_over pSt=:{ls={game={board,names,turn}}}
| game_over board = openNotice notice pSt
```

```
| otherwise = pSt                                                                    4
where                                                                                5
  won      = winner board                                                            6
  notice   = Notice (accolade ++ ["","Do␣you␣want␣to␣play␣another␣game?"])          7
               (NoticeButton "Yes" (noLS new_game))                                  8
               [NoticeButton "No"  (noLS closeProcess)]                              9
  accolade = if (isNothing won)                                                      10
               ["It␣is␣a␣draw."]                                                     11
               ["Congratulations,␣" +++ name_of names (~turn) +++ "!","You␣won."]    12
```

Whenever the game happens to be over, a notice (Fig. 1 (left-bottom)) is opened to congratulate the player. If the player presses the "No" button, then the entire program is stopped (line 9). If she decides to play a new game, then the `new_game` callback function takes care of the remaining details:

```
new_game :: (PSt GameSt) -> PSt GameSt                                               1
new_game pSt=:{ls=gameSt=:{ids={tileIds}}}                                           2
# pSt          = getPlayerNames pSt                                                  3
# (gameSt,pSt) = accPLoc (\gameSt -> let new = {gameSt & board = emptyBoard          4
                                                       , turn  = Tic                 5
                                               } in (new,new)) pSt                   6
= appPIO ( switch_turn     gameSt                                                    7
         o setControlLooks [(tid,True,(True,tile_look Clear)) \\ tid <- tileIds]     8
         o enableControls  tileIds                                                   9
         ) pSt                                                                       10
```

The players can change their names, the board is cleared, and the first player starts (lines 3-5). This information is 'synced' with the user interface (lines 7-10).

The complete Object I/O specification requires 177 lines of code. Even though it is by no means a large program, it still demonstrates that it fails to be a concise specification. We need to find better abstractions.

4 Tic-tac-toe in Racket's big-bang

In this section we present the tic-tac-toe case using Racket's `big-bang` approach. Because we want to have an equal treatment of the approaches, we render the required Racket API in Clean. The `big-bang` approach of Racket is part of its `Universe` which was designed *"to reconciling I/O with purely functional programming, especially for a pedagogical setting"* [25]. This results in an emphasis on designing a proper state model that reflects the stages an interactive program passes through. Rendering is handled completely by a single pure function that maps a state to an image. State transitions are triggered by the user via the keyboard or the mouse. Time can also be a source of state transitions. Termination is handled via a predicate that tests the current state value, again a pure function. From this account it follows that a state should be designed first:

```
:: GameSt = EnterNames Players TicTac | Play Game | Accolades Game | Stop          1
                                                                                    2
initGameSt :: Players -> GameSt                                                     3
initGameSt players = EnterNames players Tic                                         4
```

This state model describes the subsequent stages of tic-tac-toe. First, the two players can enter their names. Second, the game is played. Third, the appropriate person receives her accolades. Fourth, the players can decide to start over again or terminate the application.

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

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

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`).

```
class render a :: Size a -> Image
```

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:

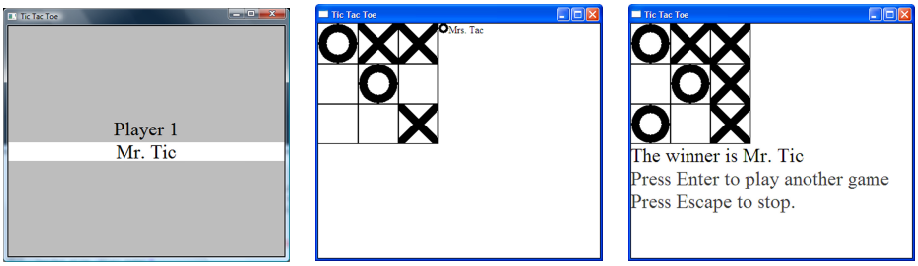

Fig. 2. The main screens in tic-tac-toe: view and enter player names (left), the game (middle), and congratulate winner (right).

