Mix and Match Deep and Shallow Embeddings for Interactive Web Based SVG Content

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

ABSTRACT

Interactive web applications need to handle the communication between server and client web browsers. In the iTasks system, this is delegated to a general purpose component, called *editor*. Editors allow fine grained control over the point of execution on the server, using native code, which is efficient, and on the client, using JavaScript, which is much less efficient. One particular use case of editors uses the W3C Scalable Vector Graphics standard to allow applications to fully customize the look and feel of an editor, called *SVG editor*. We show how we have used a mix of deep and shallow embeddings to make better use of server-side computations and reduce the amount of client-server communication.

ACM Reference Format:
Peter Achten. 2023. Mix and Match Deep and Shallow Embeddings for Interactive Web Based SVG Content. In *Proceedings of The 35th Symposium on Implementation and Application of Functional Languages (IFL'23).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

INTRODUCTION

Interactive web applications need to handle the communication between server and client web browsers. The iTasks system [6, 7] is a general purpose framework to create interactive web applications. It is a Domain Specific Language that is shallowly embedded in the host language Clean, a pure and lazy functional programming language. An iTask application consists of a web server application component and any number of web client browser applications. The server application runs in native code, and on each web client application, a Javascript image is running. In iTasks, *editors* are the chief component for programmers to create interactive tasks. Under the hood, generic programming techniques [2, 4, 5] are used to derive code automatically from the task value type of the editor task. If a web application requires a custom look and feel, then this can be done via a special kind of editor, such as the *SVG editor* for creating scalable images [1]. An SVG editor is an editor task with a model-view customization: the model, say of type `m`, corresponds with the task value (type), and the view, say of type `v`, corresponds with a value as well. Both values are used to define an interactive image of type `Image v`. The interactive image implementation is based on the W3C Scalable Vector Graphics standard [3].

The implementation presented in [1] is operational, but there was considerable room for improvement.

- At the time of developing SVG editors, the decision was made to perform the image computation entirely at the client side because font and text metrics can only be determined at the client side. This results in performance loss because the JavaScript engine at the client side is much slower compared to code executed natively at the server side. It is also overly pessimistic: images without text can be computed completely at the server side. Instead, it is better to isolate the part concerned with determining font and text metrics (which still requires a server-client round trip) from the image computation code. Because the programmer can choose to set the event handler(s) of an interactive image to local evaluation, this is a computation that must be available both on the server and client side.
- The interactive image depends solely on the current value of the model and view. This is also true for the event handlers, as they are associated with (sub) images via image attributes. This suggests to break the image computation into two parts: an initial, cheap, stage that only generates a deep embedding of the image function, and a second, less cheap, step that computes the layout of the image, using a shallow embedding. In this way, the deep embedding serves as an administration of the event handlers. Moreover, it simplifies the second step of the image computation.

In this paper we show how we have improved the code structure of SVG editors, and at the same time used the new structure to our advantage in obtaining a more efficient implementation. The remainder of this paper is organized as follows. Section 1 introduces the parts of the iTask system that are relevant to this paper. Section 2 shows a deep embedding for span expressions and how to decrease their size where possible. Section 3 shows how deep and shallow embeddings are used to compute interactive SVG images. Section 4 shows the server-client protocol that is required to allow the computation of an image on the server or client side. Section 5 discusses related work, and Section 6 presents the conclusions.

1 PRELIMINARIES

The API of the compositional and interactive images is mostly the same as presented in [1]. A representative, slightly simplified, snapshot is given in Figure 1. *Further explanation in final paper.*

Editor tasks are the chief component for programmers to create interactive tasks. The most general editor task is `updateInformation`

