Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics

Peter Achten[1] Jurriën Stutterheim[1] László Domoszlai[1,2] Rinus Plasmeijer[1]

[1]Radboud University Nijmegen, Netherlands, ICIS, MBSD
[2]Eötvös Loránd University, Budapest, Hungary, Software Technology Department
P.Achten@cs.ru.nl, j.stutterheim@cs.ru.nl, l.domoszlai@science.ru.nl, rinus@cs.ru.nl

Abstract

iTasks enables the rapid creation of multi-user web-applications by automatically generating form-based graphical user interfaces (GUIs) for any first-order type. In some situations, however, form-based GUIs are not sufficient or do not even make sense. We introduce a *purely compositional* library for creating interactive user interface components, based on Scalable Vector Graphics (SVG). Not only are all images purely compositional, interaction on them is specified by *pure functions*. The graphics library is integrated with iTasks in such a way that one can easily switch between the generic form-like GUIs and graphics-based user interfaces. Still, a large part of the library is fully iTasks-agnostic and can therefore be used in other contexts as well. We demonstrate the capabilities of this library by implementing the multi-player Ligretto card game in iTasks. This is an interesting case study because it requires a good answer to the challenges of defining multi-user, distributed applications with appealing graphics.

Categories and Subject Descriptors D.1.1, I.3.6 [*Applicative (Functional) Programming, Methodology and Techniques*]: Graphics data structures and data types, Interaction techniques, Languages

Keywords Compositional Graphics, Interactive Graphics, Pure Interaction Model, Scalable Vector Graphics, Image DSL, Task-Oriented Programming, iTasks

1. Introduction

The iTask system [20, 22] (iTasks) is an implementation of the Task Oriented Programming (TOP) paradigm in the strongly typed, lazy, purely functional programming language Clean [21]. The TOP paradigm has been designed to support the development of distributed, multi-user web applications in which humans and software systems collaborate. iTasks offers a client-server infrastructure for the coordination of the tasks being defined, where typically multiple people work closely together on the Internet, making use of standard browsers. Types play a central role in iTasks: from any first-order type, a form-like graphical user interface (GUI) is *generated* automatically. To do this successfully, it is vital that these interfaces are *purely compositional*, i.e.: the meaning of an interface is determined exclusively by its sub-components and their composition. This design principle can be traced back to Henderson's Functional Geometry [16], and indeed, the form-like GUIs generated by iTasks adhere to this property.

For many application domains, such as status displays or games, communicating information via form-like GUIs is not informative enough, or simply not appropriate. In these cases, it is better to use *dynamically adjustable interactive graphics*. Several libraries already exist that allow a programmer to create interactive graphics using JavaScript and HTML 5. However, all libraries that we have encountered impose a hidden state model on their API, e.g., by using some kind of single-canvas-abstraction, having attribute-setting operations, using canvas-wide transformations, and so on. Put in other words, they are not purely compositional. Lack of compositionality places the burden on the programmer to find out in which order the graphics operations need to be performed to create the desired images. A compositional image library would shift this problem from the programmer to the library author.

For example, for the communication with domain experts, we are currently developing Tonic [23]. It automatically *generates* a kind of task flow-chart at compile-time, called a blueprint, that displays an iTasks program's static task structure. Blueprints are augmented with concrete information at run-time to show which concrete tasks have been created, who is working on what, what progress has been made, how tasks are related to each other, etc. Generating images requires compositionality, since their sizes are generally not known beforehand. The lack of a compositional graphics library has hampered the development of this tool in such a way that we decided to design a new graphics library which *is* compositional. In the implementation we have to compensate for the lack of compositionality in the underlying libraries.

There are many real-world use-cases that can profit from compositional images. One such use-case is found in the naval domain. Modern ships include interactive plotting-boards that schematically display the ship's layout. These boards are dynamically updated when, e.g., calamities arise, such as fire or leaks. These same boards can then be used interactively to coordinate calamity mitigation efforts. At the same time, graphs and dials may indicate the fire's heat developments or a leak's water levels. We anticipate that using a compositional graphics library reduces the development time of these plotting-boards, and similar systems, significantly.

Being able to draw images in a compositional way solves the drawing problem, but we also need to be able to deal with interaction. Fortunately, this is what iTasks is designed for. In this paper, we introduce the `Graphics.Scalable` library, with which one can create custom vector-based images in a purely compositional way. We integrate this library seamlessly in the TOP concept of interactive

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IFL '2014, Oct 1–3, 2014, Boston, MA, USA.
Copyright © 2014 ACM 978-1-4503-3284-2/14/10. . . $15.00.
http://dx.doi.org/10.1145/2746325.2746329

editor tasks in order to make images interactive, using only pure functions.

The integration with iTasks turns out to be mutually beneficial. The image library profits because interaction can be specified as pure functions on model data types within editor tasks, and it can rely on the existence of task combinators to specify application behaviour. This greatly simplifies the API of the image library. Vice versa, iTasks profits because the appearance and behaviour of editor tasks can be customized to meet the needs of applications that require custom user interfaces.

A real-world use-case, which we address specifically in this paper, concerns multi-player, distributed games such as *Trax* [3] and *Ligretto*. We demonstrate how the latter card game can be created with the `Graphics.Scalable` image library and iTasks.

In this paper we make the following contributions:

- We present a *purely compositional* image API, implemented by the `Graphics.Scalable` library.
- We define interaction on images using *pure functions*.
- We integrate the library in iTasks, reusing the existing editor infrastructure.
- We demonstrate its usage by a case study: the Ligretto game.
- We map `Graphics.Scalable` images to the *Scalable Vector Graphics* (SVG) standard [8].
- We overcome the technical challenges imposed by the Internet's client/server architecture using *editlets*.

We start our explanation by first concentrating on *static, purely compositional images* as provided in `Graphics.Scalable` in Section 2. We show how to render the state of the card game *Ligretto*. This is a non-trivial show-case of compositional rendering (you are invited to browse ahead to Figure 1(**l**)). We show how static images are made *interactive* in Section 3 and turn the example into a full-fledged, multi-user application. The underlying technology of the `Graphics.Scalable` library is SVG. Mapping to SVG has proven to be challenging mostly because SVG adopts a single-canvas rendering model which conflicts with the purely compositional nature of `Graphics.Scalable`. The implementation is presented in Section 4.

Functional programming and creating images, whether they are interactive or compositional or both, share a long research history. The `Graphics.Scalable` API is greatly influenced by old and recent research. In Section 5 we discuss this in more detail. The combination of the `Graphics.Scalable` image library and TOP is a novel contribution to the field of programming interactive applications in a functional style. We conclude in Section 6.

2. Compositional Static Images

In this section we describe the compositional image library (Sections 2.1–2.6). The concepts are illustrated step by step by rendering the entire state of the Ligretto card game (Section 2.7).

2.1 Image concepts

Conceptually, an *image* is an infinitely large, perfectly transparant 'slide' that renders a value of some model type m. This is captured with the opaque type `Image m`. The 'slide' can be scaled, rotated, and skewed. There is no *global* coordinate system. When defining an image we impose a *local* coordinate system, the *span box*. The span box consists of two dimensions: the *x-span* increases from 'left' to 'right' (perfectly horizontal) and the *y-span* increases from 'above' to 'below' (perfectly vertical). The unit of measure is *pixel*, expressed with *real* values. Pixels get a physical interpretation only when the image is actually rendered on a device. This is natural in the context of scalable vector graphics. It is important to note right away that the span box is not the same as the common *bounding box* concept. The bounding box of an image is identified by the minimum and maximum coordinates of its visual content. In contrast, the span box of an image defines its conceptual size that is used for layout. We deliberately allow visual content to exist outside of the span box or within a 'tighter' bounding box. These design decisions seem to be minor, but they are not: what an image looks like should be unconnected with where it happens to be and what its size is.

Stacking 'slides' is the only way to compose new images from simpler ones. Conceptually, stacking creates a z-axis that is oriented perfectly towards the viewer. 'Higher' images can obscure 'lower' images, depending on their opacity or masking attribute (Section 2.3). We literally create a collage. The span boxes of the images are used to specify their relative positions. For that purpose layout combinators are used (Section 2.4). Note that in the presence of infinitely large images, a translation transformation does not change the image, hence our library does not support image translation. All we need to care about are the relative positions of images.