```
instance render GameSt where                                                1
  render size (EnterNames players turn)                                     2
```

```
  = overlay [above [text ("Player␣" <+ if (turn==Tic) 1 2) 24 Black
                   ,overlay [text (name_of players turn)   24 Black
                            ,rectangle size.w 30 Solid White
                            ]
                   ]
             ,rectangle size.w size.h Solid LightGrey
             ]
render size (Play game)
  = beside_align TopY
                 [render size game.board
                 ,render {w=16,h=16} game.turn
                 ,text (name_of game.names game.turn) 12 Black
                 ]
render size (Accolades game)
  = above_align LeftX
                [render size game.board
                ,case winner game.board of
                   Just t  = text ("The␣winner␣is␣" <+ name_of game.names t)
                                                          24 Black
                   nothing = text "It's␣a␣draw..."       24 Black
                ,text "Press␣Enter␣to␣play␣another␣game" 24 DarkGrey
                ,text "Press␣Escape␣to␣stop."            24 DarkGrey
                ]
render size Stop
  = 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 2-9). When playing the game, the tic-tac-toe board, the player glyph and name are displayed beside each other (lines 10-15). When receiving the accolades, the final board is shown above the name of the winner, if any, and information how to proceed (lines 16-25). The final state is rendered as the empty image (lines 26 and 27).

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

```
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.

```
keys :: GameSt KeyEvent -> GameSt
keys (EnterNames players turn) key
| key == "\\r"        = if (turn==Tic) (EnterNames players Tac)
                                       (Play (initGame players))
| key == "\\b"        = EnterNames (alterName initStr players turn) turn
| key == "␣" || alpha = EnterNames (alterName (flip (+++) key) players turn) turn
| otherwise           = EnterNames players turn
where alpha           = isAlpha (hd (fromString key))
keys (Accolades game) key
| key == "\\r"        = EnterNames game.names Tic
| key == "escape"     = Stop
keys state _          = state

alterName :: (String -> String) Players TicTac -> Players
alterName f players Tic  = {players & tic = f players.tic}
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.

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

The definition clearly states that only if the mouse is pressed in a free cell of the current tic-tac-toe board (line 3) that the board gets updated and the next player can proceed (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).

The entire `big-bang` style specification consists of 85 lines, which is a reduction of 52% of the Object I/O version. The profit is gained mainly due to the automatic syncing of the program state with its rendering. It comes at a price though: we must implement GUI elements such as text-edit boxes and buttons ourselves. We need to find an abstraction in which this price is not paid.

5 Tic-Tac-Toe in Task Oriented Programming

In this section we present the TOP iTask approach to specify tic-tac-toe. In the TOP approach work is analyzed in terms of *tasks* that must be performed in order to get the work done. Another key property of TOP is the strong coupling of data and visualization: instead of keeping track of data and controlling the way it is rendered and altered, in TOP you are usually only concerned with maintaining the data. Just like in the other cases, we start with modeling the game state which is identical to the `Game` record defined in Sect. 2:

```
:: GameSt :== Game
```

The `GameSt` is shared by all components to keep track of the game logic. This is specified by means of the `withShared` combinator:

```
play_tictactoe :: Task Void                                                          1
play_tictactoe                                                                       2
  = withShared (initGame {tic="Mr.␣Tic",tac="Mrs.␣Tac"})                             3
   (\sharedGameSt                                                                    4
       -> viewSharedInformation "Tic␣Tac␣Toe" [ ViewWith show_board ] sharedGameSt   5
             ||-                                                                     6
         (new_names sharedGameSt >>| play sharedGameSt)                              7
   )                                                                                 8
```

At the top-level, the components are a *view task* on the current value of the shared game state (line 5) and, in parallel (line 6), the task that coordinates the player actions (line 7). The view task merely expresses that at all times the current value of the game state is rendered as specified by the `show_board` function. We discuss this function after the task that coordinates the player actions. This task consists of two other tasks that are performed in sequence: first, the `new_names` task allows the players to alter their names, and second, the `play` task controls one tic-tac-toe game.

In contrast with a *view task*, the `new_names` task uses an *update task* that not only views a data model, but also provides the user with means to alter it in a type-safe way (Fig. 3 (left-top)).

```
new_names :: ((Shared GameSt) -> Task GameSt)                                        1
new_names = updateSharedInformation                                                  2
              "Enter␣names" [ UpdateWith (\{names} -> names) (const initGame) ]      3
```

In this case, the users can only view and update their `names`. Any update results in a new initial game state.