IFL'23, August 29–31, Braga, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
:: Image m        // Image is an abstract data type
:: ImageTag       // identify a (sub) image (see tag function)
:: Span           // defines the size of an Image element
:: FontDef        // defines the font family and attributes
:: Host        m = NoHost | Host (Image m)
:: OnClickAttr m = { onclick :: Int -> m -> m, local :: Bool }

px            ::         Real                      -> Span
normalFontDef :: String  Real                      -> FontDef
empty         :: Span    Span                      -> Image m
ellipse       :: Span    Span                      -> Image m
rect          :: Span    Span                      -> Image m
text          :: FontDef String                    -> Image m
tag           :: *ImageTag      (Image m)          -> Image m
collage       :: [(Span, Span)] [Image m] (Host m) -> Image m

class    (<@<) attr infixl 2 :: (Image m) (attr m) -> Image m
instance (<@<) OnClickAttr
```

Figure 1: A simplified snapshot of the image API

(omitting one non essential parameter, and simplified `UpdateOption` type):

```
updateInformation :: [UpdateOption m] m -> Task m | iTask m
:: UpdateOption m
   = E.v: UpdateUsing (m -> v) (m v -> m) (Editor v) & iTask v
```

With no option, an interactive task is generated generically, using the generic functions made available in the `iTask` class for the task value of type `m`. If an option is given, then instead of the generic interface the functions defined in the `Editor` kick in. The specific editors that concern us in this paper are SVG editors:

```
:: SVGEditor m v = { initView    :: m -> v
                   , renderImage :: m v *TagSource -> Image v
                   , updModel    :: m v -> m
                   }
fromSVGEditor :: (SVGEditor m v) -> Editor m | iTask m
```

The `initView` function relates the task model value with a value representing the view of the customized SVG editor. The `renderImage` function defines the custom, interactive, image that must be shown to the end user, using both the task model and view values. Changes to the task model value or the view value lead to a possibly new task model value, defined by `updModel`, and, as a consequence, also a new view value (via `initView`) and updated image (via `renderImage`).

The `*TagSource` is a list of abstract labels to identify image parts (using `tag`).

```
:: *TagSource :== *[TagRef]
:: *TagRef    :== *(ImageTag, *ImageTag)
```

To tag an image, a unique `ImageTag` is required. Images can be refered to arbitrarily many times, using the non-unique `ImageTag` counterpart value.

Figure 2 gives an example of an interactive image that contains text, and in which a user can increment a value by clicking on any one of the separately rendered digits of that value.

```
click :: Task Int
click = updateInformation [UpdateUsing id (\_ v = v) fromSVGEditor
                           { initView    = id
                           , renderImage = \_ = count
                           , updModel    = \_ v = v
                           }] 0

count :: Int *TagSource -> Image Int
count n _ = beside [] [] ?None [] (map digit (digits n)) NoHost
          <@< { onNclick = \mouse_clicks v = mouse_clicks + v
              , local    = False
              }

digits :: Int -> [Int]
digits n = [toInt c - toInt '0' \\ c <-: toString n]

digit :: Int -> Image Int
digit n = overlay [(AtMiddleX,AtMiddleY)] []
            [text font (toString n) <@< {fill = white}]
            (Host (rect (textxspan font (toString n) + px m)
                        (px (h+m))))
where
    font = normalFontDef "Times_New_Roman" h
    h    = 100.0
    m    = 6.0
```

Figure 2: Running example of a counter

2 DEEP EMBEDDINGS FOR SPAN EXPRESSIONS

In this section we show how span expressions are dealt with. `Span` is an abstract data type with the following operations available:

```
:: Span
instance zero Span     // zero
instance +    Span     // add
instance -    Span     // subtract
instance abs  Span     // absolute
instance ~    Span     // negate
instance *    Span     // multiply
instance /    Span     // divide

px          :: Real           -> Span // span in SVG pixels
textxspan   :: FontDef String -> Span // look up text width
imagexspan  :: ImageTag       -> Span // look up image width
imageyspan  :: ImageTag       -> Span // look up image height
columnspan  :: ImageTag Int   -> Span // look up column width
rowspan     :: ImageTag Int   -> Span // look up row height
minSpan     :: [Span]         -> Span // determine minimum span
maxSpan     :: [Span]         -> Span // determine maximum span
```

Span expressions are represented with a deep embedding:

```
:: Span
  = PxSpan      Real
  | LookupSpan  LookupSpan
  | AddSpan     Span Span
  | SubSpan     Span Span
  | MulSpan     Span Span
  | DivSpan     Span Span
  | AbsSpan     Span
```

```
  | MinSpan     [Span]
  | MaxSpan     [Span]
:: LookupSpan
  = ColumnXSpan ImageTag Int
  | RowYSpan    ImageTag Int
  | ImageXSpan  ImageTag
  | ImageYSpan  ImageTag
  | TextXSpan   FontDef String
```

The arithmetical span creation functions reduce the size of the arithmetical span expressions by extracting and combining the PxSpan components whenever possible. If a span expression can not be reduced completely to a PxSpan version, the goal is to create an arithmetical span expression representation with the left-hand side holding the PxSpan part that can be extracted and computed, and the right-hand side the remaining span expression.

3 DEEP AND SHALLOW EMBEDDINGS FOR COMPOSITIONAL IMAGES

In this section the new structure of the SVG editor implementation is presented.

In Figure 1 the API of the opaque Image functions was shown. Image is a deep embedding, shown in Figure 3.

```
:: Image m
  = Empty`       Span    Span
  | Ellipse`     Span    Span
  | Rect`        Span    Span
  | Text`        FontDef String
  | Tag`         ImageTag        (Image m)
  | Collage`     [(Span, Span)]  [Image m] (Host` m)
  | Attr`        (ImageAttr` m)  (Image m)
:: ImageAttr`      m = HandlerAttr` (ImgEventhandler m)
:: ImgEventhandler m = ImgEventhandlerOnClickAttr (OnClickAttr m)
:: Host`           m = NoHost` | Host` (Image m)
```

Figure 3: Deep embedding of image API

The API functions immediately return the corresponding data constructor, making it a cheap operation to obtain this deep representation.

For navigation purposes, the following data type and navigation function are defined:

```
:: ViaImg = ViaChild Int // ViaChild i: visit child image with index i
          | ViaHost      // ViaHost: visit host image
          | ViaAttr      // ViaAttr: visit attribute image

getImgEventhandler :: (Image m) [ViaImg] -> ?(ImgEventhandler m)
```

The programmer uses a SVGEditor to customize an editor. Internally, an SVGEditor is transformed into an editor via a so called leaf editor.

```
:: LeafEditor edit st m =
    { genUI     :: UIAttributes DataPath (EditMode m)
                               *VSt -> ((UI,st),*VSt)
    , onEdit    :: DataPath (DataPath, edit) st
                               *VSt -> ((UIChange,st),*VSt)
    , onRefresh :: DataPath m st *VSt -> ((UIChange,st),*VSt)
    , valueFromState :: st -> ?m
    }

leafEditorToEditor :: (LeafEditor edit st m) -> Editor m
    | JSDecode{|*|} edit & JSONEncode{|*|}, JSONDecode{|*|} st
```

The edit type parameter of a LeafEditor captures the messages from (the JavaScript) client to server, and the st type parameter is the server side state that needs to be (de-)serialized (using JSON).

SVGEditors use LeafEditors to create an Editor.

```
fromSVGEditor :: (SVGEditor m v) -> Editor m | iTask m
fromSVGEditor svg    = leafEditorToEditor
    { LeafEditor
    | genUI          = withClientSideInit (initClientSideUI svg)
                                          initServerSideUI
    , onEdit         = serverHandleEditFromClient  svg
    , onRefresh      = serverHandleEditFromContext svg
    , valueFromState = valueFromState
    }
```

The genUI function takes care that the server side gets initialized (initServerSideUI) as well as the client side (initClientSideUI svg). The onEdit function handles every message that is sent by the client side to the server. The onRefresh function handles every change of the task model value by the context of the task. Finally, valueFromState attempts to retrieve the current task value from the current state.

SVG editors use the following data type to communicate information from the client to the server (so this is going to be the edit type parameter of LeafEditor):