2.2 Basic images

The image library supports common shapes as *basic images*:

```
:: Span        // an opaque data type

px             :: Real -> Span     // (px x) represents x pixels

empty          :: Span Span        -> Image m
circle         :: Span             -> Image m // Circle by diameter
ellipse        :: Span Span        -> Image m
rect           :: Span Span        -> Image m
xline          :: Span             -> Image m // Lines are 1-dimensional
yline          :: Span             -> Image m // Lines are 1-dimensional
line           :: Slash Span Span -> Image m
text           :: FontDef String   -> Image m
normalFontDef :: String Real      -> FontDef

:: Slash      = Slash | Backslash
```

A number of aspects are worth noting. The `empty` image has no visual content and only an x-span and a y-span. What a piece of text looks like is determined by the used *font* as well as the *content*, hence both must be part of its specification. The `FontDef` structure collects all SVG font properties, such as font-size, font-weight, font-style, etcetera. The convenience function (`normalFontDef` *name h*) captures the frequently occuring situation that it suffices to specify the font family *name* and font height in pixels (also the y-span), setting all other font properties to `"normal"`. The x-span of the `text` image depends on the used font and text. The default renderings of the `circle`, `ellipse`, and `rect` shapes is the same as the default rendering of text, i.e.: using a stroke of one pixel and filled with the default color black. These can be changed with the image attributes (Section 2.3). Finally, lines are also drawn with a default stroke of one pixel and use the color black. In the presence of rotation a single line primitive is sufficient, but for convenience we provide primitives for horizontal, vertical, and 'tilted' lines (`xline`, `yline`, `line`). The `Slash` parameter identifies the imaginary rectangle corner points that are 'connected' by the line (`Slash`, `/`, left-bottom to right-top corner and `Backslash`, `\`, left-top to right-bottom corner).

2.3 Image attributes

Image attributes alter the appearence of visual elements *without* altering the span box. In this way, the purpose of the span box does not get mixed with the appearence of an image. In SVG, attributes are defined with *name-value* pairs. We adopt the SVG names:

```
:: StrokeAttr      m = { stroke      :: SVGColor }
:: StrokeWidthAttr m = { strokewidth :: Span     }
```

```
:: XRadiusAttr     m = { xradius     :: Span     }
:: YRadiusAttr     m = { yradius     :: Span     }
:: FillAttr        m = { fill        :: SVGColor }
:: OpacityAttr     m = { opacity     :: Real     }
:: DashAttr        m = { dash        :: [Int]    }
:: MaskAttr        m = { mask        :: Image m  }
```

Each type constructor is made an instance of a type constructor class `tuneImage`, having trivially derived operators and function.

```
class tuneImage attr :: (Image m) (attr m) -> Image m
(>@>) infixr 2 :: (attr m) (Image m) -> Image m | tuneImage attr
(<@<) infixl 2 :: (Image m) (attr m) -> Image m | tuneImage attr
tuneIf    :: Bool (Image m) (attr m) -> Image m | tuneImage attr
```

For the specification of colors we adopt the extensive set of SVG color names and the common RGB-triplets:

```
class toSVGColor a :: a -> SVGColor
instance toSVGColor String, RGB

:: RGB = { r :: Int, g :: Int, b :: Int }
```

2.4 Image composition

Images are composed by stacking. The images that are to be stacked are given in a *finite* list. Elements with lower list-index positions can be obscured by elements with higher list-index positions. This leaves only the relative layout along the x-axis and y-axis unspecified. This relative layout can be defined with or without a *host image*. A host image serves two purposes: its span box is the local coordinate system in which the positions of the stacked images are specified, and it is the background image on top of which these images are stacked. If no host image is used, then the span box equals the bounding box of the span boxes of the stacked images. Offsets are defined as a pair of an x-span and y-span value. The initial layout of images is *always* computed without the offsets. The final layout is obtained by adding the i-th offset to the initial position of the i-th image.

```
:: Layout m :== [ImageOffset] -> [Image m] -> (Host m) -> Image m
:: Host   m :== Maybe (Image m)
:: ImageOffset :== (Span, Span)
```

The image list must be finite. In the image layout functions, any other list argument need not have the same length. If they are too short, then padding values are defined for them (for offsets, this is *zero*). If they are too long, then the surplus is not evaluated. In this way we can keep the specification of the image list separate from other concerns such as offsets and alignments in the other image layout functions. It also avoids cluttering of the image list specifications.

Conceptually, the image library has only one *core* image layout function[1]:

```
collage :: Layout m
```

In a `collage`, the images are initially stacked with their left-top span box corners aligned. The final position of the i-th image is obtained by adding the i-th offset to that initial position.

Derived image layout functions are `overlay`, `grid`, `above`, `below`, and `margin`. The first of them, `overlay`, adds horizontal and vertical alignment options to the layout specification:

```
overlay        :: [ImageAlign] -> Layout m

:: ImageAlign :== (XAlign, YAlign)
:: XAlign      = AtLeft | AtMiddleX | AtRight
:: YAlign      = AtTop  | AtMiddleY | AtBottom
```

[1] Although internally, other layout combinators are modeled explicitly as well for reasons of efficiency.

In an `overlay`, the initial position of the images is determined using the list of alignments: the position of the i-th image is determined by the i-th alignment value. The final position of the i-th image is obtained by adding the i-th offset value to the i-th initial position.

Images often need to be placed in a grid-like structure:

```
:: GridDimension = Rows Int    | Columns Int
:: GridMajor     = ColumnMajor | RowMajor
:: GridXLayout   = LeftToRight | RightToLeft
:: GridYLayout   = TopToBottom | BottomToTop
:: GridLayout  :== (GridMajor, GridXLayout, GridYLayout)

grid :: GridDimension GridLayout [ImageAlign] -> Layout m
```

A grid's dimensions are specified by providing either a number of rows or a number of columns. The number of images then determines the corresponding number of columns or rows. The grid can be populated in eight different ways, determined by the grid layout: column-by-column or row-by-row (`GridMajor`), in combination with left-to-right or right-to-left (`GridXLayout`), in combination with top-to-bottom or bottom-to-top (`GridYLayout`). The span boxes and alignments of the images are used to compute the images' initial positions, which are then fine-tuned with the corresponding offsets to obtain all final positions.

Images are often placed beside or above each other:

```
beside :: [YAlign] -> Layout m
above  :: [XAlign] -> Layout m
```

These are immediately derived from the `grid` image layout function: `beside` is one row of left-aligned images and `above` is one column of top-aligned images.

Finally, it is useful to add margins around an image. This merely increments the span box but does not alter the image. We follow the convention of SVG to specify margins in several ways:

```
class margin a :: a (Image m) -> Image m
instance margin Span,
               (Span, Span),
               (Span, Span, Span),
               (Span, Span, Span, Span)
```

The 'one-span' instance a imposes a uniform margin a around the image, the 'two-span' instance (a, b) imposes margin a above/below and b left/right of the image, the 'three-span' instance (a, b, c) imposes margin a above, b left/right, c below the image, and the 'four-span' instance (a, b, c, d) imposes margin a above, b right, c below, and d left of the image.

2.5 Symbolic span expressions

The image layout functions need to manipulate span values symbolically in order to compute the desired image positions. Examples of symbolic span values are *text width*, *image width* and *height*, *column width*, and *row height*. Examples of symbolic span computations are the usual arithmetical operations as well as negating the value and taking the absolute value and determining the minimum and maximum span value. These are covered by the following span-definitions and instances of arithmetic operations:

```
:: ImageTag

// Symbolic span values:
textxspan  :: FontDef String -> Span // text width
imagexspan :: ImageTag       -> Span // image width
imageyspan :: ImageTag       -> Span // image height
columnspan :: ImageTag Int   -> Span // column width
rowspan    :: ImageTag Int   -> Span // row height

// Symbolic span arithmetic:
instance zero Span
instance +    Span
```

```
instance -    Span
instance ~    Span
instance abs  Span

class (*.) infixl 7 a :: a n -> a | toReal n
class (/.) infixl 7 a :: a n -> a | toReal n
instance *. Span, Real, Int
instance /. Span, Real, Int

minSpan :: [Span] -> Span
maxSpan :: [Span] -> Span
```

The opaque type `ImageTag` refers to an image. In case of `imagexspan` and `imageyspan`, this can be any image; in case of `columnspan` and `rowspan`, the image tag needs to be associated to a `grid` image. The number argument of the latter two functions identifies the column or row number, starting at index zero. If the image tag does not happen to refer to an image, then the symbolic span value is `zero`.

Image tags must identify an image uniquely. This is guaranteed by taking advantage of Clean's *uniqueness type system*. The image author has no means to define `ImageTag` values herself. Instead, the top-level image rendering function is provided with an infinite list of fresh image tag values. These image tag values come in pairs: the first is a *non-uniquely attributed* image tag (of type `ImageTag`) and the second is a *uniquely attributed* image tag (of type `*ImageTag`). To identify an image, the image author is forced to use the uniquely attributed image tag:

```
tag :: *ImageTag (Image m) -> Image m
```

In this way, it is statically guaranteed that an image tag is associated with an image at most once. Even if the tagged image is used several times, it is guaranteed that the tag identifies the very same image. Hence, the corresponding symbolic span values have the same size.

