

Compositional Views in Compositional Images

– extended abstract –

– category: research –

Peter Achten¹[0000–0002–3585–7165] and Pieter Koopman¹

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{P.Achten,pieter}@cs.ru.nl

Abstract. The iTask SVG Image library uses a combinator approach to specify the layout of graphical elements. These graphics can be made interactive, by associating callback functions with images. The type system is used to make sure that the callback functions operate on view model values of the correct type. A long-standing open problem with the library is how to deal with interactive images that work on different view model values. This is a highly desirable feature because it allows code reuse and modularization. A solution should not ‘pollute’ the set of image combinators with view model plumbing combinators. It should also not alter the design philosophy that considers images as being layers of images on top of a host image. We propose a solution to this problem by recursively applying editor task customization to images.

Keywords: model-view, graphics, interaction

1 Introduction

The iTasks system [11,12,13] is a general purpose framework to create interactive web applications. Its principle building blocks are *tasks*, of parameterized type `Task m`. The type parameter `m` models the information that the task shares with its environment to represent its current state. *Editor tasks* are the chief component for programmers to create interactive tasks. Ordinarily, a (graphical) user interface for an editor task is automatically generated via generic programming techniques [8,7,4]. If a custom look and feel is needed, then this can be done with an *SVG editor task* [3]. It features a compositional image library, based on the W3C Scalable Vector Graphics standard [5], and uses a combinator approach, inspired by the seminal Functional Geometry approach by Henderson [6] and the comprehensive Racket image library [1]. In SVG editor tasks, callback functions do not operate directly on the task model value but instead on the rendered view. It uses a model-view customization [9]: the model, say of type `m`, corresponds with the task value, and the view, say of type `v`, corresponds with the view value at the client side. Callback functions are pure functions on the view model of type `v`. To enforce matching types of callback functions in the rendered

views, the image type is parameterized with the type of the view model: `Image v`. For the programmer, it suffices to define the connection between the task model value of type `m` and the client view value of type `v` and its rendering of type `Image v` with three pure functions, collected in a structure of type `SVGEEditor m v`¹:

```

:: SVGEEditor m v
= { initView    :: m -> v
  , renderImage :: m v *TagSource -> Image v
  , updModel    :: m v -> m
  }

```

In the current approach images are compositional, but this is not true for the single view model type `v` that is enforced on the entire image, of type `Image v`. This severely hampers code reuse and modularization.

We eliminate this restriction by making view models compositional. The key idea is to recursively allow a `SVGEEditor v w` in an `Image v` created by a `SVGEEditor m v`. In this way, the ‘current’ view of type `v` serves as model for another view of type `w` that is rendered independently, interacts independently, and in which new values get updated into `v` again with the pure `updModel` function.

To preserve type safety, callback functions should still be associated correctly with the view model of the associated type. This is obtained for free when recursively allowing `SVGEEditors`. To preserve the compositional nature of images, it should not matter from which view model they originate. To obtain compositional images, they need to be liberated from their view value types as soon as these are no longer required. This has become possible in a completely new implementation of SVG editor tasks [2].

To demonstrate the versatility of extending images with SVG editors, we redo the extensive case study done in [10], which served as a demonstrator for compositional editor tasks.

With the enhanced image library the application programmer can develop interactive image components, concerning themselves only with the appropriate model view data types, rendering, and behaviour. The `SVGEEditor` takes care of logically connecting the their model view and the model view of their context.

2 Compositional Images

The API of the compositional and interactive images is mostly the same as presented in [3]. Figure 1 shows a representative snapshot.

Conceptually, every image is infinitely wide and perfectly transparent. Instead of pixel, the unit of measure is `px`² of type `Span`. Instead of bounding boxes for layout, we use *span boxes*. Just as bounding boxes, span boxes have a width and height, but unlike bounding boxes, span boxes allow graphical content to appear anywhere in the image, relative to the span box³. Images are either basic (`empty`, `ellipse`, `rect`, `text`) or an `overlay` of images.