```
:: ClientToServerMsg    m
    = ClientNeedsSVG        // client is ready to receive SVG
    | ClientHasNewModel m   // client has calculated a new task model value
    | ClientHasNewTextMetrics (Map FontDef Real)
                              (Map FontDef (Map String Real))
                            // client has retrieved font and text metrics
```

SVG editors use the following data type for the state that is stored at the server side (so this is going to be the st type parameter of LeafEditor):

```
:: ServerSVGState m
    = { model :: m                                // current model value
      , fonts :: Map FontDef Real                 // cached font metrics
      , texts :: Map FontDef (Map String Real)    // cached text metrics
      }
```

Hence, at the server side the current task model value is stored, as well as the cached fonts and texts metrics.

Information from server to client is passed via an SVG attribute change for which the client has registered a handler. The type of these messages is:

```
:: ServerToClientAttr m
 = ServerNeedsTextMetrics (Set FontDef) (Map FontDef (Set String))
 | ServerHasSVG           String ImgEventhandlers` ImgTags (?m)
```

In the final paper the implementations of all functions of the LeafEditor *and remaining types are explained.*

Both at the server side and the client side font and text metrics are stored. At the server side this is done via the leaf editor

state management (ServerSVGState), and at the client side local web storage is used.

The next step of the improved image computation function turns the deep image representation back into a computation:

```
toImg :: (Image m) [ViaImg]
      -> (Map FontDef Real) (Map FontDef (Map String Real))
         ImgTables -> (Img,ImgTables)
toImg (Empty`    w h)         p = empty`   w h
toImg (Ellipse`  w h)         p = ellipse` w h
toImg (Rect`     w h)         p = rect`    w h
toImg (Text`     fontDef txt) p = text`    fontDef txt
toImg (Tag`        t img)     p = tag`     t img              p
toImg (Collage` offs imgs h)  p = overlay` aligns offs imgs h p
toImg (Attr`    attr img)     p = attr`    attr img           p
```

The second argument keeps track of the current location in the deep image representation. The third and fourth arguments are the cached font and text metrics. The computation is a state transformer on ImgTables, that collect relevant information to generate SVG code from. The relevant part of this state is:

```
:: ImgTables
  = {imgUniqIds       :: ImgTagNo
    ,imgTags          :: Map ImageTag ImgTagNo
    ,imgNewFonts      :: Set FontDef
    ,imgNewTexts      :: Map FontDef  (Set String)
    ,imgSpans         :: Map ImgTagNo (Span,Span)
    ,imgEventhandlers :: Map ImgTagNo [([ViaImg],ImgEventhandler`)]
    }
```

Every (sub) image receives a fresh internal number of type ImgTagNo, which is kept track of by imgUniqIds. Because the image definition can also refer to (sub) images via ImageTag values, an additional table, imgTags, is stored that maps these ImageTag values to their corresponding ImgTagNo. References to font or texts of unknown dimensions are collected in imgNewFonts and imgNewTexts. Of every image, the currently known dimension is stored in imgSpans. Note that these span expressions may contain unresolved dimensions (*The full paper explains this in detail.*). Finally, of every image event handler a defunctionalized version is collected in imgEventhandlers together with its location inside the deep representation of the image:

```
:: ImgEventhandler`
  = { handler :: DefuncImgEventhandler`, local :: Bool }
:: DefuncImgEventhandler` = ImgEventhandlerOnClickAttr`
```

Recall that the location is sufficient to quickly find the concrete image event handler in the deep image representation. An additional benefit is that at this stage the data structures are no longer parameterized with the model type of the view that is rendered. This makes it possible to compose images that may originate from SVG editors of different view model types.

The Img result is a deep representation of the image that is more suited for generating the final SVG code.