The types of the arithmetic operations should reflect the 'physical' dimension. Span values can be added and subtracted, and their absolute and negated value can be computed. These operators do not alter the dimension, so they can be defined using ordinary operator overloading (`+`, `-`, `abs`, and `~`). For other operators this is not true: multiplication of span results in square span, division of span results in a scalar value, and comparison of span values evaluates to a boolean. For this reason the image library supports slightly different overloaded operators for these purposes: `*.` and `/.` for multiplication and division with a scalar value, and `minSpan` and `maxSpan` for determining the smallest and largest span from a list of span values. The experiments that we have conducted so far indicate that the lack of comparison operators on span values does not limit the expressiveness of symbolic span expressions.

Finally, the symbolic span expression language in combination with the `collage` image layout function is sufficiently expressive to derive all other image layout functions (shown in Figure 6). This expressive power is also available for the image author who can use the same language to define new image layout patterns herself.

2.6 Image transformations

Any (composite) image can be subject to transformation:

```
rotate :: Angle     (Image m) -> Image m
skewx  :: Angle     (Image m) -> Image m
skewy  :: Angle     (Image m) -> Image m
fit    :: Span Span (Image m) -> Image m
fitx   :: Span      (Image m) -> Image m
fity   :: Span      (Image m) -> Image m
flipx  ::           (Image m) -> Image m
flipy  ::           (Image m) -> Image m

:: Angle
rad    :: Real -> Angle
deg    :: Real -> Angle
```

Angles are expressed as radians or as degrees. In general, the span box of a rotated or skewed image differs from the span box of the original image. Non-proportional scaling is done with `fit` which ensures that the resulting image has exactly the specified x-span and y-span. Proportional scaling is done with `fitx` and `fity`: they ensure exact x-span and y-span, respectively and scale the other span proportionally. Flipping, or mirroring, an image around its x- or y-axis is done with `flipx` and `flipy`.

2.7 Case study: rendering the Ligretto state

In this section we demonstrate how to exploit the compositional features of the image library to render the state of a game of Ligretto. We first present the data types that model the game state (Section 2.7.1) and then show how it is rendered (Section 2.7.2).

2.7.1 Ligretto model types

Ligretto is a card game for two, up to twelve players. In this paper we restrict ourselves to a maximum of four players. Each player has forty cards that come in four front colors: *red*, *green*, *blue*, and *yellow*. The ten cards of one color are numbered on the front side from one through ten. For identification purposes, the back sides of the cards have a unique color for each player. These facts can be modeled in a straightforward way:

```
:: Card   = { back :: Color, front :: Color, no :: Int }
:: SideUp = Front | Back
:: Color  = Red | Green | Blue | Yellow
```

At the start of the game, each player shuffles her cards, and places them as follows on the table from right to left (Figure 1(**k**)):

- The *row* cards, which lie beside each other, faced up. The number depends on the number of players (five cards in case of two players, and up to three cards in case of four players).
- The *ligretto* pile, which is a pile of ten cards, faced up.
- The *hand* cards, which is divided in two sub piles: the *concealed* pile which at start are all remaining cards, facing down, and the *discard* pile which come from the concealed pile, facing up.

Finally, there is a shared area for all players, called the *middle* (Figure 1(**j**)). In the middle, piles of cards of the same front color are created by all players at the same time. A new pile must always start with number 1, face up. Cards with a number $n+1$ are allowed to be placed only on a middle pile of the same front color and top-most card having number n. Although players are uniquely identified via their color, we also keep track of their name and render it in the game. These facts are modeled as follows:

```
:: NoOfPlayers :== Int
:: Middle      :== [Pile]
:: Pile        :== [Card]
:: Player        = { color    :: Color
                   , name     :: String
                   , row      :: RowPlayer
                   , ligretto :: Pile
                   , hand     :: Hand
                   , seed     :: Int }
:: RowPlayer   :== [Card]
:: Hand          = { conceal  :: Pile, discard :: Pile }

no_of_cards_in_row :: NoOfPlayers -> Int
colors             :: NoOfPlayers -> [Color]
```

The complete Ligretto game state consists of the middle card piles and the participating players:

```
:: GameSt = { middle :: Middle, players :: [Player] }
```

We can now turn our attention to rendering this game state.

2.7.2 Ligretto rendering

The Ligretto game state is rendered step by step in a compositional way. The individual images are shown in Figure 1.

We start with defining images for cards and attempt to make them look similar to commercially available Ligretto cards. The physical size of these cards is 58.5mm by 90.0mm, so we adopt these values for the rendered cards as well:

```
card_width  = px 58.5
card_height = px 90.0
```

The shape of a Ligretto card is that of a rectangle with rounded corners (Figure 1**(a)**):

```
card_shape = rect card_width card_height
                  <@< {xradius = card_height /. 18}
                  <@< {yradius = card_height /. 18}
```

For rendering the text on cards we use the font family *Verdana* in several sizes:

```
cardfont size = normalFontDef "Verdana" size
```

The model colors need to be mapped to SVG colors that best match the physical cards. We select the following SVG colors:

```
instance toSVGColor Color where
  toSVGColor Red    = toSVGColor "darkred"
  toSVGColor Green  = toSVGColor "darkgreen"
  toSVGColor Blue   = toSVGColor "midnightblue"
  toSVGColor Yellow = toSVGColor "gold"
```

We abbreviate *white* and *black*:

```
white = toSVGColor "white"
black = toSVGColor "black"
```

The number on the front side of a card is displayed in a large font (Figure 1**(b)** shows `big_no 7 Red`):

```
big_no no color = text (cardfont 20.0) (toString no)
                    <@< {fill   = white}
                    <@< {stroke = toSVGColor color}
```

At the back side of the card, the text *Ligretto* is displayed (Figure 1**(c)** shows `ligretto Red`):

```
ligretto  color = text (cardfont 12.0) "Ligretto"
                    <@< {fill   = toSVGColor "none"}
                    <@< {stroke = toSVGColor color}
```

With these image functions, we can render the front side (Figure 1**(d)** or back side (Figure 1**(e)**) of a card:

```
card_image :: SideUp Card -> Image m
card_image side card
| side === Front
  = let no = margin (px 5.0)
            (big_no card.no (no_stroke_color card.front))
    in overlay [(AtMiddleX, AtTop), (AtMiddleX, AtBottom)] []
               [no, rotate (deg 180.0) no] host
| otherwise
  = overlay [(AtMiddleX, AtBottom)] []
            [skewy (deg -20.0) (ligretto card.back)] host
where
  host = Just (card_shape
                <@< {fill = if (side === Front)
                               (toSVGColor card.front)
                               white})
```

The stroke color of the card number depends on the card color:

```
no_stroke_color :: Color -> Color
no_stroke_color Red    = Blue
no_stroke_color Green  = Red
no_stroke_color Blue   = Yellow
no_stroke_color Yellow = Green
```

We introduce an 'empty card' that serves as a visual placeholder for an empty pile (Figure 1**(f)**).

```
no_card_image :: Image m
no_card_image = overlay [(AtMiddleX,AtMiddleY)] []
                        [text (pilefont 12.0) "empty"] host
where
  host = Just (card_shape <@< {fill = toSVGColor "lightgrey"})
```

The simplest way of rendering a pile of cards is to render only the top-most card. However, in this way, players have no visual clue how many cards the pile has. Instead, we display the cards as being stacked on top of the 'empty card' in reversed order and each card having a slightly increased vertical offset (Figure 1**(g)**):

```
pile_of_cards :: SideUp Pile -> Image m
pile_of_cards side pile
  = overlay [] [(zero,card_height /. 18 *. dy) \\ dy <- [0 .. ]]
               (map (card_image side) (reverse pile)) host
where
  host = Just no_card_image
```

For large piles it does not make a lot of sense to show all cards, so instead we show the top-most ten cards (if present) of a pile. For larger piles we include the total number of cards as a small number above the rendered pile (Figure 1**(h)**).

```
pile_image :: SideUp Pile -> Image m
pile_image side pile
| no_of_cards > 10 = above [ AtMiddleX ] []
                           [ text (pilefont 10.0)
                                  (toString no_of_cards)
                           , top_cards_image ]
                           Nothing
| otherwise        = top_cards_image
where
  no_of_cards      = length pile
  top_cards_image  = pile_of_cards side (take 10 pile)
```

We choose to render the player names as a bold faced text on top of a rectangle that is filled with the player's card color. Instead of scaling long or short names, we use masking to prevent long names from running outside of the host image (Figure 1**(i)** shows the result for a player named *alice* playing the red cards).

```
name_image :: Player -> Image m
name_image {name,color}
 = overlay [(AtMiddleX,AtMiddleY)] []
     [ text {cardfont 16.0 & fontweight = "bold"} name
         <@< {fill = if (color === Yellow) black white}
     ] host
     <@< {mask = rect width height <@< {fill   = white}
                                   <@< {stroke = white}}
where
    width  = card_height *. 1.8
    height = card_width  *. 0.4
    host   = Just (rect width height <@< {fill = toSVGColor color})
```