The `play` task is liberated of keeping the rendering of the game state in sync with the actual game state value.

```
play :: (Shared GameSt) -> Task Void                                                 1
play sharedGameSt                                                                    2
  = watch sharedGameSt                                                               3
    >>* [ WhenValid (\gameSt -> game_over gameSt.board)                              4
                    (\gameSt -> viewInformation "The␣winner␣is:" []                  5
                                    (name_winner gameSt)                             6
```

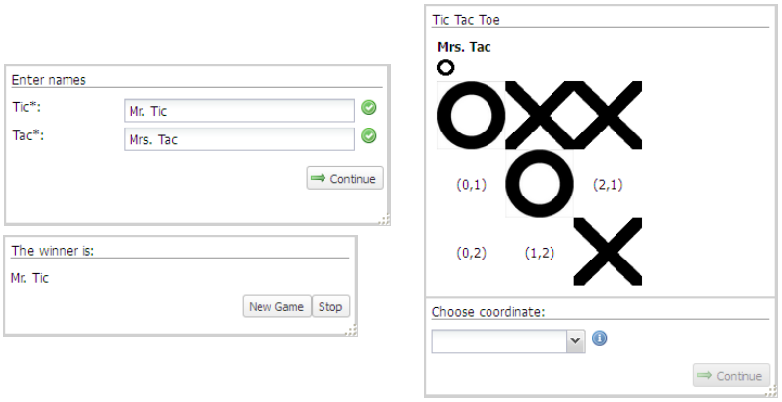

Fig. 3. The main tasks in tic-tac-toe: view and enter player names (left-top), play game (right), and congratulate winner (left-bottom).

```
                                >>* [ Always (Action "New Game" [])                    7
                                      (   new_names sharedGameSt                       8
                                      >>| play      sharedGameSt                       9
                                      )                                                10
                                    , Always (Action "Stop" []) (return Void)          11
                                    ]                                                  12
                    )                                                                  13
        , AnyTime ActionNext (make_a_move sharedGameSt)                                14
        ]                                                                              15
where                                                                                  16
  name_winner {board,names}                                                            17
    = if (isNothing (winner board)) "nobody"                                           18
                                    (name_of names (fromJust (winner board)))          19
```

It keeps inspecting the current game state value (line 3). Whenever it is detected that the game is over (line 4), the winner is displayed, again using a view task (lines 5 and 6). The players are offered the two options to either start a new game (lines 7-10) or stop (line 11) (see Fig. 3 (left-bottom)). The current player can decide to make a move (line 14). This task is specified as follows:

```
make_a_move :: (Shared GameSt) (Maybe GameSt) -> Task Void                             1
make_a_move sharedGameSt (Just gameSt=:{board,turn})                                   2
  =             enterChoice "Choose coordinate:"                                       3
                            [ChooseWith ChooseFromComboBox toString]                   4
                            (free_coordinates board)                                   5
    >>= \new -> set {gameSt & board = add_cell new turn board, turn = ~turn}           6
                    sharedGameSt                                                       7
    >>| play sharedGameSt                                                              8
```

The current player chooses one of the currently free cells in the board (lines 3-5). For this purpose the `enterChoice` task is used which creates a user interface with

which one value from a list of values can be chosen. This selected value, `new`, is used to update the board, as well as the next player (lines 6-7).

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

```
show_board :: GameSt -> [HtmlTag]                                                1
show_board {board,names,turn}                                                    2
  = [H3Tag [] [Text (name_of names turn)], TileTag (16,16) turn, tictactoe]      3
where                                                                            4
  tictactoe = TableTag [BorderAttr "0"]                                          5
                       [ TrTag [] [ cell {col=x,row=y} \\ x <- [0..2] ]          6
                       \\ y <- [0..2]                                            7
                       ]                                                         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)                               15
                         ]                                                       16
```

The rendering displays the current player, the glyph she is playing with, and the board (line 3). Instead of programming the rendering, image files are used (`"Tic.png"` and `"Tac.png"`).

In order to make this specification complete the generic machinery must be invoked to keep the used data models in sync with their rendering:

```
derive class iTask Game, Coordinate, Tile, TicTac, Players                       1
instance toString Coordinate                                                     2
where    toString {col,row} = "(" <+ col <+ "," <+ row <+ ")"                    3
```

The TOP specification is 57 lines, which is a reduction of 68% and 33% compared with the Object I/O and Racket `big-bang` versions respectively.