```
:: Img
  = { uniqId    :: ImgTagNo
    , host      :: HostImg
    , transform :: ? ImgTransform
    , overlays  :: [Img]
    , offsets   :: [(Span,Span)]
    }
```

As stated above, every (sub) image is uniquely identified via its uniqId. In this representation, every image has a host image which defines its dimensions. The host image can be a leaf element (such as an ellipse, rectangle, text) or is a composition of images (such as a collage). In either case, on 'top of' this host image, other images (overlays) can be placed at given offsets relative to the left-top corner of the host image. Note that, as before, these span expressions may contain unresolved dimensions.

Due to the presence of custom references to (sub) images, and the presence of unresolved spans, one final step is needed to compute the final layout of the image. This is done in the final pass of the algorithm.

```
resolve_all_spans
  :: (Map ImageTag ImgTagNo)          // image tag identification
     (Map FontDef Real)               // complete font metrics
     (Map FontDef (Map String Real))  // complete text metrics
     Img                              // possibly unresolved image
     (Map ImgTagNo (Span,Span))       // all (sub) image dimensions
  -> MaybeError String (Img,ImgSpans)
```

This may fail because of cyclic custom references. Otherwise, the result Img is a full specification of an SVG image that is relatively straightforward to turn into a concrete SVG implementation.

4 SERVER-CLIENT PROTOCOL

In this section we describe the new server-client protocol. We distinguish the following cases.

Initialization. At the server side the task model value and font and text metrics caches are stored (ServerSVGState). The server sends a serialized task model value and compiled JavaScript code to the client. The client registers an initializer function and a callback function that responds to server-generated attribute changes. The initializer function stores the task model value and view value (computed via the initView function that is contained in the compiled SVG editor. Finally, the initializer function tells the server that it is ready to receive the SVG image (ClientNeedsSVG).

Server computes image. The server uses the toImg function to compute the deep Img representation, together with the ImgTables, and the cached font and text metrics. If all font and text metrics are available, the final SVG image can be generated and sent to the client, who only needs to register the defunctionalized event handlers. If some font or text metrics are missing, then these can be found in the ImgTables. The server sends a request to the client to obtain their metrics. The client first tries to retrieve them in its cache, and otherwise measures the requested font or text metric, sending all metrics back to the server. The server now has all information: it updates its caches and reruns the toImg function, which will result in a complete SVG image.

Server responds to task model value change of context. The task model value can be changed by the context of the task (the cause is irrelevant to this paper). In that case the same steps occur as above, except that in addition to sending the

complete SVG image to the client (potentially including one round-trip to obtain new font and text metrics) also the new task model value is sent over.

Client changes the view model via a registered event handler. The client first retrieves the proper function via the associated path and deep image representation. This results in a new view model value and new task model value, which are both stored. The client sends the new task model value to the server. The server now acts as in the previous two steps and computes a new image at the server side (possibly with a round-trip to obtain new font and text metrics) and sends the SVG image to the client, this time without sending the model value, as it is already available on the client.

5 RELATED WORK

To appear in final paper.

6 CONCLUSIONS

In this paper we have shown the new code structure of iTask SVG editors. The new implementation improves on the old implementation in creating a server-client protocol that supports the evaluation of interactive images at either the server or client side. One crucial design decision is to isolate the logic of obtaining font and text metrics from the original algorithm, enabling to create one image creation function that can be used both on the server and the client side. This leads to a major simplification of the code base. Another crucial design decision is the delay of the actual computation of the layout of all sub images, using symbolic references within the span expressions. Once all font and text metrics are available, it is a matter of resolving these missing values to compute the final layout in one pass through the shallowly embedded SVG image.

ACKNOWLEDGEMENTS

The author thanks Bas Lijnse who designed and implemented the new iTask editor infrastructure for his advice on their usage. Steffen Michels has built the leaf editor that manages the server side state storage.

REFERENCES

[1] 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

[2] 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)

[3] 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)

[4] 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)

[5] 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)

[6] 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 (Oct 1-3, 2007)

[7] 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 (Sep 2012)