With the above ingredients we are able to render a complete Ligretto game state. The players are 'sitting' at a round table. We arrange the elements as three concentric circular tiers. The innermost tier contains the middle cards, the middle tier shows the player names, and the outermost tier shows the player cards. For this purpose we first create a general function that moves and rotates an arbitrary list of images *imgs* along a circle segment of *a* radians, and the circle having radius *r*:

```
circular :: Span Real [Image m] -> Image m
circular r a imgs
  = overlay (repeat (AtMiddleX,AtMiddleY))
            [  (~r *. cos angle, ~r *. sin angle)
            \\ i     <- [0.0, sign_a .. ]
             , angle <- [i * alpha - 0.5 * pi]]
```

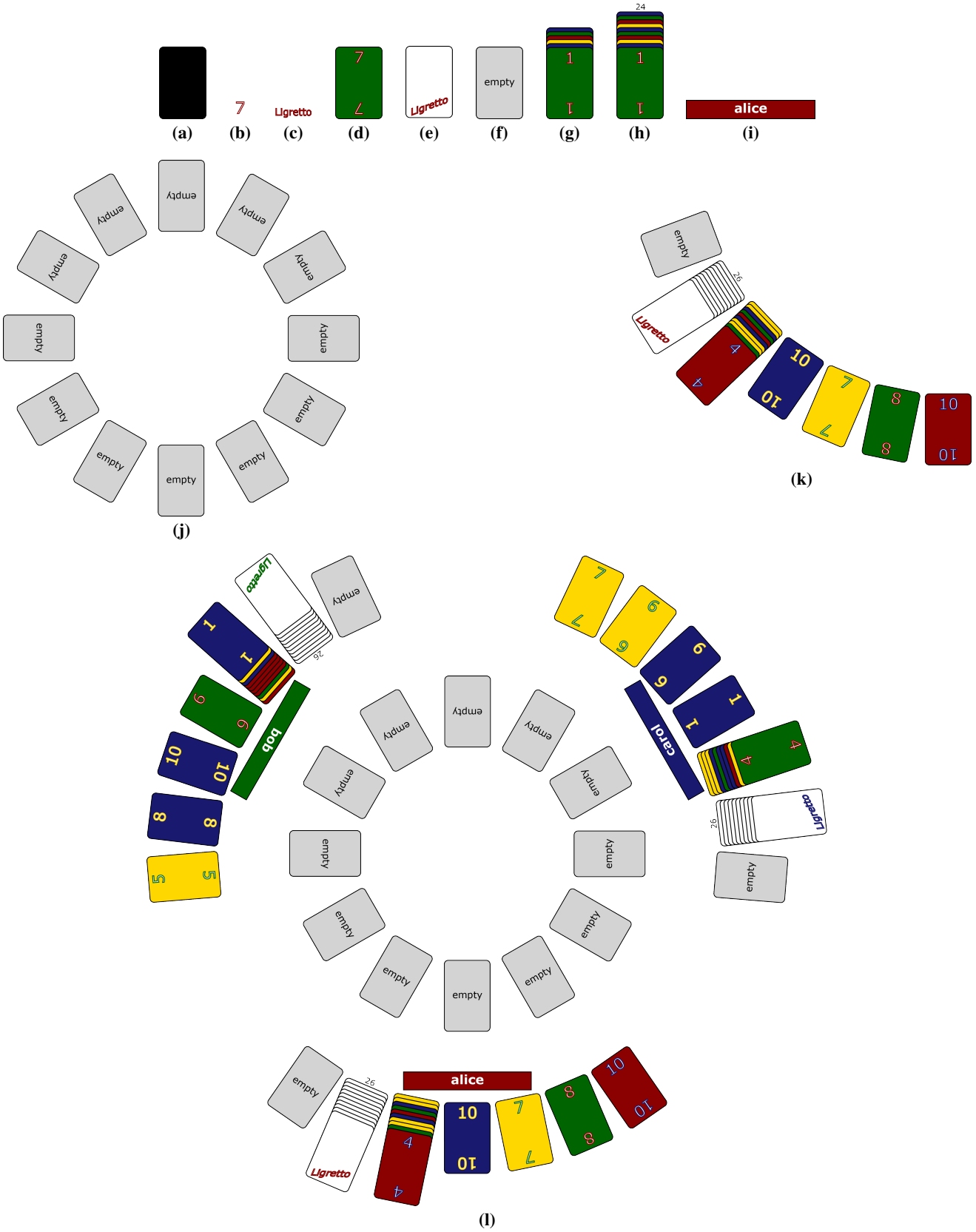

Figure 1. Compositional rendering of the Ligretto game state

```
          [  rotate (rad (i * alpha)) img
          \\ i   <- [0.0, sign_a .. ]
           & img <- imgs]
          (Just (empty (r *. 2) (r *. 2)))
where
  sign_a = toReal (sign a)
  alpha  = toRad (normalize (rad a)) / toReal (length imgs)
```

The circular image is created by stacking all images with their centers (according to their span boxes) aligned. Each image gets placed along the circle segment using the proper offset and gets oriented along that circle segment by rotating the image with the same angle.

The innermost tier, `middle_image`, simply distributes all middle piles along a full circle:

```
middle_image :: Span Middle -> Image m
middle_image r middle
  = circular r (2.0 * pi) (map (pile_image Front) middle)
```

Figure 1**(j)** shows the result of the initial middle for three players, which consists of twelve empty piles, as each player has the potential to start four piles.

The middle tier, `names_image`, distributes all player names along a full circle:

```
names_image :: Span [Player] -> Image m
names_image r players
  = circular r (2.0 * pi) (map name_image players)
```

Before we construct the outermost tier of all players, we first render the cards of a single player. These are either in a pile (the hand and Ligretto piles), or are single cards (the row cards).

```
hand_images :: Hand -> [Image m]
hand_images {conceal, discard}
  = [ pile_image Back conceal, pile_image Front discard ]

row_images :: RowPlayer -> Image m
row_images row = map (card_image Front) row
```

The player cards are placed along a circle segment that is slightly less than a quarter of a circle (Figure 1**(k)**):

```
player_arc = 0.45 * pi

player_image :: Span Player -> Image m
player_image r {row,ligretto,hand}
  = circular r player_arc (  row_images  row
                          ++ [pile_image Front ligretto]
                          ++ hand_images hand )
```

The outermost tier, `players_image`, distributes all player cards along a full circle:

```
players_image :: Span [Player] -> Image GameSt
players_image r players
  = rotate (rad angle)
      (circular zero (2.0 * pi) (map (player_image r) players))
where
  angle = player_arc / (toReal (2 * no)) - player_arc / 2.0
  no    = 3 + no_of_cards_in_row (length players)
```

Without the additional rotation, the first player's cards are displayed as shown in Figure 1**(k)**. We prefer the layout of Figure 1**(l)** and therefor rotate the entire image by half the `player_arc`, decreased with half the angle required for one card.

Finally, the entire image overlays the three tiers (Figure 1**(l)** gives the result of a typical initial Ligretto game state for three players):

```
game_image :: GameSt -> Image m
game_image {players,middle}
  = overlay (repeat (AtMiddleX,AtMiddleY)) []
          ([ middle_image   (card_height *. 2  ) middle
           , names_image    (card_height *. 3.2) players
           , players_image  (card_height *. 4  ) players
           ]) host
where
  host = Just (empty (card_height *. 12) (card_height *. 12))
```

2.8 Discussion

When thinking of an image-under-construction, we map each individual layer to an image. What an image looks like, and how we would like to use it in layout, are two distinct concepts that we have separated by replacing bounding box with span box, and thinking of images as if they are infinitely large. When thinking of the layout, we first and foremost decide on the overall layout (e.g. collage or grid, relying on span boxes), and pinpoint the exact location (alignment and offsets) later. Finally, when design choices are in a sense arbitrary, we have adopted SVG's design choices.

3. Compositional Interactive Images

In this section, we describe how to turn static images into interactive ones by integrating them in iTasks. We start with a brief description of iTasks (Section 3.1). In iTasks, user-interaction is delegated to specialized tasks; the *editor tasks*. Hence, these are the tasks that need to be enriched with images (Section 3.2). Finally, we show how to turn the static *Ligretto* images interactive, and create a complete TOP specification of a game of *Ligretto* (Section 3.3).

3.1 iTasks essentials

The TOP paradigm, as embodied in iTasks, builds on a few core concepts: *tasks*, which define the work that needs to be done; *combinators*, to compose tasks from simpler ones; *editors*, which are tasks that facilitate user interaction; and *shared data sources (SDSs)*, to handle shared information in a uniform way.