6 Comparison

In this section we compare and discuss the versions of the tic-tac-toe case study. We start with the specification of the GUI. 56% of the Object I/O specification defines the modal dialogs, the main window and its components. This immediately explains why the Racket `big-bang` version is proportionally shorter: it already implements the infrastructure for a single-document application. Both in Object I/O 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 TOP version is proportionally shorter than the other two versions: only the game playing screen is rendered explicitly, and the other screens 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. The TOP iTask version fails to implement direct manipulation of the board in which the players can click the tiles to have them filled with their glyph. Instead it resorts to provide a choice between the available tiles which results in a less intuitive and somewhat disconnected user experience.

If we compare the rendering of the game then we see that both the Object I/O and Racket `big-bang` versions use pure functions (`tile_look` and the `render` functions respectively). However, in Object I/O 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 TOP 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.

To understand the operational behavior of the Object I/O version, the callback functions and their effect on the logical state and GUI state need to be analyzed. The Object I/O 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. 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 Object I/O 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 Object I/O 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 Object I/O we must unravel the rendering and logical operations and keep track of their effects on both the GUI state and logical state. In Racket `big-bang` it is clear that we only need to study the mouse and keyboard handler. This amounts to discovering the underlying logical state machine and its transitions. Because TOP is all about tasks and their evaluation order, understanding the behavior of the application requires the least effort.

7 Related Work – draft

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 [32]. In a follow-up paper [33] 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 widget-based GUI library Haggis [9] uses a similar compositional approach and extends it to build the entire GUI

of an application. 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 [34] and spawned a number of related approaches which are enumerated elsewhere [23]. It is interesting to compare their approach to those described in this paper.

8 Conclusions

In this paper we have presented a case study of a GUI application, tic-tac-toe, expressed in three different formalisms: Object I/O, 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 76 lines. None of the approaches result in large specifications: the largest, Object I/O, is 177 lines. Their relative sizes vary greatly: in comparison with the Object I/O version (100%) the size of the Racket `big-bang` version is 48% and the TOP iTask version is 32%. 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, the line count gives an indication of the conciseness of the formalism.

In comparison with Object I/O, 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.

Of the three versions, the application behavior is hardest to distill in the Object I/O version because the callback functions need to concern themselves with the details of manipulating the shared GUI state 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. The disadvantage is that the application 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 synchronisation of the application state with respect to its rendered GUI.

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. Finne, S., Peyton Jones, S.: Composing Haggis. In: Eurographics Workshop on Programming Paradigms in Graphics, Maastricht, the Netherlands, Springer (1995) 85–101
10. Achten, P., Plasmeijer, R.: The ins and outs of Concurrent Clean I/O. Journal of Functional Programming **5**(1) (1995) 81–110
11. 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
12. 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
13. 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
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. 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.
17. 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)
18. Elsman, M., Friis Larsen, K.: Typing XHTML web applications in ML. In: Proceedings of the 6th International Symposium on the Practical Aspects of Declarative Programming, PADL '04. Volume 3057 of Lecture Notes in Computer Science., Dallas, TX, USA, Springer-Verlag (June 2004) 224–238
19. 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)
20. 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

21. 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
22. 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)
23. 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
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. Achten, P., van Eekelen, M., Plasmeijer, R.: Generic graphical user interfaces. In Michaelson, G., Trinder, P., eds.: Revised Papers of the 15th International Workshop on the Implementation of Functional Languages, IFL '03. Volume 3145 of LNCS., Edinburgh, UK, Springer-Verlag (8-10, September 2004) 152–167
29. Plasmeijer, R., Achten, P.: iData for the world wide web - Programming interconnected web forms. In: Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS '06. Volume 3945 of LNCS., Fuji Susone, Japan, Springer Verlag (24-26, April 2006) 242–258
30. 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
31. Achten, P., Wierich, M.: A tutorial to the Clean Object I/O library (version 1.2). Technical report CSI-R0003, Radboud University Nijmegen (2000)
32. 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
33. Henderson, P.: Functional geometry. Higher-Order and Symbolic Computation **15** (2002) 349–365
34. 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