¹ `*TagSource` is explained in Section 2

² Browser SVG renderers usually map (`px 1.0`) to one pixel.

³ *x*-axes increase to the right, *y*-axes increase downwards.

```

:: Image v           // Image is an abstract data type
:: ImageTag          // identify a (sub) image (see tag function)
:: Span              // measure, unit is px
:: FontDef           // font family, height, and attributes
:: *TagSource        ::= *[TagRef]
:: *TagRef            ::= *(ImageTag, *ImageTag)
:: XYAlign            ::= (XAlign, YAlign)
:: XAlign             = AtLeft | AtMiddleX | AtRight
:: YAlign             = AtTop | AtMiddleY | AtBottom
:: Host              v = NoHost | Host (Image v)
:: SyncWithServer    = SyncWithServer | NoSyncWithServer
:: OnClickAttr       v = {onClick :: Int v -> (SyncWithServer,v)}
:: OnKeyDownAttr     v = {onkeydown :: Keyboard v -> (SyncWithServer,v)}

px                :: Real                                -> Span
normalFontDef    :: String Real                          -> FontDef
empty            :: Span Span                            -> Image v
ellipse          :: Span Span                            -> Image v
rect             :: Span Span                            -> Image v
text             :: FontDef String                       -> Image v
overlay          :: [XYAlign] [(Span, Span)] [Image v] (Host v) -> Image v
tag              :: *ImageTag (Image v)                 -> Image v
class (<@<) attr infixl 2 :: (Image v) (attr v)         -> Image v
instance (<@<) OnClickAttr, OnKeyDownAttr

```

Fig. 1: A representative snapshot of the image API

An `(overlay aligns offsets imgs h)` is a stack of images `imgs`⁴ on top of a host image (`h = Host img`) or no host image (`h = NoHost`). The span box of the overlay is the span box of `img`, if available, or the bounding box of the span boxes of `imgs`. The relative layout of `imgsi` is first determined by `alignsi`⁵ and then fine-tuned with `offsetsi`⁶.

Span expressions are inductively defined by concrete `px` values, span references, and span arithmetic (see Figure 2).

Span references refer to the dimension of images: `textxspan` refers to the width of a piece of text when rendered using the given font, and `imagexspan` and `imageyspan` refer to images that are tagged with tag values that are drawn from `*TagSource` which is an infinitely long list of abstract labels. The labels come in pairs: the unique value of type `*ImageTag` tags an image, and its non-unique counterpart can be used arbitrarily many times as a span reference. Uniqueness prevents conflicting uses of the same tag, but an image tag can still be dangling if its unique counterpart does not tag an image. For the application programmer `ImageTag`

⁴ `imgsi+1` is rendered on top of `imgsi`.

⁵ The default alignment is `(AtLeft, AtTop)`.

⁶ The default offset is `(px 0.0, px 0.0)`.

```

// concrete px values:
px      :: !Real      -> Span
instance zero Span
// span references:
textxspan :: !FontDef !String -> Span // text width using font
imagexspan :: !ImageTag      -> Span // image width
imageyspan :: !ImageTag      -> Span // image height
// span arithmetic:
instance + Span          // add
instance - Span          // subtract
instance abs Span        // absolute
instance ~ Span          // negate
(*.)  :: !Span !n -> Span | toReal n // multiply by scalar
(/.)  :: !Span !n -> Span | toReal n // divide by scalar
minSpan :: ![Span] -> Span          // minimum of spans
maxSpan :: ![Span] -> Span          // maximum of spans

```

Fig. 2: Span expressions

values are fully opaque: the only way to get such a value is via the `renderImage` function of an `SVGEditor`.

Span arithmetic is addition (+), subtraction (-), taking the absolute value (`abs`), negating the sign (~), and multiplication (`.*`) and division (`/.`) by a scalar value. Spans can not be compared, but we can obtain the minimum (`minSpan`) or maximum (`maxSpan`) span of a list of span expressions.