Tasks are represented by the monad-like[2] type `(Task a)`, which has an associated *task value* of type `a`. By inspecting the current task value, other task (functions) can get informed about the state of the task (in progress or finished). Tasks can be composed sequentially, using the *step combinator* `(>>*)`, or in parallel, using the `parallel` combinator. Examples of their use are given when we continue with the case study in Section 3.3.

Editors are a means to view data or to interact with it. They are tasks that use type-driven generic programming to generate a user interface for any first-order type. Examples of editors are `viewInformation`, used to provide a read-only editor for a given type, and `updateInformation`, which allows the user to modify a value. The types of these editors are given here[3]:

```
:: ViewOption   a   = E.v: ViewWith   (a -> v)      & iTask v
:: UpdateOption a b = E.v: UpdateWith (a -> v)
                                      (a -> v -> b) & iTask v

viewInformation   :: Title [ViewOption   m]   m -> Task m | iTask m
updateInformation :: Title [UpdateOption m m] m -> Task m | iTask m
```

In both cases, the third parameter is the type of the *initial value* that is displayed or updated. Instead of providing an initial value, an editor can also be 'connected' to an SDS. In that case, the current value of the SDS serves as source for rendering, and any update coming from the editor is written to the SDS. In this way, one can define a set of parallel communicating tasks. For every above-mentioned editor, there is a share-enabled counterpart that automatically reacts to changes in the SDS they are connected with:

[2] We say monad-like, because the right-identity law does not hold for `Task`

[3] `E.`v`:` introduces an *existentially quantified type variable* v, while `& iTask` v places a type-class constraint for class `iTask` on v.

```
viewSharedInformation   :: Title [ViewOption r]
                           (ReadWriteShared r w)
                        -> Task r | iTask r
updateSharedInformation :: Title [UpdateOption r w]
                           (ReadWriteShared r w)
                        -> Task w | iTask r & iTask w
```

Figure 2 shows the result of applying these editors to a value of type `Card` (Section 2.7.1).

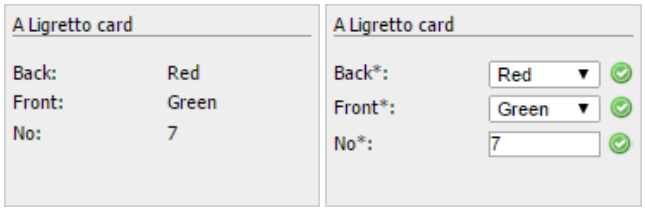

Figure 2. Generic `Card` `view-` and `updateInformation` tasks.

Clearly, neither resulting interface is the one that is required for the case study (Figure 1(**d**)). It should be noted that without special support from iTasks, the `View-` and `UpdateOption` types are of no help either: with these options the programmer can control the domain of the values that are viewed or updated but not the generic rendering. In the next section we show how to integrate the static images into these editors.

3.2 Enhancing editors with images

We first integrate static images with editors by introducing a new option for `view(Shared)Information` editors:

```
imageView :: (r -> *[*(ImageTag, *ImageTag)] -> Image r)
          -> ViewOption r | iTask r
```

With (`imageView` *render*), the rendering function *render* is used to visualize the model value of type r. Hence, with the same `Card` value that was used in Figure 2, the following editor:

```
viewInformation "A Ligretto card"
    [ViewWith (imageView (\card _ -> card_image Front card))]
    red_green_7_card_model
```

displays the card graphically, as in Figure 1(**d**).

Interactive images require more effort. First, we introduce a new option for `update(Shared)Information` editors:

```
imageUpdate :: (r -> v) (v -> *[*(ImageTag,*ImageTag)] -> Image v)
               (r -> v -> w)
            -> UpdateOption r w | iTask v
```

With (`imageUpdate` f *render* g), a source value of type r is transformed to a view model with function f, to which the *render* function is applied to create the image. Whenever the viewed value is changed by an interaction, a destination value of type w is constructed out of the original source value and changed view value with function g.

Second, we need to make the images themselves interactive. In Section 2.3 we have omitted one image attribute:

```
:: OnClickAttr m = { onclick :: m -> m }
```

If *img* has type (`Image m`) then (*img* `<@< {onclick = `f`}`) is the same image enhanced with *mouse hit-detection*. Whenever the user clicks on a part of *img*, then the function f is applied to the current model value that is associated with the image and computes a new model value, *updating* the model value. In turn, this triggers the functions on the `update(Shared)Information` editors to re-render the model value, if necessary. For example, when a change is made to a shared model value by applying some `onclick` function after an interaction, all tasks looking at this shared value will automatically be notified and updated such that they can show the new view corresponding with the new model value. Moreover, depending tasks can inspect this new task value, not knowing whether it originated from an interactive image or a generic interactive task. Compositionality is preserved because the `onclick` function is unaware of any final position, rotation, skewing, masking, or duplication of the image with which it is associated.

3.3 Case study continued: interactive Ligretto

In this section we continue with the Ligretto case study in two steps: we turn the static image of Section 2.7.2 into an interactive image (Section 3.3.1) and then proceed with the final iTask specification of the entire game (Section 3.3.2). In this section we assume the presence of the following pure functions:

```
play_row_card       :: Color Int GameSt -> GameSt
play_concealed_pile :: Color     GameSt -> GameSt
play_hand_card      :: Color     GameSt -> GameSt
```

(`play_row_card` *player no game*) moves the card of *player* found at row number *no* (counting from 1) to an available middle pile and, if such a middle pile exists, moves the top card of the player's ligretto pile to the row. (`play_concealed_pile` *player game*) moves the top three cards of the concealed pile to the discard pile of *player*, if these exist, and shuffles the discard pile back to the concealed pile, if not. Finally, (`play_hand_card` *player game*) moves the top card on the discard pile to an available middle pile, if such a pile exists. These functions are only concerned with the model types defined in Section 2.7.1. They ensure that only legal moves can be made.

3.3.1 Interactive Ligretto images

The `game_image` function defined at the very end of Section 2.7.2 shows the entire state of the game as seen from the perspective of the 'first' player. To show the game from the perspective of any player, we need to rotate the image according to that player's position in the list of participants. This is the purpose of the `player_perspective` function which is parameterized with the color of the player. This color parameter is also used to make certain that this player can only play her own cards.

```
player_perspective :: Color GameSt *[*(ImageTag, *ImageTag)]
                   -> Image GameSt
player_perspective color gameSt _
  = rotate (rad (~(toReal my_no * angle)))
           (game_image color gameSt)
where
  angle = 2.0 * pi / (toReal (length gameSt.players))
  my_no = hd [i \\ player <- gameSt.players
                 & i      <- [0 ..] | player.color === color ]
```

(Note that this function ignores the image tag source because they are not required by any of the image rendering functions.)

The new `game_image` function merely passes the player color to the outermost image tier that renders all playable and non-playable cards. The other two image tiers remain static.

```
game_image :: Color GameSt -> Image GameSt
game_image color {players, middle}
  = overlay (repeat (AtMiddleX,AtMiddleY)) []
            ([ middle_image  (card_height *. 2  )       middle
             , names_image   (card_height *. 3.2)       players
             , players_image (card_height *. 4  ) color players
             ]) host
where
  host = Just (empty (card_height *. 12) (card_height *. 12))
```

The only change to the `players_image` function is that for each player-rendering it is determined whether this rendering is going to be interactive or not.

```
players_image :: Span Color [Player] -> Image GameSt
players_image r color players
  = rotate (rad angle)
      (circular zero (2.0 * pi)
         [  player_image r (player.color === color) player
         \\ player <- players ])
where
  angle = player_arc / (toReal (2 * no)) - player_arc / 2.0
  no    = 3 + no_of_cards_in_row (length players)
```

Consequently, player_image has an additional Boolean parameter that tells whether the image is interactive. The interactive elements of a player are the row-cards and the hand-cards.

```
player_image :: Span Bool Player -> Image GameSt
player_image r interactive player
  = circular r player_arc
       (  row_images interactive player.row
       ++ [pile_image Front player.ligretto]
       ++ hand_images interactive player.hand player.color )
```

Playing a row card is defined by the pure function play_row_card. Only if the image is interactive is it added as an onclick attribute:

```
row_images :: Bool RowPlayer -> [Image GameSt]
row_images interactive row
  = [  tuneIf interactive (card_image Front row_card)
             {onclick = play_row_card row_card.back no}
    \\ row_card <- row & no <- [1 .. ] ]
```

Similarly, the two sub-piles of the hand cards behave as specified by the pure functions play_concealed_pile and play_hand_card, but only if the images are interactive:

```
hand_images :: Bool Hand Color -> [Image GameSt]
hand_images interactive {conceal,discard} color
  = [ tuneIf interactive (pile_image Back conceal)
            {onclick = play_concealed_pile color}
    , tuneIf interactive (pile_image Front discard)
            {onclick = play_hand_card color} ]
```

These extensions are sufficient to turn the static Ligretto rendering into an interactive image that can be used by editor tasks. It should be noted that the compositional style is not compromised by making these images interactive: none of these functions are aware of the ultimate position, angle or size in the fully rendered Ligretto game. The next section shows how to integrate these editor tasks into a complete distributed TOP application.

3.3.2 The Ligretto game

One of the Ligretto players takes the initiative and invites one through three friends to join in. Each player is assigned one of the Ligretto colors. In addition, we need to extract initial random values for the shuffling activities by all players. Once this is done, we can set up the shared game state and start to play:

```
play_Ligretto :: Task (Color, String)
play_Ligretto
  =            get currentUser
  >>= \me   -> invite_friends
  >>= \them -> let us = zip2 (colors (1 + length them))
                             [me : them]
                   num_us = length us
               in allTasks (repeatn num_us (get randomInt))
  >>= \rs   -> let gameSt = { middle  = repeatn (4 * num_us) []
                            , players = [  initial_player
                                           num_us
                                           c
                                           (toString u)
                                           (abs r)
                                        \\ (c, u) <- us & r <- rs]}
               in withShared gameSt (play_game us)
```

currentUser is an SDS that contains a User value describing which user is currently performing the task. randomInt is another SDS that holds random numbers. (withShared v t) creates an SDS with initial value v, and passes it to t. The invite_friends task terminates only with the correct number of friends.

```
invite_friends :: Task [User]
invite_friends
  =           enterSharedMultipleChoice
                 "Select␣friends␣to␣play␣with" [] users
  >>= \you -> if (not (isMember (length you) [1 .. 3]))
                 (    viewInformation "Oops" []
                        "number␣of␣friends␣must␣be␣1,␣2,␣or␣3"
                  >>| invite_friends)
                 (return you)
```

users is an SDS that contains all known users of the system. A selection of this list can be made with enter(Shared)MultipleChoice.

All players receive a new task to play a game of Ligretto:

```
play_game :: [(Color,User)] (Shared GameSt) -> Task (Color,String)
play_game users game_st
  = anyTask [  u @: play (c, toString u) game_st
            \\ (c, u) <- users ]
```

anyTask is a parallel task combinator that terminates as soon as one of its sub-tasks terminates. Here, each sub-task, play, is assigned to one of the players, using the task assignment combinator @:.

For each player, the game proceeds in two phases. In the first phase, cards are played until one of the participants obtains an empty ligretto pile. In the second phase, the winner receives her accolades[4].

```
play :: (Color,String) (Shared GameSt) -> Task (Color,String)
play (color,name) game_st
  =   updateSharedInformation name
        [imageUpdate id (player_perspective color) (const st)]
        game_st
  >>* [OnValue (game_over color game_st)]
where
   game_over me game_st (Value gameSt _)
    = case and_the_winner_is gameSt of
        Just {color, name}
          = let won = (color, name)
            in Just (accolades won me game_st >>| return won)
        _ = Nothing
```

The play task is an editor enhanced with the player perspective function that has been developed in Section 3.3.1. This task edits an SDS because all players manipulate the same middle cards and want to see the progress of their opponents at the same time. Players play simultaneously, but only their own cards are clickable and can be played in any order. The model functions presented in Section 3.3 guarantee that only legal moves can be made. Race conditions may occur, e.g. when two players want to play their card on top of the same middle pile. This is automatically solved by the shared system on a first-come-first-serve basis. The move of the second player is ignored. The step combinator >>* continuously checks the current value of the game state (that is manipulated by all players in parallel) to determine whether one of the players has obtained an empty ligretto pile, and if that is the case, proceeds with the accolades task. This terminates the entire play task (and therefor also the anyTask application in play_game).

Finally, to convince all other players that the winner has won fair and square, not only her name is announced, but also the entire game state. To disallow further editing of the game state, it is merely rendered as a view.

[4] This is a simplification of the rules of the game in which the remaining points need to be calculated. For brevity we omit this.

```
accolades :: (Color, String) Color (Shared GameSt) -> Task GameSt
accolades won me game_st
  = viewSharedInformation ("The␣winner␣is␣" <+++ won)
      [imageView (player_perspective me)] game_st
```

3.4 Discussion

Due to the expressive power of the iTasks editors and combinators, the definition of an interactive graphical oriented game such as Ligretto can be given in a concise declarative style. Static images can be turned into interactive ones by adding pure functions to (sub)images. No complicated mouse detections algorithms are needed to find out what has been clicked, it does not matter how the (sub)images are being transformed or used. It is clear that being compositional is a desirable property for an image library. However, it is commonly not so easy to realize this. The implementer needs strong support from the underlying graphical library.

4. Implementation

In this section, we explain how images are incorporated in iTasks' architecture (Section 4.1). We give an introduction to SVG and briefly evaluate its strengths and weaknesses (Section 4.2). Finally, we discuss how we generate SVG from images (Section 4.3).

4.1 Customizable interactive tasks

iTasks has a client-server architecture. Commonly, interactive tasks run as client in the browser while the coordination and communication between the tasks is handled by the server. Type driven generic functions are used with which form based editors can be generated for any first order type. As we have seen in Section 3.1, one can also specialize such an editor for a specific concrete type. One can even define rich client tasks, by using *editlets* [10], which can be thought of as an embedded client-side JavaScript application.

An editlet consists of two parts: one part of the editlet runs on the server (in native code) while the other part runs on the client (just-in-time compiled to JavaScript). Each part maintains its own state. A diff-based synchronization mechanism keeps the two states synchronized. Whenever the client receives a new diff, it has the ability to execute arbitrary JavaScript code. Editlet programmers do not write JavaScript code directly, but use a foreign function interface and a sophisticated cross-compilation mechanism from Clean to JavaScript [9]. This mechanism allows us to execute any Clean function in the browser. As a consequence, it is possible to write almost all code in one single language. We can decide at runtime which tasks and functions to execute on the server, and which to execute on the client.

In order to integrate interactive images in iTasks, we have created an SVG editlet which synchronizes an image's model value on the server with the client, after which the client renders the image and enables it to respond to on-click events.

4.2 SVG: Introduction, strengths, and weaknesses

SVG is a plain-text, XML-based markup language that describes vector graphics. It has been explicitly designed to work well with existing browser technologies, such as JavaScript, CSS, and the DOM. At the moment of writing this paper, SVG 1.1 Second Edition is the most recent published version of the specification. This version is largely supported by all modern mainstream browsers.

SVG has facilities for drawing both arbitrary shapes and text. For the former, it features one *primitive shape*: the *path*. A path is a sequence of individual path segments, which can either be straight or curved. All other shapes can be defined in terms of a path, although that would be cumbersome in practice. For that reason, SVG defines several *basic shapes*: rectangle, circle, ellipse, line, polygon, and polyline. Each of these basic shapes is represented by an SVG XML element. A shape's dimensions are specified with attributes on the shape element itself.

SVG also has facilities to render text, which is different from path-based shapes in that text is a sequence of font glyphs, specified in plain-text, rather than a sequence of paths. Font properties, such as the font family and font weight, are specified textually as SVG attributes on the text element. As a consequence of the way SVG implements text, one cannot determine the exact width of a piece of text until it is inserted into the browser's DOM and is rendered, even if all font properties have been specified. This is due to the fact that rendering text relies on the font definition being available on the client. If the client does not have the specified font, it chooses a fall-back font. The fall-back font may have different glyph-widths than the specified font, resulting in a different text-width. This makes images containing text harder to render with predictable results.

A collection of shapes can be grouped using the *group element* `<g />`. These shapes can then collectively be identified, transformed, interacted with, or attributed with certain properties.

All shapes can be styled by specifying properties on the individual elements. All shapes, except path, can be positioned relative to the current coordinate system by specifying `x` and `y` properties.

Shapes can be *transformed* using a *transformation matrix*. For convenience, however, SVG provides specific transformations: translation, scaling, rotation and skewing.

SVG is largely compositional by itself. Individual shapes can be drawn and positioned independently from others. However, this compositionality is lost when rotation transformations are applied; when rotating an image, its axes rotate along with it. Any subsequent transformations, such as translations, then act relative to these rotated axes. As a consequence, first rotating an image around its center and then translating it yields a different result than first translating the image and then rotating it around its center. Figure 3 shows the problem graphically.

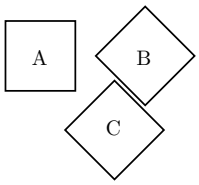

Figure 3. SVG rotation and translation in different orders

Square A is the original square. Square B is our desired result and is what we get after first translating square A along the x-axis and then rotating it 45 degrees around its center. However, when we first perform the rotation and then the translation, we end up with square C. We compensate for this behavior by wrapping an image in a group element immediately after it is rotated. Any subsequent transformations are then applied to the group, rather than the original shape. This effectively resets the image's axes, allowing us to obtain result B, regardless of the order in which the transformations have been applied.