3 Compositional Images with Compositional Views

In the `iTask` framework, SVG editor task customization occurs at the task level. Suppose that we have some model value and type `m :: M`, then the editor task with generically generated user interface is created by `updateInformation [] m` which has type `Task M`. Suppose that we have created a custom look and feel view `view :: SVGEditor M V`, for some view model type `V`, then this task is customized by `updateInformation [toUpdateOption view] m`. This customized task still has type `Task M`, but it is rendered as an image of type `Image V`. Task customization remains untouched in the proposed extension.

We extend the Image API of Figure 1 with just one new function:

```
svg :: (SVGEditor v w) -> Image v
```

Suppose we have a model value and type `m :: M`, and an image of type `Image M`, then `svg view` can be used as an image of the same type, but its initial view value `v` is defined by `view.initView m`, its user interface by `view.renderImage m v`. Whenever its view value is changed into a new value, `v'` say, this updates the current model value, as defined by `view.updModel m v'`.

The implementation is changed at several places. This is described in the remainder of this section.

Every `SVGEEditor` `renderImage` function immediately returns a deep embedding of itself (Figure 3). The deep embedding gets extended with an existentially

```

:: Image v
= Empty'   Span   Span
| Ellipse' Span   Span
| Rect'    Span   Span
| Text'    FontDef String
| Tag'     ImageTag           (Image v)
| Overlay' [XYAlign] [(Span, Span)] [Image v] (Host v)
| Attr'    (ImageAttr' v)      (Image v)
// new:
| E.w : SVGEEditor' (SVGEEditor v w)
:: ImageAttr' v
= HandlerAttr'           (ImgEventHandler v)
:: ImgEventHandler v
= ImgEventHandlerOnClickAttr (OnClickAttr   v)
| ImgEventHandlerOnKeydownAttr (OnKeyDownAttr v)
:: ImgNodePath := [ViaImg]
:: ViaImg      = ViaChild Int // ViaChild i: visit child image with index @i
                | ViaHost      // ViaHost: visit host image
                | ViaAttr       // ViaAttr: visit attribute image
// new:
                | ViaSVG        // ViaSVG: visit SVGEEditor

```

Fig. 3: Deep embedding of image API

quantified (`E.w`;) data constructor `SVGEEditor'`. The scope of `w` is only known inside the data constructor.

The key purpose of the deep image embedding is to identify callback functions via the `ImgNodePath` from the root of the image embedding to the attribute data constructor that contains the actual function. We extend the path directive with `ViaSVG` which identifies the use of an `SVGEEditor`.

Without integrated `SVGEEditors`, obtaining the callback function from the deep image embedding is a matter of ‘following’ the path directives in the `ImgNodePath`:

```
getImgEventHandler :: (Image' v) ImgNodePath -> ?(ImgEventHandler v)
```

With integrated `SVGEEditors` this straightforward approach is not possible. Instead, while navigating the deep image embedding via the `ImgNodePath` we keep track of the current view model value, and switch to the next one whenever an `SVGEEditor` is encountered. Furthermore, we need to pass around any arguments that have been generated at the client side callback function (encoded below as `ImgEventHandlerArgs`). This leads to a function of the following signature:

```

:: ImgEventHandlerArgs
= ImgEventHandlerOnNClickArg Int
| ImgEventHandlerOnKeyDownArg Keyboard

```

```
appImgEventHandler :: ImgEventHandlerArgs (Image v) v ImgNodePath -> ?(SyncWithServer,v)
```

The relevant part of the new function concerns handling the found callback function and handling an `SVGEEditor`. When the empty `ImgNodePath` arrives at the callback function, it is applied to the arguments and the view model value.

```

appImgEventHandler args (Attr' (HandlerAttr' f) _) v []
= appEventHandler args f v
where
  appEventHandler :: ImgEventHandlerArgs (ImgEventHandler v) v -> ?(SyncWithServer,v)
  appEventHandler (ImgEventHandlerOnNClickArg n)
    (ImgEventHandlerOnClickAttr {onNclick=f}) v = ?Just (f n v)
  appEventHandler (ImgEventHandlerOnKeyDownArg k)
    (ImgEventHandlerOnKeyDownAttr {onkeydown=f}) v = ?Just (f k v)
  appEventHandler _ _ _ = ?None