Transformations also pose specific challenges for text, because rotation and translation are always performed relative to an image's *origin*. In all other SVG elements, the origin is situated in the element's top-left corner. For text elements, however, the origin is situated on the left of the text's baseline, as is illustrated in Figure 4.

As a consequence of the different origin, we need to compensate when translating or rotating a piece of text. To do so accurately, we require at least the font's ascent and descent heights. However, the current SVG specification does not provide an API to obtain these metrics. A common workaround to this problem is to count pixels of a text glyph on a raster-based canvas. We choose a simpler

Figure 4. A text's origin and ascent, descent, and x-heights.

approximation: we assume that the ascent and descent heights are 75% and 25% of the text height, respectively. While this heuristic has worked reasonably well in practice so far, it is far from a general solution.

4.3 Generating SVG

Since a text's width cannot be known until it is inserted into the DOM, we are forced to interact with the browser during SVG generation. Because of this, we choose to execute all parts of the rendering process on the client. We have created an SVG editlet which synchronizes the model value between the server and client, turns that model value into an image on the client, then calculates the text widths, and finally renders that image as SVG. This process is illustrated in Figure 5.

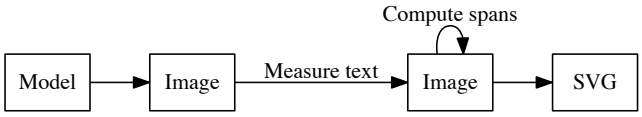

Figure 5. The SVG generation pipeline

Even with known text-widths, images can still contain lookup-spans which we need to resolve and reduce to pixel values, before we can generate SVG. Several iterations may be needed until we arrive at a fix-point and have resolved all lookup-spans. In the worst case, this process can diverge. When we have converged on a fix-point, SVG is generated and inserted in the DOM.

Generating SVG code is simplified by desugaring the internal image structure. All grids and overlays are desugared to collages, as shown in Figure 6. We then only have to concern ourselves with rendering SVG for collages. Figure 6 omits the implementation details of basic images, since they have a one-to-one correspondence to basic SVG shapes.

To translate overlays to collages, we first calculate the spans of all sub-images, after which we determine the spans for the largest image in the overlay, or the span of the host image, if present. We then calculate the offsets required to align all images relative to these spans, and add them to the offsets manually provided by the image programmer. These offsets are then used to express the overlay as collage. Translating a grid to a collage is a bit more involved. First, we obtain a list of lists of the spans of the individual images in the grid layout. Each list in the outer list represents one row, while each index in the inner lists represents one column. To calculate the offsets of each cell, we first obtain the x- and y-spans of each row and column. These spans are determined by the widest and highest cell in each row and column. Each cell's offset is calculated by adding the dimensions of previous cells together, keeping into account the alignment and manual offsets that each cell has. We end up with a list of lists of offsets, which we then flatten to obtain the list of offsets required to form a collage.

```
getXAlign _    _     AtLeft    = zero
getXAlign maxX xspan AtMiddleX = (maxX /. 2.0) - (xspan /. 2.0)
getXAlign maxX xspan AtRight   = maxX - xspan

getYAlign _    _     AtTop     = zero
getYAlign maxY yspan AtMiddleY = (maxY /. 2.0) - (yspan /. 2.0)
getYAlign maxY yspan AtBottom  = maxY - yspan

toSVG (BasicImage ..) = .. // Omitted for brevity
toSVG (Overlay aligns offsets images host) =
  let allSpans        = getAllSpans images
      (maxX, maxY)    = getMaxSpans allSpans host
      alignOffsets    = [  ( getXAlign maxX xspan align
                           , getYAlign maxY yspan align)
                        \\ (xspan, yspan) <- allSpans
                         & align          <- aligns ]
      positionOffsets = [  (alignX + offsetX, alignY + offsetY)
                        \\ (alignX, alignY)   <- alignOffsets
                         & (offsetX, offsetY) <- offsets ]
   in toSVG (Collage positionOffsets images host)
toSVG (Grid offsetss alignss imagess host) =
  let spanss  = getAllGridSpans imagess
      offsets = calculateGridOffsets (getColumnXSpans spanss)
                  (getRowYSpans spanss) alignss imagess offsetss
      calculateGridOffsets cellXSpans cellYSpans
        alignss imagess offsetss =
        fst (foldr (mkRows cellXSpans) ([], zero)
               (zip4 alignss imagess cellYSpans offsetss))
      mkRows cellXSpans (aligns, images, cellYSpan, offsets)
        (allOffsets, accYOff) =
        let cols = fst (foldr (mkCols cellYSpan accYOff) ([], zero)
                        (zip4 aligns images cellXSpans offsets))
         in ([cols : allOffsets], accYOff + cellYSpan)
      mkCols cellYSpan accYOff (align, image, cellXSpan,
        (manualXOff, manualYOff)) (allOffsets, accXOff) =
        let ( imageXSpan
            , imageYSpan) = getImageSpans image
            alignXOff     = getXAlign cellXSpan imageXSpan align
            alignYOff     = getYAlign cellYSpan imageYSpan align
            offsetPair    = ( alignXOff + accXOff + manualXOff
                            , alignYOff + accYOff + manualYOff)
         in ([offsetPair : allOffsets], accXOff + cellXSpan)
   in toSVG (Collage (flatten offsets) (flatten imagess) host)
toSVG (Collage offsets images (Just host)) =
  svgGroup []
    [ toSVG host
    , toSVG (Collage offsets images Nothing) ]
toSVG (Collage offsets images Nothing) =
  svgGroup []
    (zipWith (\off img -> svgGroup [translateAttr off] (toSVG img))
             offsets images)
```

Figure 6. Outline of the SVG conversion algorithm.

4.4 Discussion

Choosing SVG as rendering mechanism has the advantage that images are inherently scalable and are viewable in any modern browser. However, it also poses new problems.

The plain-text nature of SVG introduces problems with rendering fonts, because not all font metrics required for positioning text are available in the SVG API. Future SVG standards will likely address these problems. Additionally, we wish to add support for embedded fonts. Currently, we cannot guarantee a particular font is available on the client. With embedded fonts, we can. Both SVG 1.1 and CSS 3 support embedding fonts. An additional benefit is that we can always calculate the width of text snippets server-side if an embedded font is used, thereby eliminating the need to calculate text widths on the client.

Another problem is due to the fact that we are currently computing images completely on the client. This is significant slower than doing so on the server, because JavaScript is an interpreted, garbage-collected language, which has to work with limited heap space. We frequently trigger JavaScript's garbage collector while evaluating Clean expressions. This is due to the fact that the representation of our client-side runtime system heavily uses arrays, which it frequently creates and destroys, creating garbage on the JavaScript heap. In practice, these slowdowns make it infeasible to play a game of Ligretto on slower machines, because the computational lag can be as much as one full second. We reduce this problem by firstly reducing the size of the span-expressions as much as possible during their construction. This is not always possible, however, due to the presence of lookup-spans. Secondly, we make the client-side computations as strict as possible, eliminating unnecessary thunk evaluation. Still, these are only optimizations, rather than actual solutions. We want to pursue three solutions to this problem. Firstly, we want to generate all SVG on the server, so that we only need to send a string of SVG to the client. This requires first calculating all text widths on the client, requiring us to implement a rendering protocol. Currently, however, the editlet infrastructure does not allow for implementing protocols, so the infrastructure will need to be extended. Secondly, we want to completely eliminate the standard JavaScript garbage collector from the editlet runtime and replace it with our own. This approach is advocated by the asm.js [1] initiative, which is a highly optimizable subset of JavaScript. Pursuing this solution, we also want to generate low-level, asm.js-style JavaScript instead of the high-level, human-readable JavaScript we are currently generating. Thirdly, we want to do partial updates to the images, so that only the parts that have changed need to be recalculated and redrawn.

5. Related work

Peter Henderson's Functional Geometry (FG) is a seminal approach to purely compositional images [16]. Henderson states [17] that the design principle *"... was based on contemporary views of what was good practice in declarative systems"*. Similar to FG, we always specify the layout of sub-images relative to each other. Unlike FG, we do not abstract from 'size' (or rather, span boxes, in our terminology, because we regard images to be infinitely large). For instance, in FG, the span boxes of `beside`(p,p) and p are equal. In `Graphics.Scalable` (and most other approaches), the span box of (`beside [] [] [`p,p`] Nothing`) has twice the width of the span box of p. In FG, overlaying images consists of taking the union of graphic elements ([17] Section 5) which is a sensible choice because the primitive elements are (curved) lines only. Any approach that supports (partially) filled shapes must make the order of rendering of graphic elements explicit, either via ordering the graphics operations (typically on a canvas-model) or via a stacking concept. We have chosen the latter route and separate stacking images (z-axis) from specifying their relative layout (x- and y-axes). This idea can be traced back, although in a very different way, to Haggis [14, 15], in which *piles* of *widgets* (i.e. common user-interface elements, such as text fields) are created monadically and put in containers separately to control their layout along the x- and y-axes. At the risk of diverging, it should be mentioned here that this solution has been adopted in other GUI approaches, viz. Object IO [4], TkGofer [6], and wxHaskell [19]. More recently, the Diagrams approach by Brent Yorgey [24], very explicitly deals with stacking using lists and monoids as organizational principle of structuring the library. Diagrams features an elegant way of placing images besides each other using their outlines instead of bounding boxes. However, Diagrams is restricted to non-interactive images only, and the other approaches do not offer the usual graphical transformations such as rotation, scaling, and skewing on widget-like components. One of the advantages of using SVG as graphics back-end is that it extends to both graphics and widgets. Arbitrary HTML can be embedded in SVG document using the `<foreignObject>` element, after which it can be arbitrarily transformed like all other SVG elements.

The layout combinators of `Graphics.Scalable` were inspired by the Racket image API [2, 13], which has a mature, but rather baroque, API for the compositional specification of images. For instance, for the specification of layout, it features 22 functions. In contrast, `Graphics.Scalable` has 1 core layout function, `collage`, and 5 derived combinators (Section 2.4). These are sufficient to model all Racket image layout combinators, and more, as the Racket API does not support the grid-combinator. In addition, we profit from the orthogonality of the SVG back-end in that we can support flipping transformations, which is restricted to images without text in Racket. The Racket image API is bitmap-oriented and offers features such as manipulating bitmaps directly, extracting color-lists and bitmaps from images, 'freezing' images, and defining a pragmatic equality relation that is based on the current bitmap pixels. Except for the ability to embed bitmaps in SVG, the other features do not match naturally with the vector graphics philosophy. Both Racket and SVG offer elements that have not yet been transferred to `Graphics.Scalable` (both: Bézier curves; Racket: pinholes; SVG: paths, gradients, and filtering). We conjecture that they can be added to `Graphics.Scalable` without compromising its design principles.

An entirely different view on images is taken by Conal Elliot *et al* in their work on Pan [11], enhancing it with interaction, resulting in Fran [12] which gave birth to the paradigm of *functional reactive programming* (FRP) and, amongst others, Yampa [7, 18]. Characteristic to these approaches is to consider images as functions from coordinates to a well-defined range (Pan and Fran), animations as functions from continuous time to images, and interactive applications as functions from discrete events to animations (Yampa). A recurring theme in their work is that specifications are functions from a continuous domain to a discrete domain. The implementation 'samples' these functions. This differs greatly from our approach that advocates a 'structurally-analytic' view on image specifications and embedding in TOP to define behavior.

Another different path has been taken by Magnus Carlsson and Thomas Hallgren in their work on the Fudgets system [5]. Just like FRP and TOP, it features combinators to structure the top-level behavior of the interactive application. The basic elements are the fudgets which conceptually behave as typed value-transformers at their API-side, abstracting from the concrete way they work. This is also the key difference with iTasks and TOP that features task abstraction that processes a value. Images can be programmed in Fudgets using an approach that is similar to the Pictures abstraction that is used in the above mentioned Haggis system [15].

6. Conclusions and future work

We have presented an image library and have integrated it with iTasks to allow the creation of distributed, multi-user, web applications with custom-built interactive, graphical user interfaces. The image libary is implemented on top of SVG, produces interactive scalable vector graphics, and can be used in any modern browser. An important property of the image library is that it is purely compositional, both for static and interactive images.

The Ligretto case study demonstrates how graphically based multi-user tasks can be defined in a concise way, offering a good separation of concerns to the programmer. This involves three separate stages: first, one concentrates on modeling the game's domain, using pure data structures and pure functions; second, one defines the graphic visualization as functions from this domain to image values; third, one defines the application behavior as an iTask and integrates visualization within editors. We have observed this same

pattern of working in an earlier experiment [3] that, at that time, did not have the refined SVG support as `Graphics.Scalable`. We are going to investigate the generality of this application design pattern.

The current implementation suffers from severe performance issues of the generated client-side JavaScript code. We want to address this problem by generating asm.js-style code, replacing the garbage collection by our own, and moving calculations from client to server where possible. Early experiments that perform a round-trip to the client to measure text widths, but render the SVG on the server show promise of greatly improved performance.

Our event model is currently limited: interaction is restricted to the single model type of the entire image, and the event model is restricted to on-click events only. We want to investigate how to define and combine interactions on sub-images. We need additional ways of interacting with images such as *drag-and-drop*, *double-click*, and *right-click*, but also *keyboard input*. We want to explore more complex forms of interaction, such as touch gestures. The challenge in incorporating these interactions is that they must not compromise the way of working and thinking of the `Graphics.Scalable` library.

As mentioned in the introduction, we are using the library to draw Tonic diagrams. In these diagrams, individual nodes are connected with edges. Tonic's diagrams are simple enough that we can compute these edges in a straight-forward manner. However, this is not the case in general. Therefore, we want to introduce the concepts of connector points (which can be attached to an image), and include automatic edge routing between these connector points.

Acknowledgments

This research is partly funded by TNO's PhD fund and the Nederlandse Defensie Academie (NLDA). The authors thank the reviewers for their constructive feedback.

References

[1] asm.js, Aug. 2014. URL `http://asmjs.org/spec/latest/`.

[2] image.rkt, Dec 2014. URL `http://docs.racket-lang.org/teachpack/2htdpimage.html`.

[3] P. Achten. Why functional programming matters to me. In P. Achten and P. Koopman, editors, *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday, Festschrift*, number 8106 in LNAI, pages 79–96. Springer, August 2013. ISBN ISBN 978-3-642-40354-5.

[4] P. Achten and R. Plasmeijer. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[5] M. Carlsson and T. Hallgren. 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.

[6] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proceedings of the 2nd International Conference on Functional Programming, ICFP '97*, volume 32(8), pages 251–262, Amsterdam, The Netherlands, 9-11, June 1997. ACM Press.

[7] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Proceedings of the 5th Haskell Workshop, Haskell '01*, Sept. 2001.

[8] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable vector graphics (svg) 1.1 (second edition). Technical Report REC-SVG11-20110816, W3C Recommendation 16 August 2011, 2011.

[9] L. Domoszlai, E. Bruël, and J. Jansen. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae*, 3:76–98, 2011. URL `http://www.acta.sapientia.ro/acta-info/C3-1/info31-4.pdf`.

[10] L. Domoszlai, B. Lijnse, and R. Plasmeijer. Editlets: type based client side editors for iTasks. In S. Tobin-Hochstadt, editor, *Proceedings 26th International Workshop on the Implementation of Functional Languages, IFL '14, Boston, U.S.A.*, Oct 1–3 2014. Under submission.

[11] C. Elliot. Functional images. In J. Gibbons and O. de Moor, editors, *The fun of programming*, pages 131–150. Palgrave Macmillan, 2003. ISBN 0-333-99285-7.

[12] C. Elliot and P. Hudak. Functional Reactive Animation. In *Proceedings International Conference on Functional Programming, ICFP '97*, pages 263–273, Amsterdam, Netherlands, 1997.

[13] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System * or, Fun for Freshman Kids. In *Proceedings International Conference on Functional Programming, ICFP '09*, Edinburgh, Scotland, UK, 2009. ACM Press.

[14] S. Finne and S. Peyton Jones. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Graphics*, pages 85–101, Maastricht, the Netherlands, 1995. Springer.

[15] S. Finne and S. Peyton Jones. Pictures: A Simple Structured Graphics Model. In D. Turner, editor, *Proceedings of the 1995 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 10-12 1995. Electronic Workshops in Computing.

[16] P. Henderson. Functional geometry. In D. Friedman and D. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187, Pittsburgh, Pennsylvania, 1982. ACM Press. URL `http://www.ecs.soton.ac.uk/~ph/funcgeo.pdf`.

[17] P. Henderson. Functional geometry. *Higher-Order and Symbolic Computation*, 15:349–365, 2002.

[18] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In J. Jeuring and S. Peyton Jones, editors, *Proceedings of the 4th International Summer School on Advanced Functional Programming, AFP '03*, volume 2638 of *Lecture Notes in Computer Science* , pages 159–187. Oxford, UK, Springer-Verlag, 2003.

[19] D. Leijen. wxHaskell: a portable and concise GUI library for Haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, 2004. ACM. . URL `http://doi.acm.org/10.1145/1017472.1017483`.

[20] B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands , 2013. ISBN 978-90-820259-0-3.

[21] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

[22] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. 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*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM. ISBN 978-1-4503-1522-7.

[23] J. Stutterheim, R. Plasmeijer, and P. Achten. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, volume 8843 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014.

[24] B. Yorgey. Monoids: Theme and variations (functional pearl). In *Proceedings of The Haskell Symposium*, Copenhagen, Denmark, September 13 2012. ACM.