```

In case of an `SVGEEditor` view, the corresponding view model value `w` is computed using `view.initView`, and the corresponding image is rendered using `view.renderImage`. The function `freshImageTags` generates a new stream of `ImageTag` value pairs. If a new view model value `w'` is returned, then it is used to update the context model view using `view.updModel`.

```

appImgEventHandler args (SVGEEditor' view) v [ViaSVG : p]
= case appImgEventHandler args img w p of
  ?Just (sync,w') = ?Just (sync, view.updModel v w')
  _ = ?None
where
  w = view.initView v
  img = view.renderImage v w freshImageTags

```

4 Case study

To appear in final paper.

5 Related work

To appear in final paper.

6 Conclusions

We propose a solution to a long-standing open problem with the `iTask SVG` Image library: how to combine interactive images that rely on different view model values without introducing view model plumbing combinators and without moving away from the design philosophy of the image library. The solution is based

on incorporating the task customization of SVG editors into the compositional image library. The application programmer can now develop interactive images that operate on arbitrary view model (types), and use them in any image by defining how the local view model depends on the context and how changes to that view model affect the context. This greatly increases code reuse and modularization.

References

1. image.rkt (Nov 2023), <https://docs.racket-lang.org/teachpack/2htdpimage.html>
2. Achten, P.: Mix and match deep and shallow embeddings for interactive web based SVG content. In: The 35th Symposium on Implementation and Application of Functional Languages (2023), under submission
3. Achten, P., Stutterheim, J., Domoszlai, L., Plasmeijer, R.: Task oriented programming with purely compositional interactive scalable vector graphics. In: Tobin-Hochstadt, S. (ed.) Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages. pp. 7:1–7:13. IFL ’14, ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2746325.2746329>, <http://doi.acm.org/10.1145/2746325.2746329>
4. Alimarine, A., Plasmeijer, R.: A generic programming extension for Clean. In: Arts, T., Mohnen, M. (eds.) Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL ’01, Stockholm, Sweden. LNCS, vol. 2312, pp. 168–186. Springer-Verlag (2002)
5. Dahlström, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J.: Scalable vector graphics (svg) 1.1 (second edition). Tech. Rep. REC-SVG11-20110816, W3C Recommendation 16 August 2011 (2011)
6. Henderson, P.: Functional geometry. In: Friedman, D., Wise, D. (eds.) Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming. pp. 179–187. ACM Press, Pittsburgh, Pennsylvania (1982), <http://www.ecs.soton.ac.uk/~ph/funcgeo.pdf>
7. Hinze, R.: A new approach to generic functional programming. In: Reps, T. (ed.) Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL ’00, Boston, MA, USA. pp. 119–132. ACM Press (2000)
8. Jansson, P., Jeuring, J.: PolyP — a polytypic programming language extension. In: Conference Record of POPL ’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 470–482. ACM Press (1997)
9. Krasner, G., Pope, S.: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* **1**(3), 26–49 (Aug 1988)
10. Lijnse, B., Plasmeijer, R.: Typed directional composable editors in itasks. In: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages. p. 115–126. IFL ’20, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3462172.3462197>, <https://doi.org/10.1145/3462172.3462197>
11. Plasmeijer, M.J., Achten, P.M., Koopman, P.W.M.: iTasks: executable specifications of interactive work flow systems for the web. In: Proceedings of the 12th international conference on functional programming, ICFP’07. pp. 141–152. ACM Press, Freiburg, Germany (October 1-3, 2007)

12. 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. pp. 195–206. ACM, Leuven, Belgium (September 2012)
13. Steenvoorden, T.J.: TopHat. Task-Oriented Programming with Style. Ph.D. thesis (2022), <https://repository.ubn.ru.nl/handle/2066/253701>